



LEG

Noah Huppert and Robert Scibelli

# Chapter 1

## Management Plan

### Responsibilities

Team members will act as generalists, working on all parts of the instruction set implementation.

This will give team members an opportunity to learn about writing simulator code, developing user interfaces, and creating tooling.

### Code Management

Git will be used to track changes to source code.

Code will be synchronized through GitHub repositories which are owned by a shared GitHub organization.

The master branch of repositories will be stable at all times. Features will be developed in separate branches and merged in via GitHub pull requests. Pull requests will be reviewed by the other team member to ensure correctness and keep everyone up to date on changes.

### Work Coordination

The team will use GitHub issues to track and assign work.

Every feature or bug will have its own issue which contains a full description of the work required to complete or fix the feature or bug.

When a team member starts working on a feature or bug they will assign themselves to the associated GitHub issue to ensure duplicate work does not occur.

### Testing

Every repository will use GitHub actions (a continuous integration solution) to run integration tests on every commit. Pull requests will be required to pass these tests before they can be merged into the stable master branch.

If new features are added corresponding tests must be added to ensure the correctness of the new features.

If bugs are fixed regression tests must be added to ensure the bug is fixed and does not occur again.

# Chapter 2

## Memory

**Endianness:** Little

**Memory Organization:** Harvard

Memory is broken in to 2 parts:

- Data and instructions
- Graphics

### Data and Instruction Memory

**Word Size:** 32 bits

**Addressing Unit:** Word

**Address Space:**  $2^{32}$

Holds data and instructions. Can be manipulated directly via the load and store instructions.

**Memory Hierarchy:**

1. Level 1, SRAM: 64 KB, 4-way associative (1 cycle delay)
2. Level 2, SRAM: 256 KB, direct mapped (10 cycle delay)
3. Level 3, SRAM: 8 MB, direct mapped (40 cycle delay)
4. DRAM:  $2^{32} \cdot 32\text{bits} \simeq 17\text{GB}$  (100 cycle delay)

### Graphics Memory

The graphics memory is composed of SRAM.

There are two main pieces of graphics memory which are interacted with exclusively through custom instructions:

- Frame buffer
- Sprite library

All graphics memory shares these underlying properties:

**Word Size:** 8 bits

**Addressing Unit:** Word

Pixel organization in memory:

8 bit pixels: 3 red, 3 green, 2 blue.

The first pixel defines the top left pixel in a display, the last pixel the bottom right pixel in a display.

The width of the thing being displayed determines how many bytes there are per line.

## **Frame Buffer**

**Address Space:**  $2^{16}$

Holds pixels to be displayed to the user on a screen.

Screen size is 256 x 256 pixels.

Can only be manipulated by graphics instructions.

**Memory Hierarchy:**

1. SRAM: 64 KB (4 cycle delay)

## **Sprite Library**

A piece of memory designed to hold sprites so they can be operated on in a quick fashion.

**Address Space:**  $2^{12}$

This memory can hold up to 1 128x64 or 64x128 sprite, or several smaller sprites.

**Memory Hierarchy:**

1. SRAM: 8 KB (11 cycle delay)

## Chapter 3

# Sprite Processing Unit (Blitter)

The architecture includes a specialized sprite processing unit.

This unit is inspired by the Atari ST BLITTER chip.

Its purpose is to quickly transfer bits from the sprite library memory to the frame buffer, while performing basic logic operations along the way.

In the documentation this will be referred to as the “Blitter”.

## Blitter Registers

The Blitter has 4 registers which it uses internally to determine how to transfer data.

- Source: 12-bit address in sprite library memory marking the start of a sprite
- Destination: 16-bit address in frame buffer memory to start copying sprite to
- Width: 7-bit width of sprite
- Height: 7-bit width of sprite

These registers cannot be read by any component other than the Blitter.

They can only be set via graphics instructions.

## Blitter Operations

The Blitter can perform many different logical operations on the source against the destination and store the result:

Binary	Operation
0000	All zeros
0001	SRC & DEST
0010	SRC & ~DEST
0011	SRC
0100	~SRC & DEST
0101	DEST
0110	SRC ^ DEST
0111	SRC   DEST
1000	~SRC & ~DEST
1001	~SRC ^ DEST

Binary	Operation
1010	$\sim$ DEST
1011	SRC   $\sim$ DEST
1100	$\sim$ SRC
1101	$\sim$ SRC   DEST
1110	$\sim$ SRC   $\sim$ DEST
1111	All ones

(These are the same operations the Atari Blitter supported)

# Chapter 4

## Registers

Referred to in assembly as **R#** where # is a number.  
Some registers have aliases.

