Tronsistor-32: Final Report

<2015-12-22>, version 1.0

University of Wisconsin-Madison

ECE 554 - Fall 2015

Team: Tronsistors Team Leader: Graham Nygard

Matt Kelliher, Max Eggers, Graham Nygard, Jake Truelove, Kai Zhao, John Roy

Source code: https://github.com/meggers/ece554project

Table of Contents

1.0 System Overview and Background

The inspiration behind the creation of the Tronsistor-32 was rooted in the retro gaming consoles of the 1980's, most specifically the iconic Atari 2600. Our efforts were focused on developing a system capable of playing the 1982 video game Tron, as well as a few other nostalgic video games from that same era. On a high level, the Tronsistor-32 includes a 32-bit architecture CPU operating on a custom version of the MIPS32 ISA. The design is capable of displaying 8-bit video graphics at a 256x256 pixel resolution, and uses a simple keyboard for a player's control inputs. This design was primarily chosen to challenge our technical abilities as well as provide an enjoyable, interactive, and memorable product capable of playing some of the most famous games from the golden era of gaming.

With regard to technical challenges, this design decision offered several opportunities to grow our hardware and software development skills. For starters, this design required us to build a system essentially from the ground up. Our decision to create our own ISA necessitated the development of a specialized software framework on top of a fully custom hardware design. Furthermore, using a custom ISA required that we implement new testing techniques in order to debug and verify the correctness of our developed software. The entirety of our design relied solely on our ability to identify and resolve issues throughout each stage of the design process, and this was aimed to emulate the work and ambition needed while working in the professional computer engineering industry.

This document includes an in depth discussion of the hardware and software architecture implementations of the Tronsistor-32. **Section 2** and **Section 3** of this document will discuss our hardware specifications, specifically the hardware macro/micro architecture design as well as the ISA used to develop our applications. **Section 4** will review our software framework, which incorporates a discussion of how to develop and test programs that will eventually execute on the hardware.

2 System Architecture

2.1 Top Level Hardware Diagram

Figure 2.1 shows a top level view of the Tronsistor-32 hardware architecture. The main components of the design are the Central Processing Unit (CPU) and the Picture Processing Unit (PPU). The CPU interfaces with main memory, which contains a game's instruction space as well as the heap/stack data that is regularly accessed during execution. The PPU interfaces with specialized memory needed for graphical output, such as a game's foreground/background sprite image tables, as well as the certain color palettes needed by these images.

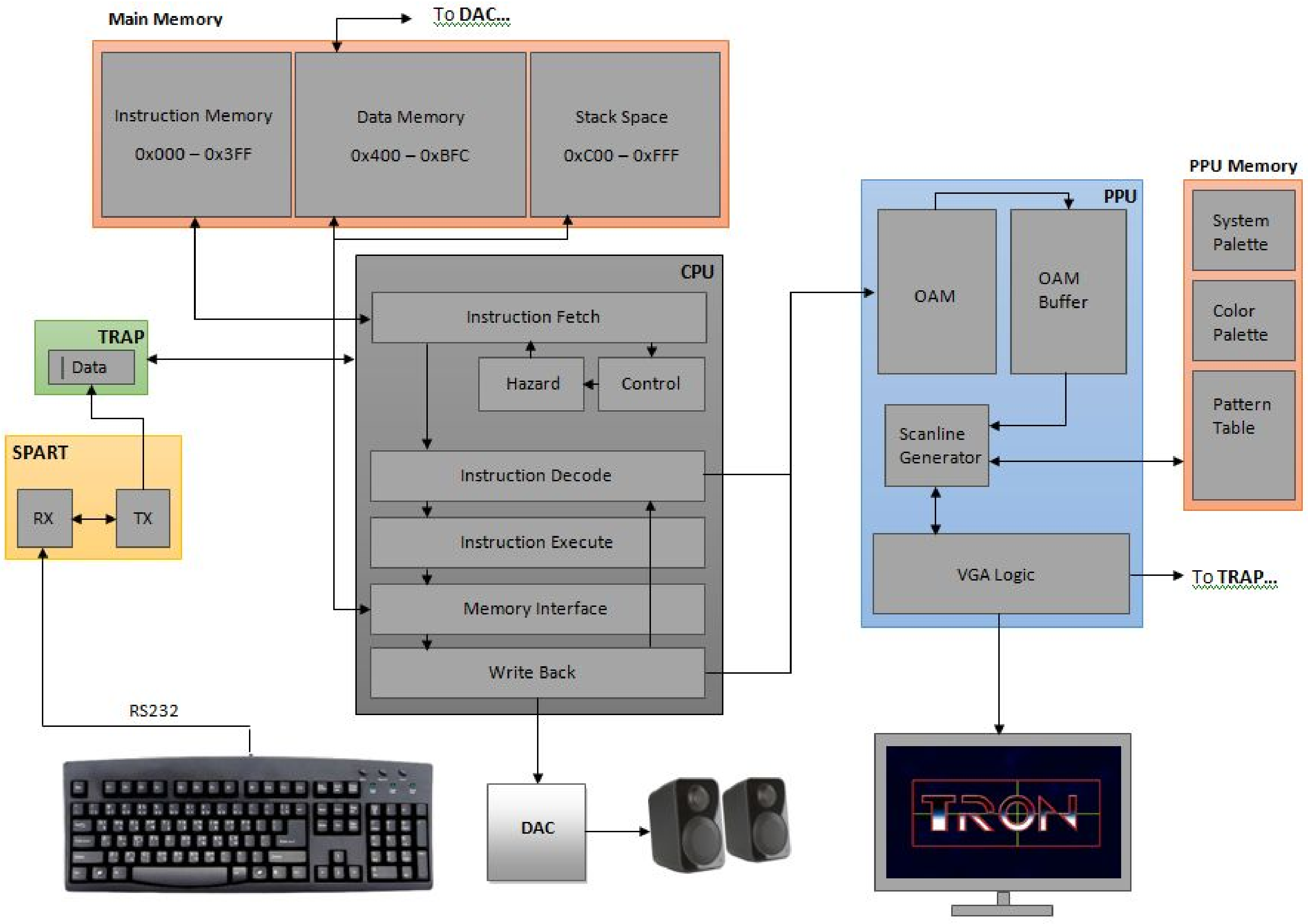


Figure 2.1: Tronsistor-32 top level hardware design

2.2.0 Picture Processing Unit

The Picture Processing Unit (PPU) uses sprite and background data to render the actual pixel data displayed on the screen. The PPU renders pixels line by line horizontally, which is synchronized with the timing of the VGA display logic. Each line is 256 pixels wide, with 256 lines, giving a display resolution of 256x256.

The purpose of having all rendering be just one line at a time is so that the PPU is able to output the most relevant pixels at at that given time. This was a decision over the alternative of rendering frame by frame, because of the format of a VGA signal. This also allows the PPU to not have to buffer an entire frame's worth of pixel data in a memory block, which would greatly reduce the available space in an already constrained area.

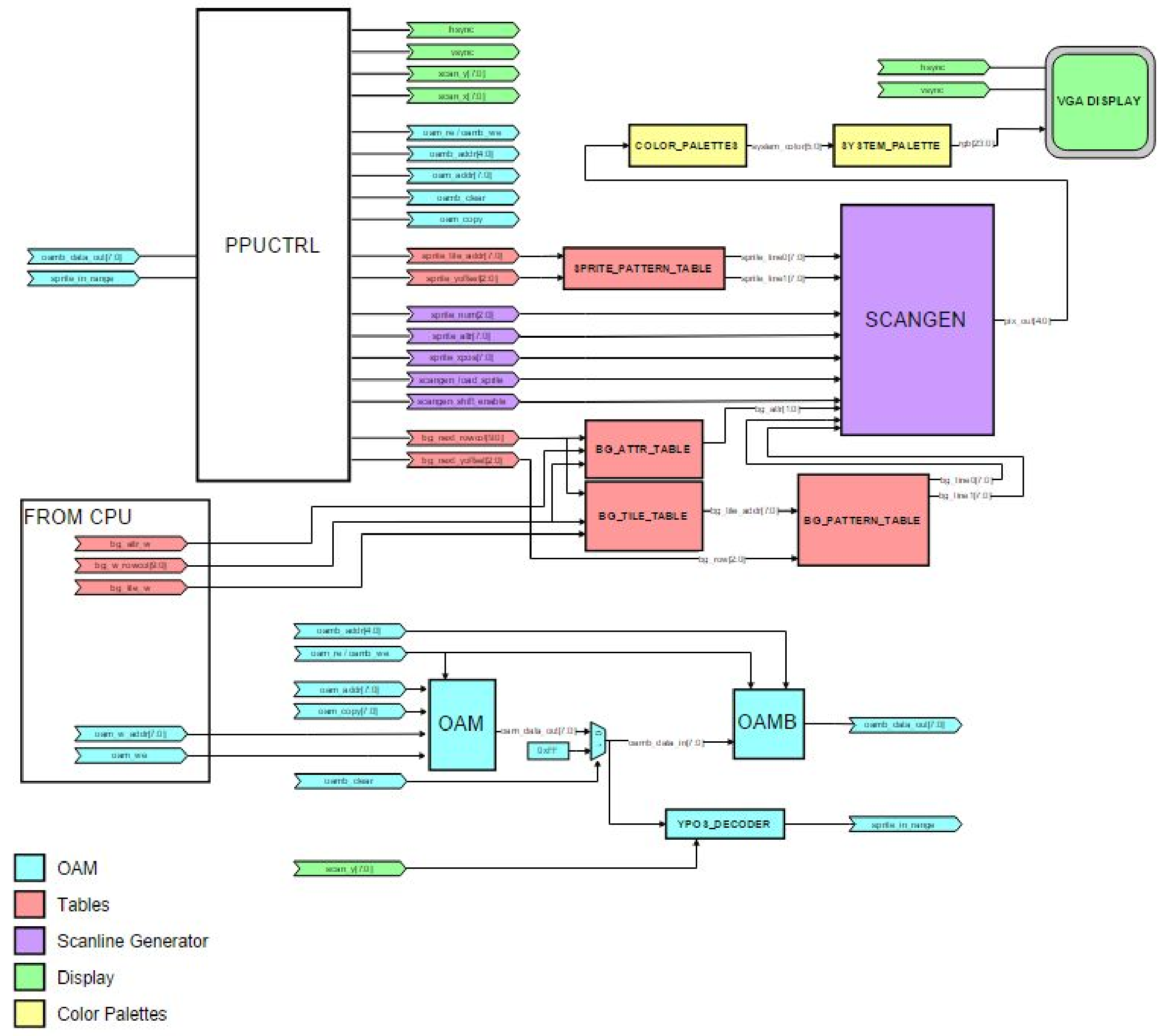


Figure 2.2.1: PPU Top Level

2.2.1 PPU Control Unit (PPUCTRL)

The PPUCTRL is the control unit that coordinates all operations of the PPU. Its main operations include scanning through the OAM and loading the OAMB with sprites that will be visible on the current scanline and loading up the SCANGEN with this data from the OAMB.

The steps the PPU takes each scanline are as follows:

1. Clear the OAMB to 0xFF at each position.
2. Check each sprite in the OAM, if it is within range, copy its data into the next available position in the OAMB. Stop checking if OAMB is full or all sprites in OAM have been checked.
3. Wait for hsync
4. Load each sprite from the OAMB into the SCANGEN. This requires fetching tile data from the sprite pattern table.
5. Load the tile and attribute data for the first two background tiles in the current row into the SCANGEN.
6. Enable the SCANGEN to start shifting out pixels.
    a. Each eight shifts, load the next background tile and attribute data into the SCANGEN.
    b. While the SCANGEN is shifting out the current line, step 1 can begin again in parallel.

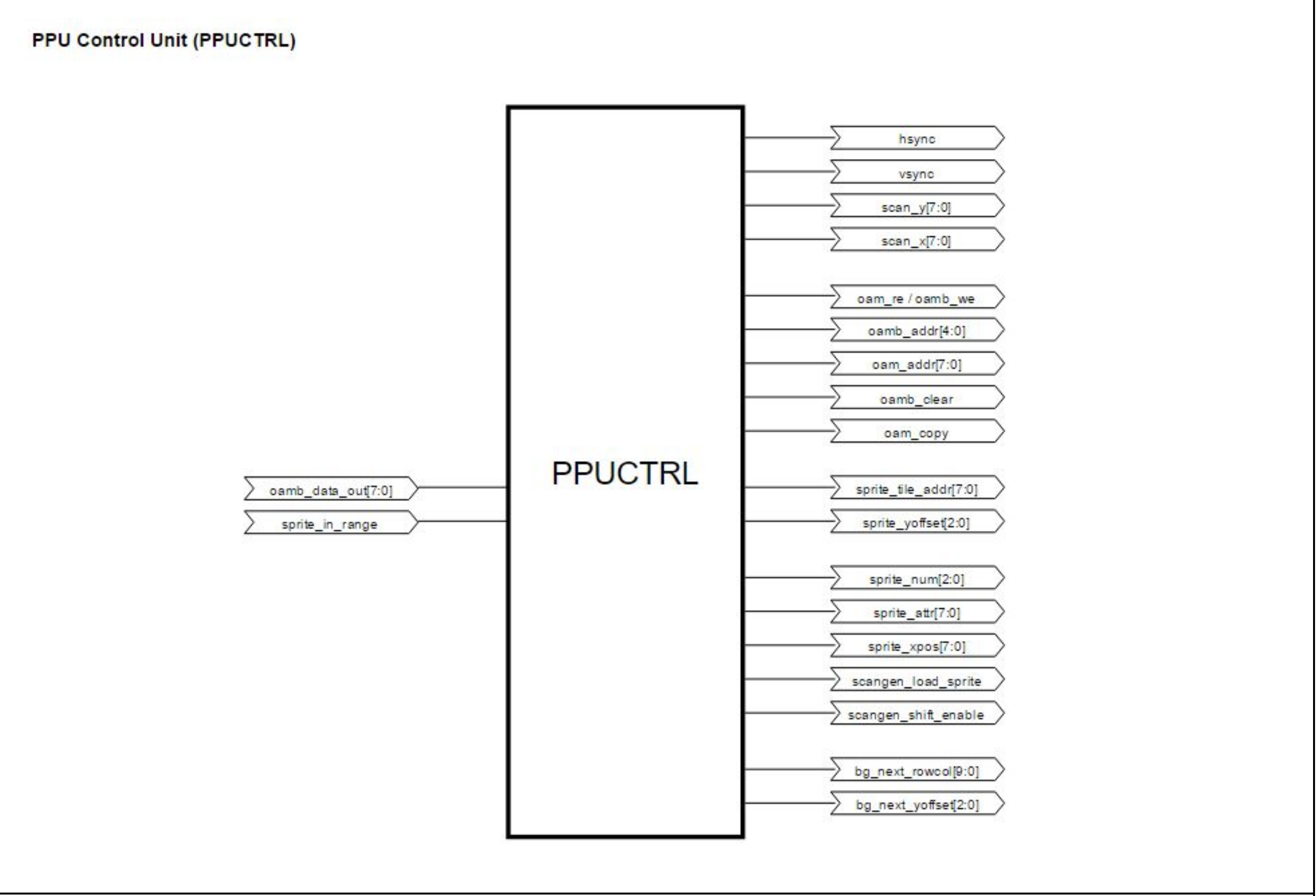


Figure 2.2.2: PPU Control Unit

2.2.2 Object Attribute Memory (OAM)

The OAM stores information about sprites that are currently displayed on the screen. Externally, the OAM can be viewed as a byte-addressable memory with concurrent read and write access. Internally, the OAM actually contains two memory blocks of equal size. One primary OAM and secondary OAM. To allow for a program to control the sprites, the CPU is given direct access to the PPU's primary OAM. The primary OAM that the CPU writes to is actually never directly accessed by the PPU other than one single occasion. Once a frame is complete, the entire contents of the primary OAM are copied into the secondary OAM over a single clock cycle. This way, there are no issues dealing with concurrent read and writes on the OAM causing sprite flickering. Therefore, if contents of the OAM change, it will not appear to have changed (when reading), until copy is held high for a clock cycle.

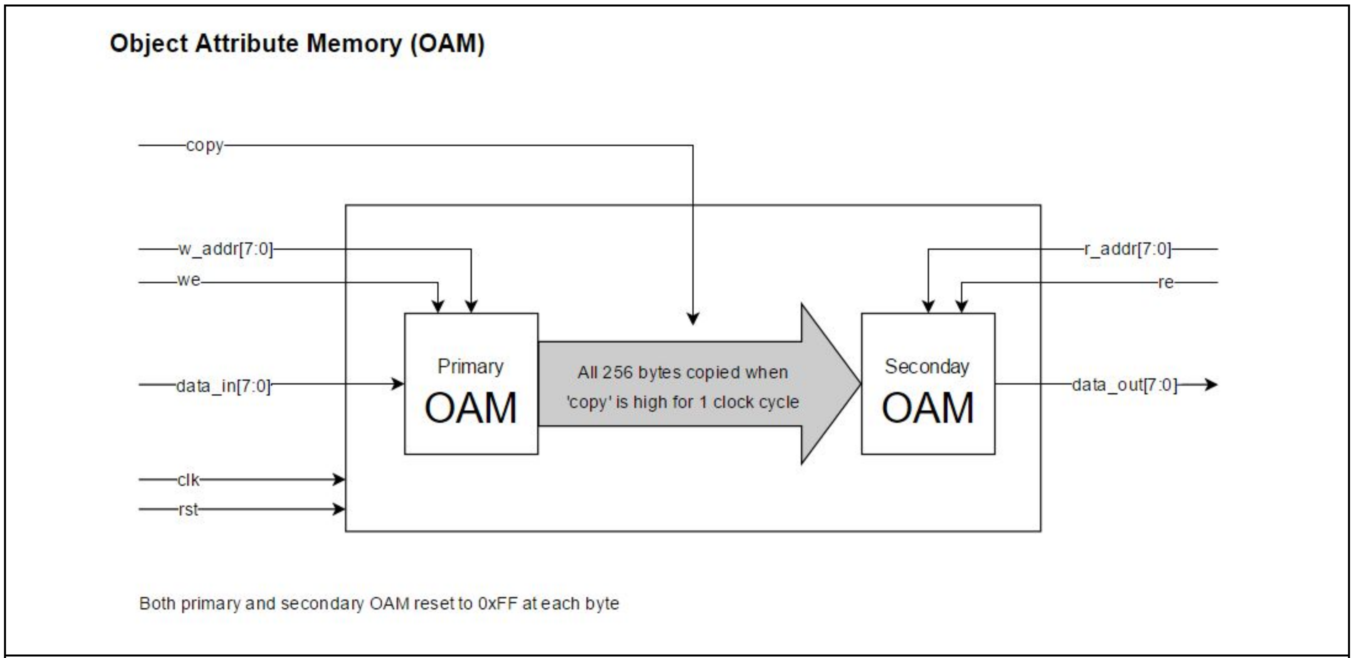

Object Attribute Memory (OAM)
copy
w_addr[7:0]
we
data_in[7:0]
clk
rst
Primary OAM
All 256 bytes copied when 'copy' is high for 1 clock cycle
Seconday OAM
r_addr[7:0]
re
data_out[7:0]
Both primary and secondary OAM reset to 0xFF at each byte

Figure 2.2.3: Object Attribute Memory

The OAM can contain information about a maximum of 64 sprites. Each sprite takes four bytes of data. The ordering of the bytes for each sprite are as follows.

Byte	Name	Description
0	Y Position	The Y position of the top-left corner of the sprite on the screen. If this byte is set to 0xFF, the sprite is considered an empty entry.
1	Tile Number	The tile used for this sprite: an index into the sprite pattern table.
2	Attributes	Attributes about how this sprite is displayed. [1:0] The color palette used for this sprite. [5:2] Unused [6] Flip horizontal if 1 [7] Flip vertical if 1
3	X Position	The X position of the top-left corner of the sprite on the screen.

2.2.3 OAM Buffer (OAMB)

The OAM Buffer (OAMB) is a sprite holding memory with a similar format to the OAM. It can only hold information about 8 sprites (32 bytes total) that will be rendered on the current scanline. The PPUCTRL module is responsible for controlling the memory loaded into this module by reading from the OAM and determining if each sprite is on the current scanline. If it is, the sprite's 4 bytes are loaded into the OAMB at its next available position. The PPUCTRL is also responsible for clearing this module (setting each of its bytes to 0xFF) before it is loaded with the next scanline's sprites. The OAMB is byte-addressable for both reads and writes. Because this module can only contain 8 sprites, that is the limit of the number of sprites that can be displayed on a scanline.

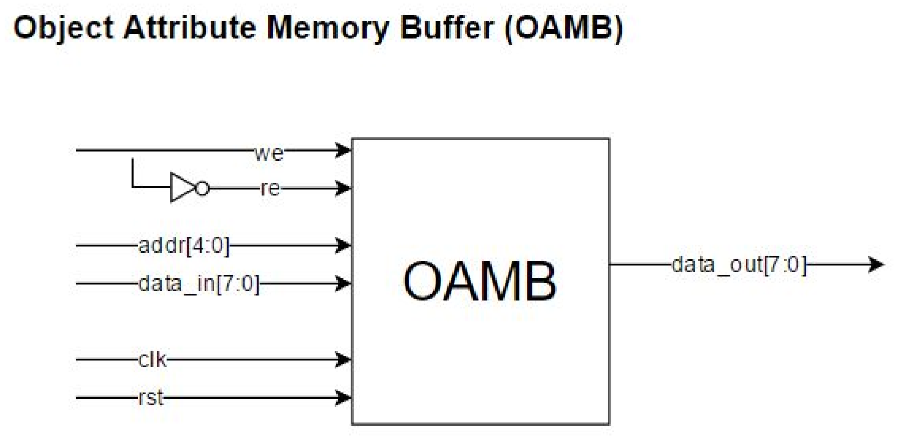


Figure 2.2.4: OAM Buffer

2.2.4 Scanline Generator (SCANGEN)

The scanline generator is responsible for shifting out color indices that index into the color palettes for display on the screen. The SCANGEN is loaded with sprite data assembled by the PPUCTRL after scanning through the OAM and loading the OAMB. Both the OAMB and the SCANGEN are capable of holding 8 sprites. The scangen prioritizes non-transparent pixels from sprites at lower indices. If no sprite pixels are available, the background pixels are used with lowest priority. Each shift out is a pixel on the current scanline.

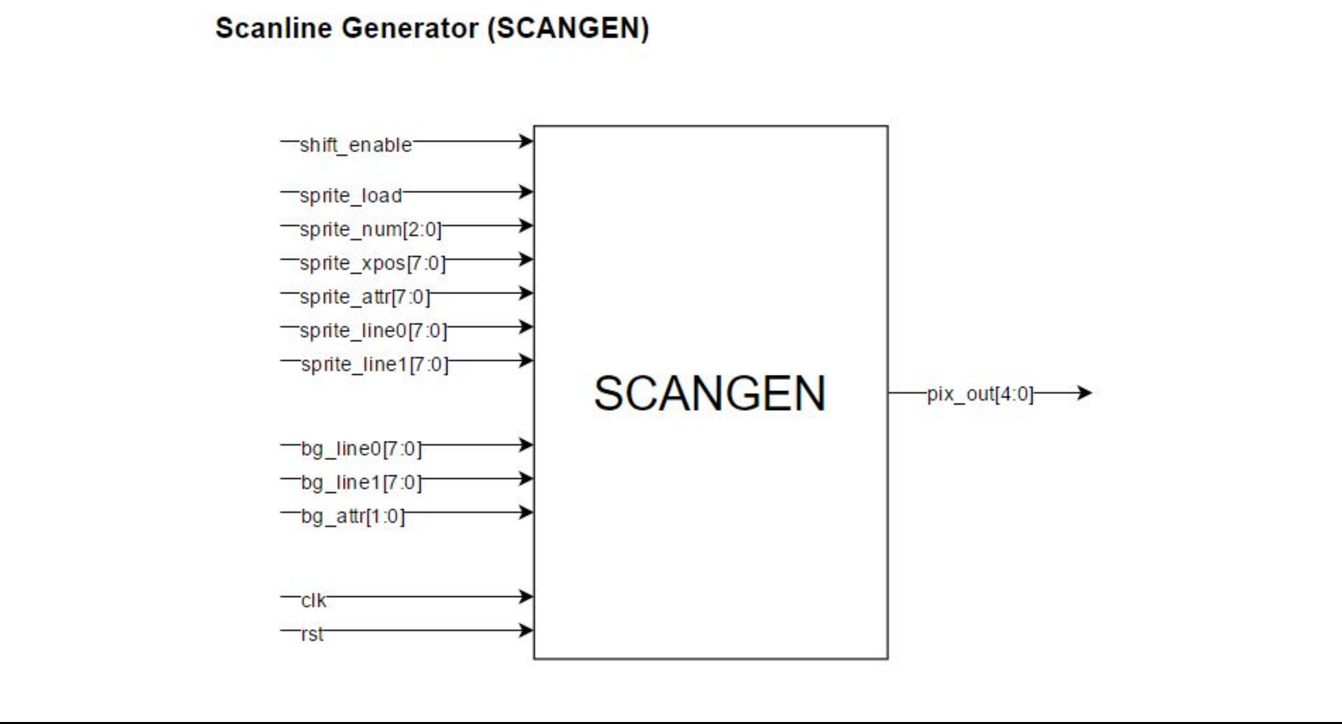


Figure 2.2.5: Scanline Generator

The SCANGEN is split up into two shifting components. One for sprites and one for backgrounds.

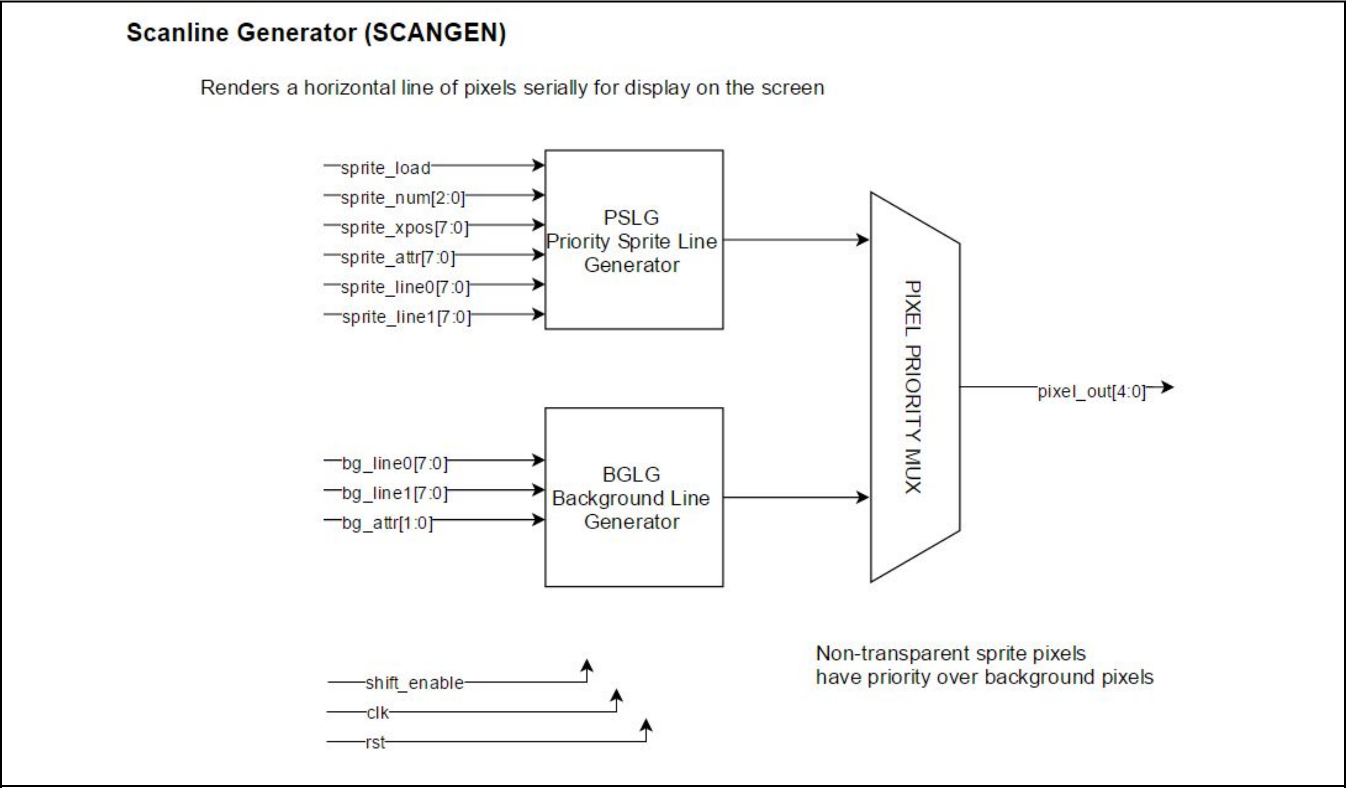

Scanline Generator (SCANGEN)
Renders a horizontal line of pixels serially for display on the screen
sprite_load
sprite_num[2:0]
sprite_xpos[7:0]
sprite_attr[7:0]
sprite_line0[7:0]
sprite_line1[7:0]
PSLG
Priority Sprite Line Generator
bg_line0[7:0]
bg_line1[7:0]
bg_attr[1:0]
BGLG
Background Line Generator
PIXEL PRIORITY MUX
pixel_out[4:0]
shift_enable
clk
rst
Non-transparent sprite pixels have priority over background pixels

Figure 2.2.6: SCANGEN Internals

2.2.5 Priority Sprite Line Generator (PSLG)

The PSLG is loaded with sprite data (maximum of eight sprites) and shifts out pixels corresponding to the sprites. Sprites with lower indices are given priority over other sprites.

