EE 552 Project Final Report

Game Core Classic

Video Game System

Powered by

PIGME

(Programmable Integrated Graphics Microprocessor Environment)

Donson Lam	donson@ualberta.ca
Jason Klaus	jwklaus@ualberta.ca
Leendert Van den Berg	leendert@ualberta.ca
Brian Eley	beley@ualberta.ca

Declaration of Original Content

The design elements of this project and report are entirely the original work of the authors and have not been submitted for credit in any other course except as follows:

- Integer to std_logic_vector conversion function obtained from Associated Professional Systems. Accessed http://users.erols.com/aaps/x84lab/INCLUDE.html on February 17, 2003
- A64x16.vhd: Modified Cypress Semiconductor Corp. VHDL model for their CY71020CV33 32k x 16 SRAM. Modifications include the increasing of the capacity to 64k x 16, addition of appropriate timing information, and file name changing.
- Joystick Circuit based on circuit from Epanorama.net Accessed http://www.epanorama.net/documents/joystick/pc_joystick.html. February 18, 2003
- VGA timing calculator spreadsheet from VESA Accessed http://www.vesa.org/public/SMT/SMT640_720x480v1.xls. February 18, 2003
- VGA timing signal information from EE552 appnote Accessed http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/2002_w/interfacing/CRT/CRT_App_Note.html. February 17, 2003
- Xilinx Spartan II Primitives instantiation templates. Accessed http://toolbox.xilinx.com/docsan/xilinx5/data/docs/lib/lib0024_8.html March 1, 2003.

Donson Lam ______________________

Jason Klaus ______________________

Leendert Van den Berg ______________________

Brian Eley ______________________

Abstract

Imagine being able to sit down for an evening and develop a classic video game with nothing more than simple sprites and assembly language. Imagine being able to play that same game on a common computer monitor in resolutions higher than most consoles without fear of losses in frame rates when the action heats up. Imagine sharing your creations with your friends, and going head to head in some two player action. Think this is all too good to be true? Well, this project is here to prove you wrong.

Gamecore classic is all this and more. It is a custom CPU with an integrated sprite and rectangle graphics unit programmed into a Xilinx Spartan II FPGA. The console supports two joysticks and displays 60Hz graphics on a standard VGA monitor. And best of all, it ships standard with a powerful and flexible assembler for creating your own games using your PC.

Table of contents

Achievements

- Designed and implemented a 16-bit CPU from scratch.
- Created a powerful assembler for the PIGME CPU.
- Created a graphics processor unit capable of displaying sprites and rectangles on a VGA monitor.
- Successfully interfaced to a standard PC joystick
- Produced a printed circuit board for 256KB of external 10ns SRAM. Difficulties in soldering the surface mount J-Lead packages prevented the use of this SRAM and internal block ram was substituted.
- Scalable design makes use of many constants and type declarations to allow for increased performance in larger FPGAs and for reduced performance in smaller FPGAs.

Description of Operation

System Overview

The main idea for the GAME CORE system was to offer a simple video game system, primarily tailored for 2D games like those seen on early gaming consoles of the 1980s. Real time arcade games can be written for the system using a straightforward assembly language.

The gamecore system uses two standard PC joysticks for input, and outputs graphics to a standard VGA monitor. A set of LEDs and a seven-segment display is available for displaying scores and for debugging programs. An RS-232 UART is available to aid in program debugging. A reset button has also been included.

The system is composed of the following units:

- Custom 16-bit Central Processing Unit (CPU).
    - ALU
    - Register File
    - Control Section
- Custom VGA Graphics Processing Unit (GPU) capable of sprite and rectangle generation.
    - VGA timing generator
- Memory Unit based on internal SRAM implementing independent memories.
    - Program and data storage
    - Graphics object property storage
    - Sprite graphics data
- IO Bus
    - Joystick controller
    - LED and 7-Segment Display controller
    - UART for serial communications

The gamecore top-level diagram is shown below in figure 1.1.

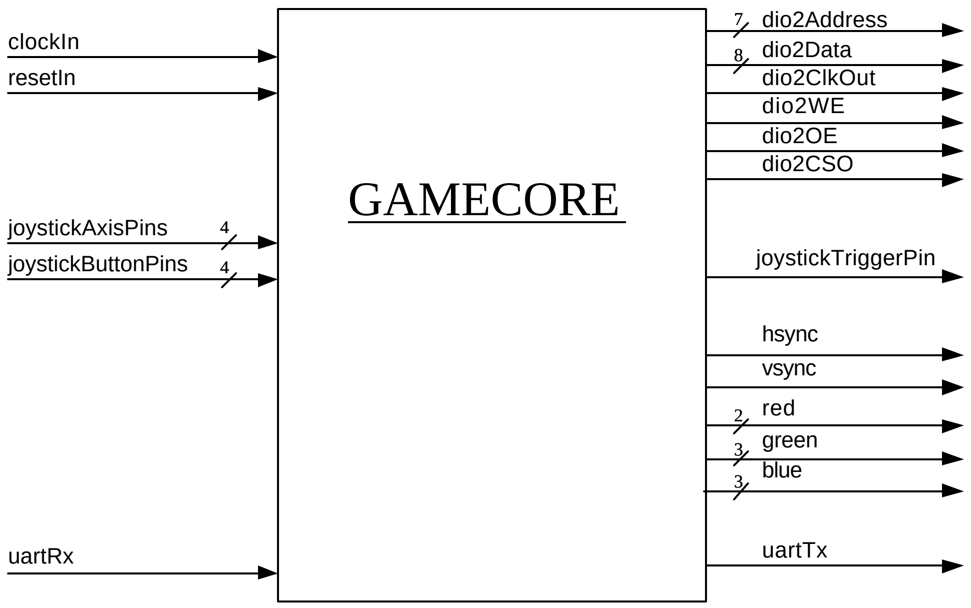


Figure 1.1: **Gamecore Top Level**

The CPU has been designed completely from scratch for use in the Gamecore system. It has the following groups of instructions:

- Integer arithmetic (add, subtract, multiply, divide)
- Graphic co-processor: Rectangle and Sprite generation and modification
- I/O Bus: In Port and Out Port to peripheral devices
- Load, Store
- Unconditional Branch, Branch on condition
- Branch to subroutine, return from subroutine
- Boolean: AND, OR, XOR, NOT, negate (2's complement)
- Bit-wise: Arithmetic and Logical Shifting, Rotates
- No operation (NOP)
- Stop to halt the CPU

The CPU operates solely on 16 bit quantities, this means that instruction opcodes, SRAM addresses, SRAM data are all 16 bit quantities (whether RAM is actually present is another issue). To simplify the CPU design, the CPU is not pipelined. A non-pipelined CPU reduces the instructions per clock cycle, but since the graphics co-processor performs many of the graphics functions, a high throughput CPU is not required. The main task of the CPU will be handling user input and updating graphics objects on the graphics co-processor. A branch to subroutine and return from subroutine call is implemented using a hardware-based stack for return addresses. It is not possible for the programmer to get or set the value of the program counter, so no software stack can be implemented on our CPU.

The CPU communicates with the graphics co-processor through a dedicated video bus. The graphics co-processor has support for sprites, rectangle objects, and a background color. The sprites and rectangles can have their properties updated by the CPU. Their list of properties includes x and y coordinates of the left, right, top, and bottom edges, as well as color for rectangles and video memory base address for sprites. Sprites and rectangles are assigned to one of 4 graphics layers. Objects in higher layers are drawn over top of objects in lower layers. Additionally, one color has been assigned as transparent and will allow objects in lower layers to show through.

The graphics controller runs at a fairly high speed compared to the CPU to ensure that the screen gets refreshed frequently. A standard 640x480 pixel VGA display is drawn at 60Hz, with each pixel encoded as 8 bits. The VGA connector on the Digilent IO board is hard wired with 3 bits of blue, 3 bits of green, and 2 bits of red. The graphics controller is a pipelined highly parallel graphics processor.

The CPU communicates with the slower I/O devices through a simple I/O bus. Attached to the IO bus is a joystick controller, an RS-232 serial port, and an LED and 7-segment display controller. I/O is not memory mapped. The I/O bus is accessed using special opcodes: PIN and POUT, which act much like load and store, with the exception being that both the address and data bus for the I/O bus are not as wide as the SRAM busses.

Figure 1.2 shows a high-level internal system block diagram.

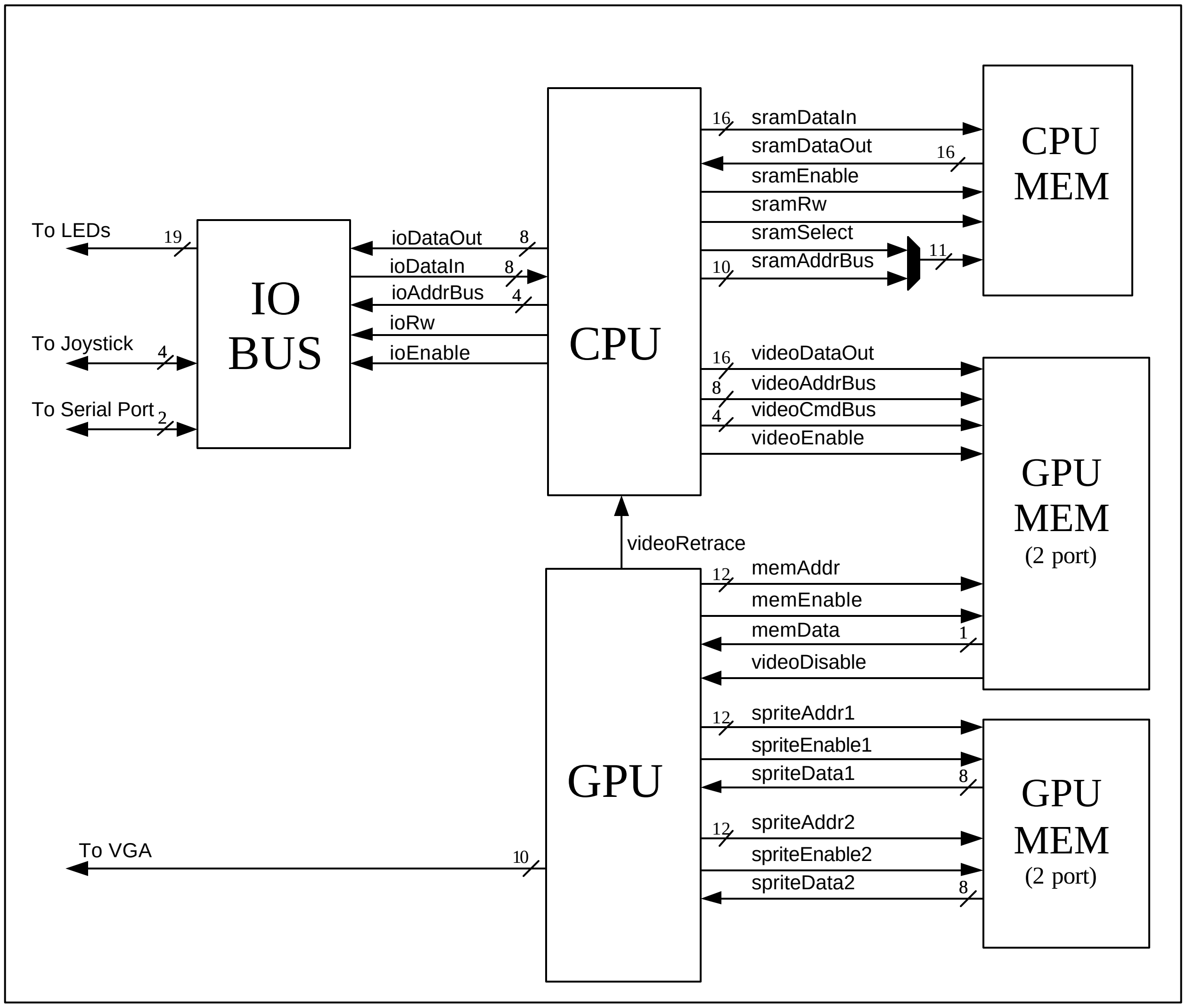


Figure 1.2 – **High Level Internal System Block Diagram**

Instruction Set

The PIGME instruction set load and store architecture is based on operations with 16 bit, signed 2's compliment quantities. For this reason, the term "word" will be used to denote a 16 bit quantity for the remainder of this specification. All operations involve register to register transactions except for loads and stores, which can involve data memory or immediate values.

Registers

There are a total of sixteen 16 bit registers defined by PIGME, with the following mnemonics:

- **`r0`** through **`r12`** are general purpose registers
- **`rt`** is a temporary general purpose register modified by certain operations
- **`rz`** is a read-only register hard-wired to **`0x0000`** (decimal **`0`**)
- **`rn`** is a read-only register hard-wired to **`0xFFFF`** (decimal **`-1`**)

The use of **`rz`** and **`rn`** as hard-wired constants not only allows access to the two most used program constants without the overhead of a **`LOAD`** immediate beforehand, it also allows the assembler to provide a richer assortment of instructions like **`CLR`** and **`INC`** without the addition of any new op codes or hardware. The savings in op codes is particularly important as instruction words are of limited size.

Memory

PIGME supplies separate instruction and data memories for both expanded memory space and added security. Instruction fetching and branching automatically refer to instruction memory, while load and store operations automatically refer to data memory, preventing a program from modifying its own code.

Both instruction and data memory consist of 2^{16} = 65536 addressable words. There is no concept of a "byte" or "long word", since memory quantities are always 16 bit words. This means that PIGME is endian independent.

Instructions

Each PIGME instruction word (16 bits) can be divided into four 4 bit fields consisting of:

- an operation code specifying the instruction or instruction type
- an instruction defined field
- a register identifier field
- another register identifier field, which may or may not be used

0 3	4 7	8 11	12 15
Op Code	Inst. Defined	Register	Register

General Instruction Word Format

Op codes and the instruction defined fields are listed along with each instruction in the instruction reference. Registers are always encoded in the following manner:

- **`r0`** through **`r12`** as **`0x0`** through **`0xC`** (**`0b0000`** through **`0b1100`**)
- **`rt`** as **`0xD`** (**`0b1101`**)

- **`rz`** as **`0xE`** (**`0b1110`**)
- **`rn`** as **`0xF`** (**`0b1111`**)

Only the **`LOAD`** and **`STORE`** instructions involve effective addressing modes, all other operations act on registers. A list of the effective addressing modes can be found under the descriptions of the **`LOAD`** and **`STORE`** instructions in the instruction reference.

