# Intel® Arc™ A-Series Graphics and Intel Data Center GPU Flex Series

# Open-Source Programmer's Reference Manual

# For the discrete GPUs code named "Alchemist" and "Arctic Sound-M"

## Volume 10: Copy Engine

March 2023, Revision 1.0

## Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks

Customer is responsible for safety of the overall system, including compliance with applicable safety-related requirements or standards.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exceptions that a) you may publish an unmodified copy and b) code included in this document is licensed subject to Zero-Clause BSD open source license (0BSD). You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

The products described 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.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

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 information here is subject to change without notice. Do not finalize a design with this information.

# Table of Contents

# Copy Engine

Copy Engine or BLT Engine is an engine that runs in parallel with Render Engine, Compute Engine and Media Engine. It is capable of moving blocks of data from one location (source) in the memory to another location (destination) in the memory. It can also fill up a specified location in the memory with fixed data. Copy Engine can perform pre-defined logical or bitwise operation on source, destination and another fixed data called pattern data and write to the destination confined by a clip rectangle. Copy Engine can be divided into two categories:

1. Classical BLT Engine - Can perform basic Copy and Fill operations as well as bit-wise operations. It is low performance.
2. Fast Copy Engine - performs basic copy and fill operations. It is high performance. It can also support compute specific copy operations other than the regular 2D operations.

Fast Copy Engine supports only few  high performance commands, whereas Classical BLT Engine, which is also known as Legacy Blitter, supports all others command, including some very complex commands involving bit-wise/logical operations.

## Blitter (BLT) Command Streamer

Blitter Command Streamer is the BLT engine command streamer where scheduler submits workloads for the BLT engine. Blitter Command streamer supports execution of the generic functions plus BLT engine commands. Each BLT Engine has its own independent command streamer.

## Blitter Engine Command Streamer (BCS)

The BCS (Blitter Command Streamer) unit primarily serves as the software programming interface between the O/S driver and the Blitter Engine. It is responsible for fetching, decoding, and dispatching of data packets (Blitter Commands) to the front-end interface module of Blitter Engine.

| Logic Functions Included |
|---|
| • MMIO register programming interface.<br>• DMA action for fetching of ring data from memory.<br>• Management of the Head pointer for the Ring Buffer.<br>• Decode of ring data and sending it to the blit engine.<br>• Handling of user interrupts.<br>• Flushing the Blitter Engine.<br>• Handle NOP. |
| • DMA action for fetching of execlists from memory.<br>• Handling of ring context switch interrupt. |

The BCS unit only claims memory mapped I/O cycles that are targeted to its range of 0x22000 to 0x224FF. The Blitter, Render and Media Engines use semaphore to synchronize their operations.

BCS operates completely independent of the other render and media command streams.

The simple sequence of events is as follows: a ring (say PRB0) is programmed by a memory-mapped register write cycle. The DMA inside BCS is kicked off. The DMA fetches commands from memory based on the starting address and head pointer. The DMA requests cache lines from memory (one cacheline CL at a time). There is guaranteed space in the DMA FIFO (8 CL deep) for data coming back from memory. The DMA control logic has copies of the head pointer and the tail pointer. The DMA increments the head pointer after making requests for ring commands. Once the DMA copy of the head pointer becomes equal to the tail pointer, the DMA stops requesting.

The parser starts executing once the DMA FIFO has valid commands. All the commands have a header DWord packet. Based on the encoding in the header packet, the command may be targeted towards Blit Engine or the command parser. After execution of every command, the actual head pointer is updated. The ring is considered empty when the head pointer becomes equal to the tail pointer.

## Context Management

### Copy Engine Register State Context

### Register State Context

| |
|---|
| EXECLIST CONTEXT |
| EXECLIST CONTEXT(PPGTT Base) |
| ENGINE CONTEXT |

| Description | MMIO Offset/Command | Unit | Dw Count | DW Offset |
|---|---|---|---|---|
| **CSFE Execlist Context** | | BCSFE | 192 | **0** |
| MI_BATCH_BUFFER_END | | CSEND | 1 | **00C0** |
| NOOP | | CSEND | 127 | **00C1** |
| | | | DW | 320 |
| | | | K Bytes | 1.28 |

### Copy Engine Power Context

This section lists the power context image of Copy Engine across generations.

### Blitter Engine Power Context

The table below captures the data from BCS power context save/restored by PM. Address offset in the below table is relative to the starting location of BCS in the overall power context image managed by PM.

### BCS Power Context Image

| Description | | | # of DW | Address Offset(PWR) | CSFE/CSBE |
|---|---|---|---|---|---|
| **CSFE Power Context with Display** | | | 192 | **0** | **CSFE** |
| NOOP | | BCS | 1 | **00D0** | **CSBE** |
| Load_Register_Immediate header | 0x1100_1003 | BCS | 1 | **00D1** | **CSBE** |
| GAB MODE REGISTER | 0x220a0 | BCS | 2 | **00D4** | **CSBE** |
| NOOP | | BCS | 8 | **00D6** | **CSBE** |
| NOOP | | BCS | 1 | **00DE** | **CSBE** |
| MI_BATCH_BUFFER_END | | BCS | 1 | **00DF** | **CSBE** |

## Blitter Command Formats

### 2D Commands

The 2D commands include various flavors of BLT operations, along with commands to set up BLT engine state without actually performing a BLT. Most commands are of fixed length, though there are a few commands that include a variable amount of "inline" data at the end of the command (in case of legacy Blitter command).

All the following commands are defined in *Blitter Instructions*.

### 2D Command Map

| Opcode (28:22) | Command |
|---|---|
| 00h | Reserved |
| 01h | XY_SETUP_BLT |
| 02h | Reserved |
| 03h | XY_SETUP_CLIP_BLT |
| 04h-10h | Reserved |
| 11h | XY_SETUP_MONO_PATTERN_SL_BLT |
| 12h-23h | Reserved |
| 24h | XY_PIXEL_BLT |
| 25h | XY_SCANLINES_BLT |
| 26h | XY_TEXT_BLT |
| 27h-30h | Reserved |
| 31h | XY_TEXT_IMMEDIATE_BLT |
| 32h-3Fh | Reserved |
| 40h | COLOR_BLT |
| 41h | XY_BLOCK_COPY_BLT |
| 42h | XY_FAST_COPY_BLT |
| 43h | SRC_COPY_BLT |
| 44h | XY_FAST_COLOR_BLT |
| 45h-47h | Reserved |
| 48h | XY_CTRL_SURF_COPY_BLT |
| 49h-4Fh | Reserved |
| 50h | XY_COLOR_BLT |
| 51h | XY_PAT_BLT |
| 52h | XY_MONO_PAT_BLT |
| 53h | XY_SRC_COPY_BLT |

| Opcode (28:22) | Command |
|---|---|
| 54h | XY_MONO_SRC_COPY_BLT |
| 55h | XY_FULL_BLT |
| 56h | XY_FULL_MONO_SRC_BLT |
| 57h | XY_FULL_MONO_PATTERN_BLT |
| 58h | XY_FULL_MONO_PATTERN_MONO_SRC_BLT |
| 59h | XY_MONO_PAT_FIXED_BLT |
| 71h | XY_MONO_SRC_COPY_IMMEDIATE_BLT |
| 72h | XY_PAT_BLT_IMMEDIATE |
| 73h | XY_SRC_COPY_CHROMA_BLT |
| 74h | XY_FULL_IMMEDIATE_PATTERN_BLT |
| 75h | XY_FULL_MONO_SRC_IMMEDIATE_PATTERN_BL |
| 76h | XY_PAT_CHROMA_BLT |
| 77h | XY_PAT_CHROMA_BLT_IMMEDIATE |
| 78h-7Fh | Reserved |

## Blitter Command Header Format

| Type | Bits | | | | |
|---|---|---|---|---|---|
| | 31:29 | 28:24 | 23 | 22 | 21:0 |
| Memory Interface (MI) | 000 | Opcode<br>00h - NOP<br>0Xh - Single DWord Commands<br>1Xh - Two+ DWord Commands<br>2Xh - Store Data Commands<br>3Xh - Ring/Batch Buffer Cmds | | Identification No./DWord Count<br>Command Dependent Data<br>5:0 - DWord Count<br>5:0 - DWord Count<br>5:0 - DWord Count | |
| Reserved | 001 | | | | |
| Reserved | 011 | | | | |

| Type | Bits | | | |
|---|---|---|---|---|
| | 31:29 | 28:22 | 21:9 | 8:0 |
| Blitter (2D) | 010 | Command Opcode | Command Dependent Data | Dword Count |

## Logical Context Support

The following are the Logical Context Support Registers:

| Register |
|---|
| BB_ADDR - Batch Buffer Head Pointer Register |
| BB_ADDR_UDW - Batch Buffer Upper Head Pointer Register |
| SBB_ADDR - Second Level Batch Buffer Head Pointer Register |
| SBB_ADDR_UDW - Second Level Batch Buffer Upper Head Pointer Register |
| SYNC_FLIP_STATUS - Wait For Event and Display Flip Flags Register |
| SYNC_FLIP_STATUS_1 - Wait For Event and Display Flip Flags Register 1 |
| SYNC_FLIP_STATUS_2 - Wait For Event and Display Flip Flags Register 2 |
| CXT_EL_OFFSET - Exec-List Context Offset |
| BB_START_ADDR_UDW - Batch Buffer Start Upper Head Pointer Register |
| BB_ADDR_DIFF - Batch Address Difference Register |
| WAIT_FOR_RC6_EXIT - Control Register for Power Management |
| SBB_STATE - Second Level Batch Buffer State Register |
| BB_OFFSET - Batch Offset Register |
| RING_BUFFER_HEAD_PREEMPT_REG - RING_BUFFER_HEAD_PREEMPT_REG |
| BB_PREEMPT_ADDR - Batch Buffer Head Pointer Preemption Register |
| BB_PREEMPT_ADDR_UDW - Batch Buffer Upper Head Pointer Preemption Register |
| SBB_PREEMPT_ADDR - Second Level Batch Buffer Head Pointer Preemption Register |
| SBB_PREEMPT_ADDR_UDW - Second Level Batch Buffer Upper Head Pointer Preemption Register |
| MI_PREDICATE_RESULT_1 - Predicate Rendering Data Result 1 |
| MI_PREDICATE_RESULT_2 - Predicate Rendering Data Result 2 |
| INDIRECT_CTX - Indirect Context Pointer |
| INDIRECT_CTX_OFFSET - Indirect Context Offset Pointer |
| BB_PER_CTX_PTR - Batch Buffer Per Context Pointer |

## Mode Registers

The following table describes the Mode Registers.

| Registers |
|---|
| BCS_CXT_SIZE - BCS Context Sizes |
| MI_MODE - Mode Register for Software Interface |
| INSTPM - Instruction Parser Mode Register |
| EXCC - Execute Condition Code Register |
| IDLEDLY - Idle Switch Delay |
| SEMA_WAIT_POLL - Semaphore Polling Interval on Wait |
| RESET_CTRL - Reset Control Register |
| HWS_PGA - Hardware Status Page Address Register |

## MI Commands for Blitter Engine

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. 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.

| Commands |
|---|
| MI_NOOP |
| MI_ARB_ON_OFF |
| MI_BATCH_BUFFER_START |

The following table lists the non-privileged registers that can be written to from a non-secure batch buffer executed from Render Command Streamer.

### User Mode Non-Privileged Registers

| MMIO Name | MMIO Offset | Size in DWords |
|---|---|---|
| BCS_GPR | 22600h | 32 |
| BCS_SWCTRL | 22200h | 32 |

| Commands |
|---|
| MI_BATCH_BUFFER_END |
| MI_CONDITIONAL_BATCH_BUFFER_END |
| MI_DISPLAY_FLIP |
| MI_LOAD_SCAN_LINES_EXCL |
| MI_LOAD_SCAN_LINES_INCL |
| MI_FLUSH_DW |
| MI_REPORT_HEAD |
| MI_STORE_DATA_IMM |
| MI_ATOMIC |
| MI_COPY_MEM_MEM |
| MI_LOAD_REGISTER_REG |
| MI_LOAD_REGISTER_MEM |
| MI_STORE_REGISTER_MEM |
| MI_SUSPEND_FLUSH |
| MI_USER_INTERRUPT |
| MI_WAIT_FOR_EVENT |
| MI_SEMAPHORE_SIGNAL |

| Commands |
|---|
| MI_SEMAPHORE_WAIT |
| MI_FORCE_WAKEUP |

## Software Control Bit Definitions

Registers in the range 22XX are not protected from the load register immediate instruction if the command is executed in the non-secure batch buffer.

**BCS_SWCTRL - BCS SW Control**

## Registers for Blitter Engine Command Streamer

These are the Registers for the Blitter Engine Command Streamer.

