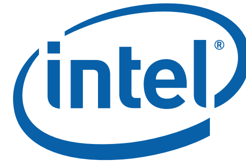


G45: Volume 1b: Graphics Core

Intel® 965G Express Chipset Family, Intel® G35 Express Chipset Graphics Controller

Programmer's Reference Manual (PRM)

January 2009
Revision 1.0a
Document Number: 321392-001

Technical queries: ilg@linux.intel.com

www.intellinuxgraphics.org

You are free:

to Share — to copy, distribute,display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

No Derivative Works. You may not alter, transform, or build upon this work.

You are not obligated to provide Intel with comments or suggestions regarding this document. However, should you provide Intel with comments or suggestions for the modification, correction, improvement, or enhancement of: 9a) this document; or (b) Intel products, which may embody this document, you grant to Intel a non-exclusive, irrevocable, worldwide, royalty-free license, with the right to sublicense Intel's licensees and customers, under Recipient intellectual property rights, to use and disclose such comments and suggestions in any manner Intel chooses and to display, perform, copy, make, have made, use, sell, and otherwise dispose of Intel's and its sublicensee's products embodying such comments and suggestions in any manner and via any media Intel chooses, without reference to the source.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Intel® 965 Express Chipset Family and Intel® G35 Express Chipset may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

I2C is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I2C bus/protocol and was developed by Intel. Implementations of the I2C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

Intel and the Intel are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

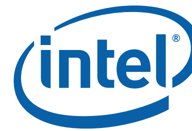
Copyright © 2008, Intel Corporation. All rights reserved.

Contents

1 Memory Interface Commands for Rendering Engine 10
1.1 Introduction 10
1.2 MI_ARB_CHECK 10
1.3 MI_ARB_ON_OFF ([DevCTG] Only) 11
1.4 MI_BATCH_BUFFER_END 12
1.5 MI_BATCH_BUFFER_START 12
1.5.1 Command Access of Privileged Memory [DevCTG] Only 15
1.5.2 Privileged Commands [DevCTG] Only 16
1.6 MI_DISPLAY_FLIP 17
1.7 MI_FLUSH 24
1.8 MI_LOAD_REGISTER_IMM 26
1.9 MI_LOAD_SCAN_LINES_EXCL 27
1.10 MI_LOAD_SCAN_LINES_INCL 28
1.11 MI_NOOP 30
1.12 MI_OVERLAY_FLIP 31
1.12.1 Turning the Overlay Off 33
1.12.2 Valid Overlay Flip Sequences 34
1.13 Surface Probing [DevCTG] 34
1.13.1 MI_PROBE [DevCTG] 34
1.13.2 MI_UNPROBE [DevCGT] 35
1.14 MI_REPORT_HEAD 36
1.15 MI_SET_CONTEXT 37
1.16 MI_STORE_DATA_IMM 40
1.17 MI_STORE_DATA_INDEX 43
1.18 MI_STORE_REGISTER_MEM 45
1.19 MI_UPDATE_GTT ([DevCTG]) 47
1.20 MI_USER_INTERRUPT 48
1.21 MI_WAIT_FOR_EVENT 49

2 Memory Interface Commands for Video Codec Engine [DevCTG+] 54
2.1 Introduction 54
2.2 MI_ARB_CHECK 54
2.3 MI_BATCH_BUFFER_END 55
2.4 MI_BATCH_BUFFER_START 55
2.5 MI_FLUSH 56
2.6 MI_LOAD_REGISTER_IMM 57
2.7 MI_NOOP 58
2.8 MI_REPORT_HEAD 59
2.9 MI_STORE_DATA_IMM 60
2.10 MI_STORE_DATA_INDEX 61
2.11 MI_USER_INTERRUPT 62
2.12 MI_WAIT_FOR_EVENT 63
2.13 Summary of Commands 64

3 Memory Interface Commands for Blitter Engine 65
3.1 Introduction 65
3.2 MI_LOAD_REGISTER_IMM 65

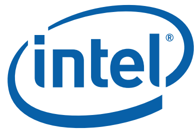


3.3 MI_NOOP ... 67
3.4 MI_ SEMAPHORE_MBOX ... 68
3.5 MI_STORE_DATA_IMM ... 70
3.6 MI_STORE_DATA_INDEX ... 72
3.7 MI_USER_INTERRUPT ... 73
3.8 MI_WAIT_FOR_EVENT ... 74

4 Graphics Memory Interface Functions 75
4.1 Introduction ... 75
4.2 Graphics Memory Clients ... 75
4.3 Graphics Memory Addressing Overview ... 76
4.3.1 Graphics Address Path ... 76
4.4 Graphics Memory Address Spaces ... 78
4.5 Address Tiling Function ... 78
4.5.1 Linear vs. Tiled Storage ... 78
4.5.2 Tile Formats ... 81
4.5.3 Tiling Algorithm ... 83
4.5.4 Tiling Support ... 84
4.5.5 Per-Stream Tile Format Support ... 88
4.6 Logical Memory Mapping ... 88
4.6.1 Logical Memory Space Mappings ... 89
4.7 Physical Graphics Memory ... 93
4.7.1 Physical Graphics Address Types ... 93
4.7.2 Main Memory ... 94

5 Device Programming Environment 96
5.1 Programming Model ... 96
5.2 Graphics Device Register Programming ... 96
5.3 Graphics Device Command Streams ... 97
5.3.1 Command Use ... 97
5.3.2 Command Transport Overview ... 97
5.3.3 Command Parser ... 98
5.3.4 The Ring Buffer ... 98
5.3.5 Batch Buffers ... 102
5.3.6 Indirect Data ... 102
5.3.7 Command Arbitration ... 103
5.3.8 Graphics Engine Synchronization ... 105
5.3.9 Graphics Memory Coherency ... 107
5.3.10 Graphics Cache Coherency ... 107

* -- Includes MI_FLUSH, Engine switch, and Context switch. 109
5.3.11 Command Synchronization ... 109
5.4 Hardware Status ... 111
5.4.1 Hardware-Detected Errors (Master Error bit) ... 111
5.4.2 Thermal Sensor Event ... 111
5.4.3 Sync Status ... 111
5.4.4 Display Plane A, B, (Sprite A, Sprite B [DevCTG] Only) Flip Pending 112
5.4.5 Overlay Flip Pending ... 112
5.4.6 Display Pipe A,B VBLANK ... 112
5.4.7 User Interrupt ... 112
5.4.8 PIPE_CONTROL Notify Interrupt ... 112
5.5 Hardware Status Writes ... 112
5.6 Interrupts ... 113
5.7 Errors ... 113

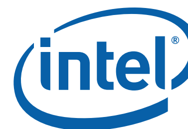


5.7.1 Error Reporting 114
5.7.2 Page Table Errors 115
5.7.3 Clearing Errors 115
5.8 Rendering Context Management 116
5.8.1 Multiple Logical Rendering Contexts 116
5.9 Reset State 119

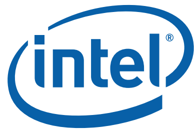
6 Frame Buffer Compression [DevCL only] 120
6.1 Overview 120
6.2 Programming Interface 120
6.2.1 FBC unit programming interface 120
6.2.2 Programming interface from Display Engine 122
6.3 Operating Modes 123
6.3.1 RLE-FBC Function Modes 123
6.3.2 Compression Modes 124
6.4 Usage Restrictions 124
6.5 Power Management Interface 125
6.6 Memory Data Structures 126
6.6.1 RLE Pixel Runs 126
6.6.2 RLE Pixel Run Sets 126
6.6.3 RLE-Compressed Line 126
6.6.4 RLE Compressed Frame and Line Length Buffers 127
6.7 Tuning Parameters 127
6.7.1 Stride 127
6.7.2 Interval 128
6.7.3 FBC Modification Counter 128
6.8 Implementation (DEBUG) 128
6.8.1 Tag Array 128
6.8.2 Compressor 129
6.8.3 Decompressor 131
6.8.4 Frame Buffer Write Detector 131
6.8.5 Coherency 132

7 Frame Buffer Compression [DevCTG] 133
7.1 DPFC Programming Interface 133
7.1.1 FBC2 supported feature and limitation 133
7.1.2 FBC2 usage model and restriction on persistent and non-persistent mode 134
7.2 DPFC Control Registers (03200h–033FFh) 136
7.2.1 DPFC_CB_BASE – DPFC Compressed Buffer Base Address 136
7.2.2 DPFC_CONTROL— DPFC Control 137
7.2.3 DPFC_RECOMP_CTL — DPFC ReComp Control 139
7.2.4 DPFC_STATUS — DPFC Status 140
7.2.5 DPFC_STATUS_2 — DPFC Status 2 140
7.2.6 DPFC_CPU_Fence_Offset — Y Offset CPU Fence Base to Display Buffer Base 141
7.2.7 PFC_SLB_DAT—DPFC SLB Data 142
7.2.8 DPFC_DEBUG_STATUS—DPFC Debug Status 143
7.2.9 DPFC_EXTRA—DPFC Extra Control Bits 144

8 BLT Engine 145
8.1 Introduction 145
8.2 Classical BLT Engine Functional Description 145
8.2.1 Basic BLT Functional Considerations 147
8.2.2 Basic Graphics Data Considerations 157
8.2.3 BLT Programming Examples 163

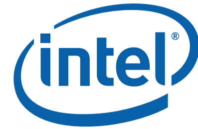


8.3 BLT Instruction Overview ... 168
8.4 BLT Engine State ... 168
8.5 Cacheable Memory Support ... 169
8.6 Device Cache Coherency: Render and Texture Caches ... 170
8.7 BLT Engine Instructions ... 170
8.7.1 Blt Programming Restrictions ... 170
8.8 Fill/Move Instructions ... 170
8.8.1 COLOR_BLT (Fill) ... 171
8.8.2 SRC_COPY_BLT (Move) ... 172
8.9 2D (X,Y) BLT Instructions ... 173
8.9.1 XY_SETUP_BLT ... 174
8.9.2 XY_SETUP_MONO_PATTERN_SL_BLT ... 176
8.9.3 XY_SETUP_CLIP_ BLT ... 177
8.9.4 XY_PIXEL_BLT ... 178
8.9.5 XY_SCANLINES_BLT ... 179
8.9.6 XY_TEXT_BLT ... 180
8.9.7 XY_TEXT_IMMEDIATE_BLT ... 181
8.9.8 XY_COLOR_BLT ... 182
8.9.9 XY_PAT_BLT ... 183
8.9.10 XY_PAT_CHROMA_BLT ... 184
8.9.11 XY_PAT_BLT_IMMEDIATE ... 186
8.9.12 XY_PAT_CHROMA_BLT_IMMEDIATE ... 188
8.9.13 XY_MONO_PAT_BLT ... 190
8.9.14 XY_MONO_PAT_FIXED_BLT ... 192
8.9.15 XY_SRC_COPY_BLT ... 197
8.9.16 XY_SRC_COPY_CHROMA_BLT ... 199
8.9.17 XY_MONO_SRC_COPY_BLT ... 201
8.9.18 XY_MONO_SRC_COPY_ IMMEDIATE_BLT ... 203
8.9.19 XY_FULL_BLT ... 205
8.9.20 XY_FULL_IMMEDIATE_PATTERN_BLT ... 207
8.9.21 XY_FULL_MONO_SRC_BLT ... 209
8.9.22 XY_FULL_MONO_SRC_IMMEDIATE_PATTERN_BLT ... 211
8.9.23 XY_FULL_MONO_PATTERN_BLT ... 213
8.9.24 XY_FULL_MONO_PATTERN_MONO_SRC_BLT ... 216
8.10 BLT Engine Instruction Field Definitions ... 218
8.10.1 BR00—BLT Opcode & Control ... 218
8.10.2 BR01—Setup BLT Raster OP, Control, and Destination Offset ... 221
8.10.3 BR05—Setup Expansion Background Color ... 224
8.10.4 BR06—Setup Expansion Foreground Color ... 225
8.10.5 BR07—Setup Color Pattern Address ... 226
8.10.6 BR09—Destination Address ... 227
8.10.7 BR11—BLT Source Pitch (Offset) ... 228
8.10.8 BR12—Source Address ... 229
8.10.9 BR13—BLT Raster OP, Control, and Destination Pitch ... 229
8.10.10 BR14—Destination Width & Height ... 231
8.10.11 BR15—Color Pattern Address ... 233
8.10.12 BR16—Pattern Expansion Background & Solid Pattern Color ... 234
8.10.13 BR17—Pattern Expansion Foreground Color ... 234
8.10.14 BR18—Source Expansion Background, and Destination Color ... 235
8.10.15 BR19—Source Expansion Foreground Color ... 235



Figures

Figure 4-1. Graphics Memory Paths....77
Figure 4-2. Rectangular Memory Operand Parameters 79
Figure 4-3. Linear Surface Layout....79
Figure 4-4. Memory Tile Dimensions 80
Figure 4-5. Tiled Surface Layout....81
Figure 4-6. Y-Major Tile Layout 82
Figure 4-7. Tiled Surface Placement....86
Figure 4-8. Global and Render GTT Mapping....90
Figure 4-9. GTT Re-mapping to Handle Differing Pitches 92
Figure 4-10. Logical-to-Physical Graphics Memory Mapping 92
Figure 4-11. Memory Interfaces 93
Figure 4-12. Memory Pages backing Color and Depth Buffers....95
Figure 5-1. Graphics Controller Command Interface 98
Figure 5-2. Ring Buffer....99
Figure 5-3. Batch Buffer Chaining 102
Figure 6-1. 32bpp Pixel Run....126
Figure 6-2. 16bpp Pixel Run....126
Figure 6-3. Pixel Run Set....126
Figure 6-4. RLE-Compression Buffers 127
Figure 8-1. Block Diagram and Data Paths of the BLT Engine....146
Figure 8-2. Block Diagram and Data Paths of the BLT Engine....152
Figure 8-3. Source Corruption in BLT with Overlapping Source and Destination Locations 154
Figure 8-4. Correctly Performed BLT with Overlapping Source and Destination Locations 155
Figure 8-5. Suggested Starting Points for Possible Source and Destination Overlap Situations 156
Figure 8-6. Representation of On-Screen Single 6-Pixel Line in the Frame Buffer 157
Figure 8-7. Representation of On-Screen 6x4 Array of Pixels in the Frame Buffer....158
Figure 8-8. Pattern Data -- Always an 8x8 Array of Pixels....160
Figure 8-9. 8bpp Pattern Data -- Occupies 64 Bytes (8 quadwords) 160
Figure 8-10. 16bpp Pattern Data -- Occupies 128 Bytes (16 quadwords) 161
Figure 8-11. 32bpp Pattern Data -- Occupies 256 Bytes (32 quadwords) 161
Figure 8-12. On-Screen Destination for Example Pattern Fill BLT 163
Figure 8-13. Pattern Data for Example Pattern Fill BLT....164
Figure 8-14. Results of Example Pattern Fill BLT 165
Figure 8-15. On-Screen Destination for Example Character Drawing BLT 166
Figure 8-16. Source Data in System Memory for Example Character Drawing BLT....166
Figure 8-17. Results of Example Character Drawing BLT 168



Tables

Table 4-1. Graphics Memory Clients ... 75
Table 4-2. Graphics Memory Address Types ... 78
Table 4-3. X-Major Tile Layout ... 82
Table 4-4. Physical Memory Address Types ... 93
Table 5-1. Ring Buffer Characteristics ... 100
Table 5-2. Graphics Memory Coherency ... 107
Table 5-3. Page Table Error Types ... 115
Table 8-1. Bit-Wise Operations and 8-Bit Codes (00-3F) ... 148
Table 8-2. Bit-Wise Operations and 8-bit Codes (40 - 7F) ... 149
Table 8-3. Bit-Wise Operations and 8-bit Codes (80 - BF) ... 150
Table 8-4. Bit-Wise Operations and 8-bit Codes (C0 - FF) ... 151

Revision History

Document Number	Revision Number	Description	Revision Date
24513	1.0a	Initial release.	January 2008
321392-001	2.0a	Cantiga Release	January 2009

§§

1 Memory Interface Commands for Rendering Engine

1.1 Introduction

This chapter describes the formats of the "Memory Interface" commands, including brief descriptions of their use. The functions performed by these commands are discussed fully in the *Memory Interface Functions* Device Programming Environment chapter.

This chapter describes MI Commands for the original graphics processing engine. The term "for Rendering Engine" in the title has been added to differentiate this chapter from a similar one describing the MI commands for the Media Decode Engine.

The commands detailed in this chapter are used across products within the GenX family. However, slight changes may be present in some commands (i.e., for features added or removed), or some commands may be removed entirely. Refer to the *Preface* chapter for product specific summary.

1.2 MI_ARB_CHECK

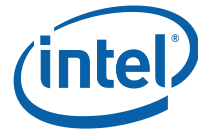
MI_ARB_CHECK			
Project: All		**Length Bias:**	1

The MI_ARB_CHECK instruction is used to check the ring buffer double buffered head pointer (register UHPTR). This instruction can be used to pre-empt the current execution of the ring buffer. Note that the valid bit in the updated head pointer register needs to be set for the command streamer to be pre-empted.

Programming Note:

- The current head pointer is loaded with the updated head pointer register independent of the location of the updated head
- If the current head pointer and the updated head pointer register are equal, hardware will automatically reset the valid bit corresponding to the UHPTR
- This instruction can be placed only in a ring buffer, never in a batch buffer.
- For pre-emption, the wrap count in the ring buffer head register is no longer maintained by hardware. The hardware updates the wrap count to the value in the UHPTR register.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 05h MI_ARB_CHECK Format: OpCode
	22:0	**Reserved** Project: All Format: MBZ



1.3 MI_ARB_ON_OFF ([DevCTG] Only)

MI_ARB_ON_OFF

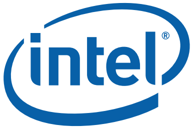
Project:	CTG+	**Length Bias:**	1

The MI_ARB_ON_OFF instruction is used to disable/enable context switching. This instruction can be used to prevent submission of a new run list from interrupting a command sequence. Note that context switching will remain disabled until re-enabled through use of this command. This command will also prevent a switch in the case of waiting on events, running out of commands or a surface probe fault. These will effectively hang the device if allowed to occur while arbitration is off (context switching is disabled.)

This command should always be used as an off-on pair with the sequence of instructions to be protected from context switch between MI_ARB_OFF and MI_ARB_ON. Software must use this arbitration control with caution since it has the potential to increase the response time of the Render Engine to pre-emption requests.

This is a privileged command; it will not be effective (will be converted to a no-op) if executed from within a non-secure batch buffer. This command can only be issued when **Per-Process Virtual Address Space and context queuing** is set; if the bit is set it will be converted to NOOP.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 08h MI_ARB_ON_OFF Format: OpCode
	22:1	**Reserved** Project: All Format: MBZ
	0	**Arbitration Enable** Format: Enable This field enables or disables context switches due to pre-emption (a new context queuing). **Value** / **Name** 0h Disabled 1h Enabled

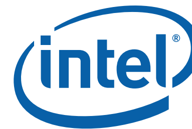


1.4 MI_BATCH_BUFFER_END

MI_BATCH_BUFFER_END			
Project: All		**Length Bias:** 1	
The MI_BATCH_BUFFER_END command is used to terminate the execution of commands stored in a *batch buffer* initiated using a MI_BATCH_BUFFER_START command.			
DWord	**Bit**	**Description**	
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	**MI Command Opcode** Default Value: 0Ah MI_BATCH_BUFFER_END Format: OpCode	
	22:0	**Reserved** Project: All Format: MBZ	

1.5 MI_BATCH_BUFFER_START

MI_BATCH_BUFFER_START			
Project: All		**Length Bias:** 2	
The MI_BATCH_BUFFER_START command is used to initiate the execution of commands stored in a *batch buffer*. For restrictions on the location of batch buffers, see Batch Buffers in the Device Programming Interface chapter of *MI Functions*. The batch buffer can be specified as secure or non-secure, determining the operations considered valid when initiated from within the buffer and any attached (chained) batch buffers. See Batch Buffer Protection in the Device Programming Interface chapter of *MI Functions*. **Programming Notes:** • Batch buffers referenced with physical addresses must not extend beyond the end of the starting physical page (can't span physical pages). However, a batch buffer initiated using a physical address can chain to another buffer in another physical page. • A batch buffer initiated with this command must end either with a MI_BATCH_BUFFER_END command or by chaining to another batch buffer with an MI_BATCH_BUFFER_START command. • For virtual batch buffers, it is essential that the address location beyond the current page be populated inside the GTT. HW performs over-fetch of the command addresses and any over-fetch requires a valid TLB entry. A single extra page beyond the batch buffer is sufficient.			
DWord	**Bit**	**Description**	
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode	

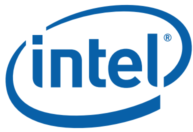


<table>
<tr><th colspan="3">MI_BATCH_BUFFER_START</th></tr>
<tr><td></td><td>28:23</td><td>MI Command Opcode
Default Value: 31h MI_BATCH_BUFFER_START Format: OpCode</td></tr>
<tr><td></td><td>22:12</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>11

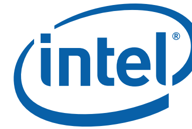
10:9</td><td>Reserved Project: All Format: MBZ

Command Arbitration Control
This field controls where command arbitration can occur during the batch buffer.

<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Arbitrate only at chain points</td><td>Legacy Mode. Overridden by MI_ARB_ON_OFF = Off</td><td>All</td></tr><tr><td>1h</td><td>Arbitrate between commands</td><td>Arbitration can occur between any pair of commands, or during execution of a primitive command. Overridden by MI_ARB_ON_OFF = Off</td><td>All</td></tr><tr><td>2h</td><td>Reserved</td><td></td><td>All</td></tr><tr><td>3h</td><td>No Arbitration</td><td>The Batch Buffer execution cannot be pre-empted until control returns to the initiating ring. I.e., command arbitration does not occur during or between batch buffer chains. This avoids software from having to place MI_ARB_ON_OFF packets around batch buffers to prevent interruption.</td><td>All</td></tr></table></td></tr>
<tr><td></td><td>8</td><td>Reserved Project: All Format: MBZ
Although Buffer Security Indicator is implemented, there is no usage model for it and it need not be validated.</td></tr>
</table>



MI_BATCH_BUFFER_START		
	7	**Memory Space Select** Project: All Specifies memory space associated with the **Buffer Start Address.** **Value** | **Name** | **Description** | **Project** 0h | Physical Memory | Physical Main (unsnooped) Memory. The 4 bits of the **Batch Buffer Start Address Extension** are prefixed to bits 31:6 of **Buffer Start Address** to specify an address within physical main memory. In this mode the hardware must not fetch data beyond a 4KB boundary. | All 1h | Graphics Memory | (GTT-mapped) Bits 31:2 of a graphics memory address. The GGTT is used to translate this address. | All **Programming Notes** | **Project** Batch buffers referenced with physical addresses must not extend beyond the end of the starting physical page (can't span physical pages). However, a batch buffer initiated using a physical address can chain to another buffer in another physical page. | All Batch buffers can chain between (but cannot span) memory spaces. | All
	6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total - Bias
1	31:6	**Batch Buffer Start Address** Project: All Address: SelectableAddress(Memory Space Select)[31:6] Surface Type: BatchBuffer This field specifies Bits 31:6 of the starting address of the 64B aligned batch buffer. The address space used depends on *Memory Space Select* (see above).



MI_BATCH_BUFFER_START		
	5:4	**Reserved** Project: All Format: MBZ
	3:0	**Batch Buffer Start Address Extension** Project: All Address: PhysicalAddressExtension[35:32] This field specifies bits 35:32 of the starting address of the 64B-aligned physical batch buffer. This field must be zero for non-physical Batch Buffers.

1.5.1 Command Access of Privileged Memory [DevCTG] Only

Memory space mapped through the global GTT is considered "privileged" memory. Commands that have the capability of accessing both privileged and unprivileged (PPGTT space) memory will contain a bit that, if set, will attempt a "privileged" access through the GGTT rather than an unprivileged access through the context-local PPGTT.

"User mode" command buffers should not be able to access privileged memory under any circumstances. These command buffers will be issued by the kernel mode driver with the batch buffer's **Buffer Security** Indicator set to "non-secure". Commands in such a batch buffer are not allowed to access privileged memory. The commands in these buffers are supplied by the user mode driver and will not be validated by the kernel mode driver. For a batch buffer marked as non-secure if **Per-Process Virtual Address Space and Run List Enable is set**, the command buffer fetches are generated using the PPGTT space.

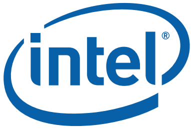
"Kernel mode" command buffers are allowed to access privileged memory. The batch buffers Buffer Security indicator is set to "secure" in this case. In some of the commands that access memory in a secure batch buffer, a bit is provided in the command to steer the access to Per process or Global virtual space. Secure batch buffers are executed from the global GTT.

Commands in ring buffers and commands in batch buffers that are marked as secure (by the kernel mode driver) are allowed to access both privileged and unprivileged memory and may choose on a command-by-command basis.

Command	Address	Allowed Access
MI_BATCH_BUFFER_START*	**Command Address**	**Selectable**
MI_DISPLAY_FLIP	**Display Buffer Base**	**GGTT Only**
MI_STORE_DATA_IMM*	**Storage Address**	**Selectable**
MI_STORE_DATA_INDEX**	**Storage Offset**	**Selectable****
MI_STORE_REGISTER_MEM	**Storage Address**	**Selectable**
MI_SEMAPHORE_MBOX	**Semaphore Address**	**Selectable**

*Command has a GGTT/PPGTT selector added to it vs. previous GenX family products.

**Added bit allows offset to apply to global HW Status Page or PP HW Status Page found in context image.

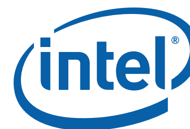


1.5.2 Privileged Commands [DevCTG] Only

A subset of the commands are privileged. These commands may be issued only from a secure batch buffer or directly from a ring. If one of these commands is parsed in a non-secure batch buffer, an error is flagged and the command is dropped. For commands that generates a write, the hardware will complete the transaction but the byte enables are turned off. Batch buffers from the User mode driver are passed directly to the kernel mode driver which does not validate them but issues them with the Security Indicator set to 'non-secure' to protect the system from an attack using these privileged commands.

Privileged Command	Function in non-privileged batch buffers
MI_ARB_ON_OFF	**Command is ignored by the hardware**
MI_LOAD_REGISTER_IMM	**Byte enables are turned off**
MI_UPDATE_GTT	**Byte enabled are turned off**
MI_STORE_DATA_IMM	**Command is translated using the Per process GTT if** Per-Process Virtual Address Space and Run List Enable is set
MI_STORE_DATA_INDEX	**Command is translated using the Per process hardware status page if** Per-Process Virtual Address Space and Run List Enable is set
MI_STORE_REGISTER_MEM	**Command is translated and completed with byte enables turned off**
MI_DISPLAY_FLIP	**Command is ignored by the hardware**

Command privilege applies the same way in single-context scheduler mode. Parsing one of the commands in the table above from a non-secure batch buffer will flag an error and convert the command to a NOOP.



1.6 MI_DISPLAY_FLIP

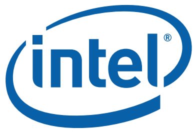
MI_DISPLAY_FLIP			
Project:	All	**Length Bias:**	

The MI_DISPLAY_FLIP command is used to request a specific display plane to switch (flip) to display a new buffer. The buffer is specified with a starting address and pitch. The tiled attribute of the buffer start address is programmed as part of the packet.

The operation this command performs is also known as a "display flip request" operation – in that the flip operation itself will occur at some point in the future. This command specifies when the flip operation is to occur: either synchronously with vertical retrace to avoid tearing artifacts (possibly on a future frame), or asynchronously (as soon as possible) to minimize rendering stalls at the cost of tearing artifacts.

Programming Notes:

1. Prior to a display flip operation being requested, software must ensure that the new display buffer is coherent in memory. This will typically require MI_DISPLAY_FLIP to be included in a PIPE_CONTROL command to flush pending rendering operations and any pending write buffers/caches, although the use of an MI_FLUSH command will also suffice albeit with greater performance penalty. (Note that completion of the MI_FLUSH command does not guarantee that previous outstanding flip operations have completed).
2. This command simply requests a display flip operation -- command execution then continues normally. There is no guarantee that the flip (even if asynchronous) will occur prior to subsequent commands being executed. (Note that completion of the MI_FLUSH command does not guarantee that outstanding flip operations have completed). The MI_WAIT_FOR_EVENT command can be used to provide this synchronization – by pausing command execution until a pending flip has actually completed. This synchronization can also be performed by use of the Display Flip Pending hardware status. See Display Flip Synchronization in the Device Programming Interface chapter of *MI Functions*.
3. After a display flip operation is requested, software is responsible for initiating any required synchronization with subsequent buffer clear or rendering operations. For multi-buffering (e.g., double buffering) operations, this will typically require updating SURFACE_STATE or the binding table to change the rendering (back) buffer. In addition, prior to any subsequent clear or rendering operations, software must typically ensure that the new rendering buffer is not actively being displayed. Again, the MI_WAIT_FOR_EVENT command or Display Flip Pending hardware status can be used to provide this synchronization. See Display Flip Synchronization in the Device Programming Interface chapter of *MI Functions.*
4. The display buffer command uses the X and Y offset for the tiled buffers from the Display Interface registers. Software is allowed to change the offset via the MMIO interface irrespective of the flip commands enqueued in the command stream. For tiled buffers, the display subsystem uses the X and Y offset in generation of the final request to memory. The offset is always updated on the next vblank for both Synchronous and Asynch Flips. It is not necessary to have a flip enqueued to update the X and Y offset
5. The display buffer command uses the linear dword offset for the linear buffers from the Display Interface registers. Software is allowed to change the offset via the MMIO interface irrespective of the flip commands enqueued in the command stream. For linear buffers, the display subsystem uses the dword offset in



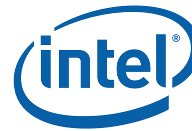
MI_DISPLAY_FLIP

generation of the final request to memory.

- For synchronous flips the offset is updated on the next vblank. It is not necessary to have a sync flip enqueued to update the dword offset.
- Linear memory does not support asynchronous flips

6. DWord 3 (panel fitter flip) must not be sent with asynchronous flips. It is only allowed to be sent with synchronous flips.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 14h MI_DISPLAY_FLIP Format: OpCode

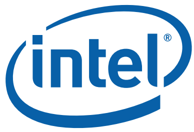


MI_DISPLAY_FLIP

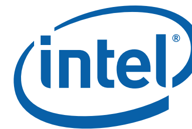
	22	**Asynchronous Flip** Project: All Format: Boolean This field specifies whether the flip operation should be performed asynchronously to vertical retrace. If FALSE, the flip will occur during the vertical blanking interval – thus avoiding any tearing artifacts. If TRUE, the flip will occur “as soon as possible” – and may exhibit tearing artifacts

Value	Name	Description	Project
0h	Asynchronous Flip		All
1h	Synchronous Flip		All

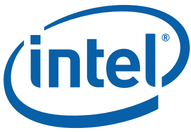
Programming Notes	Project
• This command must not be used to perform an Asynchronous Flip to the same address as specified in the previously executed Asynchronous Flip, or the device operation is UNDEFINED. • The **Display Buffer Pitch and Tile parameter** fields are ignored for asynchronous flips (i.e., the new buffer must have the same pitch/tile format as the previous buffer). • **Supported on X-Tiled Frame buffers only.** • For Asynch Flips the Buffers used must be 32KB aligned. • The display stride must be >=8KB when doing Asynch Flips together with 180 display rotation. • The display stride must be power of 2 when doing Asynch Flips. • Supported on Display Planes A and B only • Not supported via the flip queue (if this bit is set, Flip Queue Select must be 0)	All



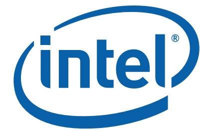
<table>
<tr><th colspan="3">MI_DISPLAY_FLIP</th></tr>
<tr><td></td><td>21:20</td><td>Display (Plane) Select
Project: All
Format: U2
This field selects which display plane is to perform the flip operation.
<table>
<tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr>
<tr><td>0h</td><td>Display Plane A</td><td></td><td>All</td></tr>
<tr><td>1h</td><td>Display Plane A</td><td></td><td>All</td></tr>
<tr><td>2h</td><td>Display Plane C</td><td></td><td>All</td></tr>
<tr><td>3h</td><td>Display Sprite A</td><td></td><td>Reserved</td></tr>
<tr><td>3h</td><td>Reserved</td><td></td><td>All</td></tr>
<tr><td>3h</td><td>Display Sprite B</td><td></td><td>Reserved</td></tr>
</table></td></tr>
<tr><td></td><td>19:6</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>5:0</td><td>DWord Length
Default Value: 0h Excludes DWord (0,1)
Format: =n Total Length - 2</td></tr>
<tr><td>1</td><td>31:30</td><td>Reserved Project: All Format: MBZ</td></tr>
</table>



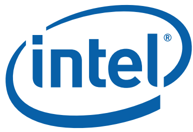
<table>
<tr><th colspan="3">MI_DISPLAY_FLIP</th></tr>
<tr><td></td><td>29</td><td>Flip Queue Select
Project: All
This field selects whether this flip is placed in the flip queue or is a standard (legacy) flip request.
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Standard Flip</td><td>Use standard (legacy) synchronous or asynchronous flipping</td><td>All</td></tr><tr><td>1h</td><td>Enqueue Flip</td><td>Enqueue Flip (see Display Functions for a description of the Flip Queue)</td><td>All</td></tr></table><table><tr><th>Programming Notes</th><th>Project</th></tr><tr><td>Performing a legacy synchronous or asynchronous flip will drop any outstanding flips in the flip queue as well as any previous synchronous flip that has not yet completed.</td><td>All</td></tr></table></td></tr>
<tr><td></td><td>28:15</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>14:3</td><td>Display Buffer Pitch
Project: All
Default Value: 0h DefaultVaueDesc
Format: U12 Quad Words
For Synchronous or Queued Flips only, this field specifies the QWord pitch of the new display buffer.
For Asynchronous Flips, this parameter is ignored. All the flips in a flip chain should maintain the same pitch as programmed with the last synchronous flip or direct thru mmio.</td></tr>
<tr><td></td><td>2:0</td><td>Reserved Project: All Format: MBZ</td></tr>
</table>



<table>
<tr><th colspan="3">MI_DISPLAY_FLIP</th></tr>
<tr><td rowspan="3">2</td><td>31:12</td><td>Display Buffer Base Address
Project: All
Address: GraphicsAddress[31:12]
This field specifies Bits 31:12 of the Graphics Address of the new display buffer. The display buffer must be pixel aligned within the Graphics Address space. (Refer to the Display Address Start Address Register description in the Display Registers chapter).
Programming Notes
• The Display buffer must reside completely in Main Memory
• This address is always translated via the global (rather than per-process) GTT</td></tr>
<tr><td>11:1</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td>0</td><td>Tile Parameter
Project: All
Default Value: 0h DefaultVaueDesc
Address: GraphicsAddress[31:0]
For Asynchronous Flips, this parameter is ignored. All the flips in a flip chain should maintain the same tile parameter as programmed with the last synchronous flip or direct thru mmio.
For Synchronous Flips, tile parameter can change for different flips in the flip chain
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Linear</td><td>For Syncronous Flips Only</td><td>All</td></tr><tr><td>1h</td><td>Tiled X</td><td></td><td>All</td></tr></table></td></tr>
<tr><td rowspan="2">3</td><td>31</td><td>Enable Panel Fitter Project: All Format: Enable
Enables the panel fitter on the pipe attached to the plane selected for this flip.</td></tr>
<tr><td>30</td><td>Panel Fitter Select
Project: All
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>7x5</td><td>Select 7x5 capable panel fitter</td><td>All</td></tr><tr><td>1h</td><td>3x3</td><td>Select 3x3 capable panel fitter</td><td>All</td></tr></table></td></tr>
</table>



MI_DISPLAY_FLIP		
	29:28	**Reserved** Project: All Format: MBZ
	27:16	**Pipe Horizontal Source Image Size** Project: All Format: U32 This 12-bit field specifies Horizontal source image size up to 4096. This determines the size of the image created by the display planes sent to the blender. The value programmed should be the source image size minus one. This field obeys all the rules of the Horizontal Source Image Size registers. The pipe affected will be the pipe attached to the plane selected for this flip.
	15:12	**Reserved** Project: All Format: MBZ
	11:0	**Pipe Vertical Source Image ReSize** Project: All Format: U32 This 12-bit field specifies the new vertical source image size up to 4096 lines. This determines the size of the image created by the display planes sent to the blender. The value programmed should be the source image size minus one. This field obeys all the rules of the Vertical Source Image Size registers. The pipe affected will be the pipe attached to the plane selected for this flip.



1.7 MI_FLUSH

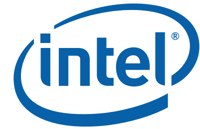
MI_FLUSH			
Project:	All	**Length Bias:**	1

The MI_FLUSH command is used to perform an internal "flush" operation. The parser pauses on an internal flush until all drawing engines have completed any pending operations and the read caches are invalidated including the texture cache accessed via the Sampler or the data port. In addition, this command can also be used to:

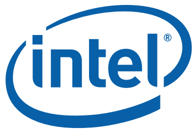
1. Flush any dirty data in the Render Cache to memory. This is done by default, however this can be inhibited.
2. Invalidate the state and command cache.

Usage note: After this command is completed and followed by a Store DWord-type command, CPU access to graphics memory will be coherent (assuming the Render Cache flush is not inhibited).

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 04h MI_FLUSH Format: OpCode
	22:6	**Reserved** Project: All Format: MBZ
	5:4	**Reserved** Project: All Format: MBZ
	3	**Global Snapshot Count Reset** Project: All Format: Boolean If set, the snapshot registers defined for the GenX debug capability are reset after the flush completes. The Statistics Counters are also reset; SW should never set this bit during normal operation since the Statistics Counters are intended to be free running. **Programming Notes** / **Project** PS_DEPTH_COUNT and TIMESTAMP are *not* reset by MI_FLUSH with this bit set. TIMESTAMP and PS_DEPTH_COUNT can be reset by writing 0 to them / All **Value** / **Name** / **Description** / **Project** 0h / Don't Reset / Do not reset the snapshot counts or Statistics Counters. / All 1h / Reset / Reset the snapshot count in GenX for all the units and reset the Statistics Counters except as noted above. / All

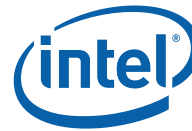


<table>
<tr><th colspan="3">MI_FLUSH</th></tr>
<tr><td></td><td>2</td><td>Render Cache Flush Inhibit Project: All Format: Boolean
If set, the Render Cache is not flushed as part of the processing of this command.
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Flush</td><td>Flush the Render Cache</td><td>All</td></tr><tr><td>1h</td><td>Don’t Flush</td><td>Do not flush the Render Cache</td><td>All</td></tr></table></td></tr>
<tr><td></td><td>1</td><td>State/Instruction Cache Invalidate Project: All Format: Boolean
If set, Invalidates the State and Instruction Cache
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Don’t Invalidate</td><td>Leave State/Instruction Cache unaffected</td><td>All</td></tr><tr><td>1h</td><td>Invalidate</td><td>Invalidate State/Instruction Cache</td><td>All</td></tr></table></td></tr>
<tr><td></td><td>0</td><td>Reserved Project: All Format: MBZ</td></tr>
</table>



1.8 MI_LOAD_REGISTER_IMM

<table>
<tr><th colspan="3">MI_LOAD_REGISTER_IMM</th></tr>
<tr><td colspan="3">Project: All | Length Bias: 2</td></tr>
<tr><td colspan="3">The MI_LOAD_REGISTER_IMM command requests a write of up to a DWord constant supplied in the command to the specified Register Offset (i.e., offset into Memory-Mapped Register Range). The register is loaded before the next command is executed.
Programming Notes:
The behavior of this command is controlled by Dword 3, Bit 8 (Disable Register Access) of the RINGBUF register. If this command is disallowed then the command stream converts it to a NOOP.
If this command is executed from a BB then the behavior of this command is controlled by Dword 0, Bit 8 (Security Indicator) of the BATCH_BUFFER_START Command. If the batch buffer is insecure then the command stream converts this command to a NOOP. Note that the corresponding ring buffer must allow a register update for this command to execute.</td></tr>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>0</td><td>31:29</td><td>Command Type
Default Value: 0h MI_COMMAND Format: OpCode</td></tr>
<tr><td></td><td>28:23</td><td>MI Command Opcode
Default Value: 22h MI_ Format: OpCode</td></tr>
<tr><td></td><td>22:12</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>11:8</td><td>Byte Write Disables
Format: Enable[4] Bit 8 corresponds to Data DWord [7:0]
Range Must specify a valid register write operation
This field specifies which bytes of the Data DWord are not to be written to the DWord offset specified in Register Offset.</td></tr>
<tr><td></td><td>7:6</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>5:0</td><td>DWord Length
Default Value: 1h Excludes DWord (0,1)
Format: =n Total Length - 2</td></tr>
<tr><td>1</td><td>31:2</td><td>Register Offset
Format: U30
Address: MmioAddress[31:2]
This field specifies bits [31:2] of the offset into the Memory Mapped Register Range (i.e., this field specifies a DWord offset).</td></tr>
<tr><td></td><td>1:0</td><td>Reserved Project: All Format: MBZ</td></tr>
</table>



MI_LOAD_REGISTER_IMM		
2	31:0	**Data DWord** Mask: Bytes Write Disables Format: U32 This field specifies the DWord value to be written to the targeted location.

1.9 MI_LOAD_SCAN_LINES_EXCL

MI_LOAD_SCAN_LINES_EXCL

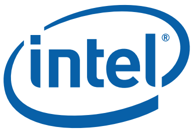
Project:	All	Length Bias:	2

The MI_LOAD_SCAN_LINES_EXCL command is used to initialize the Scan Line Window registers for a specific Display Pipe. If the display refresh is ***outside*** this window the Display Engine asserts a signal that is used by the command parser to process the WAIT_FOR_EVENT command (i.e., the parser will wait while outside). This command overrides the Scan Line Window defined by any previous MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL commands targeting the specific display pipe.

Note: The two scan-line numbers are inclusive. If programmed to the same values, that single line defines the region in question.

Always place an even number of MI_LOAD_SCAN_LINES_EXCL/INCL at a time into the ring buffer. If only a single MI_LOAD_SCAN_LINES_EXCL/INCL is desired, just add a second identical MI_LOAD_SCAN_LINES_EXCL/INCL command.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 13h MI_LOAD_SCAN_LINES_EXCL Format: OpCode
	22	**Reserved** Project: All Format: MBZ
	21:20	**Display Pipe Select** Project: All Format: U2 This field selects which Display Engine (pipe) this command is targeting. **Value** / **Name** / **Description** / **Project** 0h / Display Pipe A / / All 1h / Display Pipe B / / All



MI_LOAD_SCAN_LINES_EXCL		
	19:6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2
1	31:16	**Start Scan Line Number** Project: All Format: U16 In scan lines, where scan line 0 is the first line of the display frame. Range [0,Display Buffer height in lines-1] This field specifies the starting scan line number of the Scan Line Window.
	31:16	**End Scan Line Number** Project: All Format: U16 In scan lines, where scan line 0 is the first line of the display frame. Range [0,Display Buffer height in lines-1] This field specifies the ending scan line number of the Scan Line Window.

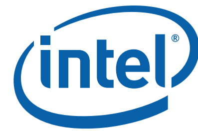
1.10 MI_LOAD_SCAN_LINES_INCL

MI_LOAD_SCAN_LINES_INCL			
Project:	All	**Length Bias:**	2

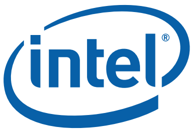
The MI_LOAD_SCAN_LINES_INCL command is used to initialize the Scan Line Window registers for a specific Display Engine. If the display refresh is ***within*** this window the Display Engine asserts a signal that is used by the command parser to process the WAIT_FOR_EVENT command (i.e., the parser will wait while inside of the window). This command overrides the Scan Line Window defined by any previous MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL commands targeting the specific display.

Always place an even number of MI_LOAD_SCAN_LINES_EXCL/INCL at a time into the ring buffer. If only a single MI_LOAD_SCAN_LINES_EXCL/INCL is desired, just add a second identical

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 12h MI_LOAD_SCAN_LINES_INCL Format: OpCode
	22	**Reserved** Project: All Format: MBZ

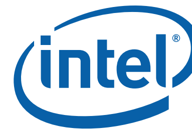


MI_LOAD_SCAN_LINES_INCL		
	21:20	**Display Pipe Select** Project: All Format: U2 This field selects which Display Engine (pipe) this command is targeting. **Value** / **Name** / **Description** / **Project** 0h / Display Pipe A / / All 1h / Display Pipe B / / All
	19:6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2
1	31:16	**Start Scan Line Number** Project: All Format: U16 In scan lines, where scan line 0 is the first line of the display frame. Range [0,Display Buffer height in lines-1] This field specifies the starting scan line number of the Scan Line Window.
	31:16	**End Scan Line Number** Project: All Format: U16 In scan lines, where scan line 0 is the first line of the display frame. Range [0,Display Buffer height in lines-1] This field specifies the ending scan line number of the Scan Line Window.



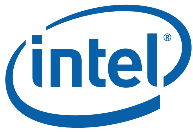
1.11 MI_NOOP

<table>
<tr><th colspan="3">MI_NOOP</th></tr>
<tr><td colspan="3">Project: All   Length Bias: 1</td></tr>
<tr><td colspan="3">The MI_NOOP command basically performs a “no operation” in the command stream and is typically used to pad the command stream (e.g., in order to pad out a batch buffer to a QWord boundary). However, there is one minor (optional) function this command can perform – a 22-bit value can be loaded into the MI NOPID register. This provides a general-purpose command stream tagging ("breadcrumb") mechanism (e.g., to provide sequencing information for a subsequent breakpoint interrupt).
Performance Note: The process time to execute a NOP command is min of 6 clock cycles. One example usage of the improved NOP throughput is for some multi-pass media application whereas some unwanted media object commands are replaced by MI_NOOP without repacking the commands in a batch buffer.</td></tr>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td rowspan="4">0</td><td>31:29</td><td>Command Type
Default Value: 0h MI_COMMAND   Format: OpCode</td></tr>
<tr><td>28:23</td><td>MI Command Opcode
Default Value: 0h MI_NOOP   Format: OpCode</td></tr>
<tr><td>22</td><td>Identification Number Register Write Enable
Project: All
Format: Enable
This field enables the value in the Identification Number field to be written into the MI NOPID register. If disabled, that register is unmodified – making this command an effective “no operation” function.
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Disable</td><td>Do not write the NOP_ID register.</td><td>All</td></tr><tr><td>1h</td><td>Enable</td><td>Write the NOP_ID register.</td><td>All</td></tr></table></td></tr>
<tr><td>31:0</td><td>Identification Number   Project: All   Format: U22
This field contains a 22-bit number which can be written to the MI NOPID register.</td></tr>
</table>

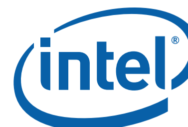


1.12 MI_OVERLAY_FLIP

<table>
<tr><th colspan="4">MI_OVERLAY_FLIP</th></tr>
<tr><td>Project:</td><td>All</td><td>Length Bias:</td><td>2</td></tr>
<tr><td colspan="4">The MI_OVERLAY_FLIP command is used to specify memory buffers that will (optionally) be used during the next Vertical Blank period to update the specified Overlay control register set and Overlay filter coefficients (respectively). The update of the Overlay registers is referred to as an “Overlay Flip”, making this command an “Overlay Flip Request”.
Programming Notes:
1. Prior to an overlay flip operation being requested, software must ensure that the memory buffer used to update the overlay registers is coherent (i.e., there are no outstanding buffered writes to that memory buffer).
2. Prior to an overlay flip operation being requested, software must ensure that the new overlay buffer is coherent in memory. This will typically require the use of an MI_FLUSH command to flush pending rendering operations and any pending write buffers/caches.
3. This command simply requests an overlay flip operation -- command execution then continues normally. There is no mechanism to prevent a new flip request from overriding any outstanding flip request. (Note that completion of the MI_FLUSH command does not guarantee that outstanding flip operations have completed). The MI_WAIT_FOR_EVENT command can be used to provide this synchronization – by pausing command execution until a pending overlay flip has actually completed or that the display refresh has proceeded past a specific scan line window. This synchronization can also be performed by use of the Overlay Flip Pending hardware status. See Overlay Flip Synchronization in the Device Programming Interface chapter of MI Functions.
4. After an overlay flip operation is requested, software is responsible for initiating any required synchronization with subsequent buffer clear or rendering operations targeting the previous (“flipped-from”) overlay buffer.
5. Registers and Coefficients are located in Main memory.</td></tr>
</table>



MI_OVERLAY_FLIP		
DWord	**Bit**	**Description**
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 11h MI_OVERLAY_FLIP Format: OpCode
	22:21	**Mode Flags** Project: All Format: U2 **Value** / **Name** / **Description** / **Project** 00b / Flip Continue / Do not flush or change the state of the Render Cache or Overlay. / All 01b / Flip On / Flush Render Cache, drawing pipeline and then set render cache in overlay Mode before executing the Flip. The Flip turns on the overlay engine. This Render Cache flush is not applicable in a Mobile Gfx controller which has an independent overlay data buffer. / All 10b / Flip Off / Flush Render Cache, drawing pipeline and then clear Overlay Mode and turn off the overlay engine. Do not update registers and coefficients from memory. This Render Cache flush is required because overlay shares the render cache in desktop graphics controllers. This bit is generally not applicable in a Mobile graphics controller which has an independent overlay data buffer. / All 11b / Reserved / / All
	20:6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2

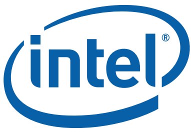


MI_OVERLAY_FLIP		
1	31:12	**Register and Coefficient Update Address** Project: All Address: GlobalGraphicsAddress[31:12] Surface Type: U32 This field specifies the memory buffer used to update the overlay registers and Coefficients. The Overlay Update Address Register specifies a **Global GTT** address used by the Overlay at the next VBLANK event to start requesting overlay control register and Coefficient data from memory. Software should ensure that the **Global GTT** address is **page-aligned**, so that the entire overlay control registers and coefficients are within one 4K page.
	11:1	**Reserved** Project: All Format: MBZ
	0	**Overlay Filter Coefficient Register Update Flag (OFC_UPDATE)** Project: All This field indicates if hardware should load overlay filter coefficients from memory. Turning overlay off without loading the Overlay Filter Coefficient registers via MI_OVERLAY_FLIP can lead to a hang. **Value** | **Name** | **Description** | **Project** 0h | Don't Update | Do not update overlay filter coefficients. | All 1h | Update | Hardware loads the overlay filter coefficients from memory to on-chip registers. | All

1.12.1 Turning the Overlay Off

The Overlay Engine is turned off by issuing an MI_OVERLAY_FLIP with the **Mode Flags** set to '10'b (aka "Flip Off), thereby flushing and reconfiguring the internal caches and putting the Overlay Engine into a low-power state. Software must ensure that the subsequent Overlay Flip has occurred at the next associated VBlank, typically by use of the **Overlay Flip Pending Wait Enable** bit of the MI_WAIT_FOR_EVENT command. In addition, the Display Pipe to which the overlay is attached must continue running until the sequence completes, or device operation is UNDEFINED.

In order to completely shutdown the Overlay Engine, and additional step is required before the use of the "Flip Off" sequence (as described above). The Overlay Enable (OV_ENBL) bit of the Overlay Command (OCOMD) Register must be cleared via a normal Overlay Register load accomplished via issuance of an MI_OVERLAY_FLIP with Mode Flags = '00'b (aka Flip Continue). This operation will effectively turn off the display of the overlay. Note that a wait-for-overlay-VBlank must be used to ensure this Flip Continue has



completed. The subsequent Flip-Off sequence (above) will reconfigure the cache for non-overlay operation and gracefully power down the Overlay Engine.

1.12.2 Valid Overlay Flip Sequences

The only architecturally valid Overlay Flip sequence is shown below:

FlipOn

some number of FlipContinues

FlipOff

For example, multiple FlipOn commands (without intervening FlipOff commands) are invalid; multiple FlipOff commands (without intervening FlipOn commands) are invalid; FlipContinue without a preceding FlipOn is invalid.

1.13 Surface Probing [DevCTG]

1.13.1 MI_PROBE [DevCTG]

MI_PROBE

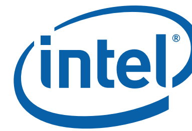
Project:	CTG+	**Length Bias:**	2

The probe command is inserted into a ring or batch buffer in order to validate the base address(es) of a surface(s) required by subsequent commands. When parsed, the probe command will do a "test" access of the surface base address to see if it is valid. The probe will also be written to the specified slot of a memory-based probe list such that it can be re-validated if the current context is switched out and then switched back in. If the test access encounters an invalid page table entry, it said to "fault". Faulting probes will trigger the current context to be switched.

A probe command containing multiple probes will process all of them regardless of which ones fault. If any probe faulted and the pipeline is busy, the next command (unless it is a probe or unprobe command) will stall until the pipeline drains. Once the pipeline is empty, the pending probes will be written to the probe list with the faulted probes ndicated and a context switch will occur.

Note that surfaces accessed through the global GTT need not be validated. It is assumed that Global GTT pages will not be invalidated while a context is switched out. Probe and unprobe are not privileged commands. The probe command can be used to insert only 512 probes in one command. Note that the total number of probes allowed in the system is 1024.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode



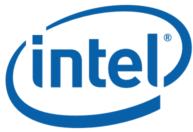
MI_PROBE		
	28:23	**MI Command Opcode** Default Value: 25h MI_PROBE Format: OpCode
	22:10	**Reserved** Project: All Format: MBZ
	9:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2
1..n	31:12	**Surface Page Base Address** Project: All Address: PerProcessGraphicsVirtualAddress[31:12] Surface Type: U32 Range 0..2^32-1 The Per Process Address to validate.
	11:10	**Reserved** Project: All Format: MBZ
	9:0	**Slot Number** Project: All Format: ProbeSlotIndex Range [0,1023] The index into the probe list where this probe will be stored.

