32-Bit Custom RISC-V Architecture:

Implementation Details

Ryan Riccio

Spring 2024

Table of Contents

Executive Summary

The scope of this project is not to develop a "finished product" architecture, rather a developmental architecture that is modular and easily added upon. Many optimizations or other hardware designs will be overlooked in favor of simplicity. This is solely due to the scholastic nature of this project, as well as time constraints. As will be shown, quality is still at the core of this project, however in terms of modern technological advancements, there will always be more efficient, smaller, and more polished applications of what this project seeks to create. What is key to this specific project, however, is that the depth and scope at which this project lies are structured in a way that an entry level computer scientist would be able to understand this platform, as well as quickly develop for it using modern hardware.

Introduction

This project will dive deep into the inner workings of a modern computer. This project will include a functionally low-level implementation of custom architecture. The architecture will be combined with other functional components to provide a simple demonstration of a functioning computer. To design a logically correct implementation, the project will first focus on drawing and creating accurate logic diagrams, as well as simulating combinational logic. To demonstrate the architecture, this project will also focus on the development of a custom emulator that will emulate the functions of traditional logic gates and data transmission methods. This emulator will be able to precisely run the machine code given to it in a repeatable and predictable manner. The machine code will instruct the computer how to execute actions, and the computer's microcode will be based on a given instruction. This machine will also be paired with its own kernel for integration with basic operating system tasks. No file system will be provided; however, peripherals, video, and serial communication will be handled by the custom kernel. The final goal of this project is to be able to compile code targeted at this architecture from "higher" level languages such as C and C++. This custom architecture will create a platform that will allow for further research into the possible ways to make modern computers simpler and more efficient. Using the open-source RISC-V (Reduced Instruction Set Computer 5) instruction set, it will be possible to make a low power, highly optimized architecture for less intensive tasks. As opposed to modern CISC (complex instruction set computer) architectures like x86, uniform data formatting as well as a simpler instruction set will lead to easier understandability and extensibility. RISC-V SoC's (system on a chip) have been gaining popularity in the single board computer scene (similar to Raspberry Pi) due to their license free, easily modifiable, efficient architecture. Some configurations of RISC-V can even support the modern Linux kernel. Canonical, the company that manages and releases Ubuntu, has even released images for running Ubuntu on RISC-V. With the ever-expanding need for a more efficient desktop architecture (see the rise of desktop ARM chips with Apple Silicon), will RISC-V be the next big thing due to its low-cost, its excellent power efficiency, and its relative availability?

Current Situation

In the current state of things, RISC-V is rapidly being adopted into many different types of systems. It is constantly becoming more and more prevalent in all parts of the computer science world. Rather than fighting with physical logic gates and circuit boards, it is necessary to build this computer virtually. This will allow complete end-to-end testing of the entire architecture. Since everything will have to be built from the ground up it is important that the virtual representation of this architecture is implemented in a cycle-accurate way, meaning that each clock cycle will have to be implemented properly. The goal for this emulator/architecture is for it to be capable of being implemented in hardware, however hardware implementation is not necessarily in the scope of this project. This emulator is more of a proving ground for this architecture.

Objectives

The goal of this project is to provide an easy to understand, low-level logical simulation of a RISC-V architecture, and then transition logical diagrams into a virtual emulation of the architecture. This program will be able to render a cycle accurate representation of the output of this computer, as well as provide diagnostic tools and some form of SDK. It will prove that RISC-V based architectures can be made efficiently, especially for small programming applications. It will also serve as a tool to help entry level software engineers understand the inner workings of RISC-V.

Past Attempts

Many people have made RISC-V emulators, however many of them either aim for pure simplicity or maximum efficiency. In terms of pure simplicity, many important features are left out, and the emulator/logical design of the core is too simplified to perform modern tasks. On the other hand, very highly efficient RISC-V emulators are capable of running Linux, however the complexity adds to the barrier to entry for people to tinker with the instruction set. This project aims to create a highly maintainable, extendable, easy to understand emulator that can completely function to an extent similar to a “retro” computer like a Commodore 64 or early IBM PC. This allows programmers to be able to tinker with the low-level side of things, while still being able to use modern programming features that they are accustomed to. This project also aims to document, at a discrete logical level, what is going on behind the scenes. This will allow end users to modify and simulate logic diagrams, and then easily implement these modifications into their emulator, seeing the effects of their changes with minimal code.

Biggest Challenges

Some of the biggest challenges of this project will be making sure all functional components work together. For example, the clock needs to interact with all components. Each of these components is required to function with the others in a specified/documented way. Those parts need to then be able to display their contents to the user in a manageable way. The architecture then needs to be robust enough to take whatever machine code is thrown at it, whether from the programmer directly or through the compiler. It needs to use this machine code in a repeatable and predictable manner. On top of this, setting up a compiler for a custom architecture is an entirely different endeavor. There are many solvable problems and challenges in developing this architecture. This is just an insight into some of the biggest challenges that will be faced.

Group Information (Qualifications)

Ryan Riccio has experience working with low level microcode whether that is on a Commodore 64, or a breadboard 8-bit computer. He has a fair bit of experience writing safe, scalable, modern C++ backends with high performance, as well as rendered front ends (OpenGL, ImGui). He enjoys abstract, high level system design for implementing multiple functional parts into complex systems. With his experience in modern C/C++ and multiple assembly languages, Ryan is qualified to implement low level operations in a high level, modular way. With experience in C++ UX design, Ryan will use this knowledge to create a software emulator to model the custom architecture. Ryan's experience with digital circuits is critical to designing this computer so that it would be feasible to implement it in physical hardware. Ryan has also done some experiments with the Clang compiler as well as GCC. This will serve as vital knowledge when trying to target this architecture during compilation. The inspiration behind this project starts with Ryan's first non-x86 computer. At about the same time, Ryan obtained a Commodore 64, 128, and an NES, all running 6502 based processors. His goal was to write software for the C64 without using the BASIC interpreter. After many small attempts to write simple software, Ryan was able to display to the screen and do basic tasks. With the programming experience Ryan gained while completing his undergraduate, he has the tools necessary to complete this project in a professional and timely manner.

Design

There are three major parts to the design of this project. There is the logic design, the emulator, and the software library. These functional components come together to form the implementation of the architecture in a way that allows iterative testing and development. The emulator will be based on the logic diagrams created, allowing the software library to be tested on either.

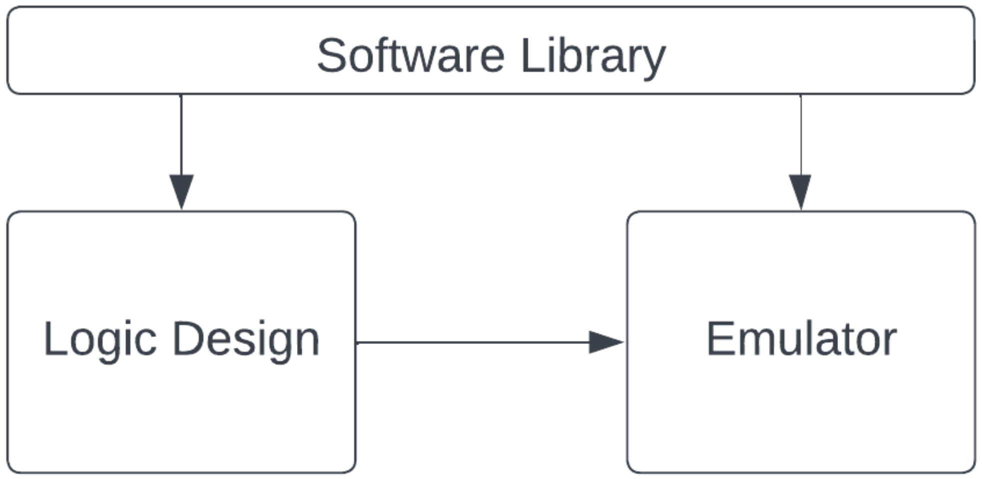


Fig. 1: Implementation Overview.

Overview

In terms of layers of abstraction, these functional components can be broken down into many small components that come together to form the overall architecture implementation. For example, the logic design will independently implement the clock, program counter, ALU, etc. These will come together to form the core, which can then be implemented with the peripherals and video. The video module and the peripheral module can work independently from the internal implementation of the core due to this abstraction. Each individual overarching section will contain more information on its specific layered implementation.

Logic Design

The logic design section contains simulated logic circuits that are designed and simulated inside of Logisim Evolution. Logisim Evolution is the latest, community upkept fork of Logisim by Carl Burch. Burch was an educator until 2014 when he abandoned the project for a position at Google. Logisim Evolution allows for accurate (but very slow) emulations of combinational logic. It also allows for the logic diagrams to be converted to VHDL, allowing for possible expansion of this project in the future to a physical development board. Logisim Evolution is also beneficial in the fact that it has some prebuilt combinational logic. For example, instead of having to implement a full adder using discrete logic, Logisim Evolution can take 2 numbers of a specified bit width and add them together. This does not minimize the number of architectural decisions that are to be made, but it allows the logic to be more focused on operations rather than the specificity of how to implement an adder in discrete logic gates.

Since the emulator will be based on logic diagrams, this has been taken into account when designing the logic. Some parts, however, do not line up exactly every single time. This requires workarounds in Logisim that might not be seen in the emulator, and vice versa.

Overall, the logic for this architecture can be broken down into the following components: CPU, Memory, Peripherals, and VGA. Each component is designed to be able to interact in a pseudo-bus style interface. It is not a real bus since nothing is buffered and relies solely on timing and restrictions. The CPU controls the address, and all the components can exchange data on the data bus. The CPU also controls memory accesses and writes. Figure 2 gives insight into the data path between units.

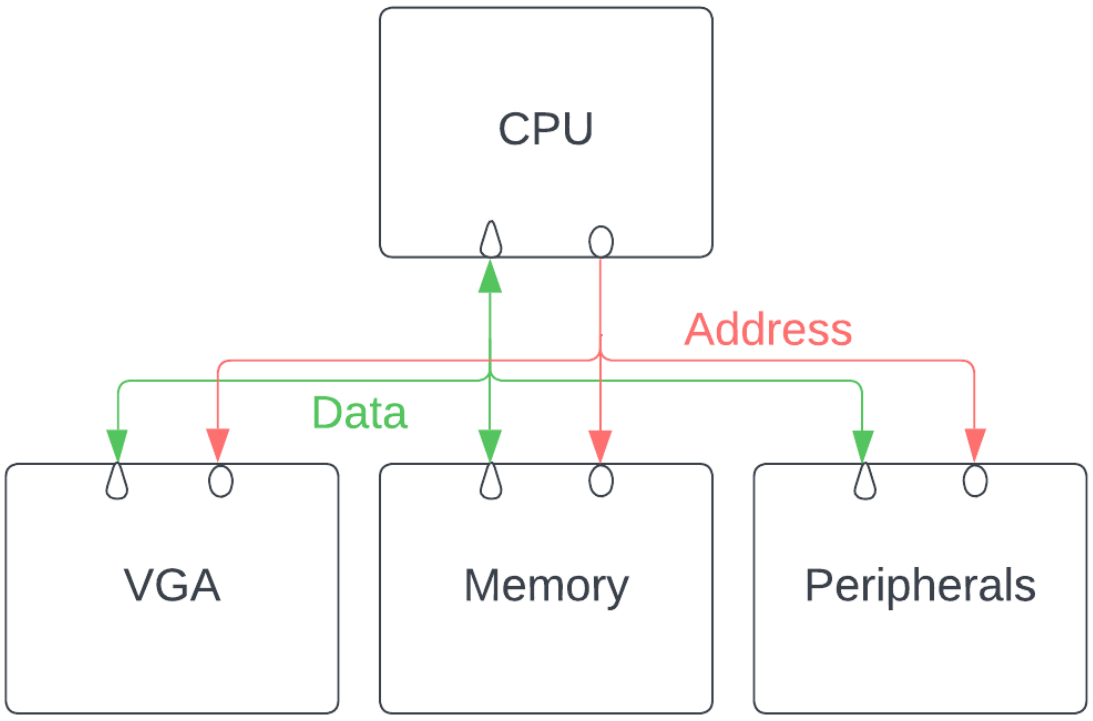


Fig. 2: Computer Data Flow

The clock is controlled by the VGA card. This allows the VGA card to synchronize memory reads and writes, mitigating the need for double buffering to prevent corruption to the screen data. Screen tearing can still occur since the video memory may be updated mid-frame. Allowing the VGA control logic to set the system clock allows for optimizations like cycle-sharing as well. In systems like the Commodore 64, the VIC-II chip (video chip) is capable of stealing clock cycles from the CPU, blocking its execution until it is finished with its memory access. This is not used in the current design; however, it is a future optimization that may be made. In the current design, RAM is not a bottleneck. This allows access to memory at two separate locations during the same clock cycle. This removes the need for cycle sharing and allows the CPU to asynchronously write to the display buffer without cycle sharing. Figure 3 shows how the system clock is generated inside of the VGA circuitry.