Also see the Observability section for related information.

**GAB_MODE - Mode Register for GAB**

# BLT Engine

## 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.

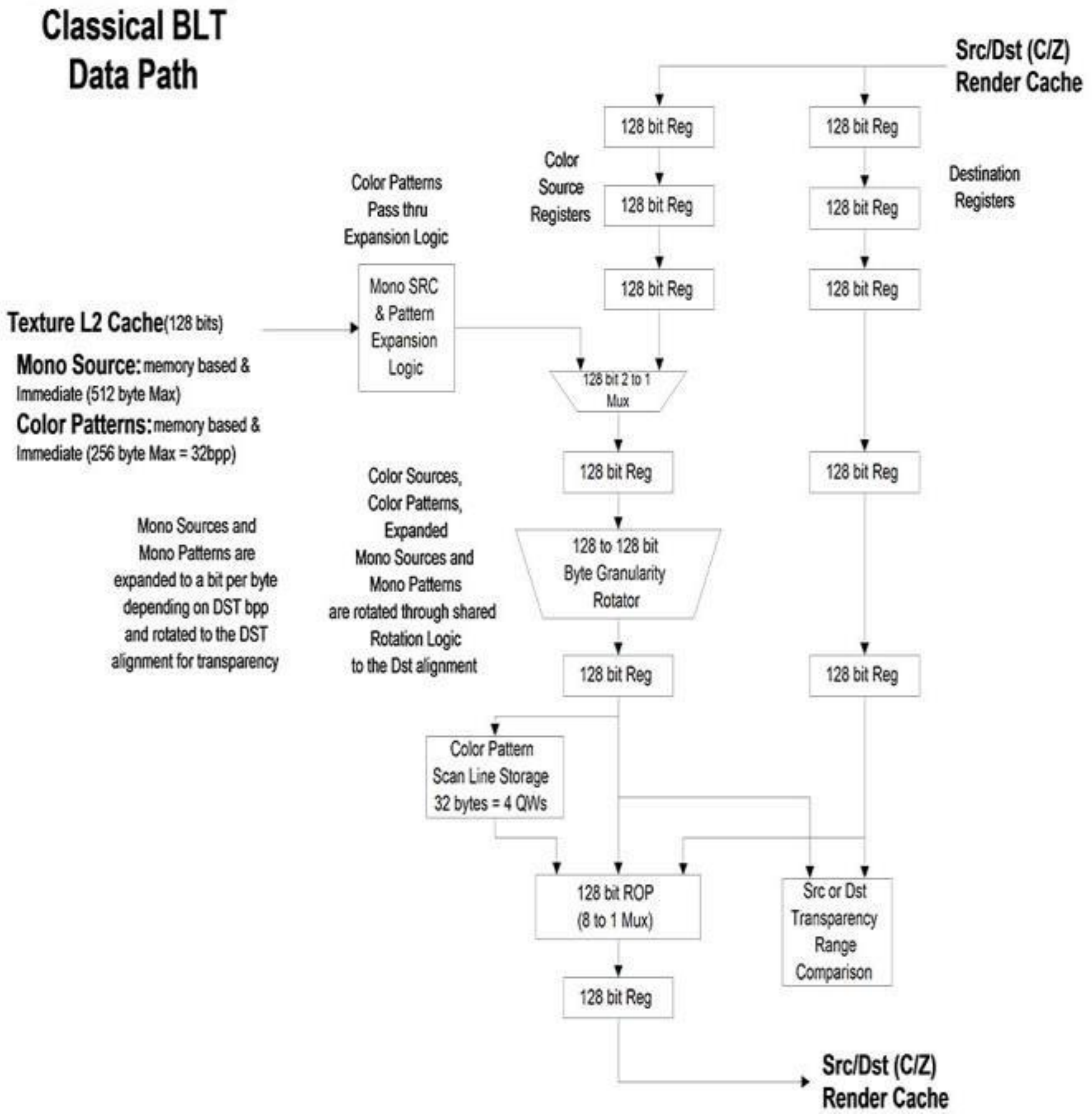
Functions such as Alpha BLTs, arithmetic (bilinear) stretch BLTs, rotations, transposing pixel maps, color space conversion, and DIBs are all considered 3D BLTs and are covered in the 3D rendering section. DIBs can be thought of as an indexed texture which uses the texture palette for performing the 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 immediate 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.

## 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.



Classical BLT Data Path
Src/Dst (C/Z) Render Cache
128 bit Reg
Color Source Registers
Destination Registers
Color Patterns Pass thru Expansion Logic
Mono SRC & Pattern Expansion Logic
Texture L2 Cache(128 bits)
Mono Source: memory based & Immediate (512 byte Max)
Color Patterns: memory based & Immediate (256 byte Max = 32bpp)
128 bit 2 to 1 Mux
Mono Sources and Mono Patterns are expanded to a bit per byte depending on DST bpp and rotated to the DST alignment for transparency
Color Sources, Color Patterns, Expanded Mono Sources and Mono Patterns are rotated through shared Rotation Logic to the Dst alignment
128 to 128 bit Byte Granularity Rotator
Color Pattern Scan Line Storage 32 bytes = 4 QWs
128 bit ROP (8 to 1 Mux)
Src or Dst Transparency Range Comparison
Src/Dst (C/Z) Render Cache

### 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.

## Basic BLT Functional Considerations

## 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 (whichever 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.

## 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.

## 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.

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.

### Bit-Wise Operations and 8-Bit Codes (00-3F)

| Code | Value Written to Bits at Destination | Code | Value Written to Bits at Destination |
|---|---|---|---|
| 00 | writes all 0's | 20 | D and ( P and ( notS )) |
| 01 | not( D or ( P or S ))) | 21 | not( S or( D xor P )) |
| 02 | D and ( not( P or S )) | 22 | D and ( notS ) |
| 03 | not( P or S ) | 23 | not( S or ( P and ( notD ))) |
| 04 | S and ( not( D or P )) | 24 | ( S xor P ) and ( D xor S ) |
| 05 | not( D or P ) | 25 | not( P xor ( D and ( not( S and P )))) |
| 06 | not( P or ( not( D xor S ))) | 26 | S xor ( D or ( P and S )) |
| 07 | not( P or ( D and S )) | 27 | S xor ( D or ( not( P xor S ))) |
| 08 | S and ( D and ( notP )) | 28 | D and ( P xor S ) |
| 09 | not( P or ( D xor S )) | 29 | not( P xor ( S xor ( D or ( P and S )))) |
| 0A | D and ( notP ) | 2A | D and ( not( P and S )) |
| 0B | not( P or ( S and ( notD ))) | 2B | not( S xor (( S xor P ) and ( P xor D ))) |
| 0C | S and ( notP ) | 2C | S xor ( P and ( D or S )) |
| 0D | not( P or ( D and ( notS ))) | 2D | P xor ( S or ( notD )) |
| 0E | not( P or ( not( D or S ))) | 2E | P xor ( S or ( D xor P )) |
| 0F | notP | 2F | not( P and ( S or ( notD ))) |
| 10 | P and ( not( D or S )) | 30 | P and ( notS ) |
| 11 | not( D or S ) | 31 | not( S or ( D and ( notP ))) |
| 12 | not( S or ( not( D xor P ))) | 32 | S xor ( D or ( P or S )) |

| Code | Value Written to Bits at Destination | Code | Value Written to Bits at Destination |
|---|---|---|---|
| 13 | not( S or ( D and P )) | 33 | notS |
| 14 | not( D or ( not( P xor S ))) | 34 | S xor ( P or ( D and S )) |
| 15 | not( D or ( P and S )) | 35 | S xor ( P or ( not( D xor S ))) |
| 16 | P xor ( S xor (D and ( not( P and S )))) | 36 | S xor ( D or P ) |
| 17 | not( S xor (( S xor P ) and ( D xor S ))) | 37 | not( S and ( D or P )) |
| 18 | ( S xor P ) and ( P xor D ) | 38 | P xor ( S and ( D or P )) |
| 19 | not( S xor ( D and ( not( P and S )))) | 39 | S xor ( P or ( notD )) |
| 1A | P xor ( D or ( S and P )) | 3A | S xor ( P or ( D xor S )) |
| 1B | not( S xor ( D and ( P xor S ))) | 3B | not( S and ( P or ( notD ))) |
| 1C | P xor ( S or ( D and P )) | 3C | P xor S |
| 1D | not( D xor ( S and ( P xor D ))) | 3D | S xor ( P or ( not( D or S ))) |
| 1E | P xor ( D or S ) | 3E | S xor ( P or ( D and ( notS ))) |
| 1F | not( P and ( D or S )) | 3F | not( P and S ) |

**Notes:**

S = Source Data<br>
P = Pattern Data<br>
D = Data Already Existing at the Destination

### 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 ))) |

| Code | Value Written to Bits at Destination | Code | Value Written to Bits at Destination |
|---|---|---|---|
| 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<br>
P = Pattern Data<br>
D = Data Already Existing at the Destination

### 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 )) |

| Code | Value Written to Bits at Destination | Code | Value Written to Bits at Destination |
|---|---|---|---|
| 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

## Bit-Wise Operations and 8-bit Codes (C0 - FF)

| Code | Value Written to Bits at Destination | Code | Value Written to Bits at Destination |
|---|---|---|---|
| C0 | P and S | E0 | P and ( D or S ) |
| C1 | not( S xor ( P or ( D and ( notS )))) | E1 | not( P xor ( D or S )) |
| C2 | not( S xor ( P or ( not( D or S )))) | E2 | D xor ( S and ( P xor D )) |
| C3 | not( P xor S ) | E3 | not( P xor ( S or ( D and P ))) |
| C4 | S and ( P or ( notD )) | E4 | S xor ( D and ( P xor S )) |
| C5 | not( S xor ( P or ( D xor S ))) | E5 | not( P xor ( D or ( S and P ))) |
| C6 | S xor ( D and ( notP )) | E6 | S xor ( D and ( not( P and S ))) |
| C7 | not( P xor ( S and ( D or P ))) | E7 | not(( S xor P ) and ( P xor D )) |
| C8 | S and ( D or P ) | E8 | S xor (( S xor P ) and ( D xor S )) |
| C9 | not( S xor ( P or D )) | E9 | not( D xor ( S xor ( P and ( not( D and S ))))) |