1.13.2 MI_UNPROBE [DevCGT]

MI_UNPROBE			
Project:	CTG+	**Length Bias:**	1

There are 2 ways to remove probes. SW may issue a new probe to the same slot as an existing probe (presumably with a new surface base address), and the old probe will be replaced with the new, effectively deleting the old probe. If it has no new probe to place in the slot, SW may issue the unprobe command to remove probes by invaliding probe slots.

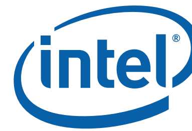
The unprobe command is used to remove probes from the probe list. No **Surface Address** is provided; the specified slot is simply marked invalid. The Unprobe command does not affect the probe list in memory; it only clears probe **Slot Valid** bits in the Probe List Slot Valid Registers (see *Memory Interface Registers*).



MI_UNPROBE		
DWord	**Bit**	**Description**
	31: 029	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 06h MI_UNPROBE Format: OpCode
	22:10	**Reserved** Project: All Format: MBZ
	9:0	**Slot Number** Project: All Format: ProbeSlotIndex Range [0,1023] The probe list index of the probe to be removed.

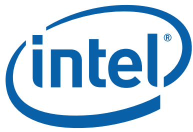
1.14 MI_REPORT_HEAD

MI_REPORT_HEAD		
Project: All		**Length Bias:** 1
The MI_REPORT_HEAD command causes the Head Pointer value of the active ring buffer to be written to a cacheable (snooped) system memory location. The location written is relative to the address programmed in the Hardware Status Page Address Register. **Programming Notes:** This command must not be executed from a Batch Buffer (Refer to the description of the HSW_PGA register).		
DWord	**Bit**	**Description**
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 07h MI_REPORT_HEAD Format: OpCode
	22:0	**Reserved** Project: All Format: MBZ



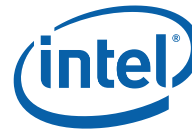
1.15 MI_SET_CONTEXT

<table>
<tr><th colspan="3">MI_SET_CONTEXT</th></tr>
<tr><td colspan="3">Project: All   Length Bias: 2</td></tr>
<tr><td colspan="3">The MI_SET_CONTEXT command is used to specify the logical context associated with the hardware context. A logical context is an area in memory used to store hardware context information, and the context is referenced via a 2KB-aligned pointer. If the (new) logical context is different (i.e., at a different memory address), the device will proceed to save the current HW context values to the current logical context address, and then restore (load) the new logical context by reading the context from the new address and loading it into the hardware context state. If the logical context address specified in this command matches the current logical context address, this command is effectively treated as a NOP.
This command also includes some controls over the context save/restore process.
• The Force Restore bit can be used to refresh the on-chip device state from the same memory address if the indirect state buffers have been modified.
• The Restore Inhibit bit can be used to prevent the new context from being loaded at all. This must be used to prevent an uninitialized context from being loaded. Once software has initialized a context (by setting all state variables to initial values via commands), the context can then be stored and restored normally.
• This command needs to be always followed by a single MI_NOOP instruction to workaround a GenX silicon issue.</td></tr>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>0</td><td>31:29</td><td>Command Type
Default Value: 0h MI_COMMAND Format: OpCode</td></tr>
<tr><td></td><td>28:23</td><td>MI Command Opcode
Default Value: 18h MI_SET_CONTEXT Format: OpCode</td></tr>
<tr><td></td><td>22:6</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>5:0</td><td>DWord Length
Default Value: 0h Excludes DWord (0,1)
Format: =n Total Length - 2</td></tr>
<tr><td>1</td><td>31:11</td><td>Logical Context Address
Project: All
Address: PhysicalAddress[31:11]
Surface Type: Logical Context
This field contains the 2KB-aligned physical address of the Logical Context that is <u>to be loaded</u> into the hardware context. If this address is equal to the CCID register associated with the current ring, no load will occur. Prior to loading this new context, the device will save the existing context as required. After the context switch operation completes, this address will be loaded into the associated CCID register.</td></tr>
</table>

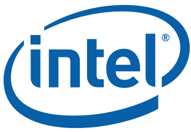


MI_SET_CONTEXT

	10	**Reserved** Project: All Format: MBZ
	9	**Reserved:** MBZ
	8	**Memory Space Select** Project: All BitFieldDesc **Value** | **Name** | **Description** | **Project** 0h | Physical Memory | Physical Main (unsnooped) Memory. The 4 bits of **Physical Start Address Extension** are prefixed to bits 31:11 to specify a 2KB aligned address within physical main memory. In this mode the hardware must not fetch data beyond a 4KB boundary. | All 1h | Global Graphics Memory | Global Graphics (GTT-mapped) Memory. Bits 31:11 of a graphics memory address. The GTT whose address is contained in the PGTBL_CTL register is used to translate this address. | All
	7:4	**Logical Context Address Extension** Project: All Address: PhysicalAddressExtension[35:32] Surface Type: Logical Context This field specified Bits 35:32 of the starting address of the 2KB-aligned physical logical context address. This field must be zero for global gtt context address.
	3	**Extended State Save Enable** Project: All Format: U32 If set, the extended state identified in the Logical Context Data section of the Memory Data Formats chapter is saved as part of switching away from this logical context. This bit will be stored in the associated CCID register to control the context save operation when switching away from this context (as part of a subsequent MI_SET_CONTEXT command).



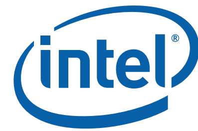
MI_SET_CONTEXT		
	2	**Extended State Restore Enable** Project: All Format: U32 If set, the extended state identified in the Logical Context Data section of the Memory Data Formats chapter is loaded (or restored) as part of switching to this logical context. This method can be used to restore things such as filter coefficients using the indirect state restore followed by a restore of the extended logical context data. This bit affects the switch (if required) to the context specified in **Logical Context Address**. This bit will also be stored in the associated CCID register to control a subsequent context save operation when switching to this context (as part of a subsequent ring buffer switch).
	1	**Force Restore** Project: All Format: U32 When switching to this logical context a comparison between Logical Context Address and the contests of the CCID register is performed. Normally, matching addresses prevent a context restore from occurring; however, when this bit is set a context restore is forced to occur. This bit cannot be set with Restore Inhibit. **Note:** This bit is not saved in the associated CCID register. It only affects the processing of this command.
	0	**Restore Inhibit** Project: All Format: U32 If set, the restore of the HW context from the logical context specified by **Logical Context Address** is inhibited (i.e., the existing HW context values are maintained). This bit must be used to prevent the loading of an uninitialized logical context. If clear, the context switch proceeds normally. This bit cannot be set with Force Restore. **Note:** This bit is not saved in the associated CCID register. It only affects the processing of this command.



1.16 MI_STORE_DATA_IMM

<table>
<tr><th colspan="3">MI_STORE_DATA_IMM</th></tr>
<tr><td colspan="3">Project: All   Length Bias: 2</td></tr>
<tr><td colspan="3">The MI_STORE_DATA_IMM command requests a write of the QWord constant supplied in the packet to the specified Memory Address. As the write targets a System Memory Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped).
Programming Notes:
• This command should not be used within a “non-secure” batch buffer to access per-process virtual space. Doing so will cause the command parser to perform the write with byte enables turned off. This command can be used within ring buffers and/or “secure” batch buffers.
• This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll un-cached memory or device registers).
• This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete “eventually”, there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations.</td></tr>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>0</td><td>31:29</td><td>Command Type
Default Value: 0h   MI_COMMAND   Format: OpCode</td></tr>
<tr><td></td><td>28:23</td><td>MI Command Opcode
Default Value: 20h   MI_STORE_DATA_IMM   Format: OpCode</td></tr>
<tr><td></td><td>22</td><td>Memory Address Type
Project: All

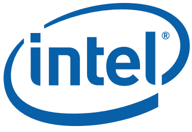
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Physical Address</td><td></td><td>All</td></tr><tr><td>1h</td><td>Graphics Address</td><td>Hardware will translate this address using the operating GTT. The GTT (global or per-process) used for the translation will be the same GTT used to access the buffer executing this command.</td><td>All</td></tr></table></td></tr>
<tr><td></td><td colspan="2"></td></tr>
</table>



<table>
<tr><th colspan="3">MI_STORE_DATA_IMM</th></tr>
<tr><td></td><td>21</td><td>BitFieldName
Project: All
This bit will be ignored and treated as if clear when executing from a non-privileged batch buffer. It is allowed for this bit to be clear when executing this command from a privileged (secure) batch buffer.

Value | Name | Description | Project
0h | Per Process Graphics Address | | All
1h | Global Graphics Address | This command will use the global GTT to translate the Address and this command must be executing from a privileged (secure) batch buffer. | All

Programming Notes
Notes</td></tr>
<tr><td></td><td>20:6</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>5:0</td><td>DWord Length
Default Value: 2h Excludes DWord (0,1) = 2 for DWord, 3 for QWord
Format: =n Total Length - 2</td></tr>
<tr><td>1</td><td>31:4</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td></td><td>3:0</td><td>Physical Start Address Extension
Project: All
Address: PhysicalAddressExtension[35:32]
Surface Type: U64
This field specifies bits 35:32 of the physical address where the data will be stored. This field must be zero for a virtual address.</td></tr>
</table>



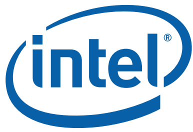
MI_STORE_DATA_IMM		
2	31:2	**Address** Project: All Address: SelectableAddress(Memory Address Type) [31:2] Surface Type: U32(2) This field specifies Bits 31:2 of the Address where the DWord will be stored. As the store address must be DWord-aligned, Bits 1:0 of that address MBZ. This address must be 8B aligned for a store "QW" command. Format = U30, Range = valid System Memory Address (not mapped by GTT) if Physical Format = Bits[31:2] of a Graphics Memory Address If Virtual
	1:0	**Reserved** Project: All Format: MBZ
3	31:0	**Data DWord 0** Project: All Format: U32 This field specifies the DWord value to be written to the targeted location. For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0).
4	31:0	**Data DWord 1** Project: All Format: U32 This field specifies the upper DWord value to be written to the targeted QWord location (DW 1).

1.17 MI_STORE_DATA_INDEX

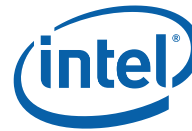
MI_STORE_DATA_INDEX			
Project:	All	**Length Bias:**	2

- The MI_STORE_DATA_INDEX command requests a write of the data constant supplied in the packet to the specified offset from the System Address defined by the Hardware Status Page Address Register. As the write targets a System Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped).
- Programming Notes:
- Use of this command with an invalid or uninitialized value in the Hardware Status Page Address Register is UNDEFINED.
- This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll uncached memory or device registers).
- This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete “eventually”, there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 21h MI_STORE_DATA_INDEX Format: OpCode
	22:21	**Reserved** Project: All Format: MBZ
	20:6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 1h Excludes DWord (0,1) = 1 for DWord, 2 for QWord Format: =n Total Length - 2



MI_STORE_DATA_INDEX		
1	31:12	**Reserved** Project: All Format: MBZ
	11:2	**Offset** Project: All Format: U10 zero-based DWord offset into the HW status page. Address: HardwareStatusPageOffset[11:2] Surface Type: U32 Range [16, 1023] This field specifies the offset (into the hardware status page) to which the data will be written. Note that the first few DWords of this status page are reserved for special-purpose data storage – targeting these reserved locations via this command is UNDEFINED.
	1:0	**Reserved** Project: All Format: MBZ
2	31:0	**Data DWord 0** Project: All Format: U32 This field specifies the DWord value to be written to the targeted location. For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0).
3	31:0	**Data DWord 1** Project: All Format: U32 This field specifies the upper DWord value to be written to the targeted QWord location (DW 1).



1.18 MI_STORE_REGISTER_MEM

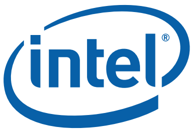
MI_STORE_REGISTER_MEM			
Project:	All	**Length Bias:**	2

The MI_STORE_REGISTER_MEM command requests a register read from a specified memory mapped register location in the device and store of that DWord to memory. The register address is specified along with the command to perform the read.

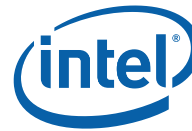
Programming Notes:

- The command temporarily halts command execution.
- The memory address for the write is snooped on the host bus.
- This command should not be used within a "non-secure" batch buffer to access per-process virtual space. Doing so will cause the command parser to perform the write with byte enables turned off. This command can be used within ring buffers and/or "secure" batch buffers.
- This command will cause undefined data to be written to memory if given register addresses for the PGTBL_CTL_0 or FENCE registers

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 24h MI_STORE_REGISTER_MEM Format: OpCode
	22	**Reserved** Project: DevBW-A,B Format: MBZ
	22	**Memory Address Type** Project: All, except DevBW-A,B **Value** / **Name** / **Description** / **Project** 0h / Physical Address / / All 1h / Graphics Address / Hardware will translate this address using the operating GTT. The GTT (global or per-process) used for the translation will be the same GTT used to access the buffer executing this command. / All
	21:6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2



MI_STORE_REGISTER_MEM		
1	31:28	**Physical Start Address Extension** Project: All Address: PhysicalAddressExtension[35:32] Surface Type: MMIO Register This field specifies bits 35:32 of the starting address of the physical address.
	27:19	**Reserved** Project: All Format: MBZ
	18:1	**Register Address** Project: All Address: MMIO Address[18:2] Surface Type: MMIO Register This field specifies Bits 18:2 of the Register offset the DWord will be read from. As the register address must be DWord-aligned, Bits 1:0 of that address MBZ. **Programming Notes** — **Project** Storing a VGA register is not permitted and will store an UNDEFINED value. — All The values of PGTBL_CTL0 or any of the FENCE registers cannot be stored to memory; UNDEFINED values will be written to memory if the addresses of these registers are specified. — All
	1	**Reserved** Project: All Format: MBZ
	0	**Reserved** Project: All Format: MBZ
2	31:2	**Memory Address** Project: All Address: SelectableAddress(Memory Address Type)[31:2] Surface Type: MMIO Register This field specifies the address of the memory location where the register value specified in the DWord above will be written. The address specifies the DWord location of the data. If Memory Address Type = 0, Range = Physical_Address [31:2] If Memory Address Type = 1, Range = GraphicsMemoryAddress[31:2]
	1:0	**Reserved** Project: All Format: MBZ



1.19 MI_UPDATE_GTT ([DevCTG])

MI_UPDATE_GTT

Project:	DevCTG+	**Length Bias:**	2

[DevBW] and [DevCL]: This is not a legal command.

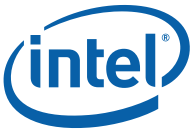
The MI_UPDATE_GTT command is used to update GTT page table entries in a coherent manner and at a predictable place in the command flow.

On [DevCTG] this command can be used to update PPGTT page table entries, but only for the currently executing context. It cannot be used when servicing a (PPGTT) page fault since the command parser cannot be relied upon to parse and complete the command until the fault is cleared.

An MI_FLUSH should be placed before this command, since work associated with preceding commands that are still in the pipeline may be referencing GTT entries that will be changed by its execution. The flush will also invalidate TLBs and read caches that may become invalid as a result of the changed GTT entries. MI_FLUSH is not required if it can be guaranteed that the pipeline is free of any work that relies on changing GTT entries (such as MI_UPDATE_GTT contained in a paging DMA buffer that is doing only update/mapping activities and no rendering).

This is a privileged command. This command will be converted to a no-op and an error flagged if it is executed from within a non-secure batch buffer.

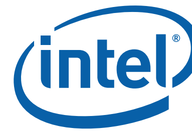
DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 23h MI_UPDATE_GTT Format: OpCode
	22	**Reserved** Project: Pre-DevCTG Format: MBZ
	22	**Entry Type** Project: CTG+ Format: Graphics Space Select Select whether to update Global GTT or Per-Process GTT **Value** / **Name** / **Description** / **Project** 0h / GGTT / GGTT Page Table Entry Update / All 1h / PPGTT / PPGTT Page Table Entry Update / All **Programming Notes** / **Project** When the **Per-Process Virtual Address Space and Context queuing Enable** bit in GFX_MODE is clear, the only valid value for this field is 0, indicating a GGTT page table entry update. Setting this bit results in UNDEFINED behavior. / All



MI_UPDATE_GTT		
	21:6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2
1	31:12	**Entry Address** Project: All Address: GraphicsAddress[31:12] For PPGTT updates ([DevCTG] only), bits [31:22] specify the directory entry pointing to the page table to be modified. Bits [21:12] specify the (first) page table entry to be updated. A single MI_UPDATE_GTT command may not modify entries in more than one page table (i.e., the Entry Address and the last address to be modified as calculated by adding the entry address and the Dword Length must be on the same 4K page.) For GGTT updates, this field simply holds the DW offset of the first table entry to be modified. Note that one or more of the upper bits may need to be 0, i.e., for a 2G aperture, bit 31 MBZ.
	11:0	**Reserved** Project: All Format: MBZ
2..n	31:0	**Entry Data** Project: All Format: PPGTT Table Entry This Dword becomes the new page table entry. See PPGTT Table Entries (PTEs) in Memory Interface Registers.

1.20 MI_USER_INTERRUPT

MI_USER_INTERRUPT		
Project: All	**Length Bias:** 1	
The MI_USER_INTERRUPT command is used to generate a User Interrupt condition. The parser will continue parsing after processing this command. See User Interrupt.		
DWord	**Bit**	**Description**
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 02h MI_USER_INTERRUPT Format: OpCode
	22:0	**Reserved** Project: All Format: MBZ



1.21 MI_WAIT_FOR_EVENT

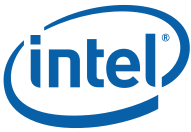
MI_WAIT_FOR_EVENT			
Project:	All	**Length Bias:**	1

The MI_WAIT_FOR_EVENT command is used to pause command stream processing until a specific event occurs or while a specific condition exists. See Wait Events/Conditions, Device Programming Interface in *MI Functions*. Only one event/condition can be specified -- specifying multiple events is UNDEFINED.

The effect of the wait operation depends on the source of the command. If executed from a batch buffer, the parser will halt (and suspend command arbitration) until the event/condition occurs. If executed from a ring buffer, further processing of that ring will be suspended, although command arbitration (from other rings) will continue. Note that if a specified condition does not exist (the condition code is inactive) at the time the parser executes this command, the parser proceeds, treating this command as a no-operation.

If execution of this command from a primary ring buffer causes a wait to occur, the active ring buffer will *effectively* give up the remainder of its time slice (required in order to enable arbitration from other primary ring buffers).

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 03h MI_WAIT_FOR_EVENT Format: OpCode
	22:19	**Reserved** Project: All Format: MBZ
	18	**Display Pipe B Start of V Blank Wait Enable** Project: All Format: Enable This field enables a wait until the start of next Display Pipe B "Vertical Blank" event occurs. This event is defined as the start of the next Display B Vertical blank period. Note that this can cause a wait for up to a frame. See Start of Vertical Blank Event in the Device Programming Interface chapter of *MI Functions*. **Errata** / **Description** / **Project** BWT013 / MBZ / DevBW



MI_WAIT_FOR_EVENT

17 — Display Pipe A Start of V Blank Wait Enable | Project: All | Format: Enable

This field enables a wait until the start of next Display Pipe A "Vertical Blank" event occurs. This event is defined as the start of the next Display A Vertical blank period. Note that this can cause a wait for up to a frame. See Start of Vertical Blank Event in the Device Programming Interface chapter of *MI Functions*.

Programming Notes	Project
Notes	All

Errata	Description	Project
BWT013	MBZ	DevBW

16 — Overlay Flip Pending Wait Enable | Project: All | Format: Enable

This field enables a wait for the duration of an Overlay "Flip Pending" condition. If a flip request is pending, the parser will wait until the flip operation has completed (i.e., the new overlay address has been loaded into the corresponding overlay registers). See Overlay Flip Pending Condition in the Device Programming Interface chapter of *MI Functions*.

16 — isplay Sprite B Flip Pending Wait Enable | Project: CTG+ | ormat: nable

is field enables a wait for the duration of a Display Sprite B "Flip Pending" condition. If a flip request is pending, the parser will wait until the flip operation has completed (i.e., the new front buffer address has now been loaded into the active front buffer registers). See Display Flip Pending Condition in the Device Programming Interface chapter of *MI Functions*.

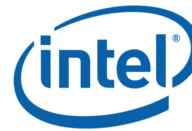
15 — Reserved | Project: All | Format: MBZ

14 — Display Pipe B H Blank Wait Enable | Project: All | Format: Enable

This field enables a wait until the start of next Display Pipe B "Horizontal Blank" event occurs. This event is defined as the start of the next Display B Horizontal blank period. Note that this can cause a wait for up to a line. See Horizontal Blank Event in the Device Programming Interface chapter of *MI Functions*.

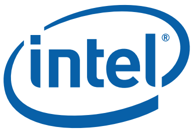
13 — Display Pipe A H Blank Wait Enable | Project: All | Format: Enable

This field enables a wait until the start of next Display Pipe A "Horizontal Blank" event occurs. This event is defined as the start of the next Display A Horizontal blank period. Note that this can cause a wait for up to a line. See Horizontal Blank Event in the Device Programming Interface chapter of *MI Functions*.

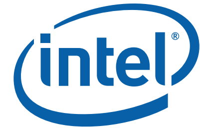


MI_WAIT_FOR_EVENT

Bits	Description
12:9	**Condition Code Wait Select** Project: All This field enables a wait for the duration that the corresponding condition code is active. These enable select one of 15 condition codes in the EXCC register, that cause the parser to wait until that condition-code in the EXCC is cleared. **Value** | **Name** | **Description** | **Project** 0h | Not Enabled | Condition Code Wait not enabled | All 1h-5h | Enabled | Condition Code select enabled; selects one of 5 codes, 0 – 4 | All 6h-15h | Reserved | | All **Programming Notes** — **Project** Note that not all condition codes are implemented. The parser operation is UNDEFINED if an unimplemented condition code is selected by this field. The description of the EXCC register (*Memory Interface Registers*) lists the codes that are implemented. — All
8	**Display Plane C Flip Pending Wait Enable** Project: All Format: Enable This field enables a wait for the duration of a Display Plane C “Flip Pending” condition. If a flip request is pending, the parser will wait until the flip operation has completed (i.e., the new front buffer address has now been loaded into the active front buffer registers). See Display Flip Pending Condition in the Device Programming Interface chapter of *MI Functions*.
7	**Display Pipe B Vertical Blank Wait Enable** Project: All Format: Enable This field enables a wait until the next Display Pipe B “Vertical Blank” event occurs. This event is defined as the start of the next Display Pipe B vertical blank period. Note that this can cause a wait for up to an entire refresh period. See Vertical Blank Event (See *Programming Interface*). **Programming Notes** — **Project** Prior to using the MI_WAIT_FOR_EVENT command to wait on Display Pipe A/B VBlank events, the corresponding Vertical Blank Interrupt Enable (bit 17) of the corresponding PIPEASTAT (70024h) or PIPEBSTAT (71024h) register must be set. Note that this does not require an actual VBlank interrupt to be enabled. — All



MI_WAIT_FOR_EVENT		
	6	**Display Plane B Flip Pending Wait Enable** Project: All Format: Enable This field enables a wait for the duration of a Display Plane B "Flip Pending" condition. If a flip request is pending, the parser will wait until the flip operation has completed (i.e., the new front buffer address has now been loaded into the active front buffer registers). See Display Flip Pending Condition (in the Device Programming Interface chapter of *MI Functions*.
	5	**Display Pipe B Scan Line Window Wait Enable** Project: All Format: Enable This field enables a wait while a Display B "In Scan Line Window" condition exists. This condition is defined as the period of time the Display B refresh is inside the scan line window as specified by a previous MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL command. If the Display B refresh is outside this window, or a window has not been specified, the parser proceeds, treating this command as a no-op. If the Display B refresh is currently inside this window, the parser will wait until the refresh exits the window. See Scan Line Window Condition in the Device Programming Interface chapter of *MI Functions*.
	4	**Frame Buffer Compression Idle Wait Enable** Project: All Format: Enable This field enables a wait while the Frame Buffer compressor is busy. The ring that this command got executed from is removed from arbitration for the wait period and is inserted into arbitration as soon as the frame buffer compressor is idle.
	3	**Display Pipe A Vertical Blank Wait Enable** Project: All Format: Enable This field enables a wait until the next Display Pipe A "Vertical Blank" event occurs. This event is defined as the start of the next Display A vertical blank period. Note that this can cause a wait for up to an entire refresh period. See Vertical Blank Event in the Device Programming Interface chapter of *MI Functions*. **Programming Notes** / **Project** Prior to using the MI_WAIT_FOR_EVENT command to wait on Display Pipe A/B VBlank events, the corresponding Vertical Blank Interrupt Enable (bit 17) of the corresponding PIPEASTAT (70024h) or PIPEBSTAT (71024h) register must be set. Note that this does not require an actual VBlank interrupt to be enabled. / All
	2	**Display Plane A Flip Pending Wait Enable** Project: All Format: Enable This field enables a wait for the duration of a Display Plane A "Flip Pending" condition. If a flip request is pending, the parser will wait until the flip operation has completed (i.e., the new front buffer address has now been loaded into the active front buffer registers). See Display Flip Pending Condition in the Device Programming Interface chapter of *MI Functions*.



MI_WAIT_FOR_EVENT							
	1	**Display Pipe A Scan Line Window Wait Enable**	Project:	All	Format:	Enable	
		This field enables a wait while a Display Pipe A "In Scan Line Window" condition exists. This condition is defined as the period of time the Display A refresh is inside the scan line window as specified by a previous MI_INCLUSIVE_SCAN_WINDOW or MI_EXCLUSIVE_SCAN_WINDOW command. If the Display A refresh is outside this window, or a window has not been specified, the parser proceeds, treating this command as a no-op. If the Display A refresh is currently inside this window, the parser will wait until the refresh exits the window. See Scan Line Window Condition in the Device Programming Interface chapter of *MI Functions*.					
	0	**Reserved**	Project:	All	Format:	MBZ	

§§

2 Memory Interface Commands for Video Codec Engine [DevCTG+]

2.1 Introduction

This chapter describes the formats of the "Memory Interface" commands, including brief descriptions of their use. The functions performed by these commands are discussed fully in the *Memory Interface Functions* Device Programming Environment chapter.

This chapter describes MI Commands for the Video Codec Engine. Note that these commands are not applicable to [DevBW] and [DevCL] (these devices do not have a parallel Video Codec Engine).

The commands detailed in this chapter are used across the later products within the GenX family. However, slight changes may be present in some commands (i.e., for features added or removed), or some commands may be removed entirely. Refer to the *Preface* chapter for details.

2.2 MI_ARB_CHECK

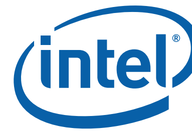
The MI_ARB_CHECK instruction is used to check the ring buffer next context ID register (RNCID) or the UHPTR register, depending on whether PPGTT/Runlists are enabled. This instruction can be used to pre-empt the current execution of the ring buffer. Note that the valid bit in the RNCID register or the UHPTR register needs to be set for the command streamer to be pre-empted.

Programming Note:

- This instruction can be placed only in a ring buffer, never in a batch buffer.

The instruction format is:

DWord	Bits	Description
0	31:29	**Instruction Type =** MI_INSTRUCTION = 0h
	28:23	**MI Instruction Opcode =** MI_ARB_CHECK = 05h
	22:0	Reserved: MBZ



2.3 MI_BATCH_BUFFER_END

The MI_BATCH_BUFFER_END command is used to terminate the execution of commands stored in a *batch buffer* initiated using a MI_BATCH_BUFFER_START command.

The MI_BATCH_BUFFER_END command format follows:

DWord	Bits	Description
0	31:29	**Command Type =** MI_COMMAND = 0h
	28:23	**MI Command Opcode =** MI_BATCH_BUFFER_END = 0Ah
	22:0	Reserved: MBZ

2.4 MI_BATCH_BUFFER_START

The MI_BATCH_BUFFER_START command is used to initiate the execution of commands stored in a *batch buffer.* For restrictions on the location of batch buffers, see Batch Buffers in the Device Programming Interface chapter of *MI Functions*.

The batch buffer can be specified as secure or non-secure, determining the operations considered valid when initiated from within the buffer and any attached (chained) batch buffers. See Batch Buffer Protection in the Device Programming Interface chapter of *MI Functions*.

The MI_BATCH_BUFFER_START command format follows:

DWord	Bits	Description
0	31:29	**Command Type =** MI_COMMAND = 0h
	28:23	**MI Command Opcode =** MI_BATCH_BUFFER_START = 31h
	22:9	Reserved: MBZ
	8	**Buffer Security Indicator:** When this command is executed directly from a ring buffer, this field is used to specify the associated batch buffer as a *secure* or *non-secure* buffer. Certain operations (e.g., MI_STORE_DATA_IMM commands) are prohibited within non-secure buffers. See Batch Buffer Protection in the Device Programming Interface chapter of *MI Functions*. When this command is executed from within a batch buffer (i.e., is a "chained" batch buffer command), this field is IGNORED and the next buffer in the chain inherits the initial buffer's security characteristics. If this bit is set, this batch buffer is non-secure and cannot execute privileged commands nor access privileged (GGTT) memory. It will be accessed via the PPGTT. If clear, this batch buffer is secure and will be accessed via the GGTT. Note that MI_STORE_DATA_IMM to non-privileged memory (via the PPGTT) *is* allowed in a non-secure batch buffer. Format = MI_BufferSecurityType 1 = MIBUFFER_NONSECURE 0 = MIBUFFER_SECURE

DWord	Bits	Description
	7:6	Reserved: MBZ
	0	**DWord Length** (Excludes D-Word 0,1) = 0
1	31:2	**Buffer Start Address**: This field specifies Bits 31:2 of the starting address of the 64B aligned batch buffer (Bits 1:0 of that address MBZ). **Programming Notes:** • A batch buffer initiated with this command must end either with a MI_BATCH_BUFFER_END command or by chaining to another batch buffer with an MI_BATCH_BUFFER_START command. • **[DevCTG]:** the selection of PPGTT vs. GGTT for the batch buffer is determined by the **Buffer Security Indicator** (bit 8). Format = Graphics Virtual Address[31:2]
	1:0	Reserved: MBZ

(MI_DISPLAY_FLIP DELETED)

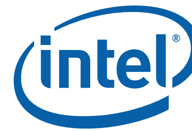
2.5 MI_FLUSH

Project:	All	**Length Bias:**	1

The MI_FLUSH command is used to perform an internal "flush" operation. The parser pauses on an internal flush until all media decode engines have completed any pending operations and any read caches are invalidated.

Usage note: After this command is completed and followed by a Store DWord-type command, CPU access to graphics memory will be coherent.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 04h MI_FLUSH Format: OpCode
	22:9	**Reserved** Project: All Format: MBZ
	8:7	**Reserved** Project: Pre-DevCTG Format: MBZ
	6	**Reserved** Project: Pre-DevCTG Format: MBZ
	5:0	**Reserved** Project: All Format: MBZ



2.6 MI_LOAD_REGISTER_IMM

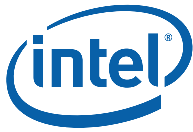
The MI_LOAD_REGISTER_IMM command requests a write of up to a DWord constant supplied in the command to the specified Register Offset (i.e., offset into Memory-Mapped Register Range). The register is loaded before the next command is executed.

Programming Notes:

- The behavior of this command is controlled by Dword 3, Bit 8 (**Disable Register Access**) of the RINGBUF register. If this command is disallowed then the command stream converts it to a NOOP.
- If this command is executed from a batch buffer then the behavior of this command is controlled by Dword 0, Bit 8 (**Security Indicator**) of the BATCH_BUFFER_START Command. If the batch buffer is non-secure then the command stream converts this command to a NOOP.

The MI_LOAD_REGISTER_IMM command format is:

DWord	Bit	Description
0	31:29	**Command Type =** MI_COMMAND = 0h
	28:23	**MI Command Opcode =** MI_LOAD_REGISTER_IMM = 22h
	22:12	Reserved: MBZ
	11:8	**Byte Write Disables**: This field specifies which bytes of the **Data DWord** are **not** to be written to the DWord offset specified in *Register Offset*. Format = Enable[4] (bit 8 corresponds to Data DWord [7:0]). Range = Must specify a valid register write operation.
	7:6	Reserved: MBZ
	5:0	**DWord Length** (Excludes DWord 0,1) = 1.
1	31:23	Reserved: MBZ
	22:2	**Register Offset**: This field specifies bits [22:2] of the offset into the Memory Mapped Register Range (i.e., this field specifies a DWord offset). Format = U30.
	1:0	Reserved: MBZ
2	31:0	**Data DWord.:** This field specifies the DWord value to be written to the targeted location. Format = U32.

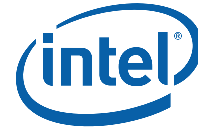


2.7 MI_NOOP

The MI_NOOP command basically performs a “no operation” in the command stream and is typically used to pad the command stream (e.g., in order to pad out a batch buffer to a QWord boundary). However, there is one minor (optional) function this command can perform – a 22-bit value can be loaded into the MI NOPID register. This provides a general-purpose command stream tagging ("breadcrumb") mechanism (e.g., to provide sequencing information for a subsequent breakpoint interrupt).

The MI_NOOP command format is:

DWord	Bit	Description
0	31:29	**Command Type =** MI_COMMAND = 0h
	28:23	**MI Command Opcode =** MI_NOOP = 00h
	22	**Identification Number Register Write Enable:** This field enables the value in the Identification Number field to be written into the MI NOPID register. If disabled, that register is unmodified – making this command an effective “no operation” function. Format = Enable. 1 = Write the NOP_ID register. 0 = Do not write the NOP_ID register.
	21:0	**Identification Number:** This field contains a 22-bit number which can be written to the MI NOPID register. Format = U22.



2.8 MI_REPORT_HEAD

The MI_REPORT_HEAD command causes the Head Pointer value of the ring buffer to be written to a cacheable (snooped) system memory location.

[DevBW], [DevCL]:

[DevCTG] when the **Per-Process Virtual Address Space and Run List Enable** bit is reset:

The location written is relative to the address programmed in the Hardware Status Page Address Register.

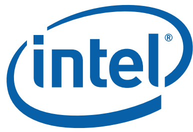
Programming Notes:

- This command must not be executed from a Batch Buffer (Refer to the description of the HWS_PGA register).

[DevCTG]: When the **Per-Process Virtual Address Space and Run List Enable** is set, the head pointer will be reported to the PP HW Status Page.

The format of the MI_REPORT_HEAD command is:

DWord	Bit	Description
0	31:29	**Command Type =** MI_COMMAND = 0h
	28:23	**MI Command Opcode =** MI_REPORT_HEAD = 07h
	22:0	Reserved: MBZ



2.9 MI_STORE_DATA_IMM

The MI_STORE_DATA_IMM command requests a write of the QWord or DWord constant supplied in the packet to the specified Memory Address. As the write targets a System Memory Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped).

Programming Notes:

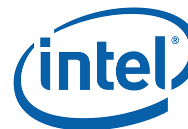
This command should not be used within a "non-secure" batch buffer except on [DevCTG] to access per-process virtual space. Doing so will cause the command parser to perform the write with byte enables turned off. This command can be used within ring buffers and/or "secure" batch buffers. If used within a non-secure batch buffer on [DevCTG], **Use Global GTT** must be clear.

This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll un-cached memory or device registers).

This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete "eventually", there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations.

The MI_STORE_DATA_IMM command format is:

DWord	Bit	Description
0	31:29	**Command Type =** MI_COMMAND = 0h
	28:23	**MI Command Opcode =** MI_STORE_DATA_IMM = 20h
	22	**[DevCTG] Use Global GTT.** If set, this command will use the global GTT to translate the Address and this command must be executing from a privileged (secure) batch buffer. If clear, the PPGTT will be used. This bit will be ignored and treated as if clear when executing from a non-privileged batch buffer. It is allowed for this bit to be clear when executing this command from a privileged (secure) batch buffer.
	22:6	Reserved: MBZ
	5:0	**DWord Length** (Excludes DWord 0,1) = 3 for QWord, 2 for DWord
1	31:0	**Reserved: MBZ**
2	31:2	**Address**: This field specifies Bits 31:2 of the Address where the DWord will be stored. As the store address must be DWord-aligned, Bits 1:0 of that address MBZ. This address must be 8B aligned for a store "QW" command. Format = Bits[31:2] of a Graphics Virtual Address
	1:0	Reserved: MBZ



DWord	Bit	Description
3	31:0	**Data DWord 0:** This field specifies the DWord value to be written to the targeted location. For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0). Format = U32
4	**31:0**	**Data Word 1:** This field specifies the upper DWord value to be written to the targeted QWord location (DW 1). Format = U32

2.10 MI_STORE_DATA_INDEX

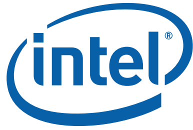
The MI_STORE_DATA_INDEX command requests a write of the data constant supplied in the packet to the specified offset from the System Address defined by the Hardware Status Page Address Register. As the write targets a System Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped).

Programming Notes:

- Use of this command with an invalid or uninitialized value in the Hardware Status Page Address Register is UNDEFINED.
- This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll uncached memory or device registers).
- This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete "eventually", there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations.

The MI_STORE_DATA_INDEX command format is:

DWord	Bit	Description
0	31:29	**Command Type =** MI_COMMAND = 0h
	28:23	**MI Command Opcode =** MI_STORE_DATA_INDEX = 21h
	22	Reserved: MBZ
	21	[DevCTG] Only: **Use Per-Process Hardware Status Page.** If this bit is set, this command will index into the per-process hardware status page at offset 20K from the LRCA. If clear, the Global Hardware Status Page will be indexed. This bit will be ignored and treated as set if this command is executed from within a non-secure batch buffer, or if the **Per-Process Virtual Address Space and context queuing Enable** bit is reset. All other devices: Reserved: MBZ.
	20:6	Reserved: MBZ
	5:0	**DWord Length** (Excludes DWord 0,1) = 2 for QWord
1	31:12	Reserved: MBZ

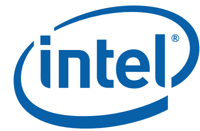


	11:2	**Offset:** This field specifies the offset (into the hardware status page) to which the data will be written. Note that the first few DWords of this status page are reserved for special-purpose data storage – targeting these reserved locations via this command is UNDEFINED. For a QWord write, the offset is valid down to bit 3 only. Format = U10 zero-based DWord offset into the HW status page. Range = [16, 1023].
	1:0	Reserved: MBZ
2	31:0	**Data DWord 0:** This field specifies the DWord value to be written to the targeted location. [For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0). Format = U32
3	31:0	**Data Word 1:** This field specifies the upper DWord value to be written to the targeted QWord location (DW 1). Format = U32

2.11 MI_USER_INTERRUPT

The MI_USER_INTERRUPT command is used to generate a User Interrupt condition. The parser will continue parsing after processing this command. See User Interrupt.

DWord	Bit	Description
0	31:29	**Command Type =** MI_COMMAND = 0h
	28:23	**MI Command Opcode =** MI_USER_INTERRUPT = 02h
	22:0	Reserved: MBZ



2.12 MI_WAIT_FOR_EVENT

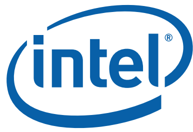
MI_WAIT_FOR_EVENT			
Project:	All	**Length Bias:**	1

The MI_WAIT_FOR_EVENT command is used to pause command stream processing until a specific event occurs or while a specific condition exists. See Wait Events/Conditions, Device Programming Interface in *MI Functions*. Only one event/condition can be specified -- specifying multiple events is UNDEFINED.

The effect of the wait operation depends on the source of the command. If executed from a batch buffer, the parser will halt (and suspend command arbitration) until the event/condition occurs. If executed from a ring buffer, further processing of that ring will be suspended, although command arbitration (from other rings) will continue. Note that if a specified condition does not exist (the condition code is inactive) at the time the parser executes this command, the parser proceeds, treating this command as a no-operation.

If execution of this command from a primary ring buffer causes a wait to occur, the active ring buffer will *effectively* give up the remainder of its time slice (required in order to enable arbitration from other primary ring buffers).

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 03h MI_WAIT_FOR_EVENT Format: OpCode
	22:2	**Reserved** Project: All Format: MBZ
	1	**Reserved**
	0	**Reserved**



2.13 Summary of Commands

Starting with [DevCTG], GenX products will have a 2nd command streamer (CS) dedicated to Video Codec Engine (VCE). This command streamer is a subset of the primary CS. The MI Commands that it can parse and decode are listed in the table below. Any other MI command or any command intended for the primary graphics pipeline will cause a parse error. The VCE CS can parse commands specific to the VCE fixed function units themselves. See the MFX chapter for the list of commands.

Command	Valid in Stream
MI_ARB_CHECK	**Both**
MI_ARB_ON_OFF [DevCTG] Only	**Primary CS Only**
MI_BATCH_BUFFER_END	**Both**
MI_BATCH_BUFFER_START	**Both**
MI_FLUSH	**Both**
MI_LOAD_REGISTER_IMM	**Both**
MI_NOOP	**Both**
MI_REPORT_HEAD	**Both**
MI_SEMAPHORE_MBOX Only	**Both**
MI_STORE_REGISTER_MEM	**Primary CS only for DevCTG**
MI_STORE_DATA_IMM	**Both**
MI_STORE_DATA_INDEX	**Both**
MI_USER_INTERRUPT	**Both**

3 Memory Interface Commands for Blitter Engine

3.1 Introduction

This chapter describes the formats of the "Memory Interface" commands, including brief descriptions of their use. The functions performed by these commands are discussed fully in the *Memory Interface Functions* Device Programming Environment chapter.

This chapter describes MI Commands for the blitter graphics processing engine. The term "for Blitter Engine" in the title has been added to differentiate this chapter from a similar one describing the MI commands for the Media Decode Engine and the Rendering Engine.

The commands detailed in this chapter are used across products within the GenX family. However, slight changes may be present in some commands (i.e., for features added or removed), or some commands may be removed entirely. Refer to the *Preface* chapter for product specific summary.

3.2 MI_LOAD_REGISTER_IMM

MI_LOAD_REGISTER_IMM			
Project:	All	**Length Bias:**	2

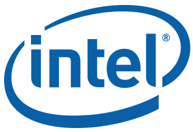
The MI_LOAD_REGISTER_IMM command requests a write of up to a DWord constant supplied in the command to the specified Register Offset (i.e., offset into Memory-Mapped Register Range). The register is loaded before the next command is executed.

Programming Notes:

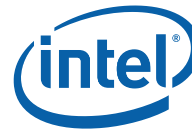
The behavior of this command is controlled by Dword 3, Bit 8 (**Disable Register Access**) of the RINGBUF register. If this command is disallowed then the command stream converts it to a NOOP.

If this command is executed from a BB then the behavior of this command is controlled by Dword 0, Bit 8 (Security Indicator**) of the BATCH_BUFFER_START Command. If the batch buffer is insecure then the command stream converts this command to a NOOP. Note that the corresponding ring buffer must allow a register update for this command to execute.**

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: h MI_ Format: OpCode
	22:12	**Reserved** Project: All Format: MBZ



MI_LOAD_REGISTER_IMM		
	11:8	**Byte Write Disables** Format: Enable[4] Bit 8 corresponds to Data DWord [7:0] Range Must specify a valid register write operation This field specifies which bytes of the **Data DWord** are **not** to be written to the DWord offset specified in *Register Offset.*
	7:6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2
1	31:2	**Register Offset** Format: U30 Address: MmioAddress[31:2] This field specifies bits [31:2] of the offset into the Memory Mapped Register Range (i.e., this field specifies a DWord offset).
	1:0	**Reserved** Project: All Format: MBZ
2	31:0	**Data DWord** Mask: Bytes Write Disables Format: U32 This field specifies the DWord value to be written to the targeted location.

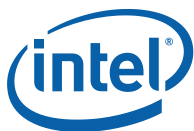


3.3 MI_NOOP

MI_NOOP			
Project:	All	**Length Bias:**	1

The MI_NOOP command basically performs a "no operation" in the command stream and is typically used to pad the command stream (e.g., in order to pad out a batch buffer to a QWord boundary). However, there is one minor (optional) function this command can perform – a 22-bit value can be loaded into the MI NOPID register. This provides a general-purpose command stream tagging ("breadcrumb") mechanism (e.g., to provide sequencing information for a subsequent breakpoint interrupt).

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 0h MI_NOOP Format: OpCode
	22	**Identification Number Register Write Enable** Project: All Format: Enable This field enables the value in the Identification Number field to be written into the MI NOPID register. If disabled, that register is unmodified – making this command an effective "no operation" function. **Value** / **Name** / **Description** / **Project** 0h / Disable / Do not write the NOP_ID register. / All 1h / Enable / Write the NOP_ID register. / All
	31:0	**Identification Number** Project: All Format: U22 This field contains a 22-bit number which can be written to the MI NOPID register.



3.4 MI_ SEMAPHORE_MBOX

MI_SEMAPHORE_MBOX

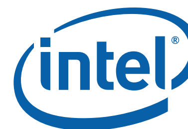
Project:	CTG+	**Length Bias:**	2

This command is provided as alternative to MI_SEMAPHORE to provide mailbox-type semaphores where there is no update of the semaphore by the checking process (the consumer). Single-bit compare-and-update semantics are also provided. In either case, atomic access of semaphores need not be guaranteed by hardware as with the previous command. This command should eventually supersede the previous command.

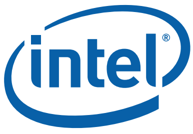
Synchronization between contexts (especially between contexts running on 2 different engines) is provided by the MI_SEMAPHORE_MBOX command. Note that contexts attempting to synchronize in this fashion must be able to access a common memory location. This means the contexts must share the same virtual address space (have the same page directory), must have a common physical page mapped into both of their respective address spaces or the semaphore commands must be executing from a secure batch buffer or directly from a ring with the **Use Global GTT** bit set such that they are "privileged" and will use the (always shared) global GTT.

MI_SEMAPHORE with the **Update Semaphore** bit set (and the **Compare Semaphore** bit clear) implements the *Signal* command, while the *Wait* command is indicated by **Compare Semaphore** being set. Note that *Wait* can cause a context switch. *Signal* increments unconditionally.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 16h MI_SEMAPHORE_MBOX Format: OpCode
	22	**Use Global GTT** Project: All Format: U32 If set, this command will use the global GTT to translate the **Semaphore Address** and this command must be executing from a privileged (secure) batch buffer. If clear, the PPGTT will be used to translate the **Semaphore Address**. This bit will be ignored (and treated as if clear) if this command is executed from a non-privileged batch buffer. It is allowed for this bit to be clear when executing this command from a privileged (secure) batch buffer or directly from a ring buffer.
	21	**Update Semaphore** Project: All Format: U32 If set, the value from the **Semaphore Data Dword** is written to memory. If **Compare Semaphore** is also set, the semaphore is not updated if the semaphore comparison fails. If clear, the data at **Semaphore Address** is not changed.



MI_SEMAPHORE_MBOX		
	20	**Compare Semaphore** Project: All Format: U32 If set, the value from the **Semaphore Data Dword** is compared to the value from the **Semaphore Address** in memory. If the value at **Semaphore Address is greater than or equal to the Semaphore Data Dword**, execution is continued from the current command buffer. If clear, no comparison takes place. **Update Semaphore** *must* be set in this case.
	19:6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2
1	31:0	**Semaphore Data Dword** Project: All Format: U32 Data dword to compare/update memory. The Data dword is supplied by software to control execution of the command buffer. If the compare is enabled and the data at **Semaphore Address** is greater than this dword, the execution of the command buffer continues. If **Update Semaphore** is set, the Data dword is constrained to be either 0 or 1. If both the compare and the update fields are set, the Data dword is constrained to be a 0.
1	31:2	**PointerBitFieldName** Project: All Address: GraphicsVirtualAddress[31:2] Surface Type: Semaphore Graphics Memory Address of the 32 bit value for the semaphore.
	1:0	**Reserved** Project: All Format: MBZ



3.5 MI_STORE_DATA_IMM

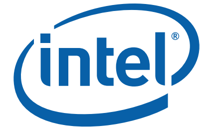
MI_STORE_DATA_IMM			
Project:	All	**Length Bias:**	2

The MI_STORE_DATA_IMM command requests a write of the QWord constant supplied in the packet to the specified Memory Address. As the write targets a System Memory Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped).

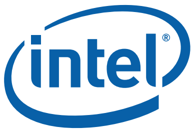
Programming Notes:

- This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll un-cached memory or device registers). However, the cacheable nature of the transaction is determined by the setting of the "mapping type" in the GTT entry.
- This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete "eventually", there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations. All writes to memory generated using this command are expected to finish in order.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 20h MI_STORE_DATA_IMM Format: OpCode
	22	**Memory Address Type** Project: All **Value** | **Name** | **Description** | **Project** 0h | Reserved | Physical address | All 1h | Reserved | Virtual address. Hardware will translate this address using the GTT. The GTT (global or per-process) used for the translation will be the same GTT used to access the buffer executing this instruction translate this address using the GTT. The GTT (global or per-process) used for the translation will be the same GTT used to access the buffer executing this instruction. | All
	21:6	**Reserved** Project: All Format: MBZ



MI_STORE_DATA_IMM		
	5:0	**DWord Length** Default Value: 2h Excludes DWord (0,1) = 2 for DWord, 3 for QWord Format: =n Total Length - 2
1	31:0	**Reserved** Project: All Format: MBZ
2	31:0	**Reserved** Project: All Format: MBZ
3	31:0	**Data DWord 0** Project: All Format: U32 This field specifies the DWord value to be written to the targeted location. For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0).
4	31:0	**Data DWord 1** Project: All Format: U32 This field specifies the upper DWord value to be written to the targeted QWord location (DW 1).



3.6 MI_STORE_DATA_INDEX

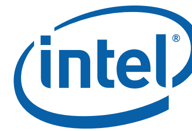
MI_STORE_DATA_INDEX			
Project:	All	**Length Bias:**	2

The MI_STORE_DATA_INDEX command requests a write of the data constant supplied in the packet to the specified offset from the System Address defined by the Hardware Status Page Address Register. As the write targets a System Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped).

Programming Notes:

- Use of this command with an invalid or uninitialized value in the Hardware Status Page Address Register is UNDEFINED.
- This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll uncached memory or device registers).
- This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete "eventually", there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations.

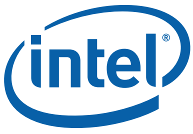
DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 21h MI_STORE_DATA_INDEX Format: OpCode
	22	**Reserved** Project: All Format: Setting this bit will cause this command to offset in the Surface Probe List instead of the hardware status page. This is intended to be used internally only (it is UNDEFINED to set this bit in a command in a ring or batch buffer.)
	21:6	**Reserved** Project: All Format: MBZ
	5:0	**DWord Length** Default Value: 1h Excludes DWord (0,1) = 1 for DWord, 2 for QWord Format: =n Total Length - 2
1	31:12	**Reserved** Project: All Format: MBZ



MI_STORE_DATA_INDEX		
	11:2	**Offset** Project: All Format: U10 zero-based DWord offset into the HW status page. Address: HardwareStatusPageOffset[11:2] Surface Type: U32 Range [16, 1023] This field specifies the offset (into the hardware status page) to which the data will be written. Note that the first few DWords of this status page are reserved for special-purpose data storage – targeting these reserved locations via this command is UNDEFINED.
	1:0	**Reserved** Project: All Format: MBZ
2	31:0	**Data DWord 0** Project: All Format: U32 This field specifies the DWord value to be written to the targeted location. For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0).
3	31:0	**Data DWord 1** Project: All Format: U32 This field specifies the upper DWord value to be written to the targeted QWord location (DW 1).

3.7 MI_USER_INTERRUPT

MI_USER_INTERRUPT		
Project: All		**Length Bias:** 1
The MI_USER_INTERRUPT command is used to generate a User Interrupt condition. The parser will continue parsing after processing this command. See User Interrupt.		
DWord	**Bit**	**Description**
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 02h MI_USER_INTERRUPT Format: OpCode
	22:0	**Reserved** Project: All Format: MBZ



3.8 MI_WAIT_FOR_EVENT

MI_WAIT_FOR_EVENT			
Project:	All	**Length Bias:**	1

The MI_WAIT_FOR_EVENT command is used to pause command stream processing until a specific event occurs or while a specific condition exists. See Wait Events/Conditions, Device Programming Interface in *MI Functions*. Only one event/condition can be specified -- specifying multiple events is UNDEFINED.

The effect of the wait operation depends on the source of the command. If executed from a batch buffer, the parser will halt (and suspend command arbitration) until the event/condition occurs. If executed from a ring buffer, further processing of that ring will be suspended, although command arbitration (from other rings) will continue. Note that if a specified condition does not exist (the condition code is inactive) at the time the parser executes this command, the parser proceeds, treating this command as a no-operation.

If execution of this command from a primary ring buffer causes a wait to occur, the active ring buffer will *effectively* give up the remainder of its time slice (required in order to enable arbitration from other primary ring buffers).

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 0h MI_COMMAND Format: OpCode
	28:23	**MI Command Opcode** Default Value: 03h MI_WAIT_FOR_EVENT Format: OpCode
	22:0	**Reserved** Project: All Format: MBZ

§§

4 Graphics Memory Interface Functions

4.1 Introduction

The major role of an integrated graphics device's Memory Interface (MI) function is to provide various client functions access to "graphics" memory used to store commands, surfaces, and other information used by the graphics device. This chapter describes the basic mechanisms and paths by which graphics memory is accessed.