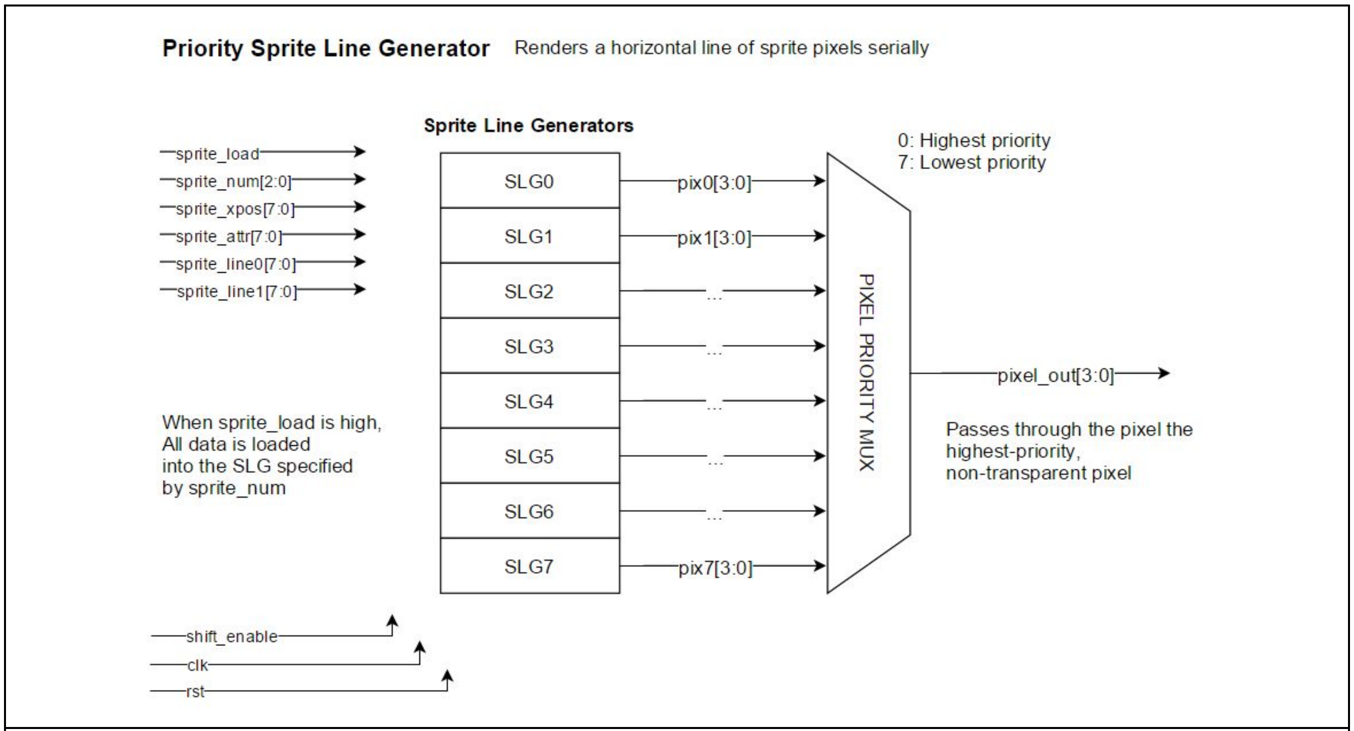


Figure 2.2.7: Priority Sprite Line Generator

2.2.6 Sprite Line Generator (SLG)

Each SLG contains a sprite's x position, its color palette, and loaded tile data. The x position is loaded into a down counter. Each cycle, this down counter is decremented. Once it reaches zero, the sprite's tile data will start shifting out pixels.

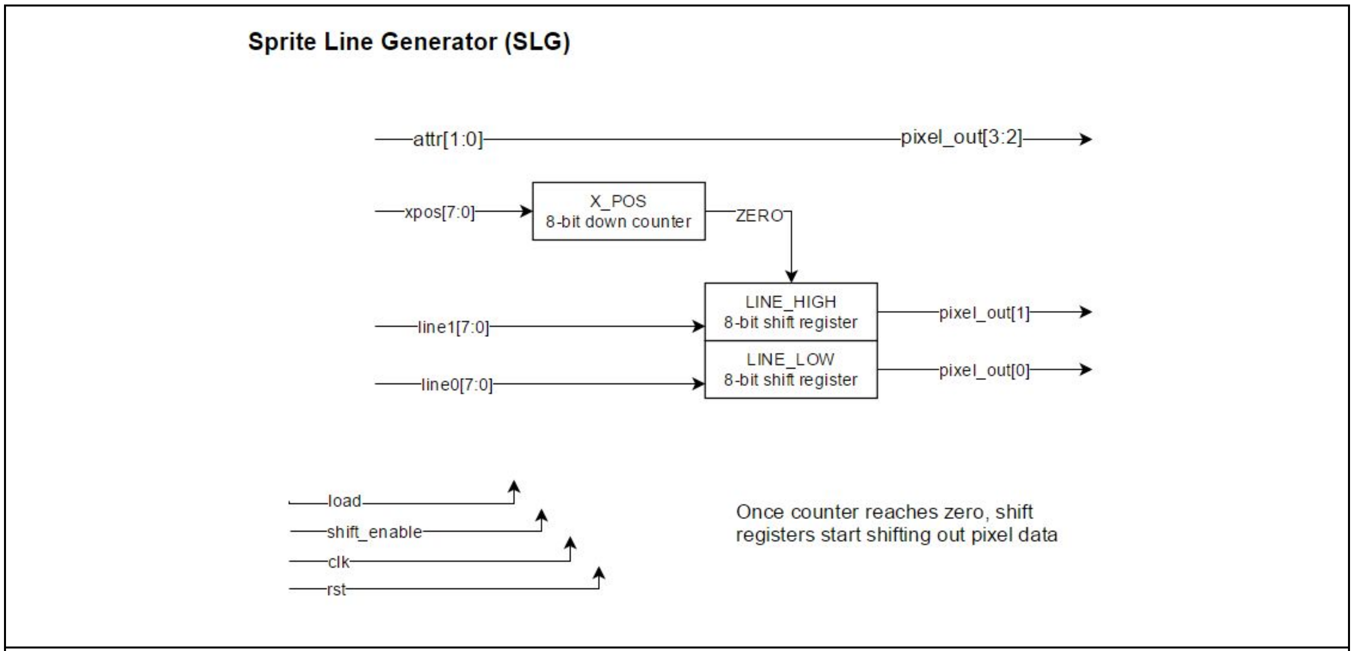
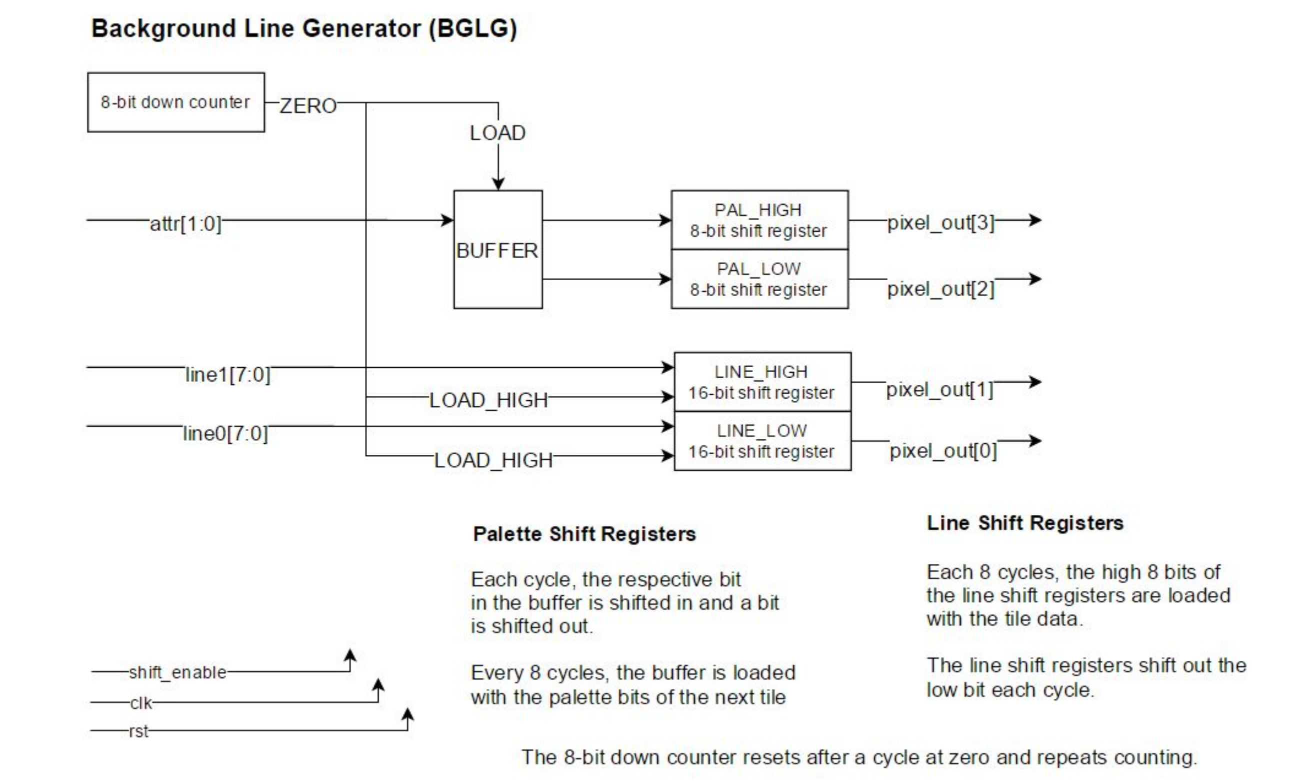


Figure 2.2.8: Sprite Line Generator

2.2.7 Background Line Generator (BGLG)

The BGLG shifts out pixels for background tiles. The BGLG contains tile data for the current background tile and the next background tile. Every 8 shifts, the next tile's data is loaded into the high byte of the shift registers. Also, at this time, the attribute bytes from the background attribute table are loaded into the buffer. These contain the color palette used for the background tile.



Palette Shift Registers

Each cycle, the respective bit in the buffer is shifted in and a bit is shifted out.

Every 8 cycles, the buffer is loaded with the palette bits of the next tile

Line Shift Registers

Each 8 cycles, the high 8 bits of the line shift registers are loaded with the tile data.

The line shift registers shift out the low bit each cycle.

The 8-bit down counter resets after a cycle at zero and repeats counting.

Figure 2.2.9: Background Line Generator

2.2.8 Pattern Tables

Within the PPU, two pattern tables exist. One for sprites and one for backgrounds. Each pattern table stores the actual bitmap data that makes up a sprite tile or background tile. These pattern tables are ROMs set by the game data. The format of these tables is described in the software section.

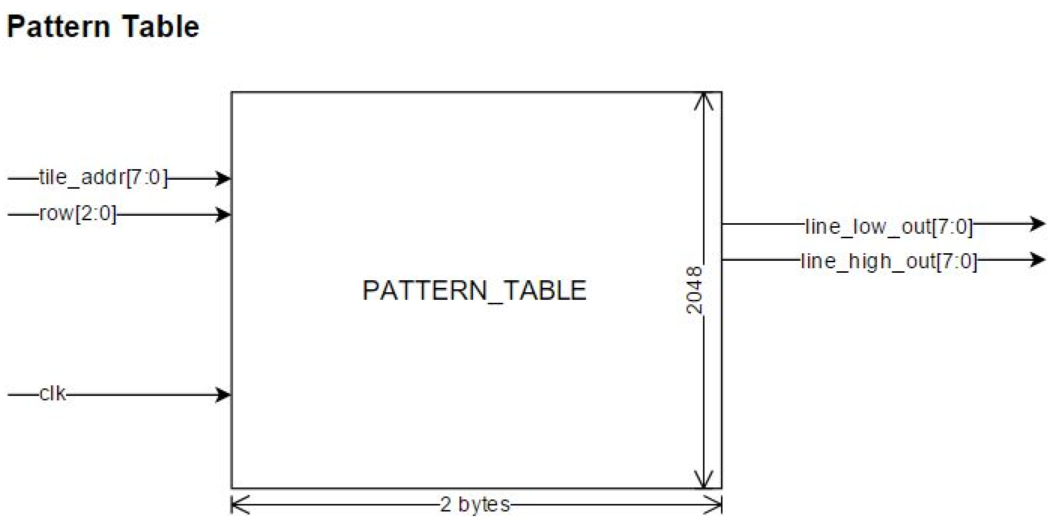
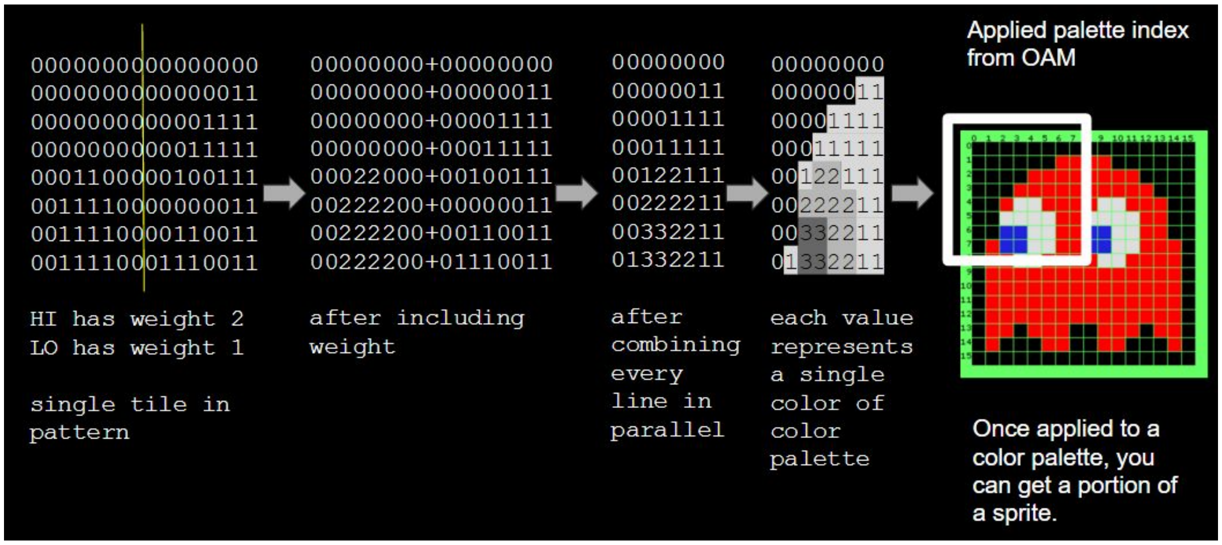


Figure 2.2.10: Pattern Table

The figure below gives an example of how a tile is stored in the Pattern Table and how its format corresponds to colors.



```
0000000000000000   00000000+00000000   00000000   00000000
0000000000000011   00000000+00000011   00000011   00000011
0000000000001111   00000000+00001111   00001111   00001111
0000000000011111   00000000+00011111   00011111   00011111
0001100000100111   00022000+00100111   00122111   00122111
0011110000000011   00222200+00000011   00222211   00222211
0011110000110011   00222200+00110011   00332211   00332211
0011110001110011   00222200+01110011   01332211   01332211
```

HI has weight 2
LO has weight 1
single tile in pattern
after including weight
after combining every line in parallel
each value represents a single color of color palette
Applied palette index from OAM
Once applied to a color palette, you can get a portion of a sprite.

Figure 2.2.11: Pattern Table Tile Example

2.2.9 Background Tile Table

Backgrounds are made up of tiles from the background pattern table. Because each tile is 8x8 pixels, the background is made up of a 32x32 array of tiles. The background tile table holds information on which tile is used for each piece of the background. The table is RAM that is initialized by game data but is writable by the CPU. This module supports concurrent PPU reads and CPU writes.

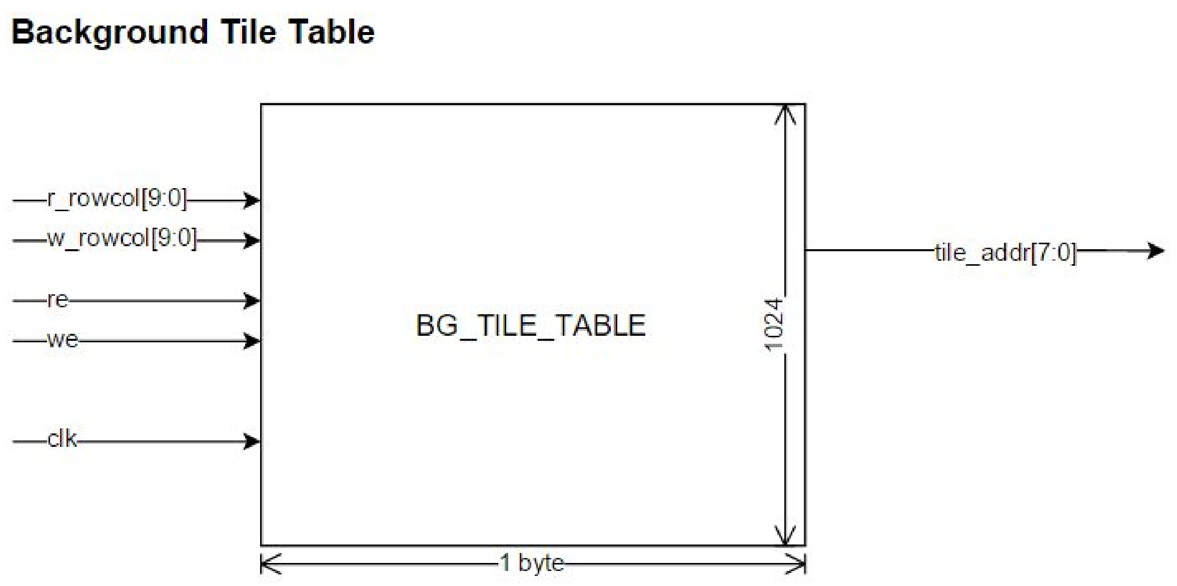


Figure 2.2.12: Background Tile Table

2.2.10 Background Attribute Table

Backgrounds are made up of tiles from the background pattern table. Because each tile is 8x8 pixels, the background is made up of a 32x32 array of tiles. The background attribute table holds information on which color palette is used for each tile that makes up the background. The table is RAM that is initialized by game data but is writable by the CPU. This module supports concurrent reads and writes (Reads by the PPU and writes by the CPU).

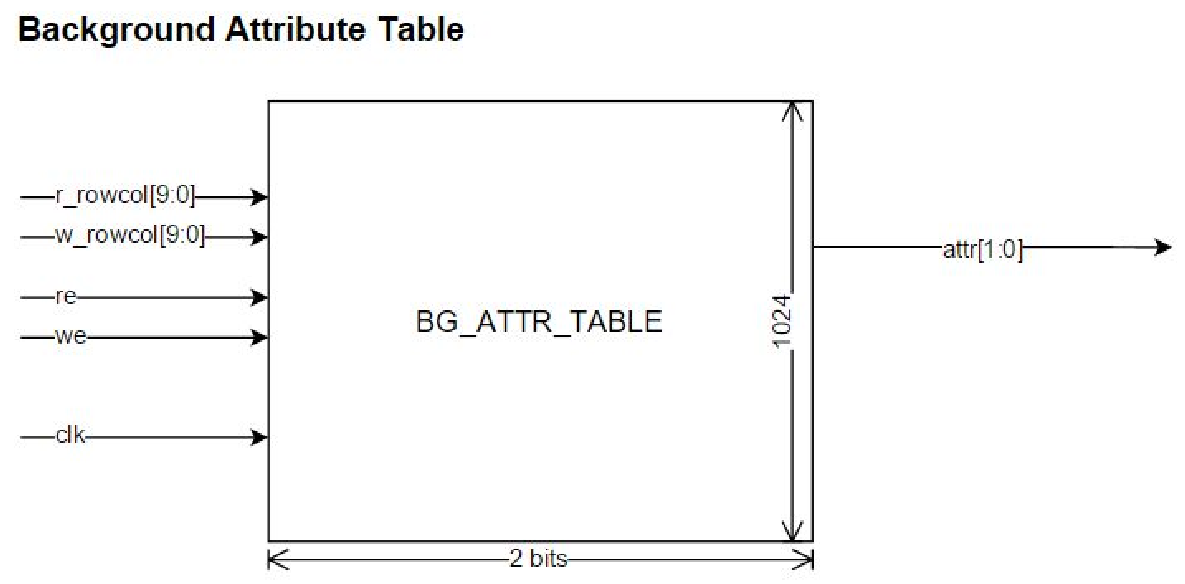


Figure 2.2.13: Background Attribute Table

2.2.11 Color Palettes

There are two ROMs that define the colors used for sprites and backgrounds. There is also a single system palette that cannot be changed.

2.2.12 Sprite and Background Color Palettes

There are a total of 8 color palettes defined by the game data. Four for sprites and four for backgrounds. It is important to note that for each of the sprite palettes, color 0 is irrelevant because the internals always interpret sprite colors of 0 as transparent. Furthermore, the backgrounds color palettes also have special handling for color 0. Regardless of the color palette used, color 0 will be always be read as color 0 of the 0th background color palette. This is referred to as the universal background color. Each color returned by any of the color palettes is actually an address of the color stored in the system color palette.

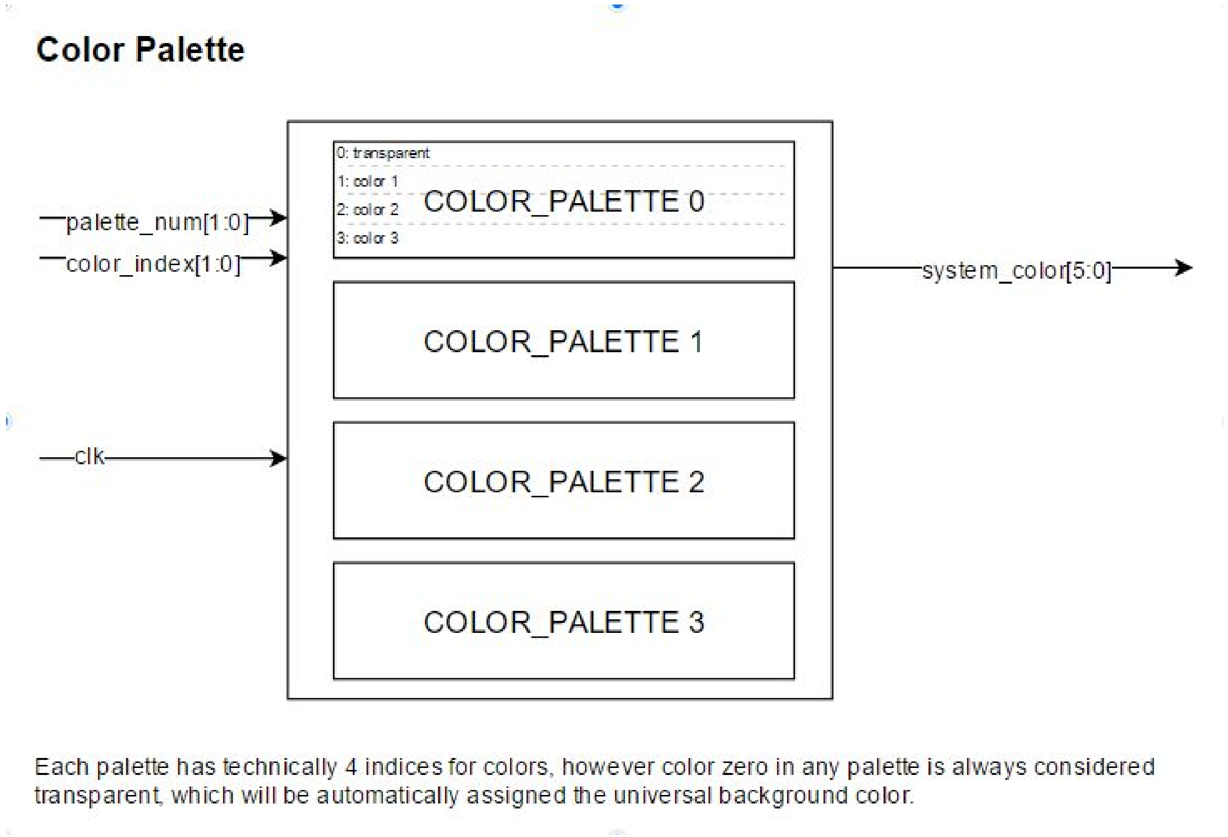
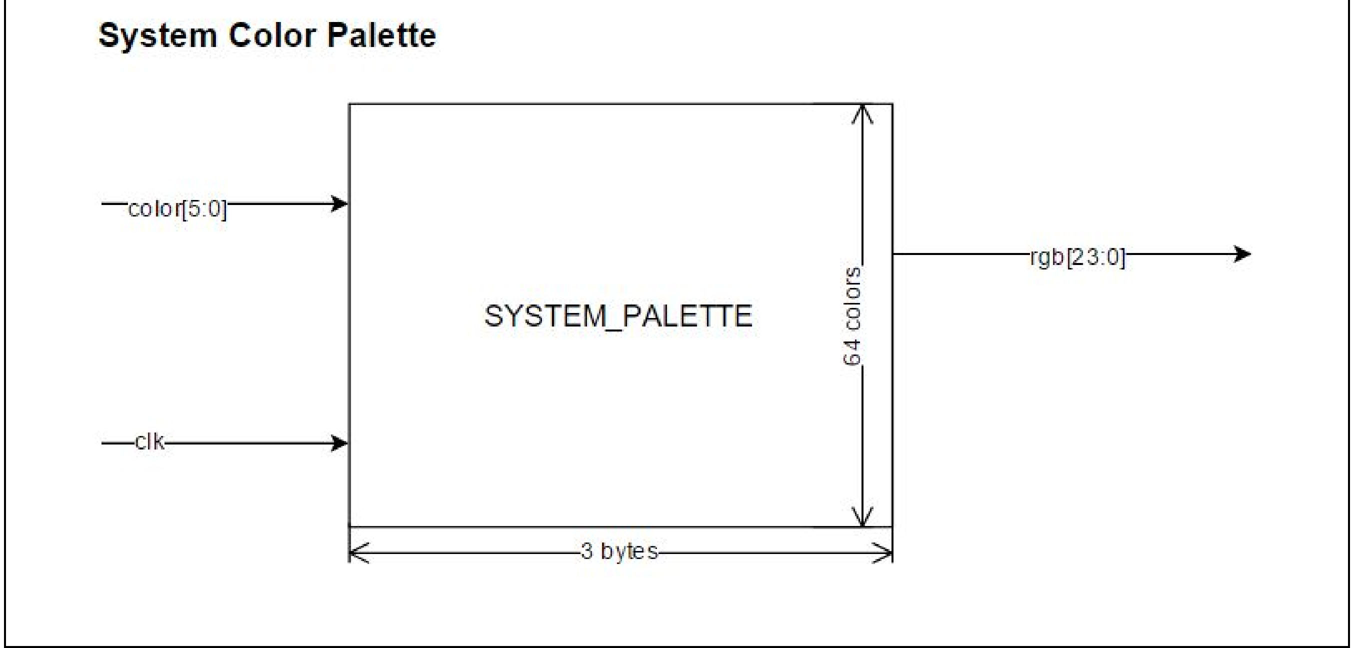
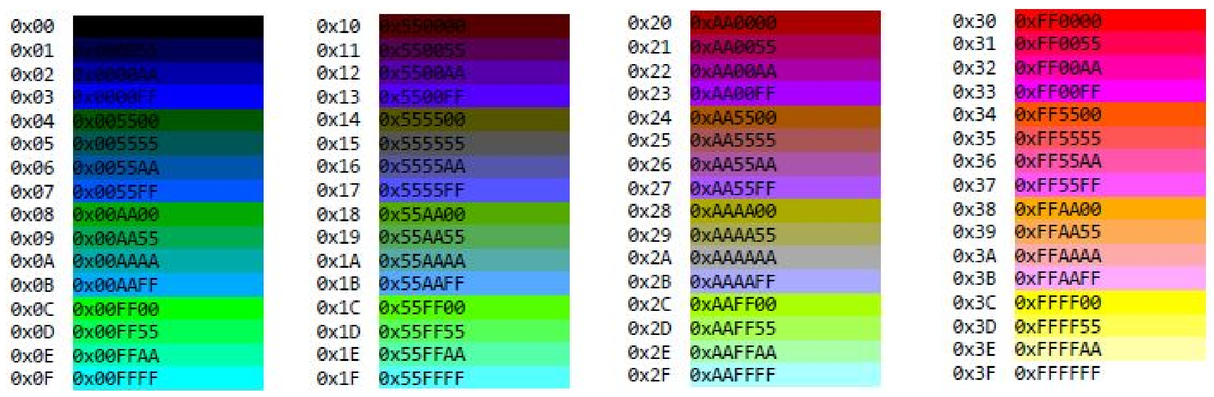

Color Palette
palette_num[1:0]
color_index[1:0]
clk
0: transparent
1: color 1
2: color 2
3: color 3
COLOR_PALETTE 0
COLOR_PALETTE 1
COLOR_PALETTE 2
COLOR_PALETTE 3
system_color[5:0]
Each palette has technically 4 indices for colors, however color zero in any palette is always considered transparent, which will be automatically assigned the universal background color.

Figure 2.2.14: Color Palette

2.2.13 System Color Palette

The system color palette is universal regardless of the game and cannot be changed. The system color palette contains the 64 possible colors that can be used by graphics in the system.





0x00		0x10	0x550000	0x20	0xAA0000	0x30	0xFF0000
0x01	0x000055	0x11	0x550055	0x21	0xAA0055	0x31	0xFF0055
0x02	0x0000AA	0x12	0x5500AA	0x22	0xAA00AA	0x32	0xFF00AA
0x03	0x0000FF	0x13	0x5500FF	0x23	0xAA00FF	0x33	0xFF00FF
0x04	0x005500	0x14	0x555500	0x24	0xAA5500	0x34	0xFF5500
0x05	0x005555	0x15	0x555555	0x25	0xAA5555	0x35	0xFF5555
0x06	0x0055AA	0x16	0x5555AA	0x26	0xAA55AA	0x36	0xFF55AA
0x07	0x0055FF	0x17	0x5555FF	0x27	0xAA55FF	0x37	0xFF55FF
0x08	0x00AA00	0x18	0x55AA00	0x28	0xAAAA00	0x38	0xFFAA00
0x09	0x00AA55	0x19	0x55AA55	0x29	0xAAAA55	0x39	0xFFAA55
0x0A	0x00AAAA	0x1A	0x55AAAA	0x2A	0xAAAAAA	0x3A	0xFFAAAA
0x0B	0x00AAFF	0x1B	0x55AAFF	0x2B	0xAAAAFF	0x3B	0xFFAAFF
0x0C	0x00FF00	0x1C	0x55FF00	0x2C	0xAAFF00	0x3C	0xFFFF00
0x0D	0x00FF55	0x1D	0x55FF55	0x2D	0xAAFF55	0x3D	0xFFFF55
0x0E	0x00FFAA	0x1E	0x55FFAA	0x2E	0xAAFFAA	0x3E	0xFFFFAA
0x0F	0x00FFFF	0x1F	0x55FFFF	0x2F	0xAAFFFF	0x3F	0xFFFFFF

These are all of the 64 colors available to graphics in the system.