### General Purpose

27 mixed 32-bit registers.

R0 through R25.

Initially all set to 0.

### Interrupt Link Register

32-bit register.

R26, INTLR.

Store the address to return to after an interrupt has completed.  
Initially set to 0.

### Interrupt Handler

32-bit register.

R27 or IHDLR.

Stores the memory address of the interrupt handler subroutine.  
Initially set to all 1's to indicate it has not been set.

See the Interrupts section for details.

### Program Counter

32-bit register.

R28, PC.

Stores the address of the current instruction being executed.  
Initially set to 0.

## Status

6-bit register.

R29, STS.

Initially set to null status with the interrupt flag unset.

See the Status Codes section for details.

## Stack Pointer

32-bit register.

R30, SP.

Stores the address of the bottom of the stack in memory. Initially set to 0.

## Link Register

32-bit register.

R31, LR.

Store the address to return to after a subroutine has completed.  
Initially set to 0.



# Chapter 5

## Status Codes

The status register is 6-bits large.

The least significant 5 bits are used to store the status of comparisons and arithmetic operations, called condition codes.

The most significant bit is an interrupt flag.

### Condition Codes

The special condition code 00000 is used to denote a null status **NS**.  
This code will match every other condition code.

Valid codes are:

Binary	Assembly	Meaning
00000	NS	Null status, matches everything
00001	NE	Not equal
00010	E	Equal
00011	GT	Greater than
00100	LT	Less than
00101	GTE	Greater than or equal to
00111	LTE	Less than or equal to
01000	OF	Overflow
01001	Z	Zero
01010	NZ	Not zero
01011	NEG	Negative
01100	POS	Positive

### Interrupt Flag

The interrupt flag signifies if an interrupt is currently being handled.  
A value of 0 means no interrupts are being handled and vise versa.

If the interrupt flag is set no new interrupts can be handled. Any new interrupts which come in will be ignored.

## Chapter 6

# Interrupts

The interrupt handler register holds the memory address for a subroutine which will handle an interrupt. Initially this register is set to all 1's, which means the handler is unset.

The simulator will internally trigger interrupts by doing the following:

- Check if the interrupt flag in the status register is set, if so do nothing
- If the interrupt handler register is all 1's the interrupt handler is not set, do nothing
- Set the interrupt flag in the status register
- Store the interrupt code in memory at address 1111 1111 1111 1111 1111 1111 1111 1111
- Set the interrupt link register to the program counter
- Set the program counter to the value of the interrupt handler register

After the interrupt handler is done it must call perform a return from interrupt Jump

The interrupt code identifies the cause of the interrupt.

When an interrupt occurs it will be loaded into main memory at address 1111 1111 1111 1111 1111 1111 1111 1111.

Interrupts codes can have the following values:

Binary	Assembly	Meaning
0000	UPARROW	Up arrow key
0001	DOWNARROW	Down arrow key
0010	LEFTARROW	Left arrow key
0011	RIGHTARROW	Right arrow key
0100	ENTER	Enter key
0101	ESCAPE	Escape key
0110	SPACE	Space key

## Chapter 7

# Binary Format

Once programs are assembled they will be in the LEG binary format.

The format is simple:

A little endian binary file. Every 32 bits will be interpreted as a memory address. Memory addresses will increment by one. The contents will be loaded directly into memory.

Program execution will begin at memory address 0. The program should be terminated by a Halt instruction.

## Chapter 8

# Assembly Format

One instruction per line. The first token in a line is a label. If no label is desired put a blank space or tab. The following tokens may be the mnemonics documented in the instructions section.

Immediate use the following syntax:

- `0dNN`: For unsigned decimal immediate values
- `0xNN`: For unsigned hexadecimal immediate values
- `0bNN`: For unsigned binary immediate values
- `0sdNN`: For signed decimal immediate values
- `0sxNN`: For signed hexadecimal immediate values
- `0sbNN`: For signed binary immediate values

The assembler will convert all immediate values to binary.

Any tokens after the `#` character will not be interpreted, use this for comments.

# Chapter 9

## Instructions

### Assembly Documentation Syntax

Instruction assembly is documented using the following syntax:

- A word in curly brackets signifies a variation of an instruction’s mnemonic. A table will be present which specifies valid values, the curly brackets and their contents should be replaced with one of these values.

