Simple Assembly

For the simple assembler version 2.x.x

Version History

Date	Document Version	Assembler Version	Description
22nd March 2025	1.0	2.0.2	Initial release

Contents

1 Introduction

This document outlines how to use the simple assembler designed for the FPGA based SR-1 System on a Chip (SoC) which includes a 16bit integer CPU and a 16bit IEEE 754 floating point GPU.

The program takes simple human readable assembly instructions and converts them into machine code formatted for use with the GOWIN FPGA Designer's IP Core Generator to pre-program the SoC's BSRAM-based RAM.

This document specifically covers the 2nd version of the assembler, as this is the most recent and saw the introduction of significant useability improvements.

2 Usage

The assembler is written entirely in python and works via the command line.

The only required library is NumPy (this can be installed via pip).

The assembler requires a file structure as follows:

- sa_assembler.py
- Assembled (Folder)
- Programs (Folder)

Programs that you write must be placed within the Programs folder and should not be within any more folders within that.

The output of the assembler will automatically be placed into the Assembled folder.

Command Line Interface

```
> python3 sa_assembler.py

This is an assembler for the SR-1 SoC.
For assembly parameters use a single space between each one and the filename
Assembler debug mode: -db
No File Output mode : -nfo
Print Hex block     : -hx

Enter file name: Programs/test.sa
```

Figure 1: Example Program use

The file name prompt provides the beginning of the Programs/ path, and only requires you to type the exact filename, including the file extension (.sa)

The optional flags allow you to modify the operation of the assembler.

-db is for the assembler developer debugging purposes (not general program debugging – this is handled by the assembler itself). It outputs all 3 main passes of the assembler

-nfo is also for debugging, as it blocks the file output stage

-hx outputs the entire program as a single continuous hexadecimal string (gives a sense of program scale to those who are less familiar with low-level computing)

The flags are added as follows:

- Enter file name: Programs/test.sa -db -nfp -hx

There is a single space between each and the filename.

Assembly Success and Failure

When the assembler completes successfully, you will be met with an output like the one displayed in Figure 2.

```
Enter file name: Programs/test.sa

WARNING: Variable (num) is used in half mode operation, but is a 2B value. Only the first byte will be used
WARNING: Variable (num) is used in half mode operation, but is a 2B value. Only the first byte will be used

Assembled Successfully
Program Size: 103B
WARNING: This program has 2 warnings
Used 0.32% of available memory
Saving at: Assembled/test.mi
```

Figure 2: Assembly Successful

As you can see in Figure 2, the assembler will occasionally issue warnings regarding variable usage and different sizes. This will be elaborated on in Section 3 and Section 5.

If the assembler meets an error, it will fail. This can be seen below in Figure 3. If an error occurs, a brief message will be displayed, usually with a line number (if caught early enough in the assembly process) or a variable or alias name (generally when caught later)

```
WARNING: Variable (num) is used in half mode operation, but is a 2B value. Only the first byte will be used
WARNING: Variable (num) is used in half mode operation, but is a 2B value. Only the first byte will be used
ERROR: Half width variable (MM_LEDS) has illegal write attempt during full mode operation

Assembly Failure (1 error, 2 warnings)
```

Figure 3: Assembly Failure

3 Instructions and Syntax

There are currently 54 instructions in the SR-1's CPU instruction set. These are copied verbatim into the assembly instructions and make up most of those available. These basic operations follow a very simple syntax which will be explained after the CPU instructions Table.

CPU Architecture

It is important to know a little bit about the CPU architecture prior to reading the CPU instructions table.