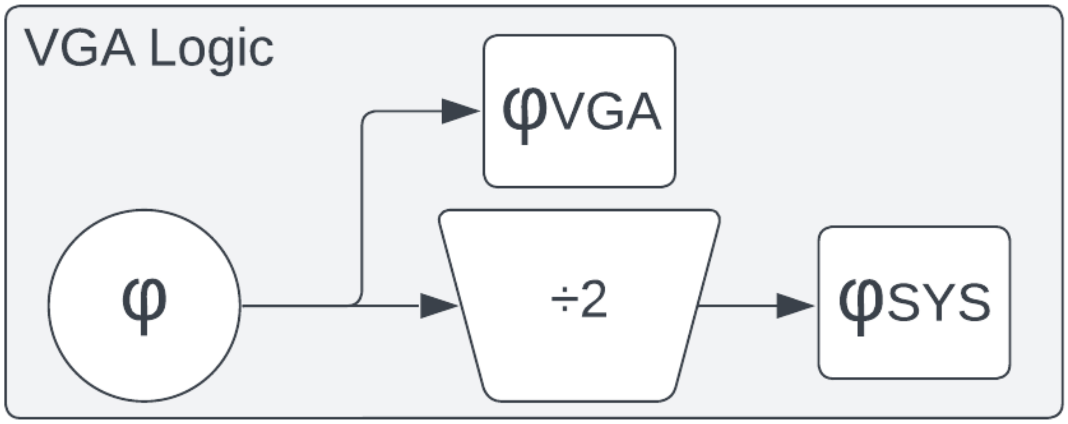


Fig. 3: System Clock Origin

In a further section, interrupts will be covered in more depth. The basic overview in terms of the high-level logic is as follows: an interrupt line is set high by any unit, causing the CPU to jump to a hard-coded interrupt handler. This interrupt handler is up to either the kernel or the programmer to determine how to clear the interrupt. Interrupts are disabled until the current interrupt is handled. Normally, this would prevent multiple interrupts from occurring at the same time, however in this system, each interrupt is a bitflag, so if multiple interrupts occur before the handler reads the interrupt register, the programmer can decode what occurred based on the specified interrupt register. In the current design, all interrupts are handled by the peripheral interface (PIA), however this can be expanded in the future simply by modifying the interrupt handler. The status of the system's interrupts is solely handled by the PIA, however this is not a limitation since all system components are capable of interrupting execution. In a normal RISC-V system, kernel calls, interrupts, traps, exceptions, etc., are handled by ECALL and EBREAK instructions, neither of which are supported in this system.

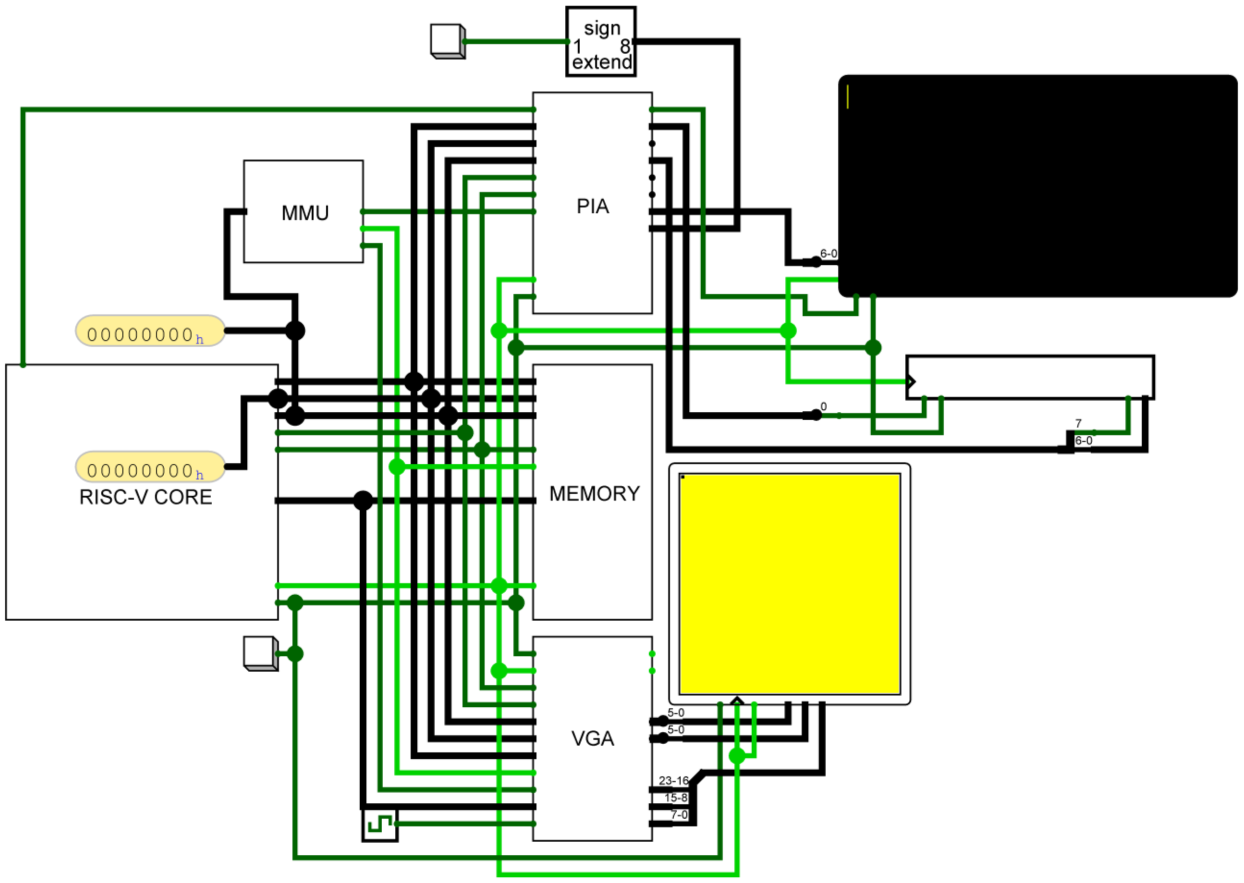
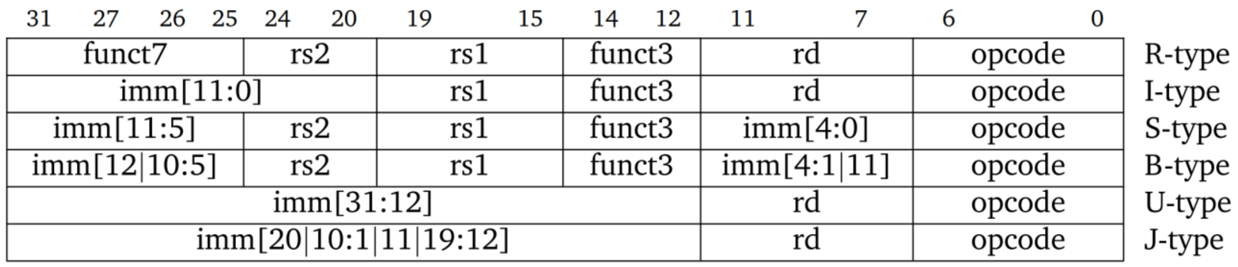


Fig. 4: Main Logic Diagram

CPU

The CPU, also known as the RISC-V Core, is a 32-bit 5-stage pipeline that implements the I (base integer) and M (multiply) extensions for RISC-V. Our pipeline consists of a Fetch (IF), Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB) stages. It does not implement ECALL and EBREAK, as all kernel type operations are handled in software, as well as hardware interrupts. In the Additional Documents section, Doc. 1 and Doc. 2 show the implemented instruction set. As with a normal implementation of RISC-V, we have R-type, I-type, S-type, B-type, U-type, and J-type instruction formats. In this load-store architecture, each of our instructions are 32-bits wide, and all opcodes are the last 7-bits. Because of this instruction format, all 32-bit reads from memory must be aligned to a word boundary.



31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Fig. 5: Instruction Format

Normally, modern RISC type systems such as MIPS have protected program memory that is separate from the data memory. Commonly, in system diagrams for these MIPS based systems, we even see these RAM/ROM sections as their own discrete part of the logic diagram. In our system, however, all program and data memory sections are the same, and it is external to the CPU core itself. This allows for the CPU to read from any address, whether that is a peripheral register, a UART controller, the VGA data, or simply just memory.

In full implementations of RISC-V, a CSR (control/status register) co-processor is included to keep track of the system's status. Since we have no definition of an exception in hardware (it is up to the programmer), a trap, or any other custom registers (such as a carry flag), arithmetic overflow exceptions and other issues will never occur, removing the need for a CSR co-processor.

Some RISC-V implementations include an FPU co-processor for floating point arithmetic. In this implementation, floating point arithmetic is handled by the software library. This is a tradeoff, sacrificing performance for simplicity. In the scope of this project, being able to support IEE-754 floats for simple calculations more than meets the expectations of this architecture.

As will be shown in more detail further on, our pipelined system requires us to do specific checks to avoid a stale pipeline. We do this by preventing data and control hazards, as well as forwarding register information back to the execution stage (FWD CTRL/HAZARD CTRL).

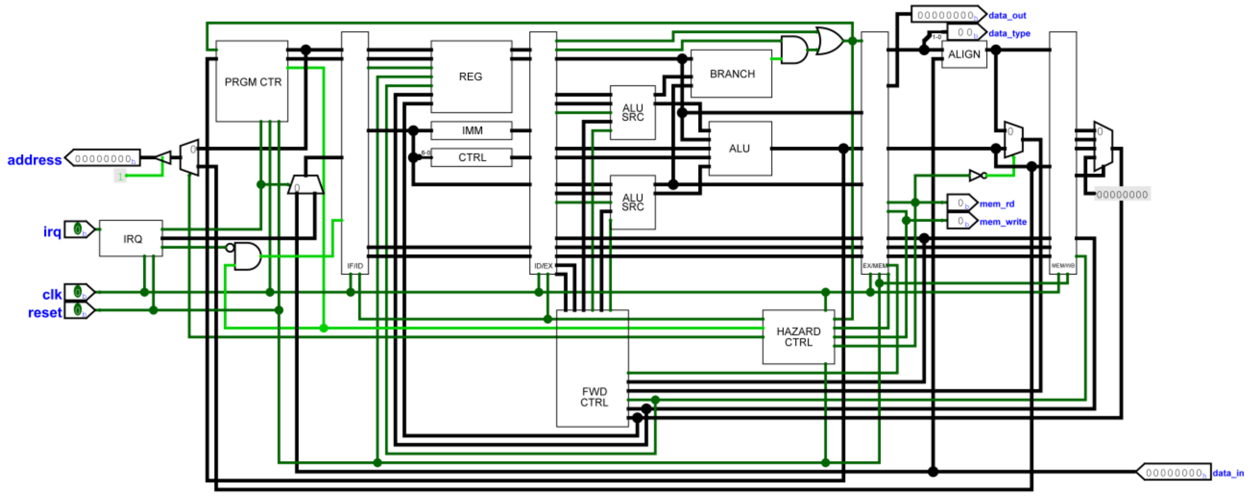


Fig. 6: RISC-V Core Logic Diagram

Control

The control logic occurs in the instruction decode phase. Simply put, the opcode is decoded to retrieve bitflags to control how the core operates on given data. These control signals are then propagated throughout the pipeline registers to end up in the proper unit for each of the stages. The control logic uses a programmable logic array to decode different types of opcodes to their given control word. The control word also instructs the ALU control unit on how to control the ALU internally. This is because the ALU, simply put, is our only major co-processor. IRQ, FWD, HAZARD, BRANCH, etc., are all conditional logic based on the main control word, or the output of the ALU. The control logic is very similar to other RISC based systems like MIPS. The logic is shown in Figure 7.