Figure 2.2.15: System Color Palette

2.3 Special Purpose Asynchronous Receive/Transmit (SPART)

The SPART is a unit that receives gameplay input data serially from a computer keyboard via an RS232 cable and transmits this data to the CPU for processing. Essentially, it is divided into three submodules, those being a driver, baud rate generator (BRG), and transceiver. Although transmitting received data back out the RS232 cable was not necessary for our design, we kept these transmit capabilities and sent all received input back to the serial terminal. This transmission module allowed us to verify that our system was receiving the correct input on any given transaction.

The driver is responsible for coordinating the receival and transmission of data from the input keyboard to the CPU. A tri-state bus interface is the connection between this driver and the SPART core (BRG/transceiver). This interface is responsible for sending data into the core for transmission or baud rate controlling, as well as receiving data from the core corresponding to the character byte pressed by the user on the connected keyboard.

The BRG controls enable signals to the transceiver’s control units. It loads a counter with a 16-bit countdown value sent from the driver that corresponds to a supported baud rate (either 4800, 9600, 19200, or 38400). When the counter reaches zero, an enable signal is sent to the transceiver to transmit or capture data on the serial line. The serial line operates at a frequency much slower than the internal clock signal of the SPART, so each serial bit is sent at a specific multiple of the internal clock frequency. Similarly, the line is sampled at a multiple of the internal clock frequency specific to the baud rate.

The transceiver’s transmit control unit takes data from the bus interface and sends it out the RS232 serial line. In order to accomplish this, it must convert data from parallel to serial by shifting bits of the parallel data out the RS232 at a specified rate. Likewise, the transceiver’s receive control unit takes data from the serial port and shifts it into a register, thus converting serial data to parallel. This shifting corresponds to the frequency of the baud rate generator’s enable signals, which ensures that the data is transferred at the rate specified for the serial line.

Figure 2.3(a) shows the SPART environment of receiving serial I/O from RS232 to the processor. Figure 2.3(b) shows the SPART diagram of transmitting and receiving data and passing it to/from the processor via the databus.

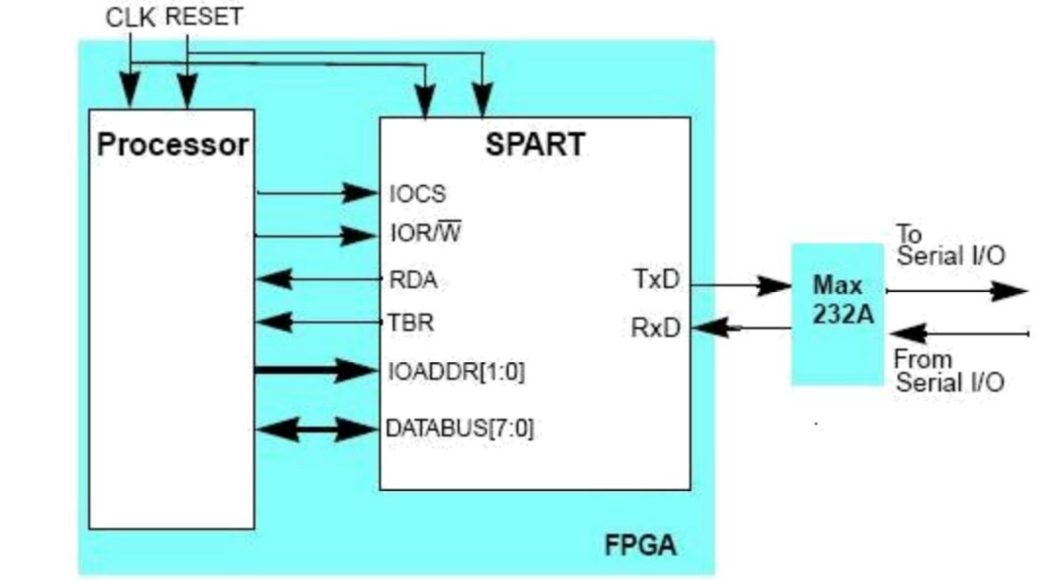


Figure 2.3(a): SPART Environment

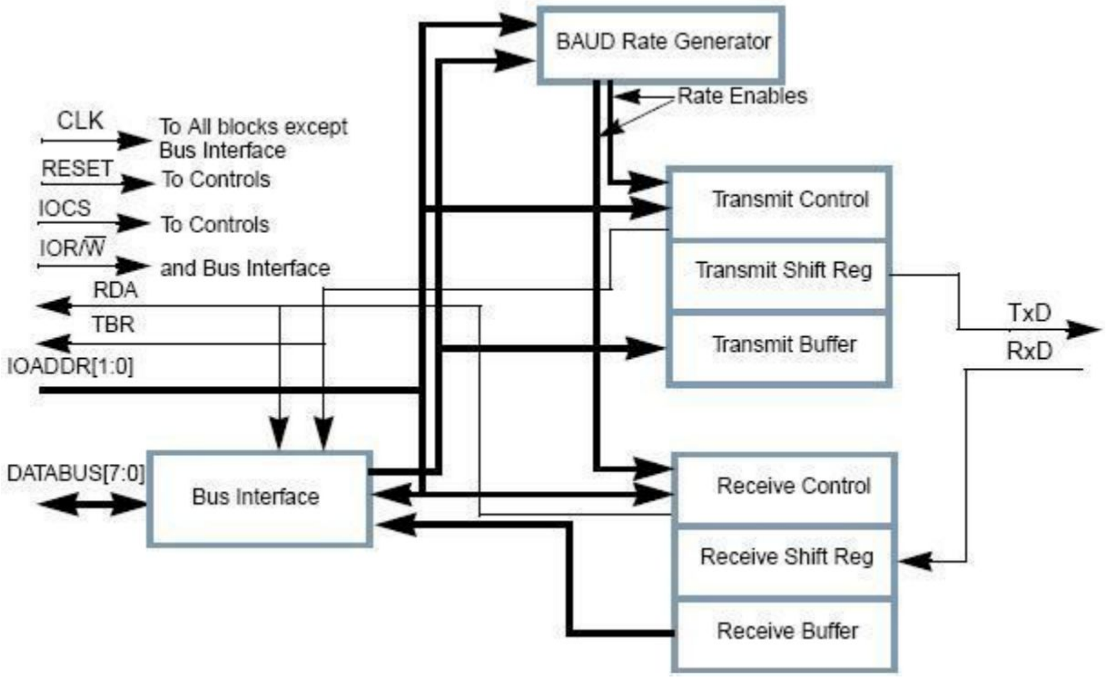


Figure 2.3(b): SPART Block Diagram

2.4 Central Processing Unit

At the heart of the Tronsistor-32 is a 32-bit RISC architecture, 5-stage pipelined processor including 32 special and general purpose registers as well as data forwarding and interrupt capabilities. We will discuss these aspects in more depth in the following sections. Our CPU recognizes 30 valid instructions, some of which (11 in total) have been developed specifically for our system and the rest have been migrated from the MIPS32 ISA. For reference, the blue lines in the image below are control signals.

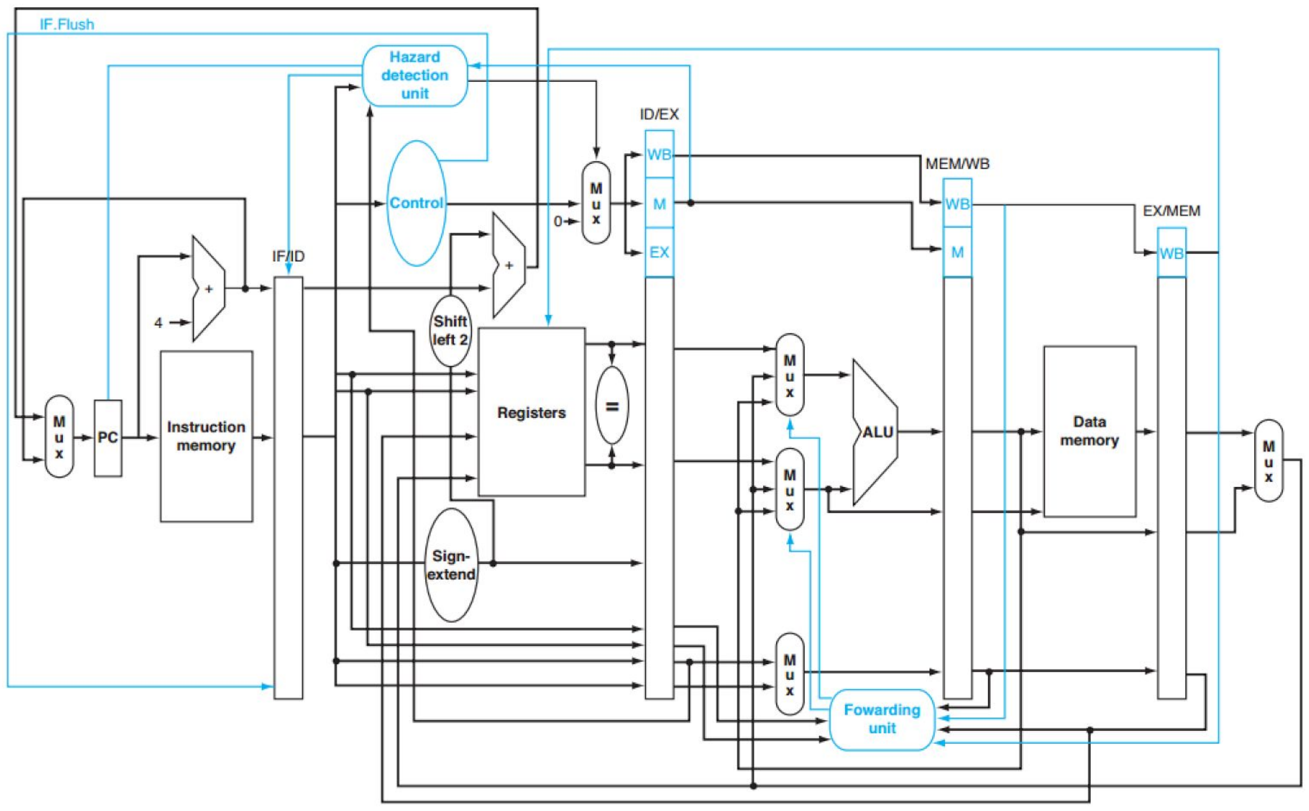


Figure 2.4.0: Relative Top Level Layout of Tronsistor-32 CPU

2.4.1 Registers

Register Number	Software Alias	Functional Description
0	$zero	Zero: This register is hardwired to supply the constant value of 0x00000000. This register is unwriteable, and is often used as the destination of an arithmetic instruction for setting branch conditions without overwriting another register value.
1	$at	Assembler Temporary: General purpose register.
2-3	$v0-$v1	Function Results: The results of function calls may be placed into these registers which circumvents the need to return them via the stack. However, they may be considered general purpose registers.
4-7	$a0-$a3	Arguments: Function parameters may be placed into these registers to avoid passing these values via the stack. However, they may be considered general purpose registers.
8-15	$t0-$t7	Temporaries: General purpose registers.
16-23	$s0-$s7	Saved Temporaries: General purpose registers.
24-26	$t8-$t10	Temporaries: General purpose registers.
27	$au	Audio: This register is reserved to hold the starting location of an audio file’s data.
28	$idr	Interrupt Data Register: This register saves the data retrieved from a keyboard interrupt for game player input verification.
29	$sp	Stack Pointer: This register always points to the top of the stack’s address space, which is fully descending. The system automatically updates the contents of this register while executing any instructions that access the stack’s data.
30	$epc	Exception Program Counter: This register holds the program counter of the instruction that was interrupted while within the Instruction Fetch stage of the pipeline. This is to ensure that the program is able to return to its normal execution after an interrupt has occurred.
31	$ra	Return Address: This register holds the return address placed on top of the stack by a function call.

2.4.2 Addressing Modes

Our ISA supports four addressing modes: immediate, register, label, and label with immediate offset. Immediate addressing requires an immediate value. Register addressing uses values inside of registers. Label addressing uses the program counter data at a particular label.

2.4.2.1 Immediate

The immediate addressing mode passes the immediate value as part of the instruction. This immediate value will be used as a direct input into the ALU for arithmetic. Immediate addressing mode is signified by a numeric value. The default immediate values are in base 10, leaving it up to the assembler to convert this number into binary machine code.

Syntax: [numeric]

Example: li $t0, 48 # load 48, ASCII value of '0', into register $t0

Encoding:

opcode [31:26]	rd [25:21]	rs [20:16]	immediate [20:0]

2.4.2.2 Register

For the register addressing mode, the numeric register (0 - 31) is passed in as part of the instruction. This target register is then passed to the register file, which then outputs the contents of the specified register. Register addressing mode is signified by the $ symbols.

Syntax: $[numeric]

Example: add $s0, $s1, $s2 # add $s1 and $s2, and store the results in $s0

Encoding:

opcode [31:26]	rs [25:21]	rt [20:16]	rd [15:11]	shamt [10:6]	funct [0:5]

2.4.2.3 Label (with Immediate Offset)

With label plus offset, the assembler will map the label to an immediate value, which gets passed in as part of the instruction. Then, the assembler adds the two immediate values together. Label addressing modes are signified by labels. Offsets are signified by appending a comma after the label.

Syntax 1:	[label]	# label addressing only
Syntax 2:	[label], [numeric]	# label plus immediate offset addressing
Example 1:	lb $t0, foreground	# load value that is labeled as foreground in memory
Example 2:	sw $t1, score, 4	# store value of $t1 in 4 words after the label score

Encoding:

opcode [31:26]	rs [25:21]	immediate [20:0]

2.4.3 Instruction Formats

Our ISA supports five instruction formats: register, immediate, jump, sprite, and sprite_remove. Register format is for instructions that uses registers such as add or subtract (sub). Immediate format is for instruction that uses immediates such as add immediate (addi). Jump format is used for branching instructions such as branch less than (blt). Sprite format is used for almost all sprite instructions to update visual data, such as set foreground tile (sft). Lastly, sprite_remove format is only used for the sprite remove (srm) instruction because that instruction takes only 1 register to index into the OAM.

Instruction Formats						
R	opcode [31:26]	rd [25:21]	rs [20:16]	rt [15:11]	shamt [10:6]	funct [5:0]
I	opcode [31:26]	rd [25:21]	rs [20:16]	immediate [15:0]		
J	opcode [31:26]	immediate [25:0]				
S_REMV	opcode [31:26]	rd [25:21] (OAM index)	not used [20:0]			
S	opcode [31:26]	rd [25:21] (OAM index)	rs [20:16] (LSBs)	not used [15:0]		

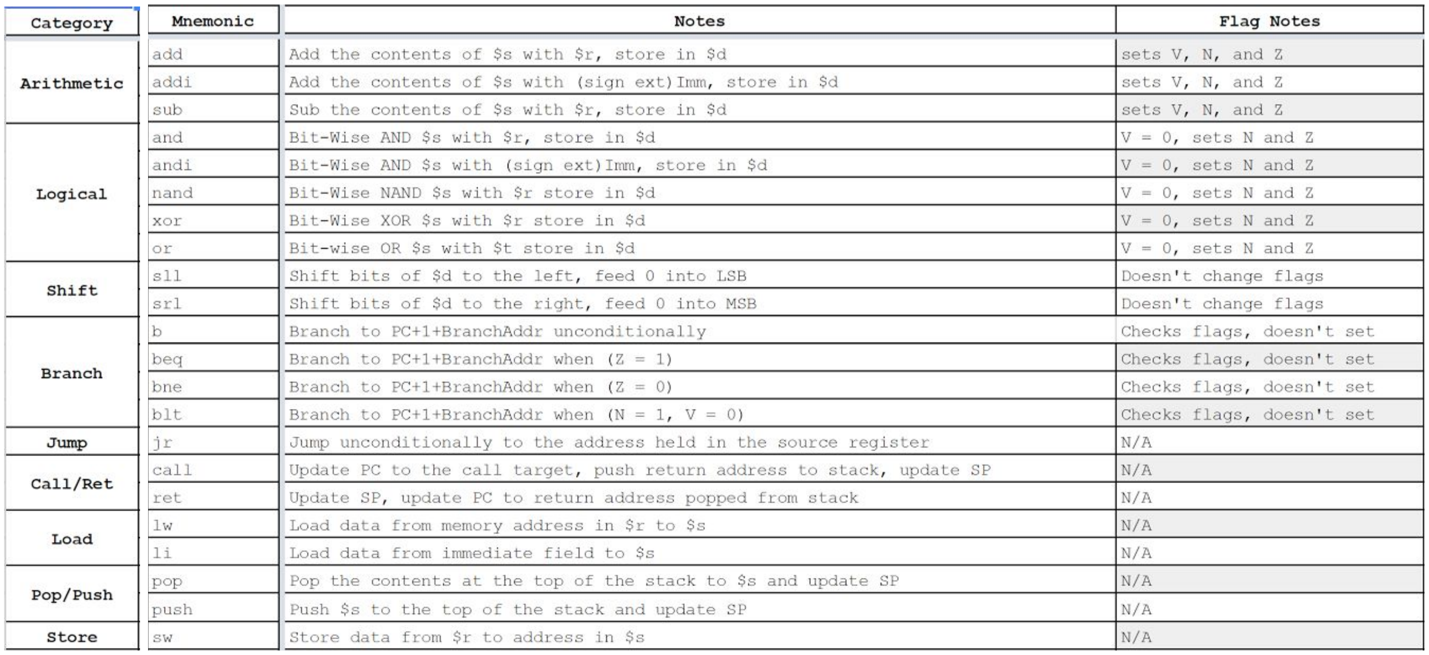
2.4.4 TronMIPStor ISA

Below is a table containing the 30 instructions supported by the Tronsistor-32 with specified instructions formats, software mnemonics, and functional descriptions.

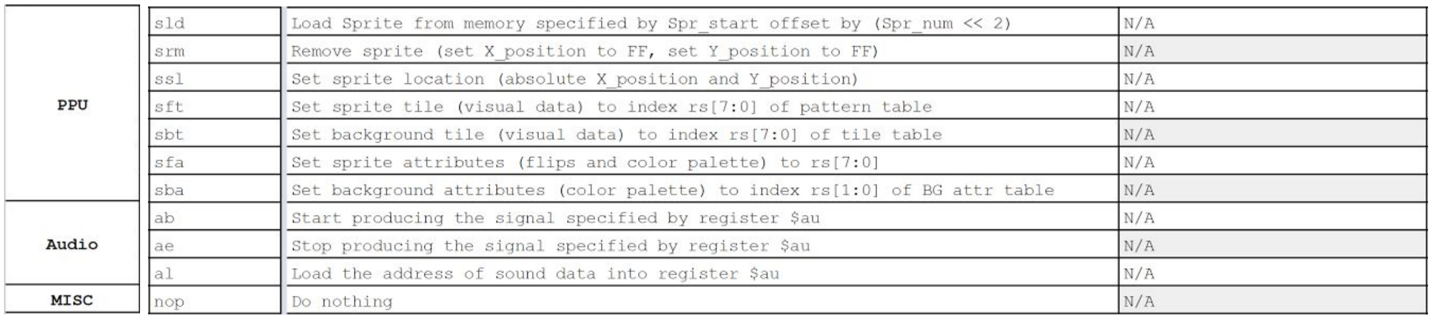
Category	Name	Mnemonic	Format	Operation
Arithmetic	Add	add	R	$d = $s + $t
	Add Imm.	addi	I	$d = $s + (sign ext)Imm
	Subtract	sub	R	$d = $s - $t
Logical	And	and	R	$d = $s & $t
	And Imm.	andi	I	$d = $s & (sign ext)Imm
	Nand	nand	R	$d = $s ~& $t
	Xor	xor	R	$d = $s ^ $t
	Or	or	R	$d = $s | $t
Shift	Shift Left Logical	sll	R	$d = $s << shamt
	Shift Right Logical	srl	R	$d = $s >> shamt
Branch	Branch (unconditional)	b	J	PC=PC+1+BranchAddr
	Branch On Equal	beq	J	if Z = 1, PC=PC+1+BranchAddr
	Branch On Not Equal	bne	J	if Z = 0, PC=PC+1+BranchAddr
	Branch Less Than	blt	J	if N = 1 and V = 0, PC=PC+1+BranchAddr
Jump	Jump Register	jr	I	Jump Register
Call/Ret	Call	call	J	PC = {PC[31..26], Imm}; M[SP] = PC+1; SP = SP-1
	Return	ret	J	SP = SP+1; PC = M[SP]
Load	Load Word	lw	I	$d = M[rs + (sign ext)Imm]
	Load Immediate	li	I	$d = (sign ext) Imm
Pop/Push	Pop	pop	I	SP = SP+1; rd = M[SP];
	Push	push	I	M[SP] = $d; SP = SP-1
Store	Store Word	sw	I	M[rs + (sign ext)Imm] = rd

PPU	Sprite Load	sld	S	OAM[rd[5:0]] = rs
	Sprite Remove	srm	S_REMV	OAM[rd[5:0]] = 0xFFFFFFFF
	Sprite Set Location	ssl	S	OAM[rd[5:0]] = {rs[15:8], 16'hFFFF, rs[7:0]}
	Set Foreground Tile Number	sft	S	OAM[rd[5:0]] = rs[7:0]
	Set Background Tile Number	sbt	S	BGTT[rd] = rs[7:0]
	Set Foreground Attributes	sfa	S	OAM[rd[5:0]][15:8] = rs[7:0]
	Set Background Attributes	sba	S	BGAT[rd[9:0]] = rs[1:0]
Audio	Audio Begin	ab	R	
	Audio End	ae	R	
	Audio Load	al	R	
MISC	Null Operation	nop	R	null

Additional Notes:



Category	Mnemonic	Notes	Flag Notes
Arithmetic	add	Add the contents of $s with $r, store in $d	sets V, N, and Z
	addi	Add the contents of $s with (sign ext)Imm, store in $d	sets V, N, and Z
	sub	Sub the contents of $s with $r, store in $d	sets V, N, and Z
Logical	and	Bit-Wise AND $s with $r, store in $d	V = 0, sets N and Z
	andi	Bit-Wise AND $s with (sign ext)Imm, store in $d	V = 0, sets N and Z
	nand	Bit-Wise NAND $s with $r store in $d	V = 0, sets N and Z
	xor	Bit-Wise XOR $s with $r store in $d	V = 0, sets N and Z
	or	Bit-wise OR $s with $t store in $d	V = 0, sets N and Z
Shift	sll	Shift bits of $d to the left, feed 0 into LSB	Doesn't change flags
	srl	Shift bits of $d to the right, feed 0 into MSB	Doesn't change flags
Branch	b	Branch to PC+1+BranchAddr unconditionally	Checks flags, doesn't set
	beq	Branch to PC+1+BranchAddr when (Z = 1)	Checks flags, doesn't set
	bne	Branch to PC+1+BranchAddr when (Z = 0)	Checks flags, doesn't set
	blt	Branch to PC+1+BranchAddr when (N = 1, V = 0)	Checks flags, doesn't set
Jump	jr	Jump unconditionally to the address held in the source register	N/A
Call/Ret	call	Update PC to the call target, push return address to stack, update SP	N/A
	ret	Update SP, update PC to return address popped from stack	N/A
Load	lw	Load data from memory address in $r to $s	N/A
	li	Load data from immediate field to $s	N/A
Pop/Push	pop	Pop the contents at the top of the stack to $s and update SP	N/A
	push	Push $s to the top of the stack and update SP	N/A
Store	sw	Store data from $r to address in $s	N/A



PPU	sld	Load Sprite from memory specified by Spr_start offset by (Spr_num << 2)	N/A
	srm	Remove sprite (set X_position to FF, set Y_position to FF)	N/A
	ssl	Set sprite location (absolute X_position and Y_position)	N/A
	sft	Set sprite tile (visual data) to index rs[7:0] of pattern table	N/A
	sbt	Set background tile (visual data) to index rs[7:0] of tile table	N/A
	sfa	Set sprite attributes (flips and color palette) to rs[7:0]	N/A
	sba	Set background attributes (color palette) to index rs[1:0] of BG attr table	N/A
Audio	ab	Start producing the signal specified by register $au	N/A
	ae	Stop producing the signal specified by register $au	N/A
	al	Load the address of sound data into register $au	N/A
MISC	nop	Do nothing	N/A

3 System Microarchitecture

3.1 Central Processing Unit (CPU)

Figure 3.1 contains a diagram of the Tronsistor-32 pipelined CPU's microarchitecture. The CPU's Instruction Fetch module fetches all instructions contained within a program via a main memory interface. However, the remaining CPU pipeline sections are only responsible for executing all non-sprite instructions (R-type, I-type, J-type). Sprite instructions (S_REMV, and S-type) are passed out of the IFID pipeline register to the Picture Processing Unit (**Section 3.2**) for further execution. The following subsections (**Section 3.1.1 - 3.1.6**) will discuss the internal microarchitecture design of the five main pipeline stages shown in below.

Note: The signals contained within each pipeline section used for data-forwarding have been omitted due to their complexity. There are two multiplexers located on the output of the Instruction Decode's inputs **ALU_input_1** and **ALU_input_2**, the Instruction Execute's inputs **ALU_in1** and **ALU_in2**, and the Instruction Execute's outputs **EX_out** and **MemWrite_data**. There is one multiplexer per signal, and the select signals are set within the DataForward module.

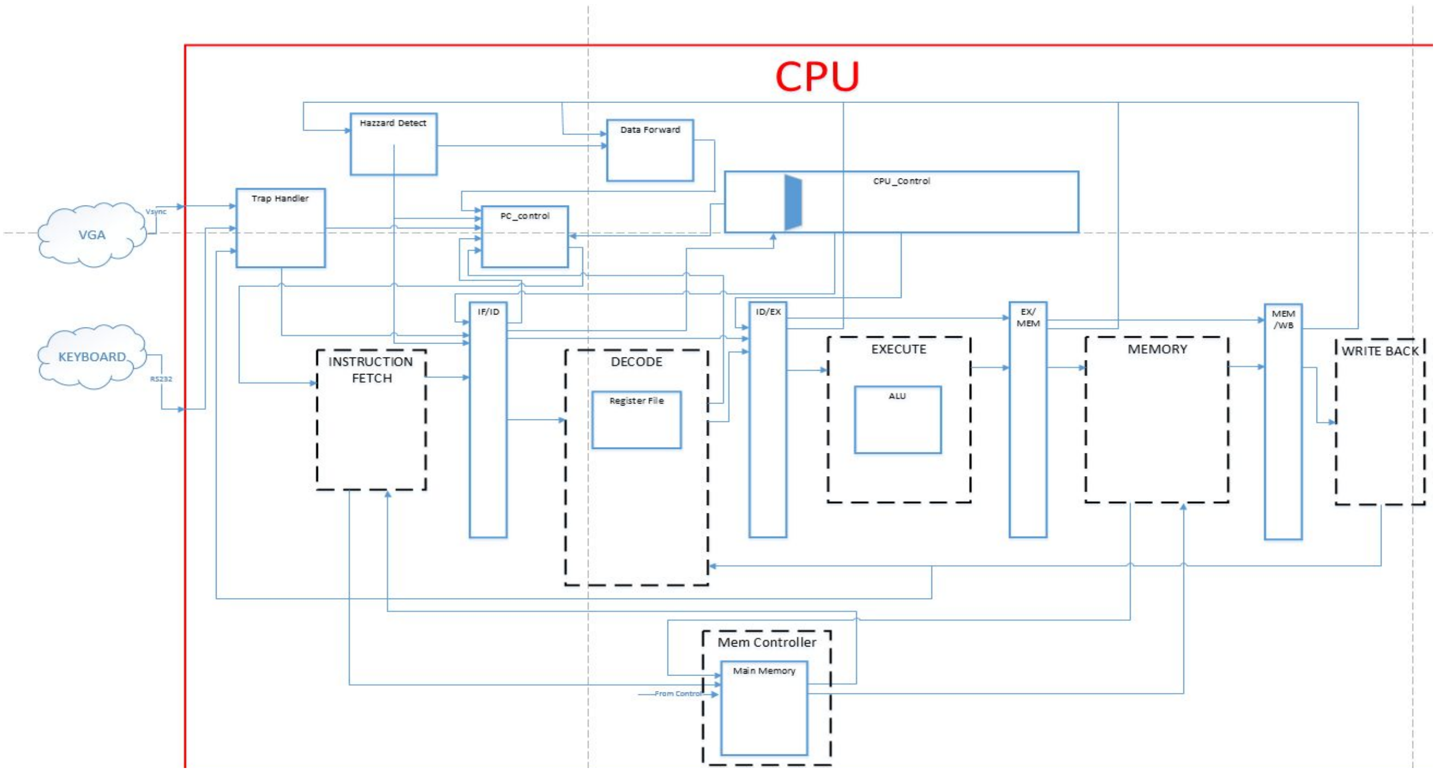


Figure 3.1: Tronsistor-32 pipelined CPU

3.1.1 Instruction Fetch

Figure 3.1.1 contains a detailed view within the Instruction Fetch section of the CPU pipeline. This section of the pipe is responsible for handling updates to the program counter as well as retrieving instructions from memory to pass into the Instruction Decode pipeline stage.

Note: The instruction memory is not actually contained within this section of the pipe. The memory unit is shown as a representation of how this pipeline section will interface with the specified section of main memory. Also, it is important to note that during an interrupt this stage of the pipeline will issue a 'load immediate' instruction that will save the interrupted program counter to the EPC register. After an interrupt has been handled, the trap handler notifies this stage to issue a 'jump register' instruction to load the program counter from the EPC register to proceed with execution. The components within this stage of the pipeline responsible for this activity have been omitted due to their complicated design.