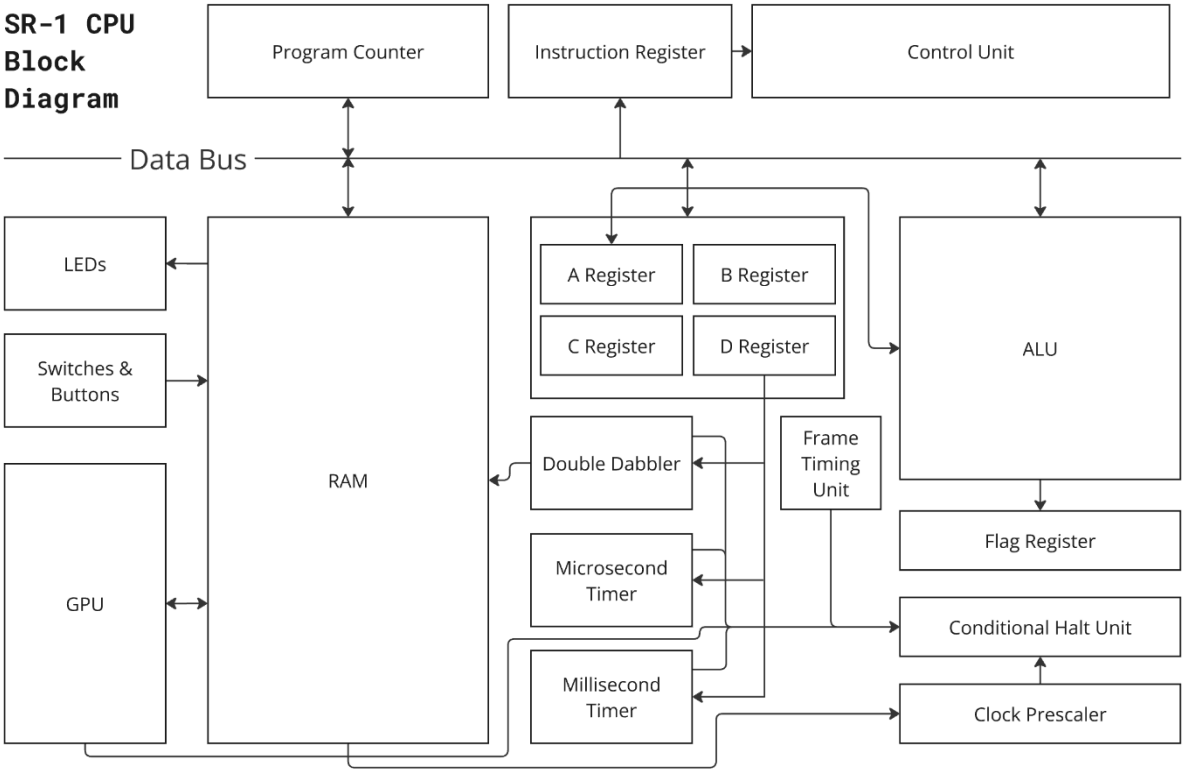

Figure 4: CPU Block Diagram

The CPU has 4 16bit registers which have different purposes:

A is the accumulator and takes the result of any ALU operation.

B and **C** are both cache registers, which are used to store values in the short term.

D is used as a common input register for both timing units, the double dabbler (binary → binary coded decimal) and the comparative jump instructions.

There are also peripheral units for various tasks:

- Millisecond timer (waits for value of (D + 1) milliseconds when started)
- Microsecond timer (waits for value of (D + 1) microseconds when started)
- Frame timing unit (waits for 16.67ms automatically for 60FPS video timing)
- Double Dabbler for binary → binary coded decimal conversion. (memory mapped ASCII output, takes D as the input)
- Clock pre-scaler (memory mapped input – see Pre-scaler Table for info)
- Conditional halt unit. (Halts the CPU until some condition is true. For example, **WMT** waits until the millisecond timer has stopped)
- Graphics Processing Unit and OLED display (see Section 7)
- 6 Memory mapped LEDs (on the Tang Nano PCB – active high)
- 1x 16bit and 1x 8bit input via DIP switches
- 7x button inputs (active high)

CPU Instructions Table

Opcode	Description	Parameters (Size in bytes)
Register and RAM Operations		
NXI	Fetch Next Instruction	N/A
LDA	Load A from RAM	Address (2)
LDAI	Load A from immediate	Immediate (v)
RDA	Read A to RAM	Address (2)
ATB	Copy register A into register B	N/A
ATC	Copy register A into register C	N/A
ATD	Copy register A into register D	N/A
LDD	Load D from RAM	Address (2)
LDDI	Load D from immediate	Immediate (v)
BTA	Copy register B into register A	N/A
CTA	Copy register C into register A	N/A
CPY	Copy 1B **to** Address 1 **from** Address 2	Address 1 (2) Address 2 (2)
CPY2	Copy 2B **to** Address 1 **from** Address 2	Address 1 (2) Address 2 (2)
Arithmetic and Logical Operations		
ADD	A + RAM value	Address (2)
SUB	A – RAM value	Address (2)
MUL	A × RAM value	Address (2)
LSHFT	A left shifted by 1 (A*2)	N/A
RSHFT	A right shifted by 1 (A/2)	N/A
AND	Bitwise A AND RAM value	Address (2)
OR	Bitwise A OR RAM value	Address (2)
XOR	Bitwise A XOR RAM value	Address (2)