Assembly Syntax

A full list of every microprocessor instruction and its assembly syntax can be found in the instruction reference. Each instruction must appear on a single line by itself, prefixed with one or more optional labels and optional white space. Labels can appear on lines without instructions, just as blank lines consisting of only white space are allowed as well. The apostrophe (‘**'**’) can appear anywhere in a line and indicates that the remainder of the line is a comment. Multi-line comments are not supported.

An assembly instruction consists of the instruction mnemonic followed by zero or more arguments, with a comma (‘**,**’), white space or both separating adjacent arguments. Instruction and register mnemonics are case insensitive. Label names, like all other placeholders, must consist of a sequence of alpha-numeric characters (‘**A**’-‘**Z**’, ‘**0**’-‘**9**’, ‘**_**’) beginning with an alphabetic character, and are also case insensitive. White space and other punctuation characters are not allowed in placeholder names. Labels, unlike other placeholders, also must be terminated by a colon (‘**:**’).

Aside from register mnemonics, the only other valid arguments to an instruction are:

- a label name as an argument to a branch instruction
- a register mnemonic in parenthesis to indicate indirect addressing for a **`LOAD`** or **`STORE`** instruction
- an absolute address for a **`LOAD`** or **`STORE`** instruction specified as a number constant or number constant placeholder
- an immediate value for a **`LOAD`** or **`STORE`** instruction specified as a number constant, number constant placeholder, or image placeholder, prefixed by a pound sign (‘**#**’)

Number constants must be prefixed appropriately if in a base other then decimal. The prefixes are:

- **`0x`** or **`$`** for a hex number constant (digits: ‘**0**’-‘**9**’, ‘**a**’-‘**f**’, ‘**A**’-‘**F**’)
- **`0`** for an octal number constant (digits: ‘**0**’-‘**7**’)
- **`0b`** for a binary number constant (digits: ‘**0**’, ‘**1**’)

Number constant placeholders attach a name to a particular constant or data memory address. Placeholder definitions are case insensitive and begin with a period (‘**.**’) followed by one of the following:

- **`const <placeholder> <number constant>`**
- **`word <placeholder> [value]`**
- **`array <placeholder> <size> [value [...] ]`**

The **`.const`** definition defines a placeholder (name) which is associated to a particular number constant which is automatically substituted into instructions by the assembler. The **`.word`** definition defines a placeholder (name) which is associated to an arbitrary data memory word, whose initial value can be specified through the optional **`value`** parameter. The **`.array`** definition defines a placeholder (name) which is associated to an arbitrary array of **`size`** data memory words, whose initial values can be specified through the optional **`value`** parameters. Both **`.word`** and **`.array`** definitions are automatically mapped to data memory by the assembler, and should be used instead of absolute address number constants in **`LOAD`** and **`STORE`** commands. The optional **`value`** parameters follow the same format as number constants. Data values are initialized to zero if not specified.

In addition to data definitions, the **`.image <placeholder> <file>`** definition loads **`file`** into video memory, associating its video memory address to **`placeholder`**. By using this placeholder, prefixed by a pound sign, as an immediate value to a load instruction, this video memory address can be passed on to a sprite via a video instruction. For more information, see the instruction reference.

All **`.const`**, **`.word`**, **`.array`** and **`.image`** placeholder names must be unique irrespective of case. They can, however, share the same name as a label. Label names must also be unique irrespective of case.

CPU Architecture

The CPU is divided into a control section, which will be described behaviorally and data path, which will for the most part be described at the register transfer level. The data path diagram is available in the appendix. Currently, the data path is separated into the following sections:

- Instruction Fetch – Connected to instruction memory, contains the program counter and the instruction register as well as an adder that calculates the next increment of the PC every cycle. Also a multiplexer is present to select between a branch target or the incremented value of the PC. In order to support subroutine functions, a hardware stack that stores the return address for subroutine calls must be added. This hardware stack is a simple 16-bit wide bi-directional shift register that simply shifts in additional return addresses when a branch to a subroutine is executed.
- Instruction Decode – This unit is also connected to instruction memory. The instruction fetch stage contains the two operand registers used by the execution unit, the branch target register and the entire register file. A set of multiplexers selects what is loaded into the operand registers. Either the next word from instruction memory is loaded (for immediate value and absolute addressing modes) or a value is loaded from the register file (for register direct and register indirect modes) into each of the operand registers.
- Instruction Execute – This unit is connected to data memory, the I/O bus and the video unit. Inside the execute stage there are several sub modules each dedicated to a class of instructions:
    - ALU – performs logical and arithmetic calculations
    - Load/Store unit – performs loads and stores to and from data memory
    - I/O unit – performs loads and stores on the I/O bus
    - Branch unit – performs a comparison on the input operands and returns success
    - Shift unit – performs rotate, arithmetic shift and logical shift operations

Since several of the units in the execution stage produce outputs that are destined for general purpose registers, a multiplexer is described in the execution stage that selects what unit's output is written to the register file.
The ALU is described behaviourally, since the addition, subtraction and Boolean logic operations are most easily expressed in this manner. The multiply unit is generated from a Xilinx core and is connected within the ALU. The ALU does not currently register any of its inputs and outputs, and is purely combinational.

The load/store unit is also described behaviourally; it simply routes the incoming operand registers to data memory address and data buses. The control unit determines when to stall for one additional cycle for the memory data to become available.

The I/O and video instructions are very simple to implement, as they only route the operand registers from the instruction decode stages onto the data and address busses of the video and I/O devices. The control section generates the enable signals for the video and data busses.

The branch unit is composed of two comparators: one that compares the operands for equality and one that compares if operand one is greater than operand two. From the output of these two comparators any comparison type can be made using some additional combinational logic. The result from this comparison will then determine if the program counter should reload with the branch target address or if it should continue with the next instruction (which is already stored in the program counter register).

The shift unit is composed of a barrel shifter and some additional logic before the barrel shifter to select the proper input data depending on the type of shift operation. The barrel shifter can rotate its inputs an arbitrary number of bits to the left. To rotate the input data to the right, the barrel shifter simply rotates its input data to the left by (barrel shifter input width – number of places to shift to the right). For instance, to shift data 3 bits to the right in a 16-bit shifter, the barrel shifter really rotates its input 13 bits to the left, which achieves the same effect. To enable arithmetic and logic shift, some additional logic is created before the barrel shifter inputs that imitates either zeros shifting in from the left or right or imitates the MSB shifting in from the left (in case of the arithmetic shift).

For the register file it was decided that the read ports were best implemented using a tri-state bus directly driven by each register. The other option is to create a large 256 to 16 multiplexer for each read port, but this would require a large amount of logic and would probably offer little speed advantages if any (since for such a large multiplexer multiple levels of logic are required). Instead the design uses a tri-state bus and a set of 4 to 16 decoders that select which register can output its value on the read ports. The same design for a 4 to 16 decoder is used to select what register is written to on the write port. A separate write port is required for the temp register which contains an extra 16 bit word output from either a multiply or divide operation because both the divide and multiply produce a 32 bit result (either the upper 16 bits of the multiplication or the remainder of a divide is stored in the temp register) should be written in the same clock cycle as the other result from the operation. This also automatically means that the temp register cannot be addressed from the normal write port select, so that no CPU operation can explicitly write to the temp register.

There is some opportunity in the execution unit of the CPU to use some externally provided HDL components. In particular, there are some pipelined division and multiplication units available as IP cores from Xilinx. Utilizing these cores reduces some of the design that would normally be required on the fairly involved logic for these operations.

CPU Control Unit

The control unit is described behaviourally and consists of one finite state machine that determines what registers are enabled at what time. The following states are currently present in the finite state machine:
reset_cpu
mem_stable
pc_load
load_ir
stall_op2
load_op_exec
load_stall
stopped

Because the memory controller's inputs are registered, the CPU control unit must always ensure that the data on the address bus is valid before the rising edge of the clock, it can then latch the read data from the memory on the next clock edge.

When the asynchronous reset is selected, the CPU always goes to the reset_cpu state. Since this reset is asynchronous, it can occur right before the clock edge. When a reset occurs, the address lines to instruction memory may not have been given enough time to stabilize to be guaranteed a valid result the next clock cycle. Therefore, it progresses to the mem_stable state so that after this cycle the addresses to the SRAM are stable. This means that after the mem_stable state, the SRAM data lines contain the next instruction.

From the mem_stable state, the next state is always pc_load. When the pc_load state is entered, the instruction register is enabled so that at the next clock edge the instruction register will hold the instruction to be executed. Also, the program counter register is enabled so that it will load the address of the next instruction on the next clock edge.

After the pc_load state, the load_ir state is entered. When the state becomes ir_load, the instruction register has just loaded. Based upon the instruction currently loaded, a few things can happen. If an additional instruction word is required (for example immediate values, absolute addresses and branch targets), the program counter must be incremented again on the next cycle (note that the program counter always points to the next instruction). Also, the operand registers must be enabled to load the operand values (either from memory or the register file). If a branch instruction is being executed, the branch target register must also be loaded. When the instruction is a load or store command, an additional cycle is required for the instruction memory to fetch the extra data, and the next state entered will be stall_op2. If the instruction is a NOP or STOP, the CPU can fetch the next instruction or stop executing instructions all together. In any other case the next state entered will be load_op_exec.

When stall_op2 is entered, it means that the current instruction is a load or store instruction and an additional word from instruction memory is required. All that the stall_op2 state has to do is enable one of the operand registers so that the operand register loads the additional word from memory.

When load_op_exec is entered, it means the operand registers have just latched their values. Since the execution units are directly connected to the operand registers and are purely combinational, the result is available as soon as the next clock edge arrives. This means the target result register in the register file must be enabled so that it will store the result from the execution unit. The only case in which the register file write port must not be enabled yet occurs when a load command is executed. A load instruction consumes one additional cycle because it has to wait for the memory data to become available.

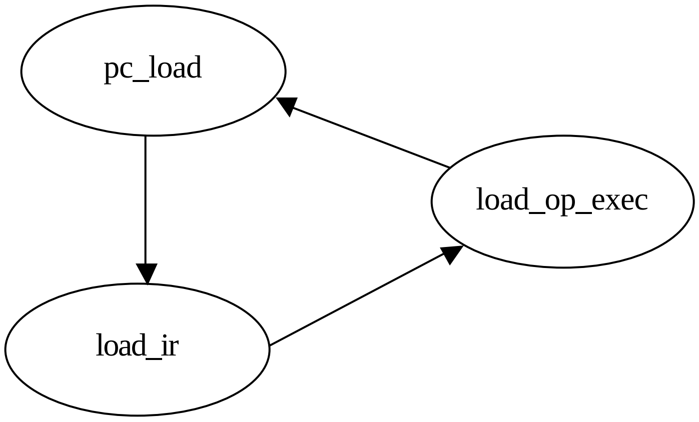
With instructions that do not require access to data memory, the execute cycle completes at this point in time and the next state entered will be pc_load. However, when a load from memory to the register file must be performed an additional stall cycle is necessary and thus the next state is load_stall. Also a store operation to data memory requires an additional stall, since the write result does not latch until at the end of the load_op_exec cycle and an extra cycle is necessary for the address bus to switch over to the program counter value (since data and instruction memory share their data and address bus). Therefore the next state after a store to memory is mem_stable.

During the load_stall cycle the general-purpose target register to which the loaded memory value must be stored is selected. Also the address bus is prepared again to load the next instruction by placing the program counter value on this bus.

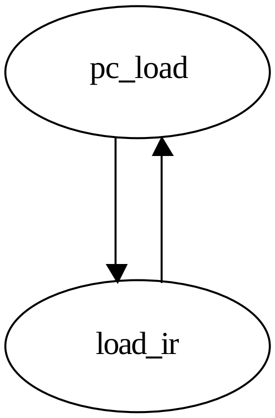
Another state, the stopped state, exists. This is the state entered when a stop instruction is executed or the stop line is high. The CPU remains in the stop cycle until reset is pressed. During the stop cycle the CPU is simply inactive and disables all memory and register enables.

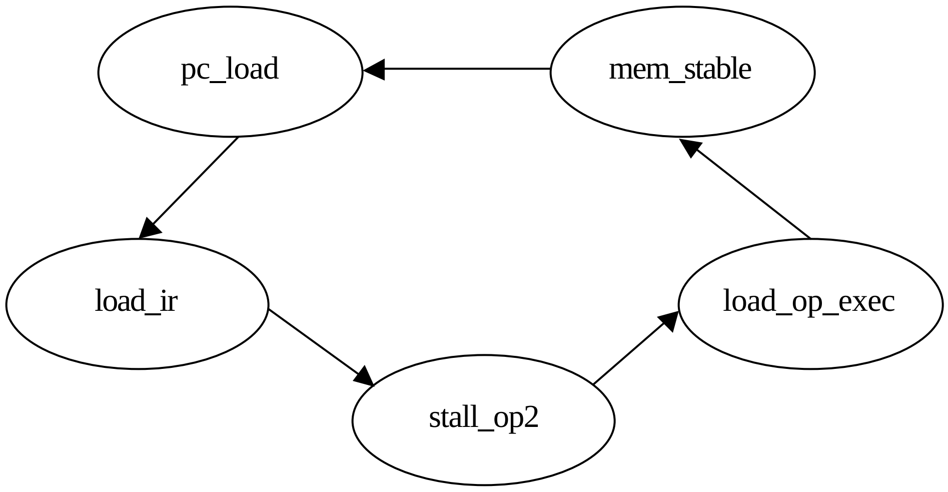
The following are a set of state diagrams for some typical instructions:

ALU, Shift, I/O and video instructions (always take three cycles):