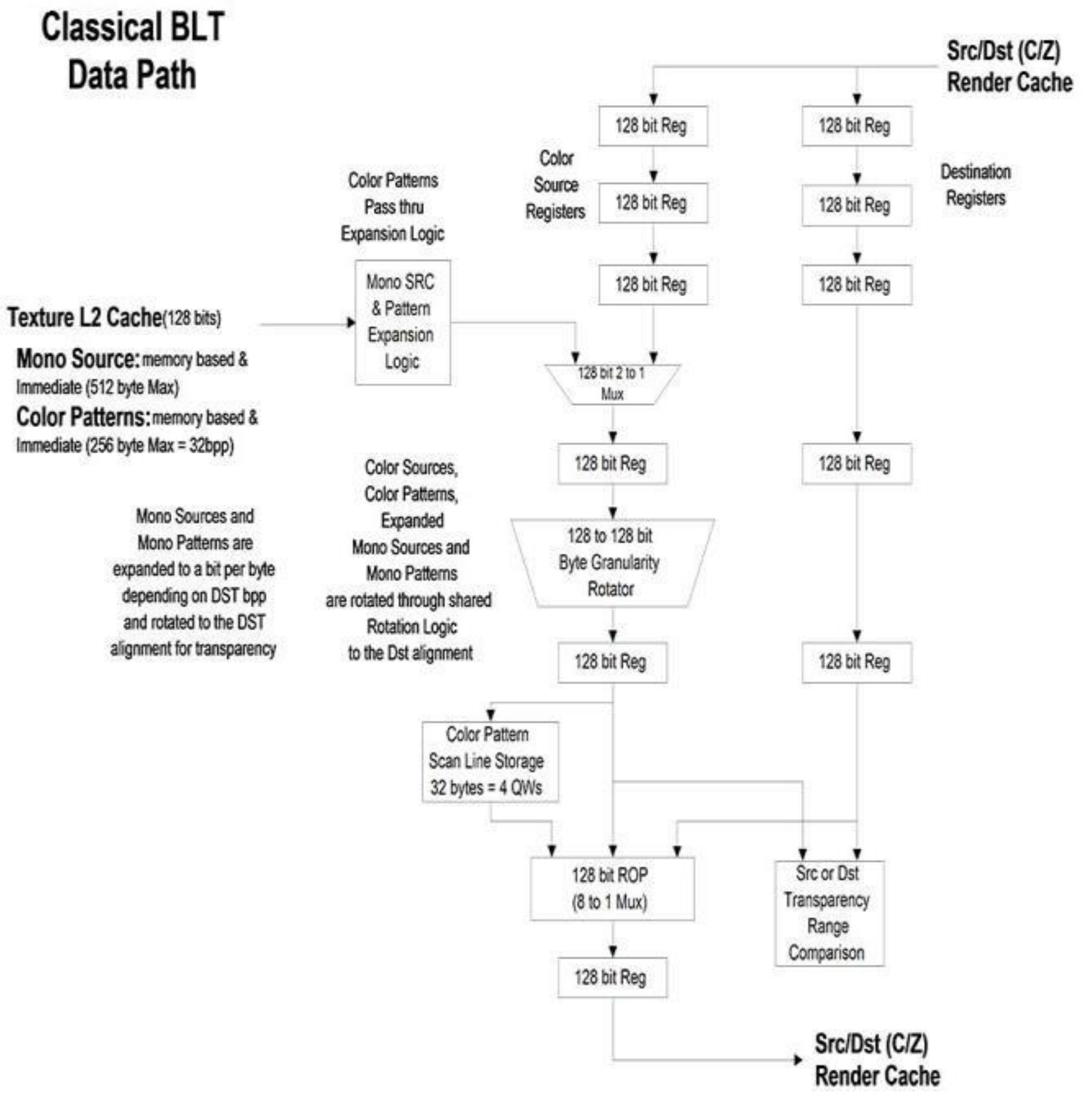
| Code | Value Written to Bits at Destination | Code | Value Written to Bits at Destination |
|---|---|---|---|
| CA | D xor ( P and ( S xor D )) | EA | D or ( P and S ) |
| CB | not( S xor ( P or ( D and S ))) | EB | D or ( not( P xor S )) |
| CC | S | EC | S or ( D and P ) |
| CD | S or ( not( D or P )) | ED | S or ( not( D xor P )) |
| CE | S or ( D and ( notP )) | EE | D or S |
| CF | S or ( notP ) | EF | S or ( D or ( notP )) |
| D0 | P and ( S or ( notD )) | F0 | P |
| D1 | not( P xor ( S or ( D xor P ))) | F1 | P or ( not( D or S )) |
| D2 | P xor ( D and ( notS )) | F2 | P or ( D and ( notS )) |
| D3 | not( S xor ( P and ( D or S ))) | F3 | P or ( notS ) |
| D4 | S xor (( S xor P ) and ( P xor D )) | F4 | P or ( S and ( notD )) |
| D5 | not( D and ( not( P and S ))) | F5 | P or ( notD ) |
| D6 | P xor ( S xor ( D or ( P and S ))) | F6 | P or ( D xor S ) |
| D7 | not( D and ( P xor S )) | F7 | P or ( not( D and S )) |
| D8 | P xor ( D and ( S xor P )) | F8 | P or ( D and S ) |
| D9 | not( S xor ( D or ( P and S ))) | F9 | P or ( not( D xor S )) |
| DA | D xor ( P and ( not( S and D ))) | FA | D or P |
| DB | not(( S xor P ) and ( D xor S )) | FB | D or ( P or ( notS )) |
| DC | S or ( P and ( notD )) | FC | P or S |
| DD | S or ( notD ) | FD | P or ( S or ( notD )) |
| DE | S or ( D xor P ) | FE | D or ( P or S ) |
| DF | S or ( not( D and P )) | FF | writes all 1's |

**Notes:**

S = Source Data<br>
P = Pattern Data<br>
D = Data Already Existing at the Destination

## 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.


Classical BLT Data Path
Src/Dst (C/Z) Render Cache
128 bit Reg
128 bit Reg
Color Source Registers
128 bit Reg
128 bit Reg
Destination Registers
128 bit Reg
128 bit Reg
Color Patterns Pass thru Expansion Logic
Mono SRC & Pattern Expansion Logic
Texture L2 Cache(128 bits)
Mono Source: memory based & Immediate (512 byte Max)
Color Patterns: memory based & Immediate (256 byte Max = 32bpp)
128 bit 2 to 1 Mux
128 bit Reg
128 bit Reg
Color Sources, Color Patterns, Expanded Mono Sources and Mono Patterns are rotated through shared Rotation Logic to the Dst alignment
Mono Sources and Mono Patterns are expanded to a bit per byte depending on DST bpp and rotated to the DST alignment for transparency
128 to 128 bit Byte Granularity Rotator
128 bit Reg
128 bit Reg
Color Pattern Scan Line Storage 32 bytes = 4 QWs
128 bit ROP (8 to 1 Mux)
Src or Dst Transparency Range Comparison
128 bit Reg
Src/Dst (C/Z) Render Cache

### 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.

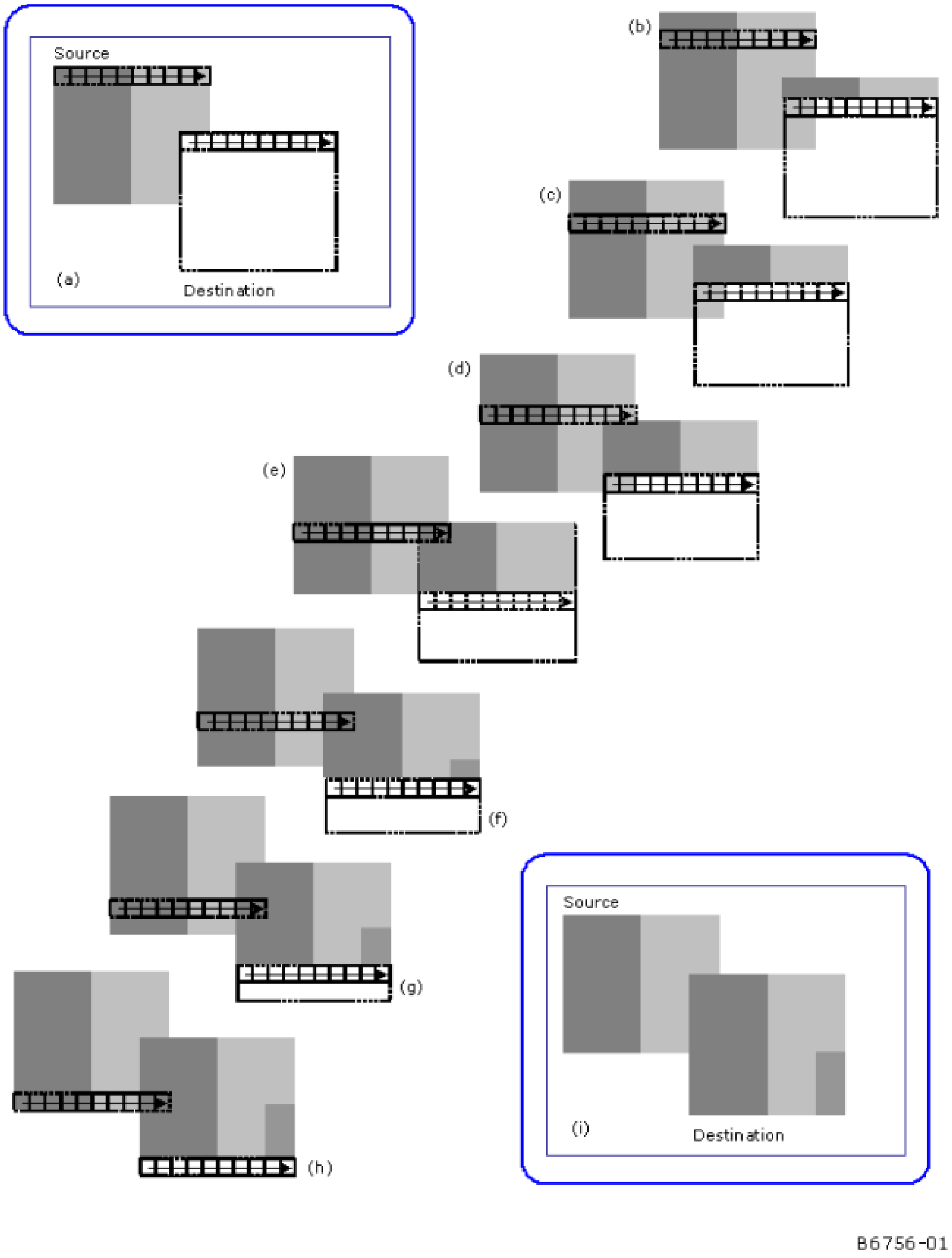
The 3-bit field, destination transparency mode, within the command packets can select per-pixel write-masking with a mask based on the results of color comparisons. The monochrome source background and foreground are range compared with either the bytes for the pixels at the destination or the source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

### 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.

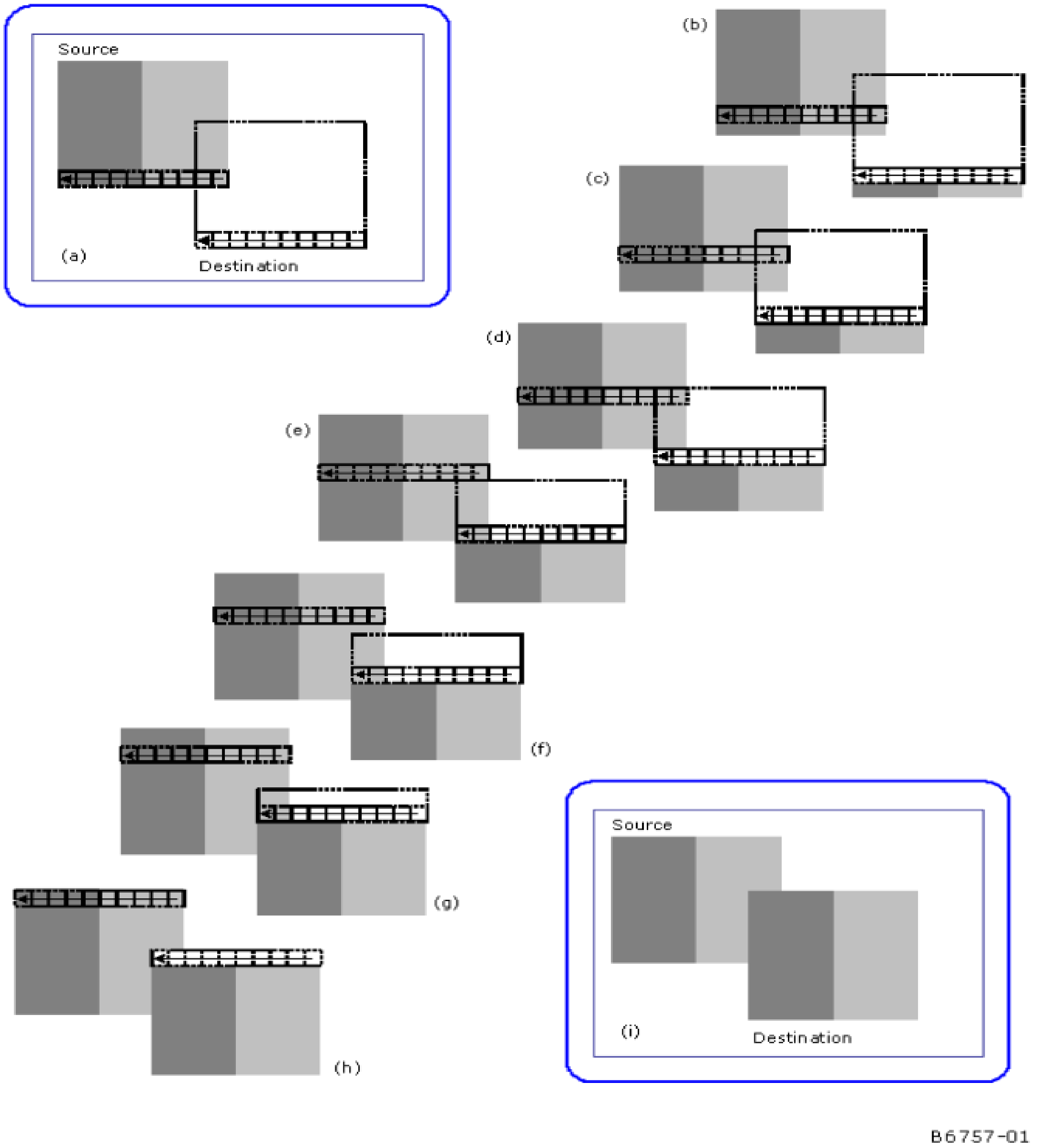
### Source Corruption in BLT with Overlapping Source and Destination Locations



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.