NOT	Bitwise NOT A	N/A
ADDI	A + immediate	Immediate (v)
SUBI	A – immediate	Immediate (v)
MULI	A × immediate	Immediate (v)
ANDI	Bitwise A AND immediate	Immediate (v)
ORI	Bitwise A OR immediate	Immediate (v)
XORI	Bitwise A XOR immediate	Immediate (v)
ANDB	Logical A AND RAM value	Address (2)
ORB	Logical A OR RAM value	Address (2)
XORB	Logical A XOR RAM value	Address (2)
NOTB	Logical NOT A	Address (2)
ANDBI	Logical A AND immediate	Immediate (v)
ORBI	Logical A OR immediate	Immediate (v)
XORBI	Logical A XOR immediate	Immediate (v)
Jumping/Branching		
JMP	Jump to address	Address (2)
JPZ	Jump if last ALU operation equalled 0	Address (2)
JPC	Jump if last ALU operation overflowed	Address (2)
JPG	Jump if A > D	Address (2)
JPL	Jump if A < D	Address (2)
JPE	Jump if A == D	Address (2)
Timers and Halting		
SMT	Start Millisecond Timer	N/A
WMT	Wait Millisecond Timer	N/A
SUT	Start Microsecond Timer	N/A
WUT	Wait Microsecond Timer	N/A
WFT	Wait Frame Timer	N/A
HLT	Stops CPU operation (requires reset to restart)	N/A
Miscellaneous		
BCD	Copies and converts contents of register D into BCD	N/A
SHM	Sets the CPU to half (8bit) mode.	N/A
SFM	Sets the CPU to full (16bit) mode	N/A

Notes:

- Parameters with size of 'v' are dependent on half mode (to be elaborated on shortly)
- Parameters must be entered in the order that they descend in
- GPU related CPU instructions can be found in Section 7
- If conditional jumps do not evaluate true, they continue the current path

CPU Instruction Syntax

A few examples of how to use these instructions:

```
// Instructions example
LDA varName (start)   // Use at least one / for comments
LDAI 2                /  This is the only character that is checked
LDDI 0x32             /  Comments only span one line
CPY storeVar loadVar
JPZ *here
JMP *start         // * and (...) are aliases and their declarations
ADD var2 (here)    // These will be explained shortly
```

Supported numerical inputs (examples in square brackets):

- Integers [3454] [2149]
- Hexadecimal (prefixed with 0x) [0xFF] [0xAD03]
- Binary (prefixed with 0b) [0b110] [0b11110001]
- Floating point (must contain a decimal point) [3.14] [3.0]

Floating point is only for GPU use. The CPU will interpret all values as integers, and likewise the GPU will interpret all values as floating point. For conversion see **gLDI** and **gSCI** in Section 7.

You will receive a warning if a numerical input is detected after a non-immediate requiring instruction (i.e. **LDA**) as this is likely unintentional.

Aliases

These are a very powerful way of not only jumping between instruction paths but editing the parameters to those instructions themselves. This allows complex program behaviour as shown in the array access example.

An alias is declared using a name enclosed in brackets at the end of some instruction line. It represents the address of the instruction and can be used as an argument by suffixing it with a *.