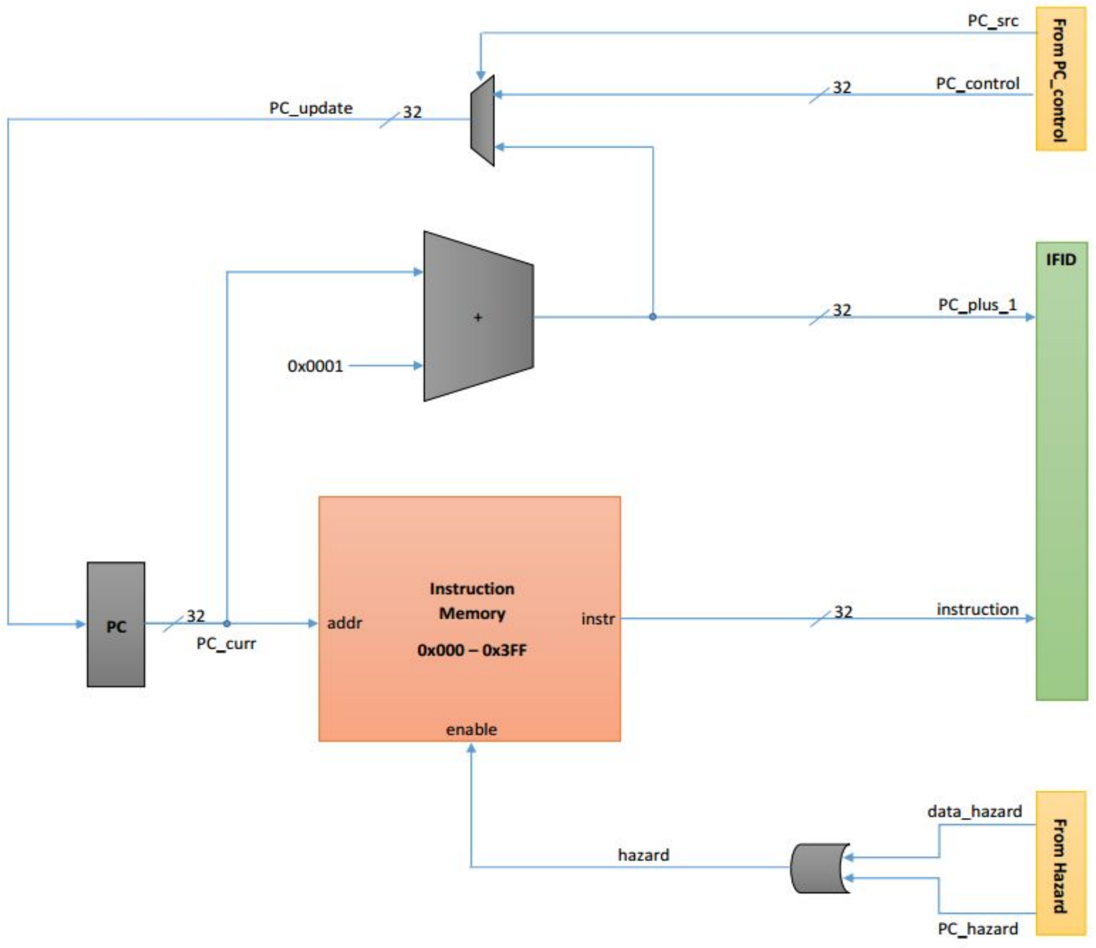


Figure 3.1.1: Instruction Fetch CPU pipeline section

3.1.2 Instruction Decode

Figure 3.1.2 contains a detailed view within the Instruction Decode section of the CPU pipeline. This section of the pipeline is responsible for accessing and modifying the contents within the 32 general and special purpose registers within the Tronsistor-32 (all contained within the register file). Also, this pipeline stage is responsible for separating out each data field from the various instruction types, which is crucial for preparing all relevant data for manipulation within the Instruction Execute stage of the pipe.

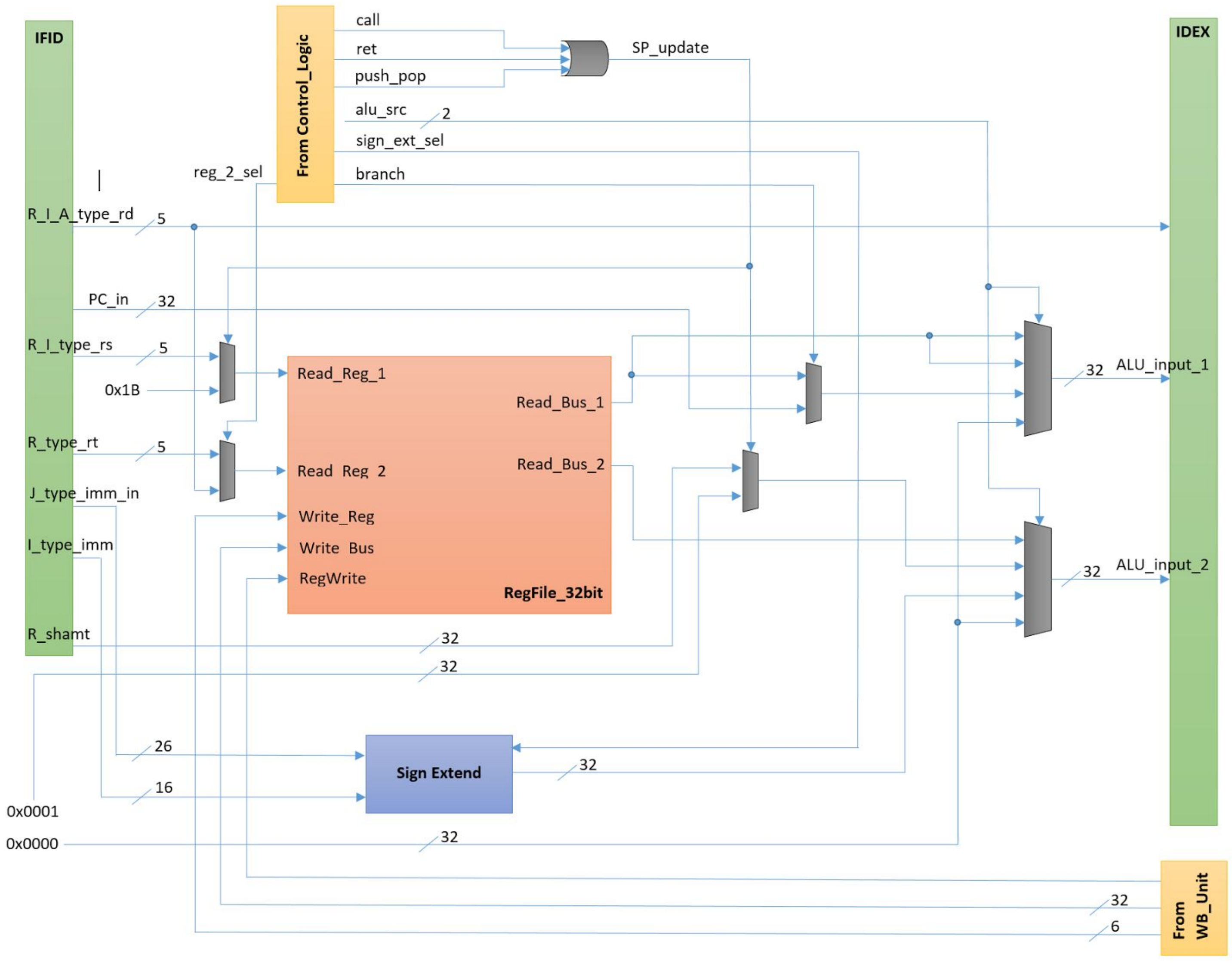


Figure 3.1.2: Instruction Decode CPU pipeline section

3.1.3 Instruction Execute

Figure 3.1.3 contains a detailed view within the Instruction Execute section of the CPU pipeline. This section of the pipeline is responsible for computing the result of all arithmetic and logic instructions (ADD, ADDI, SUB, AND, ANDI, NAND, OR, XOR, SLL, SRL) via the Arithmetic Logic Unit (ALU). The ALU computes the output of the aforementioned instructions and sets the flags that are used to verify branch conditions. This section is also responsible for selecting the proper data for accessing memory in the Memory Interface section of the pipeline.

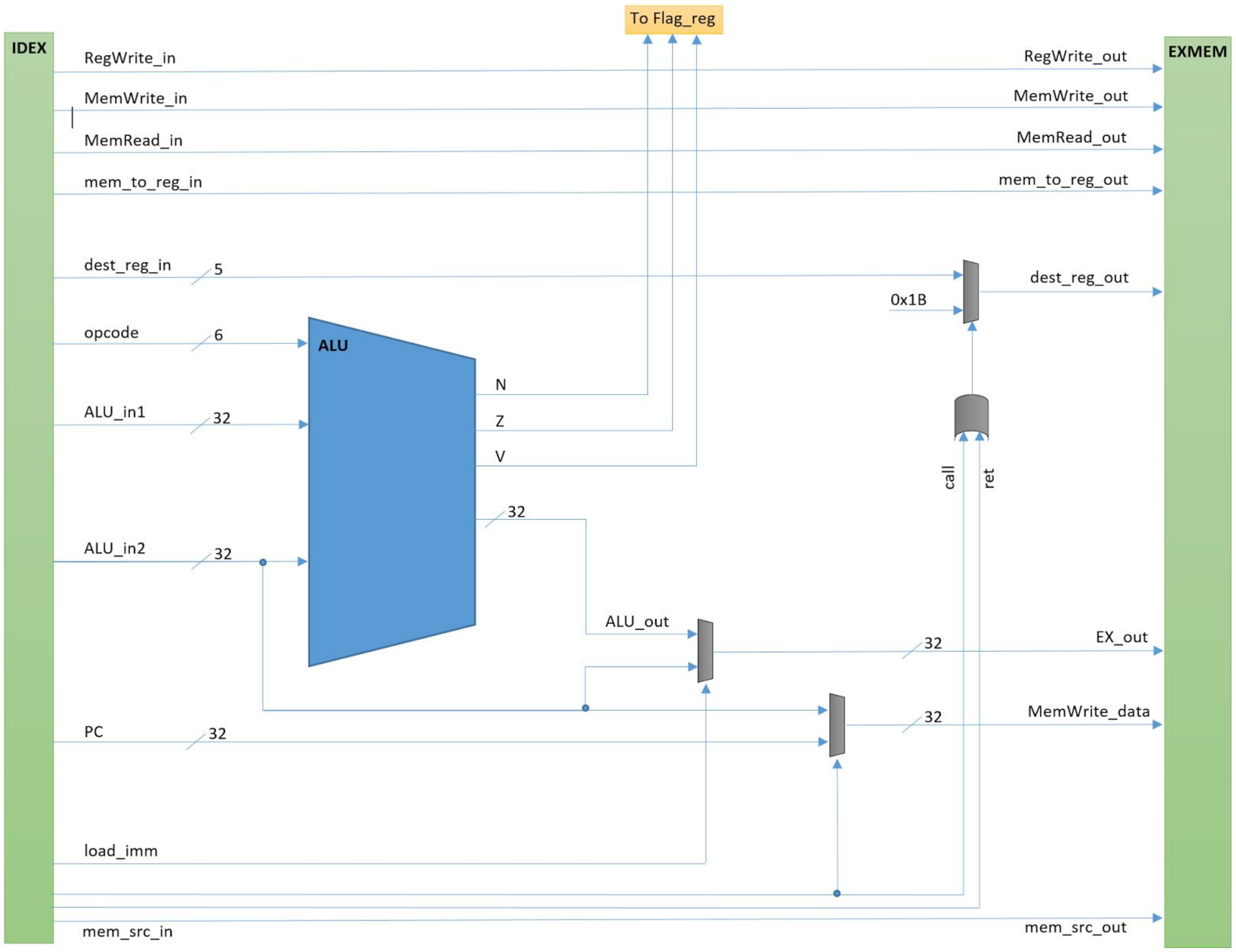


Figure 3.1.3: Instruction Execute CPU pipeline section

3.1.4 Memory Interface

Figure 3.1.4 contains a detailed view within the Memory Interface section of the CPU pipeline. This section of the pipeline is responsible for accomplishing the data memory accesses required by various instructions (SW, PUSH, LW, POP, CALL, RET).

Note: The data memory is not actually contained within this section of the pipe. The memory unit shown below is a representation of how this pipeline section will interface with the specified section of main memory.

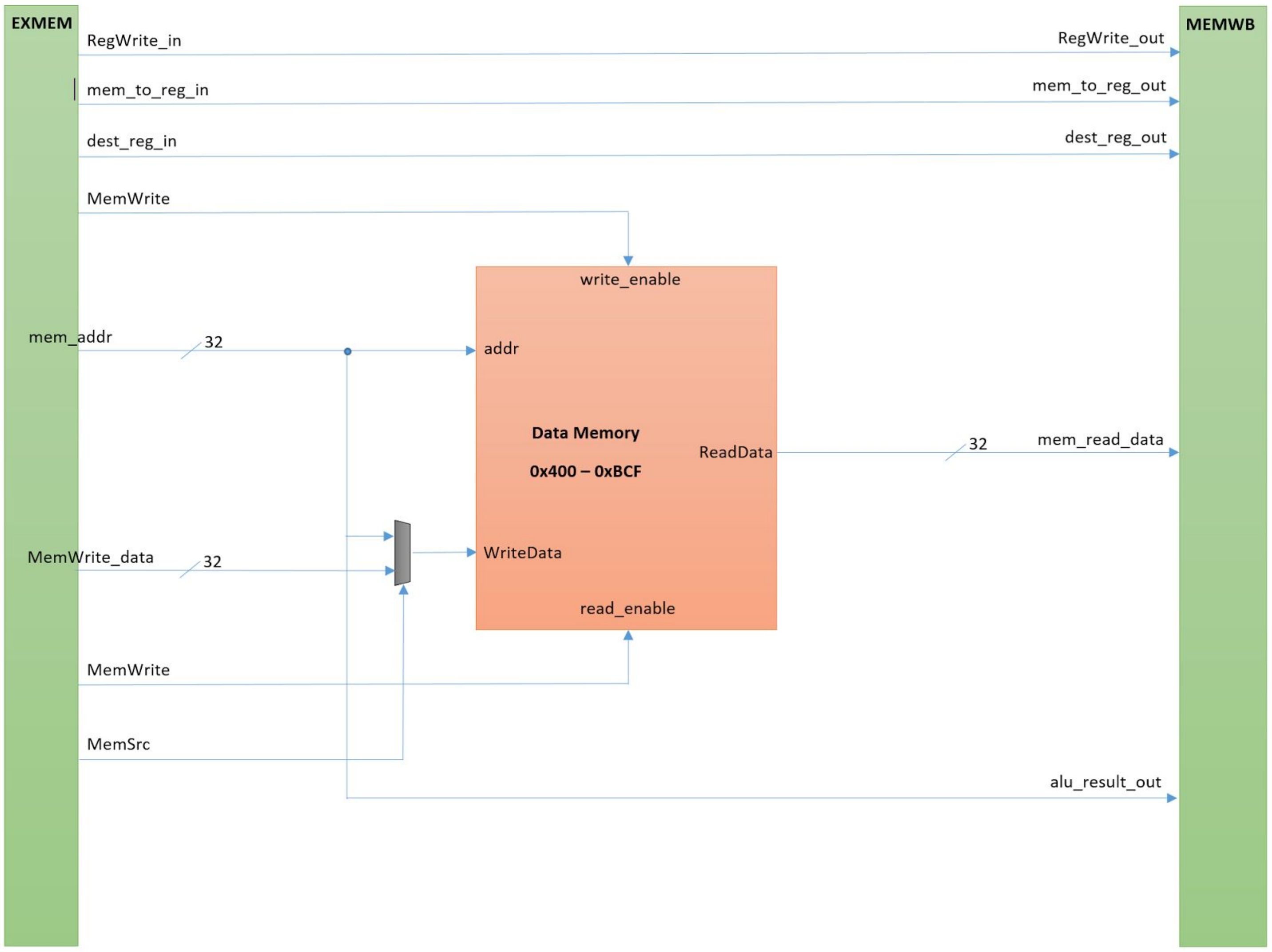


Figure 3.1.4: Memory Interface CPU pipeline section

3.1.5 Result Writeback

Figure 3.1.5 contains a detailed view within the Result Writeback section of the CPU pipeline. This section of the pipeline simply returns the results of each instruction to an appropriate destination.

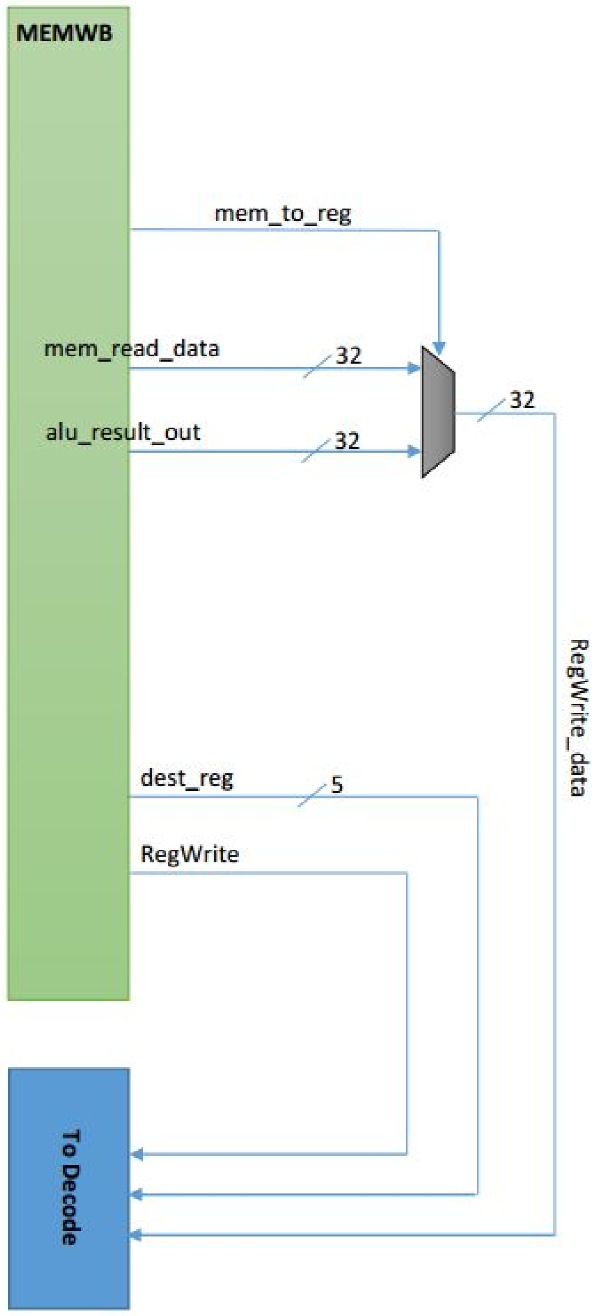


Figure 3.1.5: Result Writeback CPU pipeline section

3.1.6 Handling Asynchronous Input for Interrupts

3.1.6.1 Trap Handler

The Trap Handler is a module that resides outside the CPU and accepts several inputs from the vga_logic and the SPART. The Trap Handler gives two interrupt signals to the CPU, these interrupts are keyboard, and game tick interrupt. These interrupts are described in the following section. The Trap Handlers main responsibility is to take in signals from the SPART and vga_logic that identify an interrupt, and schedule the interrupt signals to be asserted only after all previous hazards have been handled within the CPU. Once interrupts can be handled within the CPU, the Trap Handler uses an FSM to schedule various control signals at appropriate times.

The SPART takes in data serially from the keyboard, and sends an interrupt signal to the CPU after a full byte has been received along with the byte of data that has been received. Vga_logic sends the Vsync signal to the trap handler as the game tick interrupt. This is done so that every time the screen refreshes the CPU can account for one “game tick”.

3.1.6.2 Interrupts and Exceptions in CPU

There are three interrupts/exceptions handled by the CPU to progress or halt the game state.

Interrupts:	**Priority:**
Keyboard interrupt	2
GameTick Interrupt	1
Exceptions:	
StackOverflow exception	3

When an exception or interrupt is triggered, the PC must be loaded with an address in instruction memory, the software will have two instructions to branch to another address location where ISR is located.

Address space of interrupts/exceptions:

GameTick interrupt	- 0x3FD
Keyboard Interrupt	- 0x3FE
StackOverflow exception	- 0x3FF

These locations should hold an instruction that unconditionally branches to the memory location of that particular interrupt/exception Interrupt Service Routine (ISR). When an interrupt or exception is triggered, the trap handler is responsible for coordinating what needs to happen to handle the interrupt:

<u>GameTick Interrupt:</u>

1) Flop interrupt signal and wait for all hazards to clear
2) Set hazard
3) Save interrupted PC + 1 to EPC
4) Set PC to address 0x3FD

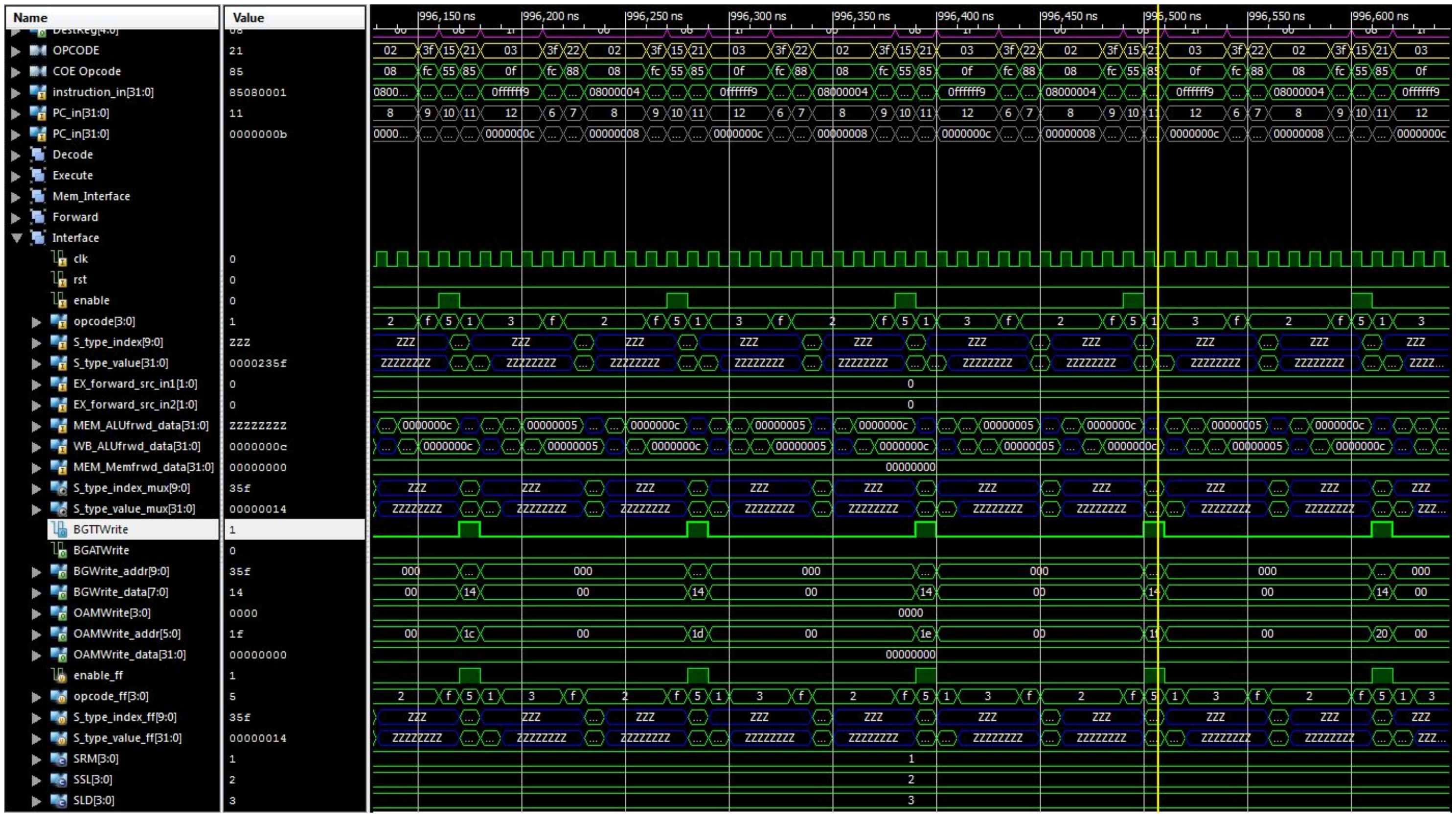
<u>Keyboard Interrupt:</u>

1) Flop interrupt signal and wait for all hazards to clear
2) Set hazard
3) Save keyboard data to IDR
4) Save PC + 1 to EPC
5) Set PC to address 0x3FE

<u>Overflow Interrupt (unimplemented):</u>

1) Flush current instruction in EX, ID, and IF stages
2) Set Overflow hazard
3) Save PC of instruction that caused overflow to EPC
4) Set PC to address 0x3FF

3.1.7 Sample CPU Simulation Waveforms



Name	Value
OPCODE	21
COE Opcode	85
instruction_in[31:0]	85080001
PC_in[31:0]	11
PC_in[31:0]	0000000b
Decode	
Execute	
Mem_Interface	
Forward	
Interface	
clk	0
rst	0
enable	0
opcode[3:0]	1
S_type_index[9:0]	zzz
S_type_value[31:0]	0000235f
EX_forward_src_in1[1:0]	0
EX_forward_src_in2[1:0]	0
MEM_ALUfrwd_data[31:0]	zzzzzzzz
WB_ALUfrwd_data[31:0]	0000000c
MEM_Memfrwd_data[31:0]	00000000
S_type_index_mux[9:0]	35f
S_type_value_mux[31:0]	00000014
BGTTWrite	1
BGATWrite	0
BGWrite_addr[9:0]	35f
BGWrite_data[7:0]	14
OAMWrite[3:0]	0000
OAMWrite_addr[5:0]	1f
OAMWrite_data[31:0]	00000000
enable_ff	1
opcode_ff[3:0]	5
S_type_index_ff[9:0]	35f
S_type_value_ff[31:0]	00000014
SRM[3:0]	1
SSL[3:0]	2
SLD[3:0]	3

Figure 3.1.7(a): Sprite Instruction Writing the Background Border of TRON

- Figure 3.1.7(a) displays a snippet from a simulation of the game TRON. The main module seen in this figure is the CPU/PPU interface, which is responsible for loading the PPU's foreground and background sprite tables with data. The highlighted signal in the figure is the "Background Tile Table Write Enable", which is set high when writing data to the background tile table. Upon closer inspection, you can see that the BGWrite_data signal is identical for each enabled write. However, the BGWrite_addr is being steadily incremented upon each enabled write, which signifies that the background location being written to is steadily moving across the screen as the border extends from right to left.

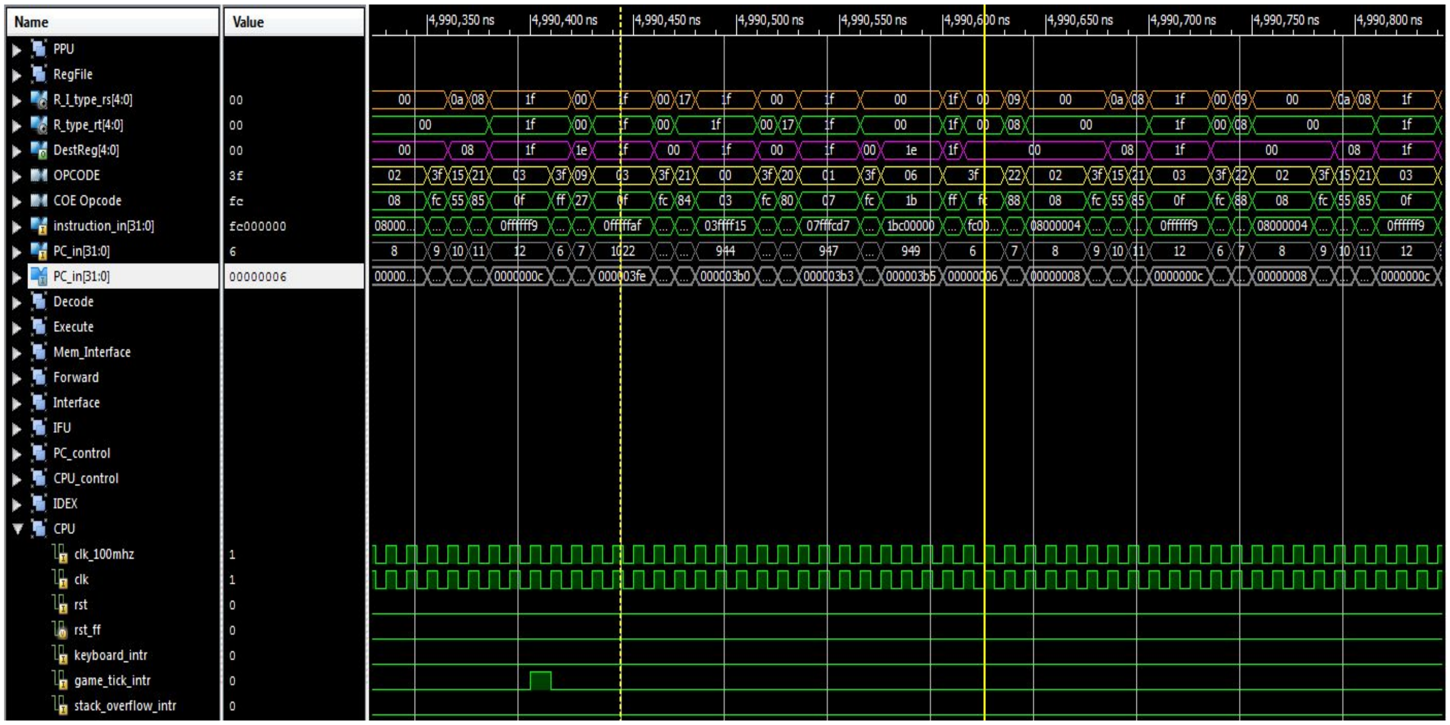


Figure 3.1.7(b): Game Tick Interrupt Jumping to Interrupt ISR

- Figure 3.1.7(b) displays a snippet from a simulation of the game TRON during a game tick interrupt pulse. The pulse shown in the lower left corner of the image shows the game tick interrupt entering the CPU, and the highlighted signal 'PC_in' can be seen jumping to 0x000003FE at the left dashed cursor line, which is actually the address of the *Keyboard interrupt*. Once here, the ISR is called, which immediately returns because there has been no user input since the last refresh of the game state. Finally, 'PC_in' jumps back to the interrupted program counter at the completion of the ISR, which can be seen at the right bold cursor line. **This occasional mishandling of the interrupt service routines was discovered at the end of our system's design process. Interrupts are crucial for correct execution of the games, and correcting this error would have undoubtedly added a great deal to our system's functionality.**