Information not presented in this chapter includes:

Microarchitectural and implementation-dependent features (e.g., internal buffering, caching and arbitration policies).

MI functions and paths specific to the operation of external (discrete) devices attached via external connections.

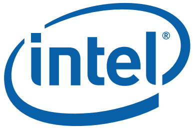
MI functions essentially unrelated to the operation of the internal graphics devices, e.g., traditional "chipset functions" (refer to the device's C-Spec for this information).

4.2 Graphics Memory Clients

The MI function provides memory access functionality to a number of external and internal graphics memory *clients*, as described in **Error! Reference source not found.**.

Table 4-1. Graphics Memory Clients

MI Client	Access Modes
Host Processor	Read/Write of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses
External PEG Graphics Device	**Write-Only** of Graphics Operands located in Main Memory via the Graphics Aperture. (This client is not described in this chapter).
Peer PCI Device	**Write-Only** of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses (i.e., mapped by GTT*). Note that DMI access to Graphics registers is not supported.*
Snooped Read/Write (internal)	Internally-generated snooped reads/writes.
Command Stream (internal)	DMA Read of graphics commands and related graphics data.
Vertex Stream (internal)	DMA Read of indexed vertex data from Vertex Buffers by the 3D Vertex Fetch (VF) Fixed Function.
Instruction/State Cache (internal)	Read of pipelined 3D rendering state used by the 3D/Media Functions and instructions executed by the EUs.



MI Client	Access Modes
Render Cache (internal)	Read/Write of graphics data operated upon by the graphics rendering engines (Blt, 3D, MPEG, etc.) Read of render surface state.
Sampler Cache (internal)	Read of texture (and other sampled surface) data stored in graphics memory.
Display/Overlay Engines (internal)	Read of display, overlay, cursor and VGA data.

4.3 Graphics Memory Addressing Overview

The Memory Interface function provides access to graphics memory (GM) clients. It accepts memory addresses of various types, performs a number of optional operations along *address paths*, and eventually performs reads and writes of graphics memory data using the resultant addresses. The remainder of this subsection will provide an overview of the graphics memory clients and address operations.

4.3.1 Graphics Address Path

Error! Reference source not found. shows the internal graphics memory address path, connection points, and optional operations performed on addresses. Externally-supplied addresses are normalized to zero-based *Graphics Memory* (*GM) addresses* (GM_Address). If the GM address is determined to be a tiled address (based on inclusion in a fenced region or via explicit surface parameters), *address tiling* is performed. At this point the address is considered a *Logical Memory address*, and is translated into a *Physical Memory address* via the GTT and associated TLBs. The physical memory location is then accessed.

CPU accesses to graphics memory are not snooped on the front side bus post GTT translation. Hence pages that are mapped cacheable in the GTT will not be coherent with the CPU cache if accessed through graphics memory aperture. Also, such accesses may have side effects in the hardware.

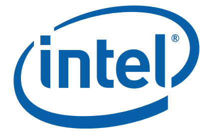
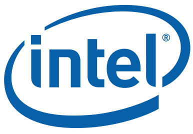


Figure 4-1. Graphics Memory Paths

CPU / Ext. Gfx Device
Device 2 PCI Graphics Memory Range
GM Range Offset Removal
Graphics Memory Address (0-based)
Graphics Memory -capable Internal Functions/ Caches
Tiled Address Determination
Fence Registers
Surface Parameters
Logical Memory Address (0-based)
Address Tiling Logic
Logical Memory Mapping
TLBs
Physical Address-based Internal Functions
Snoop Logic
Physical Memory Address
PTE Fetch
Main Memory
GTT
Graphics Addr Paths

The remainder of this chapter describes the basic features of the graphics memory address pipeline, namely Address Tiling, Logical Address Mapping, and Physical Memory types and allocation considerations.



4.4 Graphics Memory Address Spaces

Table 4-2 lists the five supported Graphics Memory Address Spaces. Note that the Graphics Memory Range Removal function is automatically performed to transform system addresses to internal, zero-based Graphics Addresses.

Table 4-2. Graphics Memory Address Types

Address Type	Description	Range
Dev2_GM_Address	Address range allocated via the Device 2 (integrated graphics device) GMADR register. The processor and other peer (DMI) devices utilize this address space to read/write graphics data that resides in Main Memory. This address is internally converted to a GM_Address.	Some 64MB, 128MB, 256MB or 512MB address range normally above TOM
GM_Address	Zero-based logical Graphics Address, utilized by internal device functions to access GTT-mapped graphics operands. GM_Addresses are typically passed in commands and contained in state to specify operand location.	[0, 64MB-1], [0, 128MB-1], [0, 256MB-1] or [0, 512MB-1]
PGM_Address	Zero-based logical Per-Process Graphics Address, utilized by internal device functions to access render GTT (PPGTT) mapped graphics operands. Memory in this space is not accessible by the processor and other peer (DMI) devices unless aliased to a GM_Address.	[0, 64MB-1], [0,128MB-1], [0,256MB-1], [0,512MB-1] or [0, 1GB – 1]

4.5 Address Tiling Function

When dealing with memory operands (e.g., graphics surfaces) that are inherently rectangular in nature, certain functions within the graphics device support the storage/access of the operands using alternative (tiled) memory formats in order to increase performance. This section describes these memory storage formats, why/when they should be used, and the behavioral mechanisms within the device to support them.

4.5.1 Linear vs. Tiled Storage

Regardless of the memory storage format, "rectangular" memory operands have a specific *width* and *height*, and are considered as residing within an enclosing rectangular region whose width is considered the *pitch* of the region and surfaces contained within. Surfaces stored within an enclosing region must have widths less than or equal to the region pitch (indeed the enclosing region may coincide exactly with the surface). Figure 4-2 shows these parameters.

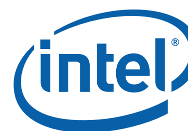
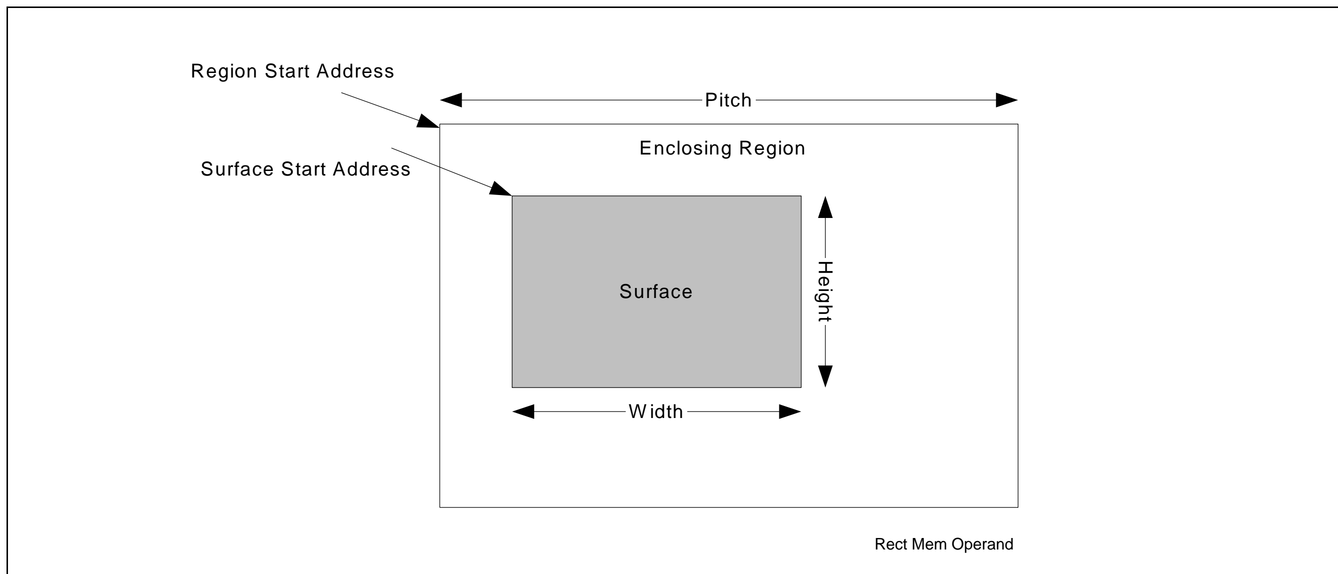
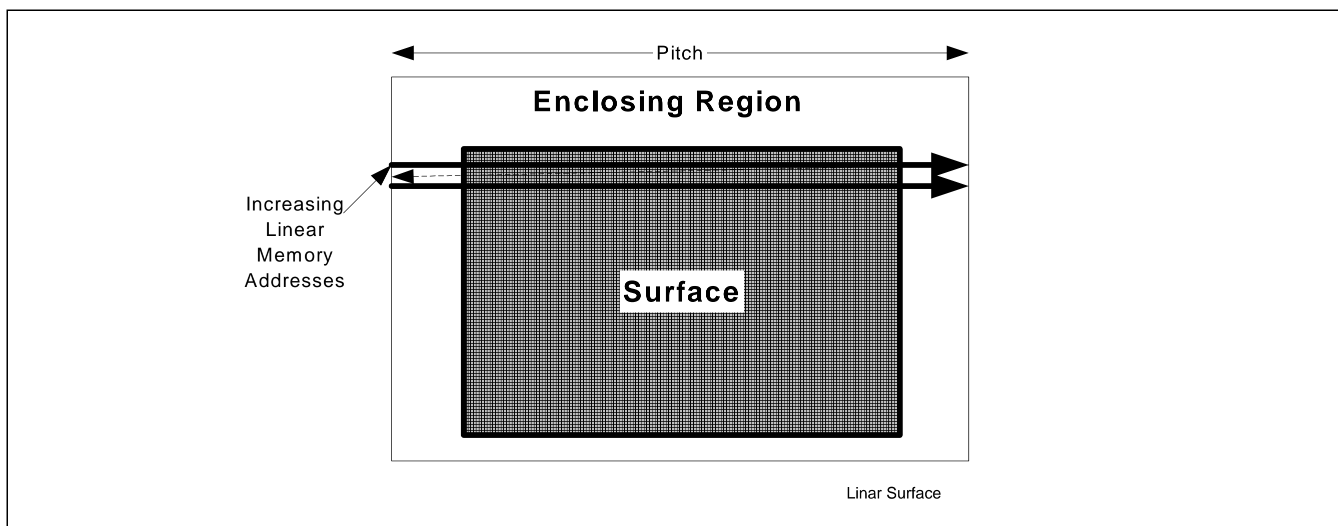


Figure 4-2. Rectangular Memory Operand Parameters

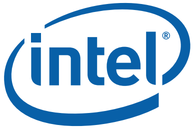


The simplest storage format is the *linear* format (see Figure 4-3), where each row of the operand is stored in sequentially increasing memory locations. If the surface width is less than the enclosing region's pitch, there will be additional memory storage between rows to accommodate the region's pitch. The pitch of the enclosing region determines the distance (in the memory address space) between vertically-adjacent operand elements (e.g., pixels, texels).

Figure 4-3. Linear Surface Layout



The linear format is best suited for 1-dimensional row-sequential access patterns (e.g., a display surface where each scanline is read sequentially). Here the fact that one object element may reside in a different memory page than its vertically-adjacent neighbors is not significant; all that matters is that horizontally-adjacent elements are stored contiguously. However, when a device function needs to access a 2D subregion within an operand (e.g., a read or write of a 4x4 pixel span by the 3D renderer, a read of a 2x2 texel block for bilinear filtering), having vertically-adjacent elements fall within different memory pages is to be avoided, as

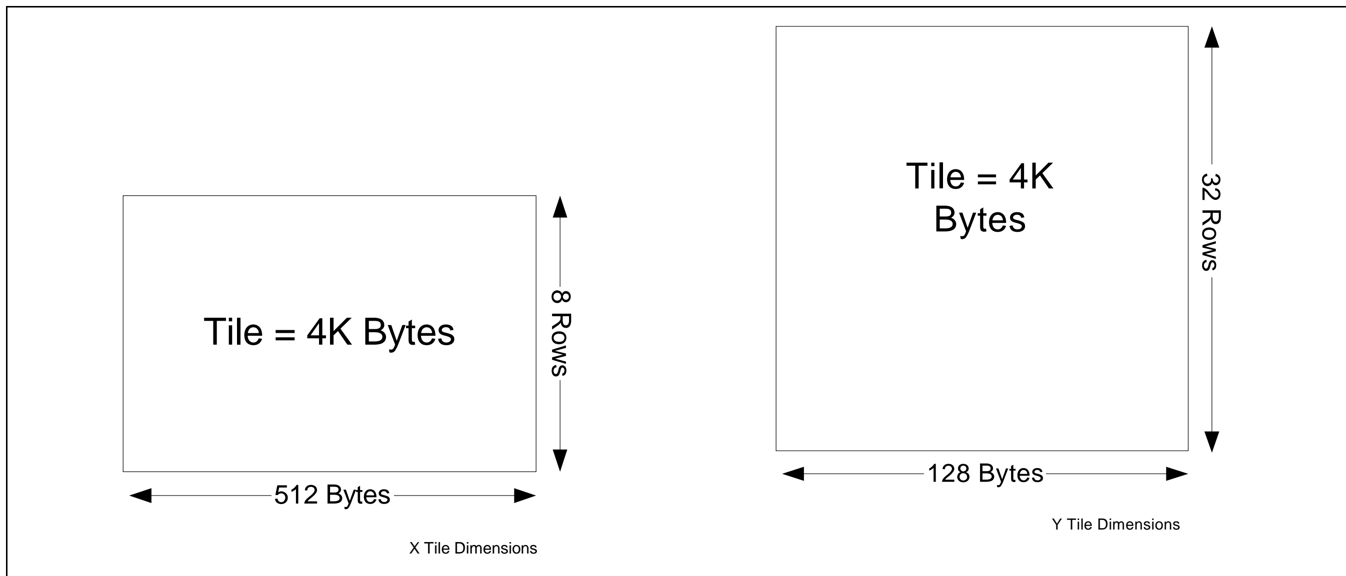


the page crossings required to complete the access typically incur increased memory latencies (and therefore lower performance).

One solution to this problem is to divide the enclosing region into an array of smaller rectangular regions, called memory *tiles*. Surface elements falling within a given tile will all be stored in the same physical memory page, thus eliminating page-crossing penalties for 2D subregion accesses within a tile and thereby increasing performance.

Tiles have a fixed 4KB size and are aligned to physical DRAM page boundaries. They are either 8 rows high by 512 bytes wide or 32 rows high by 128 bytes wide (see Figure 4-4). Note that the dimensions of tiles are irrespective of the data contained within – e.g., a tile can hold twice as many 16-bit pixels (256 pixels/row x 8 rows = 2K pixels) than 32-bit pixels (128 pixels/row x 8 rows = 1K pixels).

Figure 4-4. Memory Tile Dimensions



<u>The pitch of a tiled enclosing region must be an integral number of tile widths</u>. The 4KB tiles within a tiled region are stored sequentially in memory in row-major order.

Figure 4-5 shows an example of a tiled surface located within a tiled region with a pitch of 8 tile widths (512 bytes * 8 = 4KB). Note that it is the <u>enclosing region</u> that is divided into tiles – the surface is not necessarily aligned or dimensioned to tile boundaries.

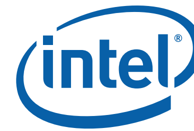
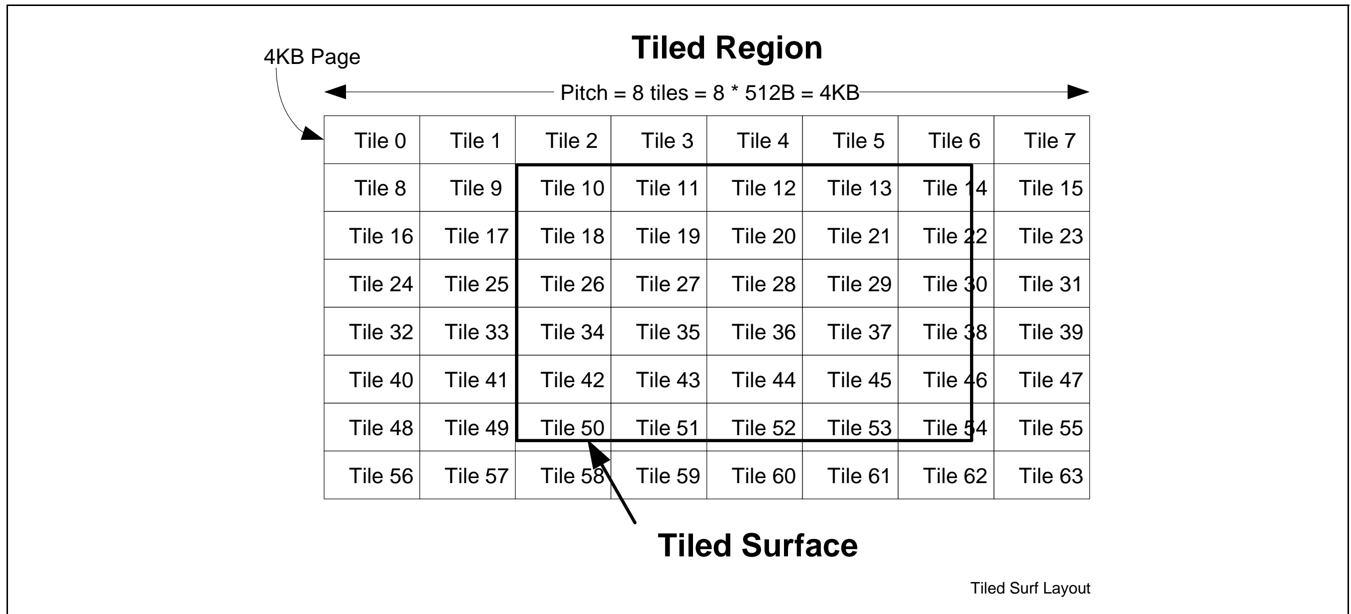


Figure 4-5. Tiled Surface Layout



4.5.2 Tile Formats

The device supports both *X-Major* (row-major) and *Y-Major* (column major) storage of tile data units, as shown in the following figures. A 4KB tile is subdivided into an 8-high by 32-wide array of 16-byte OWords for X-Major Tiles (X Tiles for short), and 32-high by 8-wide array of OWords for Y-Major Tiles (Y Tiles). The selection of tile direction only impacts the internal organization of tile data, and does not affect how surfaces map onto tiles. Note that the diagrams are not to scale – the first format defines the contents of an 8-high by 512-byte wide tile, and the 2nd a 32-high by 128-byte wide tile. The storage of tile data units in X-Major or Y-Major fashion is sometimes refer to as the *walk* of the tiling.

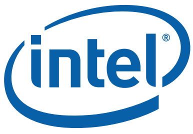
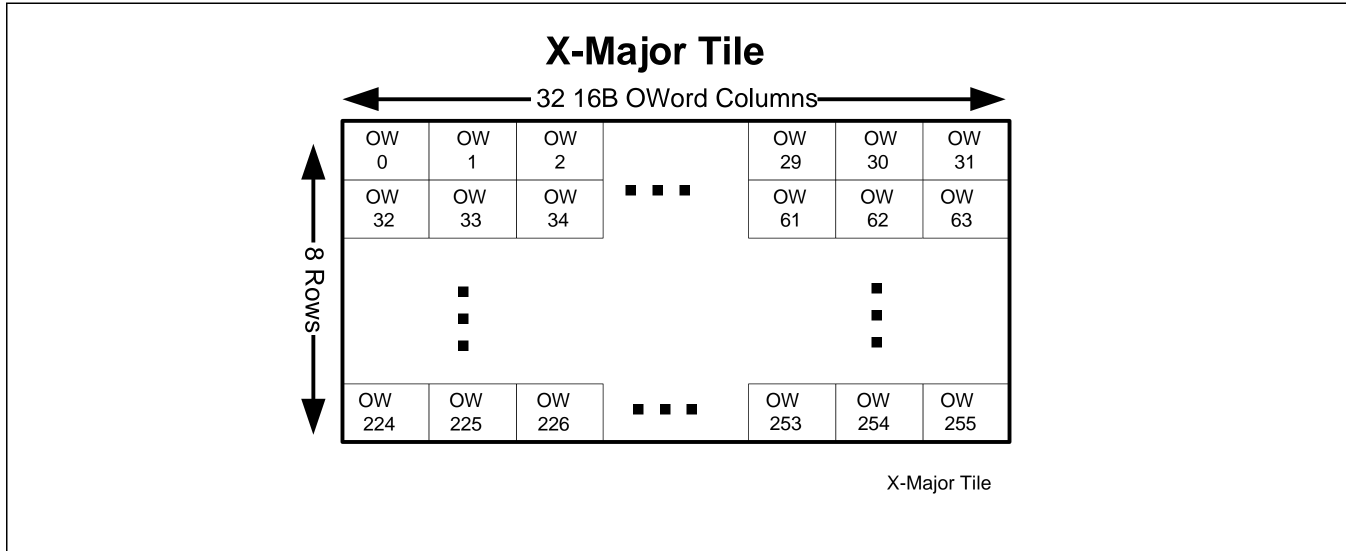
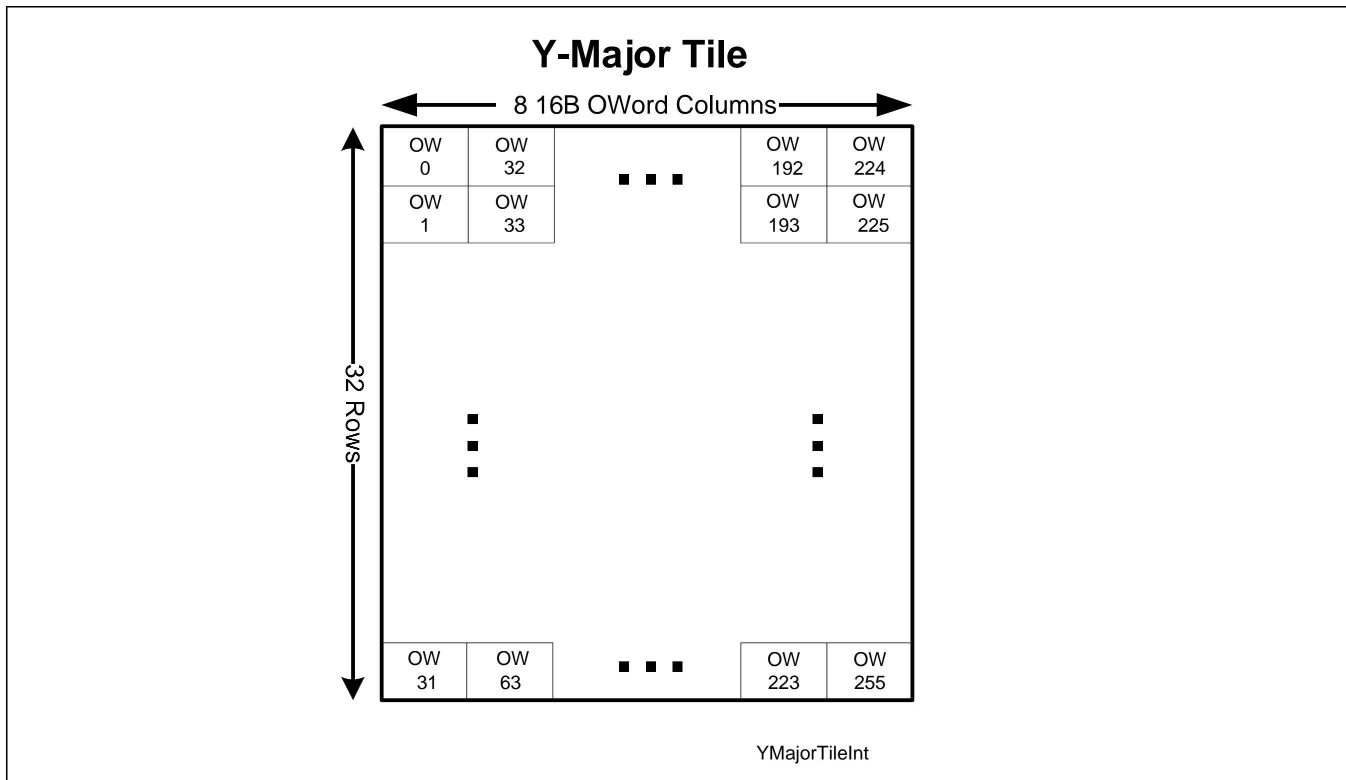


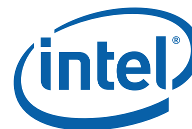
Table 4-3. X-Major Tile Layout



Note that an X-major tiled region with a tile pitch of 1 tile is actually stored in a linear fashion.

Figure 4-6. Y-Major Tile Layout





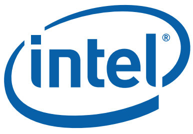
4.5.3 Tiling Algorithm

The following pseudocode describes the algorithm for translating a tiled memory surface in graphics memory to an address in logical space.

```
Inputs: LinearAddress(offset into regular or LT aperture in terms of bytes),
        Pitch(in terms of tiles),
      WalkY (1 for Y and 0 for X)

Static Parameters: TileH (Height of tile, 8 for X and 32 for Y),
                    TileW (Width of Tile in bytes, 512 for X and 128 for Y)

TileSize = TileH * TileW;
RowSize = Pitch * TileSize;

If (Fenced) {
       LinearAddress = LinearAddress – FenceBaseAddress
       LinearAddrInTileW = LinearAddress div TileW;
       Xoffset_inTile = LinearAddress mod TileW;
       Y = LinearAddrInTileW div Pitch;
       X = LinearAddrInTileW mod Pitch + Xoffset_inTile;
}

// Internal graphics clients that access tiled memory already have the X, Y
// coordinates and can start here
YOff_Within_Tile = Y mod TileH;
XOff_Within_Tile = X mod TileW;

TileNumber_InY = Y div TileH;
TileNumber_InX = X div TileW;

TiledOffsetY = RowSize * TileNumber_InY + TileSize * TileNumber_InX + TileH *
             XOff_Within_Tile + YOff_Within_Tile * 16 + (XOff_Within_Tile mod 16);

TiledOffsetX = RowSize * TileNumber_InY + TileSize * TileNumber_InX + TileW *
             YOff_Within_Tile + XOff_Within_Tile;

TiledOffset = WalkY? TiledOffsetY : TiledOffsetX;

TiledAddress = Tiled? (BaseAddress + TiledOffset): (BaseAddress + Y*LinearPitch + X);
}
```

The Y-Major tile formats have the characteristic that a surface element in an even row is located in the same aligned 64-byte cacheline as the surface element immediately below it (in the odd row). This spatial locality can be exploited to increase performance when reading 2x2 texel squares for bilinear texture filtering, or reading and writing aligned 4x4 pixel spans from the 3D Render pipeline.

On the other hand, the X-Major tile format has the characteristic that horizontally-adjacent elements are stored in sequential memory addresses. This spatial locality is advantageous when the surface is scanned in row-major order for operations like display refresh. For this reason, the Display and Overlay memory streams only support linear or X-Major tiled surfaces (Y-Major tiling is not supported by these functions). This has the



side effect that 2D- or 3D-rendered surfaces must be stored in linear or X-Major tiled formats if they are to be displayed. Non-displayed surfaces, e.g., "rendered textures", can also be stored in Y-Major order.

4.5.4 Tiling Support

The rearrangement of the surface elements in memory must be accounted for in device functions operating upon tiled surfaces. (Note that not all device functions that access memory support tiled formats). This requires either the modification of an element's linear memory address or an alternate formula to convert an element's X,Y coordinates into a tiled memory address.

However, before tiled-address generation can take place, some mechanism must be used to determine whether the surface elements accessed fall in a linear or tiled region of memory, and if tiled, what the tile region pitch is, and whether the tiled region uses X-Major or Y-Major format. There are two mechanisms by which this detection takes place: (a) an implicit method by detecting that the pre-tiled (linear) address falls within a "fenced" tiled region, or (b) by an explicit specification of tiling parameters for surface operands (i.e., parameters included in surface-defining instructions).

The following table identifies the tiling-detection mechanisms that are supported by the various memory streams.

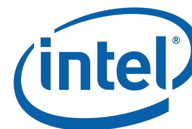
Access Path	Tiling-Detection Mechanisms Supported
Processor access through the Graphics Memory Aperture	Fenced Regions
3D Render (Color/Depth Buffer access)	Explicit Surface Parameters
Sampled Surfaces	Explicit Surface Parameters
Blt operands	Explicit Surface Parameters
Display and Overlay Surfaces	Explicit Surface Parameters

4.5.4.1 Tiled (Fenced) Regions

The only mechanism to support the access of surfaces in tiled format by the host or external graphics client is to place them within "fenced" tiled regions within Graphics Memory. A fenced region is a block of Graphics Memory specified using one of the sixteen FENCE device registers. (See *Memory Interface Registers* for details). Surfaces contained within a fenced region are considered tiled from an external access point of view. Note that fences cannot be used to untile surfaces in the PGM_Address space since external devices cannot access PGM_Address space. Even if these surfaces (or any surfaces accessed by an internal graphics client) fall within a region covered by an enabled fence register, that enable will be effectively masked during the internal graphics client access. Only the explicit surface parameters described in the next section can be used to tile surfaces being accessed by the internal graphics clients.

Each FENCE register (if its Fence Valid bit is set) defines a Graphics Memory region ranging from 4KB to the aperture size. The region is considered rectangular, with a pitch in tile widths from 1 tile width (128B or 512B) to 256 tile X widths (256 * 512B = 128KB) and 1024 tile Y widths (1024 * 128B = 128KB). Note that fenced regions must not overlap, or operation is UNDEFINED.

Also included in the FENCE register is a Tile Walk field that specifies which tile format applies to the fenced region.



4.5.4.2 Tiled Surface Parameters

Internal device functions require explicit specification of surface tiling parameters via information passed in commands and state. This capability is provided to limit the reliance on the fixed number of fence regions.

The following table lists the surface tiling parameters that can be specified for 3D Render surfaces (Color Buffer, Depth Buffer, Textures, etc.) via SURFACE_STATE.

Surface Parameter	Description
Tiled Surface	If ENABLED, the surface is stored in a tiled format. If DISABLED, the surface is stored in a linear format.
Tile Walk	If Tiled Surface is ENABLED, this parameter specifies whether the tiled surface is stored in Y-Major or X-Major tile format.
Base Address	Additional restrictions apply to the base address of a Tiled Surface vs. that of a linear surface.
Pitch	Pitch of the surface. Note that, if the surface is tiled, this pitch must be a multiple of the tile width.

4.5.4.3 Tiled Surface Restrictions

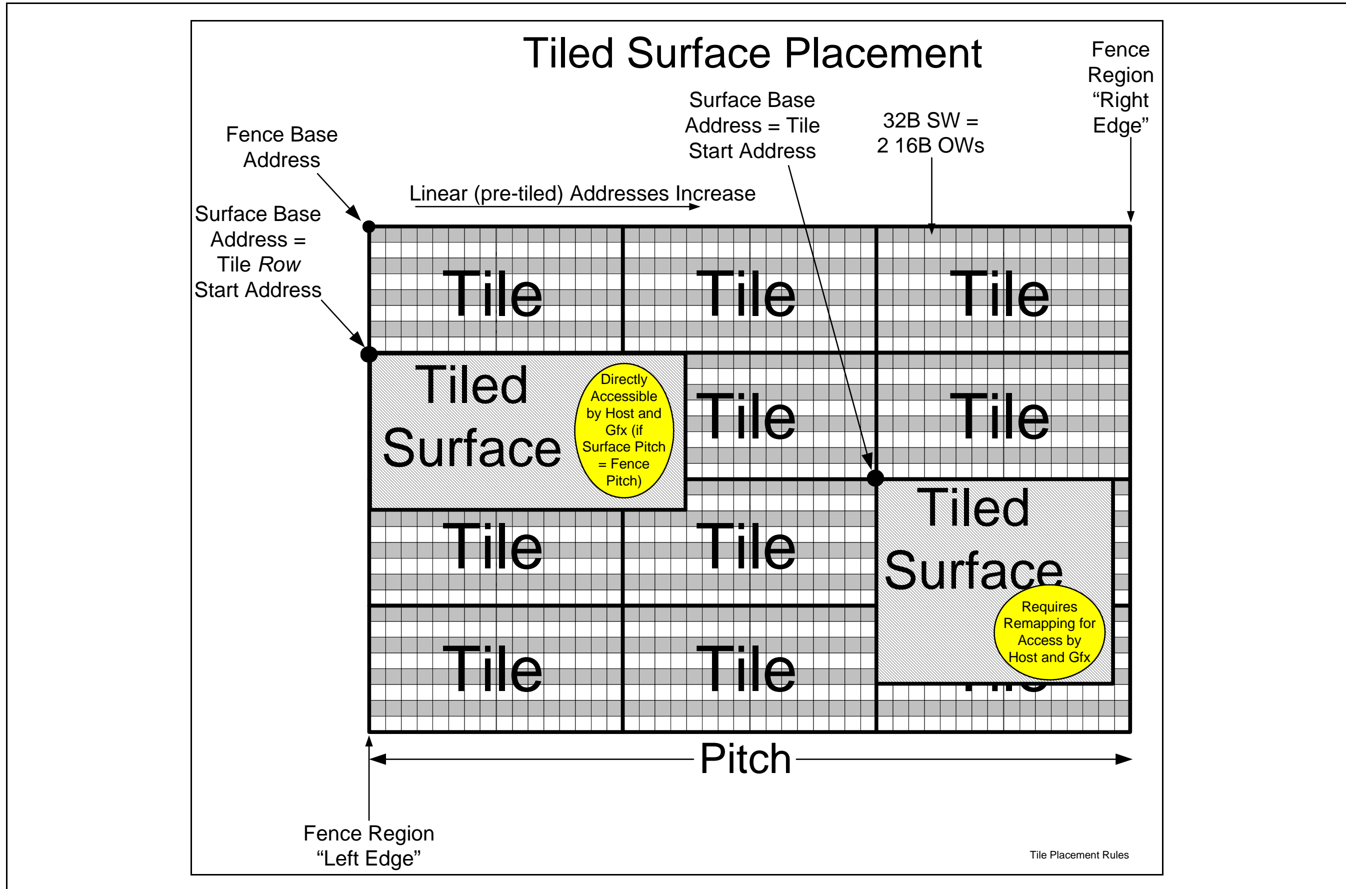
Additional restrictions apply to the Base Address and Pitch of a surface that is tiled. In addition, restrictions for tiling via SURFACE_STATE are subtly different from those for tiling via fence regions. The most restricted surfaces are those that will be accessed both by the host (via fence) and by internal device functions. An example of such a surface is a tiled texture that is initialized by the CPU and then sampled by the device.

The tiling algorithm for internal device functions is different from that of fence regions. Internal device functions always specify tiling in terms of a surface. The surface must have a base address, and this base address is not subject to the tiling algorithm. Only *offsets* from the base address (as calculated by X, Y addressing within the surface) are transformed through tiling. The base address of the surface must therefore be 4KB-aligned. This forces the 4KB tiles of the tiling algorithm to exactly align with 4KB device pages once the tiling algorithm has been applied to the offset. The width of a surface must be less than or equal to the surface pitch. There are additional considerations for surfaces that are also accessed by the host (via a fence region).

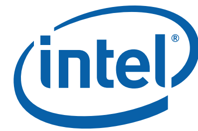
Fence regions have no base address per se. Host linear addresses that fall in a fence region are translated in their entirety by the tiling algorithm. It is as if the surface being tiled by the fence region has a base address in graphics memory equal to the fence base address, and all accesses of the surfaces are (possibly quite large) offsets from the fence base address. Fence regions have a virtual "left edge" aligned with the fence base address, and a "right edge" that results from adding the fence pitch to the "left edge". Surfaces in the fence region must not straddle these boundaries.

Base addresses of surfaces that are to be accessed both by an internal graphics client and by the host have the tightest restrictions. In order for the surface to be accessed without GTT re-mapping, the surface base address (as set in SURFACE_STATE) must be a "Tile Row Start Address" (TRSA). The first address in each tile row of the fence region is a Tile Row Start Address. The first TRSA is the fence base address. Each TRSA can be generated by adding an integral multiple of the row size to the fence base address. The row size is simply the fence pitch in tiles multiplied by 4KB (the size of a tile.)

Figure 4-7. Tiled Surface Placement

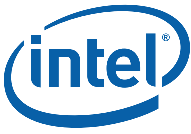


The pitch in SURFACE_STATE must be set equal to the pitch of the fence that will be used by the host to access the surface if the same GTT mapping will be used for each access. If the pitches differ, a different GTT mapping must be used to eliminate the "extra" tiles (4KB memory pages) that exist in the excess rows at the right side of the larger pitch. Obviously no part of the surface that will be accessed can lie in pages that exist only in one mapping but not the other. The new GTT mapping can be done manually by SW between the time the host writes the surface and the device reads it, or it can be accomplished by arranging for the client to use a different GTT than the host (the PPGTT -- see Logical Memory Mapping below).



The width of the surface (as set in SURFACE_STATE) must be less than or equal to both the surface pitch and the fence pitch in any scenario where a surface will be accessed by both the host and an internal graphics client. Changing the GTT mapping will not help if this restriction is violated.

Surface Access	Base Address	Pitch	Width	Tile "Walk"
Host only	**No restriction**	**Integral multiple of tile size <= 128KB**	**Must be <= Fence Pitch**	**No restriction**
Client only	**4KB-aligned**	**Integral multiple of tile size <= 256KB**	**Must be <= Surface Pitch**	**Restrictions imposed by the client (see Per-Stream Tile Format Support)**
Host and Client, No GTT Remapping	**Must be TRSA**	**Fence Pitch = Surface Pitch = integral multiple of tile size <= 256KB**	**Width <= Pitch**	**Surface Walk must meet client restriction, Fence Walk = Surface Walk**
Host and Client, GTT Remapping	**4KB-aligned for client (will be tile-aligned for host)**	**Both must be Integral multiple of tile size <=128KB, but not necessarily the same**	**Width <= Min(Surface Pitch, Fence Pitch)**	**Surface Walk must meet client restriction, Fence Walk = Surface Walk**



4.5.5 Per-Stream Tile Format Support

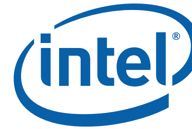
<table>
<tr><th>MI Client</th><th>Tile Formats Supported</th></tr>
<tr><td>CPU Read/Write</td><td>All</td></tr>
<tr><td>Display/Overlay</td><td>Y-Major not supported.
X-Major required for Async Flips</td></tr>
<tr><td>Blt</td><td>Linear and X-Major only
No Y-Major support</td></tr>
<tr><td>3D Sampler</td><td>All Combinations of TileY, TileX and Linear are supported. TileY is the fastest, Linear is the slowest.</td></tr>
<tr><td>3D Color,Depth</td><td>
<table>
<tr><th>Rendering Mode
Color-vs-Depth bpp</th><th>Buffer Tiling Supported</th></tr>
<tr><td>Classical
Same Bpp</td><td>Both Linear
Both TileX
Both TileY
Linear & TileX
Linear & TileY
TileX & TileY</td></tr>
<tr><td>Classical
Mixed Bpp</td><td>Both Linear
Both TileX
Both TileY
Linear & TileX
Linear & TileY
TileX & TileY</td></tr>
</table>
NOTE: 128BPE Format Color buffer (render target) MUST be either TileX or Linear.</td></tr>
</table>

4.6 Logical Memory Mapping

In order to provide a contiguous address space for graphics operands (surfaces, etc.) yet allow this address space to be mapped onto possibly discontiguous physical memory pages, the internal graphics device supports a Logical Memory Space. A global *Graphics Translation Table* (GTT) is provided to map zero-based (and post-tiled) Logical Memory Addresses into a set of 4KB physical memory pages. (This mapping is also used for external PEG devices.)

There is another logical mapping function available local to each graphics process; this works identically to the global GTT with some additional restrictions. The base address for this per-process GTT (PPGTT) is determined by the PGTBL_CTL2 register. This register is saved and restored with ring context, thus providing each graphics context with its own local translation table and protected memory space (see Rendering Context Management later in this chapter).

The GTT and PPGTT are arrays of 4-byte *Page Table Entries* (PTEs) physically located in Main Memory. The GTT and PPGTT are comprised of a number of locked (non-swappable) physically-contiguous 4KB memory pages, with a maximum size (each) of 128 4KB pages (128K DWords map 128K*4KB = 512MB max) for Global GTT, and up to 512 4KB pages for PPGTT, for total up to 2GB max. GTT and PPGTT base addresses must be 4KB-aligned.



Note that the PTEs within the global GTT must be written only through GTTADDR (see the Device #2 Config registers for a description of this range), as the MI function needs to snoop PTE updates in order to invalidate TLBs, which cache PTEs. The PGTBL_CTL register also contains a Page Table Enable bit used to enable/disable Logical Memory mapping. With the exception of processor Read, Cursor and VGA clients, access to graphics memory is not permitted when the Page Table Enable bit is clear (i.e., disabled). The PGTBL_ER debug register provides information pertaining to HW-detected errors in the Logical Memory Mapping function (e.g., invalid PTEs, invalid mappings, etc.).

The PPGTT base address is also 4KB aligned, but it is programmed directly in physical memory space rather than through an alias mechanism like GTTADDR. Note that not all clients may use the PPGTT; only the global GTT is available for processor accesses as well as graphics accesses from display engines (including overlay and cursor). Any per-process access that occurs while the PPGTT is disabled (via a bit in PGTBL_CTL2) will default to a translation via the global GTT.

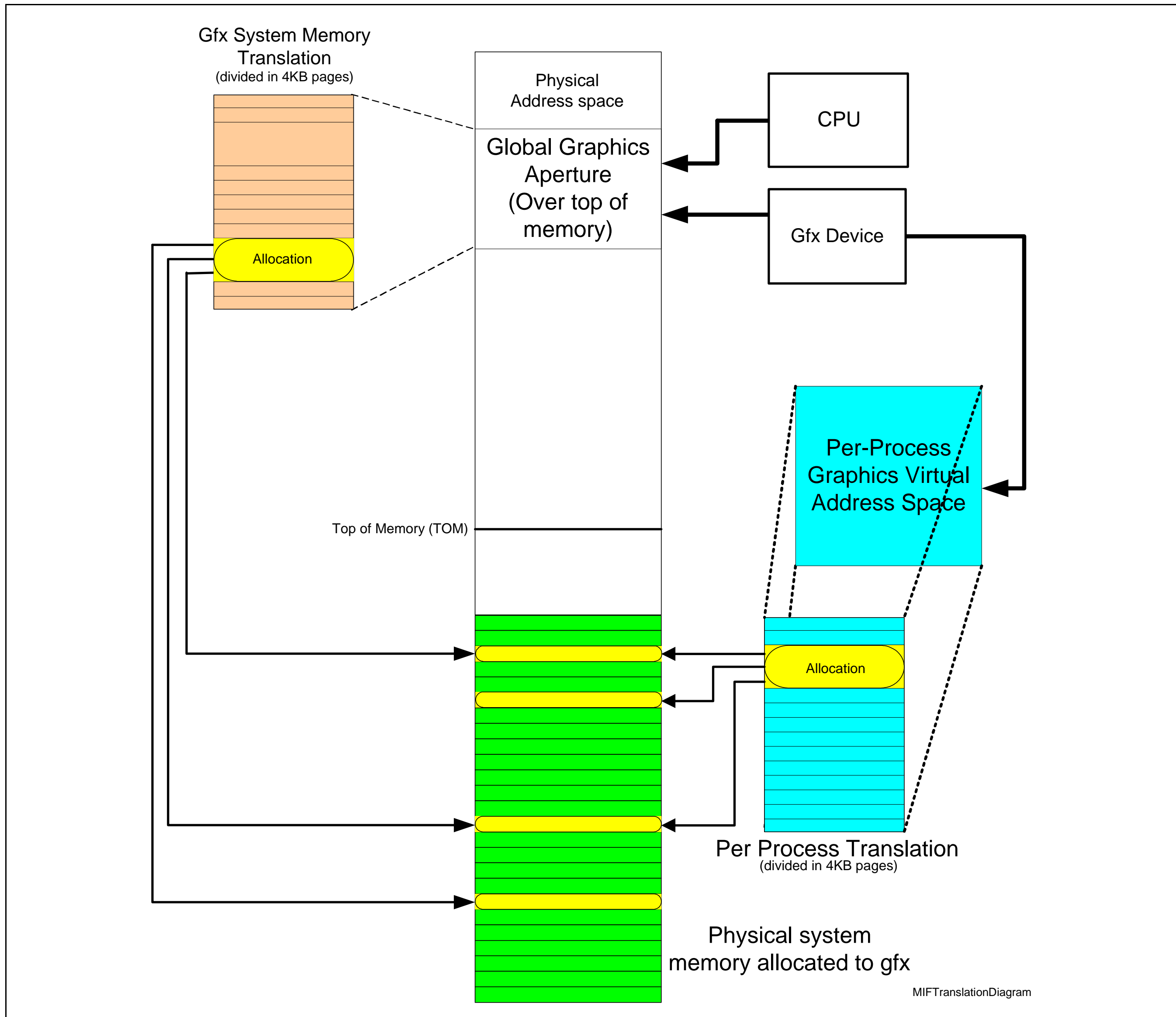
4.6.1 Logical Memory Space Mappings

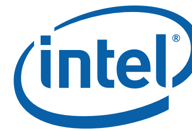
Each valid PTE maps a 4KB page of Logical Memory to an independent 4KB page of:

- MM: Main Memory (unsnooped), or
- SM: System Memory (snooped, therefore coherent with the processor cache, must not be accessed through the Dev2_GM_Address range by the CPU)

PTEs marked as invalid have no backing physical memory, and therefore the corresponding Logical Memory Address pages must not be accessed in normal operation.

Figure 4-8. Global and Render GTT Mapping





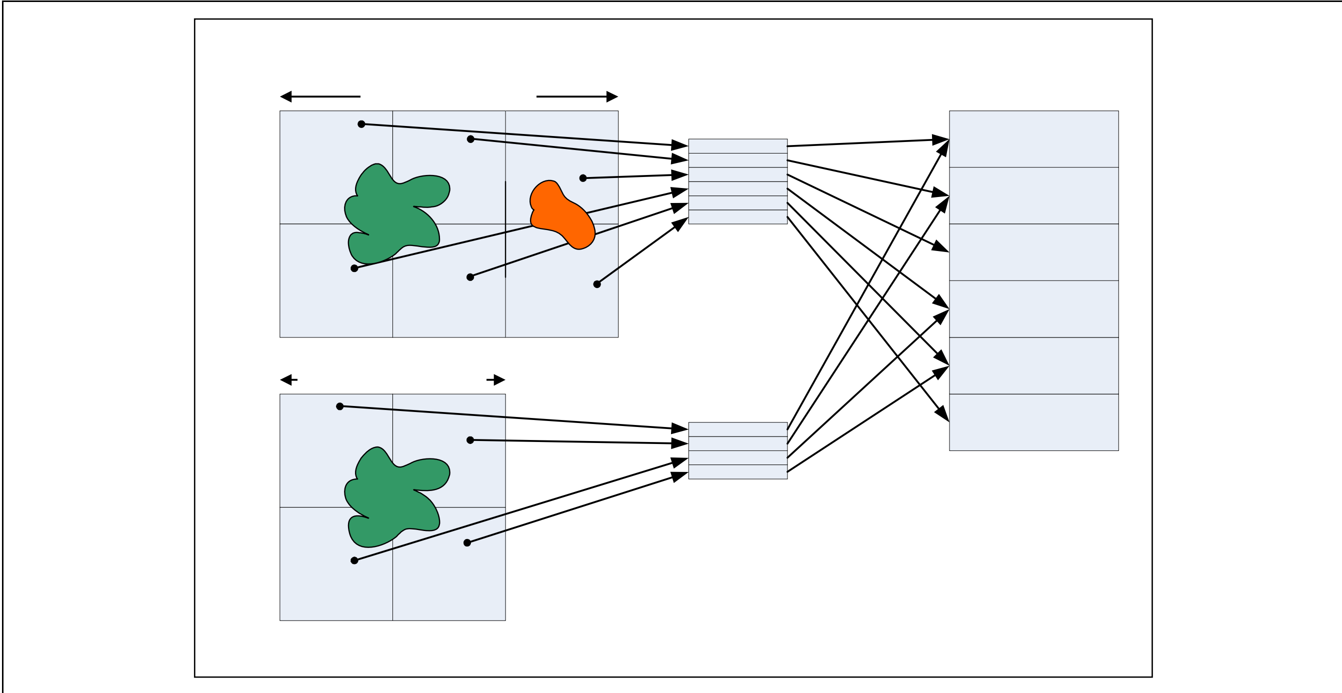
The following table lists the memory space mappings valid for each MI client:

MI Client	Logical Memory Space Mappings Supported	xGTT Usage
External Clients		
Host Processor	MM	GTT only
External PEG Device	None	n/a
Snooped Read/Write	None	n/a
Internal GPU Clients		
Render Command Ring Buffers	MM	GTT/PGTT, selected by PGTBL_STR2<2>
Render Command Batch Buffers	MM	GTT/PGTT, selected by PGTBL_STR2<5>
Indirect State Buffers	MM	GTT/PGTT, selected by PGTBL_STR2<4>
CURBE Constant Data	MM	Same xGTT used to fetch the CONSTANT_BUFFER command.
Media Object Indirect Data	MM	Same xGTT used to fetch the MEDIA_OBJECT command.
Vertex Fetch Data	MM, SM	GTT/PGTT, selected by PGTBL_STR2<3>
Sampler Cache (RO)	MM, SM	GTT/PGTT, selected by PGTBL_STR2<1>
DataPort Render Cache (R/W)	MM, SM	GTT/PGTT, selected by PGTBL_STR2<0>
Depth Buffer Cache (R/W)	MM	GTT/PGTT, selected by PGTBL_STR2<0>
Blit Engine	MM, SM	GTT/PGTT, selected by PGTBL_STR2<0>
MI_STORE_DATA_IMM Destination (if virtual addressed)	MM, SM	Same xGTT used to fetch the command.
PIPE_CONTROL Write Destination	MM, SM	GTT/PGTT, selected by the command
Display/Overlay Engines (internal)	MM	GTT only

Usage Note: Since the CPU cannot directly access memory pages mapped through a Graphics Process' local GTT (PPGTT), these pages must also be mapped though the global GTT (at least temporarily) in order for the CPU to initialize graphics data for a Graphics Process.

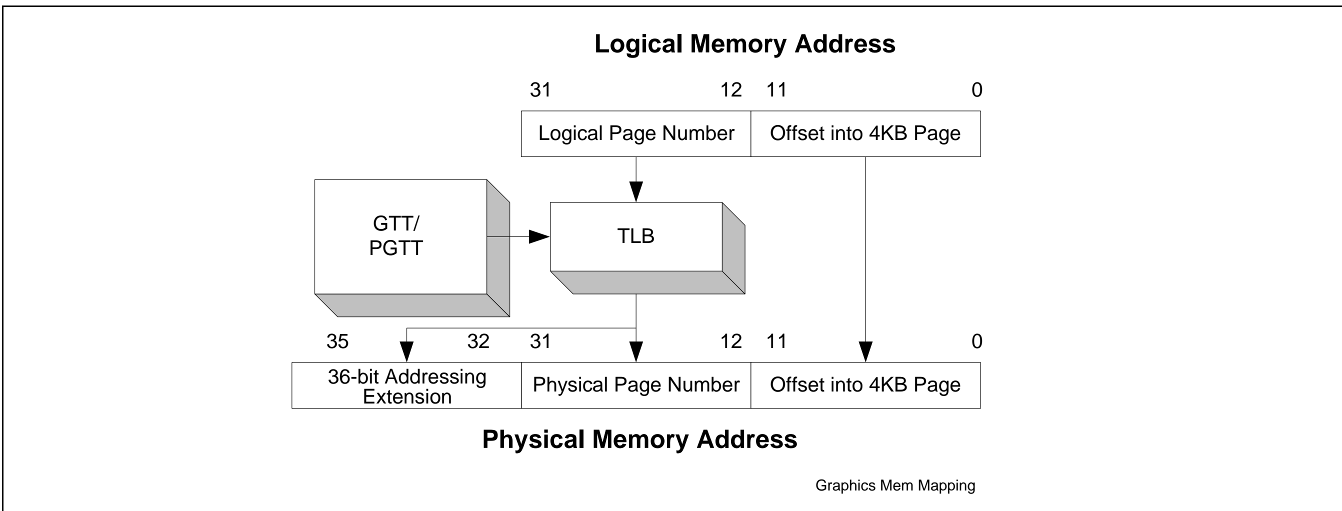
The PPGTT mechanism can be used by a client to access a surface with a pitch that is smaller than that of the fence region used by the host to initialize the surface, without having to physically move the data in memory.

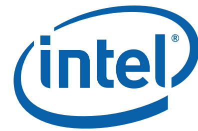
Figure 4-9. GTT Re-mapping to Handle Differing Pitches



Refer to the "Graphics Translation Table (GTT) Range (GTTADR) & PTE Description" in *Memory Interface Registers* for details on PTE formats and programming information. Refer to the *Memory Data Formats* chapter for device-specific details/restrictions regarding the placement/storage of the various data objects used by the graphics device.