	LUI	AUIPC	JAL	JALR	BXX	LX	SX	RI	RR
JP	0	0	1	1	0	0	0	0	0
BR	0	0	0	0	1	0	0	0	0
RF_SRC	10	10	01	01	00	00	00	10	10
RF_WE	1	1	1	1	0	1	0	1	1
ALU_OP	11	00	00	00	00	00	00	10	01
ALU_A	0	1	1	0	1	0	0	0	0
ALU_B	1	1	1	1	1	1	1	1	0
MEM_RD	0	0	0	0	0	1	0	0	0
MEM_WR	0	0	0	0	0	0	1	0	0

Fig. 7: Control Logic

CTRL	UNIT
RF_SRC 00	Output from Memory
RF_SRC 01	Program Counter + 4
RF_SRC 10	ALU Output
ALU_OP 00	Add
ALU_OP 01	R-Type (not Add)
ALU_OP 10	I-Type
ALU_OP 11	Bypass (output B)
ALU_A 0	Register A
ALU_A 1	Program Counter
ALU_B 0	Register B
ALU_B 1	Immediate

Fig. 8: Control Logic Information

Immediate Generation

Immediates in RISC-V are encoded directly into the 32-bit instruction. Because of this, most instruction formats can only fit 12 bits of immediate data. This is where U-type instructions become useful. To load an entire 32-bit immediate into a register, the upper bits are loaded on a separate instruction using a U-type instruction. To be able to make use of all of the different immediate encodings in the different instruction formats, we have an immediate decode control unit. The immediate decode control unit generates immediate values using the encoding in the specific instruction. It uses the top 5 bits of the opcode to determine what kind of immediate is being decoded, then uses a programmable logic array to select between the different decodings. In some implementations (with more supported extensions), immediates can be decoded a bit more efficiently, but for sake of "readability" of the logic diagrams, each immediate type is decoded manually and separately.

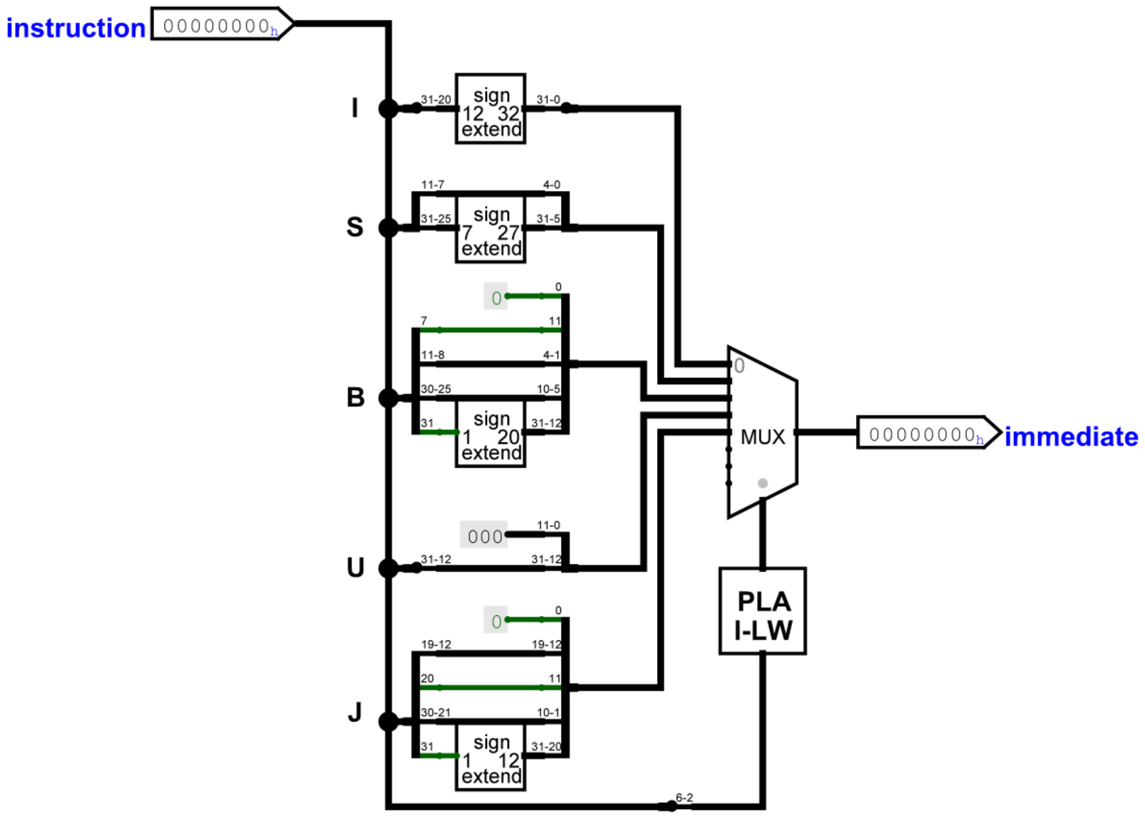


Fig. 9: Immediate Decode Logic

ALU

The arithmetic logic unit (ALU) from a general perspective is a control unit that takes in two numbers, A and B and performs an arithmetic or logical operation on these values. Different control paths allow for more complex operations. The ALU is split into a few functional components. The first part is the actual arithmetic section that performs calculations. It is split into two smaller parts, an integer extension computational unit, and a multiplication extension computational unit. The second overall part is the control unit for the ALU. This decodes control signals to provide the proper output. The last key part to the ALU is the ALU source unit. This unit is external to the ALU and is used to determine where the “A” and “B” values are coming from. As seen from the control unit diagram (Figure 8), there are different options based on the specified instruction. The ALU source unit also controls forwarding, as well as controlling the data path for registers values used in the branching logic.

The ALU, simply put, is just a collection of operations that are switched between using a multiplexer. It has the ability to “bypass” these features by just passing through the “B” value. This is useful in circumstances where immediates need to be used directly without adding anything to them. For example, in the LUI instruction, the upper bits of the immediate simply need to be passed to the register file with no modifications.

Since the ALU supports the “M” extension, the lower 8 operations are for the base “I” extension, while the upper 8 operations are from the “M” extension. RISC-V makes these easy to decode since the funct7 field of the R-type instructions will always be 0x01 when a multiplication operator is needed. This is simply passed along as the highest order bit in the control bits for the multiplexer. For all R and I-type arithmetic operations, the funct3 field is used to determine what kind of operation is needed. More details about this will be given in the “ALU Control” section.

In some cases, the ALU needs to perform different operations based on the interpretation of the bits coming in. Some data will be unsigned, and some will be 2’s complement. Logisim Evolution makes this easy to decode, since most of its sign-influenced functional blocks allow the selection of what kind of operation will be performed. This is why it may look like there are 2 comparators used for SLT and SLTU, this is because one is unsigned while the other is not. We see this show itself even more with the high multiplication operations. We sign extend (or zero extend) our input from 32 to 64 bits in order to properly multiply the signs without truncating data. We then chop off the bottom bits again back to 32 bits before sending it out of the ALU.

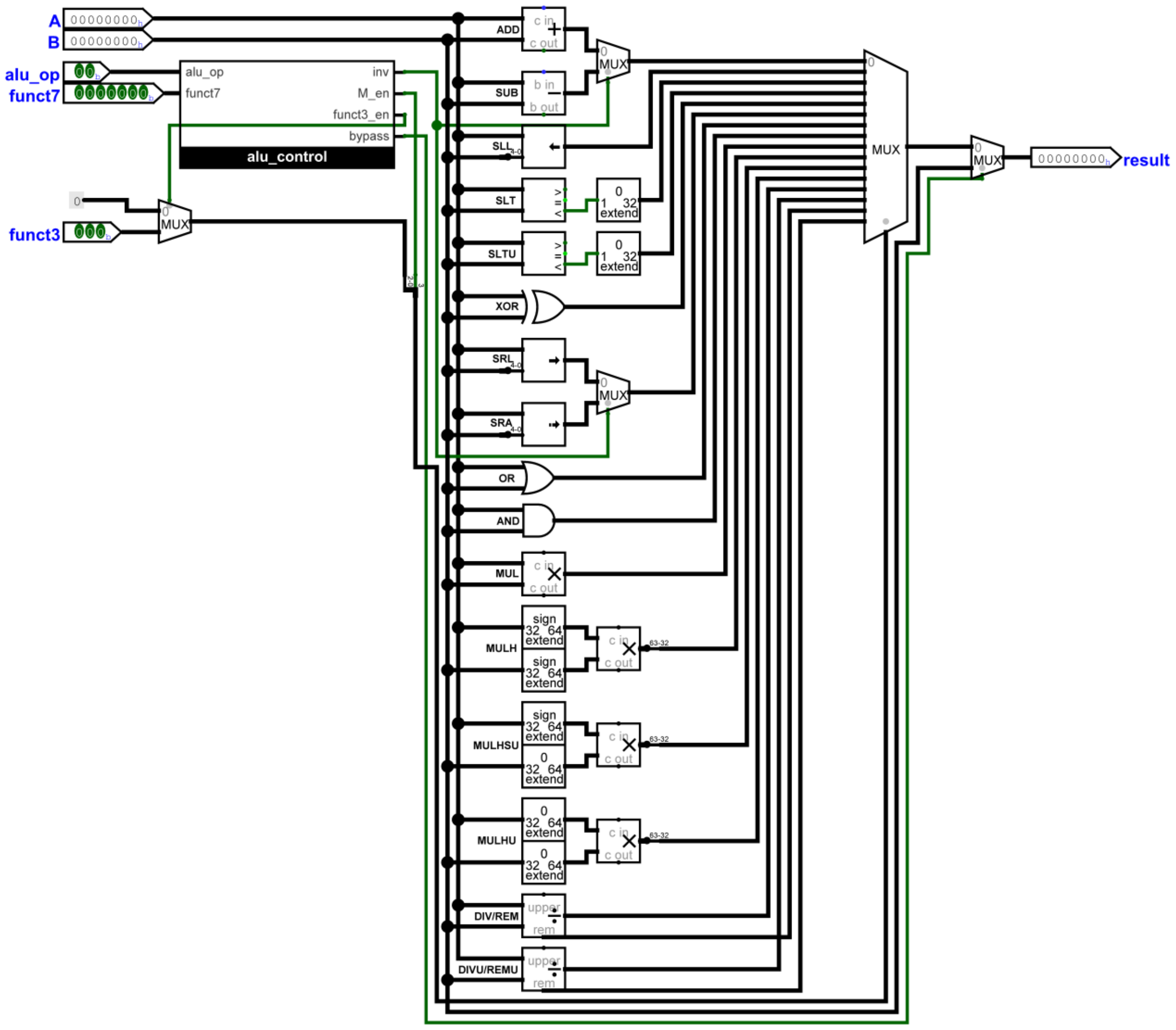


Fig. 10: ALU Diagram

ALU Control

The ALU Control unit is used to decode funct3, funct7, and ALU Op information to a specific ALU operation. This uses comparators and a programmable logic array to select different functions. It its default state, the ALU is instructed to only perform addition of operands “A” and

"B". In all other operations (aside from bypass), the funct3 field in the given instruction is used to index which function to use. Funct7 is used to determine if the inverted, or alternate function should be used (i.e. subtraction during an addition, or SRL during an SRA). Funct7 is also used to denote if a given operation is from the multiplication extension. The ALU Op is mostly used to bypass the ALU (used for passing through immediates), or for indicating whether an instruction is performed on a register or an immediate.