- Example:

DO{OPERATION}

{OPERATION} indicates that part of the mnemonic must be replaced by a value from the {OPERATION} table below.

{OPERATION}	Operation
F	Foo
B	Bar

For example a mnemonic of DOF indicates that the “Foo” operation should take place.

- A word in angle brackets signifies an instruction operand. Look for the “operands” section of the instruction documentation for more detail.
- Example:

DO <DEST> <OP1> <OP2>

In the above <DEST>, <OP1>, and <OP2> are all operands which should be replaced by operand values when writing assembly.

For example the assembly line DO R1 R2 R3 has a <DEST> operand value of R1, a <OP1> operand value of R2, and a <OP2> operand value of R3.

### Bit Organization Syntax

Instructions have a bit organization section which details the binary format of the instruction itself.

The format of this section is a table, where the top header row indicates the purpose of the bits, and the box directly beneath each item in the header row indicates how many bits are reserved for the described purpose.

The leftmost part of the table represents the least significant bits, and the rightmost part of the table represents the most significant bits.

Example:

Type	Operation	<DEST>	<OP1>	<OP2>
4	6	4	8	2

Indicates that the “Type” field takes up 4 bits, the “Operation” field takes up 6 bits, the <DEST> field takes up 4 bits, and so on.

The binary number:

```

MSB                               LSB
|                               |
v                               v
101011001011110100011001

```

Would translate to the following values for the fields defined in the example:

Field	Value
Type	1001
Operation	010001
<DEST>	1111
<OP1>	10110010
<OP2>	10

## Optional Parameters

Sometimes instructions have optional parameters. These parameters are specified by putting them inside square brackets. If this is the case the behavior if the parameter is not provided is documented below.

Example:

The instruction documentation:

`FOO[{TYPE}] <OP1>`

Indicates that the {TYPE} parameter is optional. However the <OP1> operand is still required.

## Data Types

There are 2 data types which are supported in instructions:

- 32 bit two’s complement integer
- 32 bit unsigned integer

Instructions have type variations when it matters, bit level operations do not.

## Condition Fields

All instructions currently have space for a condition field.

This field allows for predicated execution of instructions.

Currently only the jump instructions use this condition field.

All other instructions do not use this field at the moment, in the future they may. For right now the condition field will be set to null status.

## Immediate Fields

Most immediate fields, with the exception of those in graphics instruction, are sign extended to 32-bits.

Graphics instructions do not extend their immediate fields because they deal with multiple different address spaces. The size of these immediate fields has been carefully specified. See the Graphics Memory section for more details.

## Instruction Types

There are 3 instruction types:

Type	Field Binary	Type
00		Control
01		ALU
10		Memory
11		Graphics

## Arithmetic Logic Unit

### Instructions:

Typed arithmetic instructions:

- Add
- Subtract
- Multiply

Typed general instructions:

- Compare
- Shift
- Arithmetic Right
- Arithmetic Left

Untyped general instructions:

- Shift
- Logical Right
- Logical Left
- 3 operand logic
- And

- Or
- Xor
- 2 operand logic
- Not
- Move

### Bit Organization:

The operation field of each ALU instruction has the following meaning:

Binary	Operation
000001	Add unsigned integer register direct
000010	Add signed integer register direct
000011	Add unsigned integer immediate
000100	Add signed integer immediate
000101	Subtract unsigned integer register direct
000110	Subtract signed integer register direct
000111	Subtract unsigned integer immediate
001000	Subtract signed integer immediate
001001	Multiply unsigned integer register direct
001010	Multiply signed integer register direct
001011	Multiply unsigned integer immediate
001100	Multiply signed integer immediate
-	-
001101	Move
-	-
001110	Compare
-	-
001111	Arithmetic shift left register direct
010000	Arithmetic shift right register direct
010001	Arithmetic shift left immediate
010010	Arithmetic shift right immediate
-	-
010011	Logical shift left register direct
010100	Logical shift left immediate
010101	Logical shift right register direct
010110	Logical shift right immediate
-	-
010111	And register direct
011000	And immediate
011001	Or register direct
011010	Or immediate
011011	Xor register direct
011100	Xor immediate
-	-
011101	Not

## Arithmetic Instructions

Assembly:

{OPERATION}-{TYPE} <DEST> <OP1> <OP2>

3 operations \* 2 types \* 2 addressing modes = 12 total instructions.