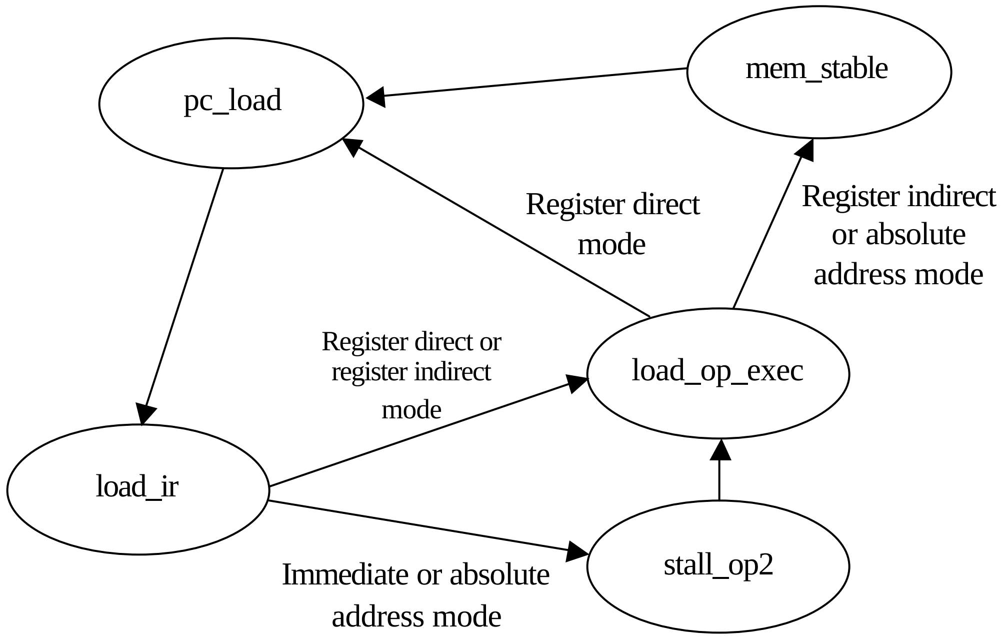


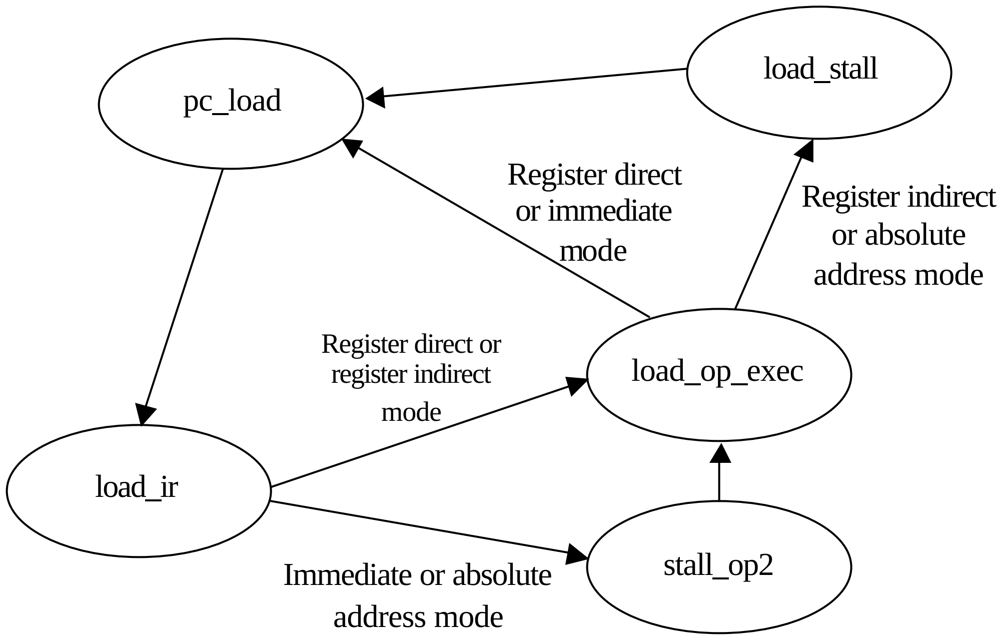
NOP instructions (always take two cycles) and branch Instructions (always take five cycles)





Load instructions (can take anywhere from three to five cycles)
All addressing modes refer to the source operand.
Write instructions (can take anywhere from three to five cycles)
All addressing modes refer to the destination operand.


mem_stable
pc_load
Register direct mode
Register indirect or absolute address mode
Register direct or register indirect mode
load_op_exec
load_ir
stall_op2
Immediate or absolute address mode


load_stall
pc_load
Register direct or immediate mode
Register indirect or absolute address mode
Register direct or register indirect mode
load_op_exec
load_ir
stall_op2
Immediate or absolute address mode

Description of GPU (Graphics co-Processor Unit)

In its current implementation (limited by the size of the FPGA and the amount of available internal block SRAM) the GPU supports 24 independent graphics objects simultaneously being displayed without any loss in performance in 8-bit colour and 640x480 resolution. These graphics objects come in two flavors: rectangles, which consist of a single colour, and sprites, which are rectangles where each individual pixel is mapped to a colour value in memory. Sprites can therefore take on any appearance the programmer wishes, including having transparent pixels to simulate different shapes. In addition to sprites and rectangles, the GPU supports a solid colour background to provide for a more appealing gameplay environment.

The GPU contains the following components:

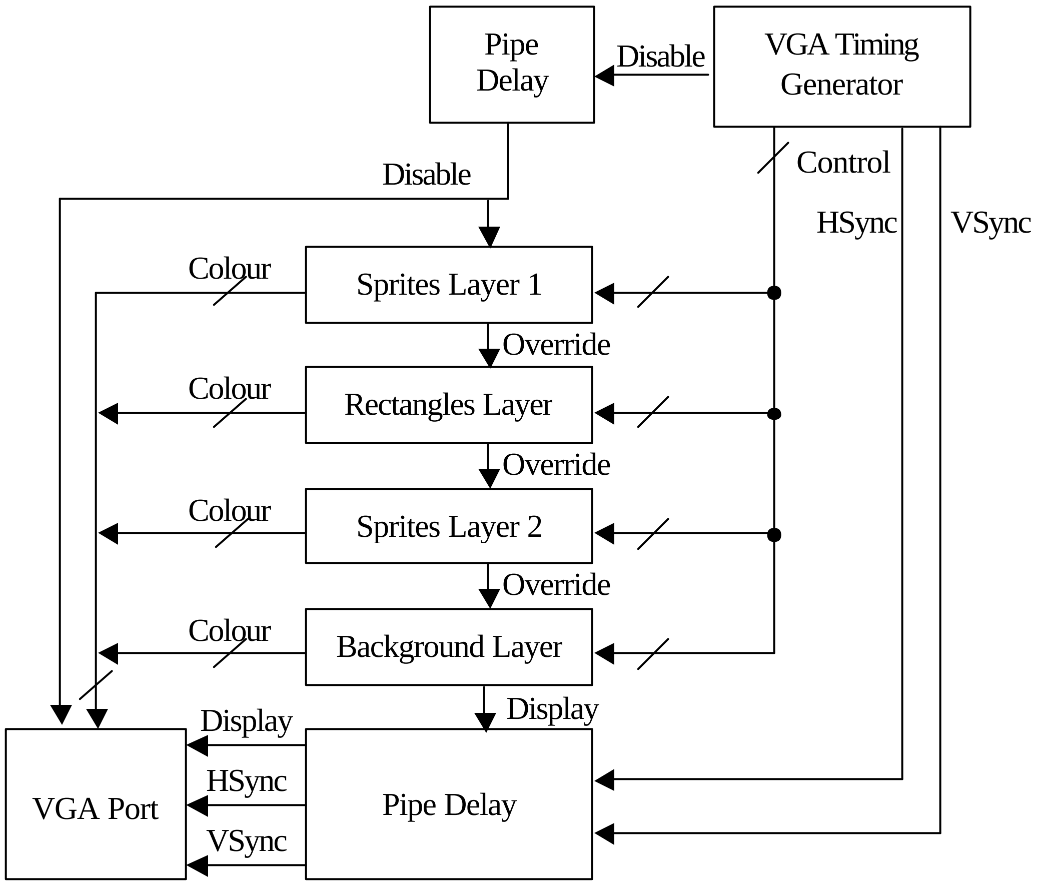
Component	**Description**
CPU Interface	Decodes commands from CPU
VGA Timing Generator	Generates control signals for other components. Generates HSYNC & VSYNC for VGA display.
Sprite Layer (x2)	Arbitrates Sprite Graphics
Rectangle Layer	Arbitrates Rectangle Objects
Background Layer	Renders a solid colour under other objects

Graphics objects are divided up into layers. Each layer is guaranteed to render properly underneath any layers above it, and above any layers below it. This includes any transparent sprite pixels taking on the colour of the pixel under them. The lower the ID of the layer, the higher it is. Graphics objects within the same layer, however, overlap imperfectly, with the lower ID object rendering on top of the higher ID object. This rendering does not respect transparency.

The various layers supported by the GPU are listed below:

Layer	**Index Range**	**Object Type**
0 (top)	0-7	Sprites 1
1	0-7	Rectangles
2	0-7	Sprites 2
3	0-0	Background Colour

Each graphics layer is responsible for deciding whether or not it is rendering a given VGA pixel, and if so, which colour it should have. Arbitration is used to allow the upper layers to override the lower layers, producing a single colour value at the output for the VGA port. Also, at any time the timing generator can prevent a particular pixel from being rendered, whether the requeset originated from the CPU, or whether it is motivated by preserving the integrity of the VGA signalling.



The GPU layers are fully pipelined so that a new pixel's colour is calculated every clock cycle. In an effort to reduce the huge amount of FPGA area required for the GPU, many of the more complicated graphics object control functions are abstracted to the timing generator. This reduces hardware and routing at the cost of complexity, since each graphics object requires control signals to operate. This does however severely increase the complexity of the timing generator, with the current implementation making use of a seventeen and a three bit state machine.

CPU Interface

All the properties of a graphics object (position, enable state, colour, sprite address) must be configurable by the programmer and hence the CPU. Since these properties are almost always in use, a copy of them is stored in FPGA block RAM. While the GPU is generating the pixel information for the current VGA frame, the CPU is free to modify these values using the GPU interface. Then, when the GPU is transitioning between VGA frames (there is a mandatory amount of time between frames in the VGA specification), the GPU serially updates each graphics object's local copy of this information. During this time, the CPU cannot update graphics object properties.

The following graphics commands can be issued from the CPU to the GPU:

Command (4 bits)	Mnemonic	Description
0x0	Row1	Sets object top value
0x1	Row2	Sets object bottom value
0x2	Col1	Sets object left value
0x3	Col2	Sets object right value
0x4	Enable	Sets object draw enable attribute
0x5	Reserved	
0x6	Colour	Sets object color value (for non-sprites)
0x7	StartAddr	Sets object start address (for sprites)
0x8	DispOn	Enables VGA output
0x9	DispOff	Disables VGA output (black screen)
0xA	Reserved	
0xB	Reserved	
0xC	Reserved	
0xD	Reserved	
0xE	Reserved	
0xF	Reserved	

The following signals connect the CPU to the GPU

Signal	Direction	Bits	Description
command	CPU -> GPU	4	Command issued to GPU
index	CPU -> GPU	6	Index of graphics object
cpuData	CPU -> GPU	16	Data bus
latch	CPU -> GPU	1	Load cpuData into registers
retrace	CPU <- GPU	1	Notifies CPU that vertical retrace is happening => CPU cannot update GPU registers.
red	CPU <- GPU	2	Red value to monitor
green	CPU <- GPU	3	Green value to monitor
blue	CPU <- GPU	3	Blue value to monitor
hsync	CPU <- GPU	1	Horizontal sync to monitor
vsync	CPU <- GPU	1	Vertical sync to monitor

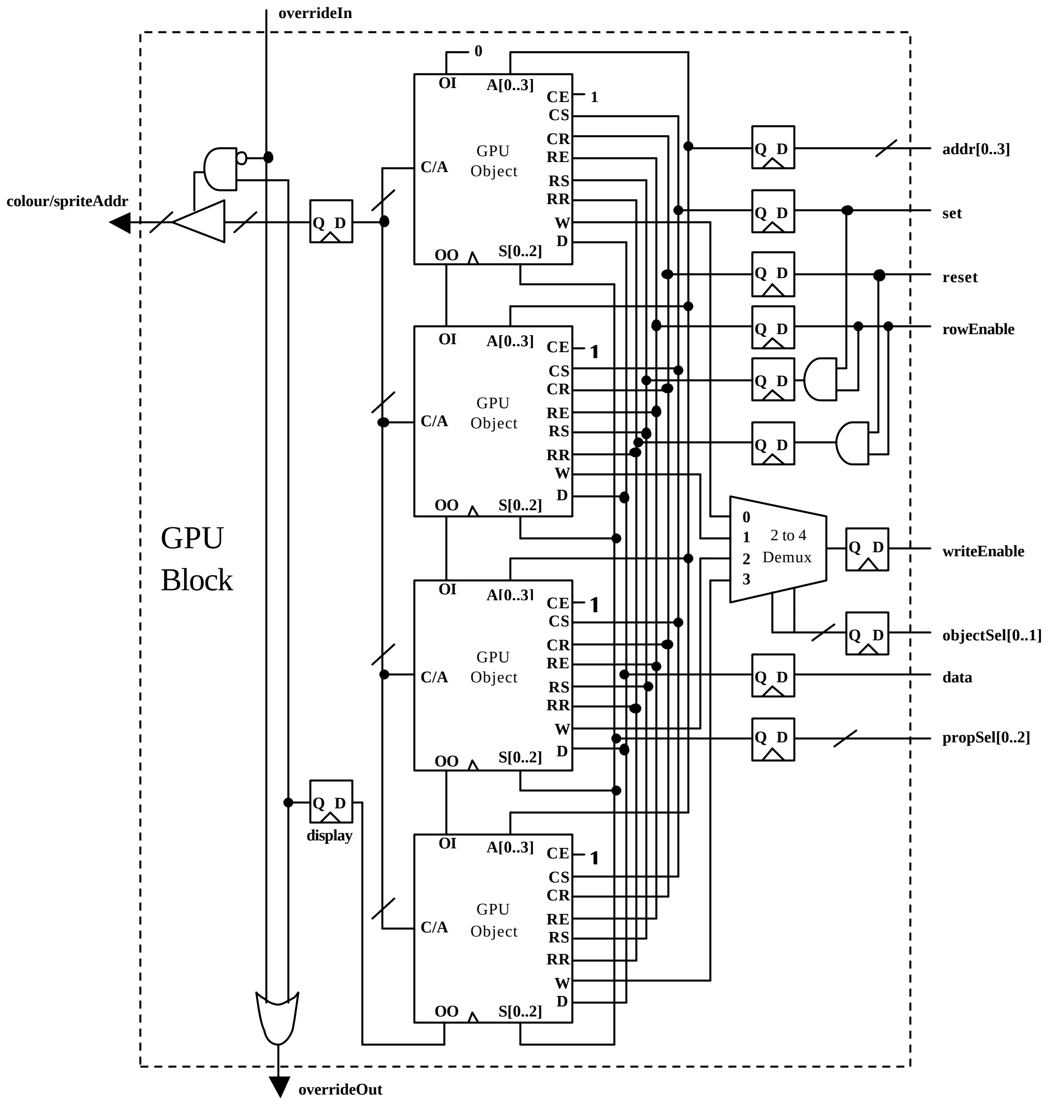
The CPU opcode will determine the command issued to the GPU. Most graphics instruction will take two register operands. The first operand (destination) will be the value placed on the 'index' lines. The 'index' signal specifies one of the graphics objects to update. The second operand (value) will contain the value to be sent to the 'cpuData' bus. The latch signal is used to synchronize writes to GPU registers.