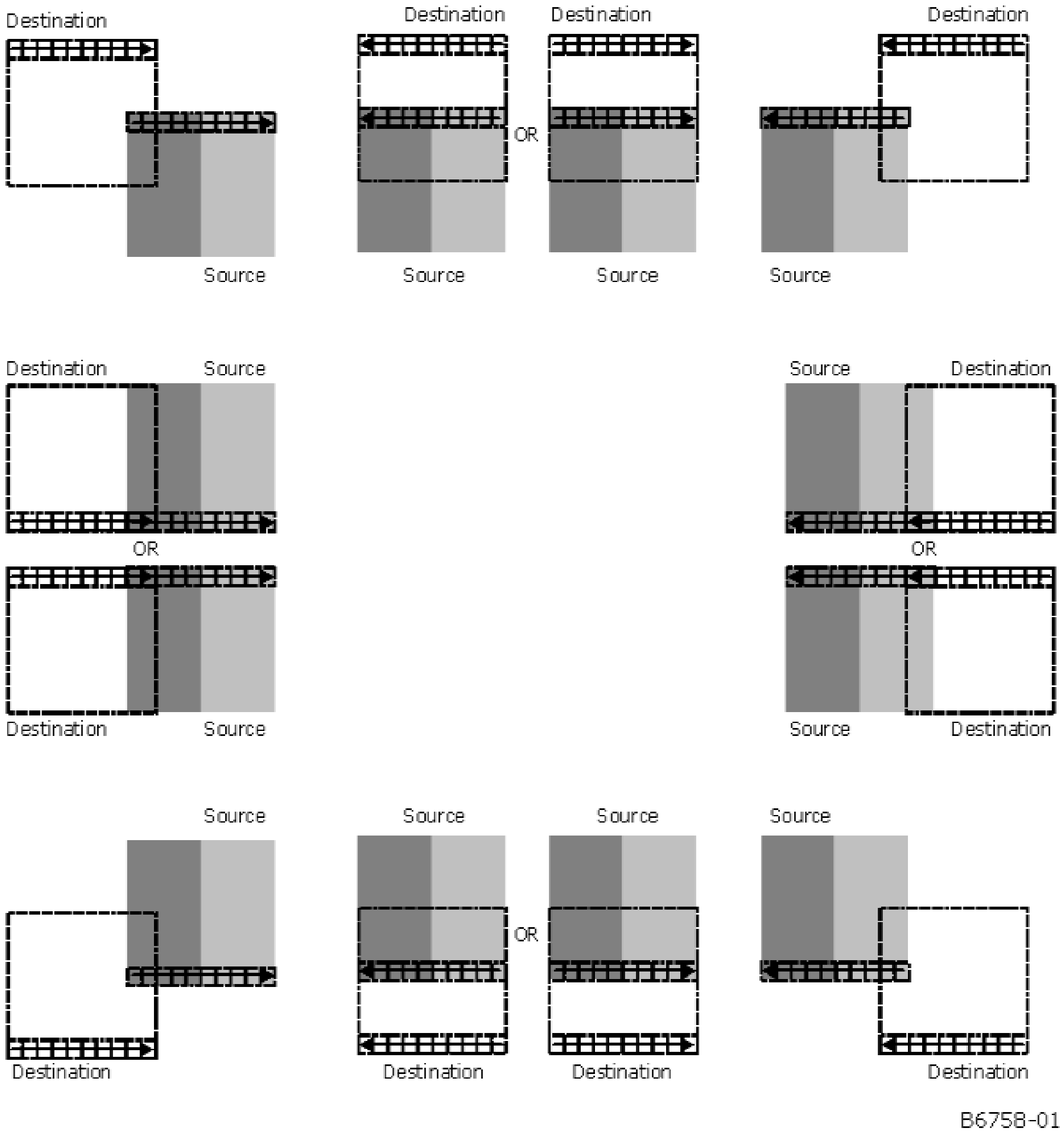
The XY command packets perform the necessary comparisons and start at the proper corner of each operand which avoids data corruption.

### Correctly Performed BLT with Overlapping Source and Destination Locations



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.

### Suggested Starting Points for Possible Source and Destination Overlap Situations



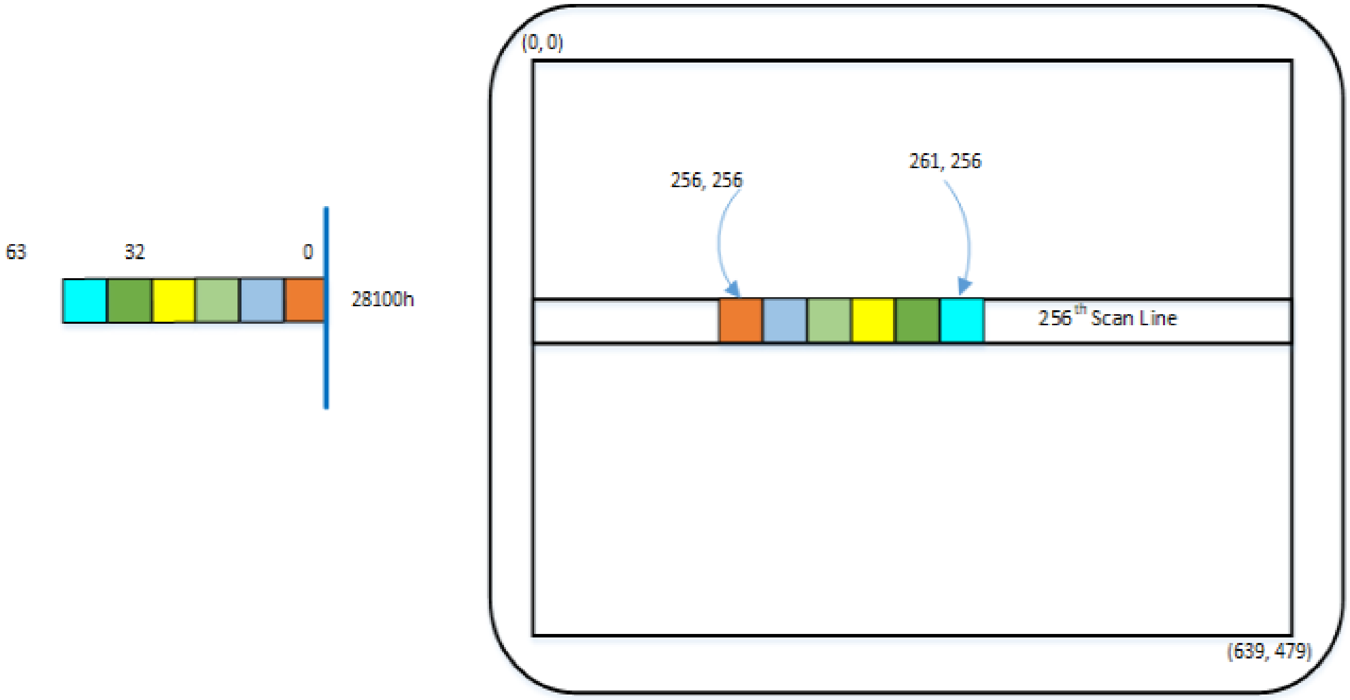
The figure above shows the recommended lines and pixels to be used as starting points in each of 8 possible ways in which the source and destination locations may overlap. In general, the starting point should be within the area in which the source and destination overlap.

## Basic Graphics Data Considerations

### 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.

**Representation of On-Screen Single 6-Pixel Line in the Frame Buffer**

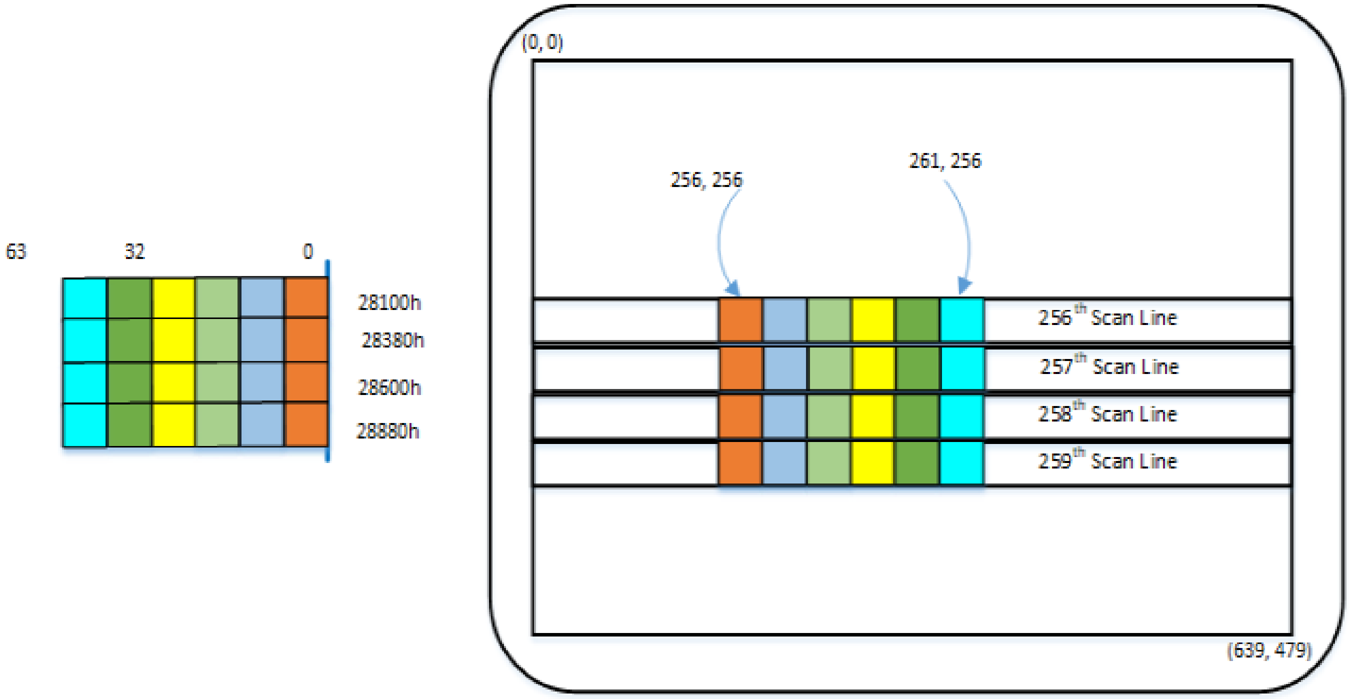


The figure above shows an example of contiguous graphics data -- a horizontal line made up of six adjacent pixels within a single scan line on a display with a resolution of 640x480. Presuming that the graphics system driving this display has been set to 8 bits per pixel and that the frame buffer's starting 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.

### 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.

## 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.

## 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.

Note that the Monosource surface start Base Address, should always be Cache Line (64byte) aligned.

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.

## Pattern Data

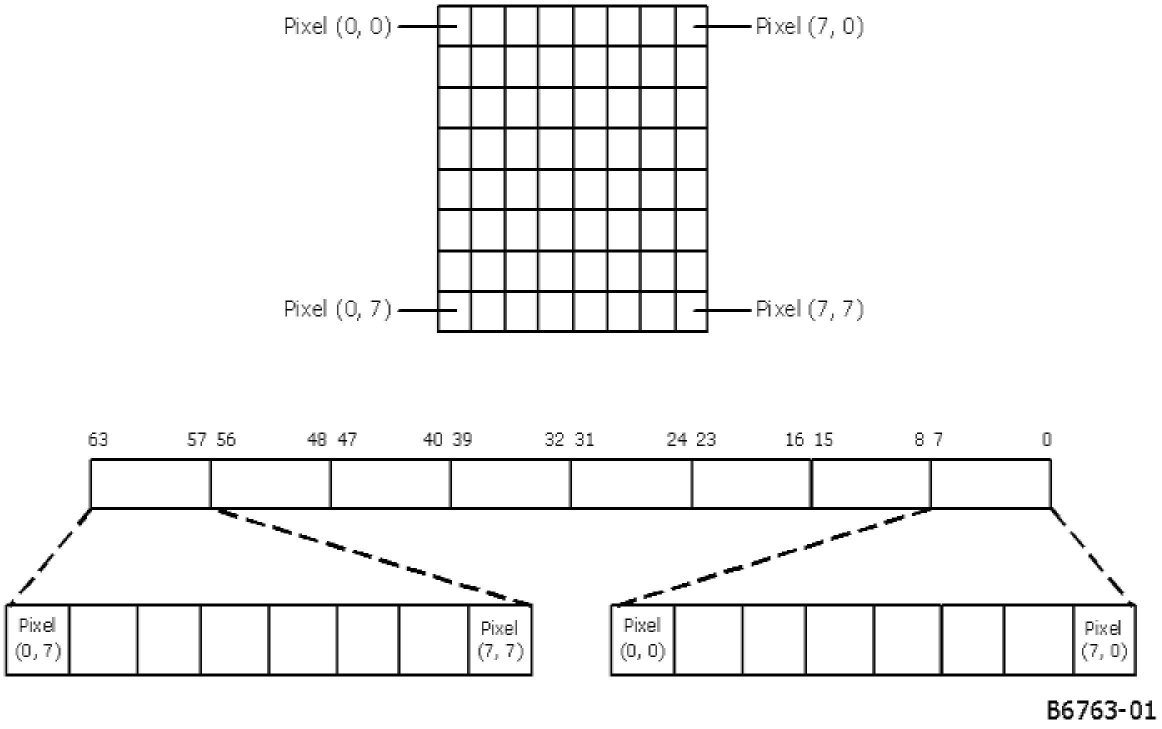
The color pattern data must exist within the frame buffer or Graphics aperture where the BLT engine may read it directly or it can be sent through the command stream. The pattern data must be located in linear memory.