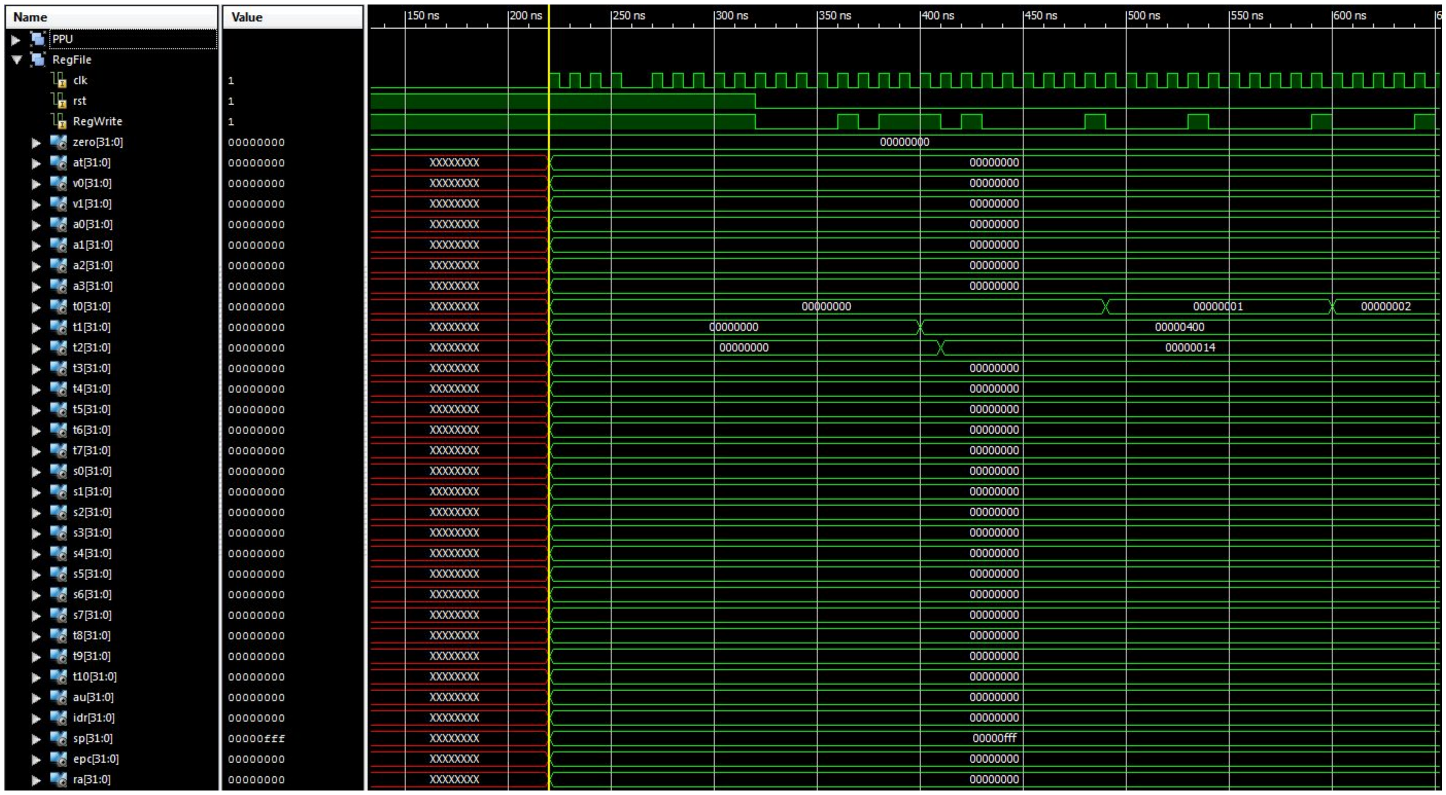
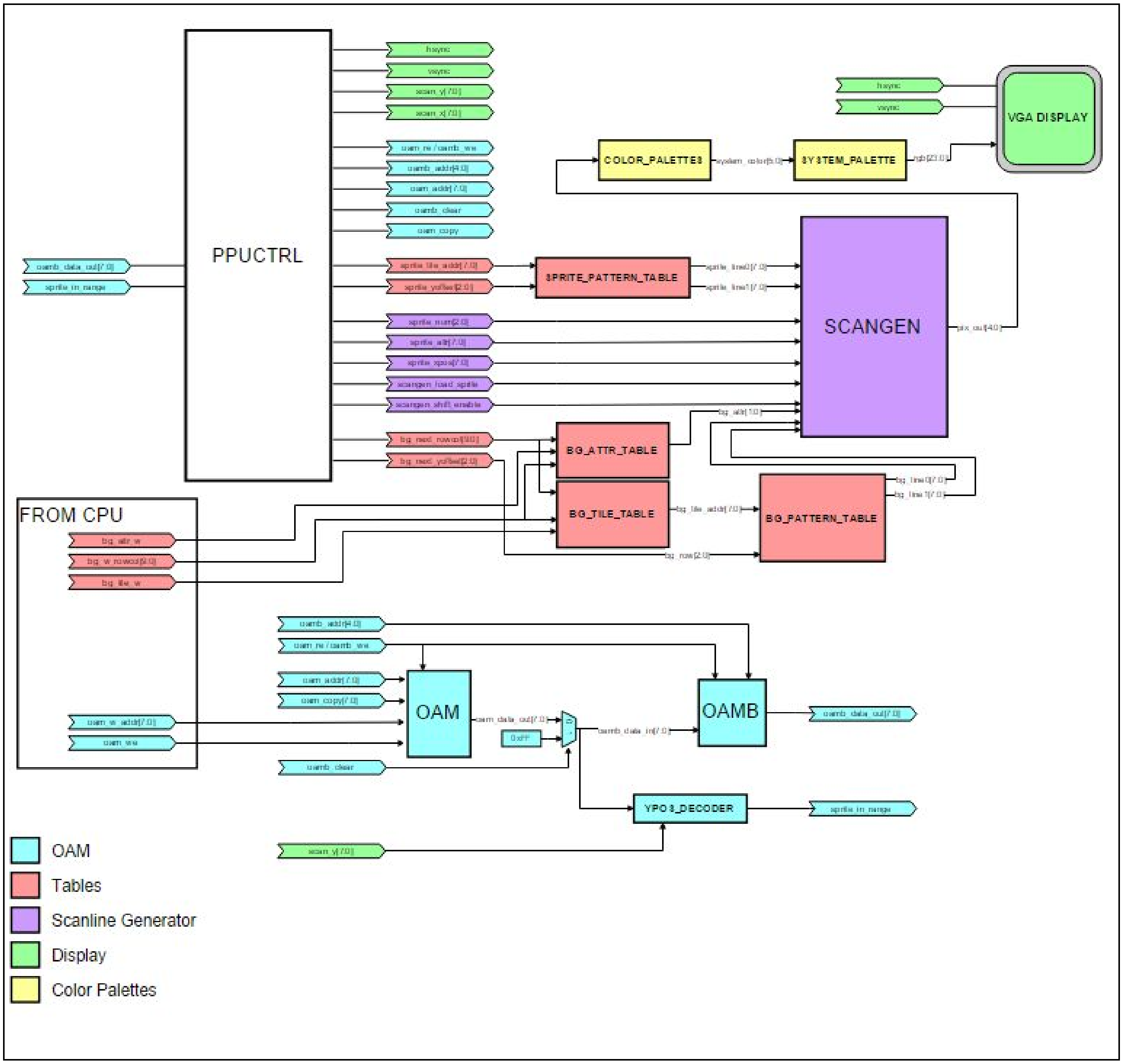


Figure 3.1.7(c): Register initialization upon Reset

- Figure 3.1.7(c) displays the contents of the 32-bit register file upon reset of the system. When the system is reset, all data contained within the register file is reset to the value of 0x00000000, except for the stack pointer register which is initialized to the top of the stack at 0x00000FFF. This ensures that all data held within the register file from previous executions is flushed, which sets up a new, clean game state.

3.2 Picture Processing Unit (PPU)

The PPU is the module responsible for generating the graphics of the system. It interfaces directly with the CPU and outputs to the VGA display. The rendering of both backgrounds and sprites is done completely synchronous with the VGA display. This means that the rendering takes place scanline by scanline. The display size supported is 256x256 pixels at a framerate of 60Hz.

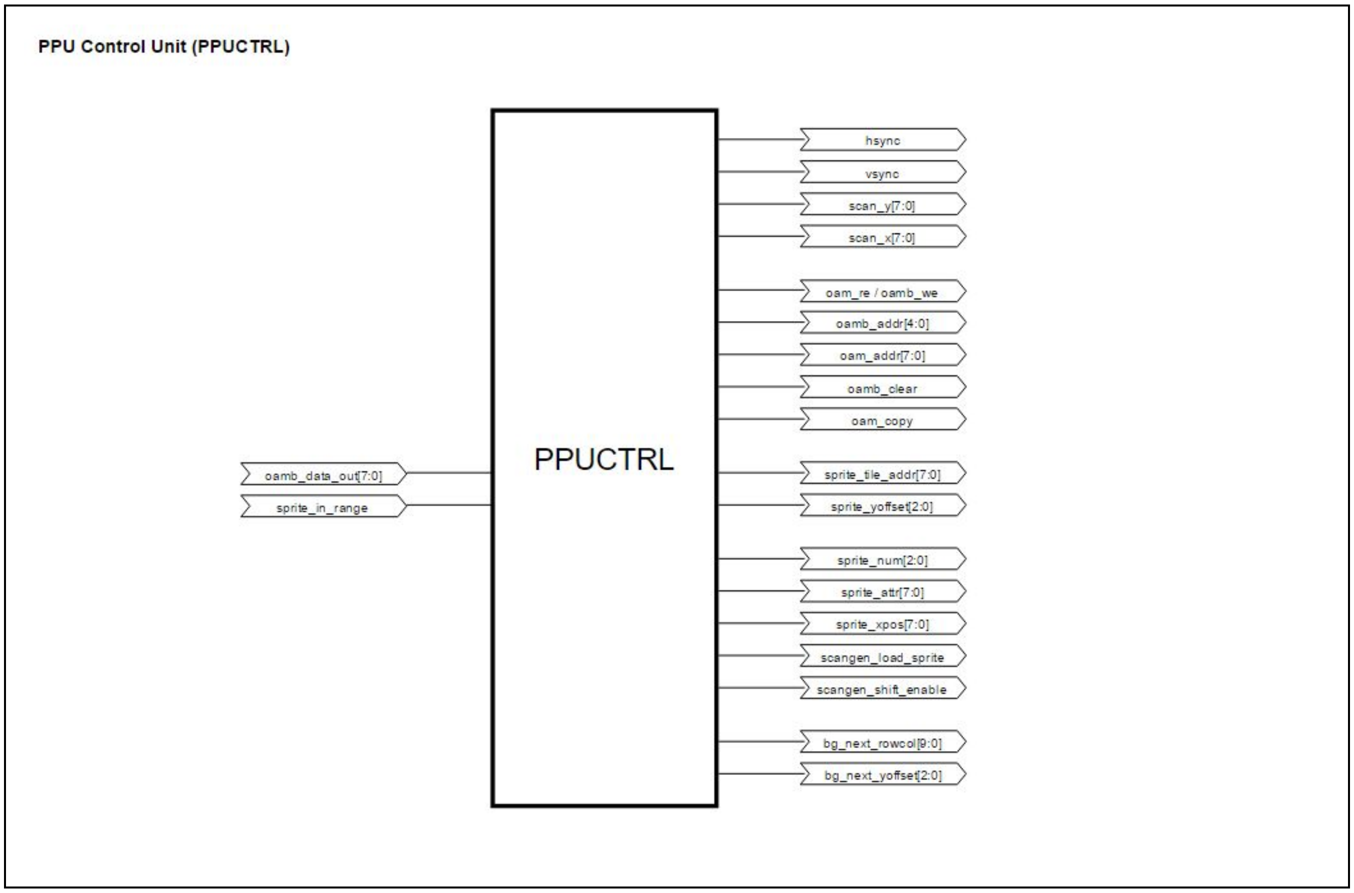


3.2.1 PPU Control Unit (PPUCTRL)

The PPUCTRL is the control unit that coordinates all operations of the PPU. Its main operations include scanning through the OAM and loading the OAMB with sprites that will be visible on the current scanline and loading up the SCANGEN with this data from the OAMB.

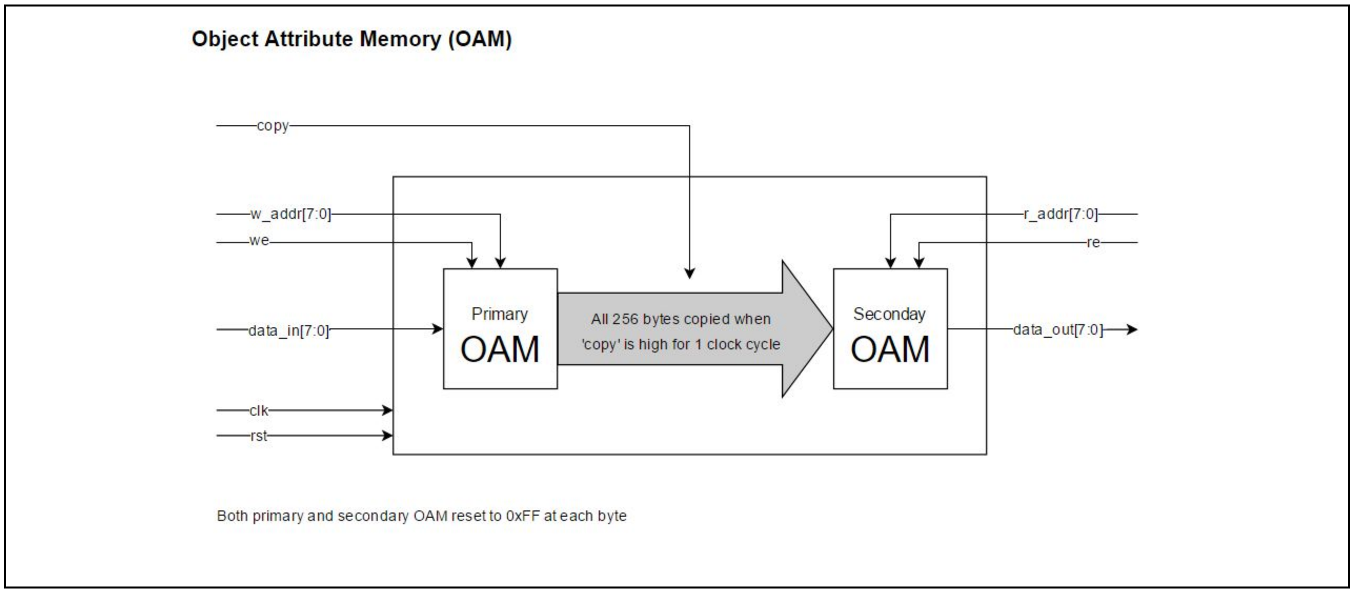
The steps the PPU takes each scanline are as follows:

1. Clear the OAMB to 0xFF at each position.
2. Check each sprite in the OAM, if it is within range, copy its data into the next available position in the OAMB. Stop checking if OAMB is full or all sprites in OAM have been checked.
3. Wait for hsync
4. Load each sprite from the OAMB into the SCANGEN. This requires fetching tile data from the sprite pattern table.
5. Load the tile and attribute data for the first two background tiles in the current row into the SCANGEN.
6. Enable the SCANGEN to start shifting out pixels.
    a. Each eight shifts, load the next background tile and attribute data into the SCANGEN.
    b. While the SCANGEN is shifting out the current line, step 1 can begin again in parallel.



3.2.2 Object Attribute Memory (OAM)

The OAM stores information about sprites that are currently displayed on the screen. Externally, the OAM can be viewed as a byte-addressable memory with concurrent read and write access. Internally, the OAM actually contains two memory blocks of equal size. One primary OAM and secondary OAM. To allow for a program to control the sprites, the CPU is given direct access to the PPU's primary OAM. The primary OAM that the CPU writes to is actually never directly accessed by the PPU other than one single occasion. Once a frame is complete, the entire contents of the primary OAM are copied into the secondary OAM over a single clock cycle. This way, there are no issues dealing with concurrent read and writes on the OAM causing sprite flickering. Therefore, if contents of the OAM change, it will not appear to have changed (when reading), until copy is held high for a clock cycle.

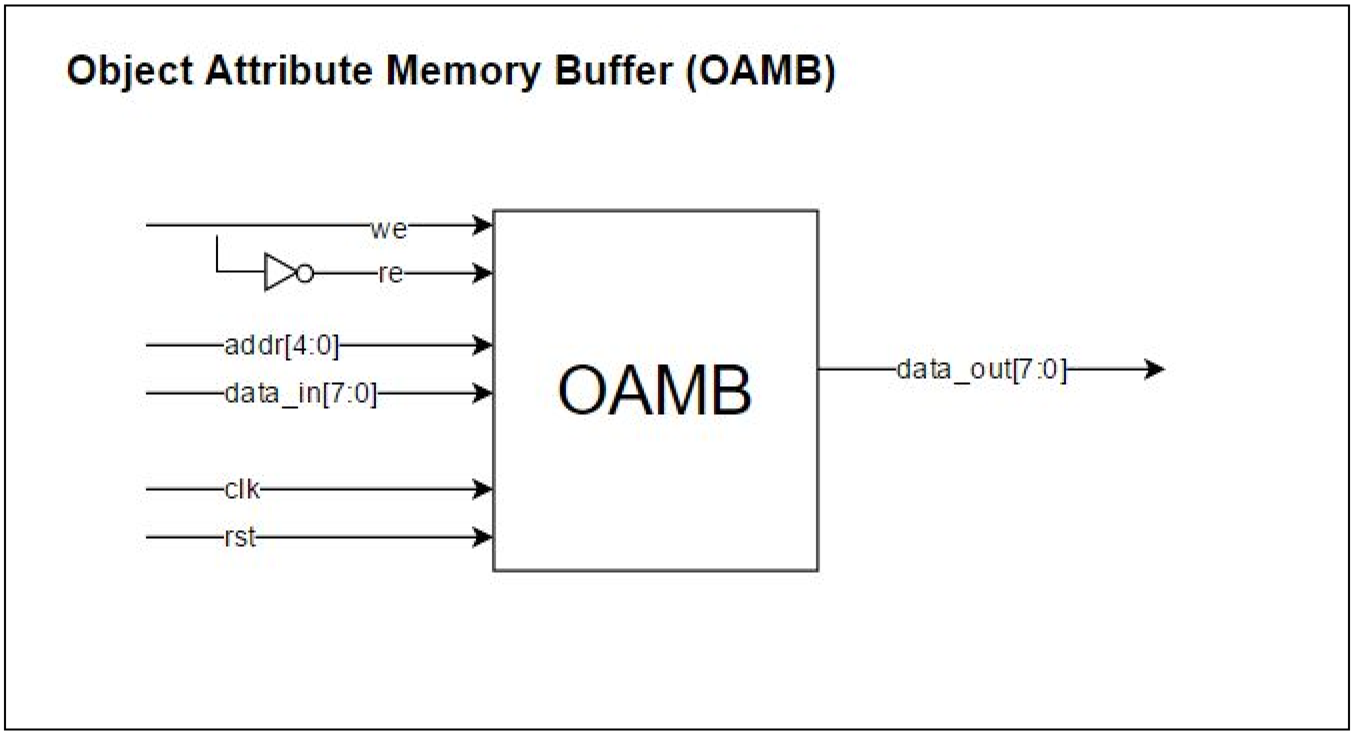

Object Attribute Memory (OAM)
copy
w_addr[7:0]
we
data_in[7:0]
clk
rst
Primary
OAM
All 256 bytes copied when 'copy' is high for 1 clock cycle
Seconday
OAM
r_addr[7:0]
re
data_out[7:0]
Both primary and secondary OAM reset to 0xFF at each byte

The OAM can contain information about a maximum of 64 sprites. Each sprite takes four bytes of data. The ordering of the bytes for each sprite are as follows.

Byte	**Name**	**Description**
0	Y Position	The Y position of the top-left corner of the sprite on the screen. If this byte is set to 0xFF, the sprite is considered an empty entry.
1	Tile Number	The tile used for this sprite: an index into the sprite pattern table.
2	Attributes	Attributes about how this sprite is displayed. [1:0] The color palette used for this sprite. [5:2] Unused [6] Flip horizontal if 1 [7] Flip vertical if 1
3	X Position	The X position of the top-left corner of the sprite on the screen.

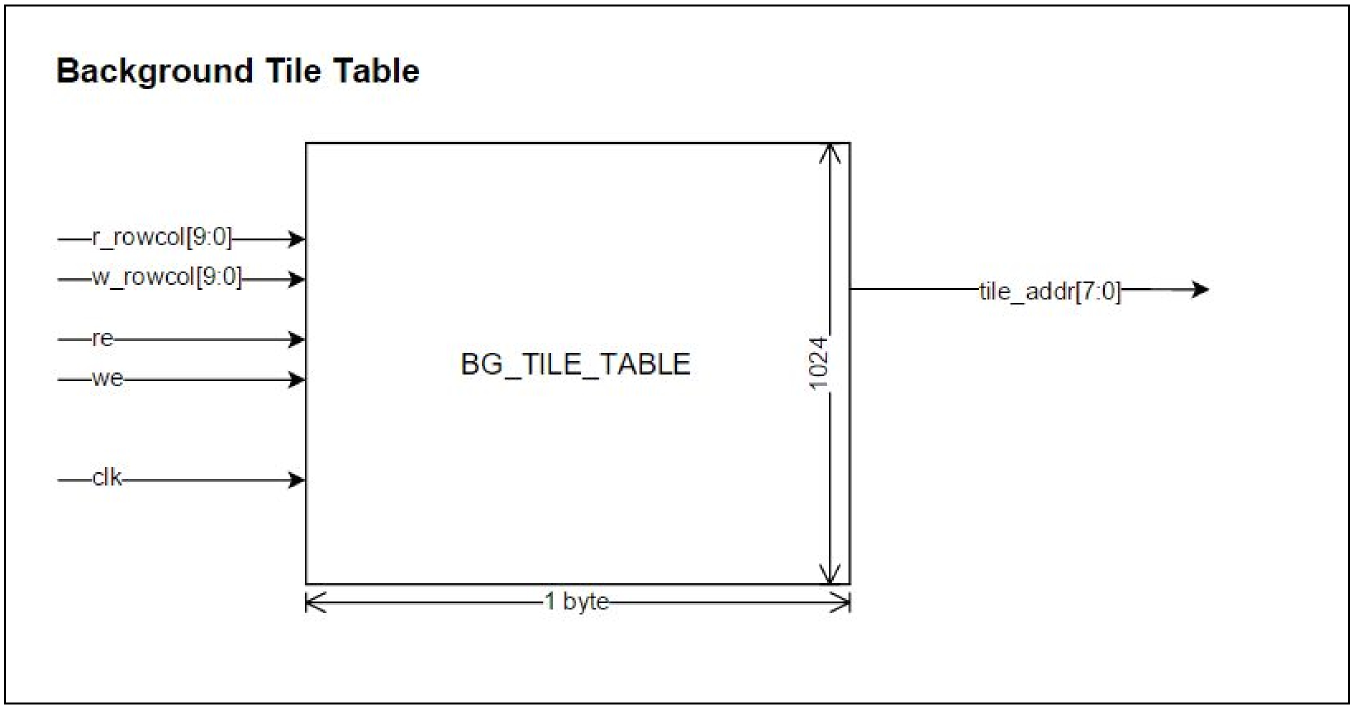
3.2.3 OAM Buffer (OAMB)

The OAM Buffer (OAMB) is a sprite holding memory with a similar format to the OAM. It can only hold information about 8 sprites (32 bytes total) that will be rendered on the current scanline. The PPUCTRL module is responsible for controlling the memory loaded into this module by reading from the OAM and determining if each sprite is on the current scanline. If it is, the sprite's 4 bytes are loaded into the OAMB at its next available position. The PPUCTRL is also responsible for clearing this module (setting each of its bytes to 0xFF) before it is loaded with the next scanline's sprites. The OAMB is byte-addressable for both reads and writes. Because this module can only contain 8 sprites, that is the limit of the number of sprites that can be displayed on a scanline.



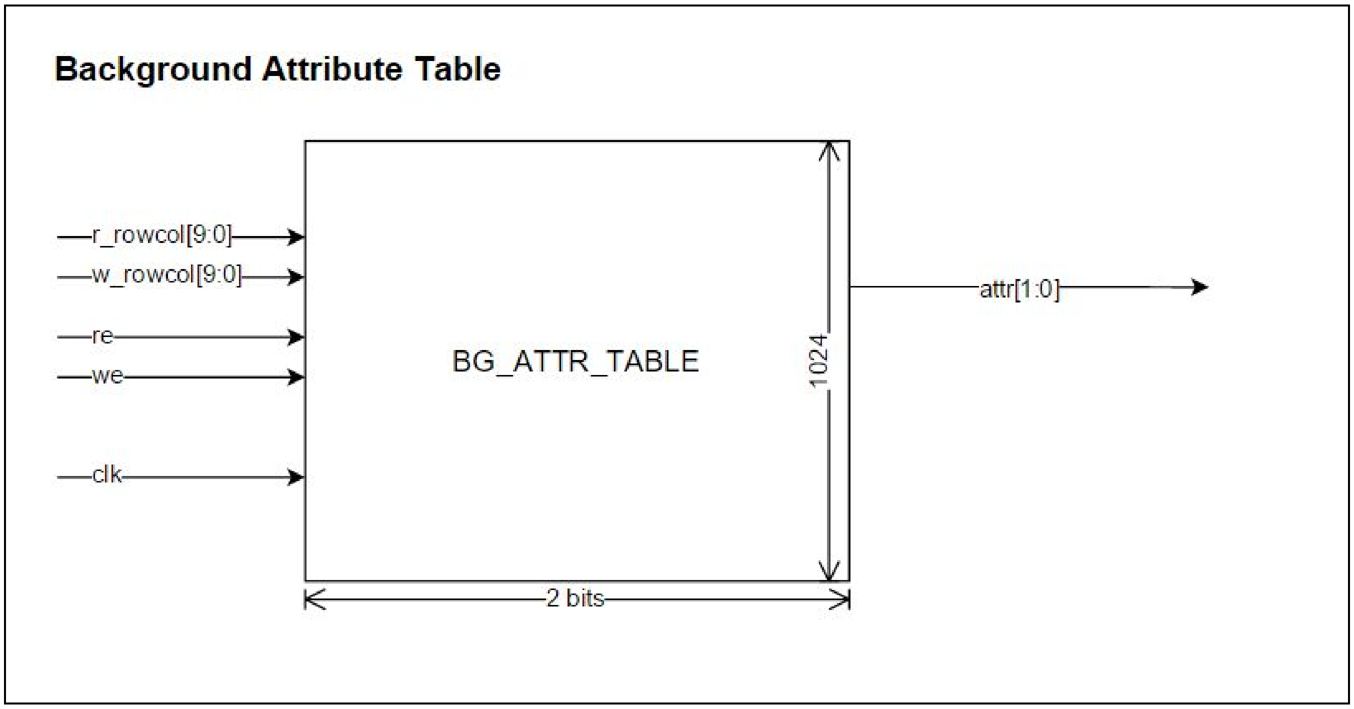
3.2.4 Background Tile Table

Backgrounds are made up of tiles from the background pattern table. Because each tile is 8x8 pixels, the background is made up of a 32x32 array of tiles. The background tile table holds information on which tile is used for each piece of the background. The table is RAM that is initialized by game data but is writable by the CPU. This module supports concurrent PPU reads and CPU writes.



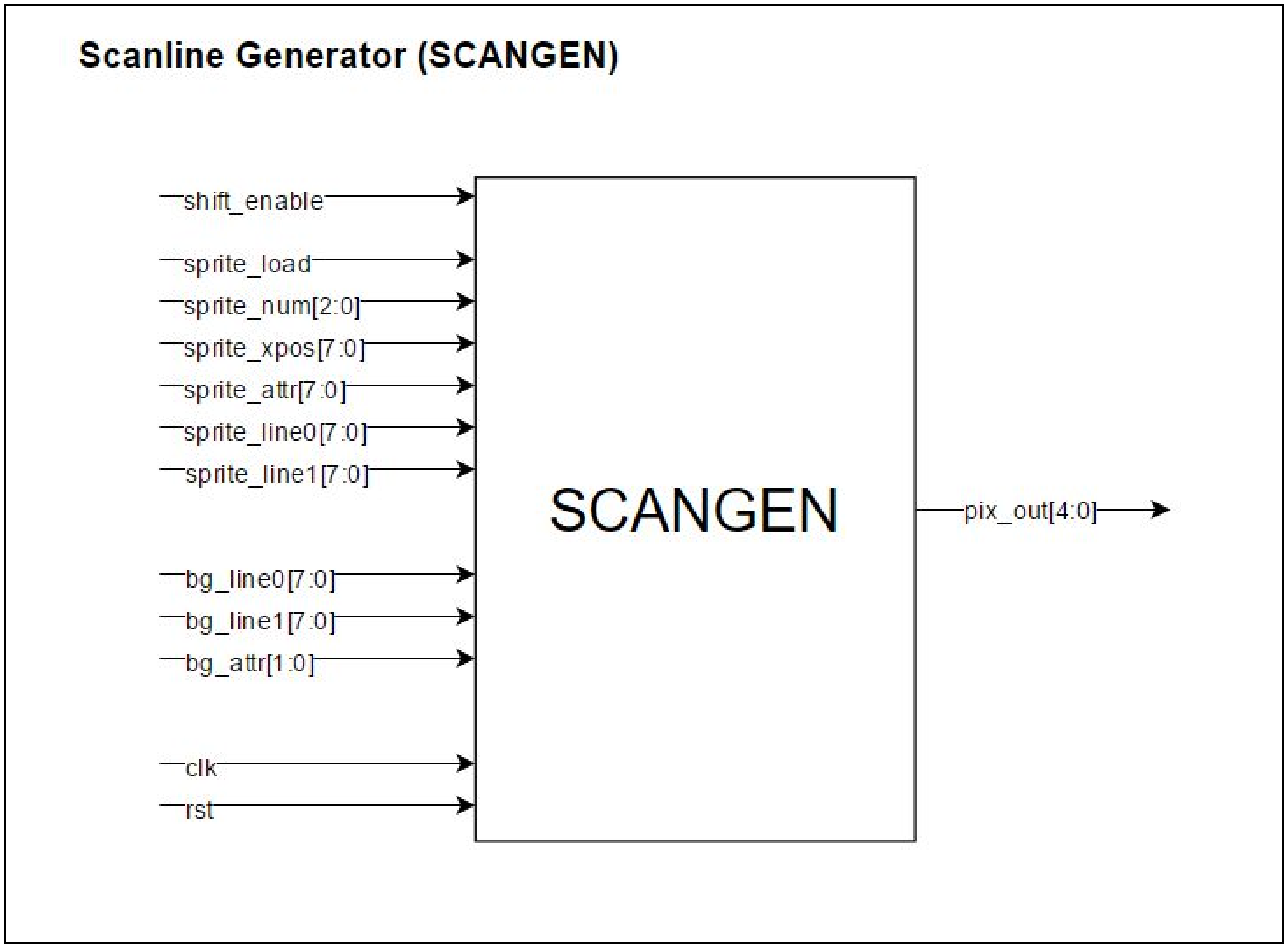
3.2.5 Background Attribute Table

Backgrounds are made up of tiles from the background pattern table. Because each tile is 8x8 pixels, the background is made up of a 32x32 array of tiles. The background attribute table holds information on which color palette is used for each tile that makes up the background. The table is RAM that is initialized by game data but is writable by the CPU. This module supports concurrent reads and writes (Reads by the PPU and writes by the CPU).