The CPU is responsible for ensuring that it does not update any GPU registers while the GPU is in between displaying frames. The CPU will only be able to write to the GPU registers when the 'retrace' signal is de-asserted.

Pipeline Organization

Rectangle and sprite layers will be almost identical. Both will be divided into blocks of graphics objects from which they arbitrate a colour value for the current pixel, if applicable. Each block that attempts to render the current pixel will contend to place its colour or image address on the bus, with the lowest numbered block winning out. Sprites that place their addresses on the bus will be arbitrated into a single winning address which will be looked up in memory. If the resultant colour is transparent, the sprite layer does not render the pixel. Otherwise, the resultant colour is placed on the colour bus to be passed to the VGA port. In the case of rectangles, the winning colour value is simply placed on the colour bus.

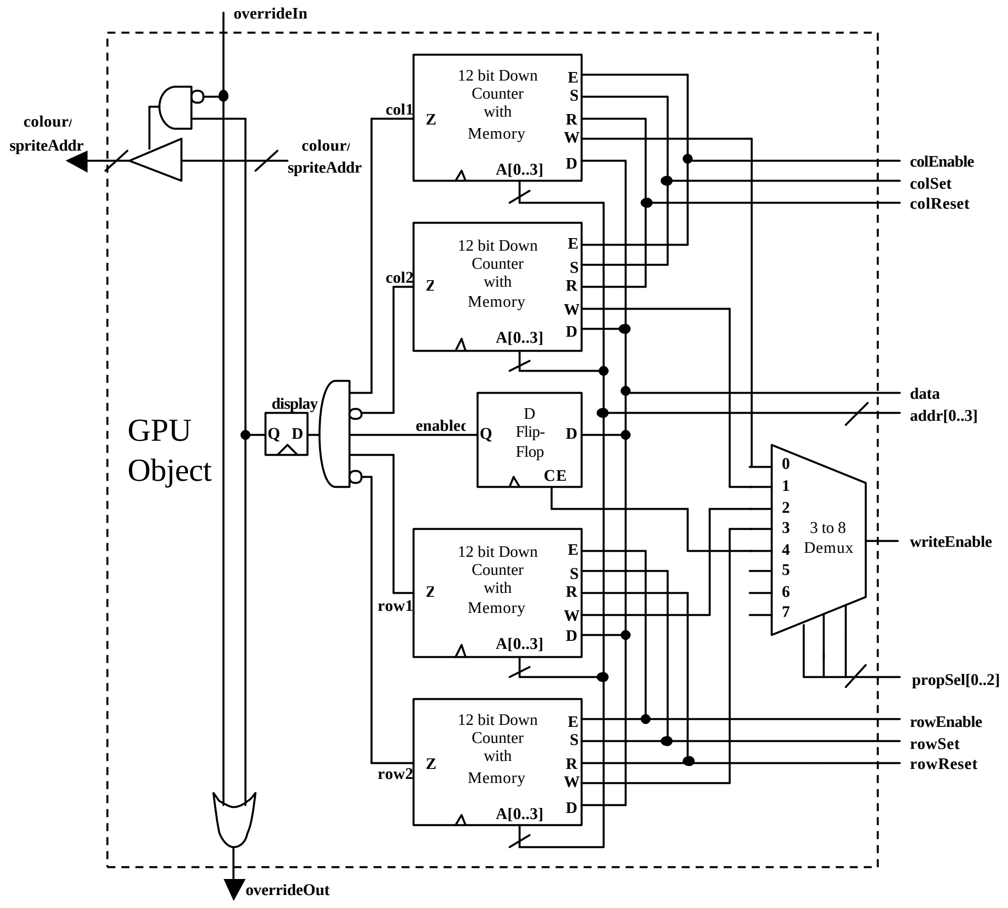
A depiction of a block of graphics objects is shown below:



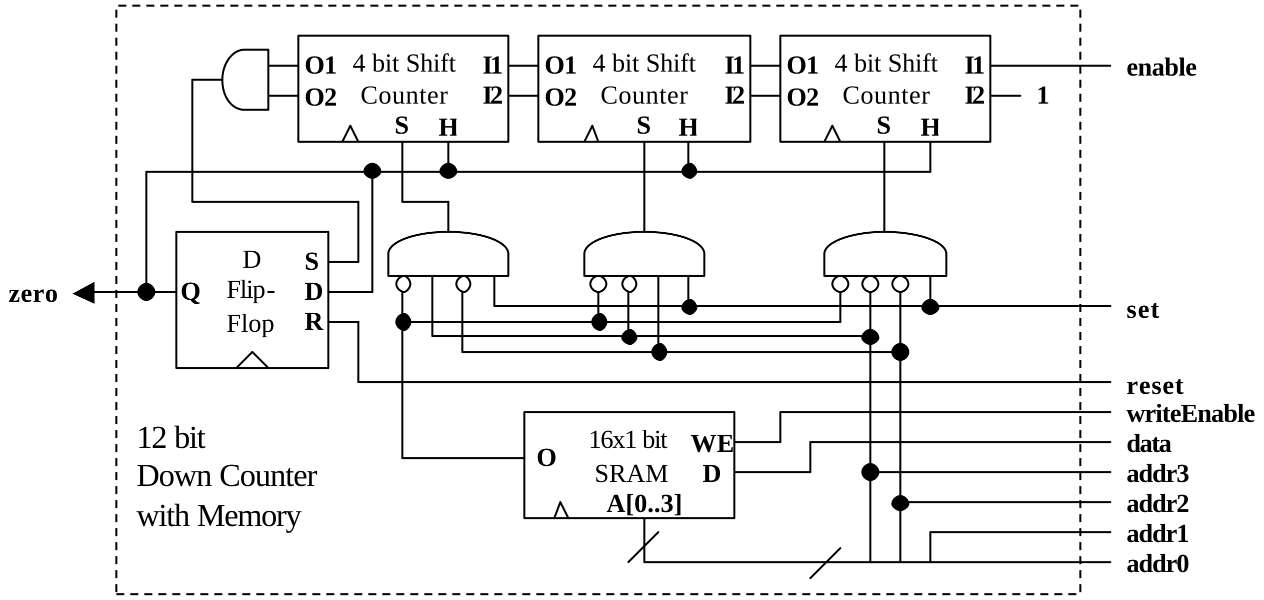
Each GPU block consists of four GPU objects that contend for which will draw the current pixel, if any of them. Control signals are also needed for the updating of object properties, since most of the logic has been abstracted to the timing generator.

Each GPU object consists of four 12-bit down counters which track that object’s row and column ranges for which is should display. It also has an “enabled” flag which needs to be set for the object to display itself. In addition, it has either a colour or a sprite address depending on whether it is a rectangle/background or a sprite. Control signals are used to reset the counters to store the positional information, as well as modify the enable state or colour/address of the object.

A depiction of a graphics object is shown below:

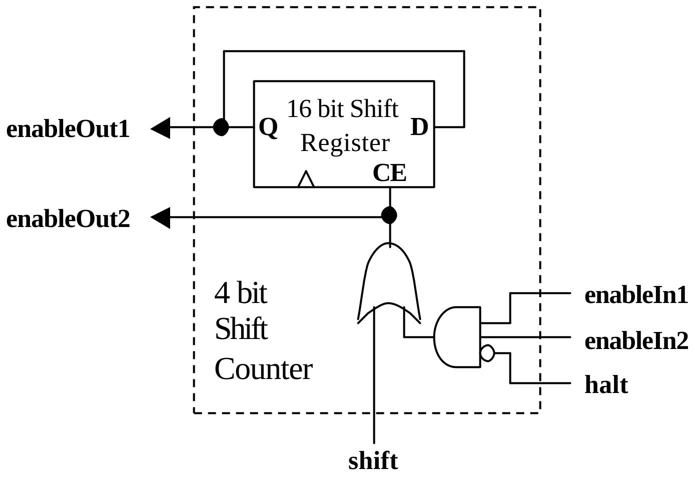


A depiction of a 12 bit down counter is shown below:



Each down counter is composed of three shift counters and a small SRAM to store the value it needs to count to.

A depiction of a 4 bit shift counter is shown below:



The motivation for this unconventional counter design is two-fold. First, Xilinx FPGAs allow a four-input look-up-table to be converted into either a 16 bit shift register or a 16 bit SRAM, providing far more logic than a single flip-flop if it can be exploited in a design. Second, with the large number of graphics objects in this design, and hence the large number of counters it makes use of, flip-flops would run out far faster than look-up-tables if the counters were flip-flop based. In addition, pipelining units like the divider require very large numbers of flip-flops while only moderate numbers of look-up-tables, so exchanging one for the other with this design proved to be advantageous in several ways.

Description of Memory

Initially, memory capacity was to come from external SRAM. This solution would have provided a much greater capacity than internal SRAM on the FPGA. There were difficulties with mounting the external SRAM chips on a PCB. This is discussed in more detail in the experiments and characterization section of this report. The end result was that internal block RAM was used as a substitute to the external RAM. The Xilinx Core Generator was used to configure the internal memories. This block memory on the Spartan 2 board is organized in 4096 bit blocks with a total of 14 blocks on our particular board. The block ram can be configured using Core Generator to various sizes and up to dual port reads and writes.

The following memories were created:

- 1536x16-bit CPU memory, logically divided into 1024x16-bit instruction memory and a 512x16-bit data memory.
- 256x16-bit GPU memory for storing graphic object information. This memory is writable by the CPU through a 16-bit port, and is readable by the GPU through a 1-bit port.
- 3072x8-bit GPU sprite memory for storing sprite data. This memory is readable through two 8-bit ports.

Six internal block RAMs were assigned to CPU program and data storage. Another six blocks were used to store sprites graphics. A single block RAM was allocated to the GPU to store graphic object properties. The CPU memory was configured with a single 16-bit read/write port. The CPU memory was configured with a single 16-bit read/write port. The sprite memory was configured with dual 8-bit read ports so that two layers of sprite data could be read simultaneously. The graphic object property memory constituted the CPU to GPU interface, and it was configured with a 16-bit write port from the CPU and a 1-bit read port for the GPU.

Description of I/O Bus

The IO bus is the link between the CPU and its peripherals. This bus is completely isolated from the main memory bus, and is accessed using special CPU instructions: PIN and POUT.

All directions specified in this description are with respect to the CPU (ie. if a data register is an 'out' register, it is for values coming out of the CPU to the peripherals).

The CPU is the bus master, and bus transactions only happen when requested by PIN and POUT instructions. Originally a bi-directional tri-state bus was used, but later the bus was converted to a synchronous bus with separate lines for each direction.

During an PIN operation, the CPU places an address on the bus, sets ioRw to 0, and set the ioEnable signal to 1. The peripheral places the requested data on the bus and at the next system clock edge the data is read and the ioEnable signal is returned to 0.

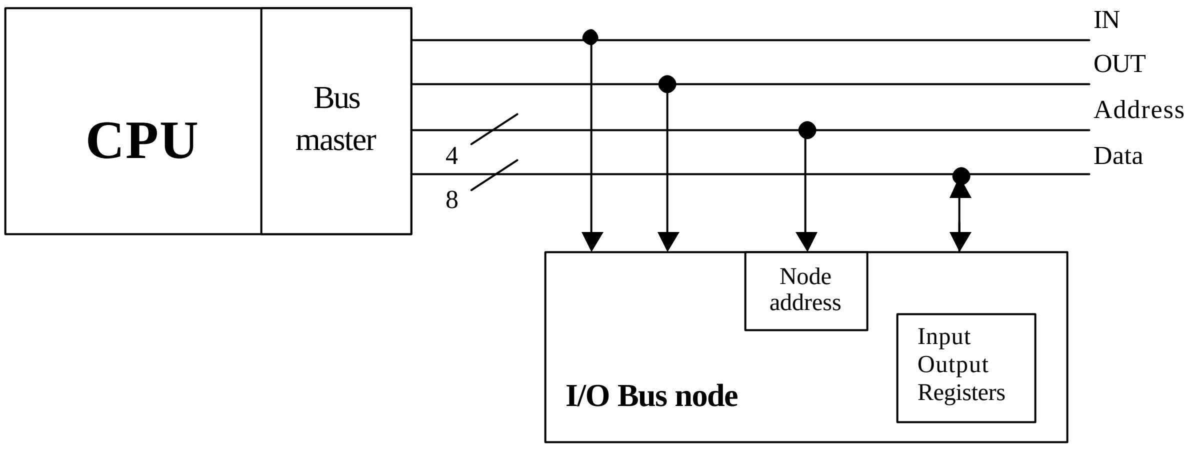
During an POUT operation, the CPU places an address on the address bus, places data on the data bus, set ioRw to 1, and sets ioEnable to 1. On the next clock edge the peripheral is written and the ioEnable signal is returned to 0.

The following table lists the most important signals of the IO Bus:

Signal	**Bits**	**Dir (CPU relative)**	**Description**
ioAddressBus	4	Out	Address of 16 different ports
ioDataIn	8	In	8-bit data into IO bus module from CPU going out to peripheral
ioDataOut	8	Out	8-bit data out of IO bus to CPU from peripheral.
ioRw	1	Out	0 for CPU to read from peripheral, 1 for CPU to write to peripheral
ioEnable	1	Out	When enabled, a read or write takes place on the next system clock edge.

The following table shows the IO Bus address assignments:

Address	Device
0	Read joystick 1 x-axis
1	Read joystick 1 y-axis
2	Read joystick 2 x-axis
3	Read joystick 2 y-axis
4	Read joystick buttons
5	unassigned
6	UART Data Register (write to TX, read to RX)
7	UART Status Register
8	Write LED lower byte
9	Write LED upper byte
10	Write 7-segment lower byte
11	Write 7-segment upper byte
12	unassigned
13	unassigned
14	unassigned
15	unassigned



Description of Joystick Controller

The joystick controller is configured to collect input from two standard analog PC joysticks. The joystick controller uses a 15 bit free-running counter clocked at 1MHz to collect joystick status information. The counter has a rollover frequency of 1MHz / 2^15 = 30.5 Hz. Each time the counter rolls over the 4 joystick buttons are latched and the 4 monostable multivibrators are triggered. The 30.5 Hz sampling frequency is slow enough to adequately debounce the joystick button inputs but is fast enough to keep with even the most avid game player.