Note also that the Color Pattern surface start Base Address, should always be Cache Line (64byte) aligned.

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.

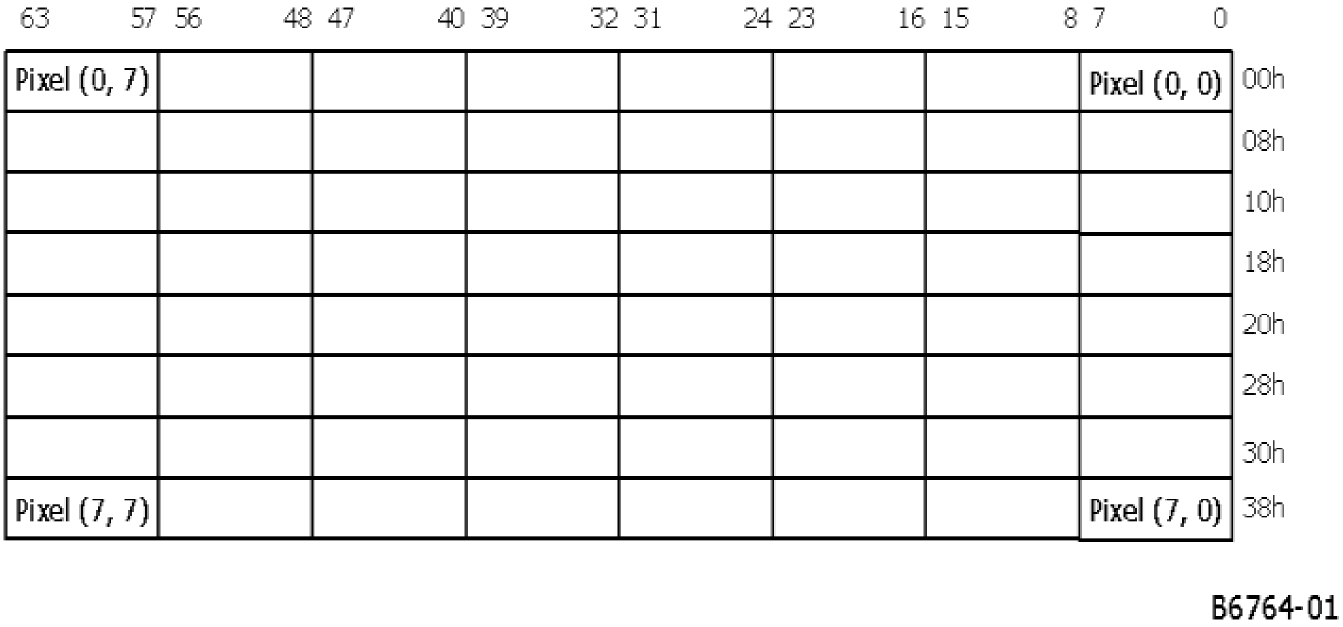
The maximum color pattern size is 256 bytes.

### 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.

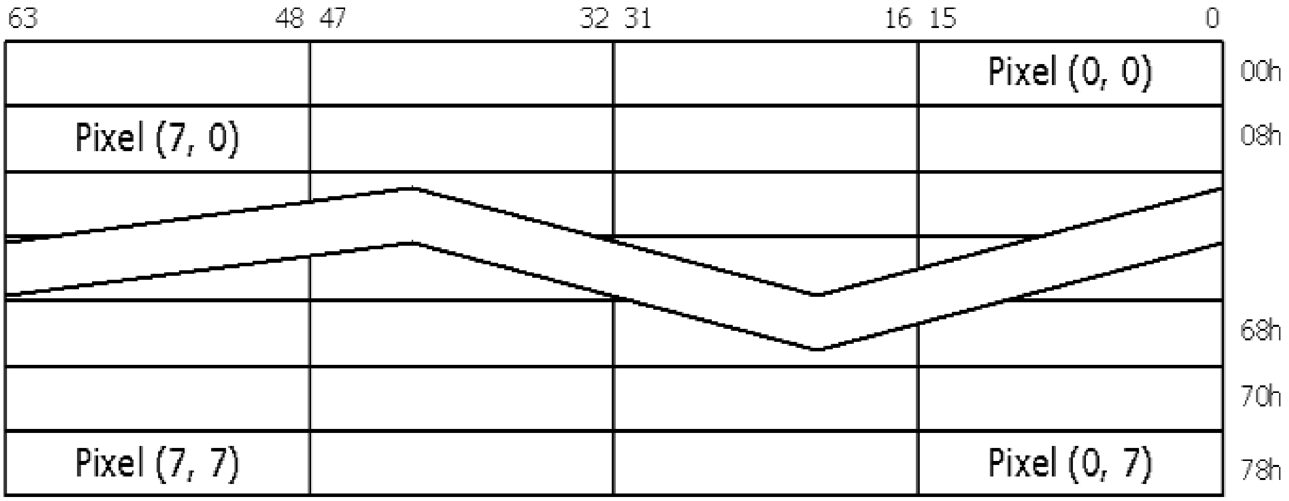
### 8bpp Pattern Data -- Occupies 64 Bytes (8 quadwords)



| 63 57 | 56 48 | 47 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 | |
|---|---|---|---|---|---|---|---|---|
| Pixel (0, 7) | | | | | | | Pixel (0, 0) | 00h |
| | | | | | | | | 08h |
| | | | | | | | | 10h |
| | | | | | | | | 18h |
| | | | | | | | | 20h |
| | | | | | | | | 28h |
| | | | | | | | | 30h |
| Pixel (7, 7) | | | | | | | Pixel (7, 0) | 38h |

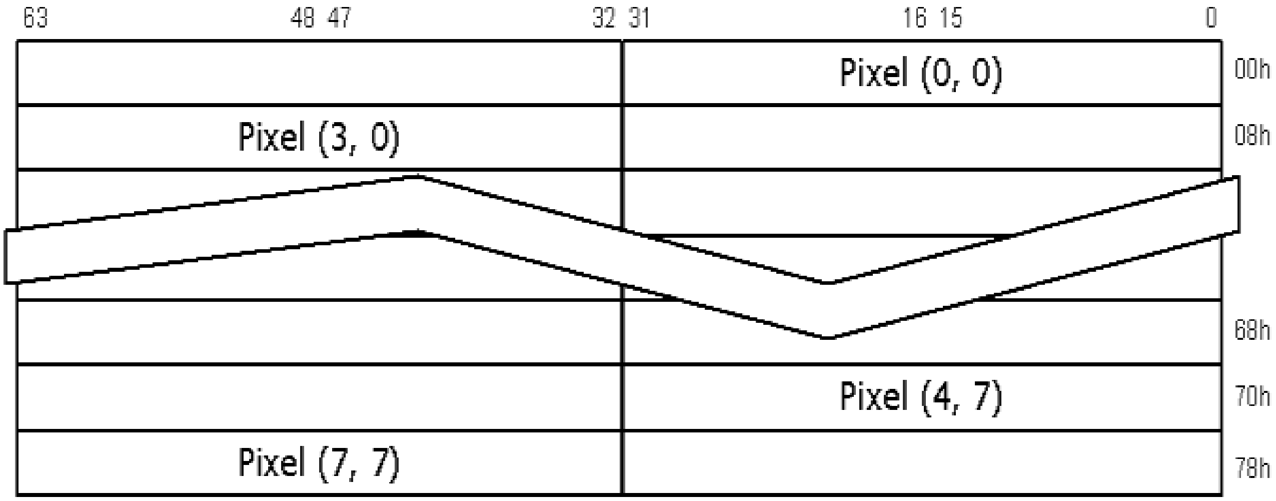
B6764-01

### 16bpp Pattern Data -- Occupies 128 Bytes (16 quadwords)



B6765-01

### 32bpp Pattern Data -- Occupies 256 Bytes (32 quadwords)



B6766-01

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.