Figure 4-10. Logical-to-Physical Graphics Memory Mapping



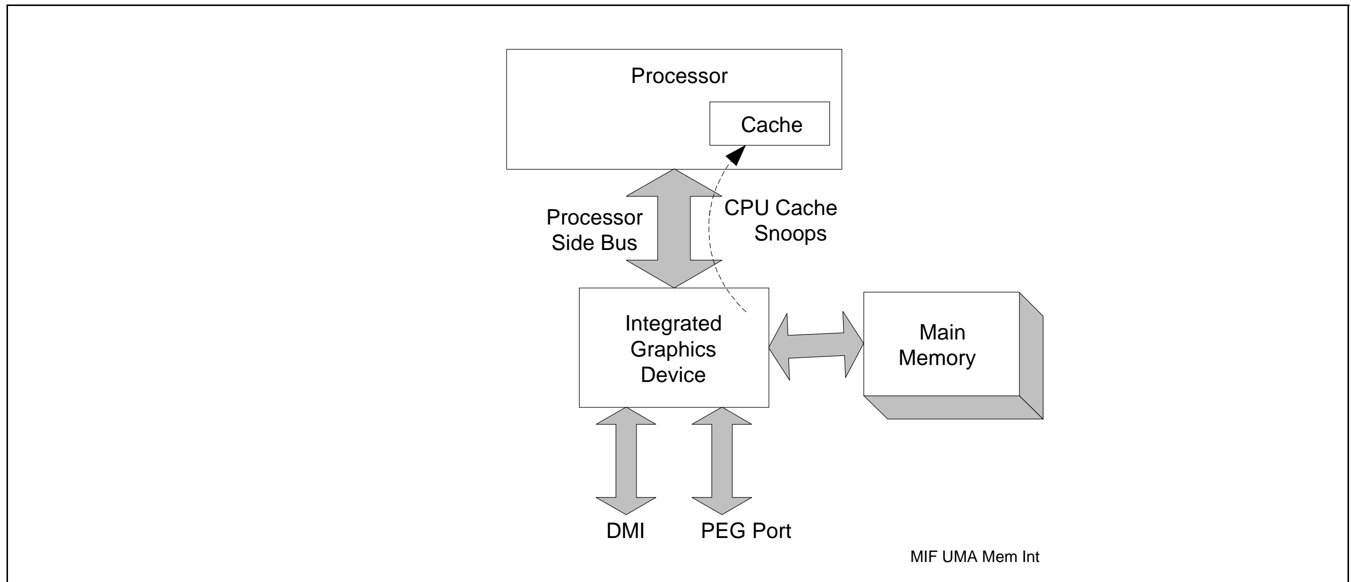


4.7 Physical Graphics Memory

The integrated graphics device satisfies all of its memory requirements using portions of main system memory. The integrated graphics device operates without any dedicated local memory, in a lower-cost configuration typically (though not necessarily officially) known as *Unified Graphics Memory* (UMA).

Figure 4-11 shows how the Main Memory is interfaced to the device.

Figure 4-11. Memory Interfaces

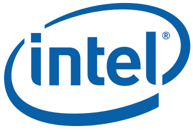


4.7.1 Physical Graphics Address Types

Error! Reference source not found. lists the various physical address types supported by the integrated graphics device. Physical Graphics Addresses are either generated by Logical Memory mappings or are directly specified by graphics device functions. These physical addresses are not subject to tiling or GTT re-mappings.

Table 4-4. Physical Memory Address Types

Address Type	Description	Range
MM_Address	Main Memory Address. Offset into physical, unsnooped Main Memory.	[0,TopOfMemory-1]
SM_Address	System Memory Address. Accesses are snooped in processor cache, allowing shared graphics/ processor access to (locked) cacheable memory data.	[0,4GB]



4.7.2 Main Memory

The integrated graphics device is capable of using 4KB pages of physical main (system) memory for graphics functions. Some of this main memory can be "stolen" from the top of system memory during initialization (e.g., for a VGA buffer). However, most graphics operands are dynamically allocated to satisfy application demands. To this end the graphics driver will frequently need to allocate locked-down (i.e., non-swappable) physical system memory pages – typically from a cacheable non-paged pool. The locked pages required to back large surfaces are typically non-contiguous. Therefore a means to support "logically-contiguous" surfaces backed by discontiguous physical pages is required. The Graphics Translation Table (GTT) that was described in previous sections provides the means.

4.7.2.1 Optimizing Main Memory Allocation

This section includes information for software developers on how to allocate SDRAM Main Memory (SM) for optimal performance in certain configurations. The general idea is that these memories are divided into some number of page types, and careful arrangement of page types both within and between surfaces (e.g., between color and depth surfaces) will result in fewer page crossings and therefore yield somewhat higher performance.

The algorithm for allocating physical SDRAM Main Memory pages to logical graphics surfaces is somewhat complicated by (1) permutations of memory device technologies (which determine page sizes and therefore the number of pages per device row), (2) memory device row population options, and (3) limitations on the allocation of physical memory (as imposed by the OS).

However, the theory to optimize allocation by limiting page crossing penalties is simple: (a) switching between open pages is optimal (again, the pages do not need to be sequential), (b) switching between memory device rows does not in itself incur a penalty, and (c) switching between pages within a particular bank of a row incurs a page miss and should therefore be avoided.

4.7.2.2 Application of the Theory (Page Coloring)

This section provides some scenarios of how Main Memory page allocation can be optimized.

4.7.2.2.1 3D Color and Depth Buffers

Here we want to minimize the impact of page crossings (a) between corresponding pages (1-4 tiles) in the Color and Depth buffers, and (b) when moving from a page to a neighboring page within a Color or Depth buffer. Therefore corresponding pages in the Color and Depth Buffers, and adjacent pages within a Color or Depth Buffer should be mapped to different page types (where a page's "type" or "color" refers to the row and bank it's in).

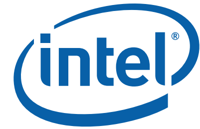


Figure 4-12. Memory Pages backing Color and Depth Buffers

Color Buffer

Page Type 0	Page Type 1	Page Type 0	Page Type 1	⋮
Page Type 2	Page Type 3	Page Type 2	Page Type 3	⋮
Page Type 0	Page Type 1	Page Type 0	Page Type 1	⋮
Page Type 2	Page Type 3	Page Type 2	Page Type 3	⋮
⋮	⋮	⋮	⋮	⋮

Depth Buffer

Page Type 3	Page Type 2	Page Type 3	Page Type 2	⋮
Page Type 1	Page Type 0	Page Type 1	Page Type 0	⋮
Page Type 3	Page Type 2	Page Type 3	Page Type 2	⋮
Page Type 1	Page Type 0	Page Type 1	Page Type 0	⋮
⋮	⋮	⋮	⋮	⋮

For higher performance, the Color and Depth Buffers could be allocated from different memory device rows.

4.7.2.2.2 Media/Video

The Y surfaces can be allocated using 4 page types in a similar fashion to the Color Buffer diagram above. The U and V surfaces would split the same 4 page types as used in the Y surface.

5 Device Programming Environment

The graphics device contains an extensive set of registers and commands (also referred to as “commands” or “packets”) for controlling 2D, 3D, video I/O, and other operations. This chapter describes the programming environment and software interface to these registers/commands. The registers and commands themselves are described elsewhere in this document.

5.1 Programming Model

The graphics device is programmed via the following three basic mechanisms:

POST-Time Programming of Configuration Registers

These registers are the graphics device registers residing in PCI space. A majority of these registers are programmed once during POST of the video device. Configuration registers are not covered in this section. For details on accessing the graphics device’s configuration space see the EDS.

Direct (Physical I/O and/or Memory-Mapped I/O) Access of Graphics Registers

Various graphics functions can only be controlled via direct register access. In addition, direct register access is required to initiate the (asynchronous) execution of graphics command streams. This programming mechanism is “direct” and synchronous with software execution on the CPU.

Command Stream DMA (via the Command Ring Buffer and Batch Buffers)

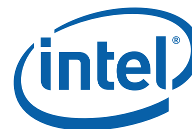
This programming mechanism utilizes the indirect and asynchronous execution of graphics command streams to control certain graphics functions, e.g., all 2D, 3D drawing operations. Software writes commands into a command buffer (either a Ring Buffer or Batch Buffer) and informs the graphics device (using the Direct method above) that the commands are ready for execution. The graphics device’s Command Parser (CP) will then, or at some point in the future, read the commands from the buffer via DMA and execute them.

5.2 Graphics Device Register Programming

The graphics device registers (except for the Configuration registers) are memory mapped. The base address of this 512 KB memory block is programmed in the MMADR Configuration register. For a detailed description of the register map and register categories, refer to the *Register Maps* chapter.

Programming Note:

Software must only access GR06, MSR0, MSR1, and Paging registers (see *Register Maps*) via Physical I/O, never via Memory Mapped I/O.



5.3 Graphics Device Command Streams

This section describes how command streams can be used to initiate and control graphics device operations.

5.3.1 Command Use

Memory-resident commands are used to control drawing engines and other graphics device functional units:

- Memory Interface (MI) Commands. The MI commands can be used to control and synchronize the command stream as well as perform various auxiliary functions (e.g., perform display/overlay flips, etc.)
- 2D Commands (BLT). These commands are used to perform various 2D (Blt) operations.
- 3D Commands. 3D commands are used to program the 3D pipeline state and perform 3D rendering operations. There are also a number of 3D commands that can be used to accelerate 2D and video operations, e.g., "StretchBlt" operations, 2D line drawing, etc.
- Video (MPEG) Decode Commands. A set of commands are supported to perform video decode acceleration including Motion Compensation operations via the Sampling Engine of the 3D pipeline.

5.3.2 Command Transport Overview

Commands are not written directly to the graphics device – instead they are placed in memory by software and later read via DMA by the graphics device's Command Parser (CP) within the Memory Interface function. The primary mechanism used to transport commands is through the use of a Ring Buffer.

An additional, indirect mechanism for command transport is through the use of Batch Buffers initiated from the Ring buffer.

The Command Parser uses a set of rules to determine the order in which commands are executed. Following sections in this chapter provide descriptions of the Ring Buffer, Batch Buffers, and Command Parser arbitration rules.

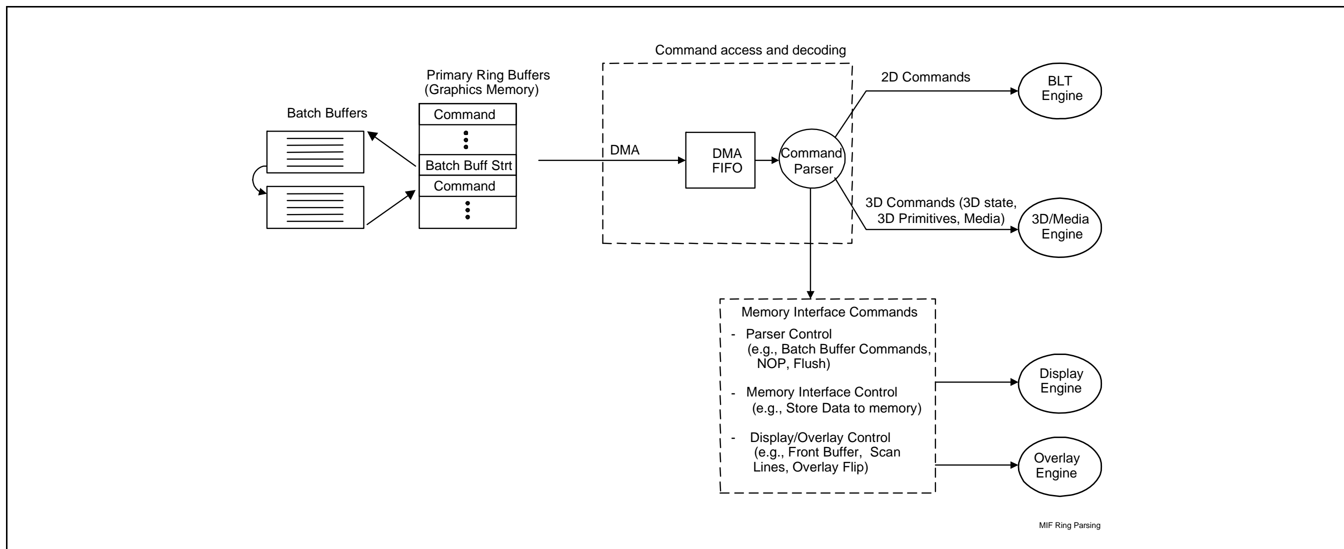
5.3.3 Command Parser

The graphics device's Command Parser (CP) is responsible for:

- Detecting the presence of commands (within the Ring Buffer).
- Reading commands from the Ring Buffer and Batch Buffers via DMA. This includes support of the automatic head report function.
- Parsing the common "Command Type" (destination) field of commands.
- Execution of Memory Interface commands that control CP functionality, provide synchronization functions, and provide display and overlay flips as well as other miscellaneous control functions.
- Redirection of 2D, 3D and Media commands to the appropriate destination (as qualified by the INSTPM register) while enforcing drawing engine concurrency and coherency rules.
- Performing the "Sync Flush" mechanism
- Enforcing the Batch Buffer protection mechanism

Figure 5-1 is a high-level diagram of the graphics device command interface.

Figure 5-1. Graphics Controller Command Interface



5.3.4 The Ring Buffer

The ring buffer is defined by a set of Ring Buffer registers and a memory area that is used to hold the actual commands. The Ring Buffer registers (described in full below) define the start and length of the memory area, and include two "offsets" (head and tail) into the memory area. Software uses the Tail Offset to inform the CP of the presence of valid commands that must be executed. The Head Offset is incremented by the CP as those commands are parsed and executed. The list of commands can wrap from the bottom of the buffer back to the top. Also included in the Ring Buffer registers are control fields that enable the ring and allow the head pointer to be reported to cacheable memory for more efficient flow control algorithms.

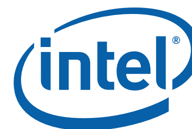
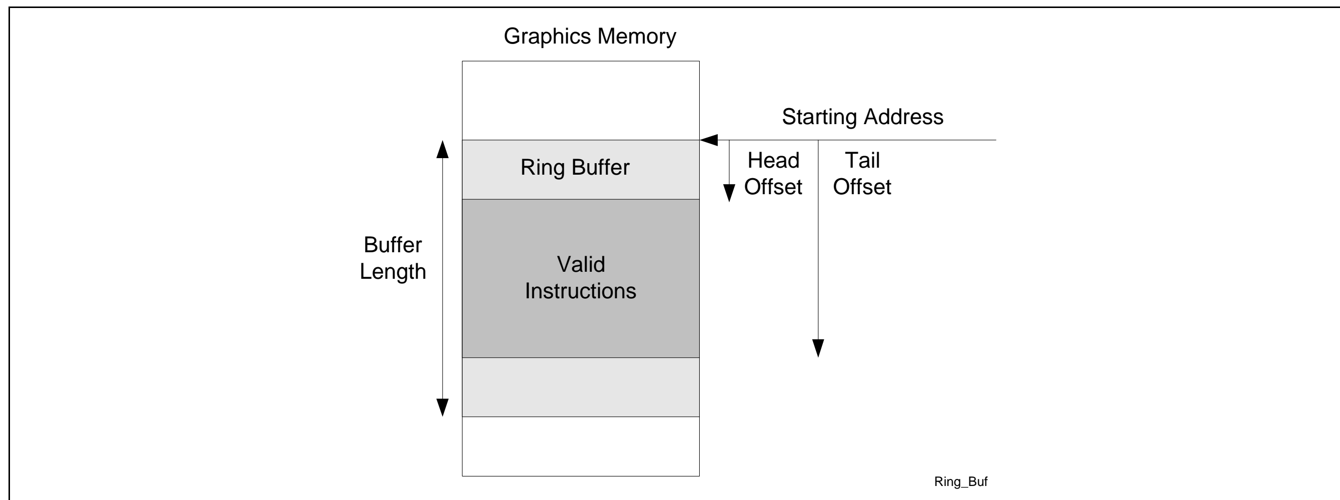


Figure 5-2. Ring Buffer



5.3.4.1 The Ring Buffer (RB)

Ring Buffer support:

Batch Buffer initiation

Indirect Data (operand access)

5.3.4.2 Ring Buffer Registers

A Ring Buffer is defined by a set of 4 Ring Buffer registers. Before a Ring Buffer can be used for command transport, software needs to program these registers. The fields contained within these registers are as follows:

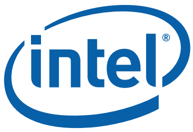
Ring Buffer Valid: This bit controls whether the Ring Buffer is included in the command arbitration process. Software must program all other Ring Buffer parameters before enabling a Ring Buffer. Although a Ring Buffer can be enabled in the non-empty state, it must not be disabled unless it is empty. Attempting to disable a Ring Buffer in the non-empty state is UNDEFINED. Enabling or disabling a Ring Buffer does not of itself change any other Ring Buffer register fields.

Start Address: This field points to a contiguous, 4KB-aligned, linear (i.e., must not be tiled), mapped graphics memory region which provides the actual command buffer area. Writing the Start Address has the side effect of clearing the Head Offset and Head Wrap Count fields.

Buffer Length: The size of the buffer, in 4KB increments, up to 2MB.

Head Offset: This is the DWord offset (from Start Address) of the next command that the CP will parse (i.e., it points one DWord past the last command parsed). The CP will update this field as commands are parsed – the CP typically continues parsing new commands before the previous command operations complete. (Note that, if commands are pending execution, the CP will likely have prefetched commands past the Head Offset). As the graphics device does not "reset" the Head Offset when a Ring Buffer is enabled, software must program the Head Offset field before enabling the Ring Buffer. Software can enable a Ring Buffer with any legal values for Head/Tail (i.e., can enable the Ring Buffer in an non-empty state). It is anticipated, but not required, that software enable The Ring Buffer with Head and Tail Offsets of 0. Once the Head Offset reaches the QWord specified by the Tail Offset (i.e., the offsets are equal), the CP considers the Ring Buffer "empty".

Head Wrap Count: This field is incremented by the CP every time the Head Offset wraps back to the start of the buffer. As it is included in the DWord written in the "report head" process, software can use this



field to track CP progress as if the Ring Buffer had a "virtual" length of 2048 times the size of the actual physical buffer (up to 4GB).

Tail Offset: This is the offset (from Start Address) of the next QWord of command data that software will request to be executed (i.e., it points one DWord past the last command DWord submitted for execution). The Tail Offset can only point to an command boundary – submitting partial commands is UNDEFINED. As the Tail Offset is a QWord offset, this requires software to submit commands in multiples of QWords (both DWords of the last QWord submitted must contain valid command data). Software may therefore need to insert a "pad" command to meet this restriction. After writing commands into the Ring Buffer, software updates the Tail Offset field in order to submit the commands for execution (by setting it to the QWord offset past the last command). The commands submitted can wrap from the end of the buffer back to the top, in which case the Tail Offset written will be less than the previous value. As the "empty" condition is defined as "Head Offset == Tail Offset", the largest amount of data that can be submitted at any one time is one QWord less than the Ring Buffer length.

IN USE Semaphore Bit: This bit (included in the Tail Pointer register) is used to provide a HW semaphore that SW can use to manage access to the individual The Ring Buffer. See the Ring Buffer Semaphore section below.

Automatic Report Head Enable: Software can request to have the hardware Head Pointer register contents written ("reported") to snooped system memory on a periodic basis. Auto-reports can be programmed to occur whenever the Head Offset crosses either a 64KB or 128KB boundary. (Note therefore that a Ring Buffer must be at least 64KB in length for the auto-report mechanism to be useful). The complete Head Pointer register will be stored at a Ring Buffer-specific DWord offset into the "hardware status page" (defined by the HWSTAM register). The auto-report mechanism is desirable as software needs to use the Head Offset to determine the amount of free space in the Ring Buffer -- and having the Head Pointer periodically reported to system memory provides a fairly up-to-date Head Offset value automatically (i.e., without having to explicitly store a Head Pointer value via the MI_REPORT_HEAD command).

Table 5-1. Ring Buffer Characteristics

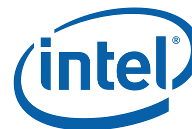
Characteristic	Description
Alignment	4 KB page aligned.
Max Size	2 MB
Length	Programmable in numbers of 4 KB pages.
Start Pointer	Programmable 4KB page-aligned address of the buffer
Head pointer	Hardware maintained DWord Offset into the ring buffer. Commands can wrap. Programmable to initially set up ring.
Tail pointer	Programmable QWord Offset into the ring buffer – indicating the *next* QWord where software can insert new commands.

5.3.4.3 Ring Buffer Placement

Ring Buffer memory buffers are defined via a Graphics Address and must physically reside in (uncached) Main Memory. There is no support for The Ring Buffer in cacheable system memory.

5.3.4.4 Ring Buffer Initialization

Before initializing a Ring Buffer, software must first allocate the desired number of 4KB pages for use as buffer space. Then the Ring Buffer registers associated with the Ring Buffer can be programmed. Once the Ring Buffer Valid bit is set, the Ring Buffer will be considered for



command arbitration, and the Head and Tail Offsets will either indicate an empty Ring Buffer (i.e., Head Offset == Tail Offset), or will define some amount of command data to be executed.

5.3.4.5 Ring Buffer Use

Software can write new commands into the "free space" of the Ring Buffer, starting at the Tail Offset QWord and up to the QWord prior to the QWord indicated by the Head Offset. Note that this "free space" may wrap from the end of the Ring Buffer back to the start (hence the "ring" in the name).

While the "free space" wrap may allow commands to be wrapped around the end of the Ring Buffer, the wrap should only occur between commands. Padding (with NOP) may be required to follow this restriction.

Software is required to use some mechanism to track command parsing progress in order to determine the "free space" in the Ring Buffer. This can be accomplished in one of the following ways:

1. A direct read (poll) of the Head Pointer register. This gives the most accurate indication but is expensive due to the uncached read.
2. The automatic reporting of the Head Pointer register in the Hardware Status Page. This has low impact as no uncached reads or command overhead is involved. However, given the 64KB/128KB granularity of auto-reports, this mechanism only works well on fairly large The Ring Buffer.
3. The explicit reporting of the Head Pointer register via the MI_REPORT_HEAD command. This allows for flexible and more accurate reporting but comes at the cost of command bandwidth and execution time, in addition to the software overhead to determine how often to report the head.
4. Some other "implicit" means by which software can determine how far the CP has progressed in retiring commands from a Ring Buffer. This could include the use of "Store DWORD" commands to write sequencing data to system memory. This has similar characteristics to using the MI_REPORT_HEAD mechanism.

Once the commands have been written and, if necessary, padded out to a QWord, software can write the Tail Pointer register to submit the new commands for execution. The uncached write of the Tail Pointer register will ensure that any pending command writes are flushed from the processor.

If the Ring Buffer Head Pointer and the Tail Pointer are on the same cacheline, the Head Pointer must not be greater than the Tail Pointer.

5.3.4.6 Ring Buffer Semaphore

When the **Ring Buffer Mutex Enable** (RBME) bit if the INSTPM MI register is clear, all Tail Pointer IN USE bits are disabled (read as zero, writes ignored). When RBME is enabled, the IN USE bit acts as a Ring Buffer semaphore. If the Tail Pointer is read, and IN USE is clear, it is immediately set after the read. Subsequent Tail Pointer reads will return a set IN USE bit, until IN USE is cleared by a Tail Pointer write.

This allows SW to maintain exclusive ring access through the following protocol: A SW agent needing exclusive ring access must read the Tail Pointer before accessing the Ring Buffer: if the IN USE bit is clear, the agent gains access to the Ring Buffer; if the IN USE bit is set, the agent has to wait for access to the Ring Buffer (as some other agent has control). The mechanism to inform pending agents upon release of the IN USE semaphore is unspecified (i.e., left up to software).

5.3.5 Batch Buffers

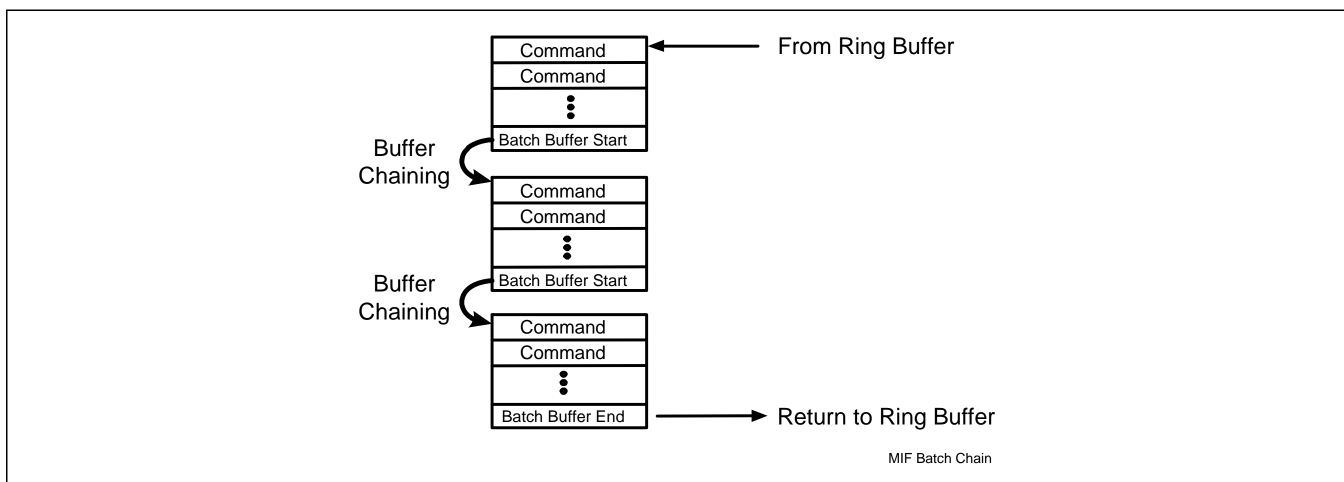
The graphics device provides for the execution of command sequences *external* to the Ring buffer. These sequences are called "Batch Buffers", and are initiated through the use of various Batch Buffer commands described below. When a Batch Buffer command is executed, a batch buffer sequence is initiated, where the graphics device fetches and executes the commands sequentially via DMA from the batch buffer memory.

5.3.5.1 Batch Buffer Chaining

What happens when the end of the Batch Buffer is reached depends on the final command in the buffer. Normally, when a Batch Buffer is initiated from a Ring Buffer, the completion of the Batch Buffer will cause control to pass back to the Ring Buffer at the command following the initiating Batch Buffer command.

However, the final command of a Batch Buffer can be another Batch Buffer-initiating command (MI_BATCH_BUFFER_START). In this case control will pass to the new Batch Buffer. This process, called *chaining*, can continue indefinitely, terminating with a Batch Buffer that does not chain to another Batch Buffer (ends with MI_BATCH_BUFFER_END) – at which point control will return to the Ring Buffer.

Figure 5-3. Batch Buffer Chaining



5.3.5.2 Ending Batch Buffers

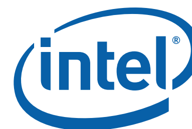
The end of the Batch Buffer is determined as the buffer is being executed: either by (a) an MI_BATCH_BUFFER_END command, or (b) a "chaining" MI_BATCH_BUFFER_START command. There is no explicit limit on the size of a Batch Buffer that uses GTT-mapped memory. Batch buffers in physical space cannot exceed one physical page (4KB).

5.3.6 Indirect Data

In addition to Ring Buffer and Batch Buffers, the MI supports the access of *indirect* data for some specific command types. (Normal read/write access to surfaces isn't considered indirect access for this discussion).

5.3.6.1 Logical Contexts

Logical Contexts, indirectly referenced via the MI_SET_CONTEXT command, must reside in (unsnooped) Main Memory.



5.3.7 Command Arbitration

The command parser employs a set of rules to arbitrate among these command stream sources. This section describes these rules and discusses the reasoning behind the algorithm.

5.3.7.1 Arbitration Policies and Rationale

The Ring buffer (RB) is considered the primary mechanism by which drivers will pass commands to the graphics device.

The insertion of command sequences into the Ring Buffer must be a "synchronous" operation, i.e., software must guarantee mutually exclusive access to the Ring Buffer among contending sources (drivers). This ensures that one driver does not corrupt another driver's partially-completed command stream. There is currently no support for unsynchronized multi-threaded insertion of commands into ring buffer.

Another requirement for asynchronous command generation arises from competing (and asynchronous) drivers (e.g., "user-mode" driver libraries). In this case, the desire is to allow these entities to construct command sequences in an asynchronous fashion, via batch buffers. Synchronization is then only required to "dispatch" the batch buffers via insertion of Batch Buffer commands inserted into the Ring Buffer.

Software retains some control over this arbitration process. The MI_ARB_ON_OFF command disables all other sources of command arbitration until re-enabled by a subsequent MI_ARB_ON_OFF command from the same command stream. This can be used to define uninterruptible "critical sections" in an command stream (e.g., where some device operation needs to be protected from interruption). Disabling arbitration from a batch buffer without re-enabling before the batch is complete is UNDEFINED.

Batch Buffers can be (a) interruptible at command boundaries, (b) interruptible only at chain points, or (c) non-interruptible. See MI_BATCH_BUFFER_START in *Memory Interface Commands* for programming details.

5.3.7.2 Wait Commands

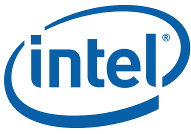
The MI_WAIT_EVENT command is provided to allow command streams to be held pending until an asynchronous event occurs or condition exists. An *event* is defined as occurring at a specific point in time (e.g., the leading edge of a signal, etc.) while a *condition* is defined as a finite period of time. A wait on an event will (for all intents and purposes) take some non-zero period of time before the subsequent command can be executed. A wait on a condition is effectively a noop if the condition exists when the MI_WAIT_EVENT command is executed.

A Wait in the Ring Buffer or batch buffer will cause the CP to treat the Ring Buffer as if it were empty until the specific event/condition occurs. This will temporarily stall the Ring Buffer.

While the Ring Buffer is waiting, the **RB Wait** bit of the corresponding RB*n*_CTL register will be set. Software can cancel the wait by clearing this bit (along with setting the **RB Wait Write Enable** bit). This will terminate the wait condition and the Ring Buffer will be re-enabled. This sequence can be included when software is required to flush all pending device operations and pending Ring Buffer waits cannot be tolerated.

5.3.7.3 Wait Events/Conditions

This section describes the wait events and conditions supported by the MI_WAIT_EVENT command. Only one event or condition can be specified in an MI_WAIT_EVENT, though different command streams can be simultaneously waiting on different events.



5.3.7.3.1 Display Pipe A,B Vertical Blank Event

The Vertical Blank event is defined as "shortly after" the *leading edge* of the next display VBLANK period of the corresponding display pipe. The delay from the leading edge is provided to allow for internal device operations to complete (including the update of display and overlay status bits, and the update of overlay registers).

5.3.7.3.2 Display Pipe A,B Horizontal Blank Event

The Horizontal Blank event is defined as "shortly after" the *leading edge* of the next display HBLANK period of the corresponding display pipe.

5.3.7.3.3 Display Plane A, B, C , Flip Pending Condition

The Display Flip Pending condition is defined as the period starting with the execution of a "flip" (MI_DISPLAY_BUFFER_INFO) command and ending with the completion of that flip request. Note that the MI_DISPLAY_BUFFER_INFO command can specify whether the flip should be synchronized to vertical refresh or completed "as soon as possible" (likely some number of horizontal refresh cycles later).

5.3.7.3.4 Overlay Flip Pending Condition

The Overlay Flip Pending condition is similar to the Display Flip Pending condition, with the exception that overlay flips are only performed synchronously with display refresh.

5.3.7.3.5 Display Pipe A,B Scan Line Window Conditions

The graphics device supports two conditions relating to the progress of refresh within a particular display stream. A "Scan Line Window" is defined as the period of time between the refresh of two specific display scan lines. The MI_WAIT_ON_EVENT command can be used to pause an command stream while a particular display refresh is inside or outside the Scan Line Window. (Actually, the MI_WAIT_EVENT command only supports waiting on the Scan Line Window condition, and the MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL are used to define an "inclusive" or "exclusive" window).

If no Scan Line Window has been defined for the particular display stream, the MI_WAIT_EVENT specifying the Scan Line Window event will never introduce a wait.

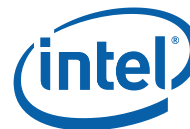
5.3.7.3.6 Semaphore Wait Condition

One of the 8 defined condition codes contained within the Execute Condition Code (EXCC) Register can be selected as the source of a wait condition. While the selected condition code bit is set, the initiating command stream will be removed from arbitration (i.e., paused). Arbitration of that command stream will resume once the condition code bit is clear. If the selected condition code is clear when the WAIT_ON_EVENT is executed, the command is effectively ignored.

5.3.7.4 Command Arbitration Points

The CP performs arbitration for command execution at the following points:

- Upon execution of an MI_ARB_CHECK command
- When the ring buffer becomes empty



5.3.7.5 Command Arbitration Rules

At an arbitration point, the CP will switch to the new head pointer contained in the UHPTR register if it is valid. Otherwise it will idle if empty, or continue execution in the current command flow if it arbitrated due to an MI_ARB_CHECK command.

5.3.7.6 Batch Buffer Protection

The CP employs a protection mechanism to help prevent random writes to system memory from occurring as a result of the execution of a batch buffer generated by a "non-secure" agent (e.g., client-mode library). Commands executed directly from a ring buffer, along with batch buffers initiated from a ring buffer and marked as "secure", will not be subject to this protection mechanism as it is assumed they can only be generated by "secure" driver components.

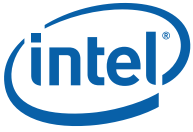
This protection mechanism is enabled via a field in a Batch Buffer command that indicates whether the associated batch buffer is "secure" or "non-secure". When the CP processes a non-secure batch buffer from the ring buffer it does not allow any MI_STORE_DATA_IMM commands that reference physical addresses, as that would allow the non-secure source to perform writes to any random DWord in the system. (Note that graphics engines will only write to graphics memory ranges, which by definition are virtual memory ranges mapped into physical memory pages by trusted driver components using the GTT/TGTT hardware). Placing an MI_STORE_DATA in a non-secure batch buffer will instead cause a Command Error. The CP will store the header of the command, the origin of the command, and an error code. In addition, such a Command Error can generate an interrupt or a hardware write to system memory (if these actions are enabled and unmasked in the IER and IMR registers respectively.) At this point the CP can be reactivated only by a **full reset**.

The security indication field of Batch Buffer instructions placed in batch buffers (i.e., "chaining" batch buffers) is ignored and the chained batch buffer will therefore inherit the security indication of the first Batch Buffer in the chain (i.e. the batch buffer that was initiated by an MI_BATCH_BUFFER_START command in the Ring Buffer).

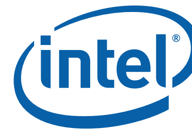
5.3.8 Graphics Engine Synchronization

This table lists the cases where engine synchronization is required, and whether software needs to ensure synchronization with an explicit MI_FLUSH command or whether the device performs an implicit (automatic) flush instead. Note that a pipeline flush can be performed without flushing the render cache, but not vice versa.

Event	Implicit Flush or Requires Explicit Flush?
PIPELINE_SELECT	Requires explicit pipeline flush
Any Non-pipelined State Command	Device implicitly stalls the command until the pipeline has drained sufficiently to allow the state update to be performed without corrupting work-in-progress
MI_SET_CONTEXT	Device performs implicit flush
MI_DISPLAY_BUFFER_INFO ("display flip")	Requires explicit render cache flush
MI_OVERLAY_FLIP	Requires explicit render cache flush



Event	Implicit Flush or Requires Explicit Flush?
3D color destination buffer (render target) used as texture (i.e., “rendered texture”)	Requires explicit render cache flush
MEDIA_STATE_POINTERS	Requires explicit pipeline flush
MEDIA_OBJECT	Requires explicit pipeline flush
Media: Previous Destination Used as Source	Requires explicit render cache flush



5.3.9 Graphics Memory Coherency

Table 5-2. Graphics Memory Coherency lists the various types of graphics memory coherency provided by the device, specifically where the CPU writes to a 64B cacheline, and the device then accesses that same cacheline. Note that the coherency policy depends on the address type (GM or MM) involved in the accesses.

Table 5-2. Graphics Memory Coherency

CPU Access	Subsequent Device Access	Example Operand	Coherency
Write GM	Read GM		TBD
Write MM	Read MM	Batch Buffer	TBD
Write GM	Write GM		Device ensures coherency following every Ring Buffer Tail Pointer write. (This can be made optional via a bit in the Tail Pointer data).
Write MM	Write MM		TBD “assumed to exclusive byte” ?
Write GM	Read MM		Device ensures coherency following every Ring Buffer Tail Pointer write. (This can be made optional via a bit in the Tail Pointer data).

5.3.10 Graphics Cache Coherency

There are several caches employed within the graphics device implementation. This section describes the impact of these caches on the programming model (i.e., if/when does software need to be concerned).

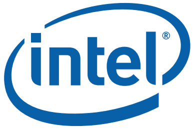
5.3.10.1 Rendering Cache

The rendering (frame buffer) cache is used by the blit and 3D rendering engines and caches portions of the frame buffer color and depth buffers. This cache is guaranteed to be flushed under the following conditions (note that the implementation may flush the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command with the **Render Flush Cache Inhibit** bit clear
- Execution of a PIPE_CONTROL instruction with the **Write Cache Flush Enable** bit set (Depth Stall must be clear).
- A SyncFlush handshake operation
- A change of rendering engines (e.g., going from 2D to 3D, 3D to 2D, etc.)
- Logical Context switch (via MI_SET_CONTEXT) The render cache must be explicitly flushed using one of these mechanisms under certain conditions. See Graphics Engine Synchronization above.

5.3.10.2 Sampler Cache

The read-only sampler cache is used to cache texels and other data read by the Sampling Engine in the 3D pipeline. This cache can be enabled or disabled via the **Texture L2 Disable** bit of the Cache_Mode_0 register (see *Memory Interface Registers*). Note that, although there may be more than one level of sampler cache within the implementation, the sampler cache is exposed as a single entity at the programming interface.



The sampler cache is guaranteed to be invalidated under the following conditions (note that the implementation may invalidate the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command with the **Map Cache Invalidate** bit set
- Execution of PIPE_CONTROL with the **Depth Stall Enable** bit clear.
- A SyncFlush handshake operation

The sampler cache must be invalidated prior to reallocation of physical texture memory (i.e., software must guarantee that stale texture data is invalidated before reusing physical texture memory for a new or modified texture).

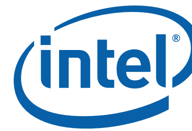
5.3.10.3 Instruction/State Cache

The read-only ISC is used to cache pipelined state and EU instructions read in from memory. It also functions as a prefetch cache by reading in additional state information and instructions beyond those immediately requested in order to decrease latency and improve performance. As with the sampler cache, there may be more than one level of ISC within the implementation. The ISC is exposed as a single entity at the programming interface.

The instruction/state cache is guaranteed to be invalidated under the following conditions (note that the implementation may invalidate the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command with the **State/Instruction Cache Invalidate** bit set
- Execution of PIPE_CONTROL with the Instruction/State Cache Flush Enable bit set.
- A SyncFlush handshake operation

The instruction/state cache must be invalidated prior to reallocation of physical state/instruction memory (i.e., software must guarantee that stale state/instruction data is invalidated before reusing physical state/instruction memory for new or modified state or instructions).



5.3.10.4 Vertex Cache

The vertex cache consists of 2 sub-caches: one that caches vertex buffer data based on address, and another that caches (possibly shaded) vertex attribute data based on index (see the *Vertex Fetch* chapter for vertex index details). The latter cache is always invalidated between primitive topologies.

Both vertex caches are guaranteed to be invalidated under the following conditions (note that the implementation may invalidate the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command
- Execution of a PIPE_CONTROL command
- A SyncFlush handshake operation
- Logical Context switch (via MI_SET_CONTEXT)

5.3.10.5 GTT TLBs

The following table summarizes when the various TLBs are invalidated.

TLB	Normal Invalidation Mechanism
Display	Refreshed on Vsync
Overlay	Refreshed on Vsync
Render/Blit	Internal Flush*
Host	Through a Page Table PTE write
Sampler Cache	Internal Flush*
Command Stream	Through a Page Table PTE write

* -- Includes MI_FLUSH, Engine switch, and Context switch.

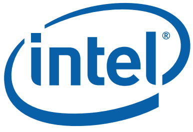
5.3.11 Command Synchronization

This section describes the hardware mechanisms that can be used by software to provide synchronization with command stream parsing and execution.

The key point here is distinguishing between command *parsing* and *retirement* – in that, for most commands, there is some finite delay between the parsing of a command and the retirement (coherent completion) of the operation it specifies.

Interrogation of the Ring Buffer Head Pointer only gives an indication of the progress of command parsing. This information is required to discern the availability of command data within the Ring Buffer or Batch Buffers. If the Head Pointer indicates the command data has been parsed, those locations can be reused; otherwise the commands must be considered still pending parsing and left alone.

Given the CP rules for command execution, it is possible to use the indication of command parsing progress to infer the retirement status of parsed commands. The only indication of instruction retirement available from instruction parsing is that parsing of an MI instruction implies retirement of previous MI instructions with the following exceptions:



- The parsing of a Memory Interface (MI) command implies that all previously-parsed MI commands have completed, with the following exceptions:
    - Display and Overlay Flip commands: Only the submission of the flip request is guaranteed. The flip operation will occur some time later. Mechanisms to detect the actual completion of a flip operation are described below.
    - "Store-Data" type commands: Only the submission of the store operation is guaranteed. The write result will be complete (coherent) some time later (this is practically a finite period but there is no guaranteed latency).
    - Batch Buffer commands: There is no guarantee that the operations performed by the batch buffer have completed.

Other than the cases described above, additional measures must be taken to discern the progress of command retirement. These measures are described in the following subsections.

5.3.11.1 MI_FLUSH

The MI_FLUSH command pauses further command parsing until all drawing engines become idle and any internal rendering cache is flushed and invalidated. All previous rendering commands can therefore be considered retired.

This flush operation is considered complete once command parsing proceeds to the next command. Software can, for example, follow an MI_FLUSH command with an MI_STORE_DATA_IMM or MI_STORE_DATA_INDEX command – where the completion of the store operation implies that the flush operation has completed. (Note that if the last DWord in a ring buffer is an MI_FLUSH instruction, there is no way by simply looking at the Ring Buffer registers to determine whether the flush operation is complete or still pending.)

The successful completion of an MI_FLUSH command does not guarantee that *all* previous operations have completed. Operations that may still be pending include:

- Store Data type commands (MI_STORE_DATA_IMM, MI_STORE_DATA_INDEX, MI_REPORT_HEAD)
- Display or Overlay Flip operations

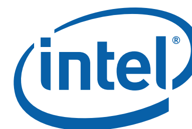
See section 5.3.10.2 for more information on when the sampler cache should be invalidated.

5.3.11.2 Sync Flush

Inserting MI_FLUSH commands, while effective at determining or forcing the retirement of previous rendering commands, may negatively impact performance if not absolutely required. For example, if the knowledge of rendering command retirement is not known a priori, it is likely undesirable to insert MI_FLUSH commands at intervals in the command stream. However, it may not be acceptable to insert an MI_FLUSH command (and wait for its completion) at the point that rendering command retirement is required – as there may be a large number of commands pending in ring/batch buffers at that point and flushing the entire device (including waiting for completion of pending commands that have not yet been parsed) may be prohibitive. There is a mechanism, however, where command stream synchronization can be performed on demand, without requiring earlier submitted commands and batch buffers to complete – it is called the "Sync Flush" mechanism.

Here's how it works:

- Software must (preferably at driver initialization time) unmask the Sync Status bit in the Hardware Status Mask Register (HWSTAM). This should be done unconditionally (at least whenever HW status writes are enabled), as any bandwidth increase due to Sync Status-initiated writes is negligible.



- At the point that synchronization is required, software must guarantee that command parsing has progressed past the point of interest in the command stream (i.e., past the last command whose retirement is required). Note that this step is required in any scheme.
- Software then reads the location where the Interrupt Status is reported in the Hardware Status Page (DWord offset 0) and saves that DWord in a temporary variable.
- Software then sets the Sync Enable bit of the Command Parser Mode Register (INSTPM) via an uncached write.
- The Command Parser will detect the Sync Enable bit set before it proceeds to the very next command (or immediately if the CP is idle). It will then perform an internal flush operation. This flush is identical to that performed by an MI_FLUSH command with all flush types enabled.
- Once this flush operation is complete, the CP will clear the Sync Enable bit of the INSTPM register and then *toggle* the Sync Status bit of the ISR register. This will initiate a write of the ISR register contents (with the toggled Sync Status) to DWord 0 of the Hardware Status page (as part the normal hardware status write mechanism).
- Software, following the write of the INSTPM register, should periodically poll the Hardware Status location. By comparing the current versus saved value of the Sync Status bit, software can then detect when the flush operation is complete. Note that the latency of this operation is typically small, as it will be initiated either immediately or at least before the next command is parsed (regardless of arbitration conditions).

5.4 Hardware Status

The graphics device supports a number of internal hardware status bits which can be used to detect and monitor hardware status conditions via polling or interrupts. This section will describe each hardware status bit. The following section describes the hardware status reporting (polling) mechanism. The mechanism to allow these status bits to generate interrupts is described in the Interrupts section. Note that the hardware status bits are actually reported in the Interrupt Status Register, so “hardware status” and “interrupt status” are used interchangeably here (though many hardware status bits won’t necessarily ever be used to generate interrupts).

The following subsections describe the various hardware (interrupt) status bits, as defined in the Interrupt Status Register.

5.4.1 Hardware-Detected Errors (Master Error bit)

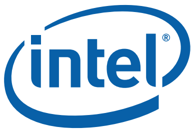
This interrupt status bit is generated whenever an “unmasked” hardware-detected error status is detected. See Errors.

5.4.2 Thermal Sensor Event

This interrupt status bit is generated by “thermal events” detected by the Thermal Sensor logic. The bit corresponding to this event in the HWSTAM register must always be masked (i.e., set to ‘1’) so that thermal sensor events do not generate HW status DWord writes. See Hardware Status Writes.

5.4.3 Sync Status

This bit should only be used as described in Sync Flush, and should not be used to generate interrupts (i.e., the corresponding interrupt should not be enabled in the IER).



5.4.4 Display Plane A, B, (Sprite A, Sprite B [DevCTG] Only) Flip Pending

These bits are used to report the status of "flip" operations on the corresponding Display Plane. Display Flip operations are requested via the MI_DISPLAY_BUFFER_INFO command. When that command is executed, the corresponding Display Flip Pending status in the ISR register will be set to '1' indicating that a display flip has been requested but has not yet been performed. (Requesting a flip operation when one is already pending is UNDEFINED). This indicates that a flip is "pending". At the appropriate time during the next vertical blank period (for that display stream), the flip operation will be performed (i.e., the display will switch to refreshing from the new display buffer). This causes the Display Flip Pending status to reset to '0'. When this occurs, and the Display Flip Pending status bit is unmasked by the Interrupt Mask Register (IMR), the Display Flip Pending status bit of the Interrupt Identity Register (IIR) is set. <u>Note that this setting of an interrupt identity bit on the falling edge of the status bit is contrary to the general definition of interrupt status bits</u>.

5.4.5 Overlay Flip Pending

This bit is similar to the Display Flip Pending bits. It is set to '1' when the MI_OVERLAY_FLIP command is executed. It is cleared to '0' after the overlay registers are read from memory during the next vertical blanking period.

5.4.6 Display Pipe A,B VBLANK

These bits are set on the leading edge of the selected Display Pipe's VBLANK signal.

5.4.7 User Interrupt

This bit is set in response to the execution of an MI_USER_INTERRUPT command. The Command Parser will continue parsing after processing that command. If a user interrupt is currently outstanding (set in the ISR) this packet has no effect.

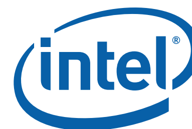
Programming Note: User interrupts can be used to notify software of the progress of instruction parsing past the MI_USER_INTERRUPT instruction. In particular, user interrupts can be inserted into the command stream but effectively disabled for "normal operation" via the IMR and HWSTAM registers. Whenever software requires the notification afforded by the user interrupts, it can unmask this bit.

5.4.8 PIPE_CONTROL Notify Interrupt

This bit is set when a PIPE_CONTROL command with the **Notify Enable** bit set reaches the end of the pipeline and all required cache flushes have occurred.

5.5 Hardware Status Writes

The graphics device supports the writing of the hardware status (ISR) bits into memory for optimized access from software. Software can select which (if any) status bits will trigger the write of the ISR contents to memory using the Hardware Status Mask (HWSTAM) register. Writing a '0' to a defined bit position in the HWSTAM register will cause any change (0 → 1 or 1→ 0) in the corresponding ISR bit to trigger the write. The complete ISR contents will be written to DWord offset 0 of the hardware status page, located at the address programmed via the Hardware Status Page Address Register (HWS_PGA).

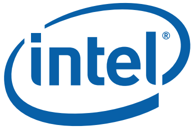


5.6 Interrupts

The graphics device supports the generation of an interrupt. This interrupt can be raised in response to one or more internal interrupt status conditions. Which interrupt status conditions are allowed to raise an interrupt is programmed via the Interrupt Mask Register (IMR) and Interrupt Enable Register (IER). The IMR is used to selectively "unmask" hardware status bits as to allow them to be reported in the Interrupt Identity Register (IIR). The IER holds a set of interrupt enable bits corresponding to each bit of the IIR – setting bits in the IER will allow interrupts to be generated by the corresponding bits in the IIR.

5.7 Errors

The graphics device supports the hardware detection of a number of *operational* and *debug-only* errors. Operational errors occur out of the immediate control of driver software and must be anticipated and tolerated to the extent required by the relevant APIs. Software must therefore support the detection and proper handling of all relevant operational errors. The (more numerous) debug-only errors are just that – detected to facilitate initial system debug but not intended to be tolerated during normal system operation. In many cases, debug-only errors are not recoverable. They require the use of debug registers to detect and diagnose.



5.7.1 Error Reporting

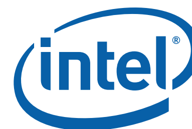
Regardless of the error classification, all errors funnel through the **Master Error** bit of the Interrupt Control Registers. This bit can be used to raise a device interrupt or trigger a hardware status write operation. (Needless to say it can also be polled directly, though this is clearly discouraged). Refer to Interrupt Control Registers in the *Memory Interface Registers* chapter for more information.

There are three registers dedicated to control, detect, and clear hardware error status conditions in a similar fashion to the Interrupt Control Registers. All three error registers share a common error status bit definition.

The Error Status Register (ESR) holds the actual error status bits (each of which may be the logical OR of "source" error bits in various functional registers). The Error Mask Register (EMR) is used to select which error status bit(s) are reported in the Error Identity Register (EIR). The EIR holds the "persistent" values of the unmasked error status bits, and is also used to clear error status conditions. Any bits set in the EIR will raise the Master Error interrupt status condition.

The error conditions corresponding to the error status bits include:

- **Page Table Error** (***Debug only***) – This is a summary of a number of possible errors associated with the mapping function of the GTT. See Table 5-3 for more information.
- **Display or Overlay Underrun** (***Debug only***) – This error is raised when a FIFO underrun condition is detected in the display or overlay isochronous streams. See the description of the Display/Overlay Status Register in the *Display Registers* chapter.
- **Command Error** (***Debug Only***) – This is a summary of a number of command data errors detected by the Command Parser. See Command Errors below for more information.



5.7.2 Page Table Errors

The following tables describe the various sources and types of Page Table Errors. Refer to the description of the PGTBL_ERR register in *Memory Interface Registers* for more details.

Table 5-3. Page Table Error Types

Error	Description	Streams
Invalid GTT PTE	In the process of mapping an address, the MI encountered a GTT PTE that was marked "Invalid". This would be the result of a programming error.	All (See **Error! Reference source not found.**)
Invalid TLB Miss	An unexpected TLB miss (detected at GTT request time) was encountered (e.g., during Display/Overlay/Sprite access).	Display, Overlay
Invalid PTE Data	Mapping to the physical page specified in the PTE is not permitted (e.g., a page in PAM, SMM or over the top of memory, etc.). This is the result of a programming error.	Host
Invalid Tiling	A tiling parameter was found inconsistent with the current operation. This includes the use of Y-Major tiling in the Render/Display/Overlay streams. This is the result of a programming error. This is detected during GTT request.	Blt, Display, Overlay

Note that Page Table Errors cannot be cleared. A device reset is required.

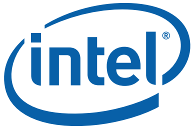
5.7.3 Clearing Errors

For operational errors, software is responsible for taking the proper steps to recover from the error and then clearing the error indication. The actions required to recover from operational errors may be discussed in the various functional areas (not here). See the Hardware-detected Error Bit Definitions in *Memory Interface Registers* for more details. This subsection describes the actions required to clear the error indication.

In order to clear operational errors, software is responsible for clearing the error condition from the source, working back to the Master Error bit. Typically this will entail the following sequence.

- First the primary source of the error must be cleared. This requires clearing the functional register(s) containing the source error indication.
- Next, clear the particular error status bit by writing a '1' to the appropriate bit of the Error Identity Register (EIR). This will clear the error status bit in the Error Status Register (ESR). If multiple errors are present, all error status bits should be cleared simultaneously.
- Next, clear the Master Error interrupt status bit by writing a '1' to the Master Error bit of the Interrupt Identity Register (IIR).

Note: Page Table Errors cannot be cleared.



5.8 Rendering Context Management

The graphics device operation (rendering, etc.) is controlled via the settings of numerous hardware state variables. These state variables are divided into *global state* and *context state*.

There is only one copy of global state variables, and changing the settings of these variables requires explicit programming of the state variables. Examples of global state include:

- MI registers (HWSTAM, Ring Buffer, etc.) with the exception of those listed in the next paragraph (i.e, registers listed there *are* saved/restored)
- Configuration registers
- Display programming registers

On the other hand, context state is associated with a specific *context*, where switching to that context causes that context's state to be restored. While the associated context is active, the state variables and registers can be programmed via the command stream. Examples of context state include the PIPELINE_STATE_POINTERS command and most non-pipelined state. The following MI registers are considered part of context state and thus saved/restored with context:

- INSTPM
- CACHE_MODE_0
- CACHE_MODE_1
- MI_ARB_STATE
- 3D Pipeline Statistics Registers

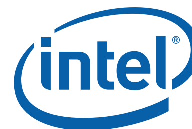
The graphics device supports both a *hardware context* and *logical contexts*. The multiple logical context support provides robust rendering context support by swapping contexts to/from memory.