A 15 bit register is used for each axis to latch the count value when it's multivibrator returns to its stable state, and this count value will be a function of the joystick axis resistance (resistance being proportional to position). The CPU can read the status of the buttons or any of the 4 axes from the IO bus. Calibration of the joystick position information will be left up to software running on the CPU.

Reading a joystick register through the IO bus interface returns an 8-bit value composed of the most significant 8-bits of the particular 15-bit joystick axis register. Reading the joystick button register across the IO bus returns the status of the four buttons in bits 0..3 of the 8-bit register. A zero indicates that a button is depressed.

Refer to the joystick circuit in the schematic diagram section of this report to see how the joysticks are interfaced to the joystick controller.

Description of LED Controller

The LED controller allows the CPU to control 16 individual LEDs and to display a data on 4 7-segment displays. The LED controller is connected to the IO bus and occupies four registers 8-bit registers: LED upper byte, LED lower byte, 7-seg upper byte, and 7-seg lower byte.

Description of UART

The UART is implemented as a full duplex single channel RS-232 UART without hardware flow control. The bit rate is fixed at 9600 BPS. The UART interfaces to the CPU via the IO Bus. The CPU can read and write the uart data register to read a received byte and to transmit a byte, respectively. The CPU can read the uart status register to see if the uart is ready to transmit and to see if a character has been received. A write of any value to the uart status register clears the RX flag bit.

The uart is useful to the system programmer because it can be used to output debugging information, and to allow data input that can't be done efficiently with the joystick. For example, a PC keyboard can be used for input to the gamecore system if connected to a PC's serial port with a communications program open.

The Digilent FPGA board has an on-board MAX chip for conversion between FPGA voltages and standard RS-232 voltage levels.

Gamecore Data Sheet

Features

- Integrated 16-bit CPU
    - 16-bit multiplier
    - Complete instruction set
    - Specialized video and IO instructions
    - Dedicated video and IO buses
    - Separate data and instruction memories (Harvard architecture)
- Integrated sprite and rectangle VGA graphics generator unit
    - Layered graphics with support for transparency
    - Hardware sprite and rectangle generators
    - Fully parallel graphic generation with immunity to flickering
- Internal SRAM for program code and data, graphic object properties, and bitmapped sprite graphics
- Support for two standard PC joysticks
- VGA monitor output
- RS-232 serial port available to user programs
- 16 individual LEDs and four 7-segment displays available to the programmer

IO Pins

Signal	Description	Direction	FPGA Pin	Connector Pin	Total I/O Pins
Colour(0)	VGA Blue Bit 0	Output	16	A40	
Colour(1)	VGA Blue Bit 1	Output	17	A39	
Colour(2)	VGA Blue Bit 2	Output	18	A38	
Colour(3)	VGA Green Bit 0	Output	20	A37	
Colour(4)	VGA Green Bit 1	Output	21	A36	
Colour(5)	VGA Green Bit 2	Output	22	A35	
Colour(6)	VGA Red Bit 0	Output	23	A34	
Colour(7)	VGA Red Bit 1	Output	24	A33	
Hsync	VGA Horizontal Sync	Output	27	A32	
Vsync	VGA Vertical Sync	Output	29	A31	
Clockin	Clock signal	Input	80	-	
Resetin	Reset signal	Input	77	-	
Dio2address(0)	DIO2 address interface	Output	59	A12	
Dio2address(1)	DIO2 address interface	Output	62	A9	
Dio2address(2)	DIO2 address interface	Output	61	A10	
Dio2address(3)	DIO2 address interface	Output	67	A7	
Dio2address(4)	DIO2 address interface	Output	63	A8	
Dio2address(5)	DIO2 address interface	Output	69	A5	
Dio2address(6)	DIO2 address interface	Output	68	A6	

Dio2data(0)	DIO2 data interface	Output	41	A23	
Dio2data(1)	DIO2 data interface	Output	37	A24	
Dio2data(2)	DIO2 data interface	Output	43	A21	
Dio2data(3)	DIO2 data interface	Output	42	A22	
Dio2data(4)	DIO2 data interface	Output	45	A19	
Dio2data(5)	DIO2 data interface	Output	44	A20	
Dio2data(6)	DIO2 data interface	Output	47	A17	
Dio2data(7)	DIO2 data interface	Output	46	A18	
Dio2clkout	DIO2 input clock	Output	60	A11	
Dio2cs0	DIO2 chip select	Output	49	A15	
Dio2oe	DIO2 output enable	Output	58	A13	
Dio2we	Dio2 write enable	Output	48	A16	
Joystickaxispins(0)	Joystick 1, x-axis	Input	174	C10	
Joystickaxispins(1)	Joystick 1, y-axis	Input	173	C11	
Joystickaxispins(2)	Joystick 2, x-axis	Input	172	C12	
Joystickaxispins(3)	Joystick 2, y-axis	Input	168	C13	
Joystickbuttonpins(0)	Joystick 1, button 1	Input	167	C14	
Joystickbuttonpins(1)	Joystick 1, button 2	Input	166	C15	
Joystickbuttonpins(2)	Joystick 2, button 1	Input	165	C16	
Joystickbuttonpins(3)	Joystick 2, button 2	Input	164	C17	
Joysticktriggerpin	Joystick trigger	Input	163	C18	
				Total:	40

Maximum Speed

The maximum speed reported by the Xilinx tools was 53MHz. This equates to a minimum clock period of about 19 ns.

Resource Usage Measurements

The resource usage for the completed design as reported by the Xilinx tools was as follows:

Resource	Usage	Total	Percent Usage
4-input LUTS	3141	4707	67%
Flip-flops	1256	4707	27%
Slices	2350	2352	100%
Tri-state Buffers	864	2464	35%
IO Pins	40	140	29%
Global Clock Buffers	3	4	75%

The following data was collected by compiling various sub-components independently. Please note that the numbers are not indicative of the final design.

GPU:
- 1446 slices
- 1785 LUTs
- 645 flip flops
- 352 tri-state buffers

ALU:
- 226 slices
- 448 4-input LUTs
- 0 flip flops

UART:
- 64 slices
- 84 4-input LUTs
- 71 flip flops

VGA Timing Generator:
- 43 slices
- 67 4-input LUTs
- 26 flip flops

Joystick Controller:
- 81 slices
- 23 4-input LUTs
- 79 flip flops

GPU Block
- 160 slices
- 207 4-input LUTs
- 31 flip flops

Results of Experiments and Characterization

Throughout the design process, many different things were tried. Some made it into the final design, while many experiments failed. Below is a summary of experiments conducted.

Graphics Processing Unit

Initially, the idea for the GPU was to keep everything as simple as possible. Each graphics object would monitor row and column lines, comparing their values against internal ones in order to decide whether or not they should render the current pixel. While the straight-forward nature of this approach was appealing, a few quick calculations showed that this design would require an exorbitant number of flip-flops in order to store all these comparison values. While the current design can only practically support 24 graphics objects at once, it was estimated that the former design would support only 12.

While an inefficient design in any normal situation would lead to a small loss in FPGA resources and is normally not worth the effort to correct, the special situation of the GPU magnifies the problem. With 24 graphics objects, there are a total of 96 down-counters which means a total of 288 shift counters. It is easy to see that even a single extra look-up-table or flip-flop in any of these sub-component designs quickly grows to a much more significant waste when that design element is replicated.

To that end, a good deal of thought and creativity was applied to the problem of designing an efficient graphics object property storage element that minimized the number of flip-flops used. The result was a 12-bit down-counter with its own memory that consists of 11 look-up tables and one flip-flop. The cost of this flip-flop savings is an increased complexity in the control logic used to control it, and the reliance on serial transmission of data instead of parallel. The later does however cause the added bonus of significant savings on routing resources as fewer signals need to read each object. Overall, this design was chosen so as to maximize the number of graphics objects that could fit in the FPGA while minimizing the number of flip-flops required for each.

External SRAM

External SRAM was a fundamental part of the initial Gamecore design. It was recognized that large amounts of memory would be desired for user program and data storage, and more importantly for storing large sprite graphics. The initial design included 256 KB of fast 10ns SRAM divided into two independent 16-bit wide memories of 128KB each. One memory was to be used for program and data storage, and the second was to be used for sprite video memory.

SRAM chips were selected and ordered, and PC boards were manufactured through Alberta PC Board. Each board could handle two 44-pin J-Lead surface mount SRAM chips from Cypress Semiconductors. Each board interfaced to the Digilent FPGA board through a 40-pin header. The SRAM chips were very difficult to solder onto the boards because of their small pin pitch and the fact that the pads were small and could not be reached by the soldering iron because the leads of a J-Lead package curl under the chip.

During testing of the RAM, it was discovered that one board had a short between power and ground that could not easily be repaired. The second board worked better, but only about half of it's data pins functioned correctly. While a large supply of extra boards, chips, and headers was purchased, time did not permit more boards to be assembled, and redesign with internal FPGA block RAM's was necessary. This action limiting the available RAM to 7KB, and somewhat crippled the Gamecore design since 256 KB was originally planned for.

Internal Block RAM

The Xilinx Core Generator was a useful tool in configuring internal block RAM. Once the SRAM size, data width, and read/write capabilities was determined, generating a chuck of SRAM could be performed in less than a minute. An option to initialize the memory was also provided. The function does experience a limitation, however. The initialization data that is provide in the *.coe file must be the same size as the data width of port A. Otherwise, the memory will either fail to generate or the initialization file cannot be read.

The process of configuring block memory create quite a few files in the project directory. As part of good housekeeping, moving these files to a subdirectory was tried, but the block memory failed to regenerate in the project. Configuring block memory is separate projects was also tried, but also failed when added to the top level project. Disadvantage of using block memory is therefore reduced capacity when compared to most externally connected SRAM and a large project directory that can easily wreak havoc on organization.

Gray Code Counters

Gray code was investigated with the goal of reducing signal lines between a few devices. Since gray code changes only one bit for each consecutive count, perhaps a single bit could be transmitted as opposed to many bits for a count. The savings in bits could be quite significant with a large count. Methods to convert between Gray code and binary were found and proved to execute correctly. A method to determine the bit that changed for a count could not be found. There did not seem to be a deterministic pattern to the way the gray code bits changed. Efforts were abandoned in this area as it proved too difficult.

Joystick Characterization and Calibration

The joystick circuit was originally tested with a function generator and an oscilloscope before connecting to the FPGA. The function generator was set to supply a 30.5 Hz square wave to the trigger inputs of the monostable multivibrators, and the output pulse width was measured with respect to the joystick position. The 30.5 Hz trigger frequency mimicked the joystick controller which uses the same triggering and sampling frequency.

The capacitor values were adjusted until a good range of output pulse widths was obtained. A duty-cycle swing of about 0% to 80% was set. Once the circuit was thus calibrated, measurements were taken which showed that the pulse width was indeed directly proportional to the joystick position.

Some non-ideal behaviours of the joystick circuit were discovered. One of these non-ideal behaviours was the jittering of the measured values. This jittering can at least in part be due to the mechanical nature of the joystick potentiometer. Another non-ideal behaviour was that the resistance of the joystick dropped off to zero for the last 10% of motion. Since precise analog control of the joystick position is not needed, the joystick input can be cleaned up in software by quantizing to a small set of possible input values.

Externally provided HDL components

Xilinx Cores were used to implement the multiplier, divider, and the block RAMs. A 16-bit combinational multiplier has been added to the CPU, and it has been verified to work properly at 25MHz (our system clock frequency) by having the CPU multiply a series of numbers an then display the results on the seven segment displays. The RAMs were tested independently in hardware, and also during the test of the multiplier since, in the process, the CPU was reading instructions out of RAM.

These components have been verified, and it is easily concluded that these components are suitable for the design. Xilinx Cores were used partly to save time, and partly to meet the requirement of using externally provided HDL components.