Adding +x (where x is an integer offset) allows parameter level access to instructions. +0 accesses the first parameter, and then to access the next would be +<param1Width>, and next after that +<param1Width+param2Width> and so on.

```
// Alias example (creates geometric sequence of base 2)
LDAI  1 (start)
RDA *loop+0
ADDI 1 (loop)
JPC *start
RDA *loop+0 // Writing to ADDI parameter
JMP *loop // Jumping to ADDI
```

Variables

All variables are declared using **VAR1** or **VAR2**. They do not need to be declared before they are used. The number after VAR determines how many bytes the variable will use. Only 1B and 2B values are supported. The **VARx** keyword requires 2 arguments, the first being a name, and the second being an initial value. To access a variable's address at runtime, use an immediate loading instruction as illustrated below.

```
// Variable example
VAR2 xCoordForGPU 32.45
VAR2 yCoordForGPU 23.0
VAR1 smallVar 223
VAR1 iterator 1

SHM // Note the use of half mode during 1B variable use
LDA smallVar
ADD iterator
RDA smallVar
SFM // Return to 16bit mode
LDAI xCoordForGPU
```

Value-Initialised Arrays

The final instruction shown previously loads the address of xCoordForGPU using **LDAI**. This allows the creation of value initialised arrays by declaring variables sequentially. These variables will be placed by the assembler in the exact order they are declared. They must however still have distinct names (a suffixed index will suffice). These elements can be accessed via their names if needed, or through an offset from the first element (example on the next page).

Zero-Initialised Arrays

For large 0-initialised blocks of RAM, you can use the **BUF** keyword. It also takes 2 arguments, a name and a width (in bytes). Internal elements must be accessed using the offset technique shown on the next page.

All addresses are 2B and should be modified during full mode only

```
//Array access example
VAR2 nums0 2149
VAR2 nums1 1234
VAR2 nums2 0x223
VAR2 nums3 0b1111 // Note 2B values, thus index * 2 needed
VAR2 index 3 // Accesses 4th element (0 indexing)

BUF largeBuffer 1024 // 1kB buffer

LDA index
MULI 2
ADDI nums0
RDA *elementLoad+0
RDA *elementSave+0
LDA 0 (elementLoad) // Using 0 as placeholder – can ignore warning
// Do something here
RDA 0 (elementSave)
```

Buffers are used in the same way, just that they are only accessible via an offset.

Pre-defined Variables

The last 256B of RAM are reserved for memory mapping. These are pre-defined in the assembler and are listed in the memory mapping table. These names cannot be used for anything else.

Memory Mapping Table

Identifier	Description	Size (bytes)	Writable
MM_CLKPRE	Set the clock pre-scaler (see clock scale table)	1	Yes
MM_BUTTON	Various buttons on the system (see button lookup table)	1	No
MM_WIDE	Bottom 16bit DIP switches	2	No
MM_THIN	Top 8bit DIP switches	1	No
MM_x	Read (x = B, C, D) for register-on-register arithmetic (ex. ADD MM_B for A + B)	2	No
MM_A_x	Read (x = L, H) the low or high byte of A	1	No
MM_DDx	Read ASCII output of binary to BCD unit	1	No
MM_LEDS	Controls the 6 LEDs on the PCB (LSB is top LED, and so on. Only use LS 6 bits.	1	Yes
CONST_ZERO	Used with CPY and CPY2 for resetting to zero	2	No

Notes:

- The GPU is also memory mapped but isn't accessed through these mappings usually (and is excluded here). See Section 7 for information.

Clock Scale Table

Value	Clock Frequency (Hz)
0	2.7×10^7
1	1.35×10^7
2	6.25×10^6
3	2.7×10^6
4	1.5×10^6
5	7.5×10^5
6	4.5×10^5
7	3×10^5
8	2.25×10^5
9	1×10^5
10	1×10^4
11	1000
12	100
13	10
14	2
15	1

Button Lookup Table

Bit Index	Button description
0	Left Dual Buttons (Right)
1	Left Dual Buttons (Left)
2	Arrow keys – Bottom
3	Arrow keys – Top
4	Arrow keys – Right
5	Arrow keys – Left
6	Enter – In the middle of the 16bit (wide) switches
7	Unassigned

To access a specific button, you can use a bitmask of the value $2^{Bit\ Index}$.

4 Writing a Basic Program

Here is a very simple program that turns an LED on and off with a period of 2 seconds.

Pseudocode

```
A     bitmask = 1
B     while (true) {
C           wait(1 second)
D           LEDS = LEDS ^ bitmask // bitwise XOR
E     }
```

Simple Assembly

```
1     VAR1 bitmask 1
2
3     LDDI 999
4     SMT (loopStart)
5     SHM
6     LDA MM_LEDS
7     XOR bitmask
8     RDA MM_LEDS
9     SFM
10    WMT
11    JMP *loopStart
```

Conversion

Line A maps perfectly to line 1

Line B is transformed into line 11 and the alias declaration on line 4.