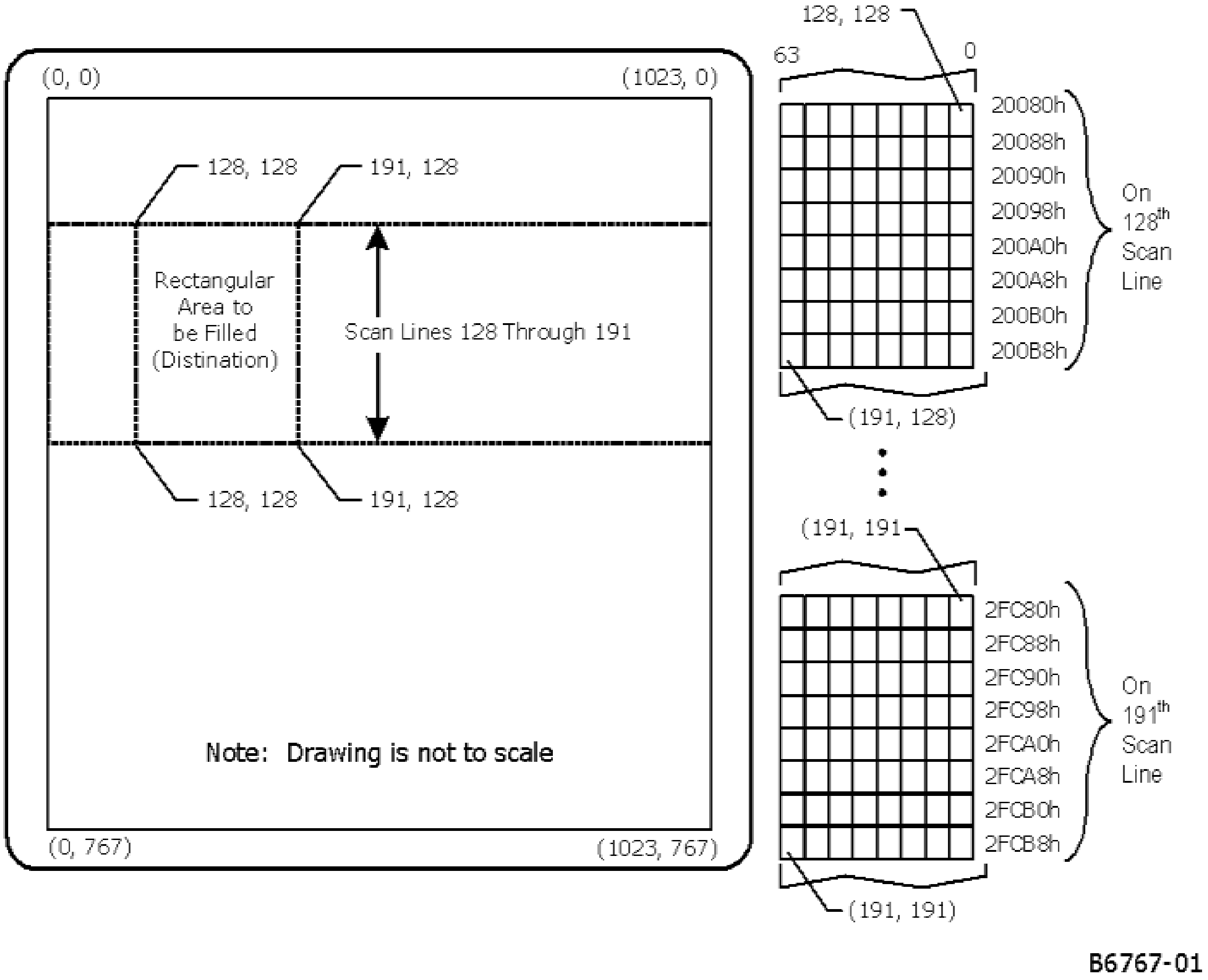
## Destination Data

## BLT Programming Examples

### Pattern Fill -- A Very Simple BLT

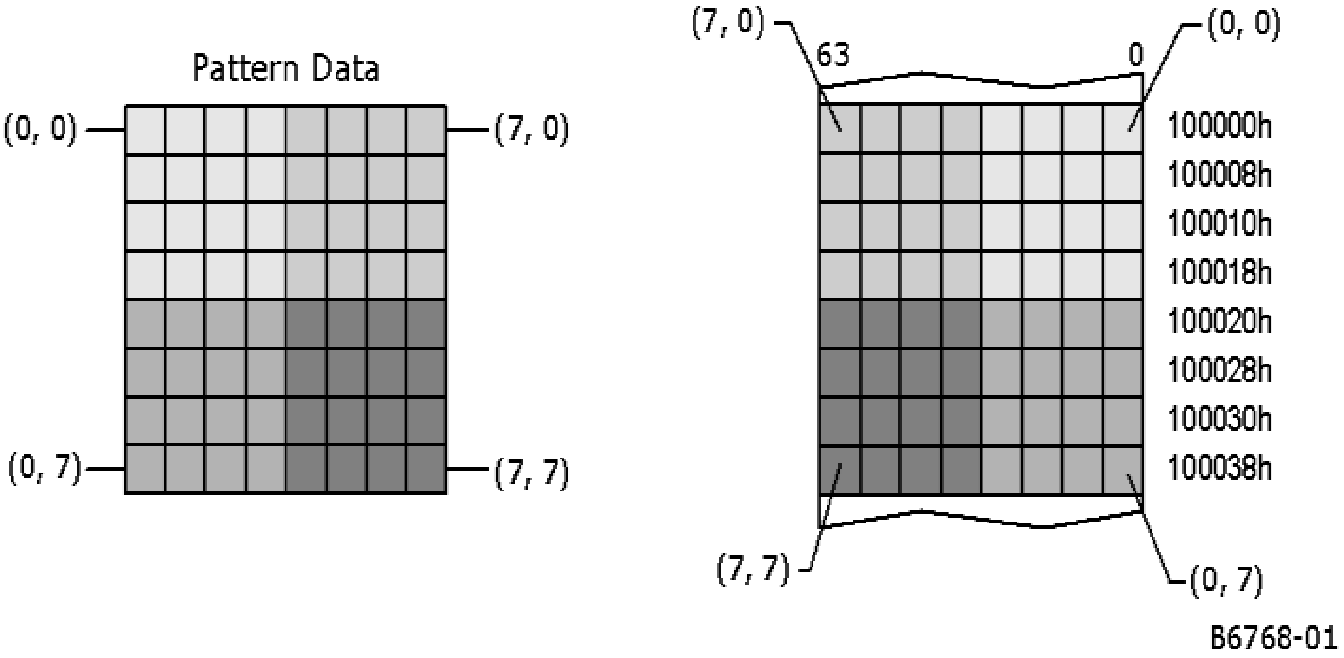
In this example, a rectangular area on the screen is to be filled with a color pattern stored as pattern data in off-screen memory. The screen has a resolution of 1024x768 and the graphics system has been set to a color depth of 8 bits per pixel.

### 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.

### 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.

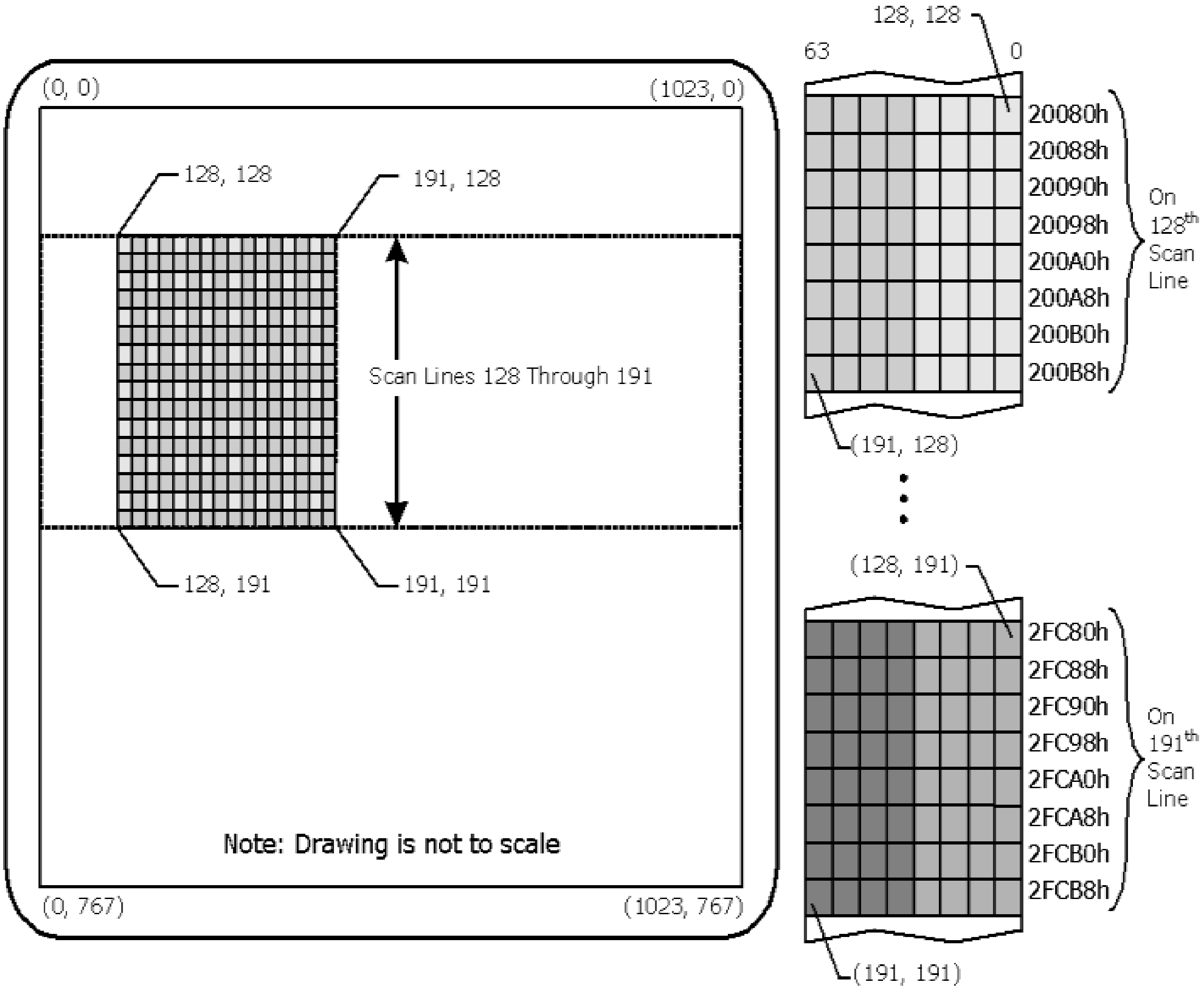
Bits [31:3] of the Pattern Address Register must be programmed with the address of the pattern data.

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.

### Results of Example Pattern Fill BLT



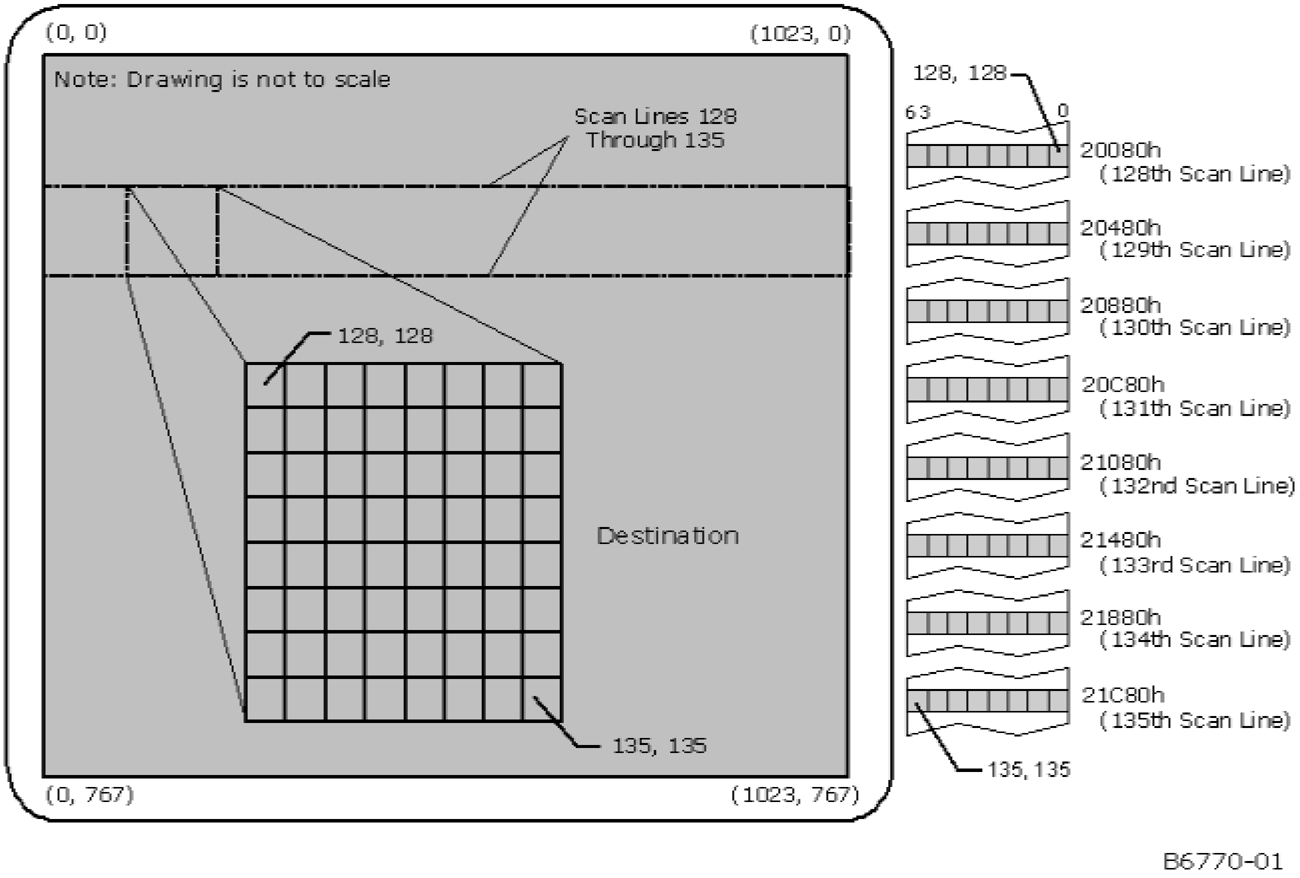
B6769-01

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.

## Drawing Characters Using a Font Stored in System Memory

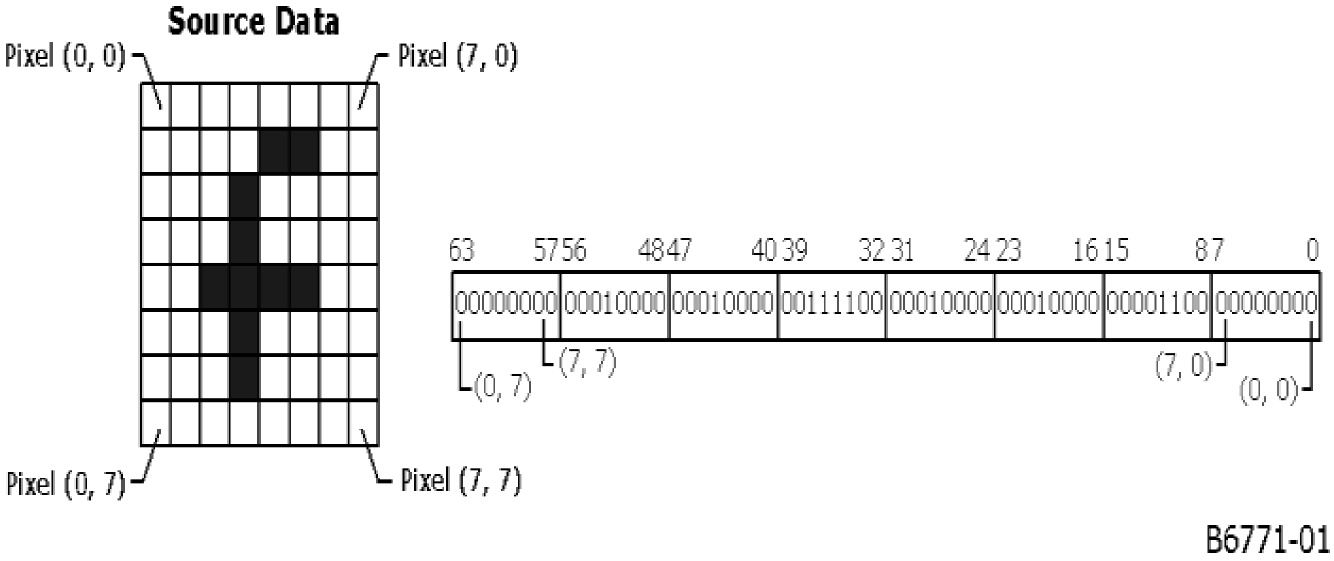
In this example BLT operation, a lowercase letter "f" is to be drawn in black on a display with a gray background. The resolution of the display is 1024x768, and the graphics system has been set to a color depth of 8 bits per pixel.