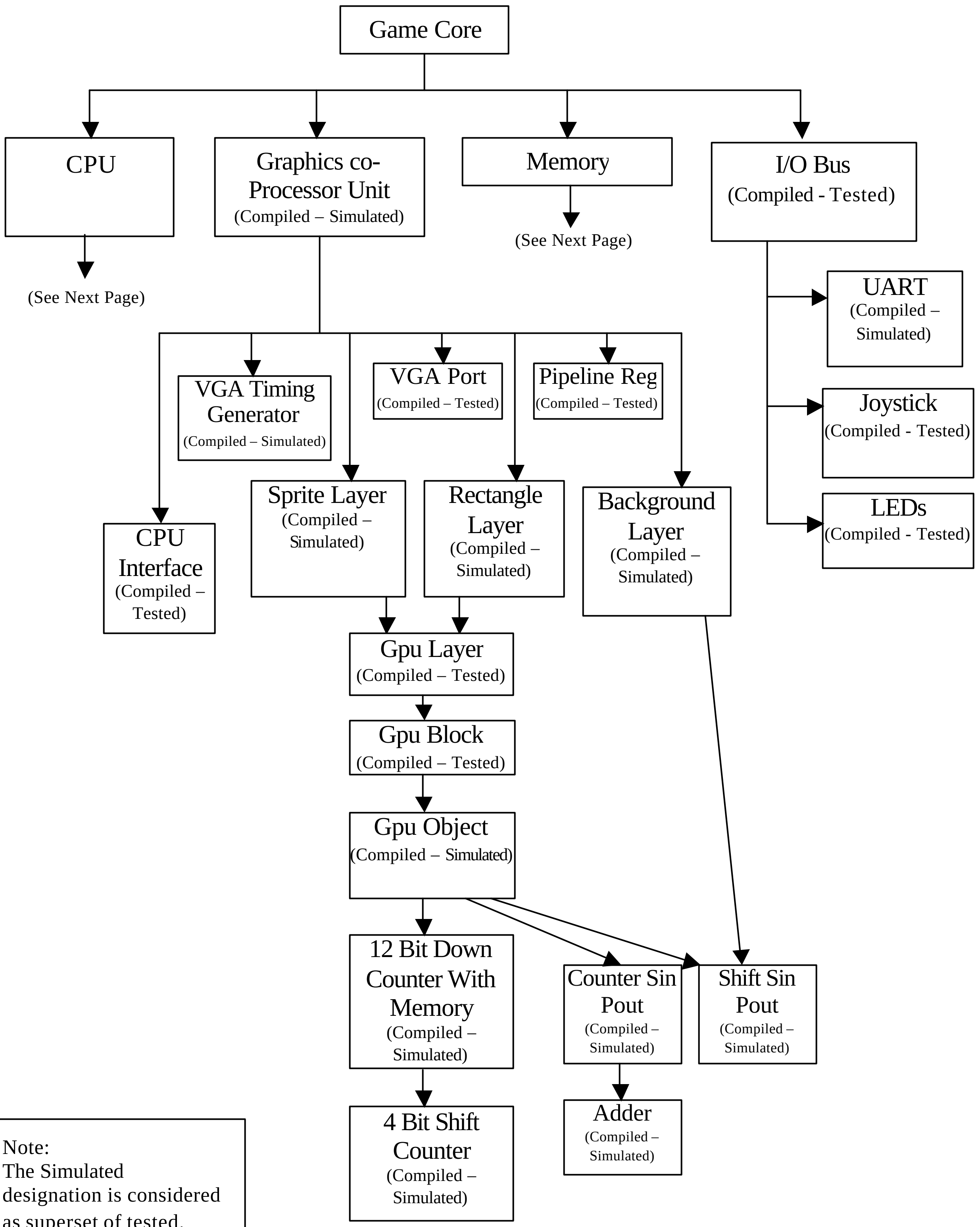
References

- [1] OSU8 Microprocessor, Paul Stoffregen, August 31, 1999
  www.pjrc.com/tech/osu8/inst_set.html
- [2] Cypress Semiconductor Corporation, 1995-2003, www.cypress.com
- [3] A Simplified VHDL UART, University of California Riverside Computer Science, June 2001, www.cs.ucr.edu/content/esd/labs/uart/uart.html
- [4] ePanorama – Joystick documents, Tomi Engdal, 1996-1998
  www.epanorama.net/documents/joystick/index.html
- [5] MIPS32 Instruction Set, MIPS Technologies Inc., March 12, 2001
  http://segfault.net/~scut/cpu/mips/MIPS32_Vol2_The_MIPS32_Instruction_Set_v0.95.pdf
- [6] The Designer's Guide to VHDL, Peter J. Ashenden, 2nd Edition, 2002.
- [7] ALSE (Advance Logic Synthesis for Electronics), 2002, www.alse-fr.com
- [8]Joystick Circuit based on circuit from Epanorama.net
  Accessed http://www.epanorama.net/documents/joystick/pc_joystick.html. February 18, 2003
- [9]VGA timing calculator spreadsheet from VESA
  Accessed http://www.vesa.org/public/SMT/SMT640_720x480v1.xls. February 18, 2003
- [10]VGA timing signal information from EE552 appnote
  Accessed
  http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/2002_w/interfacing/CRT/CRT_App_Note.html. February 17, 2003
- [11] Xilinx Spartan II Primitives instantiation templates.
  Accessed http://toolbox.xilinx.com/docsan/xilinx5/data/docs/lib/lib0024_8.html
  March 1, 2003.

Data Sheets

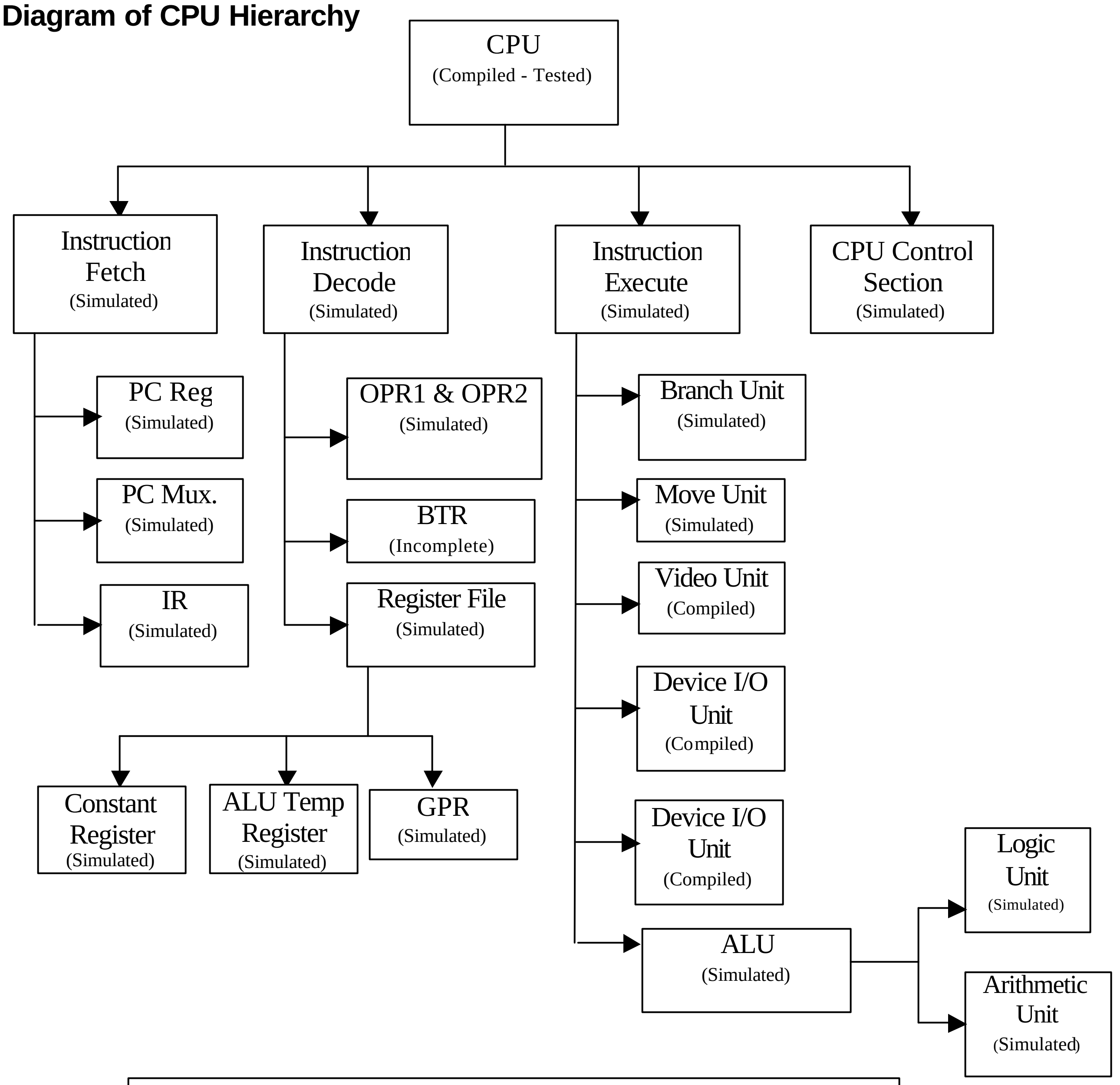
The only external integrated circuit used was a 74LS123 retrigerable monostable multivibrator. This chip was used in the joystick circuit to convert the joystick position (resistance) into a pulse width.

Diagram of Design Hierarchy


Game Core
CPU
(See Next Page)
Graphics co-Processor Unit (Compiled – Simulated)
Memory
(See Next Page)
I/O Bus (Compiled - Tested)
UART (Compiled – Simulated)
Joystick (Compiled - Tested)
LEDs (Compiled - Tested)
VGA Timing Generator (Compiled – Simulated)
VGA Port (Compiled – Tested)
Pipeline Reg (Compiled – Tested)
CPU Interface (Compiled – Tested)
Sprite Layer (Compiled – Simulated)
Rectangle Layer (Compiled – Simulated)
Background Layer (Compiled – Simulated)
Gpu Layer (Compiled – Tested)
Gpu Block (Compiled – Tested)
Gpu Object (Compiled – Simulated)
12 Bit Down Counter With Memory (Compiled – Simulated)
Counter Sin Pout (Compiled – Simulated)
Shift Sin Pout (Compiled – Simulated)
4 Bit Shift Counter (Compiled – Simulated)
Adder (Compiled – Simulated)

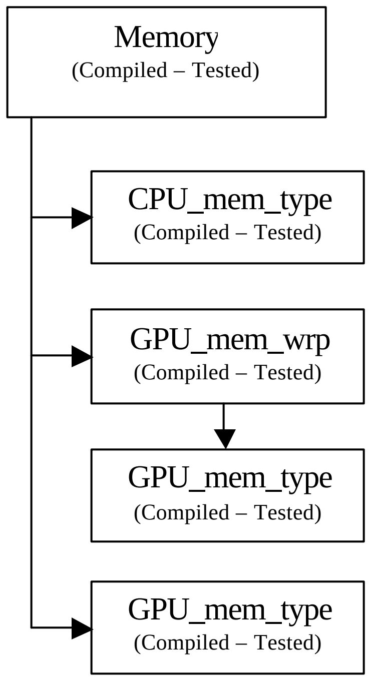
Note:
The Simulated designation is considered as superset of tested.

Diagram of CPU Hierarchy



Legend
PC = Program Counter
IR = Instruction Register
OPR1 & OPR2 = Operand Register 1 and Operand Register 2
BTR = Branch Target Register
ALU = Arithmetic-Logic Unit
Temp Reg = Temporary register where an operand from the I/O bus or memory is stored before being written to the register file
GPR = General Purpose Register
Constant Register = all zeros or all ones register
ALU Temp Register = remainder from division operation or upper 16 bits from multiplication storage

Diagram of Memory Hierarchy



Index to VHDL packages and code

PACKAGE / COMPONENT and DESCRIPTION	STATUS
Package pigme_types Package containing common data types and constants	Compiled – No errors (types and constants package)
Component pipeline_reg Pipeline registers	Compiled – Tested
Component Gamecore Gamecore project top-level	Compiled – Tested
Package memory_pkg Package of block ram components	Compiled – No errors
Component Adder N-bit adder	Compiled – Tested
Package dio2_pkg Package with body to connect to IO Board LEDs	Compiled – No errors
Package io_bus_pkg Package of all IO bus components	Compiled – No errors
Component io_bus IO bus components	Compiled – Tested
Package joystick_pkg Package and body of joystick controller	Compiled – No errors
Package uart_pkg Package and body of serial port	Compiled – No errors
Package barrel_shift_pkg Barrel Shifter package	Compiled – No errors
Component barrel_shift Barrel Shifter	Compiled – Tested
Package branch_unit_pkg Branch unit package	Compiled – No errors
Component branch_unit Branch unit	Compiled – Tested
Package cpu_controlsection_pkg CPU control path package	Compiled – No Errors
Component cpu_controlsection CPU control path definition	Compiled – Tested
Package cpu_pkg CPU architecture interface	Compiled – No errors
Component cpu CPU architecture	Compiled – Tested
Package decoder4_16_pkg Four to 16 bit address decoder, recognized and optimized by the Xilinx tools	Compiled – No Errors
Package inst_decode_pkg CPU instruction decoder package	Compiled – No errors
Component inst_decode CPU instruction decoder	Compiled – Tested
Package inst_exec_pkg CPU instruction execution package	Compiled – No errors

Component inst_exec CPU instruction execution	Compiled – Tested
Package inst_fetch_pkg CPU instruction fetch package	Compiled – No errors
Component inst_fetch CPU instruction fetch	Compiled – Tested
Package io_unit_pkg Handler of reads and writes to I/O bus	Compiled – No errors
Component io_unit Handler of reads and writes to I/O bus	Compiled – Tested
Package load_store_pkg CPU reads/writes to memory package	Compiled – No errors
Component load_store CPU reads/writes to memory	Compiled – Tested
Package register_file_pkg CPU Register File package	Compiled – No errors
Component register_file CPU Register File	Compiled – Tested
Package register_pkg Various CPU register types	Compiled – No errors
Component register Various CPU register types	Compiled – Tested
Package shift_unit_pkg Shift unit package	Compiled – No errors
Component shift_unit Shift unit package	Compiled – Tested
Package alu_pkg ALU for PIGME CPU	Compiled – No errors
Component back_layer Background generator	Compiled – Tested
Component counter_sin_pout N-bit counter with serial input and parallel output	Compiled – Tested
Component down_counter 12 bit down counter	Compiled – Tested
Component gpu_block Block of GPU objects	Compiled – Tested
Component gpu_layer Layer of GPU blocks	Compiled – Tested
Component gpu_object GPU Graphics Object	Compiled – Tested
Component gpu GPU top level	Compiled – Tested
Component joystick Joystick interface	Compiled – Tested
Component rect_layer Layer of Rectangle Generators	Compiled – Tested
Component shift_counter 4-bit shift counter	Compiled – Tested
Component sprite_layer Layer of sprite generators	Compiled – Tested

Component vga_port VGA port	Compiled – Tested
Component vgaTimeGen Provides VGA timing signals.	Compiled – Tested

VHDL Design

VHDL Code not available in the electronic version of this report.

Test Bench Documentation

Test Bench / Simulation index
Pipeline_reg_tb Test pipeline for correct delay
Io_bus_tb Test register file registers and verifies correctness
Barrel_shift_tb Verifies correct rotation of bits
Cpu_controlsection_tb Test the proper operation of the cpu control section in conjunction with the instruction fetch and decode stages as well as the ALU and move unit.
Cpu_tb Test the top level CPU interface by reading in an assembly program and verifying that the CPU writes the correct data back to the register file and data memory.
Inst_decode_tb Test the proper loading of the operand 1 and operand 2 registers from either the register file or memory. Also the loading of the branch target register is tested.
Inst_fetch_tb Test the PC, PC incrementer, and instruction register (IR)
Register_file_tb Test the register file reads and writes
Shift_unit_tb Test the various types of shift operations in both directions.
Down_counter_tb 12-bit down counter test bench
Gpu_object_tb Test a gpu object’s ability to draw itself
Shift_counter_tb Shift counter test bench
Vga_time_gen_tb Test VGA timing generation of signals

For the register file (register_file_tb.vhd) the following scenarios were tested:
Each register’s index value (or register address) will be written to that register, than some small amount of time later it is verified that the register does indeed contain the correct value. This value is output on both read ports of the register value. Of course, the constant registers cannot be written to and will always output all ones or all zeros. A special signal must be asserted for the temp register, since this temp register will be written to in parallel with another 16-bit ALU output.
Then to verify that two different registers can each write to one of the read ports, another loop is executed in which the register at index and index+1 each output their index value on one of the read

ports. Finally the functionality of the temp register is verified by writing to the temp register at the same time as one of the other general-purpose registers is written to.
The worst delay in the register counter is 12.166 ns between the register select value change and the constant register 0. To reduce the delay it is possible to exchange the tri-state output but for a pair of 256 to 16 multiplexers for the read ports, however this would increase the area used tremendously and likely yields only small timing gains, if any.