Line C is transformed into line 3 (setting the D register to (1000 – 1) 999 so that the millisecond timer waits for 1 second), line 4 and line 10. Line 4 starts the timer, and line 10 waits for it to complete.

Line D is transformed into lines 5 – 9.

5) Sets the CPU to half mode (as MM_LEDS is an 8bit value)

6) Loads the **A** register with the current value of MM_LEDS

7) XORs the **A** register with the bitmask (this inverts the active bits – in this case bit 0 ($2^0 = 1$))

8) Reads the **A** register back into MM_LEDS

9) Sets the CPU to full mode (not strictly needed in this case as there are no 16bit ALU requirements) as it is good practice to leave the CPU in either half or full mode by default and only switch to the other temporarily.

This program is then assembled into this:

```
#File_format=AddrHex
#Address_depth=32768
#Data_width=8
2:08
3:e7
4:03
5:2a
6:30
7:01
8:df
9:7f
a:12
b:15
c:00
d:03
e:df
f:7f
10:31
11:2c
12:21
13:04
14:00
15:01
```

The entire program is just 20B long. That uses less storage space than the previous sentence! And luckily with the assembler, you do not need to understand the above machine code.

5 Debugging and Error Handling

The assembler will catch many basic errors that should reduce debugging time, as well as warning you if there is a possible mistake. The error detection is limited however, and memory safety (Section 6) is still very much on the developer to consider.

These errors usually give a line number if they are caught early enough in the assembly process, but otherwise, they give as much information on the error causing line as possible. This may be an argument or variable name. They are written to be as obvious as possible.

Things like failure to parse numerical values, identifiers already existing and operations not being recognised will all cause errors. Any error will cause the assembly process to fail.

Warnings do not cause the assembly process to fail, but just flag lines that may have a potential mistake on, which helps to debug issues like numerical values where an address is expected.

6 Memory Safety

Writing 2B to 1B Variable

As a developer using simple assembly to program the SR-1, you are free to access all memory addresses in RAM should you need to. There are safeguards in place to reduce this where it was likely unintentional.

To prevent writing a 2B wide value to a 1B wide variable (and thus also overwriting the next byte in memory) the assembler will check when the system is operating in full mode (see Section 3 for details on full / half operation) and tries to write to a variable with a 1B value.

Should you as the developer need to write a 2B value to a 1B address, you can use the more explicit CPY2 instruction (see bottom of page). This will copy 2B from one address to another and has no checks in place (therefore use it carefully). This is unadvisable unless you know exactly what you are doing.

You can write 1B values into 2B memory addresses, but the assembler will warn you about this. You can also read 2B addresses as 1B addresses, and again you will be warned

Memory Mapping

Variable addressing is handled by the assembler, and it has a program size limit in place making it impossible to accidentally create a variable inside the memory mapped locations.

Any data written to read only addresses will be lost.

All memory mapped addresses can be accessed with predefined variables as stated in Section 3.

Example: If you have the value stored in register A, and wish to use CPY2 on it, load it into a 2B variable first before copying it.

7 GPU Interface

Attached to the SR-1's CPU is a small but powerful graphics processor. It operates entirely on IEEE 754 16bit floating point numbers and supports drawing lines, curves (consisting of consecutive line segments) and points. The system supports either 2D or 4D matrix multiplication via 4 matrix acceleration units (MAUs). At 13.5MHz, the system can process 500,000 4x4 * 4x1 multiplications, or roughly 2 million 2x2 * 2x1 every second. This is approximately 15 MFLOPS.