**Bit Organization:**

Register direct:

Condition	Type	Operation	<DEST>	<OP1>	<OP2>	Not Used
5	2	6	5	5	5	4

Immediate:

Condition	Type	Operation	<DEST>	<OP1>	<OP2>
5	2	6	5	5	9

**Behavior:**

Performs a basic arithmetic operation, determine by {OPERATION}:

{OPERATION}	Behavior
ADD	<OP1> + <OP2>
SUB	<OP1> - <OP2>
MLT	<OP1> * <OP2>

The type of numbers used in the arithmetic operation is specified by appending {TYPE}:

{TYPE}	Type
U	Unsigned integer
S	Signed integer

{TYPE} defaults to unsigned integer mode.

**Operands:**

- <DEST>: Register to store result
- <OP1>: Register containing first number
- <OP2>: Register containing second number or a 9-bit immediate value

**Move****Assembly:**

MV &lt;DEST&gt; &lt;SRC&gt;

1 total instruction.

**Bit Organization:**

Condition	Type	Operation	<DEST>	<SRC>	Not Used
5	2	6	5	5	9

**Behavior:**

Transfers the contents of the <SRC> register to the <DEST> register.

**Operands:**

- <DEST>: The destination register
- <SRC>: The source register

## Compare

**Assembly:**

CMP[{TYPE}] <OP1> <OP2>

1 instruction.

**Bit Organization:**

Condition	Type	Operation	<OP1>	<OP2>	Not Used
5	2	6	5	5	9

**Behavior:**

Compares <OP1> to <OP2> and stores the result in the status register.

**Operands:**

- <OP1>: Register containing first number to compare, on the left hand side of the comparison
- <OP2>: Register containing number to compare to <OP1>, on the right hand side of the comparison

## Arithmetic Shift

**Assembly:**

AS{DIRECTION} <DEST> <OP1>

2 directions \* 2 addressing modes: 4 total instructions.

**Bit Organization:**

<OP1> register direct:

Condition	Type	Operation	<DEST>	<OP1>	Not Used
5	2	6	5	5	9

<OP1> immediate:

Condition	Type	Operation	<DEST>	<OP1>
5	2	6	5	14

**Behavior:**

Performs an arithmetic shift (respects the sign of the number) on <DEST> and stores the result in <DEST>. Shifted by the amount specified in <OP1>.

<OP1> can either be an immediate value or a register.

The direction bits are shifted is specified by {DIRECTION}:

{DIRECTION}	Direction
L	Left
R	Right

#### Operands:

- <DEST>: Destination register
- <OP1>: 14-bit immediate value or register which contains amount to shift

### Logical Shift

#### Assembly:

LS{DIRECTION} <DEST> <OP1>

2 directions \* 2 addressing modes: 4 total instructions.

#### Bit Organization:

<OP1> register direct:

Condition	Type	Operation	<DEST>	<OP1>	Not Used
5	2	6	5	5	9

<OP1> immediate:

Condition	Type	Operation	<DEST>	<OP1>
5	2	6	5	14

#### Behavior:

Performs a logical shift (ignores the sign of the number) on <DEST> and stores the result in <DEST>. The amount to shift is specified by <OP1>.

<OP1> can either be an immediate value or a register.

The direction bits are shifted is specified by {DIRECTION}:

{DIRECTION}	Direction
L	Left
R	Right

#### Operands:

- <DEST>: Destination register
- <OP1>: 14-bit immediate value or register which contains amount to shift.

### 3 Operand Logic

#### Assembly:

{OPERATION} <DEST> <OP1> <OP2>

3 operations \* 2 addressing modes = 6 total instructions.

#### Bit Organization:

<OP2> register direct:

Condition	Type	Operation	<DEST>	<OP1>	<OP2>	Not Used
5	2	6	5	5	5	4

<OP2> immediate:

Condition	Type	Operation	<DEST>	<OP1>	<OP2>
5	2	6	5	5	9

#### Behavior:

Performs a logic operation on <OP1> and <OP2> and stores the result in the <DEST> register.

The logic operation is specified by {OPERATION}:

{OPERATION}	Operation
AND	And
OR	Or
NOT	Not

#### Operands:

- <DEST>: Register result will be placed
- <OP1>: Register containing value to perform logic operation on
- <OP2>: 9-bit immediate value or register to use as second operand in logic operation

### Not

#### Assembly:

NOT <DEST> <OP1>

1 total instruction.