For the instruction fetch unit another test bench is written (inst_fetch_tb.vhd).
In this test bench a limited number of CPU fetch cycles are simulated. Initially, a reset signal is sent and it is verified that the PC and IR reset properly.
After the reset, a few CPU fetch cycles are simulated in which the PC is incremented and sent down the address bus to the instruction memory to fetch an instruction. After the address has been put on the memory address bus, a value is put on the data bus and this is latched into the instruction register. It is then verified that the value put on the data bus matches the value in the instruction register.
Finally, it is verified that when a branch succeeds, the branch target register value is loaded into the PC instead of an increment of the previous address.
Inside the instruction fetch unit, the critical path lies between the PC output and the PC input through the adder and multiplexer. However, once the unit is connected to the memory controller, the time between putting the PC value on the address bus and waiting for the result will likely be the path of longest delay as the instruction fetch unit must wait for the memory controller data bus signal to stabilize.

The instruction decode unit contains the register file. A test bench (inst_decode_tb) somewhat similar to the register file test bench was created, in that the operand registers are each loaded from each register from the file register. Also, loading from the instruction bus into the operand register is tested. It is also verified that the branch target register properly loads when it is enabled.
The worst delay in the instruction decode is 16.696 ns; the critical path lies between the register select value change and the corresponding change in the register read port. There is little one can do to speed this up, since most of the delay is due to the register file. The change from a tri-state bus to the multiplexer based approach to speed up the register file read cycles can be used to increase the speed of this part of the circuit. However, since our clock period is 40ns, there is no real need to speed up this circuit, and area constraints are a bigger concern than speed concerns.

Because simulating the control unit separately takes more effort than simulating it as part of the CPU, it was decided to connect the instruction decode, instruction fetch, ALU and move unit all in one test bench. This has the added bonus that all the signals between the units are clearly visible and can be inspected. A sample instruction stream is provided to verify that the control section advances from state to state properly. Meanwhile, the functionality of the CPU as a whole is also tested. The interaction between the different components of the CPU is also confirmed when testing in this manner. The instructions supplied to the CPU can be found in the test bench file cpu_controlsection_tb.vhd.

To verify that the barrel shifter operated correctly, an input bit vector was generated and shifted from 0 through 15 bits to the left in a test bench (barrel_shift_tb.vhd). The barrel shifter's output is compared with the VHDL ROL operator some time after new inputs have been applied. The critical path in the barrel shifter is from a change in the most significant bit in the bit vector specifying the number of bits to be shifted to the least significant bit in the shifter's output. The critical path delay is 17.063 ns, but cannot be improved upon. The generic barrel shifter was compared to a design that the Xilinx tools recognize and optimize, and their logic and delay characteristics are identical.

To verify that the shift unit operates correctly, all possible shift and rotate instructions are tested in both directions. All the different shift instructions are put through a test in which their inputs are shifted from 0 to 15 bits. The output for each shift and rotate instruction is then compared to the corresponding VHDL operator's output for the same input data (shift_unit_tb.vhd).

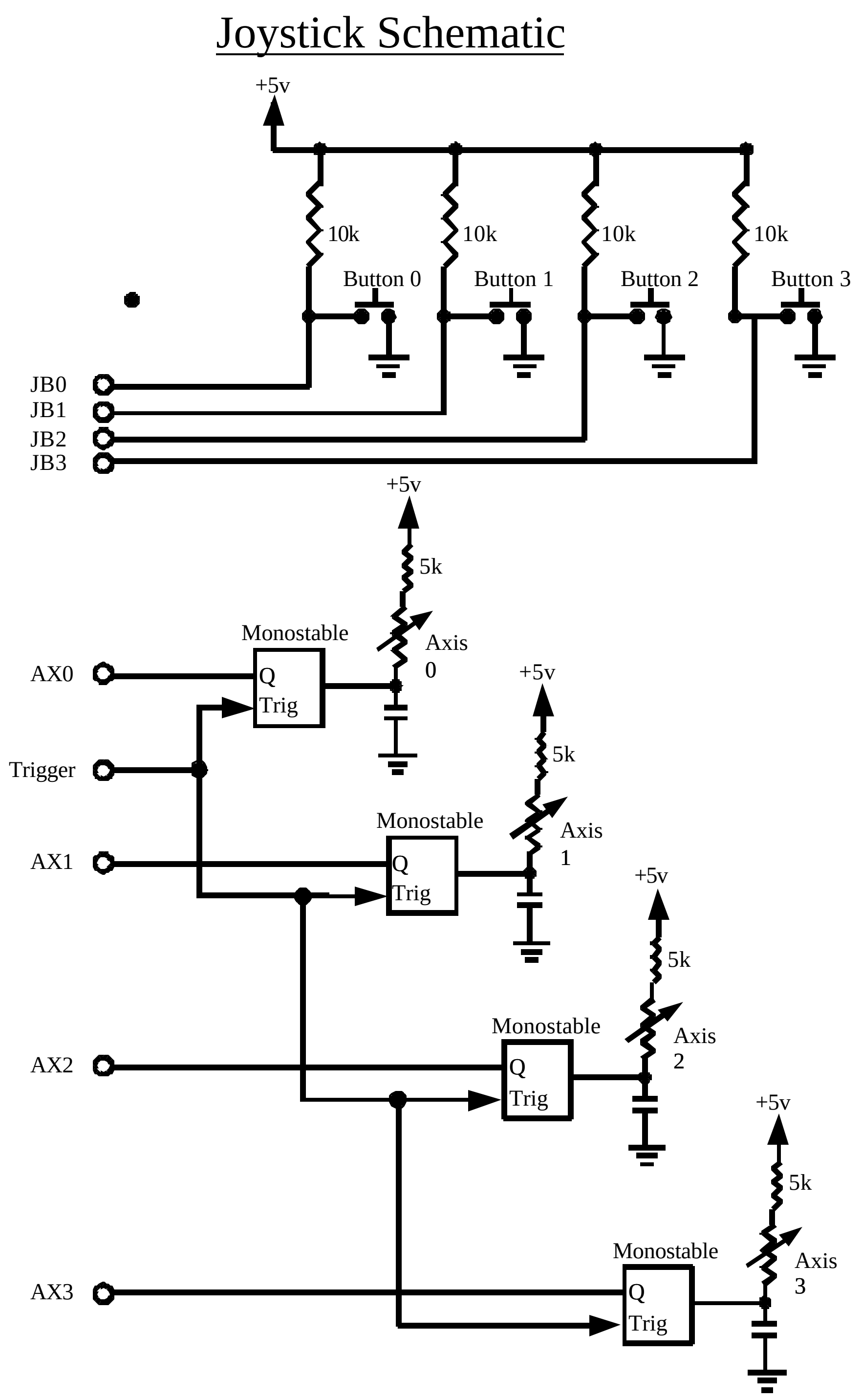
The most important part of the simulation is the verification of the top level CPU entity. In order to facilitate proper testing of the CPU, the following signals are passed through the top-level entity for testing purposes only: temp register data and enable, general purpose register select and data and the current instruction register value. The test bench starts out by reading in the ASCII instruction and data memory files. It parses the instruction streams and stores them in internal arrays for instruction and data memory. After completing reading in the program, it asserts the reset signal for a few clock cycles, and then commences simulating the program. To verify that the CPU is operating correctly, the test bench simulates the CPU instructions itself. The test bench contains a simulated register file, as well as the simulated instruction and data memory values read in at the start of the simulation. As the CPU executes instructions and writes them to the register file and data memory, the test bench monitors the outputs, and catches any errors in the address and data busses to the register file and data memory. This approach to testing the CPU allows for thorough verification of the CPU's operation. An added advantage is that the programmer can debug their program in hardware simulation and determine whether the problem they are debugging is due to faulty hardware or a fault in their software (this becomes problematic, however, for programs that execute for more than a few hundred cycles however). There are two sets of waveforms included for this test bench, one simulating the video instructions and one simulating a variety of instructions.

For the GPU object testbench (gpu_object_tb.vhd), the main testing strategy consisted of loading a set of properties into the object using the timing generator's protocol, then observing whether or not it behaved correctly. Namely, the object should not display itself when it is not supposed to, it should always display itself when it is supposed to, and it should always display itself correctly. By positioning the object in the center of a small rectangular simulated screen, the object's ability to identify when it should render itself and in what colour was verified.

For the VGA timing generator testbench, (vga_time_gen_tb.vhd), the main strategy was simulate memory values for the generator's serial read requests of bits, then observe that all the correct timing and control signals were generated. While this approach is very thorough, it does take an extremely large amount of time to simulate a frame of video elapsing. However, after much waiting, it was observed that not only did the timing generator produce the appropriate sync and disable signals for VGA, it also effectively made use of dead-times in between retraces in order to update the memory values of every graphics object.

Schematics

Joystick Schematic



Instruction Set

ADD

Signed Addition

Operation: **`rd ¬ rs1 + rs2`**

Syntax: **`ADD rd, rs1, rs2`**

Description: Adds **`rs2`** to **`rs1`** and stores the result in **`rd`**.

Notes: Overflow and Borrow are ignored.

15 12	11 8	7 4	3 0
1000	**rd**	**rs1**	**rs2**

AND

Bitwise AND

Operation: **`rd ¬ rs1 & rs2`**

Syntax: **`AND rd, rs1, rs2`**

Description: Stores the bitwise AND of **`rs1`** and **`rs2`** in **`rd`**.

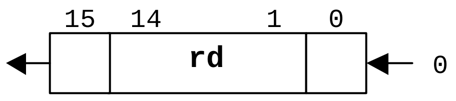
Notes: none

15 12	11 8	7 4	3 0
1100	**rd**	**rs1**	**rs2**

ASL

Arithmetic Shift Left

Operation: **`rd ¬ rd << rs`**



Syntax: **`ASL rd, rs`**

Description: Bit shifts **`rd`** to the left by the number of bits in **`rs`**, storing the result in **`rd`**. Each most significant bit shifted out of **`rd`** slides the remaining bits to the left by one, replacing the least significant bit with **`0`**.

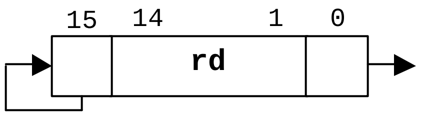
Notes: Overflow is ignored. **`rd`** is undefined if **`[rs]`** is negative.

15 12	11 8	7 4	3 0
0111	**0010**	**rd**	**rs**

ASR

Arithmetic Shift Right

Operation: `rd ¬ rd >> rs`



Syntax: `ASR rd, rs`

Description: Bit shifts **`rd`** to the right by the number of bits in **`rs`**, storing the result in **`rd`**. Each least significant bit shifted out of **`rd`** slides the remaining bits to the right by one, leaving a copy of the most significant bit in place.

Notes: Borrow is ignored. **`rd`** is undefined if **`[rs]`** is negative.

15 12	11 8	7 4	3 0
0111	**0011**	**rd**	**rs**

BEQ

Branch On Equal

Operation: `if rs1 = rs2 then pc ¬ addr`

Syntax: `BEQ rs1, rs2, addr`

Description: Branches to the address specified in the next word following the current instruction if **`rs1`** and **`rs2`** are both the same value.

Notes: **`addr`** is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	**0010**	**rs1**	**rs2**

BG

Branch On Greater Than

Operation: `if rs1 > rs2 then pc ¬ addr`

Syntax: `BG rs1, rs2, addr`

Description: Branches to the address specified in the next word following the current instruction if **`rs1`** is greater than **`rs2`**.

Notes: **`addr`** is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	**0111**	**rs1**	**rs2**

BGE

Branch On Greater Than Or Equal To

Operation: `if rs1 >= rs2 then pc ¬ addr`

Syntax: `BGE rs1, rs2, addr`

Description: Branches to the address specified in the next word following the current instruction if **`rs1`** is greater than or equal to **`rs2`**.

Notes: **`addr`** is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	**0101**	**rs1**	**rs2**

BL

Branch On Less Than

Operation: `if rs1 < rs2 then pc ¬ addr`

Syntax: `BL rs1, rs2, addr`

Description: Branches to the address specified in the next word following the current instruction if **`rs1`** is less than **`rs2`**.

Notes: **`addr`** is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	**0110**	**rs1**	**rs2**

BLE

Branch On Less Than Or Equal To

Operation: `if rs1 <= rs2 then pc ¬ addr`

Syntax: `BLE rs1, rs2, addr`

Description: Branches to the address specified in the next word following the current instruction if **`rs1`** is less than or equal to **`rs2`**.

Notes: **`addr`** is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	**0100**	**rs1**	**rs2**

BN

Branch On Negative

Operation: **`if rs < 0 then pc ¬ addr`**

Syntax: **`BN rs, addr`**

Description: Branches to the address specified in the next word following the current instruction if **`rs`** is negative.

Notes: **`addr`** is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	**0110**	**rs**	**rz**

BNE

Branch On Not Equal

Operation: **`if rs1 != rs2 then pc ¬ addr`**

Syntax: **`BNE rs1, rs2, addr`**

Description: Branches to the address specified in the next word following the current instruction if **`rs1`** and **`rs2`** are different values.

Notes: **`addr`** is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	**0011**	**rs1**	**rs2**

BNN

Branch On Not Negative

Operation: **`if rs >= 0 then pc ¬ addr`**

Syntax: **`BNN rs, addr`**

Description: Branches to the address specified in the next word following the current instruction if **`rs`** is not negative.

Notes: **`addr`** is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	**0101**	**rs**	**rz**

BNP

Branch On Not Positive

Operation: `if rs <= 0 then pc ¬ addr`

Syntax: `BNP rs, addr`

Description: Branches to the address specified in the next word following the current instruction if `rs` is not positive.

Notes: `addr` is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	0100	rs	rz

BNZ

Branch On Non Zero

Operation: `if rs != 0 then pc ¬ addr`

Syntax: `BNZ rs, addr`

Description: Branches to the address specified in the next word following the current instruction if `rs` is non zero.

Notes: `addr` is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	0011	rs	rz

BP

Branch On Positive

Operation: `if rs > 0 then pc ¬ addr`

Syntax: `BP rs, addr`

Description: Branches to the address specified in the next word following the current instruction if `rs` is positive.

Notes: `addr` is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	0111	rs	rz

BR

Unconditional Branch

Operation: `pc ¬ addr`

Syntax: `BR addr`

Description: Unconditionally branches to the address specified in the next word following the current instruction.