5.8.1 Multiple Logical Rendering Contexts

The graphics device supports multiple *logical rendering contexts* stored in Main Memory. Logical rendering contexts are referenced via a 2KB-aligned *Logical Context Address*.

The maximum size of a logical context entry (which is information required by the driver to allocate contexts) is currently 2K bytes. For forward compatibility, the maximum size of a logical context entry should be supplied to the drivers via a VBIOS mechanism as opposed to being hardcoded in the driver.

The actual size of a logical rendering context is the amount of data stored/restored during a context switch and is measured in 64B cache lines. There is a debug mechanism that allows software/BIOS to program the actual size of the logical rendering context via the CXT_SIZE register. Note that this register will default to the correct value, so software should not have to modify it.



The format of the logical rendering context in memory is considered device-dependent; software must not attempt to modify the contents of a logical rendering context directly. This restriction is motivated by forward compatibility concerns because the location and definition of fields may change between implementations.

5.8.1.1 Current Context IDs

The ring buffer has an associated *Current Context ID* (CCID) register. The CCID includes a Logical Pipeline Context Address (LPCA).

The CCID for a ring buffer is set during the processing of the new MI_SET_CONTEXT command from that ring. The MI_SET_CONTEXT command provides a new CCID value (LPCA) to be loaded into the CCID register for the associated ring buffer. The MI_SET_CONTEXT command also contains a Restore Inhibit bit used to optionally inhibit the restoration (loading) of the new rendering context. This bit must be used during context initialization to avoid the loading of uninitialized (garbage) context data from memory. Failure to do so leads to UNDEFINED operation.

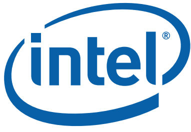
The initial values of the CCIDs are UNDEFINED. The first time a valid CCID is set from a ring buffer, the normal context save operation will be suppressed, as the previous CCID is invalid.

5.8.1.2 Intra-Ring Context Switch

Within a specific ring buffer, a new logical rendering context is specified via the MI_SET_CONTEXT command. Note that MI_SET_CONTEXT commands are permitted only within a ring buffer (not within a batch buffer).

As part of the execution of the MI_SET_CONTEXT command from within a ring buffer, the Logical Pipeline Context Address fields of the CCID register and MI_SET_CONTEXT command are compared. If they differ (or the CCID register is uninitialized), a rendering context switch operation will be performed, which includes:

1. If the CCID contents are valid, a context save operation will be performed. The contents of the HW context will be saved in memory starting at the Logical Pipeline Context Address specified in the CCID.
2. If the Restore Inhibit command field is not set, a context restore operation will be performed. Here the logical context values are read starting from the Logical Pipeline Context Address field of the command and used to set the internal HW context.
3. The relevant contents of the command will be loaded into the appropriate CCID register. (This occurs irrespective of the LPCA comparison result). At this point, the ring buffer has switched to using the new logical rendering context.



5.8.1.3 Logical Rendering Context Creation and Initialization

5.8.1.3.1 Rendering Context Creation Rules

1. Software only knows the **size** of the logical rendering context (2KB), for allocation purposes.
2. Given (1), software does **not** know the format of the context, and therefore is not allowed to write any portion of a logical rendering context. Software can, however, copy/move entire logical context blocks.
3. Given (2), software must never restore (load) a logical rendering context from memory that has not been previously stored by HW. I.e., software must never attempt to initialize a context itself and then cause it to be loaded. Breaking this rule causes UNDEFINED operation (as in the hang seen in BDG validation).
4. Initialization software must write **all** HW context variables with legal values before the first rendering context can be saved (this must be done before you can perform any rendering anyways). Given this, and the obvious rule that software must never program illegal state values, guarantees that the HW context will forever remain valid (and therefore be available to store into a logical rendering context). Note that software-visible context variables include 3D state, Blt register state, etc.

5.8.1.3.2 Context Initialization

Logical Rendering Contexts can be initialized (in memory) by software in the following way:

1. Issue an MI_SET_CONTEXT command w/ the **Restore Inhibit** bit set and the about-to-be-initialized logical pipeline context address. This will save the current rendering context and then change the LPCA to the new context (without loading it).
2. Use state commands to modify the context as desired.
3. Issue another MI_SET_CONTEXT command specifying some other LPCA (e.g., the previous one). This will cause the new context to be stored (initialized) in memory

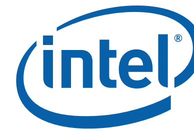
5.8.1.4 Context Save

A context save will occur anytime all of the following apply:

- A rendering context switch occurs as a result of the execution of MI_SET_CONTEXT
- The CCID of the current context (CCID register of current ring) and the new CCID (the CCID register of the newly selected ring or the new CCID in the MI_SET_CONTEXT command) differ OR an MI_SET_CONTEXT with the "Force Restore" bit set initiated the context switch
- The current CCID is valid (has been previously set)

The current rendering context will be written out to memory starting at the LPCA in the format described by Logical Context Layout in *Memory Data Formats*. Note that this includes a limited number of Memory Interface Registers whose values are saved by embedding them in an MI_LOAD_REGISTER_IMM command that is written out to memory.

The Optional Extended Context will also be written if the Extended Save Enable bit is set in the current CCID register. Context saves DO NOT modify pipelined state stored in memory.



5.9 Reset State

This section describes the state of the programming interface following a hardware reset. Refer to the individual register definitions for details on reset (default) settings.

- The settings of the hardware context state variables are UNDEFINED. Software must program all state variables prior to their use in rendering.
- The ring buffer is disabled.
- All interrupts and error status bits are "masked" (disabled). All interrupts are disabled via IER. There will be no HW activity to cause any hardware/interrupt status bits to be set.
- The Hardware Status Page is located at 1FFFF000h (though HW status writes are effectively disabled)
- All FENCE registers are INVALID
- The GTT is disabled (accesses other than CPU reads, cursor and VGA reads will generate an error).
- All INSTDONE bits are set ("DONE").
- The NOPID register is 0.
- All command groupings are enabled (via INSTPM)

6 Frame Buffer Compression [DevCL only]

6.1 Overview

The Run-Length Encoded Frame Buffer Compression (RLE-FBC) function is a mechanism to reduce display refresh memory traffic. By reducing memory reads required for display refresh, power consumption is reduced (thus extending battery life for mobile systems).

The conditions under which the RLE-FBC is most effective are:

- Display images that are well suited to RLE compression. Good examples are text windows, slide shows, etc. Poor examples are 3D games - rich in textured and smooth-shaded objects.
- Screens that are fairly static. Good examples are screens with significant portions of the background showing, 2D apps (reading mail, etc.), CPU benchmarks, etc., or conditions when the CPU is idle. Poor examples are full-screen 3D games and benchmarks that flip the display image at or near display refresh rates.

Note that this compression function is different from, and mutually exclusive with, Discard Alpha Frame Buffer Compression – which is effective for 32bpp 3D environments.

The RLE-FBC function is comprised of three subfunctions:

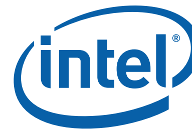
- A **Compressor** that attempts to compress the display buffer as a background task.
- A **Decompressor** in the Display engine that uses compressed lines for display refresh, if available.
- A **Frame Buffer Write Detector** that snoops writes to the uncompressed frame buffer and invalidates the corresponding compressed lines.

The RLE-FBC **Compressor** periodically compresses lines of Display Plane A (an uncompressed display source image) using run-length encoding and stores the results into a pre-allocated compressed frame buffer. During subsequent display refreshes, the Display engine **Decompressor** attempts to refresh Display A from the compressed frame buffer. Lines that were not compressed or lines that were modified since the last compression – as detected by the **Frame Buffer Write Detector** – are displayed from the uncompressed buffer.

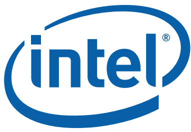
6.2 Programming Interface

6.2.1 FBC unit programming interface

The following table summarizes the register programming interface to the RLE-FBC function. Refer to the *Memory Interface Registers* chapter for details on the individual registers provided in the programming interface.



Register	Field(s)	Description
FBC_CFB_BASE	Compressed Frame Buffer Address	Specifies the location of the compressed frame buffer
FBC_LL_BASE	Compressed Line Length Buffer Address	Specifies the location of the compressed line length buffer
FBC_CONTROL	Enable	Turns the RLE-FBC function on/off
	Mode Select	Specifies Single or Periodic compression mode
	Interval	Specifies time period (in display refreshes) used in periodic mode
	Stop Compressing on Modification (DEBUG)	Specifies that the compression pass should be aborted if a line is modified during compression.
	Uncompressible Enable	Enable Uncompressible state for the tag RAM. if ENABLE compressor will mark the uncompressible scan line to prevent future compressing attempt
	Compressed Frame Buffer Stride	Specifies the stride (pitch) of the compressed frame buffer 64-byte unit
	Fence Number	Specifies the FENCE register associated with the uncompressed source frame buffer
FBC_CONTROL2		
	FBC Cx state mode	Specifies FBC behavior when PM signals CPU goes to Cx (non C0)
	CPU Fence Enable	If ENABLE the display buffer is existed within CPU fence
	Display Plane Select	Select Plane A or B for Frame Buffer Compression
FBC_YFENCE_DISP	Fence Display Buffer Y offset	Y offset from the CPU fence to the Display Buffer base
FBC_MOD_CTR	FBC modification Counter for Recompression	Recompress the Display Buffer only after the programmed number of modifications to the display buffer
FBC_COMMAND	Compression Request	Used to request compression passes in Single compression mode
FBC_STATUS	Compressing (RO)	Status indicating if the compressor is running.



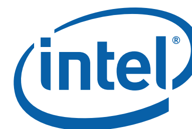
Register	Field(s)	Description
	Compressed (RO, R/W for DEBUG)	Status indicating if the compressed frame buffer is available for display
	Any Modified (RO, R/W for DEBUG)	Indicates whether any lines of the uncompressed frame buffer have been modified since the last compression pass.
	Current Line Compressing (RO)	Indicates the progress of the compressor
FBC_TAG[0..N]	Tag[i+0..i+48] (DEBUG)	Status indication for each pair of display lines.

6.2.2 Programming interface from Display Engine

The following table summarizes the indirect register programming interface to the RLE-FBC function from Display Engine. These registers are programmed in Display Engine for Display function, but they are passed to FBC unit to use for Frame Buffer Compression operation. Depend on how FBC_CONTROL2< **Display Plane Select**> is set Display Plane A or B registers are passed to FBC unit. Refer to the *Memory Interface Registers* chapter for details on the individual registers provided in the programming interface.

FBC used these registers when reading uncompressed frame buffer and building a compressed buffer that is identical to uncompressed buffer of Display Plane A or B.

Register	Field(s)	Description
DSPA**(B)**CNTR	Display A**(B)** Source Pixel Format	4-bit source Pixel format- FBC can only works with 16-bit or 32-bit Source pixel format that organize in 8-bit chunk (not 10:10:10:2 format)
DSPA**(B)**STRIDE	Display A **(B)** Stride	This value is used to determine the line to line increment for the display. FBC can work with non-power-of-two stride from 2KB to 16KB with increment of 512bytes
DSPA**(B)**SURF	Display A **(B)** Surface Base Address	This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the DSPA (B) TILEOFF register. This address must be 4K aligned.
DSPA**(B)**LINOFF	Plane Start Y-Position	These 12 bits specify the vertical position in lines of the beginning of the active display plane relative to the display surface.



Register	Field(s)	Description
	Plane Start X-Position	These 12 bits specify the horizontal offset in pixels of the beginning of the active display plane relative to the display surface.
HTOTAL**(B)**	Pipe A **(B)** Horizontal Active Display Pixels	This 12-bit field provides Horizontal Active Display resolutions up to 4096 pixels. Note that the first horizontal active display pixel is considered pixel number 0. The value programmed should be the (active pixels/line – 1).
VTOTAL**(B)**	Pipe A **(B)**Vertical Active Display Lines	This 12-bit field provides vertical active display resolutions up to 4096 lines. It should be programmed with the desired number of lines minus one.

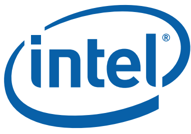
6.3 Operating Modes

6.3.1 RLE-FBC Function Modes

The RLE-FBC function (compression and decompression) is enabled or disabled via the **Enable** bit of the FBC_CONTROL register.

In order to request the disabling of the function software must set **Enable** to DISABLED. The function does not subsequently become disabled until the **Compressing** status bit of FBC_STATUS is clear. Software must ensure that the function is in fact disabled (via interrogation of the **Compressing** status bit) before re-enabling the RLE-FBC function and under the following conditions:

- Prior to changing the contents of the FBC_CFB_BASE or FBC_LL_BASE registers
- Prior to changing the contents of the following fields of the FBC_CONTROL register:
  - Mode Select
  - Interval
  - Stop Compressing on Modification
  - Uncompressible Enable
  - Compressed Frame Buffer Stride
  - Fence Number
- Prior to changing the contents of the following fields of the FBC_CONTROL2 register:
  - FBC Cx state mode
  - CPU fence Enable
  - Frame Buffer Compression Display Plane Select A/B
- Prior to changing the contents of the FBC_Fence_Display_Y_Offset register:
- Prior to changing the contents of the following fields of the FBC_MOD_CTR register:
  - FBC_mod_ctr
  - FBC_mod_ctr_valid
- Prior to changing the display mode of the source frame buffer (Display Plane A) including display pixel format, dimensions, and pitch (stride).
- Prior to entering/use of any modes listed under *Restrictions* below



Modification of DEBUG-mode controls is implementation dependent.

6.3.2 Compression Modes

The RLE-FBC compressor is capable of operating in one of two modes, Single or Periodic Compression, as specified by the **Mode Select** field of the FBC_CONTROL register.

6.3.2.1 Single Compression Mode

In this mode software can request a single compression pass via the **Compression Request** bit of the FBC_COMMAND register. The compression results will be used until another compression is requested or the RLE-FBC function is disabled. Note that subsequent modifications to the uncompressed frame buffer will invalidate corresponding compressed lines – diminishing the benefits of the function.

Single compression mode is preferred when software has knowledge that significant portions of the frame buffer <u>lines</u> will remain static for a period of time – where memory bandwidth would not be wasted further recompressing the static frame buffer data.

6.3.2.2 Periodic Compression Mode

In Periodic mode, recompression is attempted at a programmed rate in units of display refreshes. The time period is programmed via the **Interval** field of the FBC_CONTROL register. The RLE-FBC compressor will not initiate a periodic compression if there have been no modifications to the source frame buffer since the last compression.

This mode is preferred when software expects significant portions of the frame buffer line to be written on a frequent basis (or at least cannot guarantee that this will not occur). The time period can be adjusted according to the refresh rate and/or frequency and extent of (expected) frame buffer modifications.

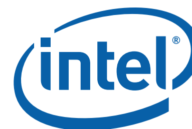
If Uncompressible **Enable** is set to ENABLED the compressor will mark a tag line uncompressible if both scan lines of a tag line are uncompressible so compressor won't attempt to compress these scan lines again in subsequent compression run unless these lines are modified by CPU or RC.

If FBC_mod_ctr_valid is SET the compressor will only attempt to recompress if the number of tag lines were modified since last compression run is greater or equal the value of FBC_mod_ctr.

6.4 Usage Restrictions

RLE Frame Buffer compression must not be enabled unless the following conditions are met:

1. If Display A is selected DSPACNTR—Display A Plane Control Register[Pixel Multiply] = No line duplication and Display A Plane Control Register[Horizontal Pixel Multiply] = 1x
2. If Display B is selected DSPBCNTR—Display B Plane Control Register[Pixel Multiply] = No line duplication and Display B Plane Control Register[Horizontal Pixel Multiply] = 1x
3. Panning of Selected Display Plane is permitted. If FBC is enabled and a compressed buffer is available when a panning event happened FBC will invalidate the current compressed buffer and recompress if necessary using the current FBC control parameters. If new uncompressed buffer required a new set of FBC control parameters then RLE-FBC must be first disabled.
4. <u>Sync flips</u> of Selected Display Plane <u>are</u> permitted. If FBC is enabled and a compressed buffer is available when sync flips event happened FBC will invalidate the current compressed buffer and recompress if necessary using the current FBC control parameters. If new uncompressed buffer required a new set of FBC control parameters then RLE-FBC must be first disabled



5. The display pixel format is 15-bit, 16-bit or 32-bit xRGB_8888 mode (as the alpha channel is removed as part of the compression).
6. Discard Alpha Frame Buffer Compression is DISABLED.
7. The uncompressed frame buffer is tiled with pitch from 2KB to 16KB in step of 0.5KB
8. The Line Width (in pixels) of the uncompressed frame buffer is a multiple of 8 in the range [640, 2048].
9. Number of lines of the uncompressed frame buffer is a multiple of 2 in the range [480, 1536].
10. Dual-wide display is not active.
11. If the pipe A is selected (i.e., DSPACNTR—Display A Plane Control Register [Display Pipe A Select] = Select Pipe A), then Pipe A Configuration Register [Interlaced modes] must be in Progressive mode.
12. If the pipe B is selected (i.e., DSPBCNTR—Display B Plane Control Register [Display Pipe B Select] = Select Pipe B), then Pipe B Configuration Register [Interlaced modes] must be in Progressive mode.
13. Compressed Frame Buffer Stride in bytes is equal or smaller than Uncompressed Frame Buffer Stride in bytes to prevent unintended buffer expansion in 16bpp frame.
14. Both Regular and SR display watermarks for 16bpp must equal 32bpp as calculated
15. Compressed Frame Buffer and Line Length buffers must reside entirely in stolen memory segment. This restriction is added so RLE-FBC can be enabled with LT. If hardware tried to access compressed buffer or line length buffer outside of stolen memory FBC unit will be invalidate compressed buffers and makes unavailable to DISPLAY.
16. Display 180 degree rotation using GenX hardware is turned off. This feature is not compatible with FBC scanline addressing. Software rotation can be enabled at the same time with FBC.
17. Async Flips are not permitted. FBC must be disabled when async flips are in use.

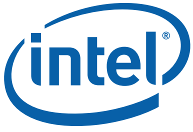
6.5 Power Management Interface

At the system level the amount of saving power of Frame Buffer Compression may be offset by power consumed by other units including CPU and memory subsystem when waiting for Frame Buffer Compression complete its pass. Device-specific power management modes need to add in to basic Frame Buffer Compression operation.

For [DevCL], different Cx state modes are used to provide a tuning mechanism between CPU low-power states (or Cx state) and FBC operation. Power Management Unit will signal to FBC that CPU is in low power state and wait for FBC to signal back that FBC is idle and no longer accessing external memory. Power Management unit then can implement global power saving scheme like putting external memory in self-refresh or clock gating FBC and/or other related units.

In DevCL, Cx state mode are specified as following:

- FBC_CONTROL2**<Cx state mode>**=IMMEDIATE IDLENESS. FBC blocks its requests to memory (read and write) and waits for all read returns to complete before asserting FBC-idle (default)
- FBC_CONTROL2**<Cx state mode>**=NORMAL IDLENESS. FBC finishes current compression pass before asserting FBC-idle
- FBC_CONTROL2**<Cx state mode>**=SCANLINE IDLENESS FBC completes the current line/line pair and skips remaining lines and makes the compressed buffer available for display before asserting FBC-idle.



- FBC_CONTROL2**<Cx state mode>**=IMMEDIATE DEBUG IDLENESS. FBC asserting FBC-idle immediately, more memory transactions may be still underway. This allows PM to find the fastest path to go to lower power state regardless of FBC operation.

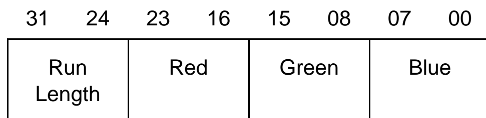
6.6 Memory Data Structures

6.6.1 RLE Pixel Runs

A compressed line contains one or more *pixel runs* of identical pixel values. A pixel run is stored as a DWord containing (1) an RGB *pixel value* and (2) a *run length* that specifies the number of times (minus one) that the pixel value is to be replicated.

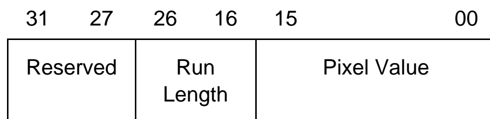
For 32bpp pixel formats, the run length is encoded in Bits 31:24 of the run Dword. This permits run lengths of 1 to 256 pixels. Any alpha value stored in Bits 31:24 is discarded. The remaining 24-bit RGB pixel value is left in place (in Bits 23:0).

Figure 6-1. 32bpp Pixel Run



For 16bpp pixel formats, the run length is encoded in Bits 26:16 of the run Dword. This permits run lengths of 1 to 2048 pixels. The 16-bit RGB pixel value is stored in Bits 15:0 (for 15bpp formats, Bit 15 is Reserved).

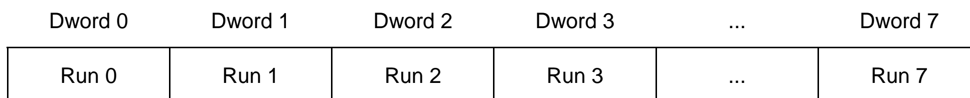
Figure 6-2. 16bpp Pixel Run



6.6.2 RLE Pixel Run Sets

The RLE-FBC function groups 8 consecutive pixel runs into 32-byte (Sword) *pixel run sets*. This matches the granularity used to read the compressed frame buffer.

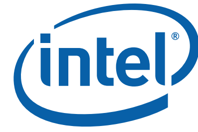
Figure 6-3. Pixel Run Set



6.6.3 RLE-Compressed Line

An RLE-compressed *line* is comprised of a horizontal series of pixel run sets corresponding to a scan line in the uncompressed frame buffer.

Note that there is no encoding for "unused" Dwords in the last pixel run set. During display the Display engine will end the decompression of pixel runs when the number of decompressed pixels per line is satisfied.



6.6.4 RLE Compressed Frame and Line Length Buffers

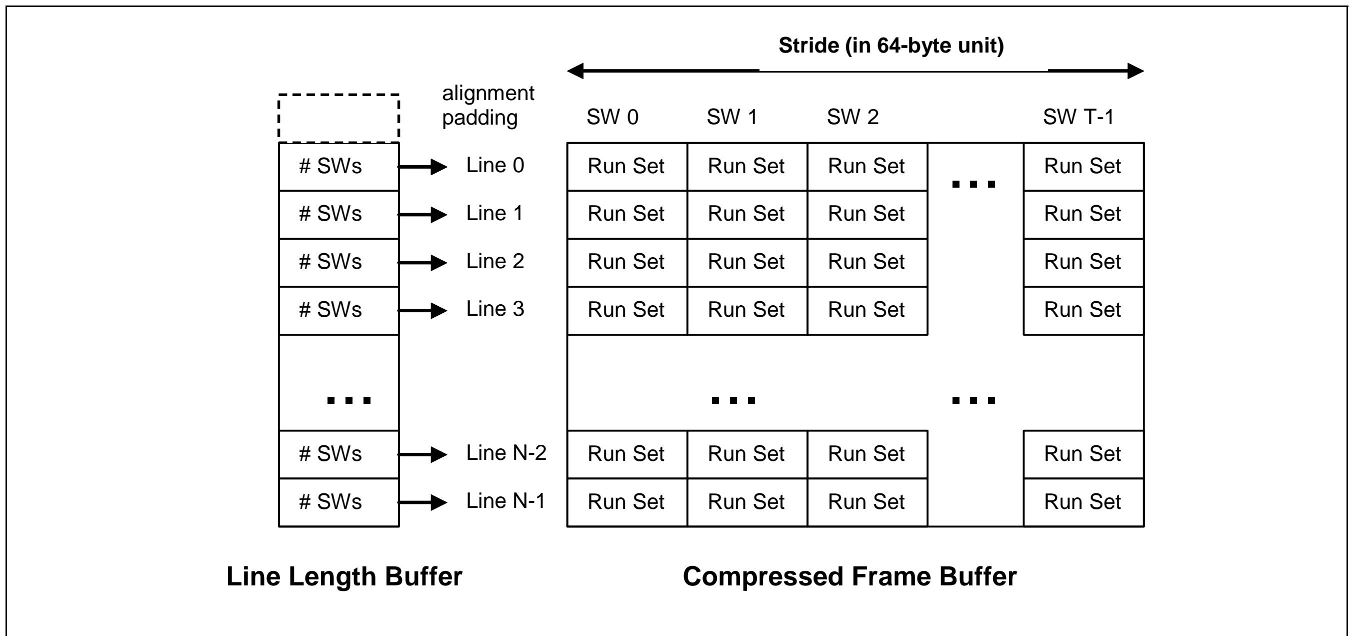
The RLE-compressed frame buffer and the Compressed Line Length Buffer must be in locked, fixed, contiguous, and uncacheable physical memory.

The RLE-Compressed Frame Buffer is a 4KB-aligned rectangular array of pixel run sets residing in physically contiguous memory (it is not mapped by the GTT). The physical address of the buffer is programmed via the FBC_CFB_BASE register.

The stride (width) of the buffer in Swords (run sets) is programmed via the **Compressed Frame Buffer Stride** field of the FBC_CONTROL register.

Different lines will typically compress to a different number of Pixel Runs. In order to record how many Swords needs to be fetched from the RLE-Compressed Frame Buffer, a Compressed Line Length Buffer is used. The Compressed Line Length Buffer is a (1536+32)-byte, 4KB-aligned list in physically contiguous memory (it is not mapped by the GTT). The physical address of the buffer is programmed via the FBC_LL_BASE register. Each byte in the buffer specifies the number of Swords (minus one) valid for the corresponding line in the RLE-Compressed Frame Buffer.

Figure 6-4. RLE-Compression Buffers

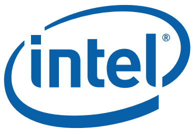


The byte in the Compressed Line Length Buffer that corresponds to Line 0 of the Compressed Frame Buffer is offset according to the alignment of the uncompressed display buffer. The Compressor and Decompressor both use the 6 least significant bit of y offset from Display Base as starting offset for line 0.

6.7 Tuning Parameters

6.7.1 Stride

The **Compressed Frame Buffer Stride** field of the FBC_CONTROL register specifies the distance (in 64-byte unit) between consecutive lines in the compressed frame buffer. If a source line cannot be compressed to fit within a compressed line, it will remain uncompressed.



The maximum compression ratio can be achieved by setting the compressed frame buffer stride to correspond with the uncompressed frame buffer line length. The stride can be set to a smaller number if there is not enough memory available for the compressed frame buffer.

6.7.2 Interval

As previously mentioned, the interval with which periodic compression passes are attempted can be adjusted as desired (e.g., as a function of refresh rate and/or expected frequency/extent of frame buffer modifications. The interval is programmed via the **Interval** field of the FBC_CONTROL register.

6.7.3 FBC Modification Counter

As previously mentioned, the FBC modification Counter can be used to reduce the number of recompression attempts if the number of modification since last attempt is small. At **Interval** expiry compressor will compare the number of accumulated tag line modifications (tag line modification counter) with the value of **FBC_mod_ctr** if the latter is larger the compressor will be back to sleep and tag line modification counter will continue counting.

6.8 Implementation (DEBUG)

This section describes the implementation of RLE_FBC function. Information in this section is not required for operational drivers – it is only required for debug activities.

6.8.1 Tag Array

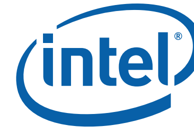
A tag associated with every two sequential lines and indicates the current status of the lines. The tag states are defined as follows:

Tag Encoding	Definition	Description
'00'	Modified	At least one of the lines of the pair has been modified since the last compression pass, or a compression pass has not been made since (a) the source buffer address has changed, (b) RLE-FBC has been enabled, or (c) Reset
'01'	Uncompressed	One of the lines has not been compressed successfully.
'10'	Uncompressible	Both of the lines are uncompressible (compressed length is larger than compressed stride)
'11'	Compressed	Both of the lines are compressed

If the first line of the uncompressed source frame buffer is in an odd address, the first tag entry is associated with only one line, the first line; the second entry is associated with the second and third frame buffer line and so on. The last line will be also alone in this case.

6.8.1.1 Transitions

The following table describes the valid transitions of the Tag value. All tags start at the Modified state upon reset.

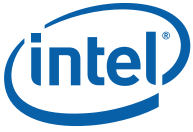


From	To	Conditions
Modified	Uncompressed	Unconditionally at the start of a compression pass.
Uncompressed	Modified	One of the lines is modified, or the source frame buffer base address was changed, or when compression becomes enabled.
Uncompressed	Compressed	Both lines were successfully compressed.
Uncompressed	Uncompressible	Both lines were unsuccessfully compressed in the previous pass
Compressed	Modified	Line was modified, or the source frame buffer base address was changed, or when compression becomes enabled.
Uncompressible	Modified	Line was modified, or the source frame buffer base address was changed, or when compression becomes enabled.

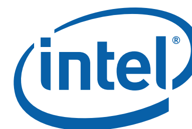
6.8.2 Compressor

The compressor will compress only if the display is on.

```
START:
if (Display Plane)
        return
on (Start of Display Vblank)
        Sample the FBC address and configuration registers
        if (Mode == Periodic)
                 Interval counter = interval counter-- % Interval

        if ((Mode == Periodic AND Interval == 0) OR Compression Request) AND
                 Display is ON AND
                 (At least one line pair is Modified) AND
                 (!Compressing) AND
                 (Local cache and write posting buffers are empty) AND
                 (Display buffer is tiled)
                        goto COMPRESSION
                        else goto START

COMPRESSION:
        {
                 Change Modified to Uncompressed // One cycle
                 Set FBC_CONTROL<Compressing>
                 Reset FBC_CONTROL<Compressed>
                 Reset FBC_CONTROL<Modified>

                 for (each and every Uncompressed line pair)
                 {
          /* By first marking and then compressing we guarantee that modification to this
          line will be marked as Modified and will not be overridden when compression is
          completed */
                        Mark the pair as Compressed
                        Compress first line
                        if (Stride exceeded)
                                 Mark pair as Uncompressed
                        else
                                 Write the compressed line length to the line-length buffer
                                 Compress second line
                                 if (Stride exceeded)
                                                   Mark pair as Uncompressed
```




```
                                  else
                                                          Write the compressed line length to the
line-length buffer
                                                          Mark pair as Compressed
                                    Set FBC_CONTROL<Compressed>
                  } // end for each uncompressed line pair
                  Reset the "Compression in progress" bit
                  Set Compressed-buffer-avail bit
  } // end compression

// If we succeeded to compress or not
if (Mode == Periodic)
        Reset the interval-counter
goto START
```



6.8.3 Decompressor

When the display streamer gets the first line request it checks for the following condition:

FBC_CONTROL<**Enable**> is set

FBC_CONTROL<**Compressing**> is clear (compression not in progress)

FBC_CONTROL<**Compressed**> is set (a compression pass has completed)

If any of these conditions are not met, only the uncompressed source buffer will be used for refresh.

If all these conditions are met, the Decompressor will, for every line:

- If the line marked as ***Compressed*** the display streamer will read the compressed line length from the compressed line length buffer, and then read the compressed line data according to this length. If the line is <u>not</u> marked as ***Compressed***, the display streamer reads the line from the uncompressed frame buffer. In both cases the pixel data is posted to the display FIFO.
- If the line is ***Compressed*** the Decompressor reads Dwords from the FIFO and sends on the pixel data multiple times according to the run length, 1 – 256 in 32-bit mode and 1 – 2048 in 16-bit modes. The Decompressor keeps track of the number of pixels and stops when it reaches the line width (in pixels) and discards any remaining Dwords.

6.8.4 Frame Buffer Write Detector

The Frame Buffer Write Detector snoops all frame buffers writes from the CPU and render engines, and marks the modified line pairs as ***Modified.***

- If Display buffer is a subset of the render buffer and cpu path is enabled via a fence, where the fence is a superset of the render buffer then frame buffers lines might be modified by both cpu write and render cache write.
- If Display buffer is a subset of render buffer and fence cpu path is disable then frame buffers might be modified by render cache line only.
- If CPU path is disabled and Render and Display are independent buffers then no modified should be happened.

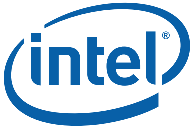
In order to detect CPU write the following FBC registers need to be programmed before the FBC is enabled:

- FBC_CONTROL2 <**CPU Fence Enable**> is set.
- FBC_CONTROL <**Fence Number**> set to match the fence that render target and Frame Buffer reside in.
- FBC_YFENCE_DISP is set to the distance from fence base address to DSPA**(B)**SURF

Chipset unit passes CPU writes that are within Graphic Aperture to FBC. FBC write detector decode the line number and marked affected line as ***modified.***

There are no register programming needed for render cache write monitor. Render cache unit pass each write to its cache to FBC. If Render Target Address match with DSPA (B) SURF, and the render cache line has the same offset with active display then the affected line pair is marked as ***Modified.***

All lines will be marked as modified whenever:



- The uncompressed source Frame Buffer base address changes (this is only permitted to happen as a result of a direct register write – flips of Selected Display Plane are not allowed when RLE-FBC is enabled)
- RLE-FBC is enabled
- Reset

If the FBC_CONTROL<**Stop Compressing on Modification**> (DEBUG) bit is set, and a source frame buffer write is detected during a compression pass, the compression is aborted and the current line pair is marked as ***Modified***. Compression will be reattempted at the next periodic compression or when the next single compression pass is requested.

6.8.5 Coherency

The display coherency is kept by keeping the following rules:

- The compressed frame buffer is not displayed during compression.
- The Compressor will only compress lines that are marked as ***Uncompressed***.
- Lines state changes from ***Modified*** to ***Uncompressed*** can only when there are no display reads or pending display writes. This is achieved by waiting for Vblank start and then starting the compression only if the render cache is empty.
- Marking a line as ***Modified*** takes precedence over the (simultaneous) transition from ***Modified*** to ***Uncompressed***.
- Before a line pair is compressed, the tag is changed from ***Uncompressed*** to ***Compressed***. This will guarantee that if a line is modified while being compressed it will transition to the ***Modified*** state.
- Compressor frame buffer reads push CPU writes to memory.
- At the end of each compression path FBC issues dummy reads to push Compressed Buffer writes to memory.

7 Frame Buffer Compression [DevCTG]

This chapter contains the register descriptions for the Cantiga Frame Buffer Compression (also known as Display Plane Frame Compressor or DPFC) portion of integrated graphics devices.

These registers do vary by device within the family of devices, so special attention needs to be paid to which devices use which registers and register fields. Different devices within the family may add, modify, or delete registers or register fields relative to another device in the same family based on the supported functions of that device.

The following table contains the break down of the register information contained within this chapter:

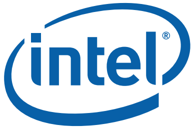
Address Offset	Register Name
3200	DPFC Compressed Buffer Address
3204	Reserved for future use
3208	DPFC Control
320C	DPFC ReComp Control
3210	DPFC Status
3214	DPFC Status 2 (Reserved)
3218	DPFC CPU Fence Offset
321C	DPFC SLB DATA
3220	DPFC Debug Status
3224	DPFC Extra Control
3300-33C3	Reserved for future use

Also of interest to Display Plane Frame Compressor is the Display FIFO Watermark Control 2 Register 0x70038 in the Display Registers chapter.

7.1 DPFC Programming Interface

7.1.1 FBC2 supported feature and limitation

- Supports up to 2K lines and 4K pixels
- Supports only xtiled memory surface format
- Supports interlaced and rotation mode
- Does not support pixel multiply mode
- Can only be enabled when output is to a local panel at the native resolution
- Does not support asynchronous flips
- Can not be enabled together with video sprite on the same display pipe



- FBC2 stride is calculated as (The stride of the Primary Plane FBC2 is assigned to) / (FBC2 compression ratio). The stolen memory needed for compressed frame buffer must be greater or equal to (FBC2 stride * active display height size).
- Supports 16-bpp and 32-bpp format. The supported format with the supported compression ratio is summarized in the following table.

Pixel format/ Compression Ratio	16bpp	32bpp
1	Not Supported	Supported
½	Supported	Supported
¼	Supported	Supported

7.1.2 FBC2 usage model and restriction on persistent and non-persistent mode

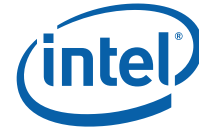
7.1.2.1 General Restrictions

- FBC2 can only track modifications onto one buffer, which is either front buffer or back buffer.
- Persistent and non persistent mode and associated functions can not be changed while FBC2 is enabled
- Async flips will cause the entire frame to be recompressed (Nuke).

7.1.2.2 Non Persistent Mode

- Supports recompression on front buffer modification
- All flips will cause a nuke
- Does not track back buffer modification
- The following mode setting is used in non persistent mode, applies to both RC and HT modifications.

Non-Persistent Mode Settings	
Persistent mode	Set to disable
MMIO SYNC Flip Nuke disable	Set to 0 to enable nuke
CS SYNC Flip Nuke disable	Set to 0 to enable nuke



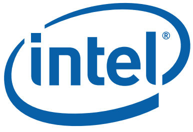
7.1.2.3 Persistent Mode

- Back buffer modifications have to be contiguous and followed by flip
    - Once the flip happens, no further modifications to that buffer are tracked
- FBC2 only tracks back buffer in persistent mode and recompress the modified lines after flip.
- Nuke on CS Sync flip and MMIO Sync Flip can be disabled. The Mode setting can only be changed when FBC2 is disabled.
- The following mode setting is used in persistent mode, applies to both RC and HT modifications.

Persistent Mode Settings	
Persistent mode	Set to enable
MMIO SYNC Flip Nuke disable	0: enable Nuke 1: disable nuke
CS SYNC Flip Nuke disable	0: enable Nuke 1: disable nuke
HT Modification Tracking bit	Write 1 when CPU fence is set.

- In order to do the HT back buffer modify in persistent mode, SW needs to follow the following steps:
    1. Set the fence to back buffer.
    2. Write 1 to the HT modification tracking bit.
    3. CPU modifies the back buffer.
    4. Flip to the back buffer. (Any flip).
    5. Repeat step 1~4.

    Note:
    1. Fence Enable/Fence number/HT modification tracking bit can be changed on the flight when FBC2 is enabled.
    2. SW writes to 1 to the HT modification tracking bit will set the HW modification in progress. HW will clear the modification in progress by flips. SW does not need to clear this bit.



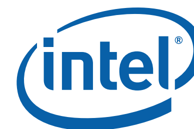
7.2 DPFC Control Registers (03200h–033FFh)

7.2.1 DPFC_CB_BASE – DPFC Compressed Buffer Base Address

Memory Offset Address: 03200h–03203h
Default: 0000 0000h
Attributes: Read/Write
Size: 32 bits

The contents of this register can not be changed while compression is enabled.

Bit	Description
31:28	**Reserved:** Write as zero
27:12	**Compressed Frame Buffer Offset Address:** This register specifies offset of the Compressed Frame Buffer from the base of stolen memory. The buffer must be 4K byte aligned.
11:0	**Reserved:** Write as zero

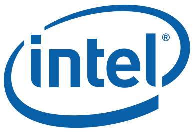


7.2.2 DPFC_CONTROL— DPFC Control

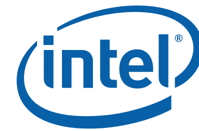
Memory Offset Address: 03208h–0320Bh
Default: 00000000h
Attributes: Read/Write
Size: 32 bits

The contents of this register can not be changed except bit 31 while compression is enabled.

Bit	Description
31	**Enable Frame Buffer Compression:** This bit is used to globally enable DPFC function at the next Vertical Blank start. 0: Disable frame buffer compression. 1: Enable frame buffer compression.
30	**Plane Select:** 0: Plane A 1: Plane B
29	**CPU Fence Enable:** 0: Display Buffer is not in a CPU fence. No modifications are allowed from CPU to the Display Buffer. 1: Display Buffer exists in a CPU fence.
28	**Reserved:** Write as zero
27	**CS SYNC FLIP NUKE Disable:** Setting this bit will disable the command streamer SYNC Flips from resetting the DPFC. 0: Enable the CS SYNC Flip Nuke. 1: Disable the CS SYNC Flip Nuke.
26	**[DevCTG-B] MMIO SYNC FLIP Nuke Disable:** Setting this bit will disable the MMIO Sync Flip from resetting the DPFC. 0: Enable the MMIO Sync Flip Nuke. 1: Disable the MMIO Sync Flip Nuke. **[DevCTG-A] Reserved**
25	**Persistent Mode:** 0: Non Persistent Mode. 1: Persistent Mode. Enable the invalid modify qualify from CS.
24:16	**Compression Control (Test mode):** Setting the bits in this register disables certain compression capabilities. Bit 8: Run length with 1 nibble Bit 7: Run length with 2 nibble Bit 6: Mono Palette Bit 5: Historical Palette Bit 4: Delta 6 Bit 3: Delta 5 Bit 2: Delta 4 Bit 1: Delta 3 Bit 0: Delta 2



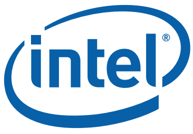
<table>
<tr><td>15</td><td>SLB Initialization Flush Disable Control (Test mode):
Setting this bit will disable the SLB flush mechanism for the first frame DPFC is on.
0: Enable the SLB initialization flush. (normal operation)
1: Disable SLB initialization flush.</td></tr>
<tr><td>14:11</td><td>Reserved: Write as zero</td></tr>
<tr><td>10</td><td>Compression SR Mode:
0: SR gates compressed data write back. (default)
1: Compressed data write back gates SR.</td></tr>
<tr><td>9</td><td>Last Pixel SR Mode Exit Disable:
Setting this bit will disable exit SR mode immediately for write back on the last pixel of the frame.
0: Exit SR mode at the last pixel of the frame for compressed data write back.
1: SR mode gates write back at the last pixel of the frame.</td></tr>
<tr><td>8</td><td>Reserved: Write as zero</td></tr>
<tr><td>7:6</td><td>Compression Limit:
This register sets a minimum limit on compression. It is also used to determine the size of the compressed buffer.
00: 1:1 compression, compressed buffer is the same size as the uncompressed buffer.
01: 2:1 compression, compressed buffer is one half the size of the uncompressed buffer.
10: 4:1 compression, compressed buffer is one quarter the size of the uncompressed buffer.
11: Reserved.
<table>
<tr><th>Compression Ratio</th><th colspan="2">Pixel Format</th></tr>
<tr><th></th><th>16 bpp</th><th>32 bpp</th></tr>
<tr><td>1</td><td>Not Supported</td><td>Supported (CFB=FB)</td></tr>
<tr><td>½</td><td>Supported (CFB=FB)</td><td>Supported (CFB=1/2 FB)</td></tr>
<tr><td>¼</td><td>Supported (CFB=1/2FB)</td><td>Supported (CFB=1/4 FB)</td></tr>
</table>
FB = Frame Buffer Size
CFB = Compressed Frame Buffer Size</td></tr>
<tr><td>5:4</td><td>Write Back Watermark:
Compressed data write back engine waits for this amount of data (per segment) to be ready before writing the data out to memory. Compression SR mode must be a 1, or SR disabled for this to take effect.
00: 4 cache lines
01: 8 cache lines
1X: Reserved</td></tr>
<tr><td>3:0</td><td>CPU Fence Number:
This field specifies the CPU visible FENCE number corresponding to the placement of the uncompressed frame buffer.</td></tr>
</table>



7.2.3 DPFC_RECOMP_CTL — DPFC ReComp Control

Memory Offset Address: 0320Ch–0320Fh
Default: 0000 0000h
Attributes: Read/Write
Size: 32 bits

Bit	Description
31:28	**Reserved:** Write as zero
27	**Enable ReComp Stall:** 0: Disabled 1: Enabled
26:16	**ReComp Stall Invalidation Watermark:** If this many or more invalidations occur in one frame, stop compression until the number falls below watermark, then start the recomp timer.
15:6	**Reserved:** Write as zero
5:0	**ReCompression Timer Count:** After invalidations fall below watermark, wait this many frames before restarting the compressor. A 0 means restart compression on the following frame.



7.2.4 DPFC_STATUS — DPFC Status

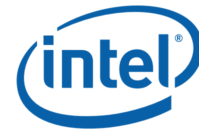
Memory Offset Address: 03210h–03213h
Default: 0000 0000h
Attributes: Read Only
Size: 32 bits

Bit	Description
31:27	**Reserved:** Read as zero
26:16	**Invalidated Segment Count:** Updated each vblank, this field indicates the number of segments that have been invalidated for the previous frame.
15:11	**Reserved:** Read as zero
10:0	**Compressed Segment Count:** Updated each vblank, this field indicates the number of segments that were fetched from the compressed frame buffer for the previous frame.

7.2.5 DPFC_STATUS_2 — DPFC Status 2

Memory Offset Address: 03214h–03217h
Default: 0000 0000h
Attributes: Read Only
Size: 32 bits

Bit	Description
31:27	**Reserved:** Read as zero
26:16	**TBD1 Count:** Updated each vblank, this field indicates something I want to count
15:11	**Reserved:** Read as zero
10:0	**TBD2 Count:** Updated each vblank, this field indicates something else I want to count or maybe a threshold I don't know yet. it might generate an interrupt or cause your computer to turn green



7.2.6 DPFC_CPU_Fence_Offset — Y Offset CPU Fence Base to Display Buffer Base

Memory Offset Address:	03218h–0321Bh
Default:	0000 0000h
Attributes:	Read/Write
Size:	32 bits

The contents of this register can not be changed while compression is enabled.

CPU Fence

Yfence_rt = Yfence disp

Display_base[31:12] = RT base[31:12]

RT

Display Buffer

Yrtmodifyoffset

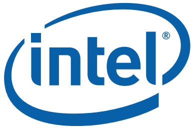
Yhtmodifyoffset

Ydispoffset

Active Display

LC modify

CPU modify



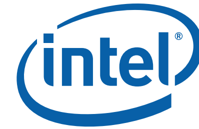
Bit	Description
31:22	**Reserved:** Write as zero
21:0	**Yfence_disp:** Y offset from the CPU fence to the Display Buffer base.

7.2.7 PFC_SLB_DAT—DPFC SLB Data

Memory Offset Address: 0321Ch–0321Fh
Default: 0000 0000h
Normal Access: Read/Write
Size: 32 bits

This register is used to read out the internal SLB data based on the line number. When writing to this register, the SLB pointer will move back to line 0. The SLB pointer is incremented by 2 lines for every read. The line number starts from 0. This is a test mode register.

Bit	Descriptions
31:26	**Reserved:** Write as zero
25:16	**SLB data for odd line numbers:** SLB Line 1, 3, 5, ...
15:10	**Reserved:** Write as zero
9:0	**SLB data for even line numbers:** SLB Line 0, 2, 4, ...

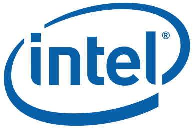


7.2.8 DPFC_DEBUG_STATUS—DPFC Debug Status

Memory Offset Address: 03220h–03223h
Default: 0000 0000h
Normal Access: Read/Write
Size: 32 bits

This register is used for debug purposes. Once detecting the error conditions specified below, the corresponding status register bit will be set. Write 1 to these register bits to clear the error bit set.

Bit	Descriptions
31:7	**Reserved:** Write as zero
6	**Recompression Stall Watermark Trip:** Modify exceeds watermark programmed in DPFC_RECOMP_CTL Register.
5	**RC Modify CAM Overflow**
4	**HT Modify CAM Overflow**
3	**Compressed Tag Underrun:** Underrun for streamer put the compressed tag to the decompressor. Need to adjust the register 70038h display FIFO watermark control 2 register.
2	**Pipe Underrun:** DPFC assigned pipe underrun.
1	**Dummy read on vblank not returned on framestart:** Dummy read issued on vblank not returned on the frame start. RC/HT modify before the vblank has not been flushed into memory yet.
0	**Compressed write back data FIFO not empty on framestart:** On the framestart, the compressed data FIFO is not empty. Compressed data is not able to be fully written back from the last pixel on the previous frame to the framestart of this frame.



7.2.9 DPFC_EXTRA—DPFC Extra Control Bits

Memory Offset Address: 03224h–03227h
Default: 0000 0000h
Normal Access: Read/Write
Size: 32 bits

Bit	Descriptions
31	**[DevCTG-B] DPFC HT Modify Tracking Bit:** Write 1 to this register when CPU fence number is set. **[DevCTG-A] Reserved**
30:0	**Reserved**

8 BLT Engine

8.1 Introduction

2D Rendering can be divided into 2 categories: classical BLTs, described here, and 3D BLTs. 3D BLTs are operations which can take advantage of the 3D drawing engine's functionality and access patterns.

Functions such as Alpha BLTs, arithmetic (bilinear) stretch BLTs, rotations, transposing pixel maps, color space conversion, and DIBs are all considered 3D BLTs and are covered in the 3D rendering section. DIBs can be thought of as an indexed texture which uses the texture palette for performing the data translation. All drawing engines have swappable context. The same hardware can be used by multiple driver threads where the current state of the hardware is saved to memory and the appropriate state is loaded from memory on thread switches.

All operands for both 3D and classical BLTs can be in graphics aperture or cacheable system memory. Some operands can be immediates which are sent through the command stream. Immediate operands are: patterns, monochrome sources, DIB palettes, and DIB source operands. All non-monochrome operands which are not tiled have a stride granularity of a double-word (4 bytes).

The classical BLT commands support both linear addressing and X, Y coordinates with and without clipping. All X1 and Y1 destination and clipping coordinates are inclusive, while X2 and Y2 are exclusive. Currently, only destination coordinates can be negative. The source and clipping coordinates must be positive. If clipping is disabled, but a negative destination coordinate is specified, the negative coordinate is clipped to 0. Linear address BLT commands must supply a non-zero height and width. If either height or width = 0, then no accesses occur.

8.2 Classical BLT Engine Functional Description

The graphics controller provides a hardware-based BLT engine to off load the work of moving blocks of graphics data from the host CPU. Although the BLT engine is often used simply to copy a block of graphics data from the source to the destination, it also has the ability to perform more complex functions. The BLT engine is capable of receiving three different blocks of graphics data as input as shown in the figure below. The source data may exist in the frame buffer or the Graphics aperture. The pattern data always represents an 8x8 block of pixels that can be located in the frame buffer, Graphics aperture, or passed through a command packet. The pattern data must be located in linear memory.. The data already residing at the destination may also be used as an input. The destination data can also be located in the frame buffer or graphics aperture.

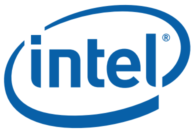
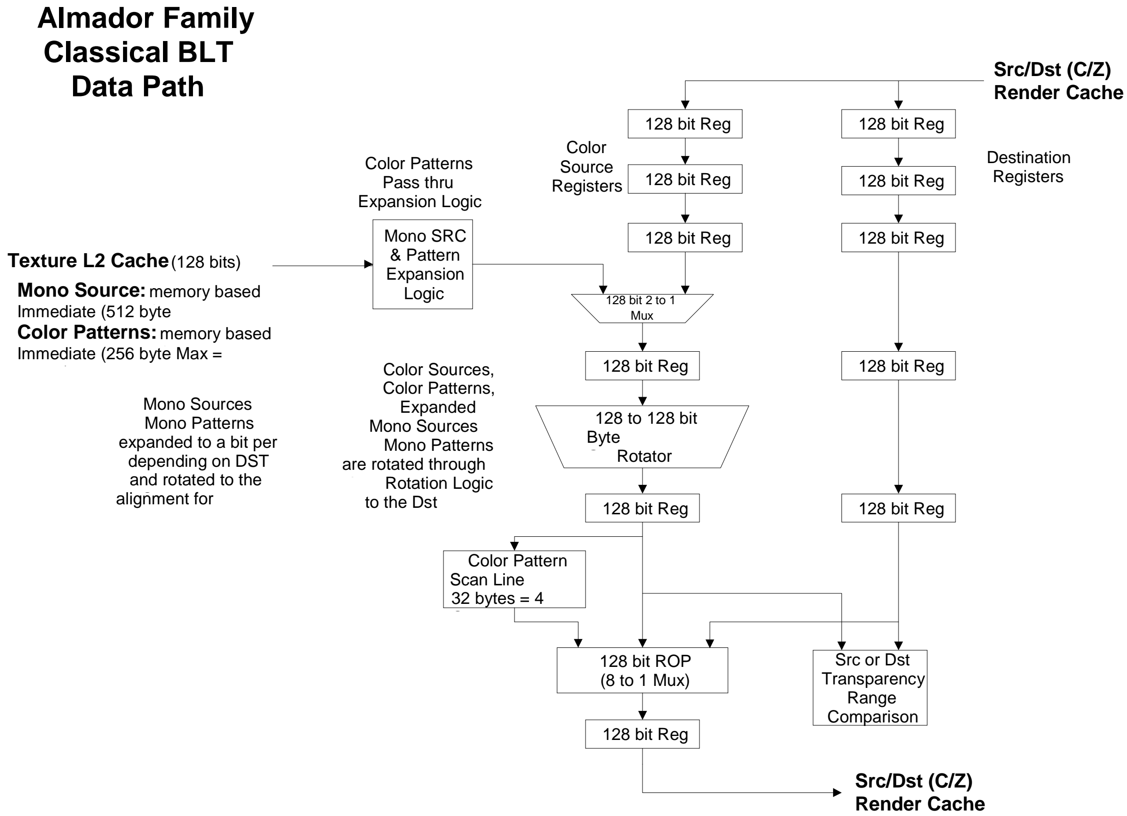
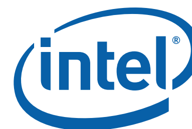


Figure 8-1. Block Diagram and Data Paths of the BLT Engine



The BLT engine may use any combination of these three different blocks of graphics data as operands, in both bit-wise logical operations to generate the actual data to be written to the destination, and in per-pixel write-masking to control the writing of data to the destination. It is intended that the BLT engine will perform these bit-wise and per-pixel operations on color graphics data that is at the same color depth that the rest of the graphics system has been set. However, if either the source or pattern data is monochrome, the BLT engine has the ability to put either block of graphics data through a process called "color expansion" that converts monochrome graphics data to color. Since the destination is often a location in the on-screen portion of the frame buffer, it is assumed that any data already at the destination will be of the appropriate color depth.