### 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).

### 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.

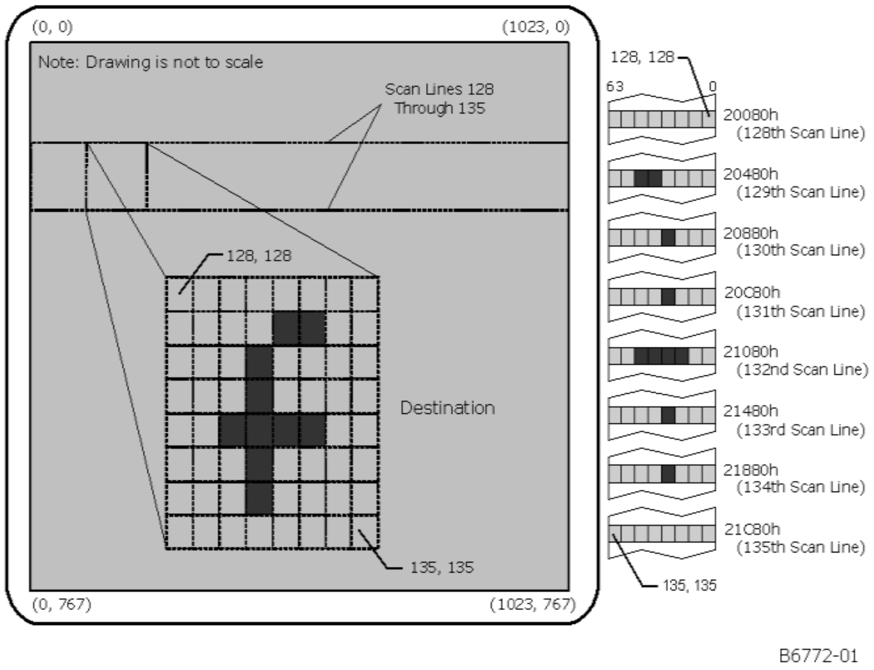
The Destination Pitch Register must be programmed with a value equal to the number of bytes in the interval between the first bytes of each adjacent scan line's worth of destination data. Since the color depth is 8 bits per pixel and the horizontal resolution of the display is 1024 pixels, the value 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.

### 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.

## Fast Copy Engine

Fast Copy Engine is capable of performing sub-resource copy or fill operation saturating the full memory bandwidth. It is a resource processor that can copy from a source of arbitrary format. Fast Copy Engine is capable of  understanding the concept of resource and sub-resources. It can support data transfer between overlapping surfaces. It can support state-full compression and decompression. Copy Engine supports both linear as well as tiled memory and operate on both system as well as local memory (when it is not a UMA system). In case of UMA system there is just a single memory.

Fast Copy Engine can be used for paging in or paging out resources to and from GPU. It also supports Fast Clear and regular clear operation of AUX enabled surfaces. It can be used to perform resolve operation on compressed surfaces. Copy Engines can operate on 2D surfaces defined using pixel coordinates as well as linear surfaces defined using virtual addresses.

Fast Copy Engine performance scales with number of memory slices in the system.

Copy workloads are submitted to the appropriate copy engines by the software through copy queues. BLT command streamers parses the commands and submits them to Fast Copy Engine. Copy Engines are split into two parts:

1. Copy Engine Front-end - splits an incoming command/BLIT into smaller commands/sub-blits
2. Copy Engine Back-end - it takes sub-blits and performs actual cachline read/write operations

Performance scaling of the Fast Copy Engine is achieved by increasing the number of copies of Back-end (Sub-copy Engine) attached to the single front-end of the fast copy engine. Fast copy engine can handle sub-blit level dependencies (producer consumer relationship). Software may have to insert appropriate flushes between commands to handle some dependencies not supported by copy engine hardware, as defined at a later section.

## BLT Instruction Overview

There are two blitter back-ends behind the single engine viz. Classical Blitter or Legacy Blitter Engine and Fast Copy Engine. Each of which operates on a separate mutually exclusive set of instructions. Commands for both the back-ends can be submitted to the same Blitter Command Streamer and based on the command it is passed on to the appropriate back end and executed there.

Each instruction consists of multiple dwords, where first dword is the header, which can identify the type of instruction, instruction opcode, length of instruction etc. Header may contain additional instruction fields as well. Fields may sometime need more than one dword.

Most of the instructions can operate independently without state other than some Legacy Blitter commands involving clip rectangles that programs some BRs (Blitter Registers) before executing those commands.

There are specific MMIO based config registers that configures how Legacy Blitter or Fast Copy Engine is expected to operate.

Legacy Blitter uses cache, so unless a flush is inserted after an instruction there is no guarantee of final data reaching the memory.

The flush is common for both the backends. Once a flush is sent command streamer stops sending further commands to the back-ends and back-ends ensure that when the current commands being processed by the back-ends are completed flush done are sent to blitter command streamer.

The actual commands have been listed out in a later section.

## 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, XY_SETUP_CLIP_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 |

The definition of each of the above registers which are part of logical context and are saved and restored during context save and restore are found in "BLT Engine Instruction Field Definitions" section.

## Device Cache Coherency: Render & 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.

## Copy Engine Fast Clear Support

Copy Engine can be used to perform Fast Clear operation on a destination surface provided the surface is mapped to CCS. A CCS cacheline is mapped to 4x4KB of main surface. The surfaces must begin on 16KB virtual address boundaries, and these surfaces must be padded to 16KB granularity. Fast Clear operation using copy engine can't be performed when BPP is 96. Fast Clear rectangles must be 128Bx1 aligned for linear and 32Bx4 aligned for tiled surfaces. Width of the Fast Clear rectangle must be integral multiple of 128 for linear and 32 for tiled surfaces. Height of the Fast Clear rectangle for tiled surfaces must be multiple of 4. Surfaces smaller than 128B cannot be Fast Cleared by Copy Engine. Sometimes Fast Clear rectangles may have to be padded to get the right size and shape for Fast Clear to work.

If software wants to use the Fast Clear capability of the copy engine it can do so by setting up the top left and bottom right coordinate of the surface in the XY_FAST_COLOR_BLT command and setting special operation mode to the kind of Fast Clear the software intends to accomplish. Appropriately aligned blocks in any sub-resource can be fast cleared. Mip-tails can't be fast cleared unless they are 128B aligned and not smaller than the minimum Fast Clearable surface. If the resource type is 3D and tiling selected is Tile64, fast clear works only if the CCS update granularity is 4 bit.

In case of irregular surfaces only the appropriately aligned part of the surface can be Fast Cleared using Copy Engine Fast Clear. Rest of the parts of the surface can be cleared using regular clear operation.

Copy Engine Hardware performs Fast Clear operation by updating the CCS cacheline with appropriate data. It can either initialize the CCS cache with all zero or all one data.

Fast Clear is not supported when it is used over Xelink.

## Verbatim Copy operation

Verbatim Copy is the special copy operation defined for compressed surface copy where both the main surface data and the associated CCS data are moved as it is from one location to another. During this operation main surface data copy and CCS data copy are performed sequentially. Driver needs to use two separate commands, XY_FAST_COPY_BLT to copy the main surface data and XY_CTRL_SURF_COPY_BLT to copy the CCS surface, together for verbatim copy. In case of regular compressed data access from Copy Engine, read data from source is always made available to copy engine as fully decompressed data and Copy Engine writes fully decompressed data back to memory and if destination compression is required this data is again compressed and written to memory. The compressed write causes both CCS and main surface write data to be modified. In case of verbatim copy "data read" operation from source the data does not decompress before sending to Copy Engine. Driver needs to set "Enable Compressed Surface Read" bit in the XY_FAST_COPY_BLT command to enable this special operation. In case of verbatim copy "data write" operation to destination surface this "raw" compressed data is sent out and this flow can be enabled by driver by setting the "Enable Compressed Surface Write" bit in the XY_FAST_COPY_BLT command.

These two commands must be used in a pair and the sequence in which the two commands must be used during verbatim copy operation varies based on whether it is being used for swap-in (moving data from local memory to system memory) or swap-out (moving data from system memory to local memory) operation for compressed data. In case of swap-out operation the sequence required is XY_FAST_COPY_BLT followed by XY_CTRL_SURF_COPY_BLT. In case of swap-in operation the sequence required is XY_CTRL_SURF_COPY_BLT followed by XY_FAST_COPY_BLT.

During verbatim copy the copy rectangles used for XY_CTRL_SURF_COPY_BLT and XY_FAST_COPY_BLT must be same. They must also use same BPP and pitch. Using base address X1, X2, Y1, Y2 from XY_FAST_COPY_BLT command start address and range is calculated.

## 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 or read through a memory mapped location; they are internal state only.

## BLT Programming Restrictions

**Overlapping Source/Destination BLTs:**

For all products *negative pitch* programming is allowed only when the source and destination surfaces are of the same type: linear source to linear destination copy, or tiled source to tiled destination copy. This is a must requirement. In such cases:

- Both the pitches must be programmed to be a negative value, if the source and destination surfaces are overlapping.
- Either of the pitches can be programmed to a negative value if required to do mirroring, but only if the complete source and destination surfaces are not overlapping anywhere on the surfaces.

| Description |
|---|
| For the XY_FAST_COPY_BLT and XY_BLOCK_COPY_BLT instruction, these restrictions apply:<br>When two sequential fast copy blits have different source surfaces, but their destinations refer to the same destination surface and therefore destinations overlap, a Flush must be inserted between the two blits.<br>For two sequential fast copy blits when the source of the second blit is the destination of the first blit or they overlap a Flush must be inserted between the two blits. |
| The **pitch** length for **Linear Surfaces** is OWord-aligned (16-byte multiple), as the BSpec says.<br>(For Tiled surfaces, the pitch length is always Cacheline aligned (64-byte multiple), as Tile surface pitches must be a multiple of Tile widths, which are always cacheline aligned).<br>For XY_FAST_COPY_BLT command (X1, X2) and (Y1, Y2) must be programmed different values so that start pixel/end pixel and start line/end line do not coincide. |

**Legacy Blits:**

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.

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.

The other way to do this is driver should always program a dummy 3D.

**NON-PIPELINE state following the BLT commands:**

For Monosource and Color Pattern surfaces, and also linear color source and destination surfaces, the start **Base Address** programmed should always be Cache Line (64 byte) aligned.

| Programming Note | |
|---|---|
| **Context:** | Fast Copy Blitter + Frame Buffer Compression |
| Fast Copy, Block Copy and Fast Color workloads targeting the Front Buffer with FBC enabled is not supported. | |

| Programming Note | |
|---|---|
| **Context:** | XY_FAST_COPY_BLT command programming restriction |
| XY_FAST_COPY_BLT command's (X1, Y1), (X2, Y2) must be programmed such way so that neither W nor H of the surface to copy are "0". | |

| Programming Note | |
|---|---|
| **Context:** | Blitter |
| Support for Tile-Yf format has been removed, so Blitter should not be programmed to support Tile-Yf for source or destination tiling. | |

| Programming Note | |
|---|---|
| **Context:** | XY_BLOCK_COPY_BLT and XY_FAST_COLOR_BLT commands |
| MSAA compression is not supported by Copy Engine, so it is illegal to program "Number of Multisamples" in the command to any value greater than 1. | |

| Programming Note | |
|---|---|
| **Context:** | XY_BLOCK_COPY_BLT usage restriction for resolve operation |
| When XY_BLOCK_COPY_BLT command is used to resolve a compressed surface, it must be ensured that the surfaces are compression block aligned. If the surfaces to be resolved are split into multiple BLITs for any reason it must be ensured that they are split based on CCS cacheline alignment. For 128B compression block size this alignment means the BLITs must be 16384 bytes aligned. | |

| Programming Note | |
|---|---|
| **Context:** | XY_CTRL_SURF_COPY_BLT command |
| If XY_CTRL_SURF_COPY_BLT command is either preceded by or succeeded by XY_FAST_COPY_BLT, XY_BLOCK_COPY_BLT or XY_FAST_COLOR_BLT command and there is address overlap between the commands software must insert explicit flush between them as Copy Engine cannot track address overlap between Control Surface Copy command and other copy commands.<br>Copy Engine can't detect address overlap between successive XY_CTRL_SURF_COPY_BLT commands and hence can't sequence them when producer consumer relationship is involved, so software must insert explicit flush between such commands. | |

| Programming Note | |
|---|---|
| **Context:** | Blitter (XY_BLOCK_COPY_BLT, XY_FAST_COLOR_BLT commands) |

X offset, Y offset and (X1, Y1), (X2, Y2) values must be programmed so that X coordinate values when added to X offset remains within the surface boundary (16384 pixels). Similarly

Y coordinate values when added to Y offset must remain within the surface boundary (16384 lines).