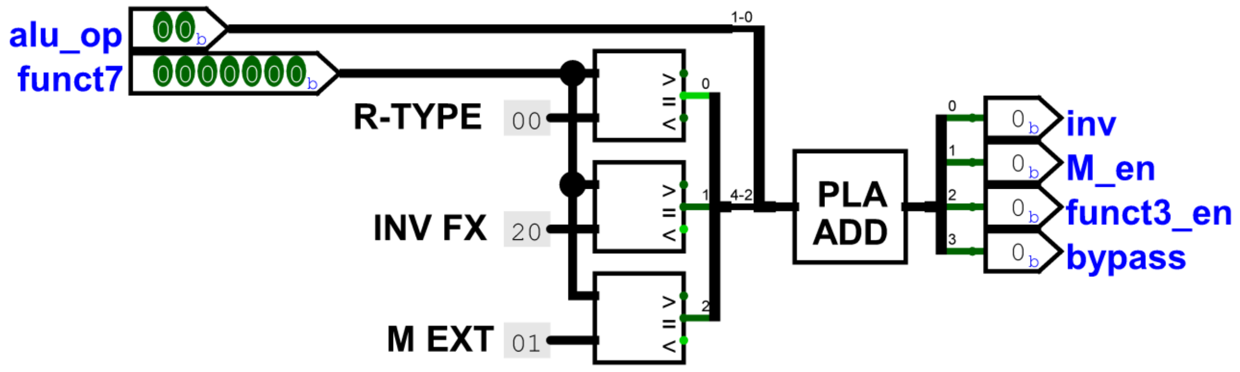


Fig. 11: ALU Control Unit Diagram

ALU Source

The ALU Source unit is primarily used for forwarding, passing immediates to the ALU, or passing register values to the branch logic. In some branch instructions, the specified immediate must be added to the program counter, in addition to a comparison. This means that the branch logic must be able to perform comparisons on register values, while still sending immediates and the program counter to the ALU. This is where the ALU Source unit comes into play. Depending on where data is coming from, we can use this like a "smart" multiplexer to only pass values when certain conditions have been met. For example, we might want to forward a register value to the branching logic before it has been calculated (see Forwarding section), this ALU source will allow us to do that. Another example would be sending the program counter to one of the ALU inputs.

This is possible with this control unit. The input pin “opt_data”, in our case, is either an immediate or the program counter. These will never be sent to the branching unit, which greatly simplifies this smart multiplexer. Also, in the case of data being forwarded from a later stage in the pipeline, this must be forwarded for both the ALU and the branch logic.

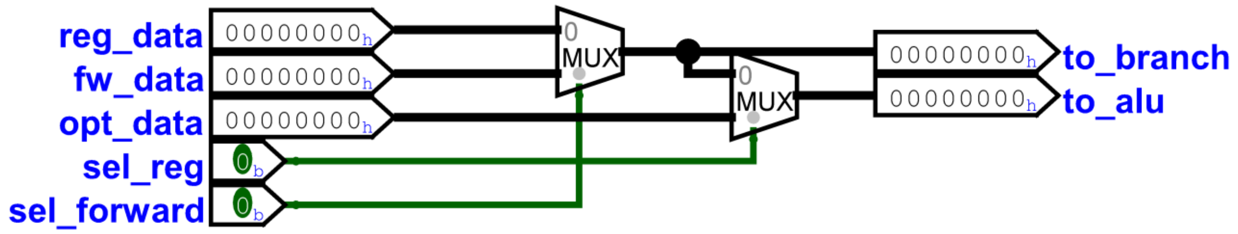


Fig. 12 ALU Source Control Unit Diagram

Registers

In modern computers, registers are the fastest memory that the computer can access. They are used for intermediate operations to store values that the computer is currently using. In this architecture, registers are faster simply because they can bypass the address calculation and gather their information in a single stage of the pipeline, however if this were a single stage architecture, with the current implementation, registers and memory would be equally as fast. Each of our registers are 32-bit wide, of which we have 32 registers. Register 0 is hardwired to zero, however, so this puts our usable registers to 31. In older CPUs like the 6502, the stack pointer is not a part of the register file. It is a register at its core, but it is accessed differently than other registers. In this architecture, all registers are equal. Any register can be used as a stack pointer, global pointer, return address pointer, etc. This is dictated by the programmer who normally lets the assembler manage the agreed upon functions of each register. There are naming conventions for RISC-V registers, however, which can be adopted by this architecture. See Document 4 for the standard naming conventions.

Our implementation of registers uses multiplexers and decoders to select registers. Decoders are used to convert an integer value into a “write enable” line for a given register (rd). Multiplexers are used to use an integer value to select which register to forward the output of. This is done twice to be able to read two registers at once (rs1 and rs2). All registers can share a clock and reset line. They also have a common write enable line to control whether the entire register file is open to writing or not.

On top of this, in a pipelined system, we will be reading from and writing to registers at the same time. How is this decoded? How is this manageable? It seems like it might cause many race conditions and undefined behavior if memory can be read from and written to at the same time. The fix for this is how the registers interpret the clock. All pipeline registers and other control units that rely on the clock store information on the rising edge of the clock. The main registers store data on the falling edge which allows predictable behavior in a pipelined system.

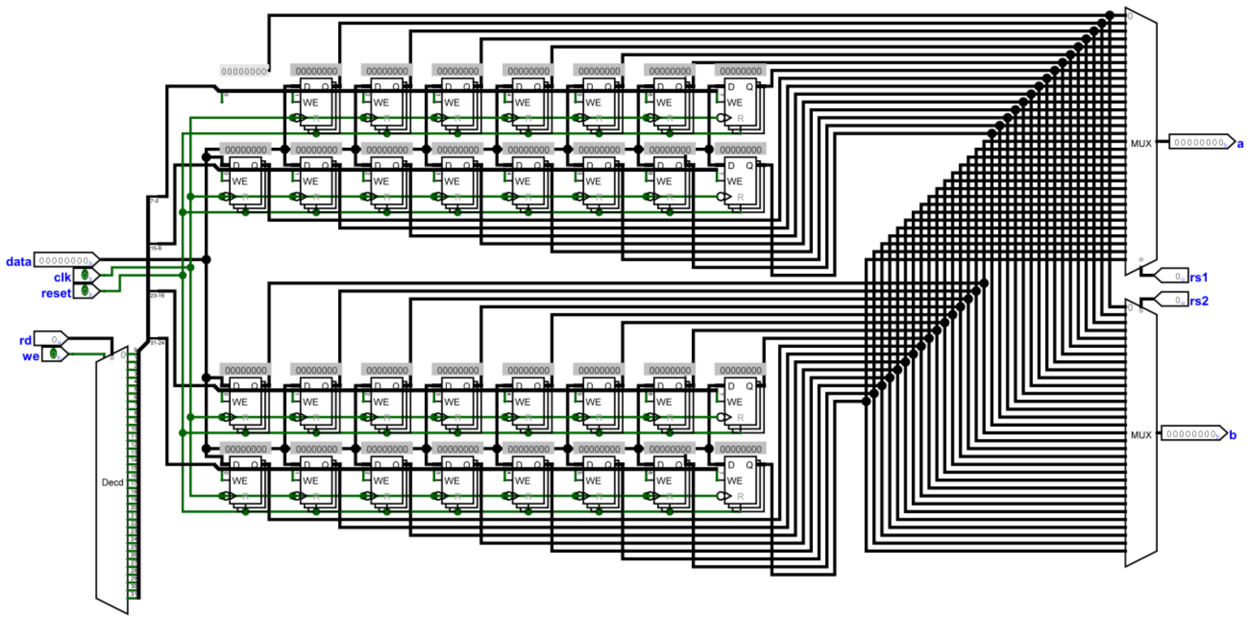


Fig. 13: Register File Diagram

Memory Operations

In this architecture, anything that is not a register is addressable as “memory”. The video circuit is memory mapped, the peripherals are memory mapped, and memory itself is mapped. All memory operations throughout the entire system are controlled by the CPU. The CPU dictates whether memory is being read from or written to. This is encoded into the instruction and decoded using the control unit. That means that all memory operation instructions are unconditional and will occur no matter the state of the system. This differs from other architectures, as this architecture requires the programmer to control the safety net of memory operations, rather than the system itself. On top of that, memory has to be “aligned”. All instructions must be on a 4-byte word boundary; however, data can be virtually wherever. This means that we must make sure that data being read in from memory is in the proper position before being ingested by the CPU. This is done with the align circuit. This circuit simply puts all non-word memory operation data at the least significant bit location, as well as extending the sign if required.

On top of data alignment, there is still the issue of memory read timing. For example, the IF stage reads instructions, while the MEM stage can either read or write from memory at the same time. This is handled by “pausing” the program counter and stalling the pipeline until the memory operation is complete. When the PC is disabled, the first pipeline stage will handle this by inserting a NOP. This allows us to continue operation without either corrupting or flushing the pipeline.

In terms of memory, how does the rest of the system know if the core wants to write a byte, half-word, or entire word? Luckily enough, funct3 encodes this in the bottom 2 bits. This is simply passed to the rest of the system. This allows all other functional units to decode the data how they see fit. Since our main registers are falling edge triggered, this also allows all memory operations to complete before data is actually written to the registers.

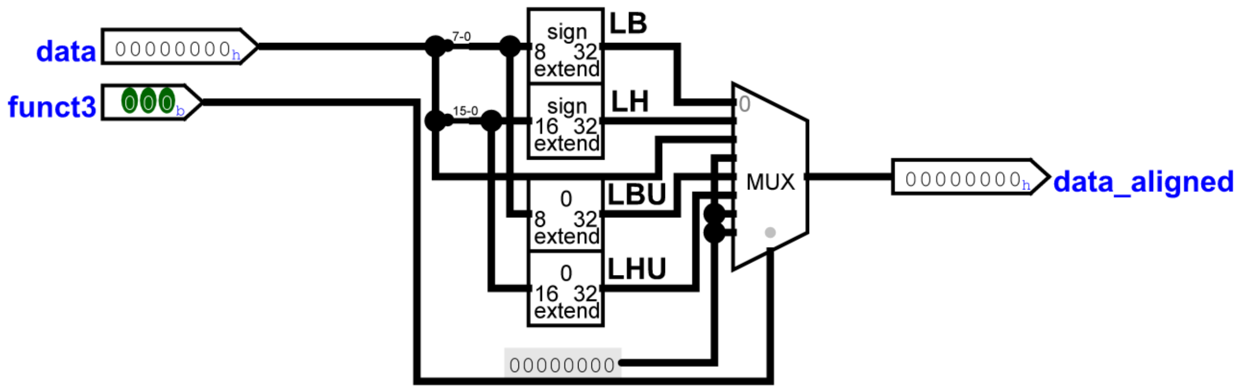


Fig. 14: Data Alignment Circuit

Interrupt Handler

Interrupts allow events external to the core to trigger code to be run. To abstract interrupt handling to the physical device itself, all the RISC-V core is required to do force a jump to a hard coded interrupt handler. To do this, it must flush the pipeline to allow any memory operations or jumps/branches to occur. It does this by using two counters. One that counts to 4 and one that counts to 5. Four clock cycles are enough for all possible memory operations to complete. A write back may still occur, but this is fine. The CPU mostly cares about the PC being changed before jumping to the IRQ vector. After the IRQ line is set, starting on the next clock cycle, the counters will increment. On the first 4 counts of the clock, a NOP is inserted into the pipeline.