8.2.1 Basic BLT Functional Considerations

8.2.1.1 Color Depth Configuration and Color Expansion

The graphics system and BLT engine can be configured for color depths of 8, 16, and 32 bits per pixel.

The configuration of the BLT engine for a given color depth dictates the number of bytes of graphics data that the BLT engine will read and write for each pixel while performing a BLT operation. It is assumed that any graphics data already residing at the destination which is used as an input is already at the color depth to which the BLT engine is configured. Similarly, it is assumed that any source or pattern data used as an input has this same color depth, unless one or both is monochrome. If either the source or pattern data is monochrome, the BLT engine performs a process called "color expansion" to convert such monochrome data to color at the color depth to which the BLT engine has been set.

During "color expansion" the individual bits of monochrome source or pattern data that correspond to individual pixels are converted into 1, 2, or 4 bytes (which ever is appropriate for the color depth to which the BLT engine has been set). If a given bit of monochrome source or pattern data carries a value of 1, then the byte(s) of color data resulting from the conversion process are set to carry the value of a specified foreground color. If a given bit of monochrome source or pattern data carries a value of 0, the resulting byte(s) are set to the value of a specified background color or not written if transparency is selected.

The BLT engine is set to a default configuration color depth of 8, 16, or 32 bits per pixel through BLT command packets. Whether the source and pattern data are color or monochrome must be specified using command packets. Foreground and background colors for the color expansion of both monochrome source and pattern data are also specified through the command packets. The source foreground and background colors used in the color expansion of monochrome source data are specified independently of those used for the color expansion of monochrome pattern data.

8.2.1.2 Graphics Data Size Limitations

The BLT engine is capable of transferring very large quantities of graphics data. Any graphics data read from and written to the destination is permitted to represent a number of pixels that occupies up to 65,536 scan lines and up to 32,768 bytes per scan line at the destination. The maximum number of pixels that may be represented per scan line's worth of graphics data depends on the color depth.

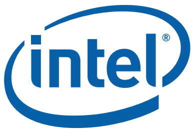
Any source data used as an input must represent the same number of pixels as is represented by any data read from or written to the destination, and it must be organized so as to occupy the same number of scan lines and pixels per scan line.

The actual number of scan lines and bytes per scan line required to accommodate data read from or written to the destination are set in the destination width & height registers or using X and Y coordinates within the command packets. These two values are essential in the programming of the BLT engine, because the engine uses these two values to determine when a given BLT operation has been completed.

8.2.1.3 Bit-Wise Operations

The BLT engine can perform any one of 256 possible bit-wise operations using various combinations of the three previously described blocks of graphics data that the BLT engine can receive as input. These 256 possible bit-wise operations are designed to be compatible with the manner in which raster operations are specified in the standard BLT parameter without translation.

The choice of bit-wise operation selects which of the three inputs will be used, as well as the particular logical operation to be performed on corresponding bits from each of the selected inputs. The BLT engine automatically foregoes reading any form of graphics data that has not been specified as an input by the choice



of bit-wise operation. An 8-bit code written to the raster operation field of the command packets chooses the bit-wise operation. The following table lists the available bit-wise operations and their corresponding 8-bit codes.

Table 8-1. Bit-Wise Operations and 8-Bit Codes (00-3F)

Code	Value Written to Bits at Destination
00	writes all 0's
01	not(D or (P or S)))
02	D and (not(P or S))
03	not(P or S)
04	S and (not(D or P))
05	not(D or P)
06	not(P or (not(D xor S)))
07	not(P or (D and S))
08	S and (D and (notP))
09	not(P or (D xor S))
0A	D and (notP)
0B	not(P or (S and (notD)))
0C	S and (notP)
0D	not(P or (D and (notS)))
0E	not(P or (not(D or S)))
0F	notP
10	P and (not(D or S))
11	not(D or S)
12	not(S or (not(D xor P)))
13	not(S or (D and P))
14	not(D or (not(P xor S)))
15	not(D or (P and S))
16	P xor (S xor (D and (not(P and S))))
17	not(S xor ((S xor P) and (D xor S)))
18	(S xor P) and (P xor D)
19	not(S xor (D and (not(P and S))))
1A	P xor (D or (S and P))
1B	not(S xor (D and (P xor S)))
1C	P xor (S or (D and P))
1D	not(D xor (S and (P xor D)))
1E	P xor (D or S)
1F	not(P and (D or S))

Code	Value Written to Bits at Destination
20	D and (P and (notS))
21	not(S or(D xor P))
22	D and (notS)
23	not(S or (P and (notD)))
24	(S xor P) and (D xor S)
25	not(P xor (D and (not(S and P))))
26	S xor (D or (P and S))
27	S xor (D or (not(P xor S)))
28	D and (P xor S)
29	not(P xor (S xor (D or (P and S))))
2A	D and (not(P and S))
2B	not(S xor ((S xor P) and (P xor D)))
2C	S xor (P and (D or S))
2D	P xor (S or (notD))
2E	P xor (S or (D xor P))
2F	not(P and (S or (notD)))
30	P and (notS)
31	not(S or (D and (notP)))
32	S xor (D or (P or S))
33	notS
34	S xor (P or (D and S))
35	S xor (P or (not(D xor S)))
36	S xor (D or P)
37	not(S and (D or P))
38	P xor (S and (D or P))
39	S xor (P or (notD))
3A	S xor (P or (D xor S))
3B	not(S and (P or (notD)))
3C	P xor S
3D	S xor (P or (not(D or S)))
3E	S xor (P or (D and (notS)))
3F	not(P and S)

Notes: S = Source Data
P = Pattern Data
D = Data Already Existing at the Destination

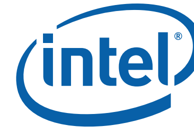


Table 8-2. Bit-Wise Operations and 8-bit Codes (40 - 7F)

Code	Value Written to Bits at Destination	Code	Value Written to Bits at Destination
40	P and (S and (notD))	60	P and (D xor S)
41	not(D or (P xor S))	61	not(D xor (S xor (P or (D and S))))
42	(S xor D) and (P xor D)	62	D xor (S and (P or D))
43	not(S xor (P and (not(D and S))))	63	S xor (D or (notP))
44	S and (notD)	64	S xor (D and (P or S))
45	not(D or (P and (notS)))	65	D xor (S or (notP))
46	D xor (S or (P and D))	66	D xor S
47	not(P xor (S and (D xor P)))	67	S xor (D or (not(P or S)))
48	S and (D xor P)	68	not(D xor (S xor (P or (not(D or S)))))
49	not(P xor (D xor (S or (P and D))))	69	not(P xor (D xor S))
4A	D xor (P and (S or D))	6A	D xor (P and S)
4B	P xor (D or (notS))	6B	not(P xor (S xor (D and (P or S))))
4C	S and (not(D and P))	6C	S xor (D and P)
4D	not(S xor ((S xor P) or (D xor S)))	6D	not(P xor (D xor (S and (P or D))))
4E	P xor (D or (S xor P))	6E	S xor (D and (P or (notS)))
4F	not(P and (D or (notS)))	6F	not(P and (not(D xor S)))
50	P and (notD)	70	P and (not(D and S))
51	not(D or (S and (notP)))	71	not(S xor ((S xor D) and (P xor D)))
52	D xor (P or (S and D))	72	S xor (D or (P xor S))
53	not(S xor (P and (D xor S)))	73	not(S and (D or (notP)))
54	not(D or (not(P or S)))	74	D xor (S or (P xor D))
55	notD	75	not(D and (S or (notP)))
56	D xor (P or S)	76	S xor (D or (P and (notS)))
57	not(D and (P or S))	77	not(D and S)
58	P xor (D and (S or P))	78	P xor (D and S)
59	D xor (P or (notS))	79	not(D xor (S xor (P and (D or S))))
5A	D xor P	7A	D xor (P and (S or (notD)))
5B	D xor (P or (not(S or D)))	7B	not(S and (not(D xor P)))
5C	D xor (P or (S xor D))	7C	S xor (P and (D or (notS)))
5D	not(D and (P or (notS)))	7D	not(D and (not(P xor S)))
5E	D xor (P or (S and (notD)))	7E	(S xor P) or (D xor S)
5F	not(D and P)	7F	not(D and (P and S))

Notes: S = Source Data
P = Pattern Data
D = Data Already Existing at the Destination

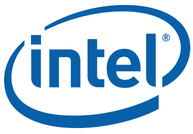


Table 8-3. Bit-Wise Operations and 8-bit Codes (80 - BF)

Code	Value Written to Bits at Destination	Code	Value Written to Bits at Destination
80	D and (P and S)	A0	D and P
81	not((S xor P) or (D xor S))	A1	not(P xor (D or (S and (notP))))
82	D and (not(P xor S))	A2	D and (P or (notS))
83	not(S xor (P and (D or (notS))))	A3	not(D xor (P or (S xor D)))
84	S and (not(D xor P))	A4	not(P xor (D or (not(S or P))))
85	not(P xor (D and (S or (notP))))	A5	not(P xor D)
86	D xor (S xor (P and (D or S)))	A6	D xor (S and (notP))
87	not(P xor (D and S))	A7	not(P xor (D and (S or P)))
88	D and S	A8	D and (P or S)
89	not(S xor (D or (P and (notS))))	A9	not(D xor (P or S))
8A	D and (S or (notP))	AA	D
8B	not(D xor (S or (P xor D)))	AB	D or (not(P or S))
8C	S and (D or (notP))	AC	S xor (P and (D xor S))
8D	not(S xor (D or (P xor S)))	AD	not(D xor (P or (S and D)))
8E	S xor ((S xor D) and (P xor D))	AE	D or (S and (notP))
8F	not(P and (not(D and S)))	AF	D or (notP)
90	P and (not(D xor S))	B0	P and (D or (notS))
91	not(S xor (D and (P or (notS))))	B1	not(P xor (D or (S xor P)))
92	D xor (P xor (S and (D or P)))	B2	S xor ((S xor P) or (D xor S))
93	not(S xor (P and D))	B3	not(S and (not(D and P)))
94	P xor (S xor (D and (P or S)))	B4	P xor (S and (notD))
95	not(D xor (P and S))	B5	not(D xor (P and (S or D)))
96	D xor (P xor S)	B6	D xor (P xor (S or (D and P)))
97	P xor (S xor (D or (not(P or S))))	B7	not(S and (D xor P))
98	not(S xor (D or (not(P or S))))	B8	P xor (S and (D xor P))
99	not(D xor S)	B9	not(D xor (S or (P and D)))
9A	D xor (P and (notS))	BA	D or (P and (notS))
9B	not(S xor (D and (P or S)))	BB	D or (notS)
9C	S xor (P and (notD))	BC	S xor (P and (not(D and S)))
9D	not(D xor (S and (P or D)))	BD	not((S xor D) and (P xor D))
9E	D xor (S xor (P or (D and S)))	BE	D or (P xor S)
9F	not(P and (D xor S))	BF	D or (not(P and S))

Notes: S = Source Data
P = Pattern Data
D = Data Already Existing at the Destination

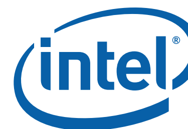


Table 8-4. Bit-Wise Operations and 8-bit Codes (C0 - FF)

Code	Value Written to Bits at Destination
C0	P and S
C1	not(S xor (P or (D and (notS))))
C2	not(S xor (P or (not(D or S))))
C3	not(P xor S)
C4	S and (P or (notD))
C5	not(S xor (P or (D xor S)))
C6	S xor (D and (notP))
C7	not(P xor (S and (D or P)))
C8	S and (D or P)
C9	not(S xor (P or D))
CA	D xor (P and (S xor D))
CB	not(S xor (P or (D and S)))
CC	S
CD	S or (not(D or P))
CE	S or (D and (notP))
CF	S or (notP)
D0	P and (S or (notD))
D1	not(P xor (S or (D xor P)))
D2	P xor (D and (notS))
D3	not(S xor (P and (D or S)))
D4	S xor ((S xor P) and (P xor D))
D5	not(D and (not(P and S)))
D6	P xor (S xor (D or (P and S)))
D7	not(D and (P xor S))
D8	P xor (D and (S xor P))
D9	not(S xor (D or (P and S)))
DA	D xor (P and (not(S and D)))
DB	not((S xor P) and (D xor S))
DC	S or (P and (notD))
DD	S or (notD)
DE	S or (D xor P)
DF	S or (not(D and P))

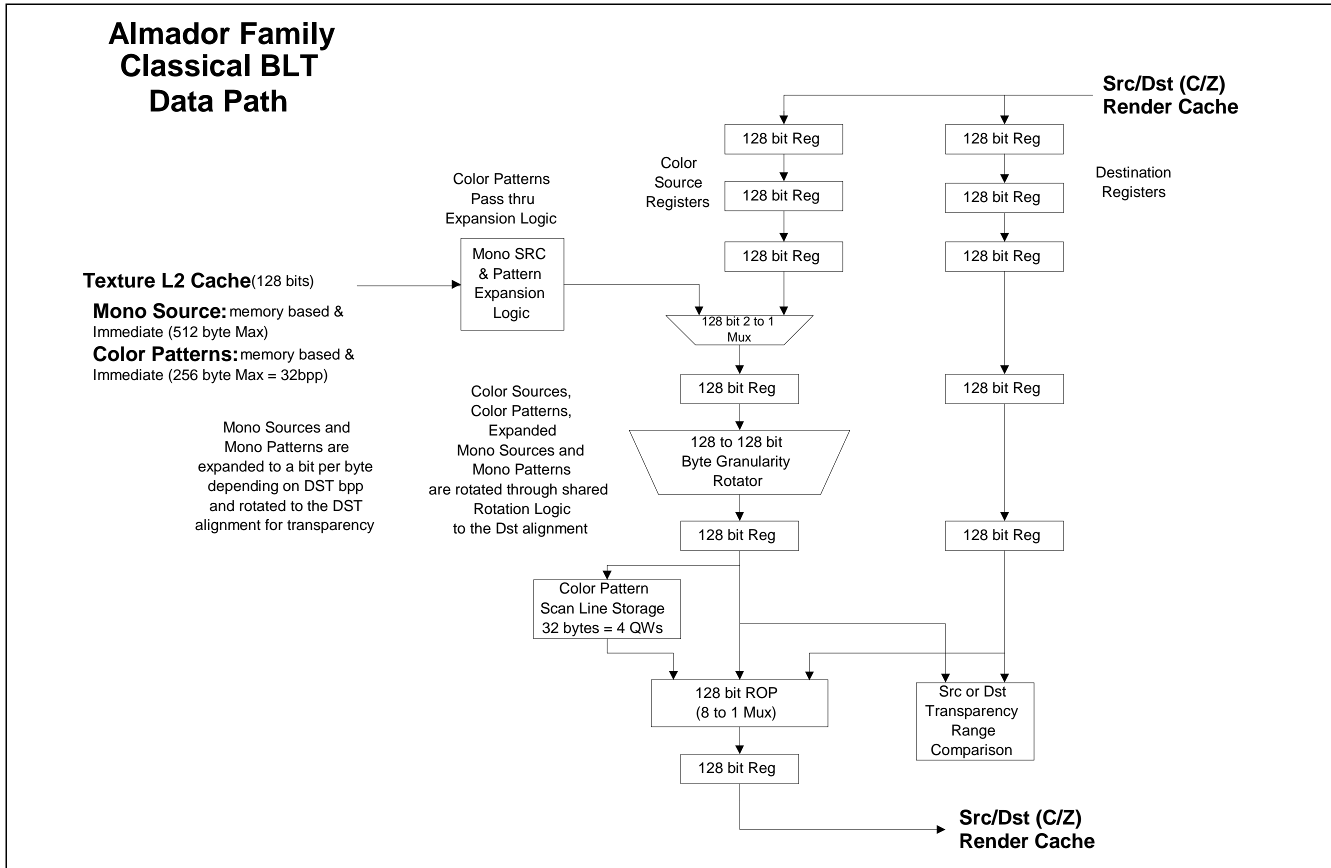
Code	Value Written to Bits at Destination
E0	P and (D or S)
E1	not(P xor (D or S))
E2	D xor (S and (P xor D))
E3	not(P xor (S or (D and P)))
E4	S xor (D and (P xor S))
E5	not(P xor (D or (S and P)))
E6	S xor (D and (not(P and S)))
E7	not((S xor P) and (P xor D))
E8	S xor ((S xor P) and (D xor S))
E9	not(D xor (S xor (P and (not(D and S)))))
EA	D or (P and S)
EB	D or (not(P xor S))
EC	S or (D and P)
ED	S or (not(D xor P))
EE	D or S
EF	S or (D or (notP))
F0	P
F1	P or (not(D or S))
F2	P or (D and (notS))
F3	P or (notS)
F4	P or (S and (notD))
F5	P or (notD)
F6	P or (D xor S)
F7	P or (not(D and S))
F8	P or (D and S)
F9	P or (not(D xor S))
FA	D or P
FB	D or (P or (notS))
FC	P or S
FD	P or (S or (notD))
FE	D or (P or S)
FF	writes all 1's

Notes: S = Source Data
P = Pattern Data
D = Data Already Existing at the Destination

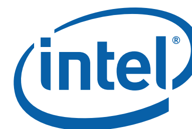
8.2.1.4 Per-Pixel Write-Masking Operations

The BLT engine is able to perform per-pixel write-masking with various data sources used as pixel masks to constrain which pixels at the destination are to be written to by the BLT engine. As shown in the figure below, either monochrome source or monochrome pattern data may be used as pixel masks. Color pattern data cannot be used. Another available pixel mask is derived by comparing a particular color range per color channel to either the color already specified for a given pixel at the destination or source.

Figure 8-2. Block Diagram and Data Paths of the BLT Engine



The command packets can specify the monochrome source or the monochrome pattern data as a pixel mask. When this feature is used, the bits that carry a value of 0 cause the bytes of the corresponding pixel at the destination to not be written to by the BLT engine, thereby preserving whatever data was originally carried within those bytes. This feature can be used in writing characters to the display, while also preserving the pre-existing backgrounds behind those characters. When both operands are in the transparent mode, the logical AND of the 2 operands are used for the write enables per pixel.



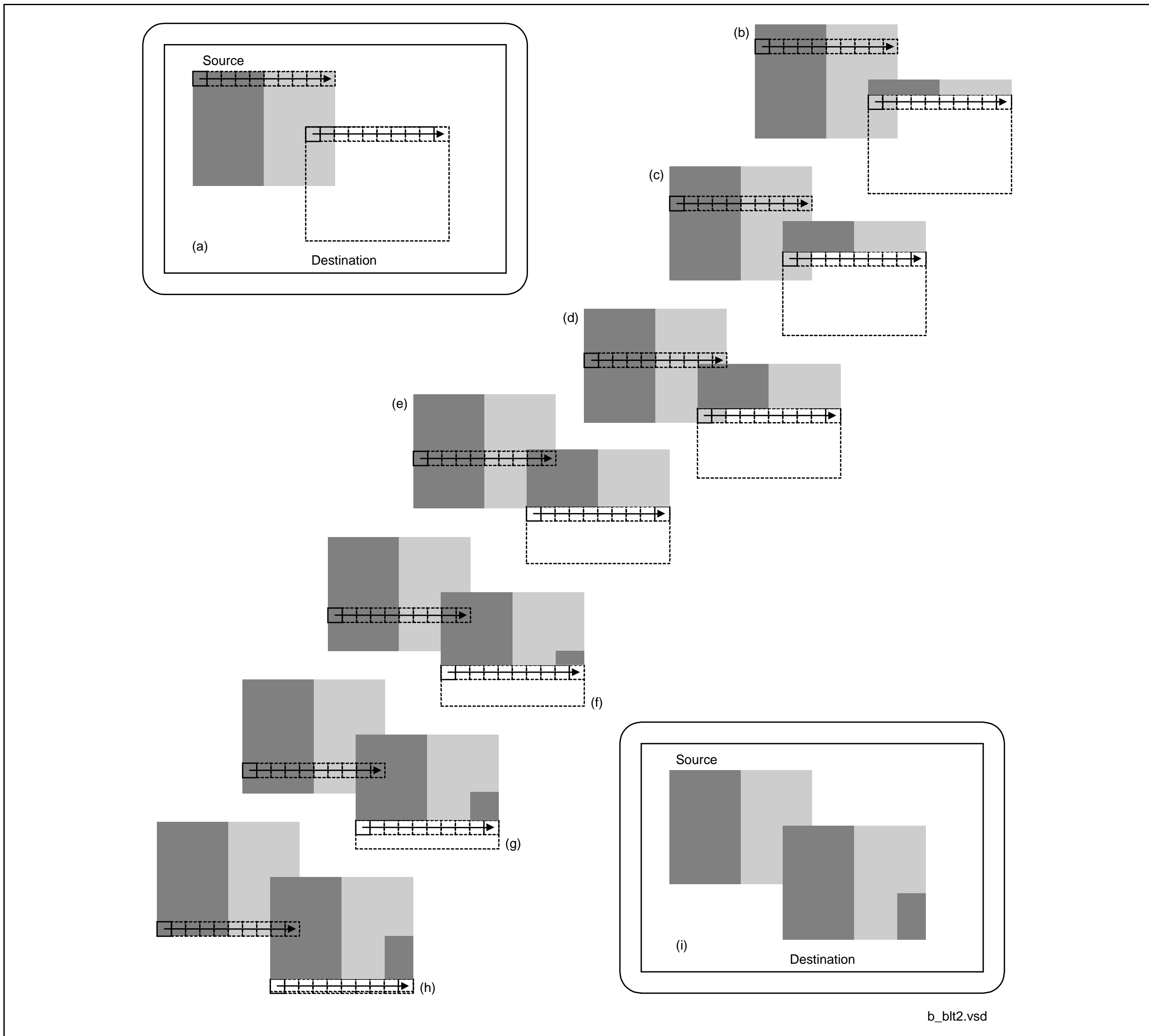
The 3-bit field, destination transparency mode, within the command packets can select per-pixel write-masking with a mask based on the results of color comparisons. The monochrome source background and foreground are range compared with either the bytes for the pixels at the destination or the source operand. This operation is described in the BLT command packet and register descriptions.

8.2.1.5 When the Source and Destination Locations Overlap

It is possible to have BLT operations in which the locations of the source and destination data overlap. This frequently occurs in BLT operations where a user is shifting the position of a graphical item on the display by only a few pixels. In these situations, the BLT engine must be programmed so that destination data is not written into destination locations that overlap with source locations before the source data at those locations has been read. Otherwise, the source data will become corrupted. The XY commands determine whether there is an overlap and perform the accesses in the proper direction to avoid data corruption.

The following figure shows how the source data can be corrupted when a rectangular block is copied from a source location to an overlapping destination location. The BLT engine typically reads from the source location and writes to the destination location starting with the left-most pixel in the top-most line of both, as shown in step (a). As shown in step (b), corruption of the source data has already started with the copying of the top-most line in step (a) — part of the source that originally contained lighter-colored pixels has now been overwritten with darker-colored pixels. More source data corruption occurs as steps (b) through (d) are performed. At step (e), another line of the source data is read, but the two right-most pixels of this line are in the region where the source and destination locations overlap, and where the source has already been overwritten as a result of the copying of the top-most line in step (a). Starting in step (f), darker-colored pixels can be seen in the destination where lighter-colored pixels should be. This errant effect occurs repeatedly throughout the remaining steps in this BLT operation. As more lines are copied from the source location to the destination location, it becomes clear that the end result is not what was originally intended.

Figure 8-3. Source Corruption in BLT with Overlapping Source and Destination Locations



The BLT engine can alter the order in which source data is read and destination data is written when necessary to avoid source data corruption problems when the source and destination locations overlap. The command packets provide the ability to change the point at which the BLT engine begins reading and writing data from the upper left-hand corner (the usual starting point) to one of the other three corners. The BLT engine may be set to read data from the source and write it to the destination starting at any of the four corners of the panel.

The XY command packets perform the necessary comparisons and start at the proper corner of each operand which avoids data corruption.

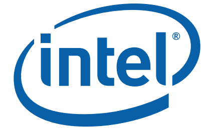
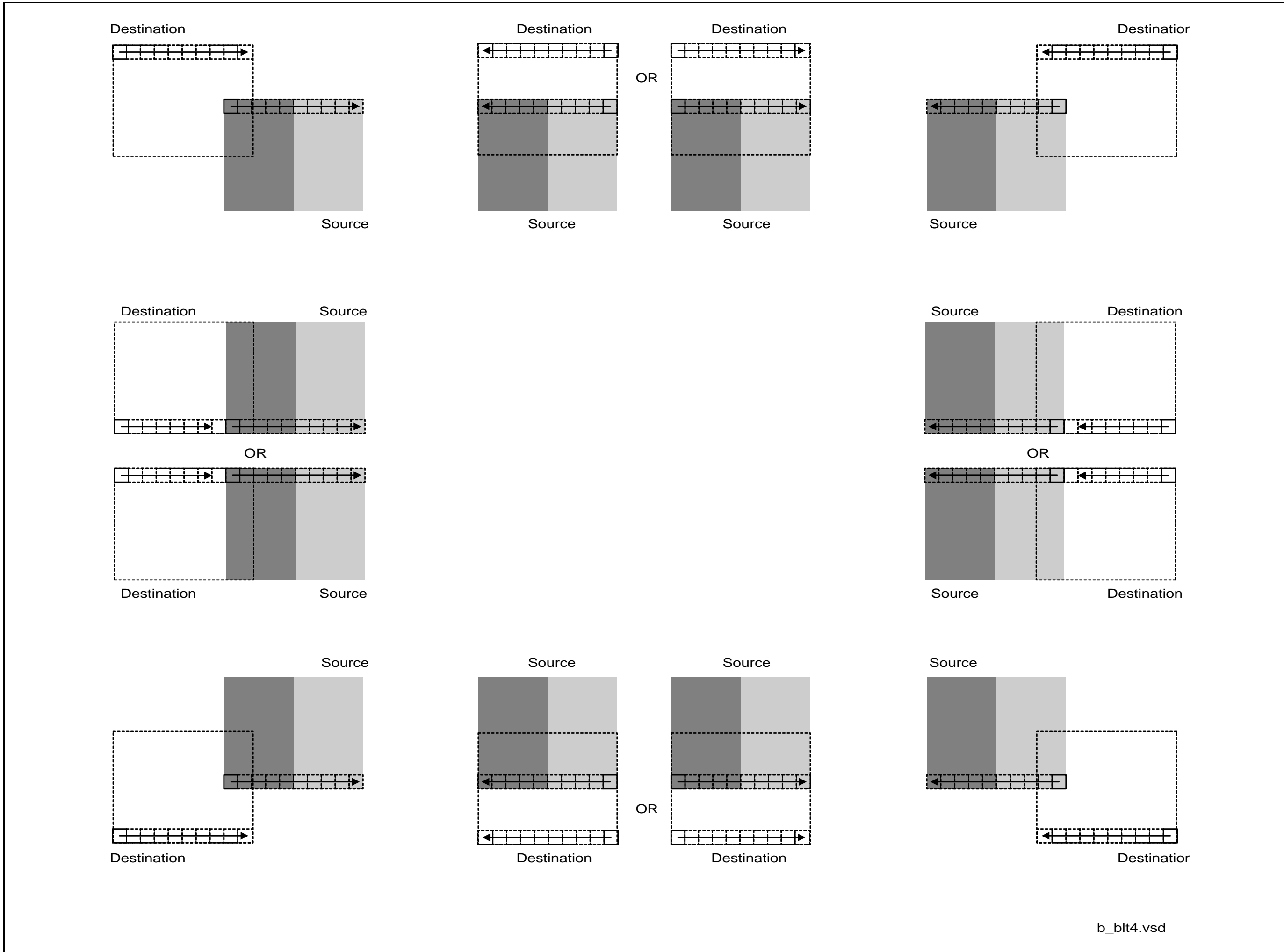


Figure 8-4. Correctly Performed BLT with Overlapping Source and Destination Locations

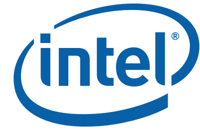
Source
(a)
Destination
(b)
(c)
(d)
(e)
(f)
(g)
(h)
Source
(i)
Destination
b_blt3.vsd

The following figure illustrates how this feature of the BLT engine can be used to perform the same BLT operation as was illustrated in the figure above, while avoiding the corruption of source data. As shown in the figure below, the BLT engine reads the source data and writes the data to the destination starting with the right-most pixel of the bottom-most line. By doing this, no pixel existing where the source and destination locations overlap will ever be written to before it is read from by the BLT engine. By the time the BLT operation has reached step (e) where two pixels existing where the source and destination locations overlap are about to be over written, the source data for those two pixels has already been read.

Figure 8-5. Suggested Starting Points for Possible Source and Destination Overlap Situations



The figure above shows the recommended lines and pixels to be used as starting points in each of 8 possible ways in which the source and destination locations may overlap. In general, the starting point should be within the area in which the source and destination overlap.

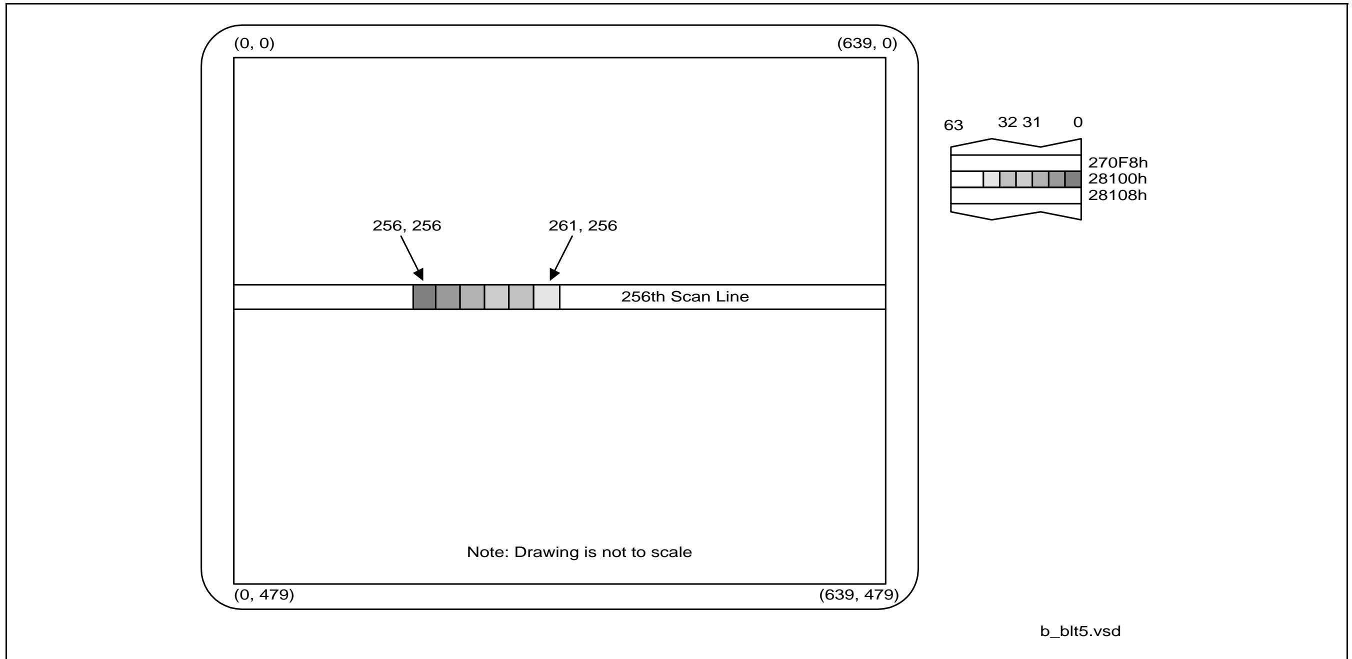


8.2.2 Basic Graphics Data Considerations

8.2.2.1 Contiguous vs. Discontinuous Graphics Data

Graphics data stored in memory, particularly in the frame buffer of a graphics system, has organizational characteristics that often distinguish it from other varieties of data. The main distinctive feature is the tendency for graphics data to be organized in a discontinuous block of graphics data made up of multiple sub-blocks of bytes, instead of a single contiguous block of bytes.

Figure 8-6. Representation of On-Screen Single 6-Pixel Line in the Frame Buffer

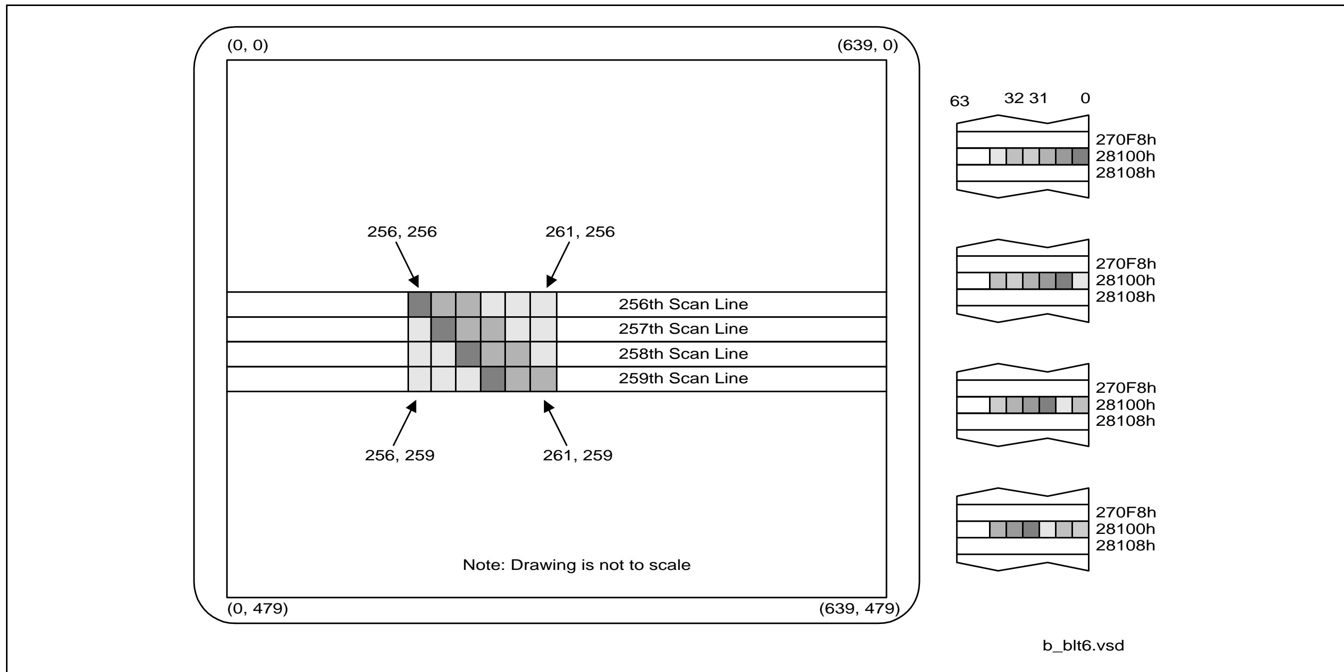


The figure above shows an example of contiguous graphics data — a horizontal line made up of six adjacent pixels within a single scan line on a display with a resolution of 640x480. Presuming that the graphics system driving this display has been set to 8 bits per pixel and that the frame buffer's starting address of 0h corresponds to the upper left-most pixel of this display, then the six pixels that make this horizontal line starting at coordinates (256, 256) occupies the six bytes starting at frame buffer address 28100h, and ending at address 28105h.

In this case, there is only one scan line's worth of graphics data in this single horizontal line, so the block of graphics data for all six of these pixels exists as a single, contiguous block comprised of only these six bytes. The starting address and the number of bytes are the only pieces of information that a BLT engine would require to read this block of data.

The simplicity of the above example of a single horizontal line contrasts sharply to the example of discontinuous graphics data depicted in the figure below. The simple six-pixel line of the figure above is now accompanied by three more six-pixel lines placed on subsequent scan lines, resulting in the 6x4 block of pixels shown.

Figure 8-7. Representation of On-Screen 6x4 Array of Pixels in the Frame Buffer



Since there are other pixels on each of the scan lines on which this 6x4 block exists that are not part of this 6x4 block, what appears to be a single 6x4 block of pixels on the display must be represented by a discontinuous block of graphics data made up of 4 separate sub-blocks of six bytes apiece in the frame buffer at addresses 28100h, 28380h, 28600h, and 28880h. This situation makes the task of reading what appears to be a simple 6x4 block of pixels more complex. However, there are two characteristics of this 6x4 block of pixels that help simplify the task of specifying the locations of all 24 bytes of this discontinuous block of graphics data: all four of the sub-blocks are of the same length, and the four sub-blocks are separated from each other at equal intervals.

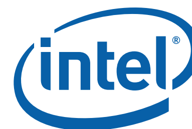
The BLT engine is designed to make use of these characteristics of graphics data to simplify the programming required to handle discontinuous blocks of graphics data. For such a situation, the BLT engine requires only four pieces of information: the starting address of the first sub-block, the length of a sub-block, the offset (in bytes), pitch, of the starting address of each subsequent sub-block, and the quantity of sub-blocks.

8.2.2.2 Source Data

The source data may exist in the frame buffer or elsewhere in the graphics aperture where the BLT engine may read it directly, or it may be provided to the BLT engine by the host CPU through the command packets. The block of source graphics data may be either contiguous or discontinuous, and may be either in color (with a color depth that matches that to which the BLT engine has been set) or monochrome.

The source select bit in the command packets specifies whether the source data exists in the frame buffer or is provided through the command packets. Monochrome source data is always specified as being supplied through an immediate command packet.

If the color source data resides within the frame buffer or elsewhere in the graphics aperture, then the Source Address Register, specified in the command packets is used to specify the address of the source.



In cases where the host CPU provides the source data, it does so by writing the source data to ring buffer directly after the BLT command that requires the data or uses an IMMEDIATE_INDIRECT_BLT command packet which has a size and pointer to the operand in Graphics aperture.

The block of bytes sent by the host CPU through the command packets must be quadword-aligned and the source data contained within the block of bytes must also be aligned.

To accommodate discontinuous source data, the source and destination pitch registers can be used to specify the offset in bytes from the beginning of one scan line's worth source data to the next. Otherwise, if the source data is contiguous, then an offset equal to the length of a scan line's worth of source data should be specified.

8.2.2.3 Monochrome Source Data

The opcode of the command packet specifies whether the source data is color or monochrome. Since monochrome graphics data only uses one bit per pixel, each byte of monochrome source data typically carries data for 8 pixels which hinders the use of byte-oriented parameters when specifying the location and size of valid source data. Some additional parameters must be specified to ensure the proper reading and use of monochrome source data by the BLT engine. The BLT engine also provides additional options for the manipulation of monochrome source data versus color source data.

The various bit-wise logical operations and per-pixel write-masking operations were designed to work with color data. In order to use monochrome data, the BLT engine converts it into color through a process called color expansion, which takes place as a BLT operation is performed. In color expansion the single bits of monochrome source data are converted into one, two, or four bytes (depending on the color depth) of color data that are set to carry value corresponding to either the foreground or background color that have been specified for use in this conversion process. If a given bit of monochrome source data carries a value of 1, then the byte(s) of color data resulting from the conversion process will be set to carry the value of the foreground color. If a given bit of monochrome source data carries a value of 0, then the resulting byte(s) will be set to the value of the background color. The foreground and background colors used in the color expansion of monochrome source data can be set in the source expansion foreground color register and the source expansion background color register.

The BLT Engine requires that the bit alignment of each scan line's worth of monochrome source data be specified. Each scan line's worth of monochrome source data is word aligned but can actually start on any bit boundary of the first byte. Monochrome text is special cased and it is bit or byte packed, where in bit packed there are no invalid pixels (bits) between scan lines. There is a 3 bit field which indicates the starting pixel position within the first byte for each scan line, Mono Source Start.

The BLT engine also provides various clipping options for use with specific BLT commands (BLT_TEXT) with a monochrome source. Clipping is supported through: Clip rectangle Y addresses or coordinates and X coordinates along with scan line starting and ending addresses (with Y addresses) along with X starting and ending coordinates.

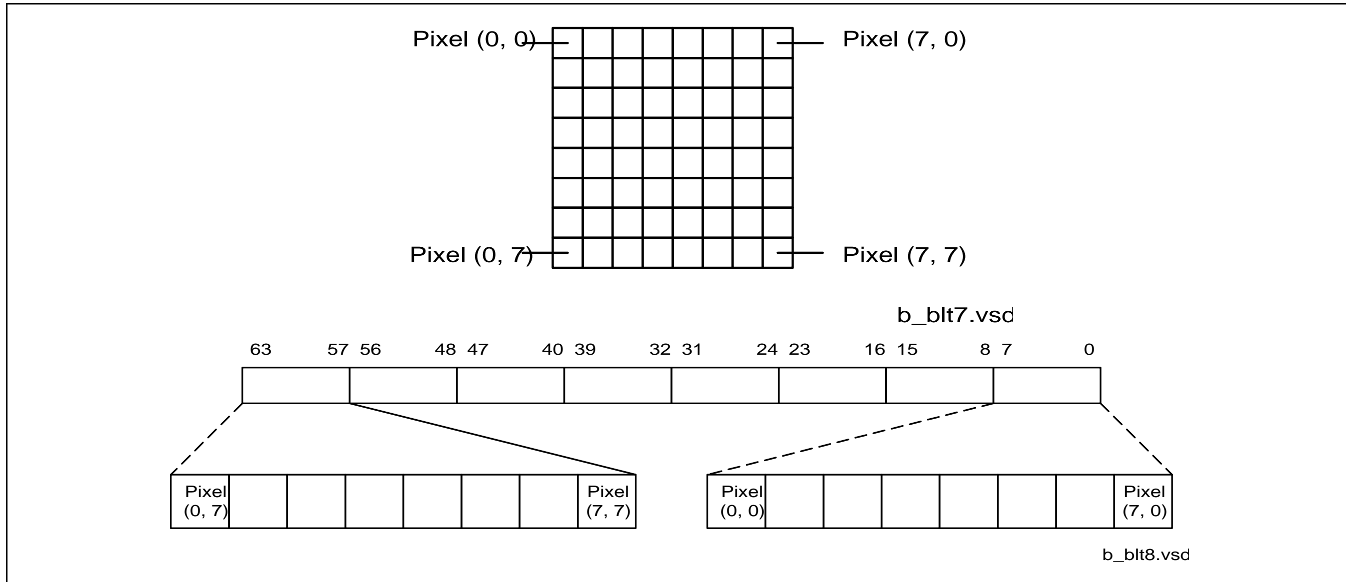
The maximum immediate source size is 128 bytes.

8.2.2.4 Pattern Data

The color pattern data must exist within the frame buffer or Graphics aperture where the BLT engine may read it directly or it can be sent through the command stream. The pattern data must be located in linear memory. Monochrome pattern data is supplied by the command packet when it is to be used. As shown in figure below, the block of pattern graphics data always represents a block of 8x8 pixels. The bits or bytes of a block of pattern data may be organized in the frame buffer memory in only one of three ways, depending upon its color depth which may be 8, 16, or 32 bits per pixel (whichever matches the color depth to which the BLT engine has been set), or monochrome.

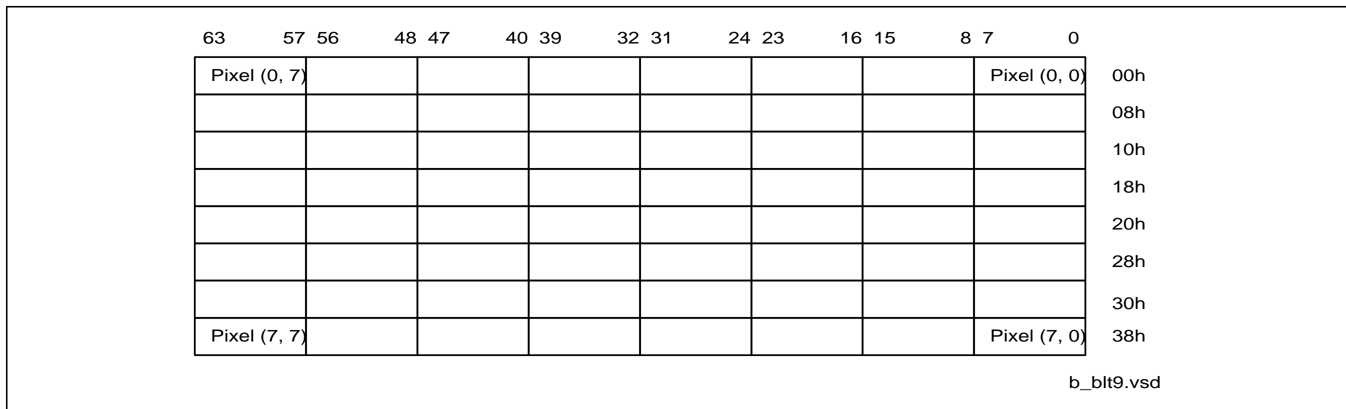
The maximum color pattern size is 256 bytes.

Figure 8-8. Pattern Data -- Always an 8x8 Array of Pixels



The Pattern Address Register is used to specify the address of the color pattern data at which the block of pattern data begins. The three least significant bits of the address written to this register are ignored, because the address must be in terms of quadwords. This is because the pattern must always be located on an address boundary equal to its size. Monochrome patterns take up 8 bytes, or a single quadword of space, and are loaded through the command packet that uses it. Similarly, color patterns with color depths of 8, 16, and 32 bits per pixel must start on 64-byte, 128-byte and 256-byte boundaries, respectively. The next 3 figures show how monochrome, 8bpp, 16bpp, and 32bpp pattern data , respectively, is organized in memory.

Figure 8-9. 8bpp Pattern Data -- Occupies 64 Bytes (8 quadwords)



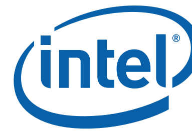


Figure 8-10. 16bpp Pattern Data -- Occupies 128 Bytes (16 quadwords)

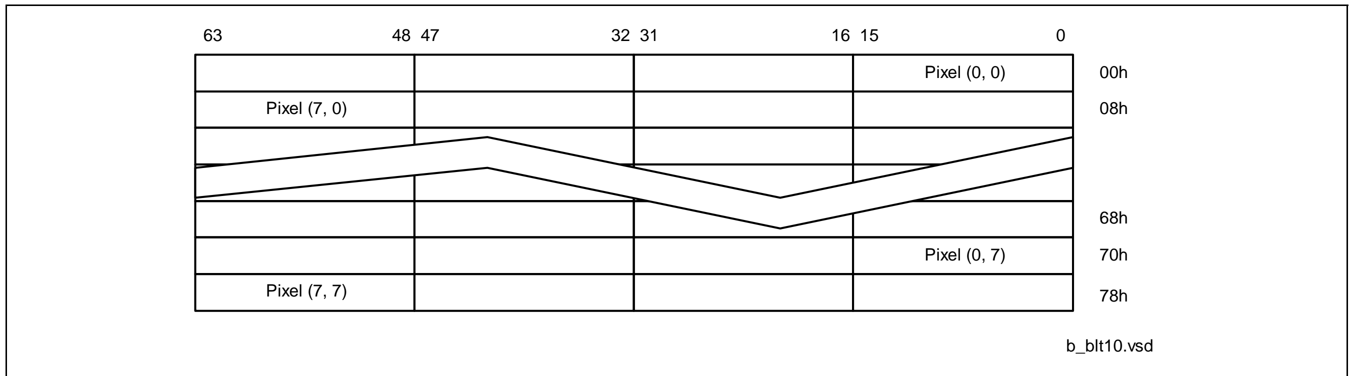
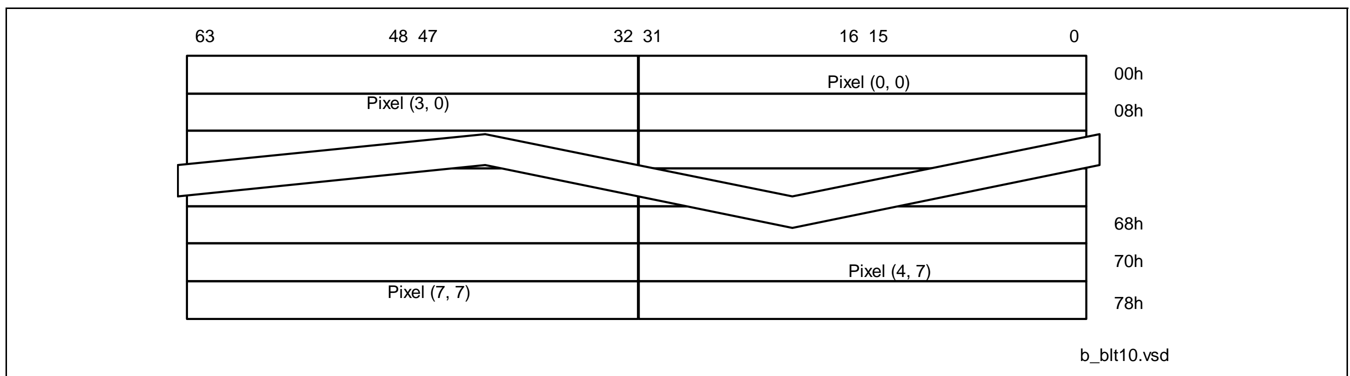


Figure 8-11. 32bpp Pattern Data -- Occupies 256 Bytes (32 quadwords)

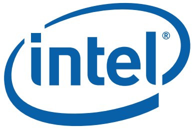


The opcode of the command packet specifies whether the pattern data is color or monochrome. The various bit-wise logical operations and per-pixel write-masking operations were designed to work with color data. In order to use monochrome pattern data, the BLT engine is designed to convert it into color through a process called "color expansion" which takes place as a BLT operation is performed. In color expansion, the single bits of monochrome pattern data are converted into one, two, or four bytes (depending on the color depth) of color data that are set to carry values corresponding to either the foreground or background color that have been specified for use in this process. The foreground color is used for pixels corresponding to a bit of monochrome pattern data that carry the value of 1, while the background color is used where the corresponding bit of monochrome pattern data carries the value of 0. The foreground and background colors used in the color expansion of monochrome pattern data can be set in the Pattern Expansion Foreground Color Register and Pattern Expansion Background Color Register.

8.2.2.5 Destination Data

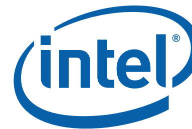
There are actually two different types of "destination data": the graphics data already residing at the location that is designated as the destination, and the data that is to be written into that very same location as a result of a BLT operation.

The location designated as the destination must be within the frame buffer or Graphics aperture where the BLT engine can read from it and write to it directly. The blocks of destination data to be read from and written to the destination may be either contiguous or discontinuous. All data written to the destination will have the color depth to which the BLT engine has been set. It is presumed that any data already existing at the destination which will be read by the BLT engine will also be of this same color depth — the BLT engine neither reads nor writes monochrome destination data.



The Destination Address Register is used to specify the address of the destination.

To accommodate discontinuous destination data, the Source and Destination Pitch Registers can be used to specify the offset in bytes from the beginning of one scan line's worth of destination data to the next. Otherwise, if the destination data is contiguous, then an offset equal to the length of a scan line's worth of destination data should be specified.