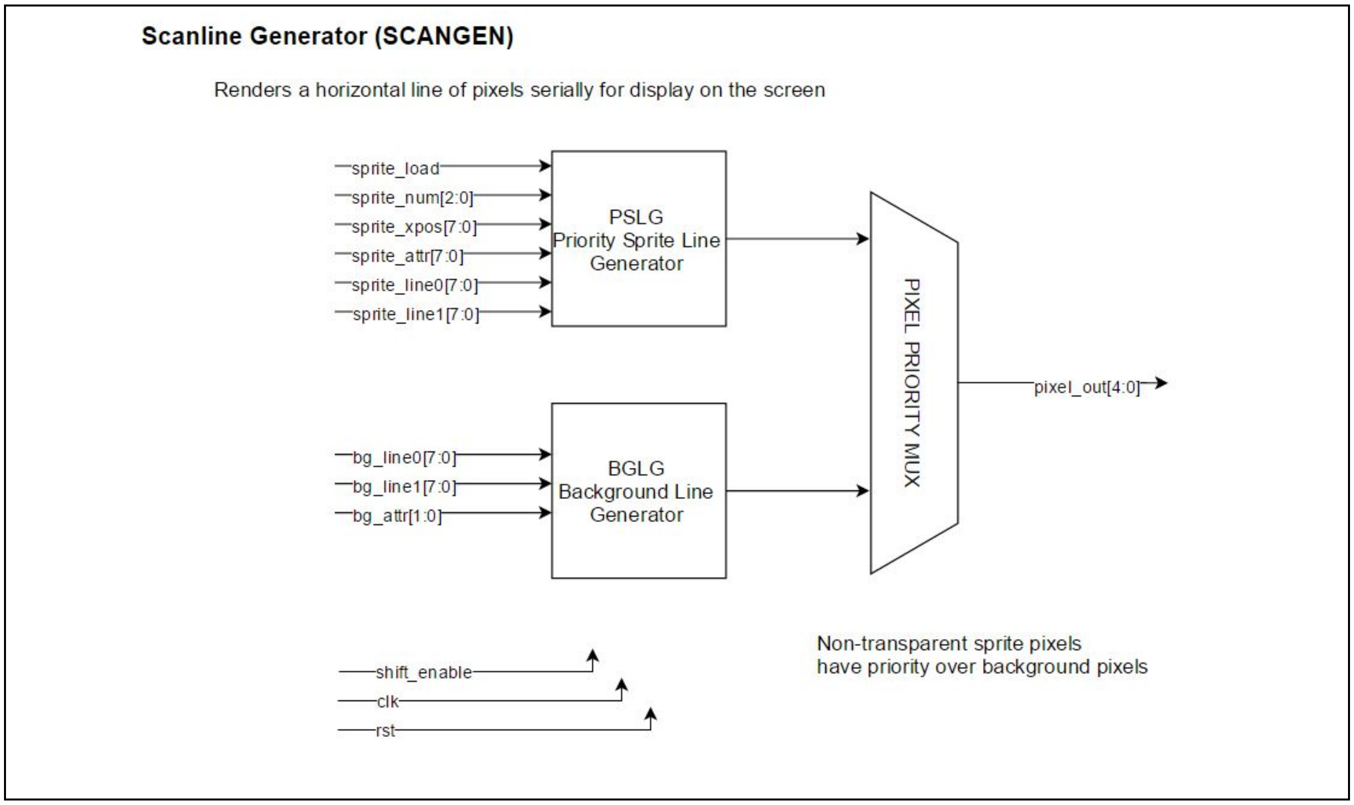


3.2.6 Scanline Generator (SCANGEN)

The scanline generator is responsible for shifting out color indices that index into the color palettes for display on the screen. The SCANGEN is loaded with sprite data assembled by the PPUCTRL after scanning through the OAM and loading the OAMB. Both the OAMB and the SCANGEN are capable of holding 8 sprites. The scangen prioritizes non-transparent pixels from sprites at lower indices. If no sprite pixels are available, the background pixels are used with lowest priority. Each shift out is a pixel on the current scanline.

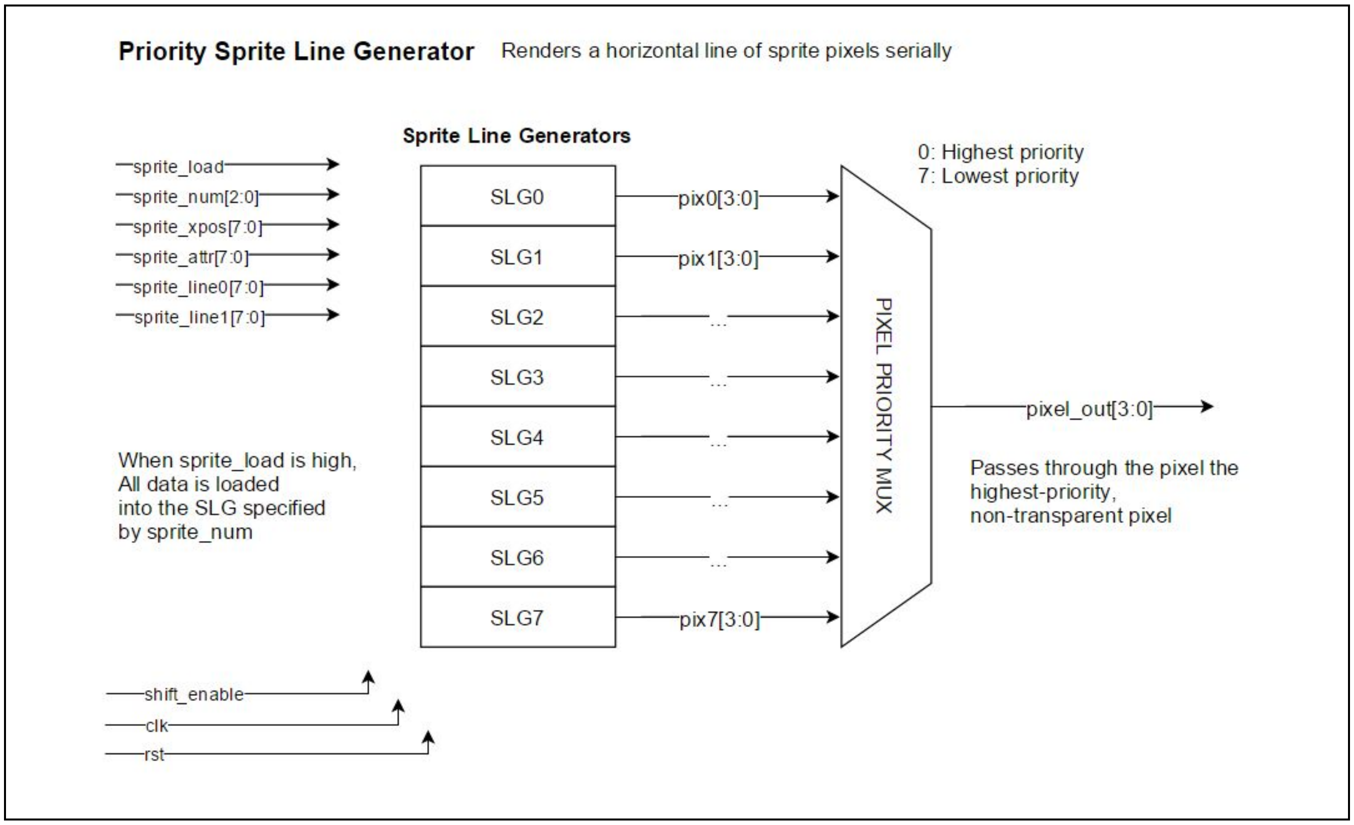


The SCANGEN can be split up into two shifting components. One for sprites and one for backgrounds.


Scanline Generator (SCANGEN)
Renders a horizontal line of pixels serially for display on the screen
sprite_load
sprite_num[2:0]
sprite_xpos[7:0]
sprite_attr[7:0]
sprite_line0[7:0]
sprite_line1[7:0]
PSLG
Priority Sprite Line Generator
bg_line0[7:0]
bg_line1[7:0]
bg_attr[1:0]
BGLG
Background Line Generator
PIXEL PRIORITY MUX
pixel_out[4:0]
shift_enable
clk
rst
Non-transparent sprite pixels have priority over background pixels

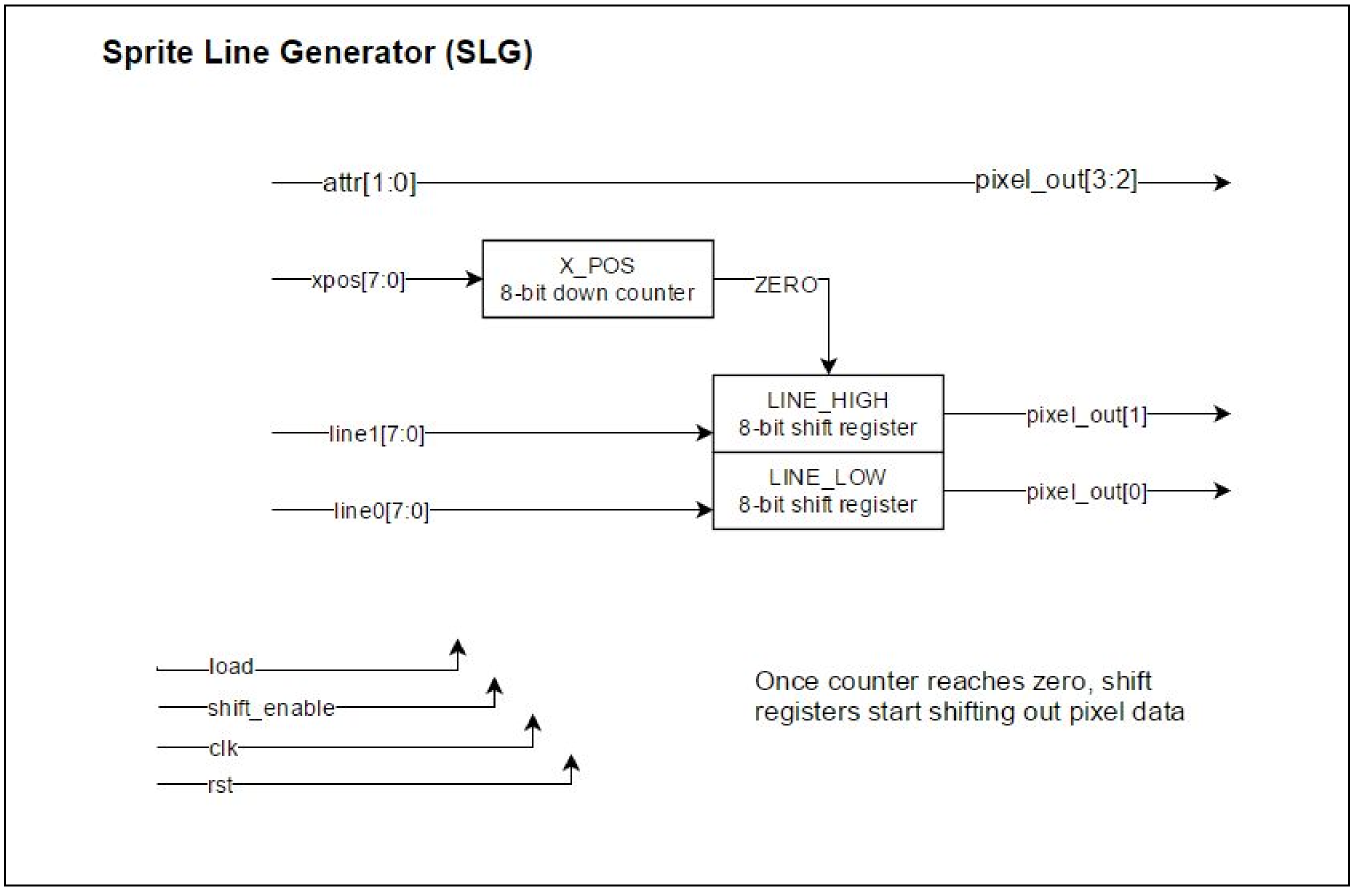
3.2.7 Priority Sprite Line Generator (PSLG)

The PSLG is loaded with sprite data (maximum of eight sprites) and shifts out pixels corresponding to the sprites. Sprites with lower indices are given priority over other sprites.



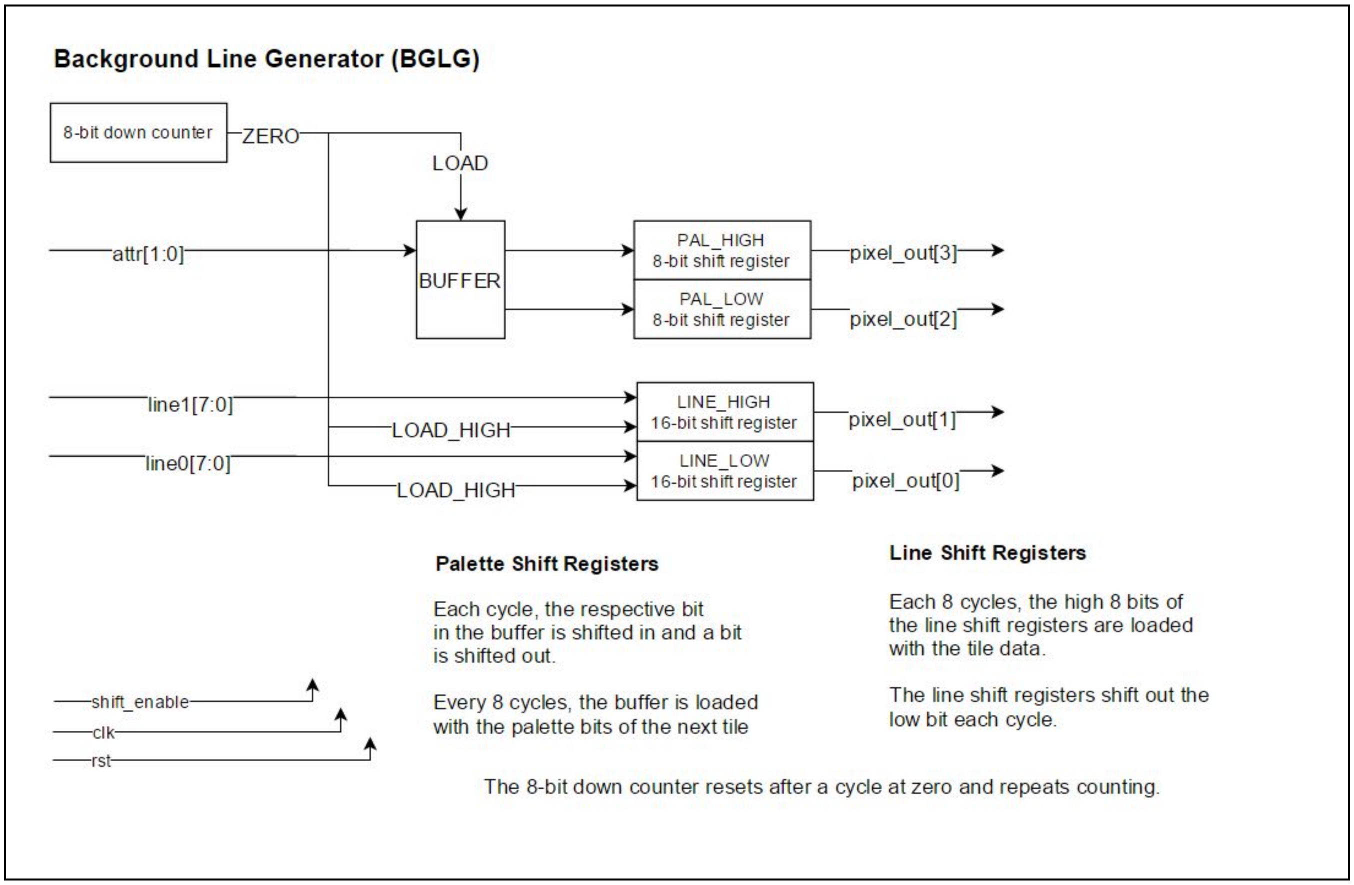
3.2.8 Sprite Line Generator (SLG)

Each SLG contains a sprite's x position, its color palette, and loaded tile data. The x position is loaded into a down counter. Each cycle, this down counter is decremented. Once it reaches zero, the sprite's tile data will start shifting out pixels.



3.2.9 Background Line Generator (BGLG)

The BGLG shifts out pixels for background tiles. The BGLG contains tile data for the current background tile and the next background tile. Every 8 shifts, the next tile's data is loaded into the high byte of the shift registers. Also, at this time, the attribute bytes from the background attribute table are loaded into the buffer. These contain the color palette used for the background tile.


Background Line Generator (BGLG)
8-bit down counter
ZERO
LOAD
attr[1:0]
BUFFER
PAL_HIGH 8-bit shift register
PAL_LOW 8-bit shift register
pixel_out[3]
pixel_out[2]
line1[7:0]
LOAD_HIGH
line0[7:0]
LOAD_HIGH
LINE_HIGH 16-bit shift register
LINE_LOW 16-bit shift register
pixel_out[1]
pixel_out[0]
shift_enable
clk
rst

Palette Shift Registers

Each cycle, the respective bit in the buffer is shifted in and a bit is shifted out.

Every 8 cycles, the buffer is loaded with the palette bits of the next tile

Line Shift Registers

Each 8 cycles, the high 8 bits of the line shift registers are loaded with the tile data.

The line shift registers shift out the low bit each cycle.

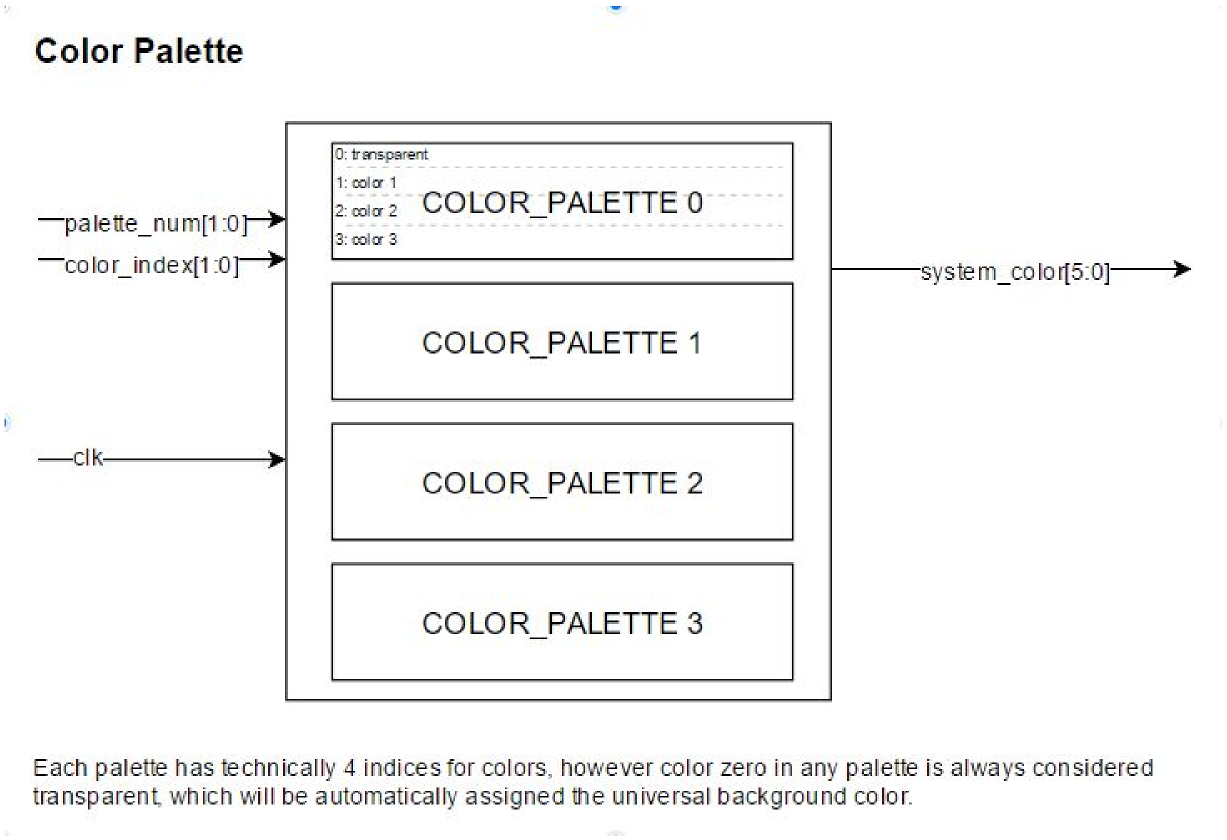
The 8-bit down counter resets after a cycle at zero and repeats counting.

3.2.10 Color Palettes

There are two ROMs that define the colors used for sprites and backgrounds. There is also a single system palette that cannot be changed.

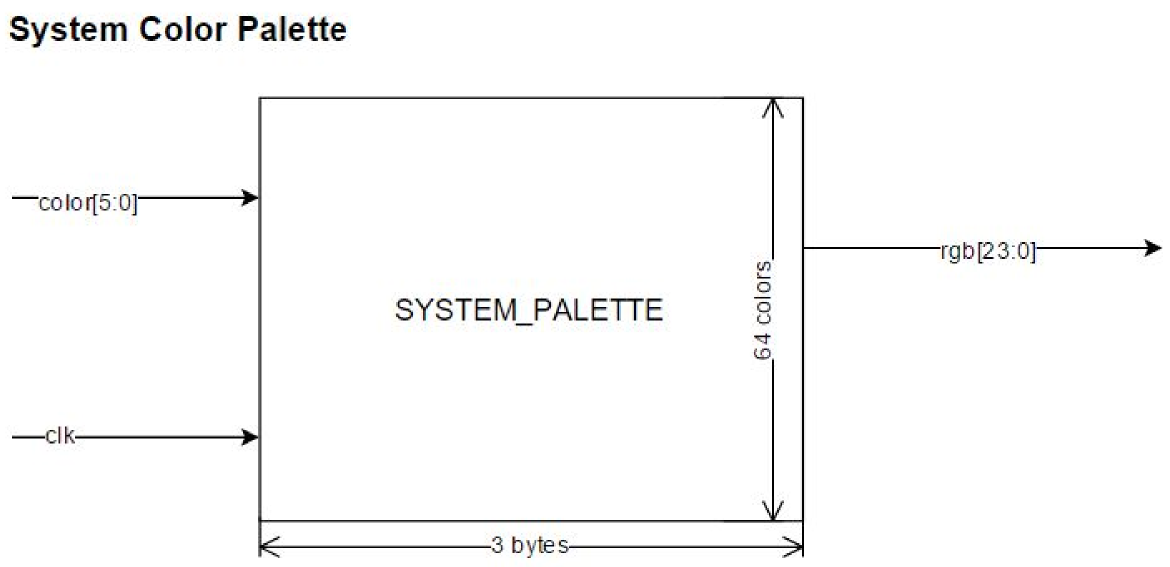
3.2.11 Sprite and Background Color Palettes

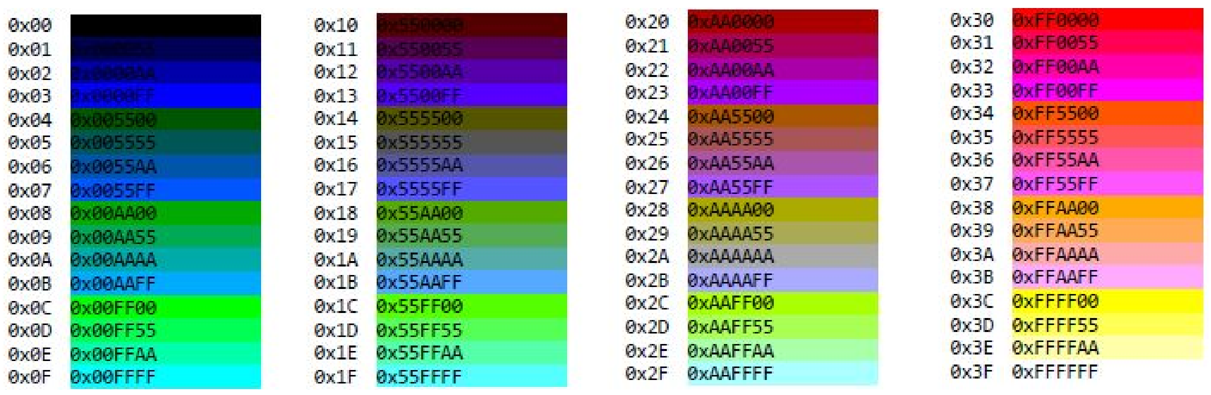
There are a total of 8 color palettes defined by the game data. Four for sprites and four for backgrounds. It is important to note that for each of the sprite palettes, color 0 is irrelevant because the internals always interpret sprite colors of 0 as transparent. Furthermore, the backgrounds color palettes also have special handling for color 0. Regardless of the color palette used, color 0 will be always be read as color 0 of the 0th background color palette. This is referred to as the universal background color. Each color returned by any of the color palettes is actually an address of the color stored in the system color palette.


Color Palette
palette_num[1:0]
color_index[1:0]
clk
0: transparent
1: color 1
2: color 2
3: color 3
COLOR_PALETTE 0
COLOR_PALETTE 1
COLOR_PALETTE 2
COLOR_PALETTE 3
system_color[5:0]
Each palette has technically 4 indices for colors, however color zero in any palette is always considered transparent, which will be automatically assigned the universal background color.

3.2.12 System Color Palette

The system color palette is universal regardless of the game and cannot be changed. The system color palette contains the 64 possible colors that can be used by graphics in the system.



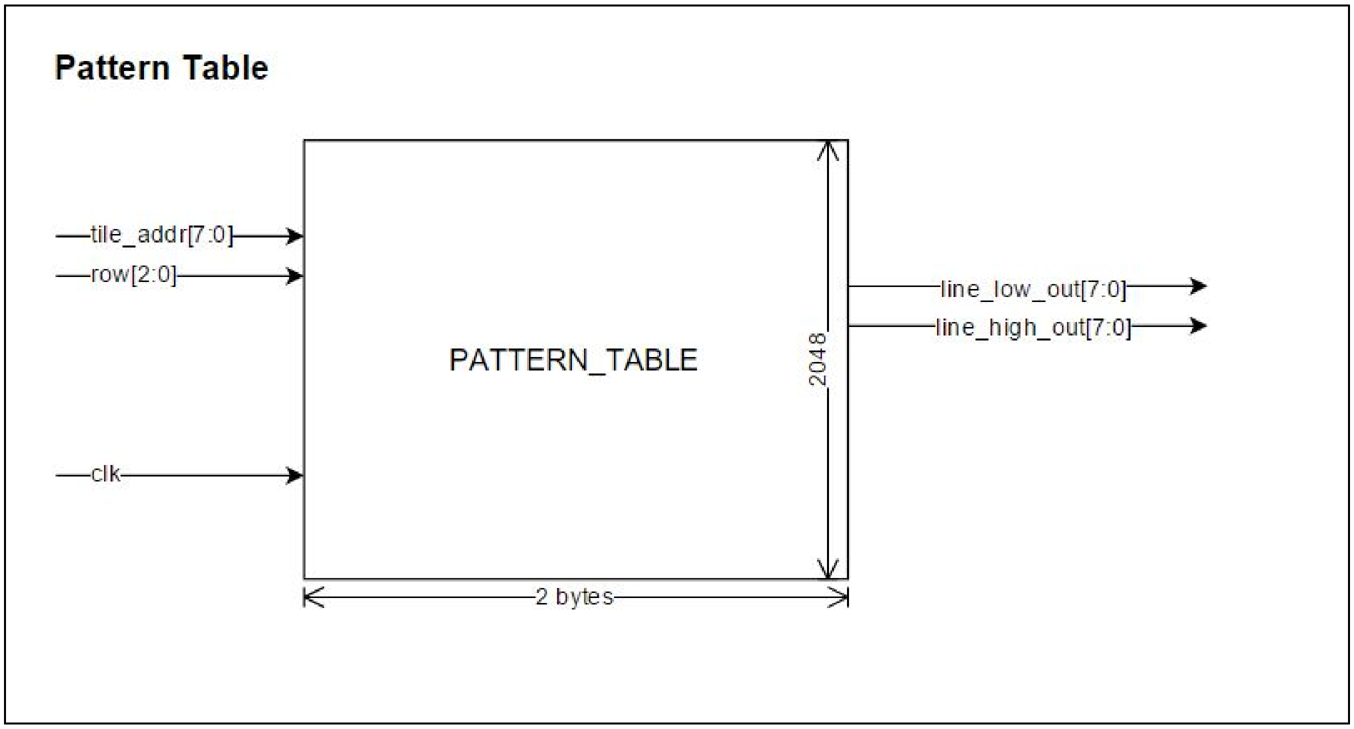


0x00	[illegible]	0x10	0x550000	0x20	0xAA0000	0x30	0xFF0000
0x01	0x000055	0x11	0x550055	0x21	0xAA0055	0x31	0xFF0055
0x02	0x0000AA	0x12	0x5500AA	0x22	0xAA00AA	0x32	0xFF00AA
0x03	0x0000FF	0x13	0x5500FF	0x23	0xAA00FF	0x33	0xFF00FF
0x04	0x005500	0x14	0x555500	0x24	0xAA5500	0x34	0xFF5500
0x05	0x005555	0x15	0x555555	0x25	0xAA5555	0x35	0xFF5555
0x06	0x0055AA	0x16	0x5555AA	0x26	0xAA55AA	0x36	0xFF55AA
0x07	0x0055FF	0x17	0x5555FF	0x27	0xAA55FF	0x37	0xFF55FF
0x08	0x00AA00	0x18	0x55AA00	0x28	0xAAAA00	0x38	0xFFAA00
0x09	0x00AA55	0x19	0x55AA55	0x29	0xAAAA55	0x39	0xFFAA55
0x0A	0x00AAAA	0x1A	0x55AAAA	0x2A	0xAAAAAA	0x3A	0xFFAAAA
0x0B	0x00AAFF	0x1B	0x55AAFF	0x2B	0xAAAAFF	0x3B	0xFFAAFF
0x0C	0x00FF00	0x1C	0x55FF00	0x2C	0xAAFF00	0x3C	0xFFFF00
0x0D	0x00FF55	0x1D	0x55FF55	0x2D	0xAAFF55	0x3D	0xFFFF55
0x0E	0x00FFAA	0x1E	0x55FFAA	0x2E	0xAAFFAA	0x3E	0xFFFFAA
0x0F	0x00FFFF	0x1F	0x55FFFF	0x2F	0xAAFFFF	0x3F	0xFFFFFF

These are all of the 64 colors available to graphics in the system.