#### Bit Organization:

Condition	Type	Operation	<DEST>	<OP1>	Not Used
5	2	6	5	5	9

#### Behavior:

Inverts all the bits in <OP1> and stores them in <DEST>.

**Operands:**

- <DEST>: Register to store result
- <OP1>: Register containing value to invert

## Memory

5 total instructions.

Word based operations:

- Load
- Store
- Push
- Pop

**Bit Organization:**

The operation field of each memory instruction has the following meaning:

Binary	Operation
000	Load register direct
001	Load immediate
010	Store register direct
011	Store immediate
100	Push
101	Pop

## Load

**Assembly:**

LDR <DEST> <ADDR>

2 addressing modes = 2 total instructions.

**Bit Organization:**

Register direct:

Condition	Type	Operation	<DEST>	<ADDR>	Not used
5	2	3	5	5	12

Immediate:

Condition	Type	Operation	<DEST>	<ADDR>
5	2	3	5	17

**Behavior:**

Reads a word of memory from the address specified by the <ADDR> register into the <DEST> register.

If in the immediate form the address immediate is sign extended and added to the incremented program

counter value.

#### Operands:

- <DEST>: Register to store result
- <ADDR>: Register or signed immediate value which will be added to PC + 1 containing the memory address to access

### Store

#### Assembly:

STR <SRC> <ADDR>

2 addressing modes = 2 total instructions.

#### Bit Organization:

Register direct:

Condition	Type	Operation	<SRC>	<ADDR>	Not Used
5	2	3	5	5	12

Immediate:

Condition	Type	Operation	<SRC>	<ADDR>
5	2	3	5	17

#### Behavior:

Writes a word of data from the <SRC> register to the memory address specified by the <ADDR> register.

If in the immediate form the address immediate is sign extended and added to the incremented program counter value.

#### Operands:

- <SRC>: Register containing data
- <ADDR>: Register or signed immediate field added to PC + 1 containing the memory address to store data

### Push

#### Assembly:

PUSH <SRC>

1 total instruction.

#### Bit Organization:

Condition	Type	Operation	<SRC>	Not Used
5	2	3	5	17

**Behavior:**

Decrements the stack pointer and stores the contents of the <SRC> register at the address specified by stack pointer.

**Operands:**

- <SRC>: Register containing the data to be stored on stack

**Pop****Assembly:**

POP <DEST>

1 total instruction.

**Bit Organization:**

Condition	Type	Operation	<DEST>	Not Used
5	2	3	5	17

**Behavior:**

Reads a word from the memory address specified by the stack pointer register into the <DEST> register. Then increments the stack pointer register by one.

**Operands:**

- <DEST>: The destination register for data being popped off stack

**Control**

- Halt
- Jump

**Bit Organization:**

The operation field of each memory instruction has the following meaning:

Binary	Operation
0	Halt
1	Jump

**Halt****Assembly:**

HALT

1 instruction.

**Bit Organization:**

Condition	Type	Operation	Not Used
5	2	1	24

#### Behavior:

Signals to the hardware that the program is done running. Any instructions after this will not be interpreted.

Note that due to the way the bit pattern is constructed a memory value of 0 will be interpreted as a halt instruction:

- Condition 00000 is the null status
- Type 00 is the control type
- Operation 0 is the halt instruction
- The rest of the bits are unused

## Jump

#### Assembly:

<CONDITION>JMP{SPECIAL} <ADDR>

2 addressing modes = 2 total instructions.

#### Bit Organization:

Register direct:

Condition	Type	Operation	<ADDR>	Not Used
5	2	1	5	19

Immediate:

Condition	Type	Operation	<ADDR>
5	2	1	24

#### Behavior:

Conditionally executes a jump based on if the <CONDITION> operand matches the condition in the status register.

By default jump just sets the program counter as described in the next paragraph. The {SPECIAL} part of the instruction can be set to perform a modified behavior jump:

{SPECIAL}	Behavior
S	Subroutine jump
I	Return from interrupt jump
(Empty)	Normal jump

A subroutine jump sets the link register to the program counter register plus one. Then it performs a normal jump.

A return from interrupt jump performs the following actions:



- Checks if the interrupt flag is set in the status register, if not exits the instruction
- Sets the interrupt flag in the set status register to false
- Sets the program counter to the value in the interrupt link register
- Then performs a normal jump to the value stored in the interrupt link register

A normal jump sets the program counter register to the value specified by the <ADDR> operand.

In the register direct version of this instruction the program counter is set to the value in the <ADDR> register.