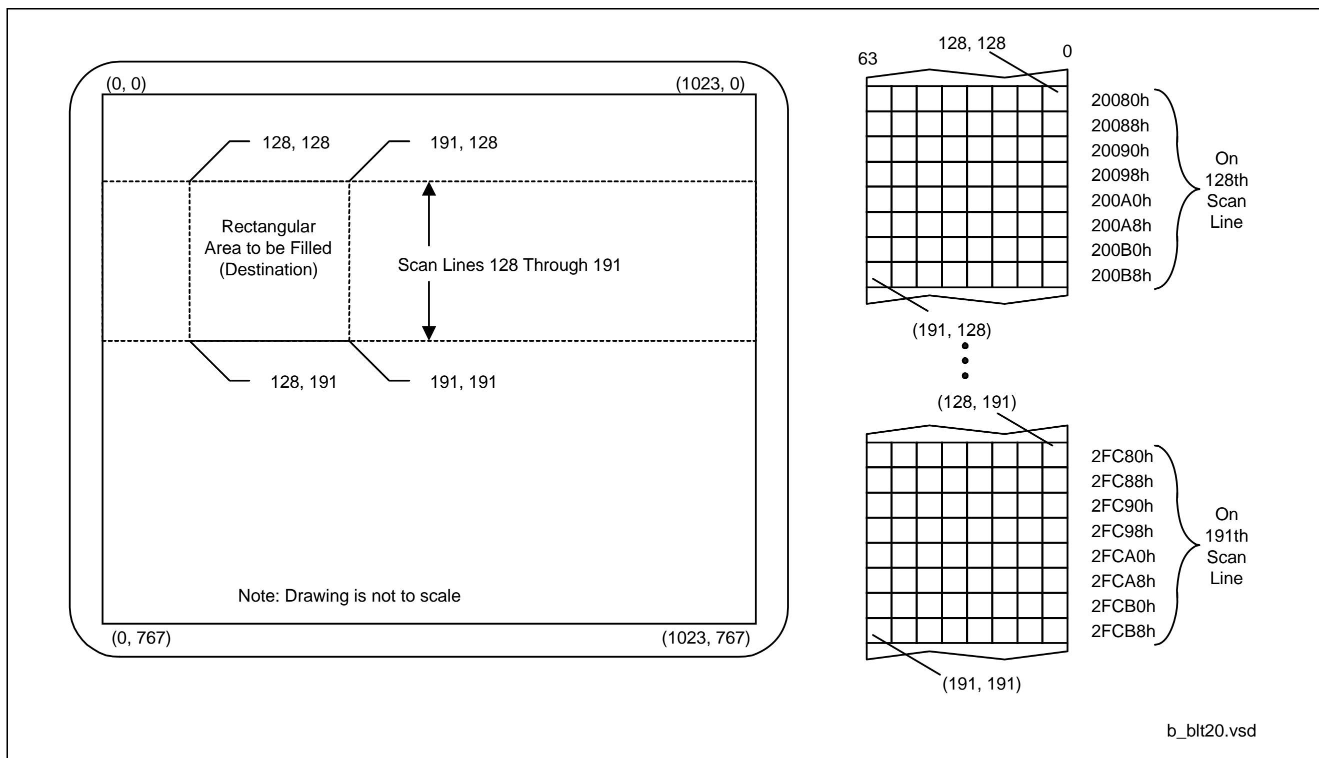


8.2.3 BLT Programming Examples

8.2.3.1 Pattern Fill — A Very Simple BLT

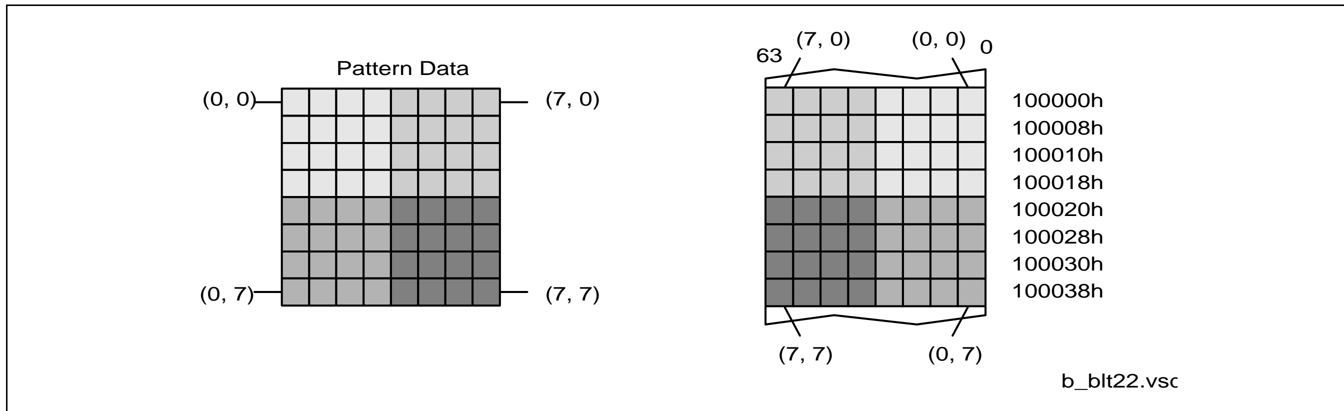
In this example, a rectangular area on the screen is to be filled with a color pattern stored as pattern data in off-screen memory. The screen has a resolution of 1024x768 and the graphics system has been set to a color depth of 8 bits per pixel.

Figure 8-12. On-Screen Destination for Example Pattern Fill BLT



As shown in the figure above, the rectangular area to be filled has its upper left-hand corner at coordinates (128, 128) and its lower right-hand corner at coordinates (191, 191). These coordinates define a rectangle covering 64 scan lines, each scan line's worth of which is 64 pixels in length — in other words, an array of 64x64 pixels. Presuming that the pixel at coordinates (0, 0) corresponds to the byte at address 00h in the frame buffer memory, the pixel at (128, 128) corresponds to the byte at address 20080h.

Figure 8-13. Pattern Data for Example Pattern Fill BLT



As shown in figure above, the pattern data occupies 64 bytes starting at address 100000h. As always, the pattern data represents an 8x8 array of pixels.

The BLT command packet is used to select the features to be used in this BLT operation, and must be programmed carefully. The vertical alignment field should be set to 0 to select the top-most horizontal row of the pattern as the starting row used in drawing the pattern starting with the top-most scan line covered by the destination. The pattern data is in color with a color depth of 8 bits per pixel, so the dynamic color enable should be asserted with the dynamic color depth field should be set to 0. Since this BLT operation does not use per-pixel write-masking (destination transparency mode), this field should be set to 0. Finally, the raster operation field should be programmed with the 8-bit value of F0h to select the bit-wise logical operation in which a simple copy of the pattern data to the destination takes place. Selecting this bit-wise operation in which no source data is used as an input causes the BLT engine to automatically forego either reading source data from the frame buffer.

The Destination Pitch Register must be programmed with number of bytes in the interval from the start of one scan line's worth of destination data to the next. Since the color depth is 8 bits per pixel and the horizontal resolution of the display is 1024, the value to be programmed into these bits is 400h, which is equal to the decimal value of 1024.

Bits [31:3] of the Pattern Address Register must be programmed with the address of the pattern data.

Similarly, bits [31:0] of the Destination Address Register must be programmed with the byte address at the destination that will be written to first. In this case, the address is 20080h, which corresponds to the byte representing the pixel at coordinates (128, 128).

This BLT operation does not use the values in the Source Address Register or the Source Expansion Background or Foreground Color Registers.

The Destination Width and Height Registers (or the Destination X and Y Coordinates) must be programmed with values that describe to the BLT engine the 64x64 pixel size of the destination location. The height should be set to carry the value of 40h, indicating that the destination location covers 64 scan lines. The width should be set to carry the value of 40h, indicating that each scan line's worth of destination data occupies 64 bytes. All of this information is written to the ring buffer using the PAT_BLT (or XY_PAT_BLT) command packet.

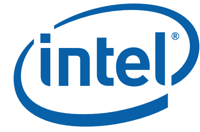
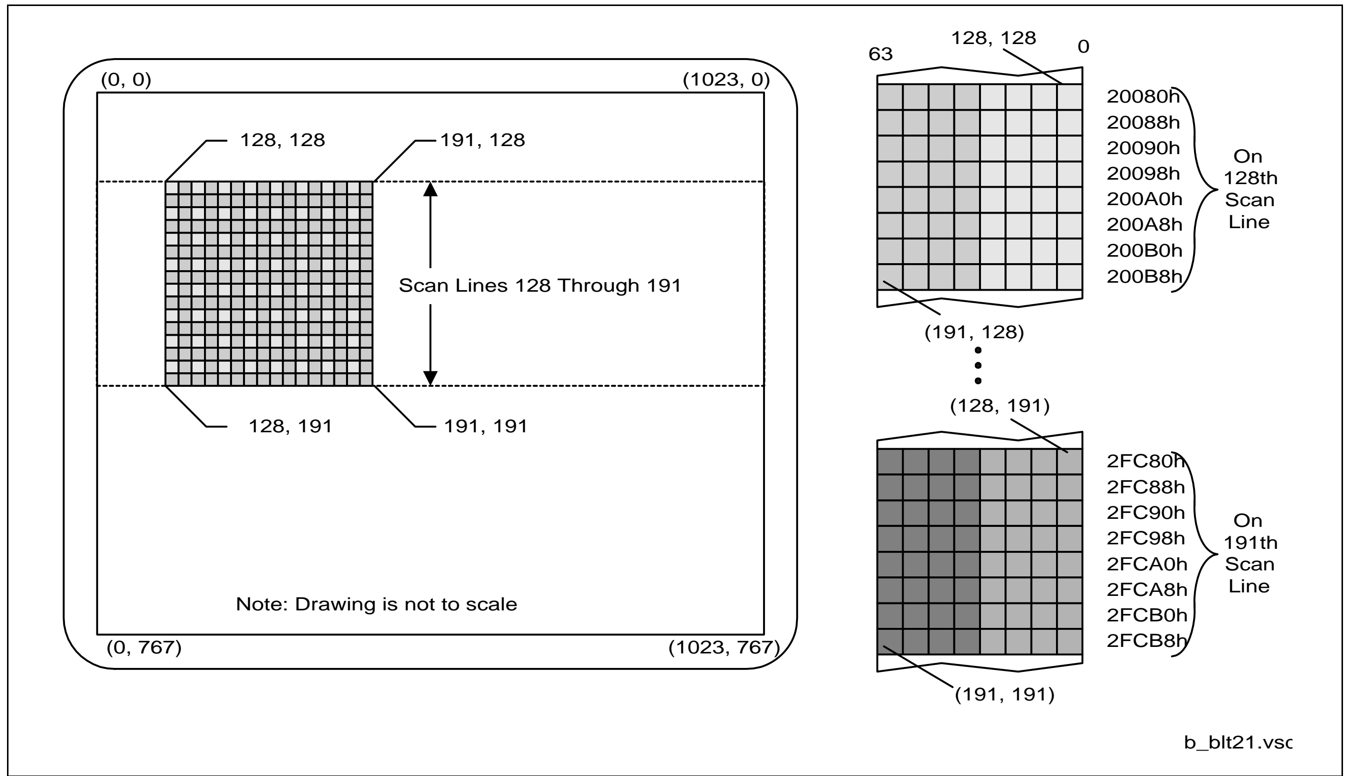


Figure 8-14. Results of Example Pattern Fill BLT

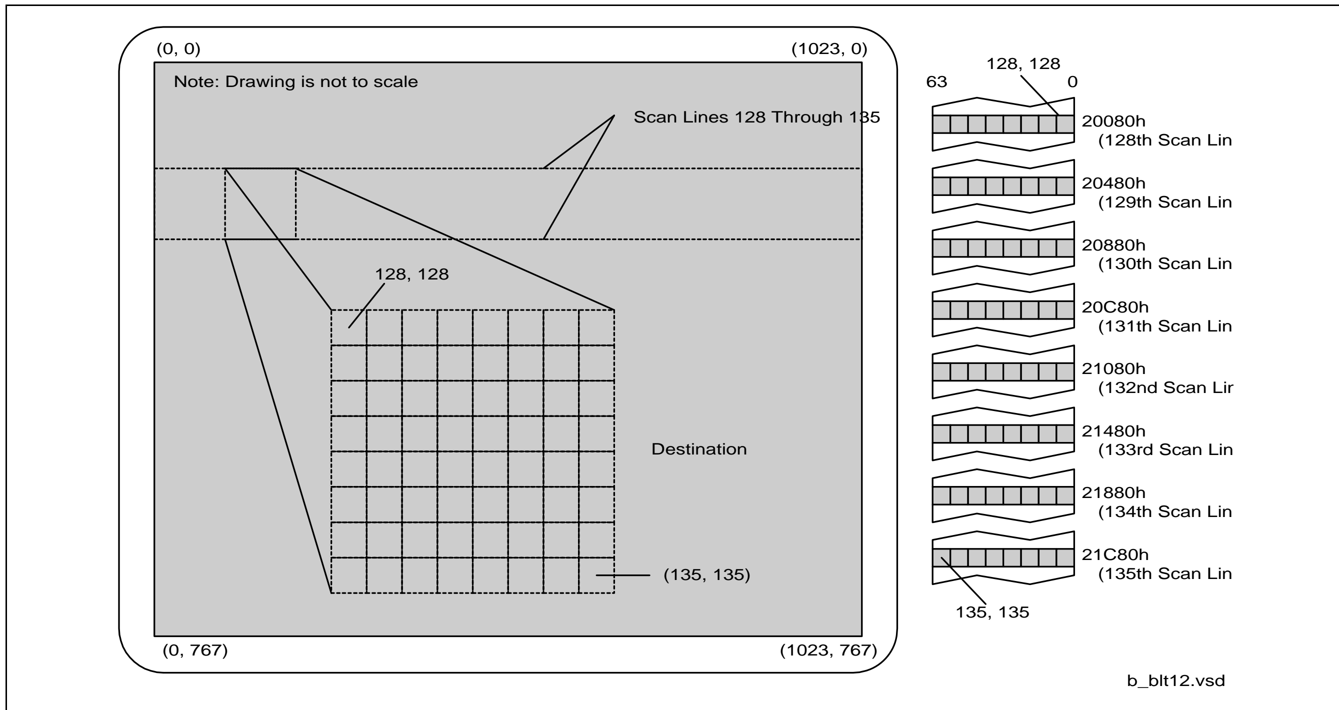


The figure above shows the end result of performing this BLT operation. The 8x8 pattern has been repeatedly copied ("tiled") into the entire 64x64 area at the destination.

8.2.3.2 Drawing Characters Using a Font Stored in System Memory

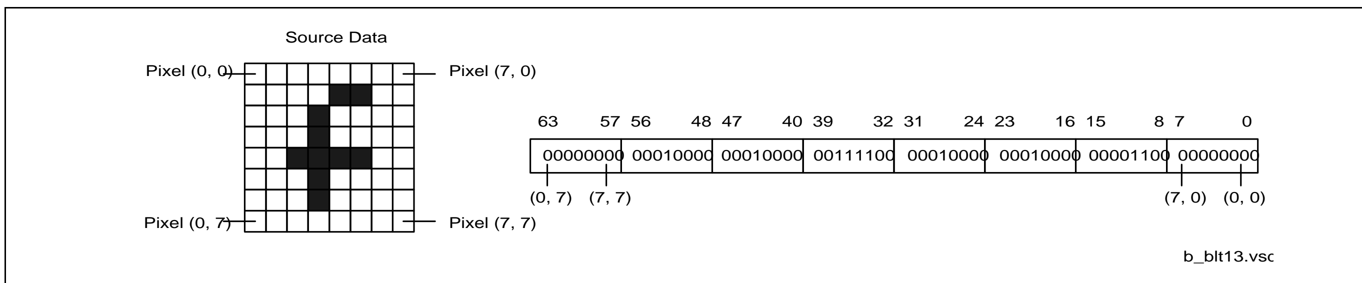
In this example BLT operation, a lowercase letter "f" is to be drawn in black on a display with a gray background. The resolution of the display is 1024x768, and the graphics system has been set to a color depth of 8 bits per pixel.

Figure 8-15. On-Screen Destination for Example Character Drawing BLT

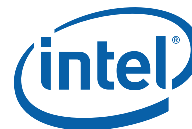


The figure above shows the display on which this letter "f" is to be drawn. As shown in this figure, the entire display has been filled with a gray color. The letter "f" is to be drawn into an 8x8 region on the display with the upper left-hand corner at the coordinates (128, 128).

Figure 8-16. Source Data in System Memory for Example Character Drawing BLT



The figure above shows both the 8x8 pattern making up the letter "f" and how it is represented somewhere in the host's system memory — the actual address in system memory is not important. The letter "f" is represented in system memory by a block of monochrome graphics data that occupies 8 bytes. Each byte carries the 8 bits needed to represent the 8 pixels in each scan line's worth of this graphics data. This type of pattern is often used to store character fonts in system memory.



During this BLT operation, the host CPU will read this representation of the letter "f" from system memory, and write it to the BLT engine by performing memory writes to the ring buffer as an immediate monochrome BLT operand following the BLT_TEXT command. The BLT engine will receive this data through the command stream and use it as the source data for this BLT operation. The BLT engine will be set to the same color depth as the graphics system — 8 bits per pixel, in this case. Since the source data in this BLT operation is monochrome, color expansion must be used to convert it to an 8 bpp color depth. To ensure that the gray background behind this letter "f" is preserved, per-pixel write masking will be performed, using the monochrome source data as the pixel mask.

The BLT Setup and Text_immediate command packets are used to select the features to be used in this BLT operation. Only the fields required by these two command packets must be programmed carefully. The BLT engine ignores all other registers and fields. The source select field in the Text_immediate command must be set to 1, to indicate that the source data is provided by the host CPU through the command packet. Finally, the raster operation field should be programmed with the 8-bit value CCh to select the bit-wise logical operation that simply copies the source data to the destination. Selecting this bit-wise operation in which no pattern data is used as an input, causes the BLT engine to automatically forego reading pattern data from the frame buffer.

The Setup Pattern/Source Expansion Foreground Color Register to specify the color with which the letter "f" will be drawn. There is no Source address. All scan lines of the glyph are bit packed and the clipping is controlled by the ClipRect registers from the SETUP_BLT command and the Destination Y1, Y2, X1, and X2 registers in the TEXT_BLT command. Only the pixels that are within (inclusive comparisons) the clip rectangle are written to the destination surface.

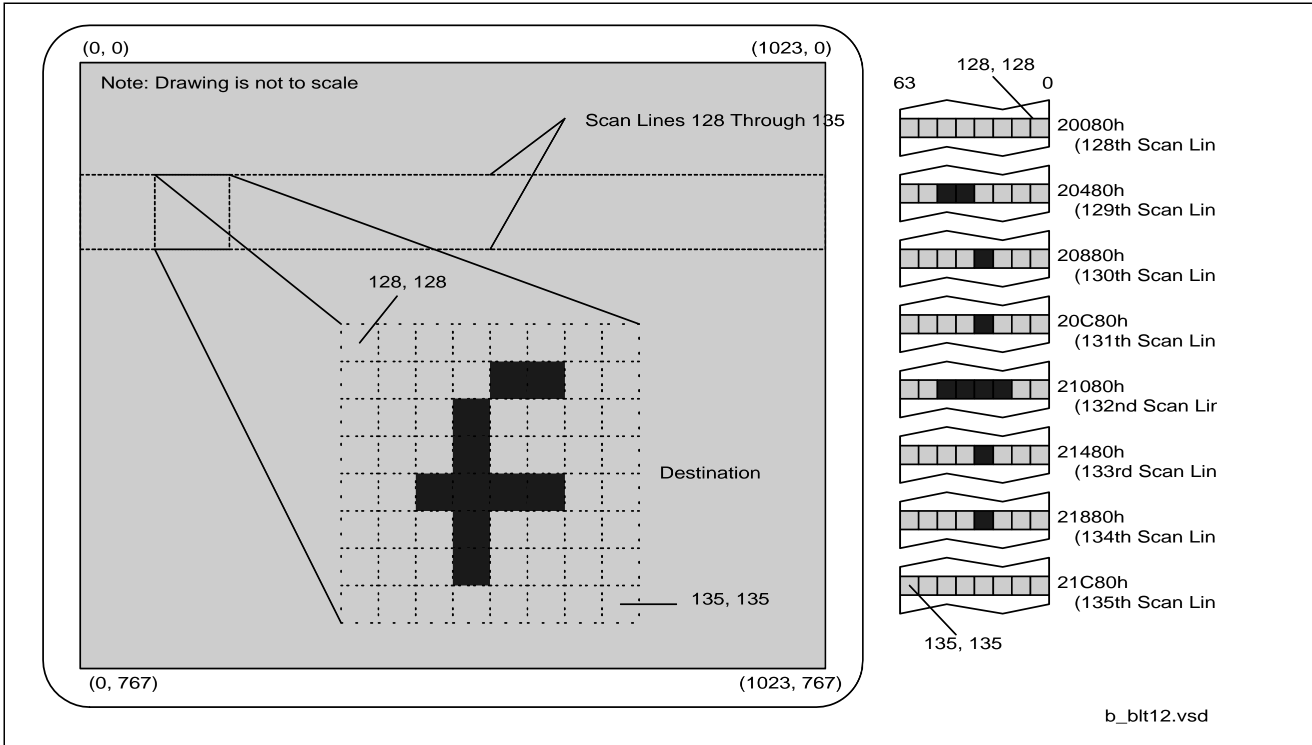
The Destination Pitch Register must be programmed with a value equal to the number of bytes in the interval between the first bytes of each adjacent scan line's worth of destination data. Since the color depth is 8 bits per pixel and the horizontal resolution of the display is 1024 pixels, the value to be programmed into these bits is 400h, which is equal to the decimal value of 1024. Since the source data used in this BLT operation is monochrome, the BLT engine will not use a byte-oriented pitch value for the source data.

Since the source data is monochrome, color expansion is required to convert it to color with a color depth of 8 bits per pixel. Since the Setup Pattern/Source Expansion Foreground Color Register is selected to specify the foreground color of black to be used in drawing the letter "f", this register must be programmed with the value for that color. With the graphics system set for a color depth of 8 bits per pixel, the actual colors are specified in the RAMDAC palette, and the 8 bits stored in the frame buffer for each pixel actually specify the index used to select a color from that palette. This example assumes that the color specified at index 00h in the palette is black, and therefore bits [7:0] of this register should be set to 00h to select black as the foreground color. The BLT engine ignores bits [31:8] of this register because the selected color depth is 8 bits per pixel. Even though the color expansion being performed on the source data normally requires that both the foreground and background colors be specified, the value used to specify the background color is not important in this example. Per-pixel write-masking is being performed with the monochrome source data as the pixel mask, which means that none of the pixels in the source data that will be converted to the background color will ever be written to the destination. Since these pixels will never be seen, the value programmed into the Pattern/Source Expansion Background Color Register to specify a background color is not important.

The Destination Width and Height Registers are not used. The Y1, Y2, X1, and X2 are used to describe to the BLT engine the 8x8 pixel size of the destination location. The Destination Y1 and Y2 address (or coordinate) registers must be programmed with the starting and ending scan line address (or Y coordinates) of the destination data. This address is specified as an offset from the start of the frame buffer of the scan line at the destination that will be written to first. The destination X1 and X2 registers must be programmed with the starting and ending pixel offsets from the beginning of the scan line.

This BLT operation does not use the values in the Pattern Address Register, the Source Expansion Background Color Register, or the Source Expansion Foreground Color Register.

Figure 8-17. Results of Example Character Drawing BLT



The preceding shows the end result of performing this BLT operation. Only the pixels that form part of the actual letter "f" have been drawn into the 8x8 destination location on the display, leaving the other pixels within the destination with their original gray color.

8.3 BLT Instruction Overview

This chapter defines the instructions used to control the 2D (BLT) rendering function.

The instructions detailed in this chapter are used across devices. However, slight changes may be present in some instructions (i.e., for features added or removed), or some instructions may be removed entirely. Refer to the *Device Dependencies* chapter for summary information regarding device-specific behaviors/interfaces/features.

The XY instructions offload the drivers by providing X and Y coordinates and taking care of the access directions for overlapping BLTs without fields specified by the driver.

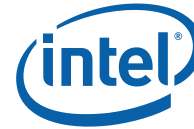
Color pixel sizes supported are 8, 16, and 32 bits per pixel (bpp). All pixels are naturally aligned.

8.4 BLT Engine State

Most of the BLT instructions are state-free, which means that all states required to execute the command is within the instruction. If clipping is not used, then there is no shared state for many of the BLT instructions. This allows the BLT Engine to be shared by many drivers with minimal synchronization between the drivers.

Instructions which share state are:

All instructions that are X,Y commands and use the Clipping Rectangle by asserting the Clip Enable field



All XY_Setup Commands (XY_SETUP_BLT and XY_SETUP_MONO_PATTERN_SL_BLT) load the shared state for the following commands:

XY_PIXEL_BLT	(Negative Stride (=Pitch) Not Allowed)
XY_SCANLINES_BLT	
XY_TEXT_BLT	(Negative Stride (=Pitch) Not Allowed)
XY_TEXT_IMMEDIATE_BLT	(Negative Stride (=Pitch) Not Allowed)

State registers that are saved & restored in the Logical Context:

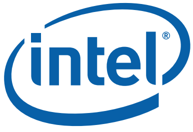
BR1+	Setup Control (Solid Pattern Select, Clipping Enable, Mono Source Transparency Mode, Mono Pattern Transparency Mode, Color Depth[1:0], Raster Operation[7:0], & Destination Pitch[15:0]) + 32bpp Channel Mask[1:0], Mono / Color Pattern
BR05	Setup Background Color
BR06	Setup Foreground Color
BR07	Setup Pattern Base Address
BR09	Setup Destination Base Address
BR20	DW0 for a Monochrome Pattern
BR21	DW1 for a Monochrome Pattern
BR24	ClipRectY1'X1
BR25	ClipRectY2'X2

8.5 Cacheable Memory Support

The BLT Engine can be used to transfer data between cacheable ("system") memory and uncached ("main", or "UC") graphics memory using the BLT instructions. The GTT must be properly programmed to map memory pages as cacheable or UC. Only linear-mapped (not tiled) surfaces can be mapped as cacheable.

Transfers between cacheable sources and cacheable destinations are not supported. Patterns and monochrome sources cannot be located in cacheable memory.

Cacheable write operands do not snoop the processor's cache nor update memory until evicted from the render cache. Cacheable read or write operands are not snooped (nor invalidated) from either internal cache by external (processor, hublink,...) accesses.



8.6 Device Cache Coherency: Render and Texture Caches

Software must initiate cache flushes to enforce coherency between the render and texture caches, i.e., both the render and texture caches must be flushed before a BLT destination surface can be reused as a texture source. Color sources and destinations use the render cache, while patterns and monochrome sources use the texture cache.

8.7 BLT Engine Instructions

The Instruction Target field is used as an opcode by the BLT Engine state machine to qualify the control bits that are relevant for executing the instruction. The descriptions for each DWord and bit field are contained in the *BLT Engine Instruction Field Definition* section. Each DWord field is described as a register, but none of these registers can be written of read through a memory mapped location – they are internal state only.

8.7.1 Blt Programming Restrictions

Overlapping Source/Destination BLTs: The following condition must be avoided when programming the Blt engine: Linear surfaces with a cache line in scan line Y for the source stream overlapping with a cache line in scan line Y-1 for the dest stream (=> non-aligned surface pitches). The cache coherency rules combined with the Blitter data consumption rules result in UNDEFINED operation. (Note that this restriction will likely follow forward to future products due to architectural complexities.) There are two suggested software workarounds:

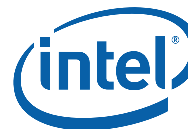
- In order to perform coherent overlapping Blts, (a) the Source and Destination Base Address registers must hold the same value (without alignment restriction), and (b) the Source and Destination Pitch registers (BR11,BR13) must both be a multiple of **64 bytes**.
- If (a) isn't possible, do overlapping source copy BLTs as two blits, using a separate intermediate surface.

All reserved fields must be programmed to 0s.

When using monosource or text data (bit/byte/word aligned): do not program pixel widths greater than 32,745 pixels.

8.8 Fill/Move Instructions

These instructions use linear addresses with width and height. BLT clipping is not supported.



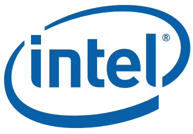
8.8.1 COLOR_BLT (Fill)

COLOR_BLT is the simplest BLT operation. It performs a color fill to the destination (with a possible ROP). The only operand is the destination operand which is written dependent on the raster operation. The solid pattern color is stored in the pattern background register.

This instruction is optimized to run at the maximum memory write bandwidth.

The typical Raster operation code = F0 which performs a copy of the pattern background register to the destination.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode) :** 40h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:05	**Reserved.** Note no tiling specification allowed for this non-XY blit command. Only linear blits are allowed.
	04:00	**DWord Length:** 03h
1 = BR13	31:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color. 01 = 16 bit color (656). 10 = 16 bit color (1555). 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch (signed):** Destination pitch in bytes (Same as before).
2 = BR14	31:16	**Destination Height (in scan lines):**
	15:00	**Destination Width (in bytes):**
3 = BR09	31:00	**Destination Address:** Address of the first byte to be written
4 = BR16	31:00	**Solid Pattern Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]

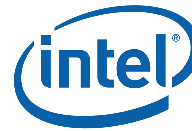


8.8.2 SRC_COPY_BLT (Move)

This BLT instruction performs a color source copy where the only operands involved is a color source and destination of the same bit width.

The source and destination operands may overlap. The command must indicate the horizontal and vertical directions: either forward or backwards to avoid data corruption. The X direction (horizontal) field applies to both the destination and source operands. The source and destination pitches (stride) are signed.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h – 2D Processor
	28:22	**Instruction Target (Opcode) :** 43h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:05	**Reserved.** Note no tiling specification allowed for this non-XY blit command. Only linear blits are allowed.
	04:00	**Dword Length:** 04h
1 = BR13	31	**Reserved.**
	30	**X Direction** (1 = written from right to left (decrementing = backwards); 0 = incrementing)
	29:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch (signed):** Destination pitch in bytes (Same as before).
2 = BR14	31:16	**Destination Height (in scan lines):**
	15:00	**Destination Width (in bytes):**
3 = BR09	31:00	**Destination Address:** Address of the first byte to be written
4 = BR11	31:14	**Reserved.**
	15:00	**Source Pitch:** (double word aligned and signed)
5 = BR12	31:00	**Source Address:** Address of the first byte to be read.

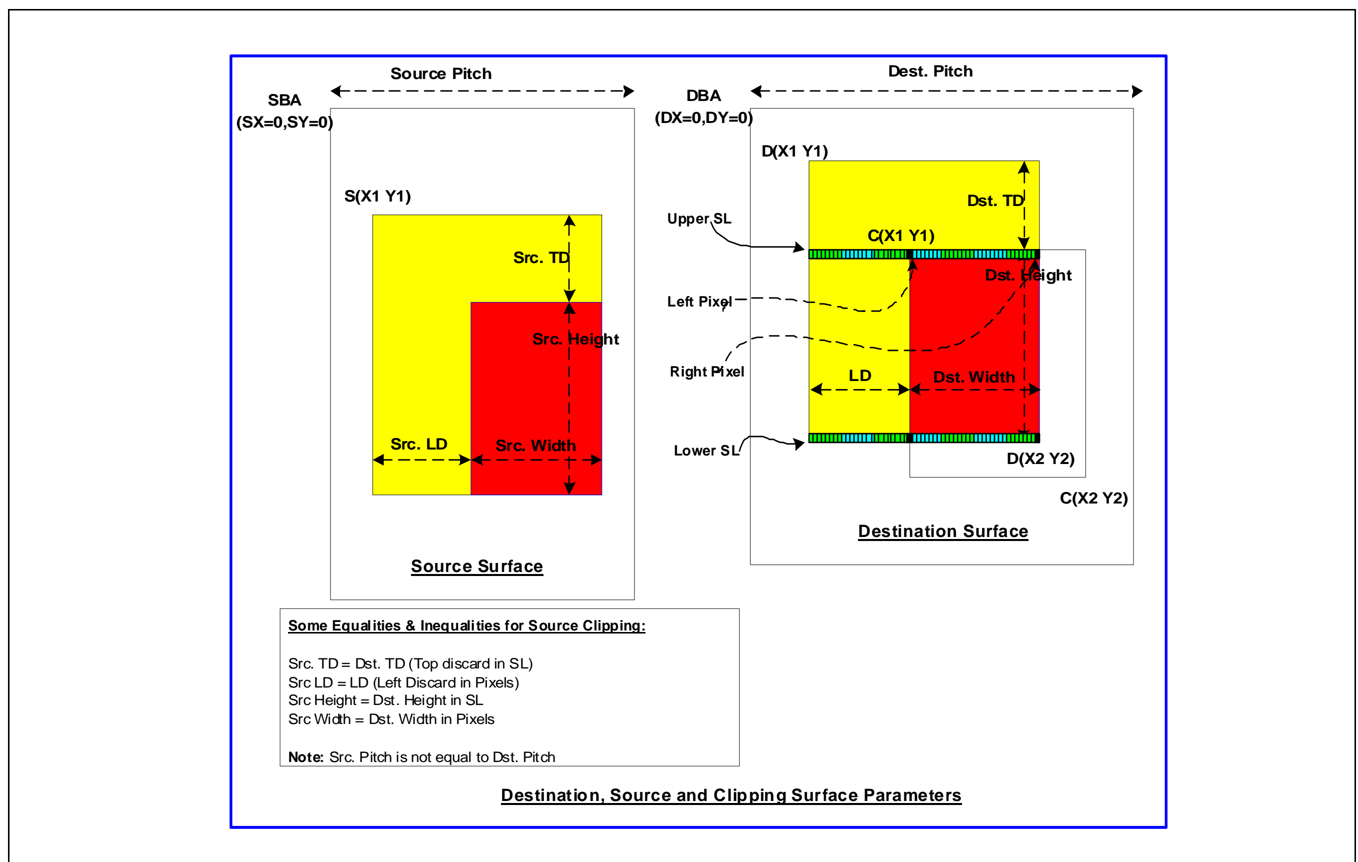


8.9 2D (X,Y) BLT Instructions

Most BLT instructions (prefixed with "XY_") use 2D X,Y coordinate specifications vs. lower-level linear addresses. These instructions also support simple 2D clipping against a clip rectangle.

The top and left Clipping coordinates are inclusive. The bottom and right coordinates are exclusive. The BLT Engine performs a trivial reject for all CLIP BLT instructions before performing any accesses.

Negative destination and source coordinates are supported. In the case of negative source coordinates, the destination X1 and Y1 are modified by the absolute value of the negative source coordinate before the destination clip checking and final drawing coordinates are calculated. The absolute value of the source negative coordinate is added to the corresponding destination coordinate. The BLT engine clipping also checks for (DX2 [or = DX1) or (DY2 [or = DY1) after this calculation and if true, then the BLT is totally rejected.



Destination, Source and Clipping Surface Parameters

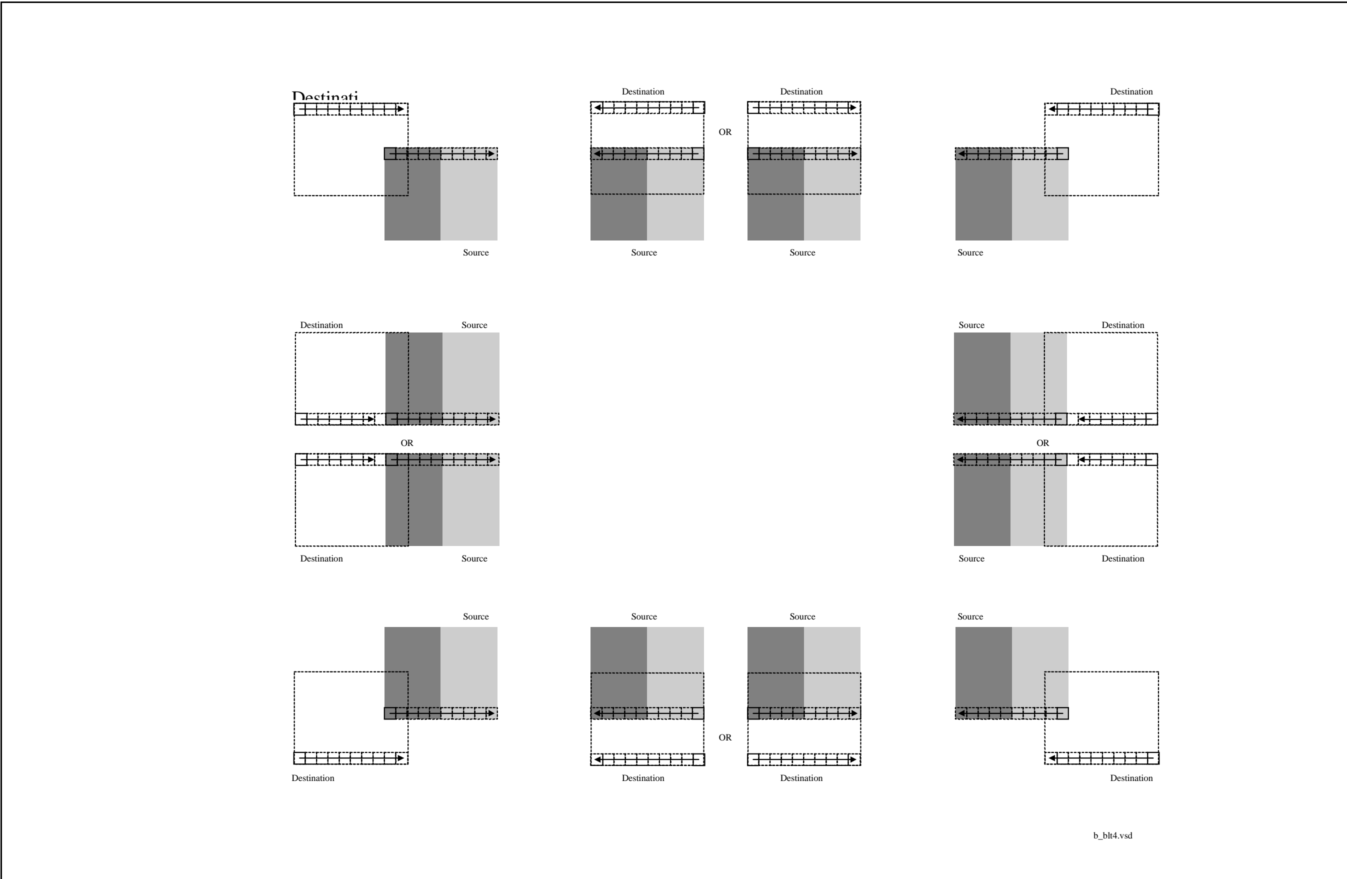
DX1, DY1, CX1, and CY1 are inclusive, while DX2, DY2, CX2, and CY2 are exclusive.

Destination pixel address = (Destination Base Address + (Destination Y coordinate * Destination pitch) + (Destination X coordinate * bytes per pixel)).

Source pixel address = (Source Base Address + (Source Y coordinate * Source pitch) + (Source X coordinate * bytes per pixel)).

Since there is 1 set of Clip Rectangle registers, the Interrupt Ring BLT commands either MUST NEVER enable clipping with these command and never use the XY_Pixel_BLT, XY_Scanline_BLT, nor XY_Text_BLT commands or it must use context switching. The Interrupt rings can also use the non-clipped, linear address commands specified before this section.

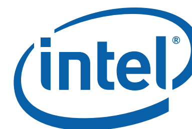
The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses are performed backwards.



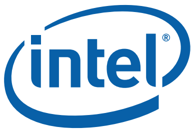
8.9.1 XY_SETUP_BLT

This setup instruction supplies common setup information including clipping coordinates used by the XY commands: XY_PIXEL_BLT, XY_SCANLINE_BLT, XY_TEXT_BLT, and XY_TEXT_BLT_IMMEDIATE.

These are the only instructions that require that state be saved between instructions other than the Clipping parameters. There are 5 dedicated registers to contain the state for these 3 instructions. All other BLTs use a temporary version of these. The 5 double word registers are: DW1 (Setup Control), DW6 (Setup Foreground color), DW5 (Setup Background color), DW7 (Setup Pattern address), and DW4 (Setup Destination Base Address).



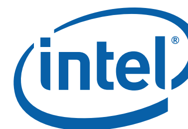
DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 01h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:12	**Reserved.**
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	**Reserved**
	07:00	**Dword Length:** 06h
1 = BR01	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29	**Mono Source Transparency Mode:** (1 = transparency enabled; 0 = use background)
	28:26	**Reserved.**
	25:24	**Color Depth:** All 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement (Negative Pitch Not allowed for Pixel nor Text) For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR24	31:16	**ClipRect Y1 Coordinate (Top):** (30:16 = 15 bit positive number)
	15:00	**ClipRect X1 Coordinate (Left):** (14:00 = 15 bit positive number)
3 = BR25	31:16	**ClipRect Y2 Coordinate (Bottom):** (30:16 = 15 bit positive number)
	15:00	**ClipRect X2 Coordinate (Right):** (14:00 = 15 bit positive number)
4 = BR09	31:00	**Setup Destination Base Address:** (base address of the destination surface: X=0, Y=0) **When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.**
5 = BR05	31:00	**Setup Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0] All
6 = BR06	31:00	**Setup Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0] (SLB & TB only)
7 = BR07	31:00	**Setup Pattern Base Address for Color Pattern:** (26:06 are implemented) (SLB only) (Note no NPO2 change here). The pattern data must be located in linear memory.



8.9.2 XY_SETUP_MONO_PATTERN_SL_BLT

This setup instruction supplies common setup information including clipping coordinates used exclusively with the following instruction: XY_SCANLINE_BLT (SLB) - 1 scan line of monochrome pattern and destination are the only operands allowed.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 11h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:12	**Reserved.**
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	**Reserved**
	07:00	**Dword Length:** 07h
1 = BR01	31	**Solid Pattern Select:** (1 = solid pattern; 0 = no solid pattern) - (SLB & Pixel only)
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29	**Reserved.**
	28	**Mono Pattern Transparency Mode:** (1 = transparency enabled; 0 = use background)
	27:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement (Negative Pitch Not allowed for Pixel nor Text) For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR24	31:16	**ClipRect Y1 Coordinate (Top):** (30:16 = 15 bit positive number)
	15:00	**ClipRect X1 Coordinate (Left):** (14:00 = 15 bit positive number)
3 = BR25	31:16	**ClipRect Y2 Coordinate (Bottom):** (30:16 = 15 bit positive number)
	15:00	**ClipRect X2 Coordinate (Right):** (14:00 = 15 bit positive number)
4 = BR09	31:00	**Setup Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR05	31:00	**Setup Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]

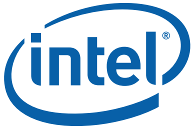


DWord	Bit	Description
6 = BR06	31:00	**Setup Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR20	31:00	**DW0 (least significant) for a Monochrome Pattern:**
8 = BR21	31:00	**DW1 (most significant) for a Monochrome Pattern:**

8.9.3 XY_SETUP_CLIP_ BLT

This command is used to only change the clip coordinate registers. These are the same clipping registers as the Setup clipping registers above.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 03h
	21:12	**Reserved.**
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	**Reserved**
	07:00	**Dword Length:** 01h
1 = BR24	31:16	**ClipRect Y1 Coordinate (Top):** (30:16 = 15 bit positive number)
	15:00	**ClipRect X1 Coordinate (Left):** (14:00 = 15 bit positive number)
2 = BR25	31:16	**ClipRect Y2 Coordinate (Bottom):** (30:16 = 15 bit positive number)
	15:00	**ClipRect X2 Coordinate (Right):** (14:00 = 15 bit positive number)



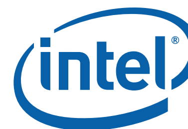
8.9.4 XY_PIXEL_BLT

The Destination X coordinate and Destination Y coordinate is compared with the ClipRect registers. If it is within all 4 comparisons, then the pixel supplied in the XY_SETUP_BLT instruction is written with the raster operation to (Destination Y Address + (Destination Y coordinate * Destination pitch) + (Destination X coordinate * bytes per pixel)).

ROP field must specify pattern or fill with 0's or 1's. There is no source operand.

Negative Stride (= Pitch) specified in the Setup command is Not Allowed

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 24h
	21:12	**Reserved.**
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Reserved**
	07:00	**Dword Length :** 00h
1 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)



8.9.5 XY_SCANLINES_BLT

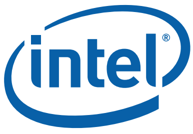
All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

Solid pattern should use the XY_SETUP_MONO_PATTERN_SL_BLT instruction.

ROP field must specify pattern or fill with 0's or 1's. There is no source operand.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 25h
	21:15	**Reserved.**
	14:12	**Pattern Horizontal Seed:** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (scan line of the 8x8 pattern to start on corresponding to DST Y=0)
	07:00	**Dword Length:** 01h
1 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
2 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)



8.9.6 XY_TEXT_BLT

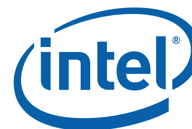
All source scan lines and pixels that fall within the ClipRect Y and X coordinates are written. The source address corresponds to Destination X1 and Y1 coordinate.

Text is either bit or byte packed. Bit packed means that the next scan line starts 1 pixel after the end of the current scan line with no bit padding. Byte packed means that the next scan line starts on the first bit of the next byte boundary after the last bit of the current line.

Source expansion color registers are always in the SETUP_BLT.

Negative Stride (= Pitch) is NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 26h
	21:17	**Reserved.**
	16	**Bit (0) / Byte (1) packed:** Byte packed is for the NT driver
	15:12	**Reserved.**
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	**Reserved**
	07:00	**Dword Length:** 02h
1 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
2 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
3 = BR12	31:00	**Source Address:** (address of the first byte on scan line corresponding to Dst X1,Y1) (Note no NPO2 change here)



8.9.7 XY_TEXT_IMMEDIATE_BLT

This instruction allows the Driver to send data through the instruction stream that eliminates the read latency of reading a source from memory. If an operand is in system cacheable memory and either small or only accessed once, it can be copied directly to the instruction stream versus to graphics accessible memory.

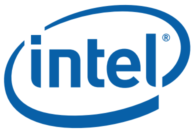
The IMMEDIATE_BLT data MUST transfer an even number of doublewords. The BLT engine will hang if it does not get an even number of doublewords.

All source scan lines and pixels that fall within the ClipRect X and Y coordinates are written. The source data corresponds to Destination X1 and Y1 coordinate.

Source expansion color registers are always in the SETUP_BLT.

NEGATIVE STRIDE (= PITCH) IS NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h – 2D Processor
	28:22	**Instruction Target (Opcode):** 31h
	21:17	**Reserved.**
	16	**Bit (0) / Byte (1) packed:** Byte packed is for the NT driver
	15:12	**Reserved.**
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	**Reserved**
	07:00	**Dword Length :** 01+ DWL = (Number of Immediate double words)h
1 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
2 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
3	31:00	**Immediate Data DW 0:**
4	31:00	**Immediate Data DW 1:**
5 thru DWL+3	S	**Immediate Data DWs 2 through DWORD_LENGTH (DWL):**



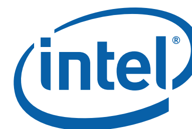
8.9.8 XY_COLOR_BLT

COLOR_BLT is the simplest BLT operation. It performs a color fill to the destination (with a possible ROP). The only operand is the destination operand which is written dependent on the raster operation. The solid pattern color is stored in the pattern background register.

This instruction is optimized to run at the maximum memory write bandwidth.

The typical (and fastest) Raster operation code = F0 which performs a copy of the pattern background register to the destination.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 50h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:12	**Reserved.**
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Reserved**
	07:00	**Dword Length:** 04h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)



DWord	Bit	Description
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR16	31:00	**Solid Pattern Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]

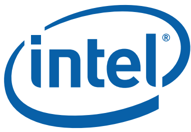
8.9.9 XY_PAT_BLT

PAT_BLT is used when there is no source and the color pattern is not trivial (is not a solid color only).

If clipping is enabled, all scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 51h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:15	**Reserved.**
	14:12	**Pattern Horizontal Seed** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed: (**Starting Scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Dword Length:** 04h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**



DWord	Bit	Description
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR15	31:00	**Pattern Base Address:** (28:06 are implemented) (Note no NPO2 change here). The pattern data must be located in linear memory.

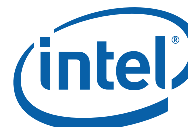
8.9.10 XY_PAT_CHROMA_BLT

PAT_BLT is used when there is no source and the color pattern is not trivial (is not a solid color only).

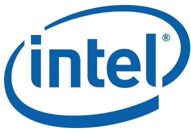
All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 76h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	**Transparency Range Mode:** (chroma-key) – Dst Chroma-key modes ONLY (SRC ILLEGAL)
	16:15	**Reserved.**
	14:12	**Pattern Horizontal Seed** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed: (**Starting Scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Dword Length:** 06h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29:26	**Reserved.**



DWord	Bit	Description
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR15	31:00	**Pattern Base Address:** (26:06 are used, other bits are ignored) (Note no NPO2 change here). The pattern data must be located in linear memory.
6 = BR18	31:00	**Transparency Color Low:** (Chroma-key Low = Pixel Greater or Equal)
7 = BR19	31:00	**Transparency Color High:** (Chroma-key High = Pixel Less or Equal)



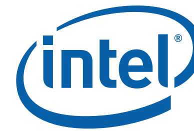
8.9.11 XY_PAT_BLT_IMMEDIATE

PAT_BLT_IMMEDIATE is used when there is no source and the color pattern is not trivial (is not a solid color only) and the pattern is pulled through the command stream. The immediate data sizes are 64 bytes (16 DWs), 128 bytes (32 DWs), or 256 (64DWs) for 8, 16, and 32 bpp color patterns.

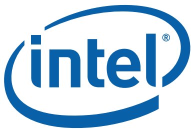
All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 72h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:15	**Reserved.**
	14:12	**Pattern Horizontal Seed** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Dword Length:** 03+ DWL = (Number of Immediate double)h
1 = BR13	31	**Reserved**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)



DWord	Bit	Description
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5	31:00	**Immediate Data DW 0:**
6	31:00	**Immediate Data DW 1:**
7 thru DWL+3	S	**Immediate Data DWs 2 through DWORD_LENGTH (DWL):**



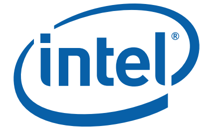
8.9.12 XY_PAT_CHROMA_BLT_IMMEDIATE

PAT_BLT_IMMEDIATE is used when there is no source and the color pattern is not trivial (is not a solid color only) and the pattern is pulled through the command stream. The immediate data sizes are 64 bytes (16 DWs), 128 bytes (32 DWs), or 256 (64DWs) for 8, 16, and 32 bpp color patterns.

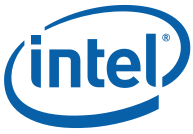
All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 77h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	**Transparency Range Mode:** (chroma-key) – Dst Chroma-key modes ONLY (SRC ILLEGAL)
	16:15	**Reserved.**
	14:12	**Pattern Horizontal Seed** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Dword Length:** 05+ DWL = (Number of Immediate double)h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable** (1 = enabled; 0 = disabled)
	29:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)



DWord	Bit	Description
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR18	31:00	**Transparency Color Low:** (Chroma-key Low = Pixel Greater or Equal)
6 = BR19	31:00	**Transparency Color High:** (Chroma-key High = Pixel Less or Equal)
7	31:00	**Immediate Data DW 0:**
8	31:00	**Immediate Data DW 1:**
9 thru DWL+3	S	**Immediate Data DWs 2 through DWORD_LENGTH (DWL):**



8.9.13 XY_MONO_PAT_BLT

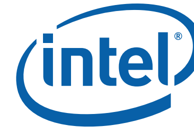
MONO_PAT_BLT is used when we have no source and the monochrome pattern is not trivial (is not a solid color only). The monochrome pattern is loaded from the instruction stream.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

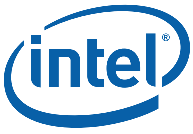
The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

The monochrome pattern transparency mode indicates whether to use the pattern background color or de-assert the write enables when the bit in the pattern is 0. When the pattern bit is 1, then the pattern foreground color is used in the ROP operation.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode) :** 52h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:15	**Reserved.**
	14:12	**Pattern Horizontal Seed:** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Dword Length:** 07h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable** (1 = enabled; 0 = disabled)
	29	**Reserved.**
	28	**Mono Pattern Transparency Mode:** (1 = transparency enabled; 0 = use background)
	27:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).



DWord	Bit	Description
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** 31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR16	31:00	**Pattern Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
6 = BR17	31:00	**Pattern Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR20	31:00	**Pattern Data 0:** (least significant DW)
8 = BR21	31:00	**Pattern Data 1:** (most significant DW)



8.9.14 XY_MONO_PAT_FIXED_BLT

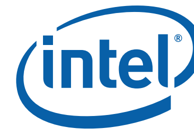
MONO_PAT_FIXED_BLT is used when we have no source and the monochrome pattern is not trivial (is not a solid color only). The monochrome pattern is one of 10 fixed patterns described below. The pattern seeds can still be used with the fixed patterns, creating even more fixed patterns. This eliminates 2 doublewords compared to the XY_MONO_PAT_BLT command packet.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

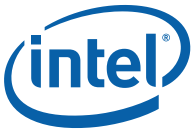
The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

The monochrome pattern transparency mode indicates whether to use the pattern background color or de-assert the write enables when the bit in the pattern is 0. When the pattern bit is 1, then the pattern foreground color is used in the ROP operation.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target** (Opcode)**:** 59h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19	**Reserved.**
	18:15	**Fixed Pattern:** 0000 HS_HORIZONTAL 0001 HS_VERTICAL 0010 HS_FDIAGONAL 0011 HS_BDIAGONAL 0100 HS_CROSS 0101 HS_DIAGCROSS 0110 Reserved 0111 Reserved 1000 Screen Door 1001 SD Wide 1010 Walking Bit (one) 1011 Walking Zero 1100 Reserved 1101 Reserved 1110 Reserved
	14:12	**Pattern Horizontal Seed:** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Dword Length:** 05h

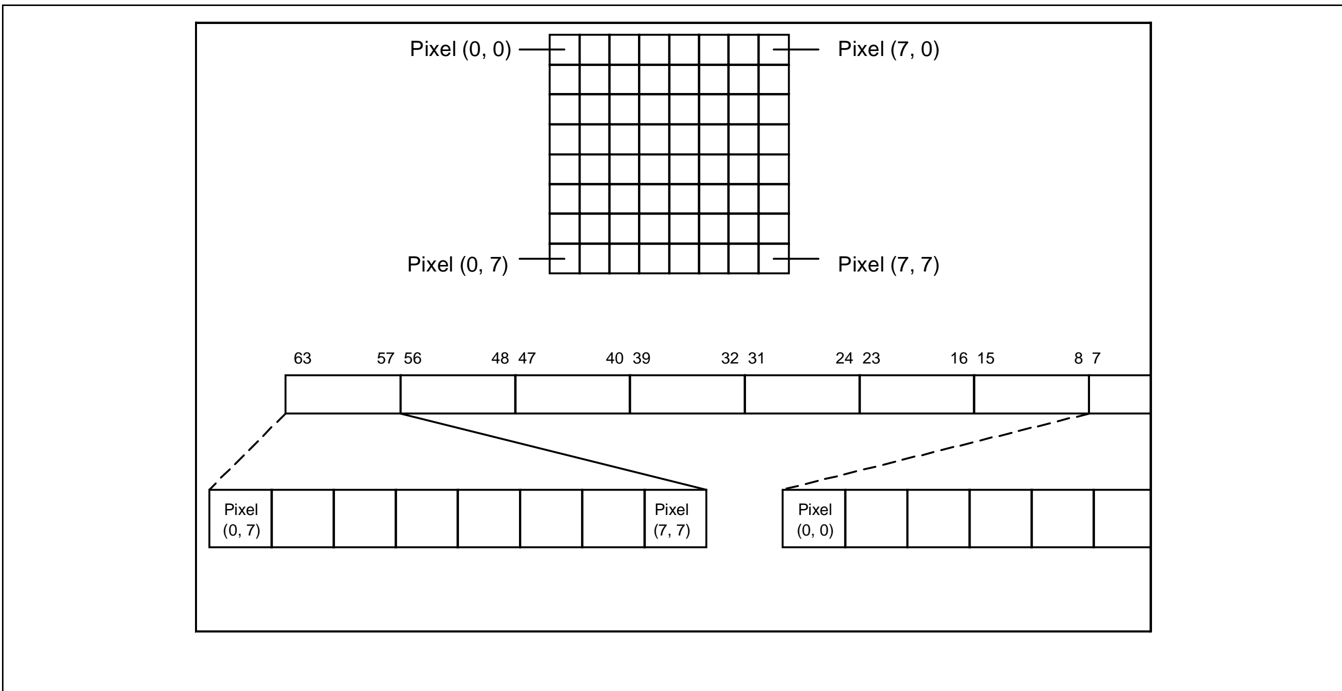


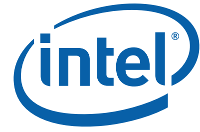
DWord	Bit	Description
1 = BR13	31	**Reserved.**
	30	**Clipping Enable** (1 = enabled; 0 = disabled)
	29	**Reserved.**
	28	**Mono Pattern Transparency Mode:** (1 = transparency enabled; 0 = use background)
	27	**Bit Mask Enable:** (1 = use bit mask register for bit writes; 0 = disabled)
	27:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR16	31:00	**Pattern Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
6 = BR17	31:00	**Pattern Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]



8.9.14.1 Monochrome Pattern Memory Format

The monochrome pattern is made of 8 bytes that correspond to the 8 pixels per scan line and 8 scan lines. Byte 0 corresponds to scan line 0, byte 1 corresponds to scan line 1,..., and byte 7 corresponds to scan line 7. The bits within each byte are transposed. Pixel 0 is bit 7, pixel 1 is bit 6,..., pixel 7 is bit 0. The diagram below illustrates the byte and bit relationship to the pixels of the pattern.