Notes: `addr` is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	0000	----	----

BSR

Branch to Subroutine

Operation: `(sp)++ ¬ pc;   pc ¬ addr`

Syntax: `BSR addr`

Description: Unconditionally branches to the address specified in the next word following the current instruction. A subsequent `RTS` instruction returns execution to the instruction following the `BSR`.

Notes: `addr` is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	1000	----	----

BVD

Branch On Video Drawing

Operation: `if 'video is drawing' then pc ¬ addr`

Syntax: `BVD addr`

Description: Branches to the address specified in the next word following the current instruction if the video unit is busy redrawing the screen and cannot accept new parameters.

Notes: `addr` is specified as a label which is resolved by the assembler. Video object properties should only be changed when the video is not drawing.

15 12	11 8	7 4	3 0
0100	1100	----	----

BZ

Branch On Zero

Operation: `if rs = 0 then pc ¬ addr`

Syntax: `BZ rs, addr`

Description: Branches to the address specified in the next word following the current instruction if **`rs`** is zero.

Notes: **`addr`** is specified as a label which is resolved by the assembler.

15 12	11 8	7 4	3 0
0100	**0010**	**rs**	**rz**

CLR

Clear All Register Bits

Operation: `r ¬ 0x0000`

Syntax: `CLR r`

Description: Clears all of the specified register’s bits (sets the register to zero).

Notes: none

15 12	11	10 8	7 4	3 0
0101	**0**	**000**	**r**	**rz**

DEC

Signed Decrement By 1

Operation: `r ¬ r - 1`

Syntax: `DEC r`

Description: Decrements **`r`** by **`1`** and stores the result back into **`r`**.

Notes: Borrow is ignored.

15 12	11 8	7 4	3 0
1000	**r**	**r**	**rn**

DIV

Signed Division

Operation: `rd ¬ rs1 / rs2`   `rt ¬ rs1 % rs2`

Syntax: `DIV rd, rs1, rs2`

Description: Divides `rs2` into `rs1`, placing the 16 bits result in `rd`. The 16 bit remainder is placed in the temporary register `rt`.

Notes: This instruction modifies `rt`. Therefore `rd` cannot be `rt`.

15 12	11 8	7 4	3 0
1011	**rd**	**rs1**	**rs2**

INC

Signed Increment By 1

Operation: `r ¬ r + 1`

Syntax: `INC r`

Description: Increments `r` by `1` and stores the result back into `r`.

Notes: Overflow is ignored.

15 12	11 8	7 4	3 0
1001	**r**	**r**	**rn**

LOAD

Register or Memory Transfer to Register

Operation: `rd ¬ rs` `rd ¬ (rs)` `rd ¬ as` `rd ¬ #is`

Syntax: `LOAD rd, rs` `LOAD rd, as`

`LOAD rd, (rs)` `LOAD rd, #is`

Description: Moves a 16 bit value from source to the destination register using one of the following effective addressing modes:

Code	Mnemonic	Name	Description
`0b000`	**`rs`**	Register Direct	Source is the register specified by **`rs`**.
`0b001`	**`(rs)`**	Register Indirect	Source is at the memory address contained in the register specified by **`rs`**.
`0b010`	**`as`**	Absolute Address	Source is the memory address specified in the syntax instead of a register.
`0b011`	**`#is`**	Immediate Value	Source is the immediate 16 bit constant specified in the syntax instead of a register, prefixed by a '**`#`**'.
`0b1??`	**`n/a`**	Reserved	Reserved for future implementation

Source Effective Addressing Modes

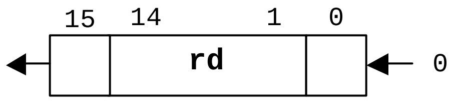
If absolute addressing or immediate value is used, the next word following the current instruction is that absolute memory address or immediate value. Whenever an addressing mode requiring an additional 16 bit argument is used, the contents of the **`rs`** register field is undefined.

Notes: none

15 12	11	10 8	7 4	3 0
0101	**0**	**sea**	**rd**	**rs**

LSL

Logical Shift Left



Operation: **`rd ¬ rd << rs`**

Syntax: **`LSL rd, rs`**

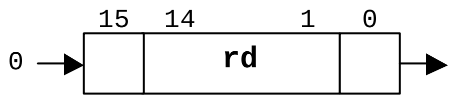
Description: Bit shifts **`rd`** to the left by the number of bits in **`rs`**, storing the result in **`rd`**. Each most significant bit shifted out of **`rd`** slides the remaining bits to the left by one, replacing the least significant bit with **`0`**.

Notes: Overflow is ignored. **`rd`** is undefined if **`[rs]`** is negative.

15 12	11 8	7 4	3 0
0111	**0000**	**rd**	**rs**

LSR

Logical Shift Right



Operation: **`rd ¬ rd >> rs`**

Syntax: **`LSR rd, rs`**

Description: Bit shifts **`rd`** to the right by the number of bits in **`rs`**, storing the result in **`rd`**. Each least significant bit shifted out of **`rd`** slides the remaining bits to the right by one, replacing the most significant bit with **`0`**.

Notes: Borrow is ignored. **`rd`** is undefined if **`[rs]`** is negative.

15 12	11 8	7 4	3 0
0111	**0001**	**rd**	**rs**

MULT

Signed Multiplication

Operation: **`rt:rd ¬ rs1 * rs2`**

Syntax: **`MULT rd, rs1, rs2`**

Description: Multiplies **`rs2`** and **`rs1`**, placing the lower 16 bits of the result in **`rd`**. The upper 16 bits of the result are placed in the temporary register **`rt`**.

Notes: This instruction modifies **`rt`**. Therefore **`rd`** cannot be **`rt`**.

15 12	11 8	7 4	3 0
1010	**rd**	**rs1**	**rs2**

NEG

2's Compliment Negation

Operation: `rd ¬ -rs`

Syntax: `NEG rd, rs`

Description: Stores the 2's compliment negation of `rs` in `rd`.

Notes: none

15 12	11 8	7 4	3 0
1001	**rd**	**rz**	**rs**

NOP

No Operation

Operation: none

Syntax: `NOP`

Description: Causes the microprocessor to wait for one clock cycle.

Notes: none

15 12	11 8	7 4	3 0
0000	**0000**	**----**	**----**

NOT

Bitwise NOT

Operation: `rd ¬ ~rs`

Syntax: `NOT rd, rs`

Description: Stores the bitwise NOT of `rs` in `rd`.

Notes: none

15 12	11 8	7 4	3 0
1110	**rd**	**rs**	**rn**

OR

Bitwise OR

Operation: `rd ¬ rs1 | rs2`

Syntax: `OR rd, rs1, rs2`

Description: Stores the bitwise OR of **`rs1`** and **`rs2`** in **`rd`**.

Notes: none

15 12	11 8	7 4	3 0
1101	**rd**	**rs1**	**rs2**

PIN

I/O Bus Port Input

Operation: `rd ¬ port(rp)`

Syntax: `PIN rp, rd`

Description: Reads a word from the I/O port whose id is in **`rp`**, storing it in **`rd`**.

Notes: Undefined if **`rp`** contains an invalid I/O port id.

15 12	11 8	7 4	3 0
0010	**0000**	**rp**	**rd**

POUT

I/O Bus Port Output

Operation: `port(rp) ¬ rs`

Syntax: `POUT rp, rs`

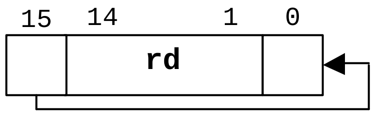
Description: Writes the word stored in **`rs`** to the I/O port whose id is in **`rp`**.

Notes: Undefined if **`rp`** contains an invalid I/O port id.

15 12	11 8	7 4	3 0
0010	**0001**	**rp**	**rs**

ROTL

Rotate Left



Operation: `rd ¬ rd ↺ rs`

Syntax: `ROTL rd, rs`

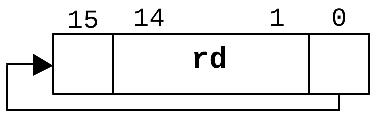
Description: Bit rotates **`rd`** to the left by the number of bits in **`rs`**, storing the result in **`rd`**. Each most significant bit rotated out of **`rd`** slides the remaining bits to the left by one, and is put back into the least significant bit.

Notes: Overflow is ignored. **`rd`** is undefined if **`[rs]`** is negative.

15 — 12	11 — 8	7 — 4	3 — 0
0111	**0100**	**rd**	**rs**

ROTR

Rotate Right



Operation: `rd ¬ rd ↻ rs`

Syntax: `ROTR rd, rs`

Description: Bit rotates **`rd`** to the right by the number of bits in **`rs`**, storing the result in **`rd`**. Each least significant bit rotated out of **`rd`** slides the remaining bits to the right by one, and is put back into the most significant bit.

Notes: Borrow is ignored. **`rd`** is undefined if **`[rs]`** is negative.

15 — 12	11 — 8	7 — 4	3 — 0
0111	**0101**	**rd**	**rs**

RTS

Return from Subroutine

Operation: `pc ¬ --(sp)`

Syntax: `RTS`

Description: Returns execution to the instruction following the last **`BSR`** instruction.

Notes: **`addr`** is specified as a label which is resolved by the assembler. Undefined behaviour if there was no matching **`BSR`** instruction.

15 — 12	11 — 8	7 — 4	3 — 0
0100	**1001**	**----**	**----**

STOP

Stop Program Execution

Operation: halt processor

Syntax: `STOP`

Description: Causes the microprocessor to stop executing instructions. This the normal method for terminating a program cleanly

Notes: Any instructions after a **`STOP`** instruction will not be executed.

15 12	11 8	7 4	3 0
0000	**0001**	**----**	**----**

STORE

Register Transfer to Register or Memory

Operation: `rd ¬ rs` `(rd) ¬ rs` `ad ¬ rs`

Syntax: `STORE rd, rs` `STORE (rd), rs` `STORE ad, rs`

Description: Moves a 16 bit value from the source register to the destination using one of the following effective addressing modes:

Code	Mnemonic	Name	Description
0b000	**rd**	Register Direct	Destination is the register specified by **`rd`** respectively.
0b001	**(rd)**	Register Indirect	Destination is the memory address contained in the register specified by **`rd`**.
0b010	**ad**	Absolute Address	Destination is the memory address specified in the syntax instead of a register.
0b011	**n/a**	Invalid	Invalid addressing mode
0b1??	**n/a**	Reserved	Reserved for future implementation

Destination Effective Addressing Modes

If absolute addressing is used, the next word following the current instruction is that absolute memory address. Whenever an addressing mode requiring an additional 16 bit argument is used, the contents of the **`rd`** register field is undefined.

Notes: none

15 12	11	10 8	7 4	3 0
0101	**1**	**dea**	**rd**	**rs**

SUB

Signed Subtraction

Operation: `rd ¬ rs1 - rs2`

Syntax: `SUB rd, rs1, rs2`

Description: Subtracts `rs2` from `rs1` and stores the result in `rd`.

Notes: Overflow and Borrow are ignored.

15 12	11 8	7 4	3 0
1001	**rd**	**rs1**	**rs2**

VBP

Set Video Object Bottom Position

Operation: `video(rv).bottom ¬ rs`

Syntax: `VBP rv, rs`

Description: Sets the bottom position of the video object, whose id is in `rv`, to the value stored in `rs`.

Notes: Undefined if `rv` contains an invalid video object id, if the target object is not a sprite or shape, or if the value of `rs` is outside the valid range of the screen.

15 12	11 8	7 4	3 0
0011	**0011**	**rv**	**rs**

VC

Set Video Object Colour

Operation: `video(rv).colour ¬ rs`

Syntax: `VC rv, rs`

Description: Sets the color of the video object, whose id is in `rv`, to the value stored in `rs`.

Notes: Undefined if `rv` contains an invalid video object id, if the value of `rs` is outside the valid range of colours, or if the target object is not a shape or the background.

15 12	11 8	7 4	3 0
0011	**0100**	**rv**	**rs**

VD

Disable Video Object

Operation: `video(rv).disable()`

Syntax: `VD rv`

Description: Disables the video object whose id is in **`rv`**.

Notes: Undefined if **`rv`** contains an invalid video object id, or if the target object is not a sprite or shape.

15 12	11 8	7 4	3 0
0011	**1001**	**rv**	**----**

VE

Enable Video Object

Operation: `video(rv).enable()`

Syntax: `VE rv`

Description: Enables the video object whose id is in **`rv`**.

Notes: Undefined if **`rv`** contains an invalid video object id, or if the target object is not a sprite or shape.

15 12	11 8	7 4	3 0
0011	**1000**	**rv**	**----**

VI

Set Video Object Image Address

Operation: `video(rv).image ¬ rs`

Syntax: `VI rv, rs`

Description: Sets the image address of the video object, whose id is in **`rv`**, to the value stored in **`rs`**.

Notes: Undefined if **`rv`** contains an invalid video object id, or if the target object is not a sprite.

15 12	11 8	7 4	3 0
0011	**0101**	**rv**	**rs**

VLP

Set Video Object Left Position

Operation: `video(rv).left ¬ rs`

Syntax: `VLP rv, rs`

Description: Sets the left position of the video object, whose id is in **`rv`**, to the value stored in **`rs`**.

Notes: Undefined if **`rv`** contains an invalid video object id, if the target object is not a sprite or shape, or if the value of **`rs`** is outside the valid range of the screen.

15 12	11 8	7 4	3 0
0011	0000	rv	rs

VRP

Set Video Object Right Position

Operation: `video(rv).right ¬ rs`

Syntax: `VRP rv, rs`

Description: Sets the right position of the video object, whose id is in **`rv`**, to the value stored in **`rs`**.

Notes: Undefined if **`rv`** contains an invalid video object id, if the target object is not a sprite or shape, or if the value of **`rs`** is outside the valid range of the screen.

15 12	11 8	7 4	3 0
0011	0001	rv	rs

VTP

Set Video Object Top Position

Operation: `video(rv).top ¬ rs`

Syntax: `VTP rv, rs`

Description: Sets the top position of the video object, whose id is in **`rv`**, to the value stored in **`rs`**.

Notes: Undefined if **`rv`** contains an invalid video object id, if the target object is not a sprite or shape, or if the value of **`rs`** is outside the valid range of the screen.

15 12	11 8	7 4	3 0
0011	0010	rv	rs

XOR

Bitwise XOR

Operation: **`rd ¬ rs1 ⊕ rs2`**

Syntax: **`XOR rd, rs1, rs2`**

Description: Stores the bitwise XOR of **`rs1`** and **`rs2`** in **`rd`**.

Notes: none

15 12	11 8	7 4	3 0
1110	**rd**	**rs1**	**rs2**