GPU Architecture

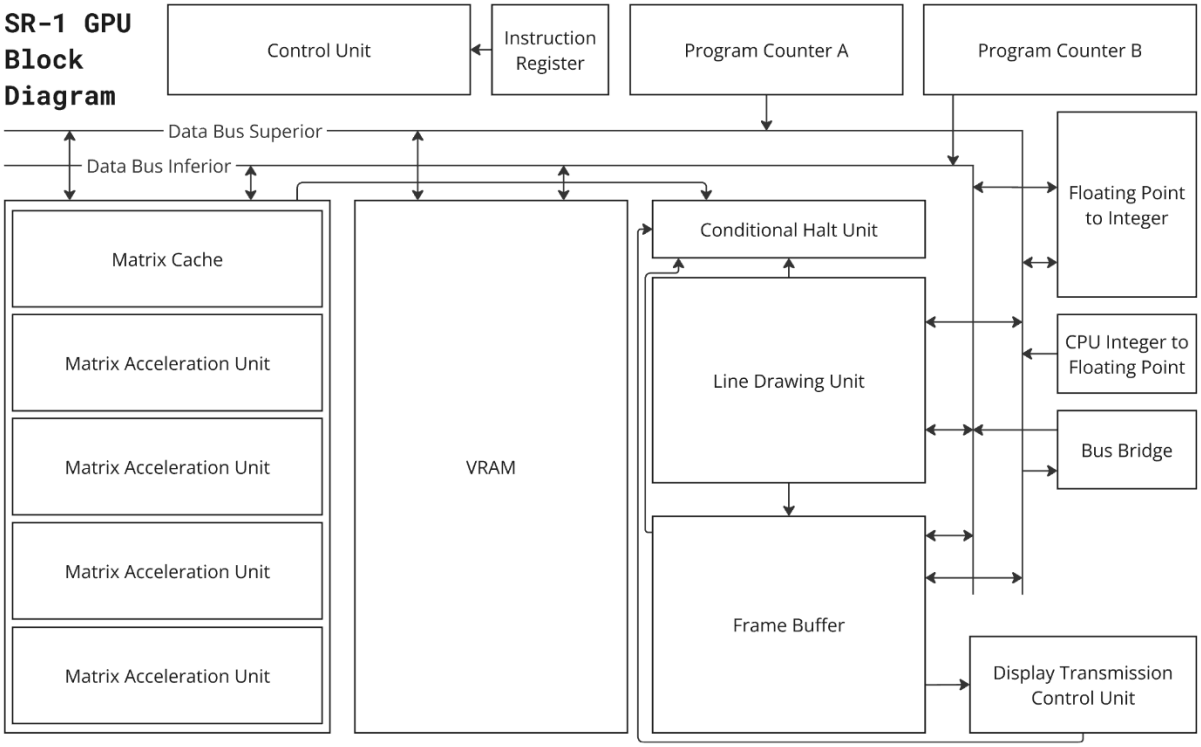

Figure 5: GPU Block Diagram

The GPU is constructed as shown in Figure 5. Only a very basic understanding of the architecture is required to be able to program it. Programming is done via instructions in the CPU-GPU Instruction Table. The GPU interface is memory mapped into RAM as according to the GPU Memory Mapping Table.

CPU-GPU Instruction Table

Opcode	Description	Parameters (Size in bytes)
GDT3	Load 3 GPU arguments	Opcode (1) Counter A Initial Value (2) Data to transfer (2)
GDT7	Load all 7 GPU arguments	Opcode (1) Counter A Initial Value (2) Data to transfer (2) Counter B Initial Value (2) Counter A Increment (1) Counter B Increment (1) Repeat Amount (1)
GPU	Wait and then start the GPU	N/A
WGPU	Wait for the GPU	N/A

Notes:

- These are used in the exact same way as the instructions in Section 3.
- The CPU will wait for the GPU to be ready before sending any instructions to the GPU, thus making it impossible for any instruction to be missed.

GPU Memory Mapping Table

Identifier	Description	Size (bytes)	Writable
MM_GREPEAT	Repeat amount	1	Yes
MM_GPCBI	Counter B Increment	1	Yes
MM_GPCAI	Counter A Increment	1	Yes
MM_GPCB	Counter B initial value	2	Yes
MM_C2G	Read (x = B, C, D) for register-on-register arithmetic (ex. ADD MM_B for A + B)	2	Yes
MM_GPCA	Counter A initial value	2	Yes
MM_GINSTR	GPU Instruction to execute	1	Yes
MM_G2C	GPU direct floating-point output	2	No
MM_GFP2I	GPU converted integer output	2	No

Notes:

- These can be set manually via the identifiers, but it is much easier to use the dedicated transfer instructions (**GDT3** and **GDT7**)
- The outputs are read like normal variables.