3.2.13 Pattern Tables

Within the PPU, two pattern tables exist. One for sprites and one for backgrounds. Each pattern table stores the actual bitmap data that makes up a sprite tile or background tile. These pattern tables are ROMs set by the game data. The format of these tables is described in the software section.

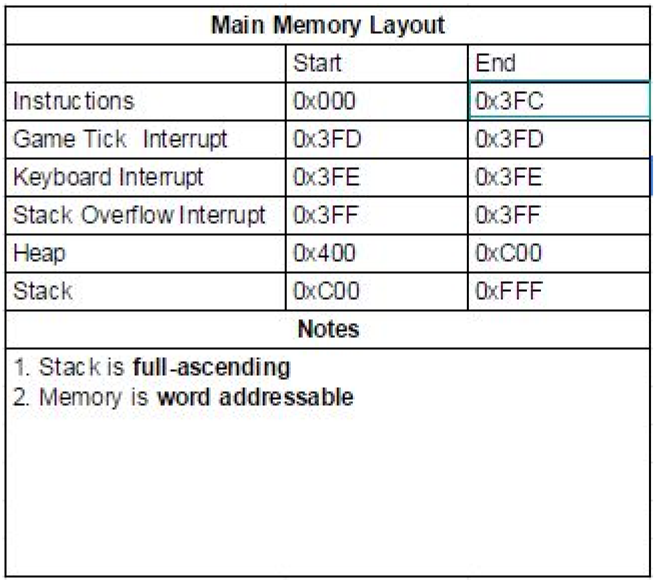


4 Software Architecture and Framework

4.1 Memory Layout

The Tronsistor-32 architecture specifies four (4) block RAM memory segments that must be initialized with Core Generator files. These are:

1. Main Memory. This includes 1021 words of instructions, 3 words for trap handler translations, 2048 words for the heap, and 1024 words for the stack, for a total of 4096 32-bit words. In retrospect, we should have allocated more words to instructions and fewer words to both the heap and the stack. This allocation is shown graphically below:



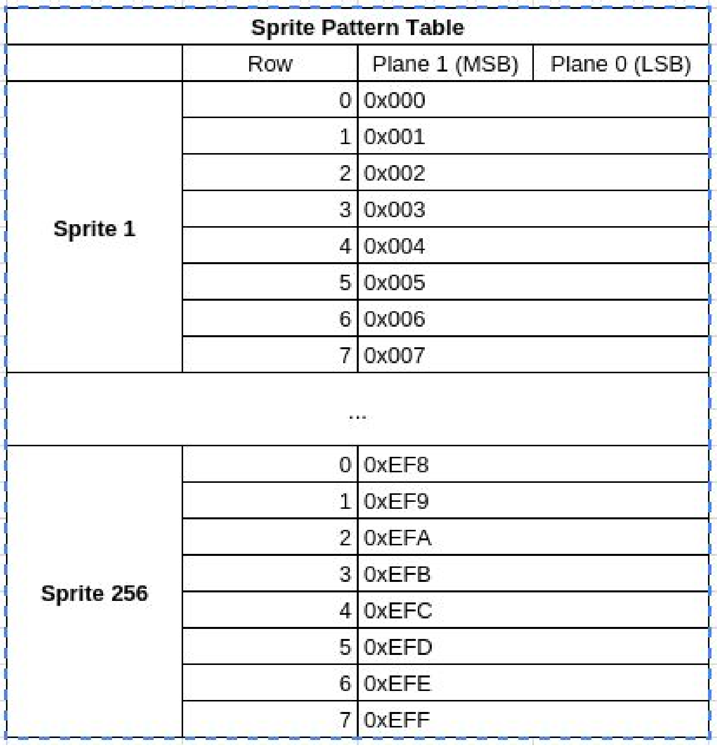
Main Memory Layout

	Start	End
Instructions	0x000	0x3FC
Game Tick Interrupt	0x3FD	0x3FD
Keyboard Interrupt	0x3FE	0x3FE
Stack Overflow Interrupt	0x3FF	0x3FF
Heap	0x400	0xC00
Stack	0xC00	0xFFF

Notes

1. Stack is **full-ascending**
2. Memory is **word addressable**

2. Foreground Sprite Pattern Table. This contains 256 sets of 8 16-bit entries. Each 16-bit entry represents the indexes into a color palette for 1 line of a 8x8 sprite. Thus, 8 lines represents an entire sprite of indexes, and the entire pattern table represents 256 sprites. This is illustrated below:



Sprite Pattern Table			
	Row	Plane 1 (MSB)	Plane 0 (LSB)
Sprite 1	0	0x000	
	1	0x001	
	2	0x002	
	3	0x003	
	4	0x004	
	5	0x005	
	6	0x006	
	7	0x007	
...			
Sprite 256	0	0xEF8	
	1	0xEF9	
	2	0xEFA	
	3	0xEFB	
	4	0xEFC	
	5	0xEFD	
	6	0xEFE	
	7	0xEFF	

3. Background Sprite Pattern Table. This is the same as the Foreground Sprite Pattern Table outlined above.

4. Foreground & Background Color Palettes. This is simply eight sets (four foreground, four background) each containing four indexes into the system color palette. Each index into the system color palette is 8 bits, giving this a total bit size of 8*8*4.

4.2 Assembler

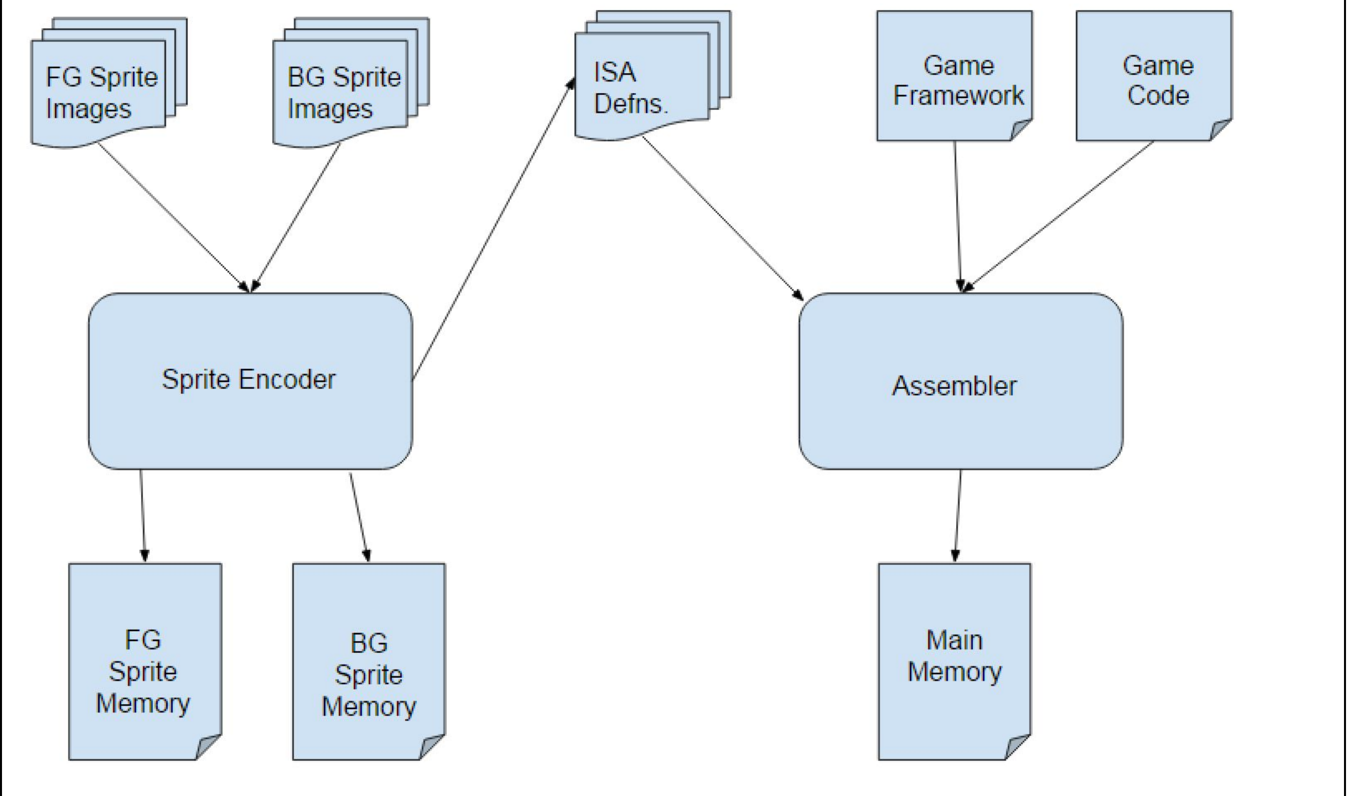


Figure 4.2: Assembler Layout.

We have implement a custom python assembler for the Tronsistor-32 ISA. It is implemented by making two passes at the assembly code:

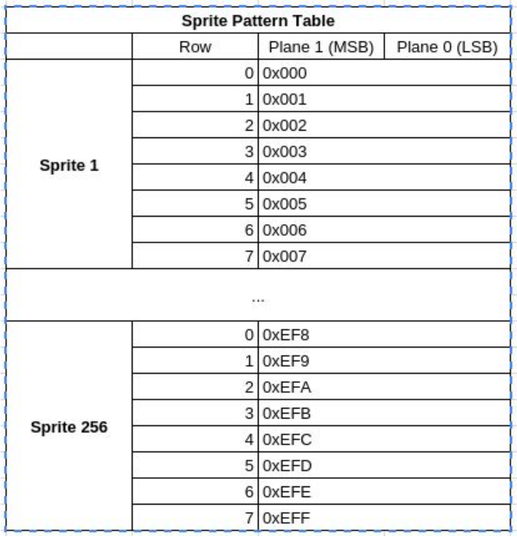
1. First Pass - Parses assembly line by line, taking special action when an empty line, start of instructions directive, or data directive is encountered. If a label is encountered it is added to a lookup table preseeded with register test-to-binary translations. Comments are stripped, and the instruction and arguments are decoded. If an instruction is invalid for any reason the program will stop. There are a few special cases while parsing a line, described as follows:
    a. Empty line - skip line and move on, do not increment PC
    b. Data directive - allocate space in heap as necessary to contain data, increment PC by corresponding number of words in the allocated space
    c. Start of Instructions - Change PC to Memory Mapped start of instructions location
2. Second Pass - Goes through parsed assembly line by line, translating instruction to machine code. Any argument fields that are not immediates are decoded using the lookup table generated from the previous pass. Any immediate fields are translated to 2s complement binary integers from either hex or decimal formats. If any arguments are not found in the

lookup table or not a hexadecimal or decimal immediate, the program throws an error and exits.

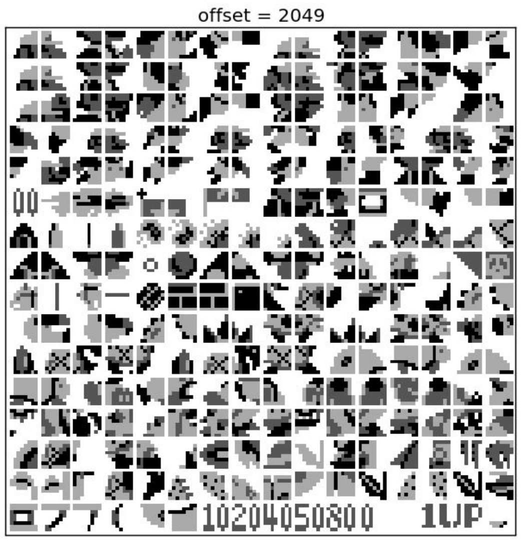
Lastly, the assembler formats the list of machine-code instructions into the '.coe' file layout and dumps to the specified output file. The implementation of the Tronsister-32 assembler is open source and can be found at the link below.

https://github.com/meggers/tronsister32-framework/tree/master/assembler

4.3 Sprite Generator



Sprite Pattern Table			
	Row	Plane 1 (MSB)	Plane 0 (LSB)
Sprite 1	0	0x000	
	1	0x001	
	2	0x002	
	3	0x003	
	4	0x004	
	5	0x005	
	6	0x006	
	7	0x007	
...			
Sprite 256	0	0xEF8	
	1	0xEF9	
	2	0xEFA	
	3	0xEFB	
	4	0xEFC	
	5	0xEFD	
	6	0xEFE	
	7	0xEFF	



The Tronsister-32 software framework provides a utility to generate chomp addressable '.coe' seed files given a list of sprite image files. The generator utility expects the dimensions of each image file to be divisible by 8 pixels, the size of a single sprite. If the image height or width is a multiple of 8, the image will be broken into several 8x8 pixel chunks and parsed serially from left-to-right and top-to-bottom.

Each sprite image must be composed of only four (4) rgba colors, specifically red (255, 0, 0, 255), green (0, 255, 0, 255), blue (0, 0, 255, 255) and black (0, 0, 0, 255). Black will be treated as "transparent." If the sprite generator encounters any other color during sprite generation, it will throw an error and exit.

Lastly, the sprite generator spits out a mapping of each sprite chunk to its sprite index in CSV format to be used by the assembler in seeding the heap. This allows the game developer the freedom of not having to generate references to each sprite by hand in the data section of his or her game logic.

Instead they can use an agreed upon format of sprite_file_name-chunk# as a reference to each sprite chunk.

The implementation of the Tronsister-32 Sprite Generator can be found at the following link.

https://github.com/meggers/tronsister32-framework/tree/master/sprites

4.4 Tronsistor-32 Framework Core Functions

This section lays out the method headers and descriptions for the functionality provided in the Tronsister-32 framework. These functions can be leveraged by programmers in order to create games involving multiple complicated sprites quickly and efficiently. In addition to these functions, the Tronsistor-32 framework also outlines various heap fields common to any game development, the most notable of which is 64 words that serves as a copy of the oam in heap memory. This can be used to keep track of the state of the oam, because the oam cannot be read. These fields are outlined in their entirety here:

```
x_mask:              .word 0xFF000000
sprite_index_mask:   .word 0x0000FF00
vertical_flip_mask:  .word 0x00800000
horiz_flip_mask:     .word 0x00400000
color_palette_mask:  .word 0x00030000
y_mask:              .word 0x000000FF
clear_sprite:        .word 0xFFFFFFFF

TRUE:                .word 0xFFFFFFFF
FALSE:               .word 0x00000000

oam_copy: .space 64
```

The aforementioned functions are outlined below:

```
This function, given a binary number 0-9 and a background tile location will set the
background tile corresponding to that number at that location.
```

```
# Function: draw_number             #
#                                   #
# Arguments: draw_number            #
#   $a0: number to draw             #
#   $a1: background position        #
#                                   #
# Return:                           #
#   N/A                             #
```

This function, given a binary number ranged 0-99 and a background tile location will draw that number at that location.

```
# Function: display_2digit_decimal  #
#                                   #
# Arguments:                        #
#   $a0: number to display          #
#   $a1: bg position to display at  #
#                                   #
# Return:                           #
#   N/A                             #
```

This function checks if a sprite represented by a rectangular boundary exceeds the game screen, returning an error code representing what boundaries were exceeded.

```
# Function: check_oob               #
#                                   #
# Arguments:                        #
#   0(sf): left sprite x pos        #
#   1(sf): top sprite y pos         #
#   2(sf): sprite width             #
#   3(sf): sprite height            #
#                                   #
# Return:                           #
#   $v0: 0000 if not oob            #
#        0001 if top                #
#        0010 if right              #
#        0100 if bottom             #
#        1000 if left               #
#        0011 if top right          #
#        1001 if top left           #
#        0110 if bottom right       #
#        1100 if bottom left        #
```

This function negates a two’s complement number.

```
# Function: negate                  #
# Arguments:                        #
#   $a0: number to negate           #
# Return:                           #
#   $v0: negates number             #
```

This function checks for a collision between two sprites represented as rectangular boundaries.

```
# Function: check_collision         #
#                                   #
# Arguments:                        #
#   0(sf): sprite a start oam data  #
#   1(sf): sprite b start oam data  #
#   2(sf): height a                 #
#   3(sf): height b                 #
#   4(sf): width a                  #
#   5(sf): width b                  #
#                                   #
# Return:                           #
#   $v0: TRUE if collision          #
#        FALSE if no collision      #
```

This function moves all sprites associated with a single sprite image by the given offset.

```
# Function: move_sprite_img         #
#                                   #
# Defn: move sprite by specified    #
#   number of pixels.               #
#                                   #
# Arguments:                        #
#   0(sf): starting oam slot        #
#   1(sf): sprite_size              #
#   2(sf): x delta                  #
#   3(sf): y delta                  #
#                                   #
# Returns:                          #
#   0(sf) - top                     #
#   1(sf) - bottom                  #
#   2(sf) - left                    #
#   3(sf) - right                   #
```

This function loads all the sprites associated with an image starting at the given oam position and the given x and y coordinates.

```
# Function: load_sprite_img         #
#                                   #
# Defn: load sprite into oam        #
#                                   #
# Arguments:                        #
#   0(sf): sprite index             #
#   1(sf): sprite height            #
#   2(sf): sprite width             #
#   3(sf): left x (8 lsb)           #
#   4(sf): top y (8 lsb)            #
#   5(sf): starting oam slot        #
#                                   #
# Returns:                          #
#   $v0 - next free oam slot        #

# get_x: Gets X value from sprite register layout data
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#
# Returns:
#   $v0 - x value

# set_x: Sets x value in sprite register layout data
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#   $a1 - x data to set (lsb 8 bits)
#
# Returns:
#   $v0 - SRL data with new x

# get_y: Gets Y value from sprite register layout data
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#
# Returns:
#   $v0 - y value

# set_x: Sets y value in sprite register layout data
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#   $a1 - y data to set (lsb 8 bits)
#
# Returns:
#   $v0 - SRL data with new y
# get_tile_no: Gets tile number from sprite register layout data
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#
# Returns:
#   $v0 - tile number

# set_tile_no: Sets tile number in sprite register layout data
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#   $a1 - Tile number to set (lsb 8 bits)
#
# Returns:
#   $v0 - S.R.L. data with new tile number

# get_v_flip: Gets vertical flip from s.r.l.d.
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#
# Returns:
#   $v0 - vertical flip

# set_v_flip: Sets vertical flip in sprite register layout data
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#   $a1 - Vertical flip bit to set (lsb 1 bit)
#
# Returns:
#   $v0 - S.R.L. data with vertical flip

# get_h_flip: Gets horizontal flip from s.r.l.d.
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#
# Returns:
#   $v0 - horizontal flip

# set_h_flip: Sets horizontal flip in sprite register layout data
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#   $a1 - Horizontal flip bit to set (lsb 1 bit)
#
```

```
# Returns:
#   $v0 - S.R.L. data with horizontal flip

# get_color_palette: Gets color palette from s.r.l.d.
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#
# Returns:
#   $v0 - color palette

# set_color_palette: Sets color palette to s.r.l.d.
#
# Arguments:
#   $a0 - Sprite Register Layout formatted data
#   $a1 - Color Palette bits
#
# Returns:
#   $v0 - S.R.L. data with new color palette
```

The implementation of these functions can be found at the following link:

https://github.com/meggers/tronsister32-framework/blob/master/tronsister32_framework.asm

5 Assembler and Runtime Simulator (TARS)

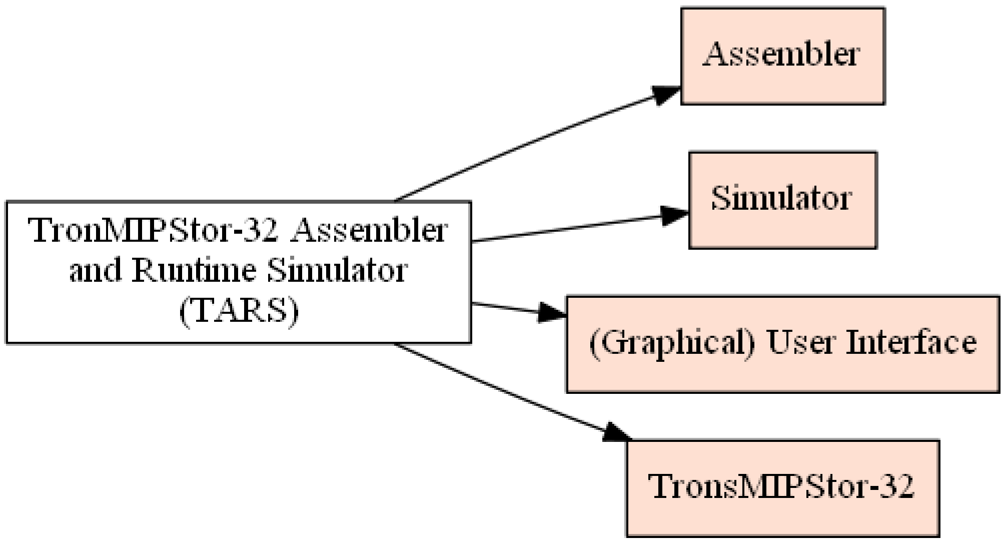

TronMIPStor-32 Assembler and Runtime Simulator (TARS)
Assembler
Simulator
(Graphical) User Interface
TronsMIPStor-32

Figure 5.0: Overview TARS implementation, which includes an assembler, simulator, and (graphical) user interface for our TronsMIPStor-32 ISA.

The high level implementation of TARS is shown in figure 5.0. TARS is heavily modified from MARS (MIPS Assembler and Runtime Simulator). As the name suggests, TARS is an assembler and runtime simulator for our TronsMIPStor-32 ISA. TARS is written in Java, which is portable to any machine capable of running a Java virtual machine. TARS is an alternative to the Tronsistor-32 framework used to test many software components, such as the COE format generator.

5.1 TARS Assembler

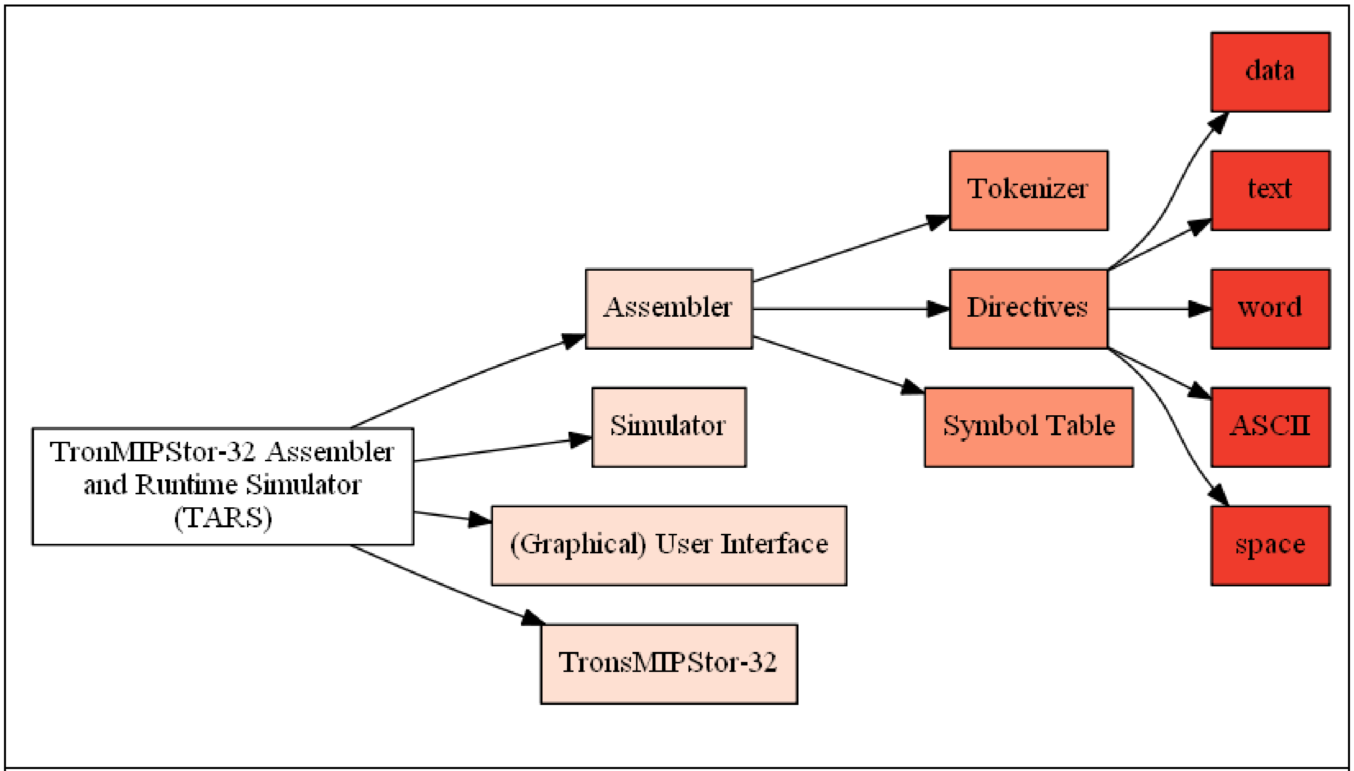
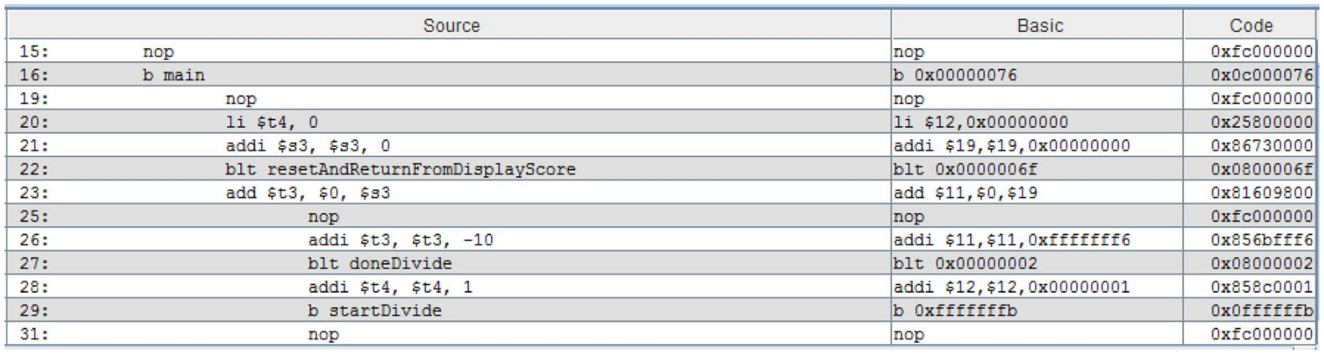


Figure 5.1.1: TARS assembler implementation, which includes an tokenizer to parse assembly language, directives to initialize data, and symbol table for branches and memory references.

The assembler converts assembly language to machine code. The assembler directives (data, text, word, ASCII, and space) are used to initialize data memory. All branches and jumps to labels are replaced with the relative line number. Any references sprites are replace with the index into the sprite pattern table.

Similar to the assembler in the Tronsistor-32 framework, the TARS assembler also does two passes through the assembly source code. The first pass of the assembler verifies syntax, generates symbol table, and initializes data segment. The second pass of the assembler translates assembly language to machine code.



Source		Basic	Code
15:	nop	nop	0xfc000000
16:	b main	b 0x00000076	0x0c000076
19:	nop	nop	0xfc000000
20:	li $t4, 0	li $12,0x00000000	0x25800000
21:	addi $s3, $s3, 0	addi $19,$19,0x00000000	0x86730000
22:	blt resetAndReturnFromDisplayScore	blt 0x0000006f	0x0800006f
23:	add $t3, $0, $s3	add $11,$0,$19	0x81609800
25:	nop	nop	0xfc000000
26:	addi $t3, $t3, -10	addi $11,$11,0xfffffff6	0x856bfff6
27:	blt doneDivide	blt 0x00000002	0x08000002
28:	addi $t4, $t4, 1	addi $12,$12,0x00000001	0x858c0001
29:	b startDivide	b 0xfffffffb	0x0ffffffb
31:	nop	nop	0xfc000000

Figure 5.1.2: TARS assembler example. The left column is the source code. The middle column is the basic instruction after the first pass. The right column is the machine code.