On the fifth clock count, a

```
jalr x4, 0x100(zero)
```

is inserted into the pipeline. On the next clock cycle, the core returns to normal operation, now running the interrupt handler. The code at the interrupt handler vector can be specified by the programmer and loaded from memory.

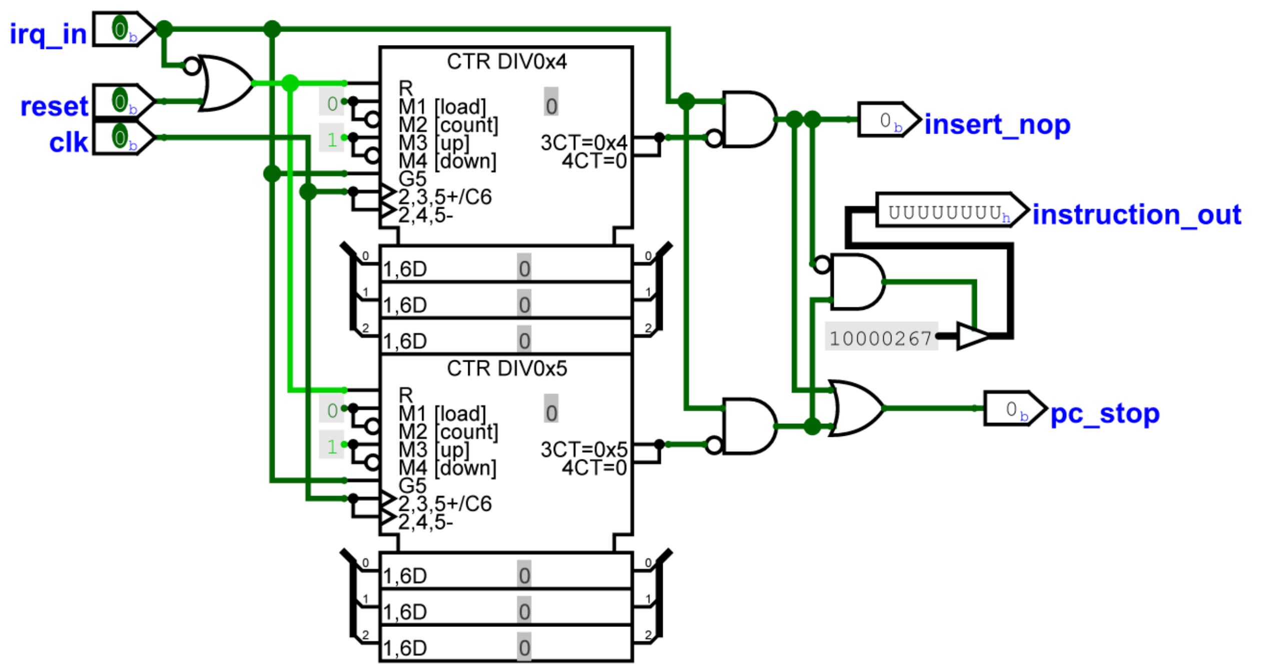


Fig. 15: CPU Interrupt Handler Diagram

Pipeline

Our RISC-V core implements a 5-stage pipeline consisting of the following stages: Fetch (IF), Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The benefits of a pipeline show greatly in performance. The pipeline allows each section of the computer to be used during a clock cycle. This raises the number of instructions per clock that we can execute greatly. It is not a linear improvement of 5 exactly, since some operations require stalling the pipeline to execute correctly, however in terms of complexity and possible inefficiencies, overall pipelining nets many benefits. As seen in Figure 16, the blue section shows all 5 stages of the pipeline being used at once during a single clock cycle. This requires the pipeline to be full, but many times it is. Modern assemblers and compilers can optimize code for a pipelined system. As seen in the upcoming sections, there are more things to handle with pipelining. This includes forwarding data from future stages, preventing hazards, and storing intermediate data between stages in registers.

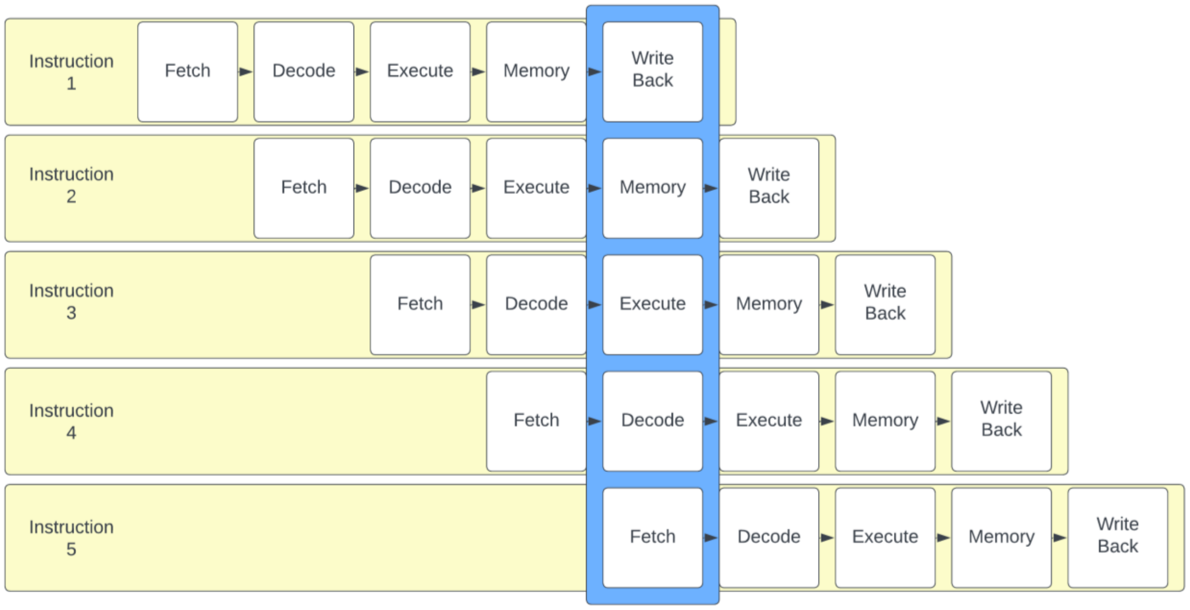


Fig. 16: Pipeline Diagram

Registers

The pipeline uses 4 intermediate registers to store the state between stages (IF/ID, ID/EX, EX/MEM, MEM/WB). These registers perform very minimal logical operations on the data passing through them. They simply allow each section of the pipeline to work on a distinct operation at once. The major exception to this is the IF/ID register. This register will inset a NOP instruction into the pipeline if the program counter is disabled. Since all values in each stage of the pipeline are calculated asynchronously, as well as before the clock cycle ends, each input to the register can be loaded into the register on the rising edge of the next clock cycle. This prevents issues where data will change on the inputs of a given stage, malforming the output going to the next stage. Each register is not only required to store the information needed for the next (adjacent) stage, but also for any future stages that might need the same data. For example, the control word is decoded from the instruction in the "Decode" stage, however the control word is needed in all future stages of the pipeline in order to properly command the parts of the computer how to operate. Even if the "Execute" stage does not need the memory related control signals, the ID/EX pipeline register as well as the EX/MEM register must still contain this information.

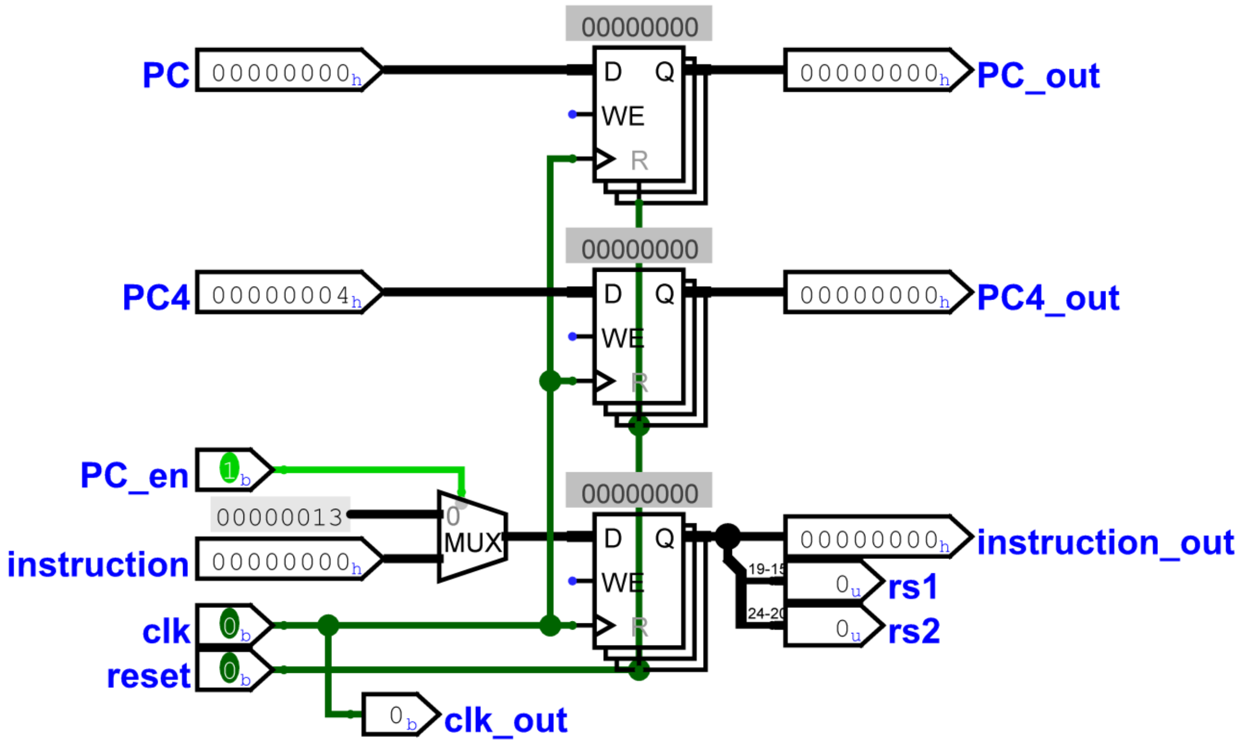


Fig. 17: IF/ID Pipeline Register

Forwarding

Forwarding is the process of taking information from stages later in the pipeline and moving them to earlier stages. The recipient stage of the forwarding operation is always the "Execute" stage. This is because this is the only stage that can modify data, the rest of the stages are for reading data, interpreting data, or writing already calculated data. We can see the forwarded data utilized in the ALU Source unit as described in the ALU section. The only data that must be forwarded is data from a specific register. Immediates are always encoded into the instruction, and memory writes will always receive data from a value stored in an existing register, mitigating the need for forwarding with this kind of data.

An example where forwarding would be needed would be two consecutive instructions that reference the same register. For example, with the following code:

```
li    t1, 123
add   t0, t1, t1
```

We are loading 123 into register t1, then squaring it and storing it into register t0. When the "add" instruction reaches the "Execute" stage, the "li" will be in the "Memory Access" stage, in which nothing will occur. The value 123 is actually stored on the next clock cycle in the "Write Back" stage. The "add" instruction has already fetched register t1 before it was even updated, how will this work? This is mitigated by forwarding the data from the "Memory Access" stage, back into the "Execute" stage. Forwarding can source its information from either the "Memory Access" stage or the "Write Back" stage, depending on what data is needed.

Forwards are calculated by viewing the current destination register indexes in each of the two stages (MEM & WB), as well as the "RF_WE" (Register File Write Enable) control signal. If any of the destination registers of the future stages match either rs1 or rs2 in the Execute stage, and the destination register is being written to, or is going to be written to, a forward will occur. This is done using comparators and AND gates. Multiplexers are used to determine what data to forward to the given register.

Forwarding is an ideal solution for mitigating data hazards, as will be discussed in the following section.