8.9.14.2 HS_HORIZONTAL 0

Bit 7 0
0,0 7,0

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

8.9.14.3 HS_VERTICAL 1

Bit 7 0
0,0 7,0

0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0

8.9.14.4 HS_FDIAGONAL 2

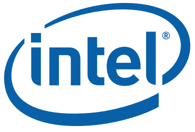
Bit 7 0
0,0 7,0

1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1

8.9.14.5 HS_BDIAGONAL 3

Bit 7 0
0,0 7,0

0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0



8.9.14.6 HS_CROSS 4

Bit 7 0
0,0 7,0

0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0

8.9.14.7 HS_DIAGCROSS 5

Bit 7 0
0,0 7,0

1	0	0	0	0	0	0	1
0	1	0	0	0	0	1	0
0	0	1	0	0	1	0	0
0	0	0	1	1	0	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	1	0	0
0	1	0	0	0	0	1	0
1	0	0	0	0	0	0	1

8.9.14.8 Screen Door 8

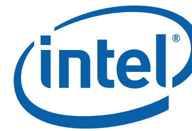
Bit 7 0
0,0 7,0

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

8.9.14.9 SD Wide 9

Bit 7 0
0,0 7,0

1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1



8.9.14.10 Walking Bit (One) A

Bit 7 0
0,0 7,0

1	0	0	0	1	0	0	0
0	1	0	0	0	1	0	0
0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	1
1	0	0	0	1	0	0	0
0	1	0	0	0	1	0	0
0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	1

8.9.14.11 Walking Zero B

Bit 7 0
0,0 7,0

0	1	1	1	0	1	1	1
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0
0	1	1	1	0	1	1	1
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0

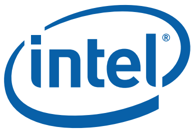
8.9.15 XY_SRC_COPY_BLT

This BLT instruction performs a color source copy where the only operands involved is a color source and destination of the same bit width.

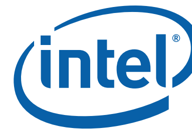
The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

The ROP value chosen must involve source and no pattern data in the ROP operation.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 53h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:16	**Reserved.**
	15	**Src Tiling Enable:** 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)



DWord	Bit	Description
	14:12	**Reserved**
	11	**Dest Tiling Enable:** **0 = Tiling Disabled (Linear)** **1 = Tiling enabled (Tile-X only)**
	10: 8	**Reserved**
	7:0	**Dword Length:** 06h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement **For Tiled Dest (bit 11 enabled) this ptich is of 512Byte granularity and can be up to 128Kbytes (or 32KDwords).**
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
5 = BR26	31:16	**Source Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Source X1 Coordinate (Left):** (15:00 = 16 bit signed number)
6 = BR11	31:16	**Reserved**
	15:00	**Source Pitch (double word aligned) and in DWords:** [15:00] 2's complement. For Tiled Src (bit 15 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
7 = BR12	31:00	**Source Base Address:** (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.



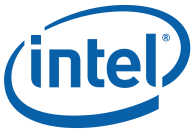
8.9.16 XY_SRC_COPY_CHROMA_BLT

This BLT instruction performs a color source copy with chroma-keying where the only operands involved is a color source and destination of the same bit width.

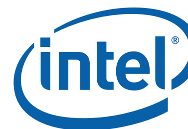
The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

The ROP value chosen must involve source and no pattern data in the ROP operation.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 73h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	**Transparency Range Mode:** (chroma-key)
	16	**Reserved**
	15	**Src Tiling Enable:** 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	14:12	**Reserved**
	11	**Dest Tiling Enable:** 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	10:08	**Reserved**
	07:00	**Dword Length:** 08h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**



DWord	Bit	Description
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled Dest (bit 11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
5 = BR26	31:16	**Source Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Source X1 Coordinate (Left):** (15:00 = 16 bit signed number)
6 = BR11	31:16	**Reserved.**
	15:00	**Source Pitch (double word aligned) and in DWords:** [15:00] 2's complement. For Tiled Src (bit 15 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
7 = BR12	31:00	**Source Base Address:** (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.
8 = BR18	31:00	**Transparency Color Low:** (Chroma-key Low = Pixel Greater or Equal)
9 = BR19	31:00	**Transparency Color High:** (Chroma-key High = Pixel Less or Equal)



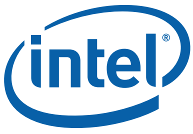
8.9.17 XY_MONO_SRC_COPY_BLT

This BLT instruction performs a monochrome source copy where the only operands involved is a monochrome source and destination. The source and destination operands cannot overlap therefore the X and Y directions are always forward.

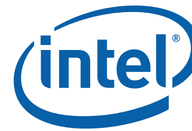
All non-text monochrome sources are word aligned. At the end of a scan line of monochrome source, all bits until the next word boundary are ignored. The monochrome source data bit position field [2:0] indicates the bit position within the first byte of the scan line that should be used as the first source pixel which corresponds to the destination X1 coordinate.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation. The ROP value chosen must involve source and no pattern data in the ROP operation.
Negative Stride (= Pitch) is NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 54h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	**Monochrome source data bit position of the first pixel within a byte per scan line.**
	16:12	**Reserved.**
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Reserved**
	07:00	**Doubleword Length:** 06h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29	**Mono Source Transparency Mode:** (1 = transparency enabled; 0 = use background)
	28:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**



DWord	Bit	Description
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR12	31:00	**Source Address:** (address corresponding to DST X1,Y1) (Note no NPO2 change here)
6 = BR18	31:00	**Source Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR19	31:00	**Source Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]



8.9.18 XY_MONO_SRC_COPY_ IMMEDIATE_BLT

This instruction allows the Driver to send monochrome data through the instruction stream, eliminating the read latency of the source during command execution.

The IMMEDIATE_BLT data MUST transfer an even number of doublewords and the exact number of quadwords.

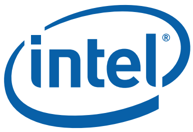
All non-text monochrome sources are word aligned. At the end of a scan line of monochrome source, all bits until the next word boundary are ignored. The Monochrome source data bit position field [2:0] indicates the bit position within the first byte of the scan line that should be used as the first source pixel which corresponds to the destination X1 coordinate.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation. The ROP value chosen must involve source and no pattern data in the ROP operation.

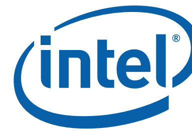
The monochrome source data supplied corresponds to the Destination X1 and Y1 coordinates.

Negative Stride (= Pitch) is NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 71h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	**Monochrome source data bit position of the first pixel within a byte per scan line.**
	16:12	**Reserved.**
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Reserved**
	07:00	**Dword Length:** 05+ DWL = (Number of Immediate double words)h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29	**Mono Source Transparency Mode:** (1 = transparency enabled; 0 = use background)
	28:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color



DWord	Bit	Description
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR18	31:00	**Source Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
6 = BR19	31:00	**Source Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7	31:00	**Immediate Data DW 0:**
8	31:00	**Immediate Data DW 1:**
9 thru DWL+4	S	**Immediate Data DWs 2 through DWORD_LENGTH (DWL):**



8.9.19 XY_FULL_BLT

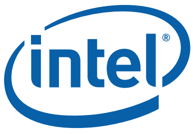
The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The source and pattern operands are the same bit width as the destination operand.

The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

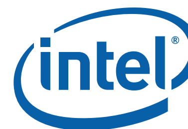
All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	**Client:** 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 55h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:16	**Reserved.**
	15	**Src Tiling Enable:** 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	14:12	**Pattern Horizontal Seed** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Dest Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Doubleword Length:** 07h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29:26	**Reserved.**



DWord	Bit	Description
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled Dest (bit 11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
5 = BR11	31:16	**Reserved.**
	15:00	**Source Pitch (double word aligned and signed) and in DWords:** [15:00] 2's complement. For Tiled Src (bit 15 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
6 = BR26	31:16	**Source Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Source X1 Coordinate (Left):** (15:00 = 16 bit signed number)
7 = BR12	31:00	**Source Base Address:** (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.
8 = BR15	31:00	**Pattern Base Address:** (28:06 are implemented) (Note no NPO2 change here). The pattern data must be located in linear memory.



8.9.20 XY_FULL_IMMEDIATE_PATTERN_BLT

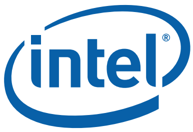
The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The source and immediate pattern operands are the same bit width as the destination operand. The immediate data sizes are 64 bytes (16 DWs), 128 bytes (32 DWs), or 256 (64 DWs) for 8, 16, and 32 bpp color patterns.

The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

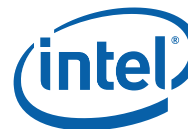
All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	**Client**: 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 74h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:16	**Reserved.**
	15	**Src Tiling Enable:** 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	14:12	**Pattern Horizontal Seed:** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Dest Tiling Enable:** 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	10:8	**Pattern Vertical Seed:** (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	7:0	**Doubleword Length:** 06+ DWL = (Number of Immediate double words)h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29:26	**Reserved.**



DWord	Bit	Description
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled Dest (bit 11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
	31:16	**Reserved.**
5 = BR11	15:00	**Source Pitch (double word aligned and signed) and in DWords:** [15:00] 2's complement. For Tiled Src (bit 15 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
6 = BR26	31:16	**Source Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Source X1 Coordinate (Left):** (15:00 = 16 bit signed number)
7 = BR12	31:00	**Source Base Address:** (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.
8	31:00	**Immediate Data DW 0:**
9	31:00	**Immediate Data DW 1:**
A thru DWL+4	S	**Immediate Data DWs 2 through DWORD_LENGTH (DWL):**



8.9.21 XY_FULL_MONO_SRC_BLT

The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The source operand is monochrome and the pattern operand is the same bit width as the destination.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation.

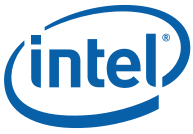
All non-text and non-immediate monochrome sources are word aligned. At the end of a scan line the monochrome source, the remaining bits until the next word boundary are ignored. The Monochrome source data bit position field [2:0] indicates which bit position within the first byte should be used as the first source pixel which corresponds to the Destination X1 coordinate.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

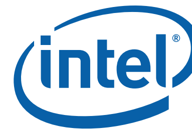
The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

Negative Stride (= Pitch) is NOT ALLOWED

DWord	Bit	Description
0 = BR00	31:29	**Client**: 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 56h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	**Monochrome source data bit position of the first pixel within a byte per scan line.**
	16:15	**Reserved.**
	14:12	**Pattern Horizontal Seed:** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (starting address of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Doubleword Length :** 07h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29	**Mono Source Transparency Mode:** (1 = transparency enabled; 0 = use background)



DWord	Bit	Description
	28:27	**Reserved.**
	26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR12	31:00	**Mono Source Address:** (address corresponds to DST X1, Y1) (Note no NPO2 change here)
6 = BR18	31:00	**Source Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR19	31:00	**Source Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
8 = BR15	31:00	**Pattern Base Address:** (28:06 are implemented **)** (Note no NPO2 change here). The pattern data must be located in linear memory.



8.9.22 XY_FULL_MONO_SRC_IMMEDIATE_PATTERN_BLT

The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The source operand is a monochrome and the immediate pattern operand is the same bit width as the destination. The immediate data sizes are 64 bytes (16 DWs), 128 bytes (32 DWs), or 256 (64DWs) for 8, 16, and 32 bpp color patterns.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation.

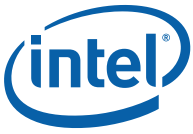
All non-text monochrome sources are word aligned. At the end of a scan line the monochrome source, the remaining bits until the next word boundary are ignored. The Monochrome source data bit position field [2:0] indicates which bit position within the first byte should be used as the first source pixel which corresponds to the destination X1 coordinate.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

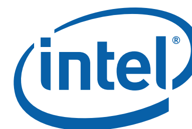
The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

Negative Stride (= Pitch) is NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	**Client**: 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 75h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	**Monochrome source data bit position of the first pixel within a byte per scan line.**
	16:15	**Reserved.**
	14:12	**Pattern Horizontal Seed:** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (starting address of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Doubleword Length :** 06+ DWL = (Number of Immediate double words)h
1 = BR13	31	**Reserved.**
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)



DWord	Bit	Description
	29	**Mono Source Transparency Mode:** (1 = transparency enabled; 0 = use background)
	28:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR12	31:00	**Mono Source Address:** (address corresponds to DST X1, Y1) (Note no NPO2 change here)
6 = BR18	31:00	**Source Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR19	31:00	**Source Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
8	31:00	**Immediate Data DW 0:**
9	31:00	**Immediate Data DW 1:**
A thru DWL+4	S	**Immediate Data DWs 2 through DWORD_LENGTH (DWL):**



8.9.23 XY_FULL_MONO_PATTERN_BLT

The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The pattern operand is monochrome and the source operand is the same bit width as the destination operand.

The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

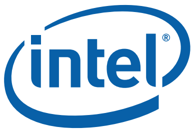
The monochrome pattern transparency mode indicates whether to use the pattern background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the pattern foreground color is used in the ROP operation.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

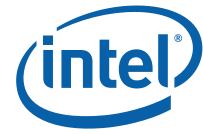
The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

Setting both Solid Pattern Select =1 & Mono Pattern Transparency = 1 is mutually exclusive. The device implementation results in NO PIXELs DRAWN.

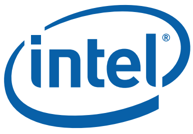
DWord	Bit	Description
0 = BR00	31:29	**Client**: 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 57h
	21:20	**32 bpp byte mask:** (21 =1= write alpha channel; 20=1= write RGB channels)
	19:16	**Reserved.**
	15	**Src Tiling Enable:** 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	14:12	**Pattern Horizontal Seed:** (pixel of the scan line to start on corresponding to DST X=0)
	11	**Dest Tiling Enable:** 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	**Dword Length :** 0Ah
1 = BR13	31	**Solid Pattern Select:** (1 = solid pattern; 0 = no solid pattern)



DWord	Bit	Description
	30	**Clipping Enable:** (1 = enabled; 0 = disabled)
	29	**Reserved.**
	28:27	**Mono Pattern Transparency Mode:** (1 = transparency enabled; 0 = use background)
	26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color 10 = 16 bit color (1555) 11 = 32 bit color (565)
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled Dest (bit 11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
5 = BR11	31:16	**Reserved.**
	15:00	**Source Pitch (double word aligned and signed) and in DWords:** [15:00] 2's complement. For Tiled Src (bit 15 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
6 = BR26	31:16	**Source Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Source X1 Coordinate (Left):** (15:00 = 16 bit signed number)
7 = BR12	31:00	**Source Base Address:** (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.
8 = BR16	31:00	**Pattern Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
9 = BR17	31:00	**Pattern Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
A = BR20	31:00	**Pattern Data 0:** (least significant DW)



DWord	Bit	Description
B = BR21	31:00	**Pattern Data 1:** (most significant DW)



8.9.24 XY_FULL_MONO_PATTERN_MONO_SRC_BLT

The full BLT provides the ability to specify all 3 operands: destination, source, and pattern. The pattern and source operands are monochrome.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation.

All non-text monochrome sources are word aligned. At the end of a scan line the monochrome source, the remaining bits until the next word boundary are ignored. The Monochrome source data bit position field [2:0] indicates which bit position within the first byte should be used as the first source pixel which corresponds to the destination X1 coordinate.

The monochrome pattern transparency mode indicates whether to use the pattern background color or de-assert the write enables when the bit in the pattern is 0. When the source bit is 1, then the pattern foreground color is used in the ROP operation. The monochrome source transparency mode works identical to the pattern transparency mode.

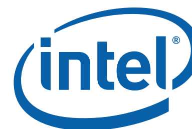
All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

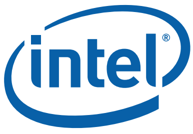
Setting both Solid Pattern Select =1 & Mono Pattern Transparency = 1 is mutually exclusive. The device implementation results in NO PIXELs DRAWN.

Negative Stride (= Pitch) is NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	**Client**: 02h - 2D Processor
	28:22	**Instruction Target (Opcode):** 58h
	21:20	**32 bpp byte mask:** (21 = 1 = write alpha channel; 20 = 1 = write RGB channels)
	19:17	**Monochrome source data bit position of the first pixel within a byte per scan line.**
	16:15	**Reserved.**
	14:12	**Pattern Horizontal Seed:** (pixel of the scan line to start on corresponding to DST X = 0)
	11	**Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	**Pattern Vertical Seed:** (starting scan line of the 8x8 pattern corresponding to DST Y = 0)
	07:00	**Doubleword Length :** 0Ah



DWord	Bit	Description
1 = BR13	31	**Solid Pattern Select:** (1 = solid pattern; 0 = no solid pattern)
	30	**Clipping Enable** (1 = enabled; 0 = disabled)
	29	**Mono Source Transparency Mode:** (1 = transparency enabled; 0 = use background)
	28	**Mono Pattern Transparency Mode:** (1 = transparency enabled; 0 = use background)
	27:26	**Reserved.**
	25:24	**Color Depth:** 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	**Raster Operation:**
	15:00	**Destination Pitch in DWords:** [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	**Destination Y1 Coordinate (Top):** (31:16 = 16 bit signed number)
	15:00	**Destination X1 Coordinate (Left):** (15:00 = 16 bit signed number)
3 = BR23	31:16	**Destination Y2 Coordinate (Bottom):** (31:16 = 16 bit signed number)
	15:00	**Destination X2 Coordinate (Right):** (15:00 = 16 bit signed number)
4 = BR09	31:00	**Destination Base Address:** (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR12	31:00	**Source Address:** (address corresponding to Dst X1,Y1) (Note no NPO2 change here)
6 = BR18	31:00	**Source Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR19	31:00	**Source Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
8 = BR16	31:00	**Pattern Background Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
9 = BR17	31:00	**Pattern Foreground Color:** 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
A =BR20	31:00	**Pattern Data 0:** (least significant DW)
B =BR21	31:00	**Pattern Data 1:** (most significant DW)



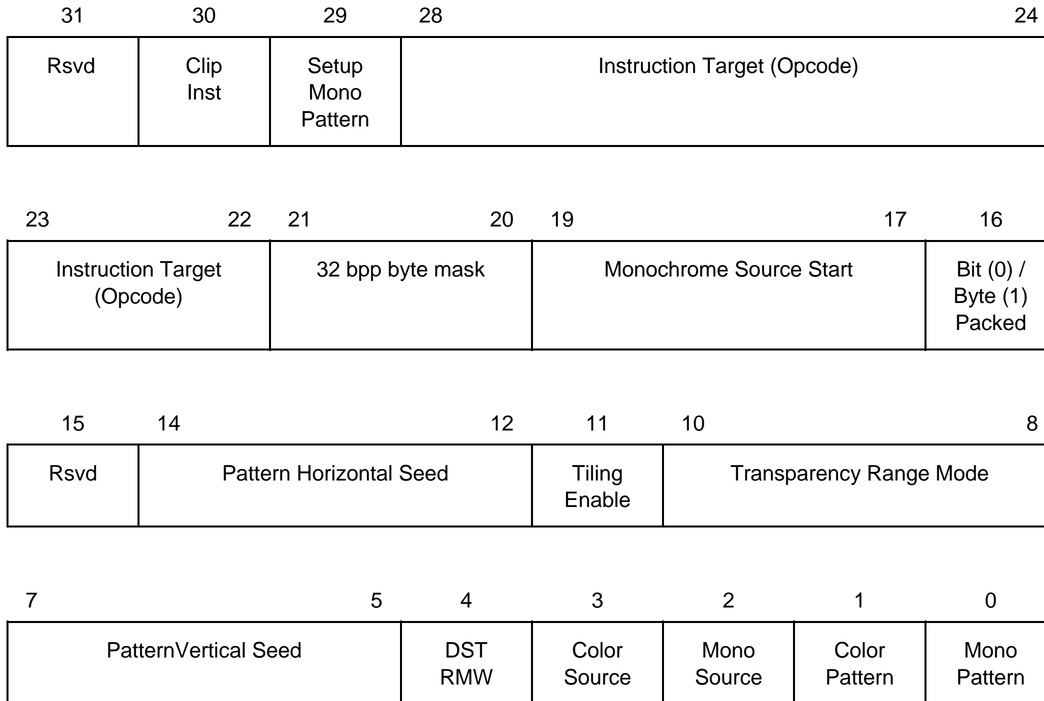
8.10 BLT Engine Instruction Field Definitions

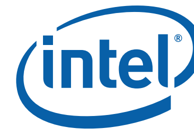
This section describes the BLT Engine instruction fields. These descriptions are in the format of register descriptions. These registers are internal and are not readable. Some of these registers are state that is saved and restored for supporting separate software threads.

8.10.1 BR00—BLT Opcode & Control

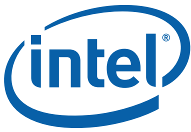
Memory Offset Address: none
Default: 0000 0000
Attributes: not accessible

BR00 is the last executed instruction DWord 0. Bits [22:5] are written by every DW0 of every instruction. Bits [31:30] and [4:0] are status bits. Bits [28:27] are written from the DW0 [15:14] of a Setup instruction and Bit 29 is written with a 1 when ever a Setup instruction is written. Bit 29 is a decode of the Setup instruction Opcode.

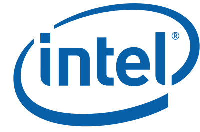




Bit	Descriptions
31	**BLT Engine Busy.** This bit indicates whether the BLT Engine is busy (1) or idle (0). This bit is replicated in the SETUP BLT Opcode & Control register. 1 = Busy 0 = Idle
30	**Setup Instruction Instruction.** The current instruction performs clipping (1).
29	**Setup Monochrome Pattern.** This bit is decoded from the Setup instruction opcode to identify whether a color (0) or monochrome (1) pattern is used with the SCANLINE_BLT instruction. 1 = Monochrome 0 = Color
28:22	**Instruction Target (Opcode).** This is the contents of the Instruction Target field from the last BLT instruction. This field is used by the BLT Engine state machine to identify the BLT instruction it is to perform. The opcode specifies whether the source and pattern operands are color or monochrome.
21:20	**32 bpp byte mask:** 21 = 1 = write alpha channel [31:24]; 20 = 1 = write RGB channels [23:00]. This field is only used for 32bpp.
19:17	**Monochrome Source Start.** This field indicates the starting monochrome pixel bit position within a byte per scan line of the source operand. The monochrome source is word aligned which means that at the end of the scan line all bits should be discarded until the next word boundary.
16	**Bit/Byte Packed.** Byte packed is for the NT driver 0 = Bit 1 = Byte
15	**Src Tiling Enable:** 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
14:12	**Horizontal Pattern Seed.** This field indicates the pattern pixel position which corresponds to X = 0.
11	**Dest Tiling Enable:** 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only) When set to '1', this means that Blitter is executing in Tiled-X mode. If '0' it means that Blitter is in Linear mode. Blitter never executes in Tiled-Y mode. On reset, this bit will be '0'. This definition applies to only X,Y Blits. Non-XY blits (COLOR_BLT, SRC_COPY_BLT), will support only linear mode and will not support tiling and for them this bit will remain reserved.



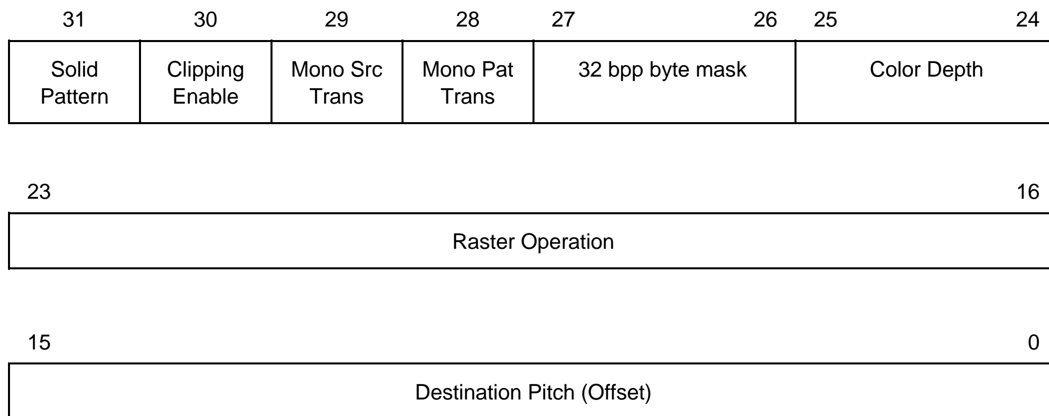
Bit	Descriptions
10:8	**Transparency Range Mode.** These bits control whether or not the byte(s) at the destination corresponding to a given pixel will be conditionally written, and what those conditions are. This feature can make it possible to perform various masking functions in order to selectively write or preserve graphics data already at the destination. XX0 = No color transparency mode enabled. This causes normal operation with regard to writing data to the destination. 001 = **[Source color transparency]** The **Transparency Color Low:** (Pixel Greater or Equal) (source background register) and the **Transparency Color High:** (Pixel Less or Equal) (source foreground register) are compared to the source pixels. The range comparisons are done on each component (R,G,B) and then logically ANDed. If the source pixel components are not within the range defined by the Transparency Color registers, then the byte(s) at the destination corresponding to the current pixel are written with the result of the bit-wise operation. 011 = **[Source and Alpha color transparency]** The **Transparency Color Low:** (Pixel Greater or Equal) (source background register) and the **Transparency Color High:** (Pixel Less or Equal) (source foreground register) are compared to the source pixels. The range comparisons are done on each component (A,R,G,B) and then logically ANDed. If the source pixel components are not within the range defined by the Transparency Color registers, then the byte(s) at the destination corresponding to the current pixel are written with the result of the bit-wise operation. 101 = **[Destination and Alpha color transparency]** The **Transparency Color Low:** (Pixel Greater or Equal) (source background register) and the **Transparency Color High:** (Pixel Less or Equal) (source foreground register) are compared to the destination pixels. The range comparisons are done on each component (A,R,G,B) and then logically ANDed. If the destination pixels are within the range, then the byte(s) at the destination corresponding to the current pixel are written with the result of the bit-wise operation. 111 = **[Destination color transparency]** The **Transparency Color Low:** (Pixel Greater or Equal) (source background register) and the **Transparency Color High:** (Pixel Less or Equal) (source foreground register) are compared to the destination pixels. The range comparisons are done on each component (R,G,B) and then logically ANDed. If the destination pixels are within the range, then the byte(s) at the destination corresponding to the current pixel are written with the result of the bit-wise operation.
7:5	**Pattern Vertical Seed.** This field specifies the pattern scan line which corresponds to Y=0.
4	**Destination Read Modify Write.** This bit is decoded from the last instruction's opcode field and Destination Transparency Mode to identify whether a Destination read is needed.
3	**Color Source.** This bit is decoded from the last instructions opcode field to identify whether a color (1) source is used.
2	**Monochrome Source.** This bit is decoded from the last instructions opcode field to identify whether a monochrome (1) source is used.
1	**Color Pattern.** This bit is decoded from the last instructions opcode field to identify whether a color (1) pattern is used.
0	**Monochrome Pattern.** This bit is decoded from the last instructions opcode field to identify whether a monochrome (1) pattern is used.

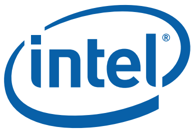


8.10.2 BR01—Setup BLT Raster OP, Control, and Destination Offset

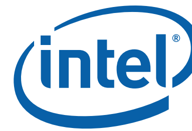
Memory Offset Address: none
Default: 0000 xxxx
Attributes: State accessible

BR01 contains the contents of the last Setup instruction DWord 1. It is identical to the BLT Raster OP, Control, and Destination Offset definition, but it is used with the following instructions: PIXEL_BLT, SCANLINE_BLT, and TEXT_BLT.

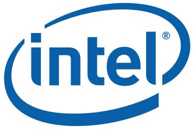




Bit	Descriptions
31	**Solid Pattern Select.** This bit applies only when the pattern data is monochrome. This bit determines whether or not the BLT Engine actually performs read operations from the frame buffer in order to load the pattern data. Use of this feature to prevent these read operations can increase BLT Engine performance, if use of the pattern data is indeed not necessary. The BLT Engine is configured to accept either monochrome or color pattern data via the opcode field. 0 = This causes normal operation with regard to the use of the pattern data. The BLT Engine proceeds with the process of reading the pattern data, and the pattern data is used as the pattern operand for all bit-wise operations. 1 = The BLT Engine forgoes the process of reading the pattern data, the presumption is made that all of the bits of the pattern data are set to 0, and the pattern operand for all bit-wise operations is forced to the background color specified in the Color Expansion Background Color Register.
30	**Clipping Enabled**: 1 = Enabled; 0 = Disabled
29	**Monochrome Source Transparency Mode.** This bit applies only when the source data is in monochrome. This bit determines whether or not the byte(s) at the destination corresponding to the pixel to which a given bit of the source data also corresponds will actually be written if that source data bit has the value of 0. This feature can make it possible to use the source as a transparency mask. The BLT Engine is configured to accepted either monochrome or color source data via the opcode field. 0 = This causes normal operation with regard to the use of the source data. Wherever a bit in the source data has the value of 0, the color specified in the background color register is used as the source operand in the bit-wise operation for the pixel corresponding to the source data bit, and the bytes at the destination corresponding to that pixel are written with the result. 1 = Wherever a bit in the source data has the value of 0, the byte(s) at the destination corresponding to the pixel to which the source data bit also corresponds are simply not written, and the data at those byte(s) at the destination are allowed to remain unchanged.
28	**Monochrome Pattern Transparency Mode.** This bit applies only when the pattern data is monochrome. This bit determines whether or not the byte(s) at the destination corresponding to the pixel to which a given bit of the pattern data also corresponds will actually be written if that pattern data bit has the value of 1. This feature can make it possible to use the pattern as a transparency mask. The BLT Engine is configured to accepted either monochrome or color pattern data via the opcode field. 0 = This causes normal operation with regard to the use of the pattern data. Wherever a bit in the pattern data has the value of 0, the color specified in the background color register is used as the pattern operand in the bit-wise operation for the pixel corresponding to the pattern data bit, and the bytes at the destination corresponding to that pixel are written with the result. 1 = Wherever a bit in the pattern data has the value of 0, the byte(s) at the destination corresponding to the pixel to which the pattern data bit also corresponds are simply not written, and the data at those byte(s) at the destination are allowed to remain unchanged.
27:26	**32 bpp byte mask.** 21 = 1 = write alpha channel [31:24]; 20 = 1 = write RGB channels [23:00]. This field is only used for 32bpp.

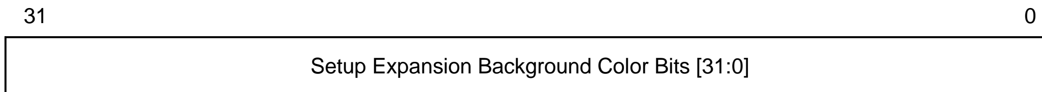


Bit	Descriptions
25:24	**Color Depth.** 00 = 8 Bit Color Depth 01 = 16 Bit Color Depth 10 = 16 Bit Color Depth 11 = 32 Bit Color Depth
23:16	**Raster Operation Select.** These 8 bits are used to select which one of 256 possible raster operations is to be performed by the BLT Engine. The 8-bit values, and their corresponding raster operations, are intended to correspond to the 256 possible raster operations specified for graphics device drivers in the Windows* environment. The opcode field must indicate a monochrome source if ROP = F0.
15:0	**Destination Pitch (Offset).** For non-XY Blits, the signed 16bit field allows for specifying upto $\pm$ 32Kbytes signed pitches in bytes (same as before). For X, Y Blits with tiled (X) surfaces, the pitch for Destination will be 512Byte aligned and should be programmable upto $\pm$ 128Kbytes. In this case, this 16bit signed pitch field is used to specify upto $\pm$ 32K**DWords**. For X, Y blits with nontiled surfaces (linear surfaces), this 16bit field can be programmed to byte specification of upto $\pm$ 32Kbytes (same as before). These 16 bits store the signed memory address offset value by which the destination address originally specified in the Destination Address Register is incremented or decremented as each scan line's worth of destination data is written into the frame buffer by the BLT Engine, so that the destination address will point to the next memory address to which the next scan line's worth of destination data is to be written. If the intended destination of a BLT operation is within on-screen frame buffer memory, this offset is normally set so that each subsequent scan line's worth of destination data lines up vertically with the destination data in the scan line, above. However, if the intended destination of a BLT operation is within off-screen memory, this offset can be set so that each subsequent scan line's worth of destination data is stored at a location immediately after the location where the destination data for the last scan line ended, in order to create a single contiguous block of bytes of destination data at the destination.

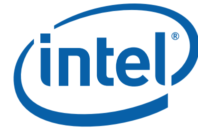


8.10.3 BR05—Setup Expansion Background Color

Memory Offset Address: none
Default: None
Attributes: State accessible

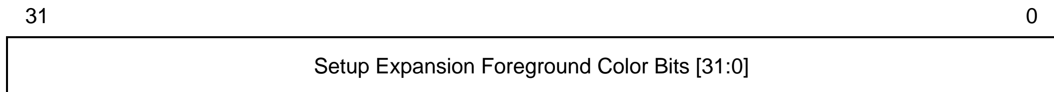


Bit	Descriptions
31:0	**Setup Expansion Background Color Bits [31:0].** These bits provide the one, two, or four bytes worth of color data that select the background color to be used in the color expansion of monochrome pattern or source data for either the SCANLINE_BLT or TEXT_BLT instructions. BR05 is also used as the solid pattern for the PIXEL_BLT instruction. Whether one, two, or three bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.

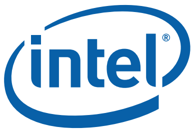


8.10.4 BR06—Setup Expansion Foreground Color

Memory Offset Address: none
Default: None
Attributes: State accessible

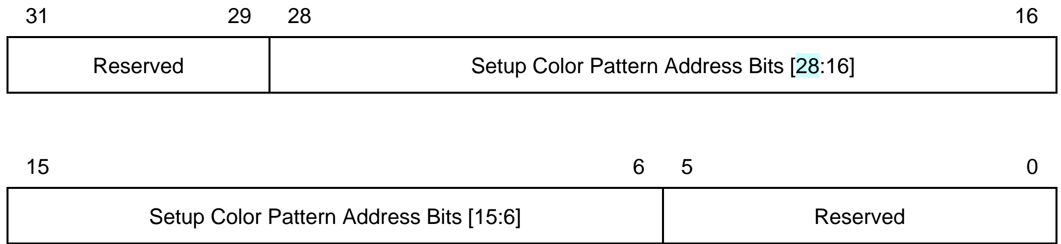


Bit	Descriptions
31:24	**Reserved.**
31:0	**Setup Expansion Foreground Color Bits [31:0].** These bits provide the one, two, or four bytes worth of color data that select the foreground color to be used in the color expansion of monochrome pattern or source data for either the SCANLINE_BLT or TEXT_BLT instructions. Whether one, two, or three bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.

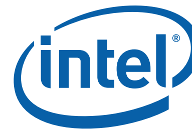


8.10.5 BR07—Setup Color Pattern Address

Memory Offset Address: none
Default: None
Attributes: State accessible

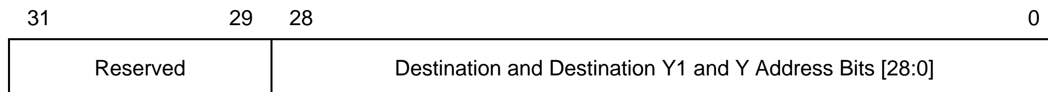


Bit	Descriptions
31:29	**Reserved.** The maximum GC graphics address is 512 MBs.
28:6	**Pattern Address.** These 23 bits specify the starting address of the color pattern from the SETUP_BLT instruction. This register works identically to the Pattern Address register, but this version is only used with the SCANLINE_BLT instruction execution. The pattern data must be located in linear memory. The pattern data must be located on a pattern-size boundary. The pattern is always of 8x8 pixels, and therefore, its size is dependent upon its pixel depth. The pixel depth may be 8, 16, or 32 bits per pixel if the pattern is in color (the pixel depth of a color pattern must match the pixel depth to which the graphics system has been set). Monochrome patterns require 8 bytes and are supplied through the instruction. Color patterns of 8, 16, and 32 bits per pixel color depth must start on 64-byte, 128-byte and 256-byte boundaries, respectively.
5:0	**Reserved.** These bits always return 0 when read.

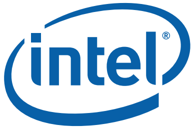


8.10.6 BR09—Destination Address

Memory Offset Address: None
Default: None
Attributes: State accessible

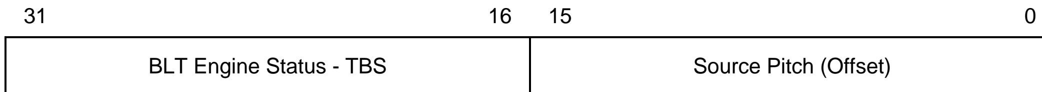


Bit	Descriptions
31:29	**Reserved.**
28:0	**Destination Address Bits.** When tiling is enabled for XY-blits, this base address should be limited to 4KB. Otherwise for XY blits, there is no restriction and it is same as before. These 29 bits specify the starting pixel address of the destination data. This register is also the working destination address register and changes as the BLT Engine performs the accesses. Used as the scan line address (Destination Y Address & Destination Y1 Address) for BLT instructions: PIXEL_BLT, SCANLINE_BLT, and TEXT_BLT. In this case the address points to the first pixel in a scan line and is compared with the ClipRect Y1 & Y2 address registers to determine whether the scan line should be written or not. The Destination Y1 address is the top scan line to be written for text. Note that for non-XY blits (COLOR_BLT, SRC_COPY_BLT), this address points to the first byte to be written. This register is always the last register written for a BLT drawing instruction. Writing BR09 starts the BLT engine execution. ***Note:*** Some instructions affect only one scan line (requiring only one coordinate); other instructions affect multiple scan lines and need both coordinates.

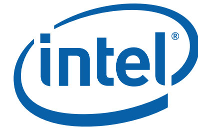


8.10.7 BR11—BLT Source Pitch (Offset)

Memory Offset Address: None
Default: None
Attributes: Not accessible

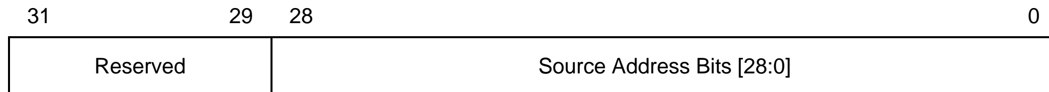


Bit	Descriptions
31:16	**BLT Engine Status.** This field is used to read back important debug status. It will be specified in the future.
15:0	**Source Pitch (Offset)** For non-XY Blits with color source operand (SRC_COPY_BLT), the signed 16bit field allows for specifying upto ± 32Kbytes signed pitch in bytes (same as before). For X, Y Blits with tiled (X) surfaces, the pitch for Color Source will be 512Byte aligned and should be programmable upto ± 128Kbytes. In this case, this 16bit signed pitch field is used to specify upto ± 32K**DWords**. For X, Y blits with nontiled color source surfaces (linear surfaces), this 16bit field can be programmed to byte specification of upto ± 32Kbytes (same as before). When the color source data is located within the frame buffer or AGP aperture, these signed 16 bits store the memory address offset (pitch) value by which the source address originally specified in the Source Address Register is incremented or decremented as each scan line's worth of source data is read from the frame buffer by the BLT Engine, so that the source address will point to the next memory address from which the next scan line's worth of source data is to be read. Note that if the intended source of a BLT operation is within on-screen frame buffer memory, this offset is normally set to accommodate the fact that each subsequent scan line's worth of source data lines up vertically with the source data in the scan line, above. However, if the intended source of a BLT operation is within off-screen memory, this offset can be set to accommodate a situation in which the source data exists as a single contiguous block of bytes where in each subsequent scan line's worth of source data is stored at a location immediately after the location where the source data for the last scan line ended.



8.10.8 BR12—Source Address

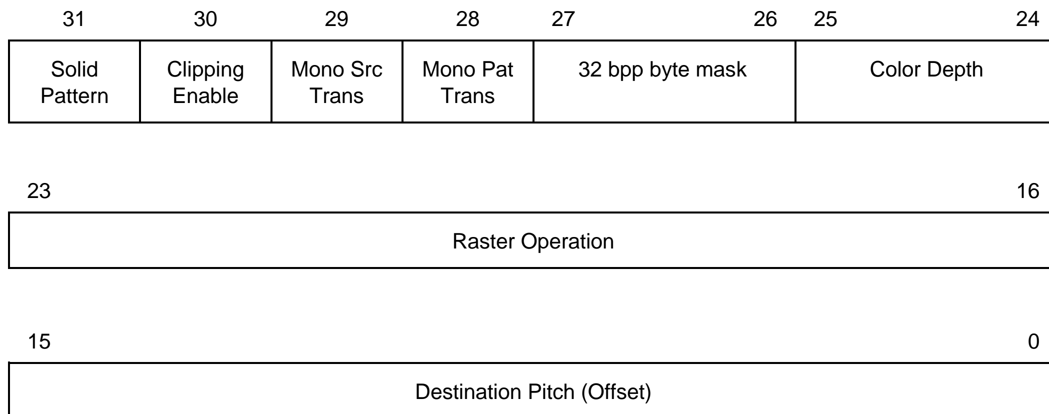
Memory Offset Address: None
Default: None
Attributes: Not accessible

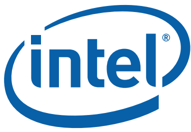


Bit	Descriptions
31:29	**Reserved.** The maximum GC Graphics address is 512 MBs.
28:0	**Source Address Bits [28:0].** When tiling is enabled for XY-blits with Color source surfaces, this base address should be limited to 4KB. Otherwise for XY blits, there is no restriction and it is same as before, including for monosource and text blits. Note that for non-XY blit with Color Source (SRC_COPY_BLT), this address points to the first byte to be read. These 29 bits are used to specify the starting pixel address of the color source data. The lower 3 bits are used to indicate the position of the first valid byte within the first Quadword of the source data.

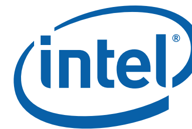
8.10.9 BR13—BLT Raster OP, Control, and Destination Pitch

Memory Offset Address: None
Default: 0000 xxxx
Attributes: Not accessible





Bit	Descriptions
31	**Solid Pattern Select.** This bit applies only when the pattern data is monochrome. This bit determines whether or not the BLT Engine actually performs read operations from the frame buffer in order to load the pattern data. Use of this feature to prevent these read operations can increase BLT Engine performance, if use of the pattern data is indeed not necessary. The BLT Engine is configured to accept either monochrome or color pattern data via the opcode field. 0 = This causes normal operation with regard to the use of the pattern data. The BLT Engine proceeds with the process of reading the pattern data, and the pattern data is used as the pattern operand for all bit-wise operations. 1 = The BLT Engine forgoes the process of reading the pattern data, the presumption is made that all of the bits of the pattern data are set to 0, and the pattern operand for all bit-wise operations is forced to the background color specified in the Color Expansion Background Color Register.
30	**Clipping Enabled**: 1 = Enabled; 0 = Disabled
29	**Monochrome Source Transparency Mode.** This bit applies only when the source data is in monochrome. This bit determines whether or not the byte(s) at the destination corresponding to the pixel to which a given bit of the source data also corresponds will actually be written if that source data bit has the value of 0. This feature can make it possible to use the source as a transparency mask. The BLT Engine is configured to accepted either monochrome or color source data via the opcode field. 0 = This causes normal operation with regard to the use of the source data. Wherever a bit in the source data has the value of 0, the color specified in the background color register is used as the source operand in the bit-wise operation for the pixel corresponding to the source data bit, and the bytes at the destination corresponding to that pixel are written with the result. 1 = Where a bit in the source data has the value of 0, the byte(s) at the destination corresponding to the pixel to which the source data bit also corresponds are simply not written, and the data at those byte(s) at the destination are allowed to remain unchanged.
28	**Monochrome Pattern Transparency Mode.** This bit applies only when the pattern data is monochrome. This bit determines whether or not the byte(s) at the destination corresponding to the pixel to which a given bit of the pattern data also corresponds will actually be written if that pattern data bit has the value of 1. This feature can make it possible to use the pattern as a transparency mask. The BLT Engine is configured to accepted either monochrome or color pattern data via the opcode in the Opcode and Control register. 0 = This causes normal operation with regard to the use of the pattern data. Where a bit in the pattern data has the value of 0, the color specified in the background color register is used as the pattern operand in the bit-wise operation for the pixel corresponding to the pattern data bit, and the bytes at the destination corresponding to that pixel are written with the result. 1= Wherever a bit in the pattern data has the value of 0, the byte(s) at the destination corresponding to the pixel to which the pattern data bit also corresponds are simply not written, and the data at those byte(s) at the destination are allowed to remain unchanged.
25:24	**Color Depth.** 00 = 8 Bit Color Depth 01 = 16 Bit Color Depth 10 = 24 Bit Color Depth 11 = Reserved

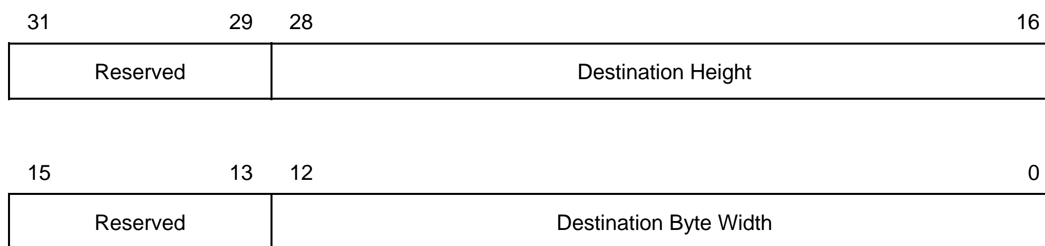


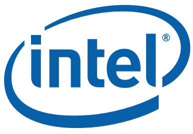
Bit	Descriptions
23:16	**Raster Operation Select.** These 8 bits are used to select which one of 256 possible raster operations is to be performed by the BLT Engine. The 8-bit values, and their corresponding raster operations, are intended to correspond to the 256 possible raster operations specified for graphics device drivers in the Windows* environment. The opcode must indicate a monochrome source operand if ROP = F0.
15:0	**Destination Pitch (Offset).** These 16 bits store the signed memory address offset value by which the destination address originally specified in the Destination Address Register is incremented or decremented as each scan line's worth of destination data is written into the frame buffer by the BLT Engine, so that the destination address will point to the next memory address to which the next scan line's worth of destination data is to be written. If the intended destination of a BLT operation is within on-screen frame buffer memory, this offset is normally set so that each subsequent scan line's worth of destination data lines up vertically with the destination data in the scan line, above. However, if the intended destination of a BLT operation is within off-screen memory, this offset can be set so that each subsequent scan line's worth of destination data is stored at a location immediately after the location where the destination data for the last scan line ended, in order to create a single contiguous block of bytes of destination data at the destination.

8.10.10 BR14—Destination Width & Height

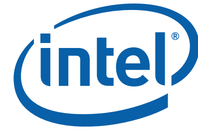
Memory Offset Address: None
Default: None
Attributes: Not accessible

BR14 contains the values for the height and width of the data to be BLT. If these values are not correct, such that the BLT Engine is either expecting data it does not receive or receives data it did not expect, the system can hang.



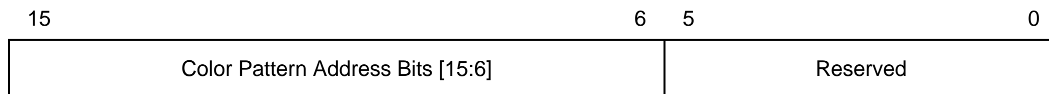


Bit	Descriptions
31:29	**Reserved.**
28:16	**Destination Height.** These 13 bits specify the height of the destination data in terms of the number of scan lines. This is a working register.
15:13	**Reserved.**
12:0	**Destination Byte Width.** These 13 bits specify the width of the destination data in terms of the number of bytes per scan line. The number of pixels per scan line into which this value translates depends upon the color depth to which the graphics system has been set.



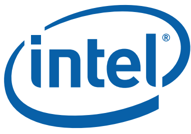
8.10.11 BR15—Color Pattern Address

Memory Offset Address: None
Default: None
Attributes: Not accessible



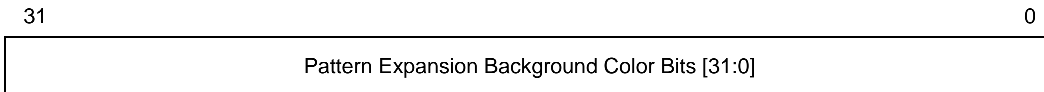
.

Bit	Descriptions
31:29	**Reserved.** The maximum GC graphics address is 512 MBs.
28:6	**Color Pattern Address.** There is no change to the Color Pattern address specification due to Non-Power-of-2 change. It remains the same as before. The pattern data must be located in linear memory. These 23 bits specify the starting address of the pattern. The pattern data must be located on a pattern-size boundary. The pattern is always of 8x8 pixels, and therefore, its size is dependent upon its pixel depth. The pixel depth may be 8, 16, or 32 bits per pixel if the pattern is in color (the pixel depth of a color pattern must match the pixel depth to which the graphics system has been set). Monochrome patterns require 8 bytes and are applied through the instruction. Color patterns of 8, 16, and 32 bits per pixel color depth must start on 64-byte, 128-byte and 256-byte boundaries, respectively.
5:0	**Reserved.** These bits always return 0 when read.



8.10.12 BR16—Pattern Expansion Background & Solid Pattern Color

Memory Offset Address: 40040h
Default: None
Attributes: RO; DWord accessible



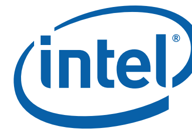
.

Bit	Descriptions
31:0	**Pattern Expansion Background Color Bits [31:0].** These bits provide the one, two, or four bytes worth of color data that select the background color to be used in the color expansion of monochrome pattern data during BLT operations. Whether one, two, or four bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.

8.10.13 BR17—Pattern Expansion Foreground Color

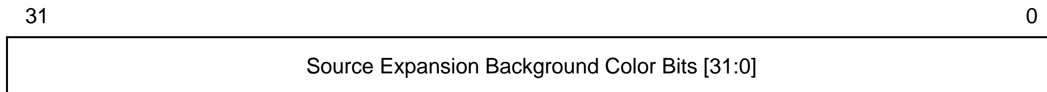
Memory Offset Address: None
Default: None
Attributes: Not accessible

Bit	Descriptions
31:0	**Pattern Expansion Foreground Color Bits [31:0].** These bits provide the one, two, or four bytes worth of color data that select the foreground color to be used in the color expansion of monochrome pattern data during BLT operations. Whether one, two, or four bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.



8.10.14 BR18—Source Expansion Background, and Destination Color

Memory Offset Address: None
Default: None
Attributes: Not accessible



Bit	Descriptions
31:0	**Source Expansion Background Color Bits [31:0].** These bits provide the one, two, or four bytes worth of color data that select the background color to be used in the color expansion of monochrome source data during BLT operations. This register is also used to support destination transparency mode and Solid color fill. Whether one, two, three, or four bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.

8.10.15 BR19—Source Expansion Foreground Color

Memory Offset Address: None
Default: None
Attributes: Not accessible

Bit	Descriptions
31:0	**Pattern/Source Expansion Foreground Color Bits [31:0].** These bits provide the one, two, or four bytes worth of color data that select the foreground color to be used in the color expansion of monochrome source data during BLT operations. Whether one, two, or four bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.