5.2 TARS Simulator

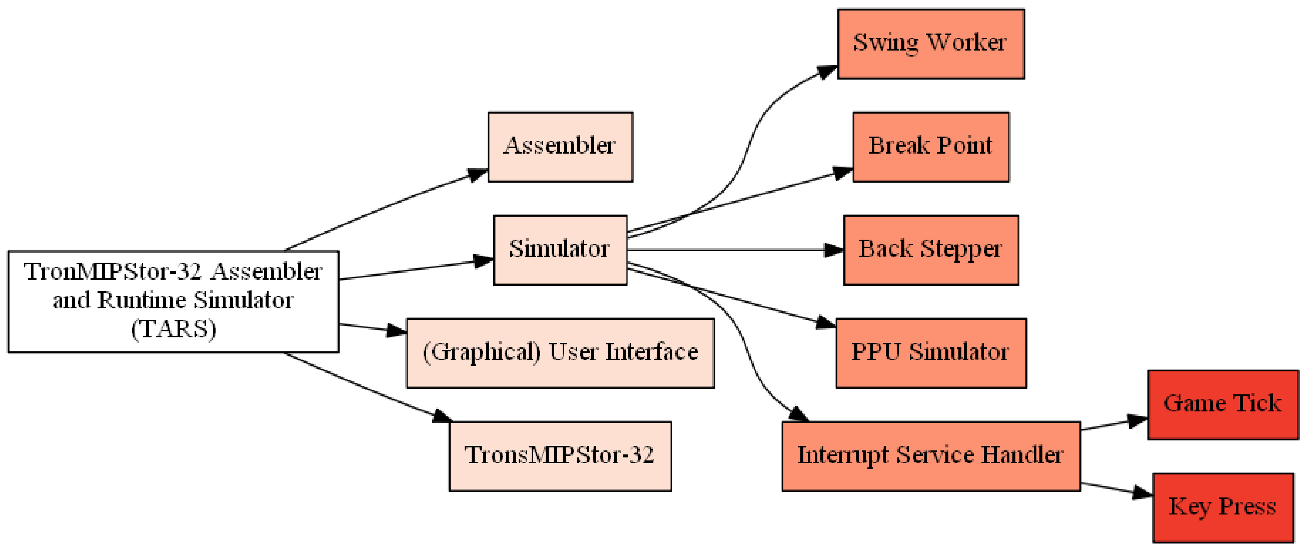
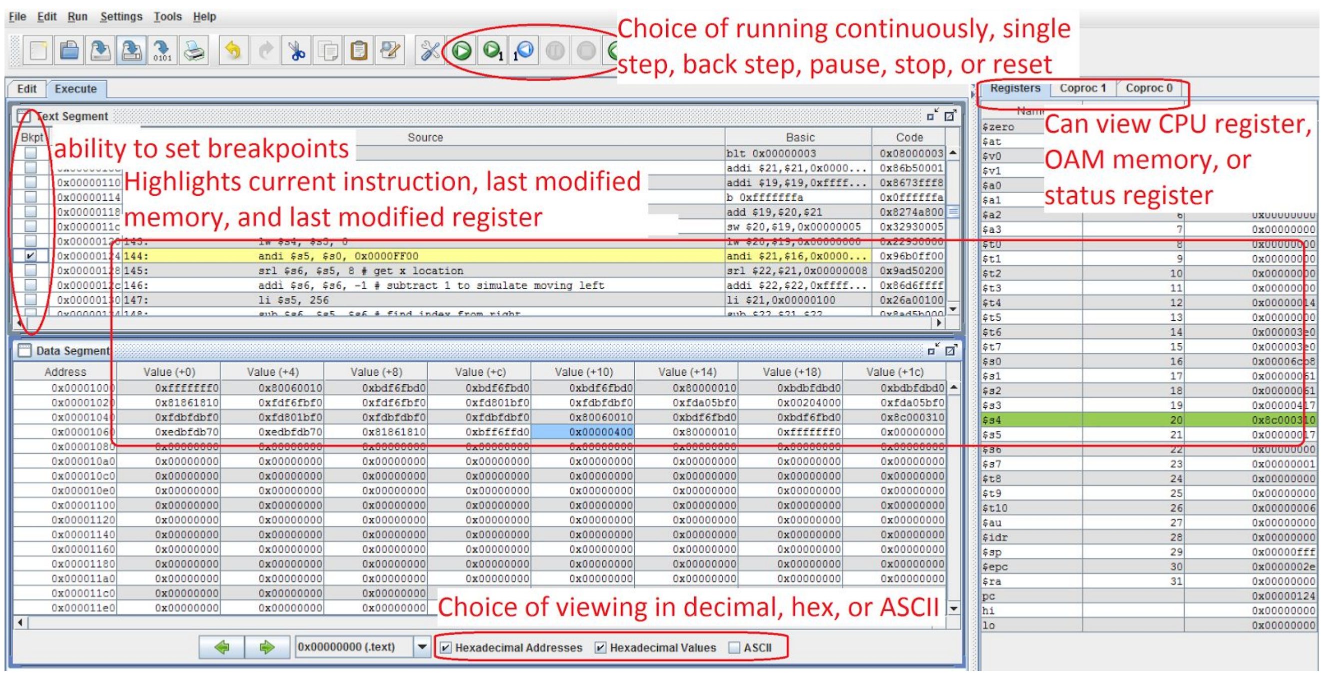

TronMIPStor-32 Assembler and Runtime Simulator (TARS)
Assembler
Simulator
(Graphical) User Interface
TronsMIPStor-32
Swing Worker
Break Point
Back Stepper
PPU Simulator
Interrupt Service Handler
Game Tick
Key Press

Figure 5.2.1: TARS simulator implementation, which includes a swing worker for parallelization, break point and back stepper for debugging, PPU simulator for visual, and interrupt service handler to simulate game tick and key presses.

The TARS simulator uses swing workers for high thread-level parallelization of TARS to increase performance. The breakpoints, single stepper, and back stepper works together for debugging a particular section of assembly code. Stepping through the code works for PPU instructions as well for the programming to test different visual data, color palettes, or flips on the screen. The interrupt service handler handles game tick and key press interrupts. A game tick interrupt occurs every 4096 cycles. A key listener triggers the key press interrupt when the user presses a key in the PPU screen. Both interrupts are handled right before simulating an instruction by checking the interrupt flags. Like hardware, upon an interrupt, the simulator will store the current PC and the status register and jump to the interrupt handler. The interrupt handler is finished upon executing jump to the address stored in the exception program counter register ("jr $epc"), in which case, the status register will be restored. Also like hardware, an interrupt cannot occur inside of another interrupt.


File Edit Run Settings Tools Help
Edit Execute
Choice of running continuously, single step, back step, pause, stop, or reset
Registers Coproc 1 Coproc 0
Can view CPU register, OAM memory, or status register
Text Segment
ability to set breakpoints
Highlights current instruction, last modified memory, and last modified register
Data Segment
Choice of viewing in decimal, hex, or ASCII

Bkpt		Source	Basic	Code
☐			blt 0x00000003	0x08000003
☐			addi $21,$21,0x0000...	0x86b50001
☐	0x00000110		addi $19,$19,0xffff...	0x8673fff8
☐	0x00000114		b 0xfffffffa	0x0ffffffa
☐	0x00000118		add $19,$20,$21	0x8274a800
☐	0x0000011c		sw $20,$19,0x00000005	0x32930005
☐	0x00000120	143: lw $s4, $s3, 0	lw $20,$19,0x00000000	0x22930000
✓	0x00000124	144: andi $s5, $s0, 0x0000FF00	andi $21,$16,0x0000...	0x96b0ff00
☐	0x00000128	145: srl $s6, $s5, 8 # get x location	srl $22,$21,0x00000008	0x9ad50200
☐	0x0000012c	146: addi $s6, $s6, -1 # subtract 1 to simulate moving left	addi $22,$22,0xffff...	0x86d6ffff
☐	0x00000130	147: li $s5, 256	li $21,0x00000100	0x26a00100
☐	0x00000134	148: sub $s6, $s5, $s6 # find index from right	sub $22,$21,$22	0x8ad5b000

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00001000	0xfffffff0	0x80060010	0xbdf6fbd0	0xbdf6fbd0	0xbdf6fbd0	0x80000010	0xbdbfdbd0	0xbdbfdbd0
0x00001020	0x81861810	0xfdf6fbf0	0xfdf6fbf0	0xfd801bf0	0xfdbfdbf0	0xfda05bf0	0x00204000	0xfda05bf0
0x00001040	0xfdbfdbf0	0xfd801bf0	0xfdbfdbf0	0xfdbfdbf0	0x80060010	0xbdf6fbd0	0xbdf6fbd0	0x8c000310
0x00001060	0xedbfdb70	0xedbfdb70	0x81861810	0xbff6ffd0	0x00000400	0x80000010	0xfffffff0	0x00000000
0x00001080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000010a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000010c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000010e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00001100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00001120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00001140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00001160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00001180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000011a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000011c0	0x00000000	0x00000000	0x00000000					
0x000011e0	0x00000000	0x00000000	0x00000000					

0x00000000 (.text) [✓ Hexadecimal Addresses] [✓ Hexadecimal Values] ☐ ASCII

Name		
$zero		
$at		
$v0		
$v1		
$a0		
$a1		
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000014
$t5	13	0x00000000
$t6	14	0x000003e0
$t7	15	0x000003e0
$s0	16	0x00006cb8
$s1	17	0x00000061
$s2	18	0x00000061
$s3	19	0x00000417
$s4	20	0x8c000310
$s5	21	0x00000017
$s6	22	0x00000000
$s7	23	0x00000001
$t8	24	0x00000000
$t9	25	0x00000000
$t10	26	0x00000006
$au	27	0x00000000
$idr	28	0x00000000
$sp	29	0x00000fff
$epc	30	0x0000002e
$ra	31	0x00000000
pc		0x00000124
hi		0x00000000
lo		0x00000000

Figure 5.2.2: TARS simulator example. The top middle gives user the choice of continuous run, single step, back step, pause, stop, or reset. The top right gives the user of view contents of the register file, OAM memory, or the status register. The left gives the breakpoint option at every line. The middle highlights the current instruction, last modified memory, and last modified register. The bottom gives the option to view in decimal, hexadecimal, or ASCII.

5.3 TARS User Interface

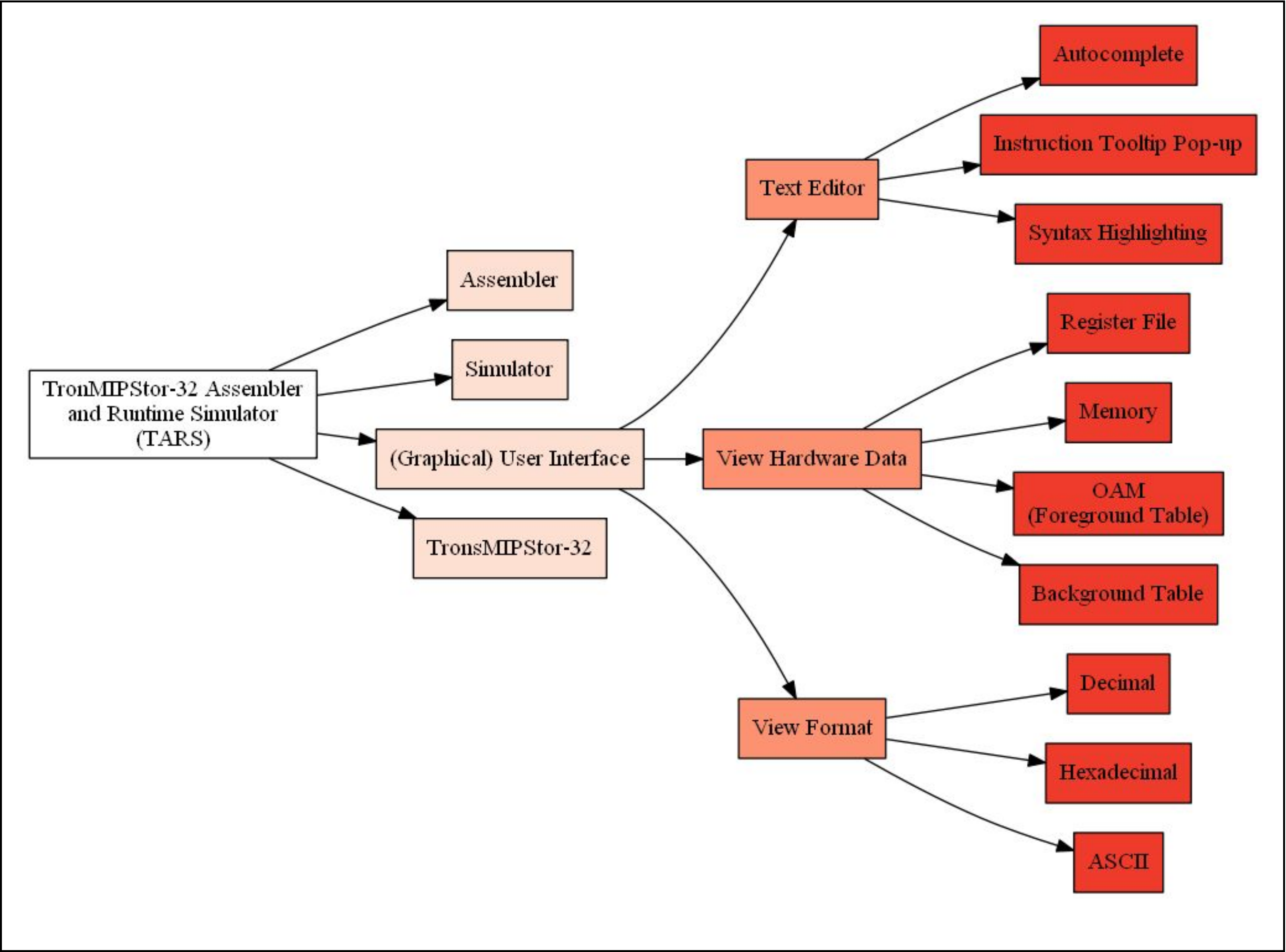
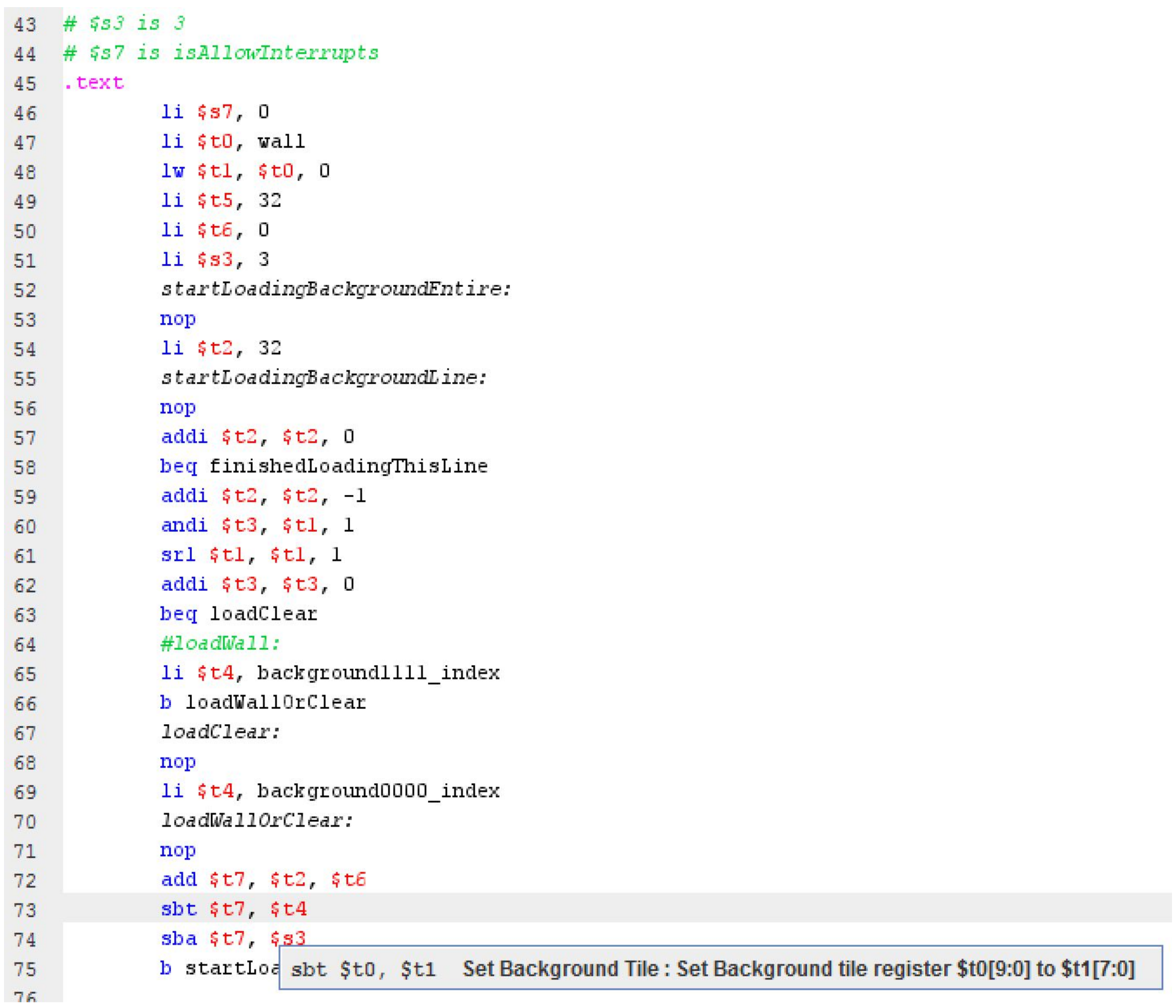


Figure 5.3.1: TARS user interface implementation, which includes a text editor to write assembly code, hardware viewer to view registers and memory for debugging, and format viewer for decimal, hexadecimal, and ASCII for debugging.

The TARS graphical user interface is written using the Swing Java, which includes a text editor and ability to view hardware data in multiple formats. The text editor has autocomplete, tooltip text for every instruction, and syntax coloring/highlighting for programmers to write their games. When simulating, the programmer can view and modify data in hardware in decimal, hex, or ASCII format for debugging.



```
# $s3 is 3
# $s7 is isAllowInterrupts
.text
        li $s7, 0
        li $t0, wall
        lw $t1, $t0, 0
        li $t5, 32
        li $t6, 0
        li $s3, 3
        startLoadingBackgroundEntire:
        nop
        li $t2, 32
        startLoadingBackgroundLine:
        nop
        addi $t2, $t2, 0
        beq finishedLoadingThisLine
        addi $t2, $t2, -1
        andi $t3, $t1, 1
        srl $t1, $t1, 1
        addi $t3, $t3, 0
        beq loadClear
        #loadWall:
        li $t4, background1111_index
        b loadWallOrClear
        loadClear:
        nop
        li $t4, background0000_index
        loadWallOrClear:
        nop
        add $t7, $t2, $t6
        sbt $t7, $t4
        sba $t7, $s3
        b startLoa sbt $t0, $t1    Set Background Tile : Set Background tile register $t0[9:0] to $t1[7:0]
```

Figure 5.3.2: TARS user interface example. This screen shows syntax coloring (assembler directives in pick, comments in green, instructions in blue, registers in red, and immediates in black) and tooltip help text for the current instruction that the user is typing.

5.4 TARS TronsMIPStor-32 ISA

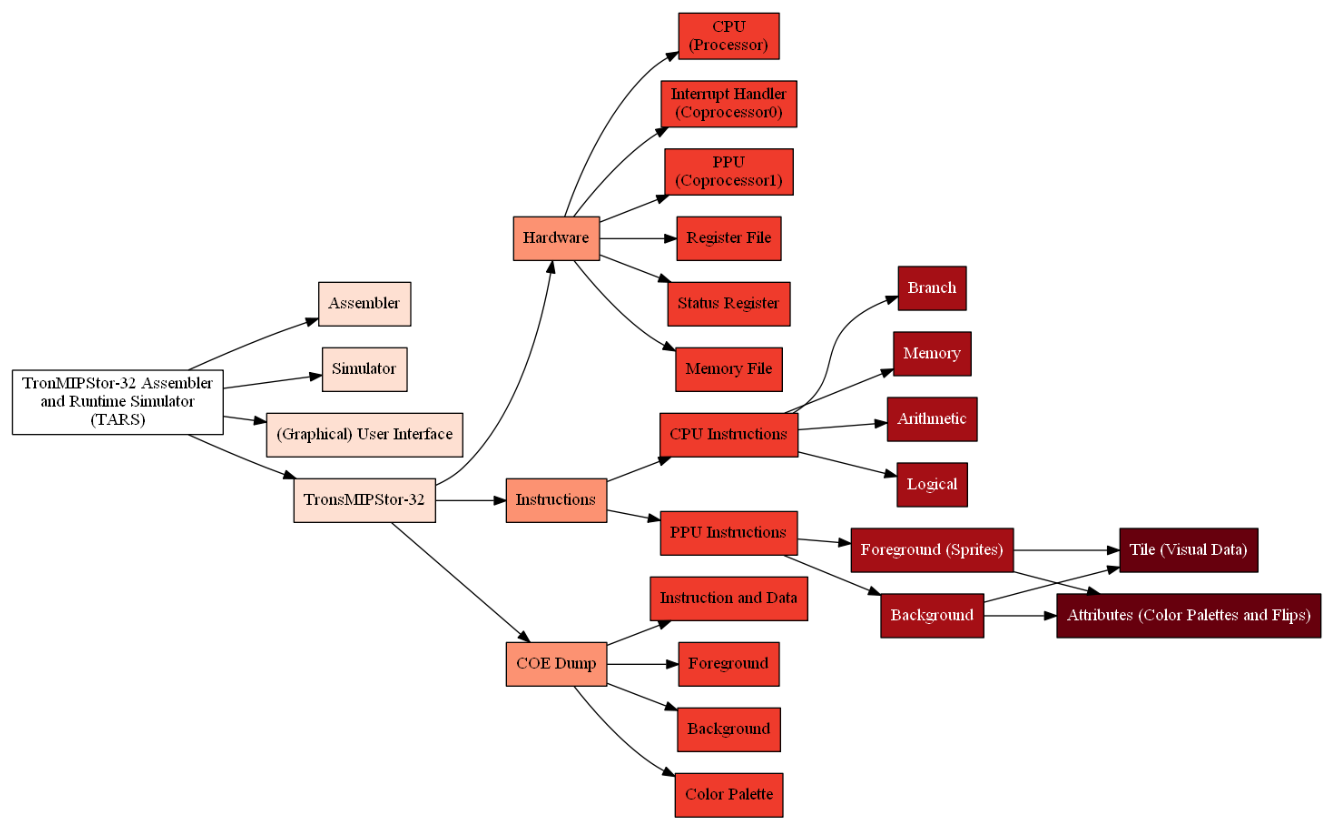


Figure 5.4: TronsMIPStor-32 ISA implementation in TARS, which includes hardware to manage data, instructions to manipulate data, and COE dump for Xilinx and FPGA.

TARS was heavily modify from MARS to change the core ISA from MIPS to TronsMIPStor-32.

In hardware: Coproessor1 was repurposed as the OAM while coprocessor0 was repurposed as the status register and interrupt handler. The register names and initial values were updated to match TronsMIPStor-32. The memory file was modified from byte-addressable to word addressable. The carry/negative/zero flag was added as a status register.

In instructions: Processor was updated to support arithmetic, logical, branch, and memory instructions, and remove pseudo instructions. Coprocessor1 was updated to handle and simulate PPU foreground and background tile and attribute instructions.

In COE Dump: The COE format was added to dump instruction and memory data, foreground data, background data, and color palette data.

6 Games

The main purpose of the TronsMIPStor-32 ISA is to develop retro games. Therefore, multiple games are developed to demonstrate that our ISA and architecture is capable of supporting many retro games.

6.1 Etch a Sketch

Etch a sketch is a demo program used to debug hardware. There is 1 sprite that the user controls, which leaves behind a trail of background tiles.

6.2 Tron

Tron is a computer video game where users controls the direction of a sprite that continuously moves and leaves a wall/trail behind. The objective of the game is to cut-off/trap the opponent so that they would crash. Last player to not crash wins the round.

6.3 Pong

Pong is one of the earliest arcade video games. It simulates a tennis game with a ball and paddles in simple two-dimensional graphics. Each player on either side controls their paddle to move up and down to hit the bouncing ball to their opponent's side. The opponent scores a point if the player misses with the paddle. Spins and velocity are supported in Pong in TronsMIPStor-32.

Video of Pong gameplay (without user input) can be found at the following link:

https://www.youtube.com/watch?v=oM4yzaYnHYI

Implementation of the Pong game can be found at the following link:

https://github.com/meggers/tronsister32-pong

6.4 Dance Dance Revolution (DDR)

DDR is a game where the player matches the steps/keys that scrolls on screen. Pressing the corresponding key at the correct time interval (above the blue line) will result in a point. Otherwise, the player will lose a point. A random number generator is supported by XORing the score, an internal counter, and the current letter.

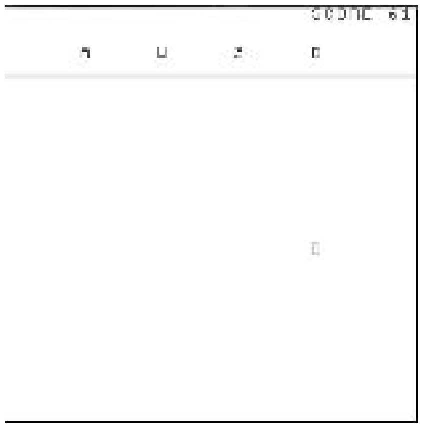


Figure 6.4: Screenshot of DDR in the PPU simulator.

6.5 Maze Navigator (beginning of Pacman)

Pacman in a game where the player controls Pac-Man through a maze to eat pac-dots. Maze navigator is just the beginning of Pacman, where the user can navigate around a maze. Collision detection is supported, so the player will navigate through the maze faster if he/she doesn't crash.

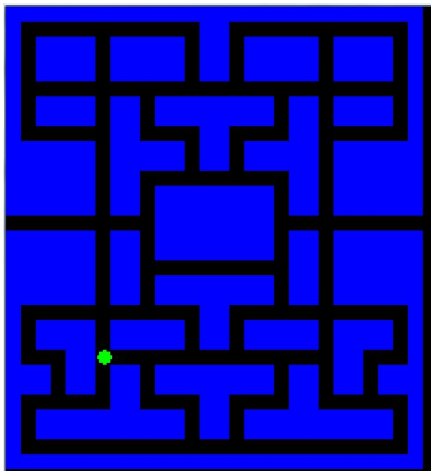


Figure 6.5: Screenshot of maze navigator in the PPU simulator.

7 Team Contributions

7.1 Hardware Team

7.1.1 Graham Nygard: Signed: GRAHAM NYGARD Date: 12/22/2015

Acting team leader throughout the semester. Contributed heavily to designing and maintaining the Tronsister-32 ISA. Designed and implemented the core pipeline stages of the Central Processing Unit (CPU), and connected them all at the CPU top level. Performed the initial testing of the CPU and integrated changes made by Jake Truelove to the overall design throughout its development. Verified that the CPU was functionally correct and synthesizable after each architecture update, and made changes where necessary. Initiated integration of the final top level design and made considerable contributions to debugging the system.

Verilog Modules: Instruction Fetch, Instruction Decode, Instruction Execute, Memory Unit, Writeback Unit, all pipeline registers, CPU Control, PC Control, Hazard Detect, CPU/PPU Interface, 32-bit Register File.

7.1.2 Jake Truelove: Signed: JAKE TRUELOVE Date: 12/22/2015

Contributed heavily to designing and maintaining the Tronsister-32 ISA. Made the initial memory interface of the CPU that served as the connection between the pipeline and the main memory. Updated the CPU with data-forwarding and interrupt capabilities, which necessitated changes throughout many stages of the pipeline. Verified the functional correctness of the design after each addition to the CPU before submitting these changes to Graham Nygard. Aided in debugging the system.

Verilog Modules: Memory Interface, Data Forwarder, Trap Handler, ALU (made appropriate changes to other modules affected by data-forwarding and interrupt handling).

7.1.3 Matt Kelliher: Signed: MATTHEW KELLIHER Date: 12/22/2015

Contributed heavily to designing and maintaining the Tronsister-32 ISA. Designed and implemented the Picture Processing Unit (PPU) and integrated this module with each of the memory units used for the graphical renderings in each of our programs. Verified that sprite layouts were being displayed appropriately on a screen with proper orientation, layering, scaling, etc. Made considerable contributions to integrating the final top level design and debugging the system.

Verilog Modules: PPU Control, Object Attribute Memory, Foreground Pattern Table, Background Pattern Table, Background Attribute Table, Color Palette.

7.2 Software Team

7.2.1 Max Eggers: Signed: MAXWELL HENRY RHODES EGGERS Date: 12/22/2015

- Contributed heavily to designing and maintaining the Tronsister-32 ISA.
- Wrote the Tronsistor-32 framework
  - Assembles from TronMIPStor-32 assembly to machine code
  - Generates COE files
    - Main Memory
    - FG/BG Sprite Palettes
  - Wrote many functions to ease assembly game programming
    - Load and display sprites
    - Move sprites
    - Check collision
    - Displaying text using background tiles
- Drew and generated sprite visual data
  - ASCII characters and pong/tron foreground sprites
- Wrote Pong

7.2.2 Kai Zhao: Signed: KAI ZHAO Date: 12/22/2015

- Contributed heavily to designing and maintaining the Tronsister-32 ISA.
- Maintained TARS CPU simulator

Wrote and maintained TARS PPU simulator

Wrote TARS interrupt handler simulator

Wrote TARS assembler for sprite instructions

Wrote TARS COE generator

Wrote Tron logic in MIPS to determine the set of instructions our ISA needs

Wrote Tron in a high level programming language

Wrote DDR

Wrote a maze navigator (the beginning of Pacman)

7.2.3 John Roy: Signed: JOHN ROY Date: 12/22/2015

Contributed heavily to designing and maintaining the Tronsister-32 ISA.

Modified MARS to TARS

Added TronsMIPStor-32 CPU instructions

Removed MIPS (pseudo) instructions

Wrote etch a sketch

Wrote Tron logic in TronsMIPStor-32 ISA

Wrote Tron graphics

8 Challenges Faced and Lessons Learned

Integrating the main modules of the design, the Picture Processing Unit and the Central Processing Unit, was arguably the greatest challenge that our group faced this semester. Each module appeared to function correctly as a standalone entity throughout the design process, but the integration of these two units revealed a considerable amount of bugs within the design that had not been foreseen while the modules were separated. For example, the CPU developers had not considered the need for data-forwarding of sprite instruction data. This necessity was revealed after we had integrated the two designs and realized that each of the sprite instructions was feeding garbage data to the PPU, because the desired data had not yet been written back into the register file.

Although it was beyond our control, our inability to reliably instantiate new IP cores throughout the design process posed a considerable challenge throughout the semester. Each time the instantiation process would fail, it would take roughly 20-30 minutes to complete the process through trial and error on other machines throughout the lab.

Another notable challenge was that there were intrinsic differences in behavior between our synthesized hardware and our custom MARS simulator, which was used to verify the functional correctness of our assembly programs. Because each program was initially tested and debugged using the simulator, assumptions that the simulator made about the underlying hardware sometimes caused issues for the programs as they ran on the actual FPGA. For example, one difference that was noticed late in development was that the simulator began each program by defaulting all of the register contents to 0x00000000 (aside from the Stack Pointer, initialized to 0x00000FFF). However, the actual hardware was implemented assuming that a program must first initialize a register before its contents can be reliably known and used. Thus, if a program were to use an uninitialized register during its execution, the behaviors of the simulator and the hardware would diverge. The simulator's program would function normally, but the hardware's program would would often fall into an unrecoverable state (Program Counter out of bounds, heap data accesses inadvertently access instruction space, inconsistent state of the flags registers used for branch condition verification, etc.). We strived to resolve these issues immediately as they were discovered, but the complexity of the CPU's design often made it difficult to pinpoint the source of any inconsistencies between the simulator and hardware.

Our custom ISA had to be simple because it needs to be implemented in hardware. Therefore, it is worth noting that writing fully functional games in an assembly language using a custom ISA is incredibly time consuming and requires an acute attention to detail. To put this into perspective, consider some of the following statistics gathered from a few of our completed games. Excluding comments and metadata (data used to assemble the game code), the game 'TRON' had roughly 1000 lines of CPU instructions. Likewise, the game 'PONG' had around 800 lines of instructions, and a game as simple as 'Etch a Sketch' had nearly 100 lines of pure assembly instructions.

A lessons learned is that our PPU graphics instructions should be memory mapped as opposed to writing to a separate OAM memory file. Doing so will make it easier to debug as we can initialize the background tiles with assembler directives as opposed to using many lines of assembly code to do so. Using memory mapped OAM would have also decrease the hardware overhead.