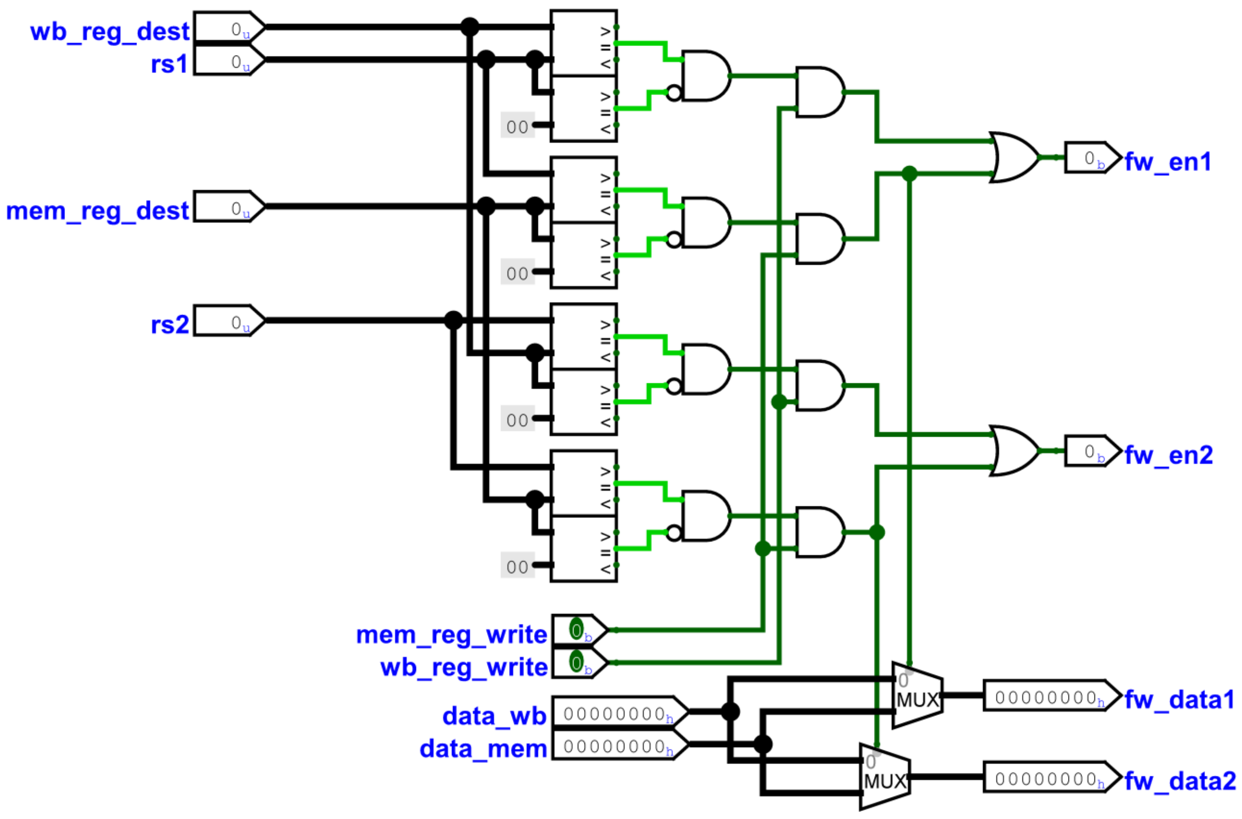


Fig. 18: Forwarding Control Unit

Hazards

In a pipelined system, hazards are anything that can cause conflicts, issues, or inconsistencies in the function of the computer. There are 3 major types of hazards to overcome: structural hazards, data hazards, and control hazards.

Structural hazards occur when two different parts of the computer need to access a single resource. These hazards are the easiest to handle in this system due to the design of a single core RISC-V architecture. There will be no instance of a resource being unavailable since each stage is discrete and contained. An exception to this may be memory accesses, since memory is external to the core, however we already have mitigated this by inserting a NOP when memory accesses (aside from instruction fetches) are required.

Data hazards occur when an instruction depends on the result of a prior instruction. As seen in the forwarding section, this occurs when two adjacent instructions are accessing the same register. In our system, the only stage at which a data hazard will cause an issue is the "Execute" stage. Because of this, all data hazards can be mitigated by the Forwarding Unit described previously.

Control hazards occur when the control flow changes during operation. This happens during jumps and branches. The reason this is an issue is because the two instructions after the jump or branch will be loaded into the "Fetch" and the "Decode" stages before the branch occurs. Since these instructions are not supposed to be executed (because a jump or branch will bring new instructions to load), we must flush the pipeline and start over, removing the two instructions we already fetched. This operation is handled in the Hazard Control unit.

The Hazard Control unit is used to calculate if the pipeline should be flushed. It takes the branch signal from the EX stage, as well as the branch signal from the MEM stage. It uses this to initiate a flush for one clock cycle (until the EX branch signal moves to the MEM stage). The hazard unit also calculates whether or not the program counter should be enabled, and if it is disabled, what address should be read from next. This enable signal is used to determine if a NOP should be inserted into the pipeline, as described in the Pipeline Registers section. The PC is disabled for any memory access in the MEM stage.

Any control hazard decreases efficiency as well as instructions per clock. There are more complicated branch prediction algorithms to determine if the pipeline should be flushed, increasing efficiency. This is not implemented due to the complexity, scope, and general lack of "readability" of the logic diagrams needed to compute this.

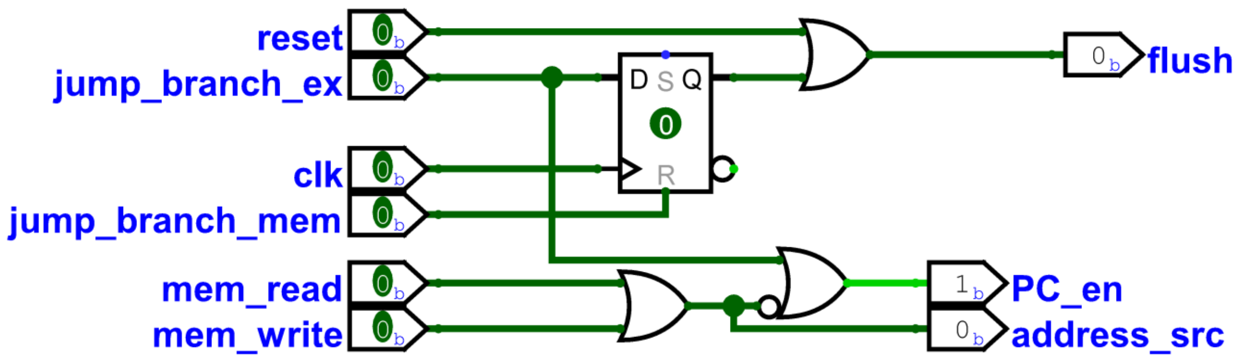


Fig. 19: Hazard Control Unit

Program Counter

The Program Counter determines which "line" of the program should be executed. More specifically, it is the memory address that must be read from next. The program counter must be able to increment by one word each clock cycle (4 bytes). It must be able to handle interrupt vectors, and it must be able to have its value changed by the execute stage.

The program counter, at its core, is just a register. Every clock cycle, both the current value, and the value + 4 are calculated. On the next clock cycle, the value + 4 will become the new value of the register. When loading a new value into the program counter register, since all program data must be aligned in RISC-V, the new address is forced to an even number by NANDing the address with 1 (forcing the LSB to be 0).

In the event of an interrupt, we must stop the program counter until we have reached the interrupt vector (the start of the interrupt handler). It is also necessary to change the PC+4 signal to equal the PC value. This is because our interrupt logic uses JALR, which sets the destination register to the value of the PC+4. However, we interrupted the instruction, it was never completed, thus when the interrupt handler returns, we must return to the interrupted instruction, not the next instruction as would be executed during a normal return. This is handled using multiplexers to select the source for the program counter, as well as the PC+4 output going to the rest of the computer.

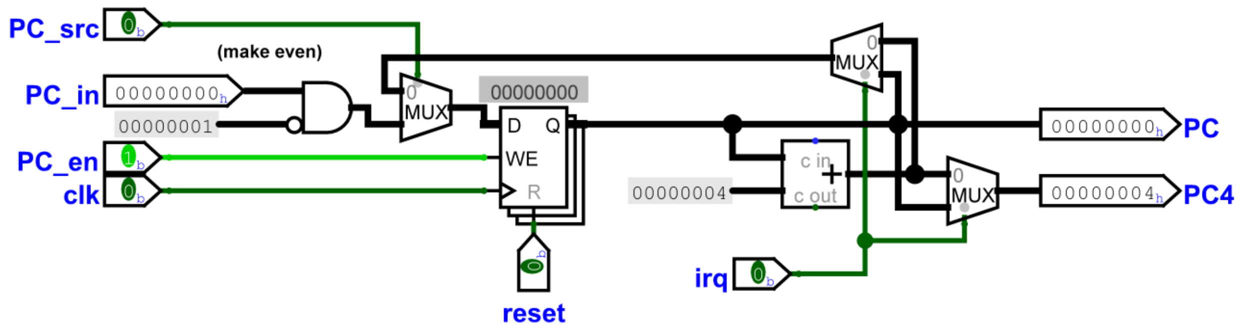


Fig. 20: Program Counter Diagram

Branch Logic

The last piece of the internal workings of the computer core is the branching logic. The branching logic is very simple. Branches are always conditional; jumps are always absolute. Branches and jumps are both enabled by the control logic; however, branches must meet given conditions before the branch is accepted.

The ALU is capable of comparisons, however there are occasions where the ALU needs to calculate an address. This is because branch addresses are relative to the program counter. For example, a BEQ instruction takes two registers and a relative position offset. This position offset must be added to the program counter, but the registers must also be compared. This is why the branching logic is separate from the ALU. The branching logic is also more efficient at detecting branches than the ALU would be. To perform a comparison, the branch logic takes in both source register values (or forwarded register values), and the ALU Source opt_data is passed to the ALU (immediate and program counter are passed to ALU). The branch logic then uses the funct3 field of the instruction to control the multiplexer selecting which type of comparison to use. These comparisons are done with comparators and OR gates. If the specific type of comparison returns a true value, and the control logic allows for a branch, the PC will be loaded with the value calculated from the ALU.

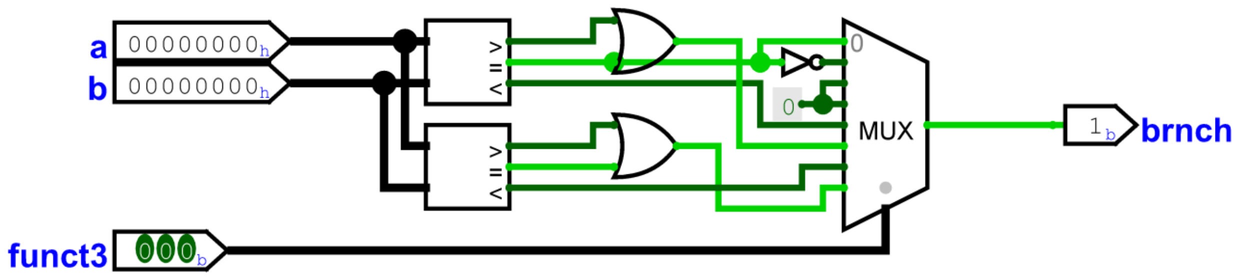


Fig. 21: Branch Control Logic

Memory

All memory in this architecture is external to the RISC-V core. All peripherals or control units that are external to the core are memory mapped. In this section, we will be talking about memory (data storage) specifically, and how it is implemented.

Memory Map

Our memory has a few features that make it dynamic based on the program being run. Our IRQ handler, heap, stack, and program data are all dynamically sized by the compiler or assembler. The heap grows to higher memory, while the stack grows towards lower memory. It is important that the programmer make sure they do not interact, although chances of this are low for the scope of the programs to be run on this architecture. All calculated VGA data is not included in this memory map since it is up to the programmer to assign their location. Normally, these would be placed on the heap. This allows the programmer to pass a pointer to the VGA memory allocation to the VGA control unit. This means that the programmer can change how VGA memory is allocated at runtime. This is especially important in different videos modes, some of which do not need as much information as others.