GPU Instruction Table

Opcode	Description	Counter A	Data Transfer	Counter B	Increment A	Increment B	Repeat Amount
gLDC ($)	Load VRAM from Data Transfer	Address	Data to load	N/A	N/A	N/A	N/A
gLDI ($)	Load VRAM from I2FP(Data Transfer) *	Address	Data to load	N/A	N/A	N/A	N/A
gSDC ($)	Set Draw Colour	Address	LSB 0 = Black LSB 1 = White	N/A	N/A	N/A	N/A
gLMV	Load matrix from VRAM consisting of R elements	MatRAM Address	N/A	VRAM Address	1	2	(R / 2) – 1
gMM2	2x2 by 2xR matrix multiplication	Read Address	MatRAM Address	Write Address	2	2	R – 1
gMM4	4x4 by 4xR matrix multiplication	Read Address	MatRAM Address	Write Address	2	2	R – 1
gDRL	Draw lines or curves (consisting of R line segments)	Point 0 X Address	N/A	Point 1 X Address	Any	>1	R – 1
gDRP	Draw R points	X Coordinate Address	N/A	N/A	>1	>1	R – 1
gMA2	2x1 Matrix / Vector addition**	Read Address	MatRAM Address	Write Address	1	2	R – 1
gSCO ($)	Output Value to MM_G2C***	Address	N/A	N/A	N/A	N/A	N/A
gSCI ($)	Output Value to MM_GFP2I***	Address	N/A	N/A	N/A	N/A	N/A
gINI ($)	Initialise the OLED display	N/A	N/A	N/A	N/A	N/A	N/A
gSNF ($)	Send the next frame	N/A	N/A	N/A	N/A	N/A	N/A
gCPY	Copy R values starting at A to starting at B	Read Address	N/A	Write Address	>0 (usually 1)	>0 (usually 1)	R-1

Notes:

- Any Instruction marked with a ($) can be used with GDT3, otherwise a full argument transfer (GDT7) is required.
- For further information on the repeat system and counter increments, please see the subsection about the GPU Instruction Repeat System.
- If something is marked as N/A, just set it to zero.
- Arguments are always in the exact order as shown in the GPU Instruction Table.
- All references to addresses are GPU addresses. No argument should be set to a RAM address or variable name.
- Currently VRAM and MatRAM are managed by the developer, who must keep track of variable addresses themselves.
- * I2FP means Integer to Floating Point. It is unsigned, and must be used if you want to send any processed numerical data to the GPU (even if it is for display coordinates – it will be converted back when drawing)
- ** Matrix addition requires a special matrix that will be covered by in the Matrix / Vector Addition section
- *** These are memory mapped variables. G2C sends the data over directly whereas GFP2I first sends the data through the Floating Point to Integer (FP2I) unit before sending to the CPU. The sign of the data is ignored in the FP2I units.

GPU Instruction Repeat System

Some GPU instructions can be repeated on consecutive data via the repeat register and counter increments. This allows for line-segment-based lines, multiple point drawing, transformation of entire coordinate objects and transfers of large amounts of data into MatRAM with single instruction calls from the CPU.

In the GPU Instruction Table, there is a column called Repeat Amount. This has an equation involving a parameter R. It states that if you want to repeat the instruction for a total of 10 cycles, you set R to 10 and evaluate the result. This gives the value that the repeat register should be set to. The increments shown are usually applied once per cycle but may sometimes (such as in **gMM2** and **gMM4**) be applied multiple times. These numbers may be a single value, such as 2. In this case the increments must be this value and only this value. If a range of values is shown, then a range of values can be chosen, so long as the equality is respected. For example, sometimes you may want an increment of 2 2 when drawing lines as this will yield a curve drawn from consecutive points in VRAM. Other times you may want to draw multiple lines originating from a single point, in which case an increment of 0 2 or 2 0 will work (the first drawing lines from a fixed P0 and the second drawing lines from a fixed P1.

VRAM

The GPU has 12,288B of VRAM, with a word size of 2B, yielding 6,144 addresses. Addressing starts at zero, and currently use must be recorded by the developer, as there are no graphics variables yet. The VRAM is dual read and dual write, which is utilised extensively in many instructions (usually reading from one and writing to the other.)