In the immediate version of this instruction the <ADDR> value is added to the program counter and the program counter is set to the result.

#### Operands:

- <ADDR>: Register containing new program counter value or a 24-bit immediate

## Graphics

8 total instructions.

- Load Sprite
- Set Bit Block Transfer Memory
- Set Bit Block Transfer Dimensions
- Bit Block Transfer

#### Bit Organization:

The operation field of each graphics instruction has the following meaning:

Binary	Meaning
00	Load Sprite
01	Set Bit Block Transfer Memory
10	Set Bit Block Transfer Dimensions
11	Bit Block Transfer

## Load Sprite

#### Assembly:

GLOD <DEST> <SRC> <LEN>

2 addressing modes = 2 total instructions.

#### Bit Organization:

Register direct:

Condition	Type	Operation	<DEST>	<SRC>	<LEN>	Not Used
5	2	2	5	5	5	20

Immediate:

Condition	Type	Operation	<DEST>	<SRC>	<LEN>	Not Used
5	2	2	12	10	12	1

**Behavior:**

Copies a sprite of length <LEN> bytes from <SRC> in main memory to <DEST> in the sprite library memory.

Both <DEST> and <SRC> are the start address of the sprite.

The memory region  $\text{SRC} + \text{CEILING}(\text{LEN} / 4)$  is copied from main memory to the region  $\text{DEST} + \text{LEN}$  in sprite library memory.

It should be noted that <DEST> is an address in the sprite library memory's 8-bit word address space, and <SRC> is an address in the main memory's 32-bit word address space.

**Operands:**

- <DEST>: 12-bit immediate or first 12 least significant bits of a register indicating the address in the sprite library memory to start copying to
- <SRC>: 10-bit signed integer added to the program counter or register holding the start address in main memory of the sprite
- <LEN>: 12-bit immediate or first 12 least significant bits of a register indicating the length of the sprite in bytes

**Set Bit Block Transfer Memory****Assembly:**

BLITMEM <SRC> <DEST>

1 addressing mode = 1 total instruction.

**Bit Organization:**

Condition	Type	Operation	<SRC>	<DEST>	Not Used
5	2	2	5	5	13

**Behavior:**

Sets the internal sprite library memory source address register of the Blitter to the first 12 bits contained in the <SRC> register.

Sets the frame buffer destination register of the Blitter to the first 16 bits contained in the <DEST> register.

**Operands:**

- <SRC>: Register who's first 12 least significant bits are an address to a sprite's start in the sprite library memory.
- <DEST>: Register who's first 16 least significant bits are an address to the start location in the frame buffer.

**Set Bit Block Transfer Dimensions****Assembly:**

BLITDIMS <WIDTH> <HEIGHT>

2 addressing modes = 2 total instructions.

**Bit Organization:**

Register direct:

Condition	Type	Operation	<WIDTH>	<HEIGHT>	Not Used
5	2	2	5	5	13

Immediate:

Condition	Type	Operation	<WIDTH>	<HEIGHT>	Not Used
5	2	2	7	7	9

#### Behavior:

Sets the internal width and height registers of the Blitter to the values specified by the <WIDTH> and <HEIGHT> operands.

#### Operands:

- <WIDTH>: 7-bit immediate value or register whose first 7 least significant bits will be used as the Blitter's width value
- <HEIGHT>: 7-bit immediate value or register whose first 7 least significant bits will be used as the Blitter's height value

## Bit Block Transfer

#### Assembly:

BLIT <OP> <DEST MASK>

2 addressing modes = 2 total instructions.

#### Bit Organization:

Register direct:

Condition	Type	Operation	<BLIT OP>	<DEST MASK>	Not Used
5	2	2	5	5	13

Immediate:

Condition	Type	Operation	<BLIT OP>	<DEST MASK>	Not Used
5	2	2	4	8	11

#### Behavior:

Executes a bit block transfer operation using the Blitter.

The Blitter operation which takes place is specified by the <BLIT OP> operand. See the Blitter Operations section for valid operation codes.

The <DEST MASK> is an 8-bit and-style-mask which will be applied to the bits of each pixel after it has been run through the Blitter operation and before it is copied to the destination.

#### Operands:

- <BLIT OP>: 4-bit immediate value or first 4 least significant bits of a register determining which Blitter

operation to perform, see Blitter Operations section

- **<DEST MASK>**: 8-bit immediate value or first 8 least significant bits of a register which will be used as an and-style-mask on pixels