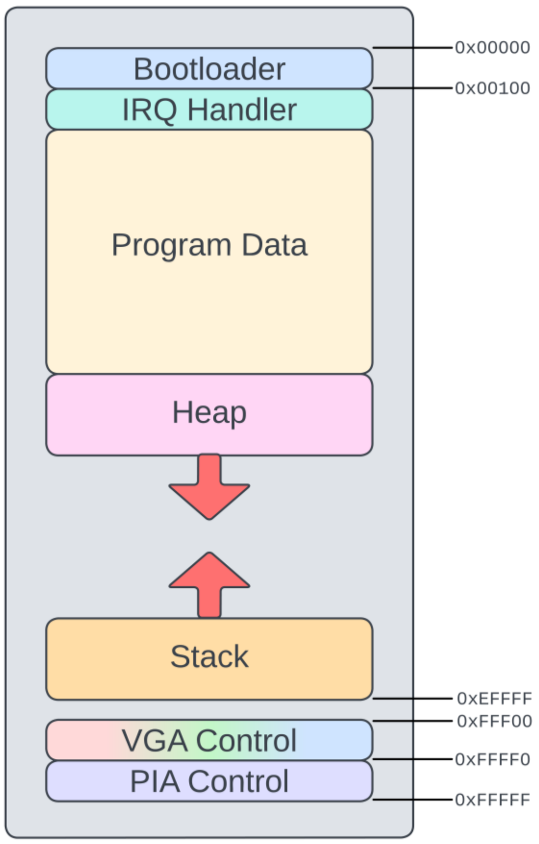


Fig. 22: Memory Map

Memory Management Unit

It is important that different parts of the computer know when to output or input their data. This is performed using a memory management unit. Unlike modern MMUs, this unit just allows control of the memory map of different hardware. The MMU has hardcoded values for the location of different control units and cannot be changed at runtime. It might be beneficial to allow these to be moved around, however most of the control units only contain a few registers for control, and these are located at the end of memory, so the added complexity of being able to remap memory outweighs the benefits of allowing it.

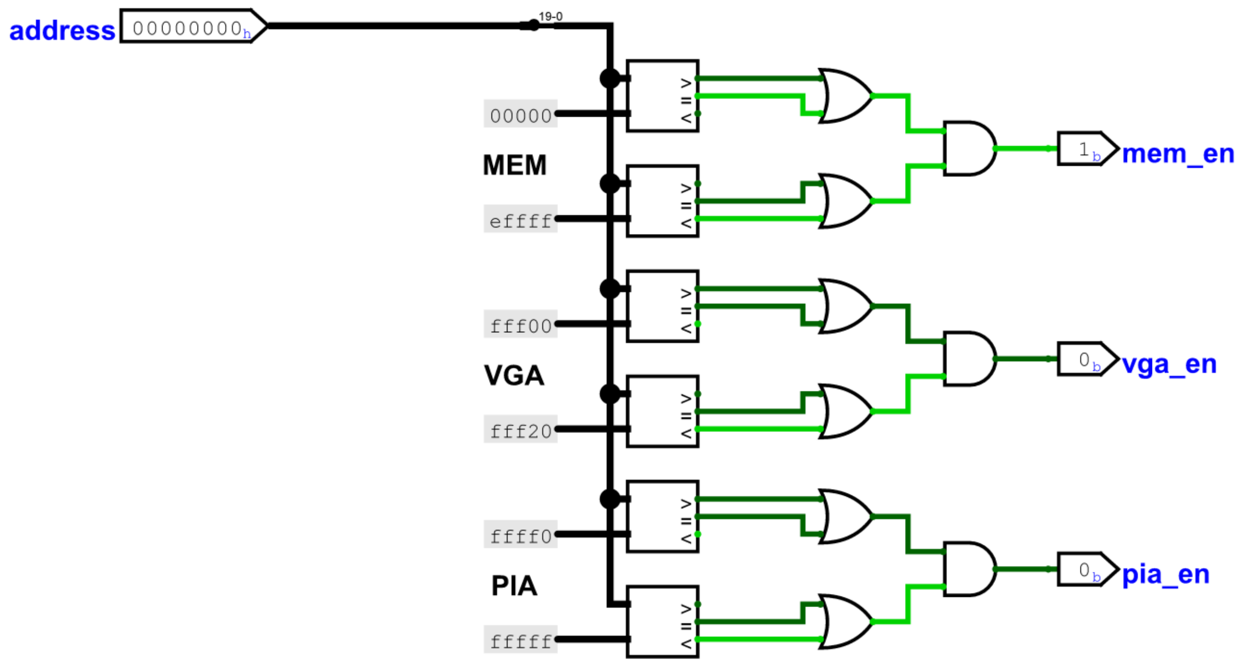


Fig. 23: MMU Diagram

RAM

Our RAM is split into two types, Dual Read RAM and Single Read RAM. Both types of RAM can only be written to at a single address at a time, however Dual RAM allows for two different addresses to be read at the same time. This is beneficial for VGA circuits to work asynchronously from the CPU.

Single RAM

Single RAM is simply just R/W memory that can read 4 bytes at a time, and write 1, 2, or 4 bytes based on the "data type" that the CPU specifies. This CPU has 1MB of memory, and its output can be buffered by the "enable" line.

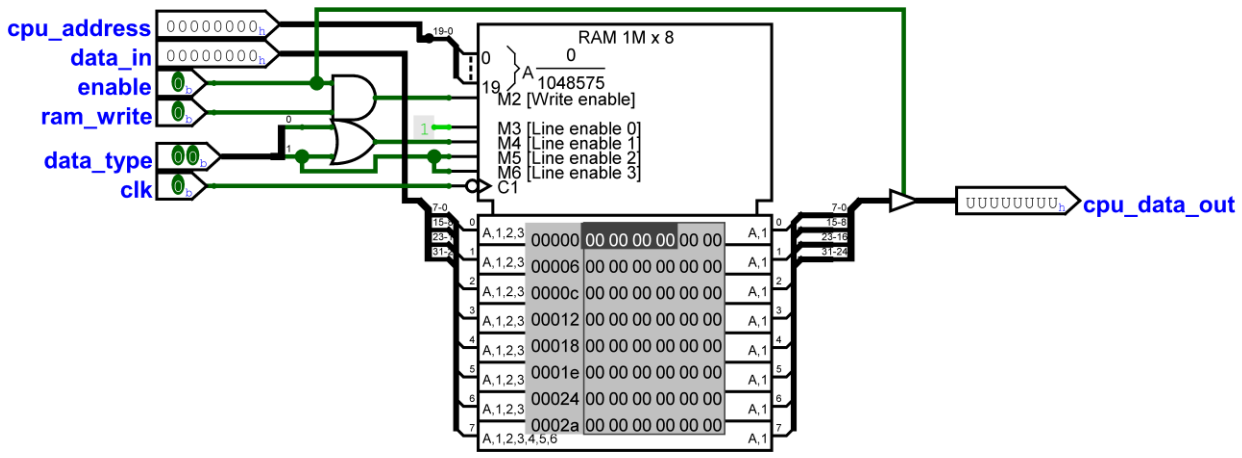


Fig. 24: Single RAM Diagram

Dual RAM

Dual RAM allows only a single write per clock cycle, but multiple asynchronous reads. This is obtained by doubling the clock speed, and on the first double cycle, writing to two identical RAM sections, then on the second cycle, reading at the other specified addresses. In Logisim, the clock speed cannot be doubled, only divided, so the dual RAM circuit generates a half-clock cycle for the rest of the computer. A dual RAM circuit diagram can be seen in the VGA Memory section.

The system clock is generated using the RAM Clock circuit, pictured below:

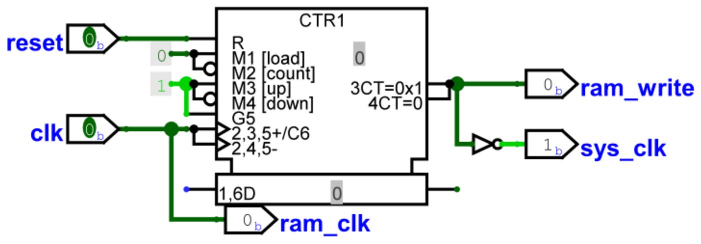


Fig. 25: RAM Clock Diagram

ROM

This computer contains no ROM, as all data is accessible and all ROM-type data is loaded by the program. This allows total flexibility of the programmer to control the contents of memory on the computer. This means, that for example, the programmer is not constrained to a default character set, as there is no character ROM to overwrite. This also aids in programming since it allows all data to be added at compilation easily, making updates to “ROMs” included in the program itself.

Peripherals

The Peripheral Interface Adapter (PIA) is used to give an abstract interface to external devices to the CPU. It is very similar to the CIA used in Commodore 64 or PIAs used in the PET, however with less features. This PIA is more suited for specific memory-controlled devices and UART control rather than a complex peripheral control. It is more focused at connecting external control units rather than acting as a driver for, say, a keyboard. It is memory mapped and has control registers to control data content, flow, and interrupt control. The PIA is the source of control for the interrupt line going to the computer. The interrupt handler uses registers on the PIA to control the status of a given interrupt. The PIA also handles UART data and other data ports. Data referenced to as a port is a controllable, unidirectional parallel data output, while UART is bidirectional.

The PIA uses decoders and multiplexers to determine which section of the PIA to utilize. Its output is stored in a register so it can be persistent until modified. It is also buffered so output can be controlled.

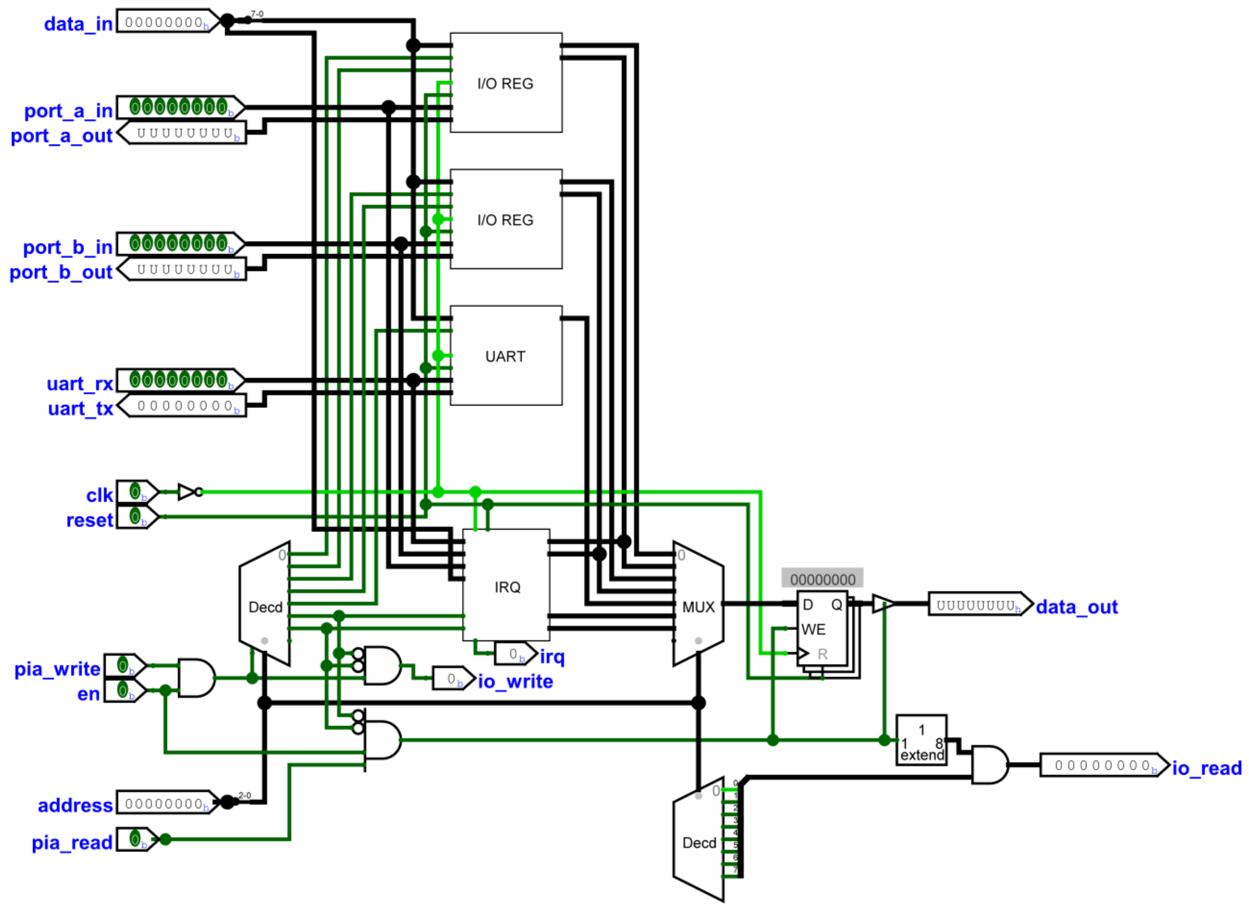


Fig. 26: Peripheral Interface Adapter Diagram

Ports

There are two 8-bit ports, A and B. These ports allow for data to unidirectionally be controlled. The direction can be set using DDRA and DDRB registers, similar to other embedded platforms like Arduino. These data direction bitmasks allow for each individual bit of the port to have its direction set.