Copy operation between two overlapping surfaces are supported only when the base address and pitch of both the source and destination surfaces are identical, irrespective of the surface formats.

| Programming Note | |
|---|---|
| **Context:** | Programming Restrictions for XY_BLOCK_COPY_BLT and XY_FAST_COLOR_BLT commands for when compression is enabled. |

*Note: Whenever source or destination of a copy operation is System Memory and data is compressed, the copy operation needs to be verbatim copy, where main surface and the associated metadata (AUX) must be copied separately as it is to their respective location treating them as uncompressed data.

| 3D Copy operations | | | Destination | | | |
|---|---|---|---|---|---|---|
| | | | System Memory | | Local Memory | |
| | | | Compressed | Decompressed | Compressed | Decompressed |
| Source | System Memory | Compressed | Valid* | Invalid | Valid* | Invalid |
| | | Decompressed | Invalid | Valid | Valid | Valid |
| | Local Memory | Compressed | Valid* | Valid | Valid | Valid |
| | | Decompressed | Invalid | Valid | Valid | Valid |
| Media Copy operations | | | Destination | | | |
| | | | System Memory | | Local Memory | |
| | | | Compressed | Decompressed | Compressed | Decompressed |
| Source | System Memory | Compressed | Valid* | Invalid | Valid* | Invalid |
| | | Decompressed | Invalid | Valid | Invalid | Valid |
| | Local Memory | Compressed | Valid* | Valid | Valid | Valid |
| | | Decompressed | Invalid | Valid | Invalid | Valid |

## 2D (XY) 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.

*Source and destination pitches have the additional explanation given next. The below statements are applicable for pitch field in all of the Blit commands:*

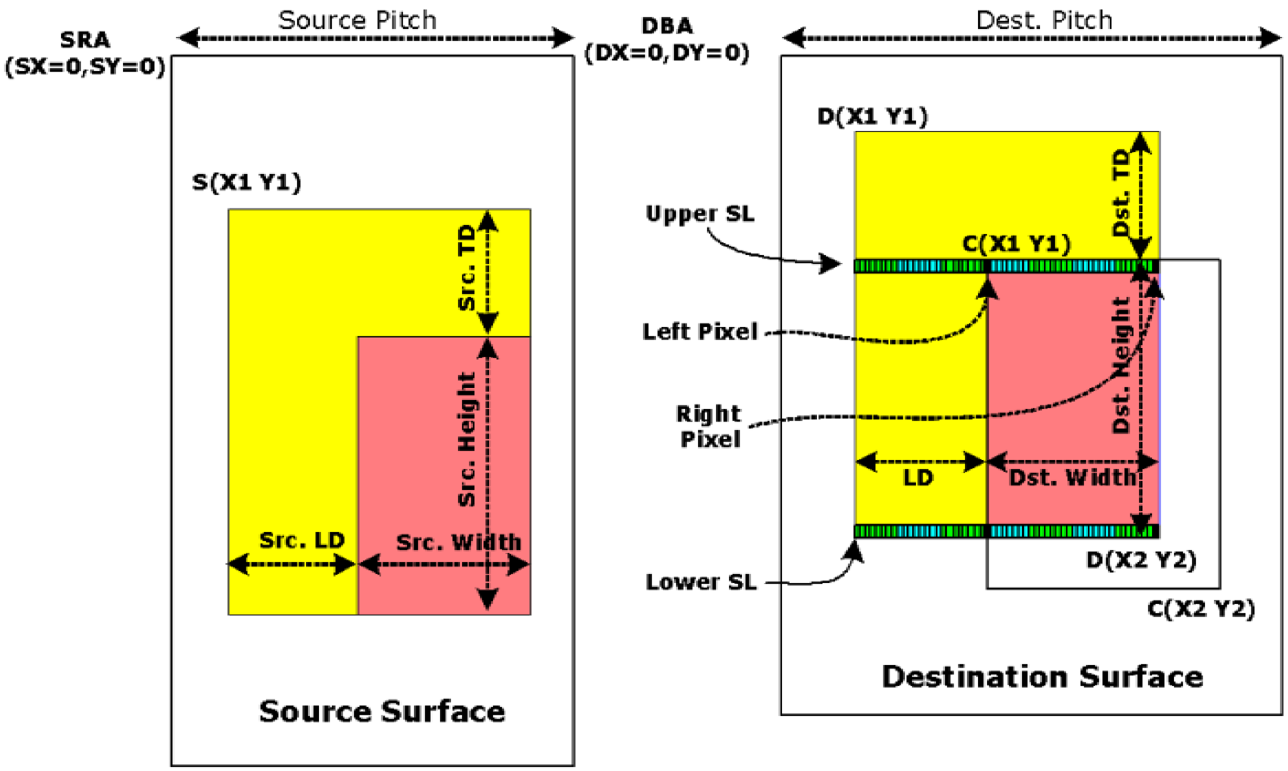
1. For Linear surfaces, the pitch is programmed in bytes. For Tiled surfaces the pitch programmed is in Dwords count.
2. For Tiled surfaces this pitch is of 512Byte granularity for Tile-X: This means the tiled-x surface pitch can be (512, 1024, 1536, 2048...)/4 (in Dwords).
3. For Tiled surfaces this pitch is of 128B granularity for Tile-Y/Yf and of 512B granularity for Tile-Ys: This means the tiled-y surface pitch can be (128, 256, 384, 512...)/4 (in Dwords).
4. Another way to indicate this is, for tiled surfaces, the pitch is programmed in Dwords and is an integral multiple of the tile width.

### BLT Instructions

| Instructions |
|---|
| XY_BLOCK_COPY_BLT |
| XY_FAST_COPY_BLT |
| XY_FAST_COLOR_BLT |
| XY_CTRL_SURF_COPY_BLT |
| XY_COLOR_BLT |
| XY_FULL_BLT |
| XY_FULL_IMMEDIATE_PATTERN_BLT |
| XY_FULL_MONO_PATTERN_BLT |
| XY_FULL_MONO_PATTERN_MONO_SRC_BLT |
| XY_FULL_MONO_SRC_BLT |
| XY_FULL_MONO_SRC_IMMEDIATE_PATTERN_BLT |
| XY_MONO_PAT_BLT |
| XY_MONO_PAT_FIXED_BLT |
| XY_MONO_SRC_COPY_BLT |
| XY_MONO_SRC_COPY_IMMEDIATE_BLT |
| XY_PAT_BLT |

| Instructions |
|---|
| XY_PAT_BLT_IMMEDIATE |
| XY_PAT_CHROMA_BLT |
| XY_PAT_CHROMA_BLT_IMMEDIATE |
| XY_PIXEL_BLT |
| XY_SCANLINES_BLT |
| XY_SETUP_BLT |
| XY_SETUP_CLIP_BLT |
| XY_SETUP_MONO_PATTERN_SL_BLT |
| XY_SRC_COPY_BLT |
| XY_SRC_COPY_CHROMA_BLT |
| XY_TEXT_BLT |
| XY_TEXT_IMMEDIATE_BLT |

## Some Equalities & Inequalities for Source Clipping



**Some Equalities & Inequalities for Source Clipping:**

Src. TD = Dst. TD (Top discard in SL)
Src LD = LD (Left Discard in Pixels)
Src Height = Dst. Height in SL
Src Width = Dst. Width in Pixels

**Note:** Src. Pitch is not equal to Dst. Pitch

B6773-01

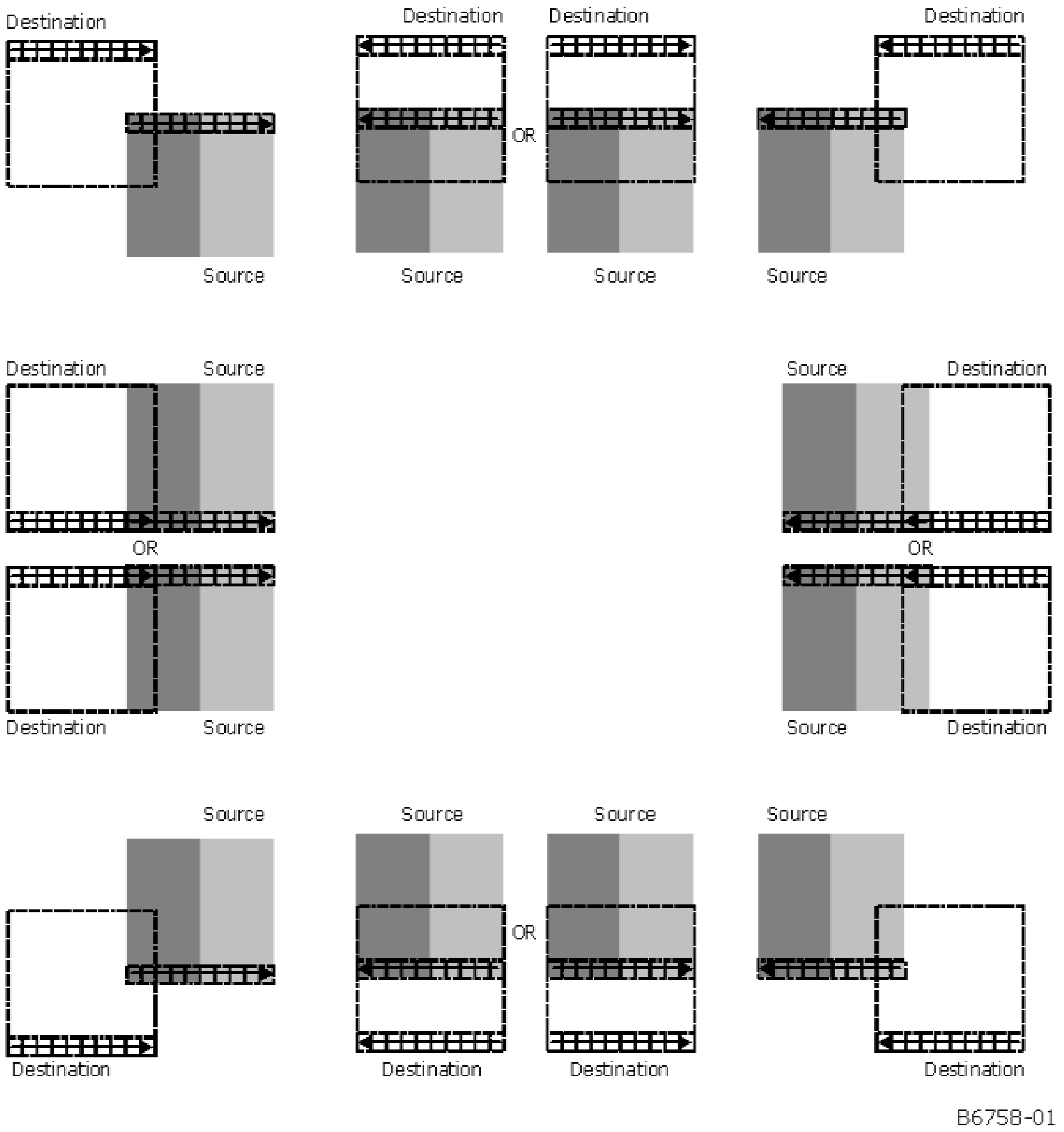
DX1, DY1, CX1, and CY1 are inclusive, while DX2, DY2, CX2, and CY2 are exclusive.

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 this 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.


Destination
Source
Destination
Destination
OR
Source
Source
Destination
Source
Destination
Source
Source
Destination
OR
Destination
Source
OR
Source
Destination
Source
Source
Source
Source
Destination
OR
Destination
Destination
Destination
B6758-01

## 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.

| Register |
|---|
| BR00 - BLT Opcode and Control |
| BR01 - Setup BLT Raster OP, Control, and Destination Offset |
| BR05 - Setup Expansion Background Color |
| BR06 - Setup Expansion Foreground Color |
| BR07 - Setup Blit Color Pattern Address Lower Order Address bits |
| BR30 - Setup Blit Color Pattern Address Higher Order Address |
| BR09 - Destination Address Lower Order Address Bits |
| BR27 - Destination Address Higher Order Address |
| BR11 - BLT Source Pitch (Offset) |
| BR12 - Source Address Lower order Address bits |
| BR28 - Source Address Higher order Address |
| BR13 - BLT Raster OP, Control, and Destination Pitch |
| BR14 - Destination Width and Height |
| BR15 - Color Pattern Address Lower order Address bits |
| BR29 - Color Pattern Address Higher order Address |
| BR16 - Pattern Expansion Background and Solid Pattern Color |
| BR17 - Pattern Expansion Foreground Color |
| BR18 - Source Expansion Background and Destination Color |
| BR19 - Source Expansion Foreground Color |