Examples

```
GDT3 gLDC 27 32.4
GPU // Transferring 32.4 to a VRAM address of 27
GDT3 gLDI 28 10
GPU // Transferring I2FP(10) to a VRAM address of 28

// Loading 4 variables via a loop into VRAM starting at address 0
VAR2 x0 123.53
VAR2 y0 4332.73
VAR2 x1 -3.141
VAR2 y1 32.33

LDAI 0
ATB // Setting A and B regs to zero
LDDI 4 // Setting loop repeat amount
BTA (loopStart) // Getting index
MULI 2 // Multiply by 2 as VAR2s being used
ADDI x0 // Getting first variable address
RDA *transfer+3 // Read into 3rd argument of GDT3
GDT3 gLDC 0 0 (transfer)
GPU
BTA
ADDI 1
RDA *transfer+1 // Read into 2nd argument of GDT3
ATB
JPL *loopStart
```

MatRAM

The matrix Cache or MatRAM is the fast quad-read RAM attached to the MAUs. It has only 512 addresses with a word size of 4B (2 16bit words from VRAM are stored at a single address). This means that a 4x4 matrix only takes up 8 MatRAM addresses, and a 2x2 matrix only takes 2. As with the VRAM, all addresses must be recorded by the developer.

The gLMV instruction is used to store data from VRAM in MatRAM. The repeat equation is different from all the others, as 2 values are transferred per cycle.

Examples

```
GDT7 16 0 0 1 2 7
GPU // Transferring a 4x4 matrix stored at VRAM 16 to MatRAM 0
GDT7 64 0 8 1 2 1
GPU // Transferring a 2x2 matrix stored at VRAM 64 to MatRAM 8
```

Matrix / Vector Addition

You can use the **gMA2** instruction to add two 2D coordinates, however this requires one of the coordinates to be in MatRAM in an "addition matrix" shown below.

$$\begin{bmatrix} 1 & a \\ 1 & b \end{bmatrix}$$

The result of this against a 2D vector $\begin{bmatrix} x \\ y \end{bmatrix}$ stored in VRAM is $\begin{bmatrix} x+a \\ y+b \end{bmatrix}$.

The reason for this unconventional addition system is that it uses the MAUs because the GPU doesn't have any dedicated addition units.

CPU-GPU Instruction Examples

```
// This snippet is the initialisation sequence that was used in
// most included programs
LDDI 999
SMT
WMT // Waits for 1s before sending pre-initialised frame (of the
    // computer's logo)
SMT // Start timer again
CPY MM_CLKPRE clkspd
GDT3 gINI 0 0 // Initialising the display
GPU
GDT3 gSNF 0 0 // Sending first frame
GPU
WMT // Wait until 1s passed before starting main program

//7 Argument example
GDT7 gMM4 16 24 32 2 2 3
GPU // Multiplication of 4 4D coordinates (starting at VRAM 16)
    // against a 4x4 matrix stored at MatRAM 24 before saving the
    // result to VRAM 32 onwards.
```

8 Example Programs

Simple Software Divider

Pseudocode

```
// For 16bit numbers
dividend = <number>
divisor = <number>
multiplicand = 0x8000
while (multiplicand > 0) {
      temp = divisor * multiplicand
      if (dividend - temp > 0) {
            dividend = dividend – temp
            quotient = quotient | multiplicand // bitwise OR
      }
      multiplicand = multiplicand / 2
}
remainder = dividend
```

Simple Assembly

```
VAR2 dividend <number>
VAR2 divisor <number> //Integer division
VAR2 quotient 0
VAR2 remainder 0
VAR2 multiplicand 0x8000

LDA divisor (start)
MUL multiplicand
JPC *shiftMult //If carries, result will go negative
ATB
LDA dividend
SUB MM_B
JPC *shiftMult //If carries, result went negative
RDA dividend
LDA multiplicand
OR quotient
RDA quotient
LDA multiplicand (shiftMult)
RSHFT
RDA multiplicand
```

```
JPZ *exit
JMP *start
CPY2 remainder dividend (exit)
```

Notes:

- Currently this can only be run once, as the multiplicand and quotient are not reset at the start. If you wish to use this in a program, make sure to include something like:
  `CPY2 multiplicand multiplicandRESET`
  `CPY2 quotient CONST_ZERO`
  at the start. multiplicandRESET is set to the value 0x8000 just like above.
- This simple software divider can be found pre-implemented under divider.sa