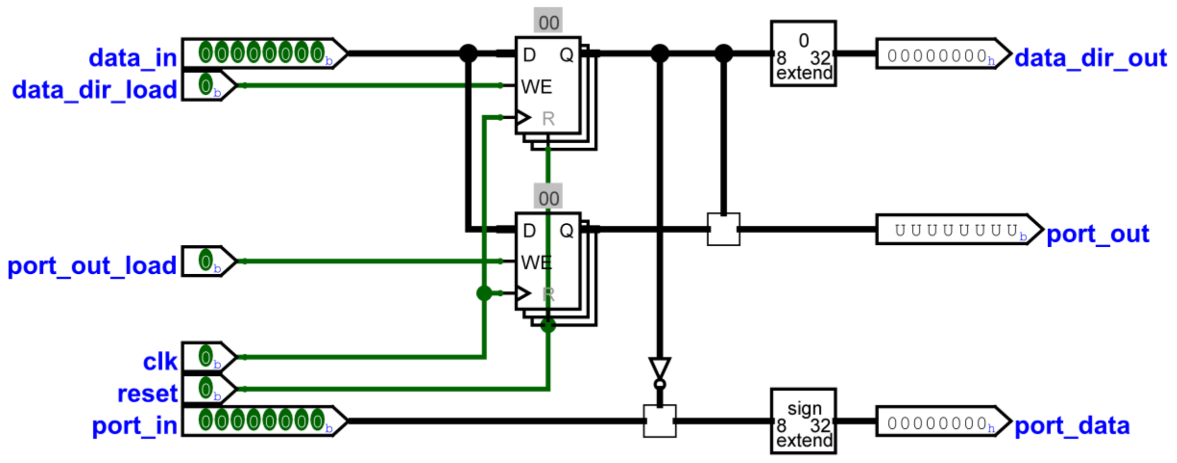


Fig. 27: I/O Register (Port) Diagram

UART

UART, or universal asynchronous receiver/transmitter is protocol to control serial devices. UART itself is a serial protocol, however with the goal of having this architecture have the ability to be physically implemented, it is key that we can interface with UART control ICs. Most of these ICs take in a parallel byte, then transmit or receive it over a serial connection. That is why our UART controller is a bidirectional, 8-bit parallel source, rather than a true serial bitstream. This UART controller is very similar to a port, except it does not have a DDRA/B. It is simply a sign-extended receiver and a register to store the transmitted data until it is overwritten.

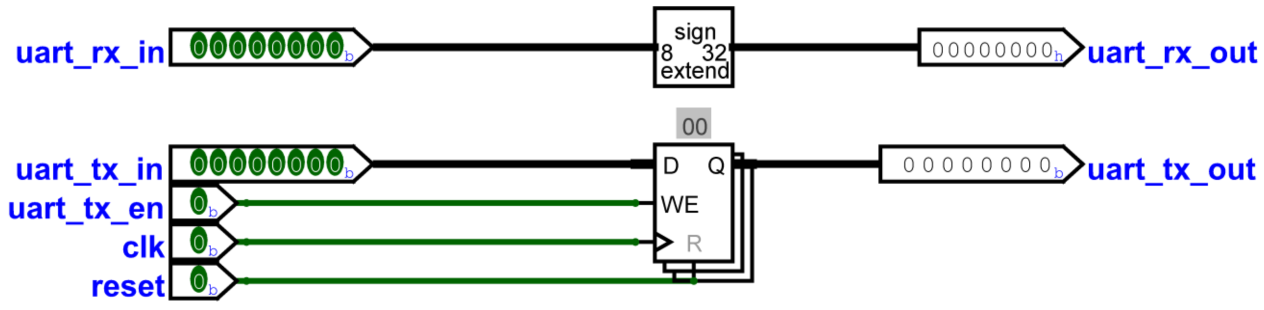


Fig. 28: UART Register Diagram

Interrupts

Interrupts are triggered asynchronously by any of the receiving pins on the PIA. It uses a latching system to latch a change anywhere during a clock cycle, and then is reset when the interrupt is handled. Our interrupt system also contains a control register to determine if interrupts are enabled or not. Our interrupt system returns an IRQ vector (different than the IRQ vector memory location) that contains a bitflag for each of the interrupts that have occurred. Once the interrupt is handled, the IRQ handler can set the handle flag to 0, allowing for the CPU IRQ pin to be cleared and normal operation to be resumed. It is not automatically resumed since it is up to the interrupt handler to return to the previous address (stored in the thread pointer).

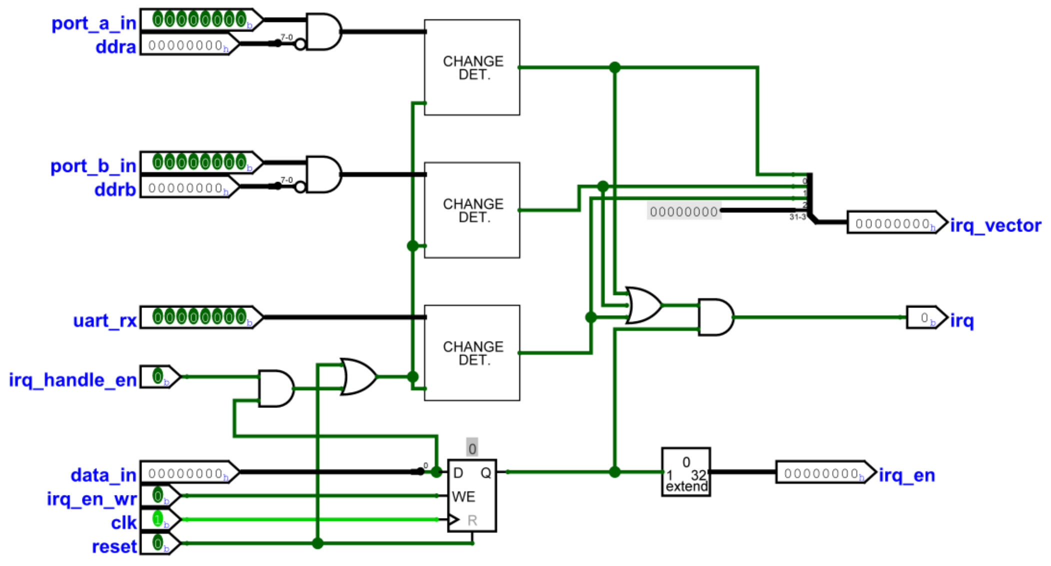


Fig. 29: Interrupt Control Diagram

Control Registers

The PIA is controlled by a set of registers that allow it to function properly. These registers are memory mapped as follows:

ADDR	NAME	R/W
0xFFFF0	PRA	R/W
0xFFFF1	DDRA	R/W
0xFFFF2	PRB	R/W
0xFFFF3	DDRB	R/W
0xFFFF4	UART	R/W
0xFFFF5	IRQ EN	R/W
0xFFFF6	IRQ HANDLE	W
0xFFFF6	IRQ VECTOR	R

Fig. 30: PIA Control Registers

VGA

VGA Graphics works on the principle of generating analog RGB values, then outputting them with adjacent sync signals. Since our graphics are not analog, the Logisim Simulator requires that we generate an X and Y position for the pixel we are trying to draw. Aside from that, our entire implementation is compatible with a VGA signal (minus the digital to analog converter (DAC)).

Our Logisim simulator only supports 64x64 24-bit graphics while our final emulation will support 320x240. We will base our timings off the 640x480@60Hz industry standard VGA timings (see Document 5). If we want to cut the resolution in half in each dimension, we must start by cutting the pixel clock in half. This allows each raster line to have half as many pixels, or for each pixel to take as much time as 2 pixels normally would. We use counters to pick which pixel to draw. Our pixel counter will count from 0 to 319. However, if our pixel clock is cut in half, the amount of time each line takes is still the same, so our line counter must still count from 0-479. We can still perform math to figure out which pixel we are on. If we shift the line counter to the right by 1 bit before using it to address our VGA memory, we will get the proper pixel while still counting lines properly to generate the sync signal.

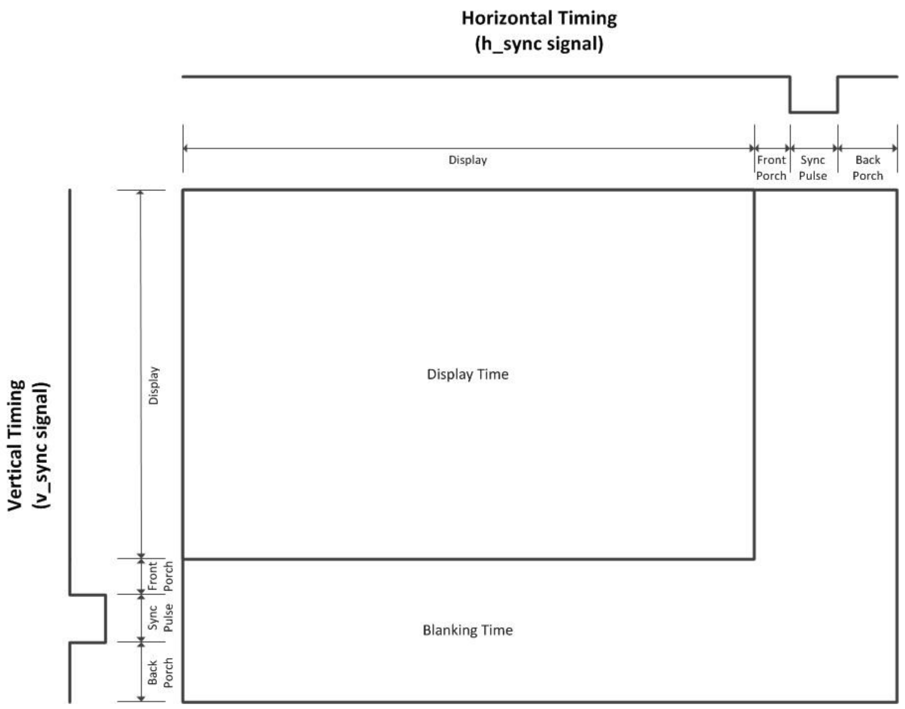


Fig. 31: VGA Timing Diagram

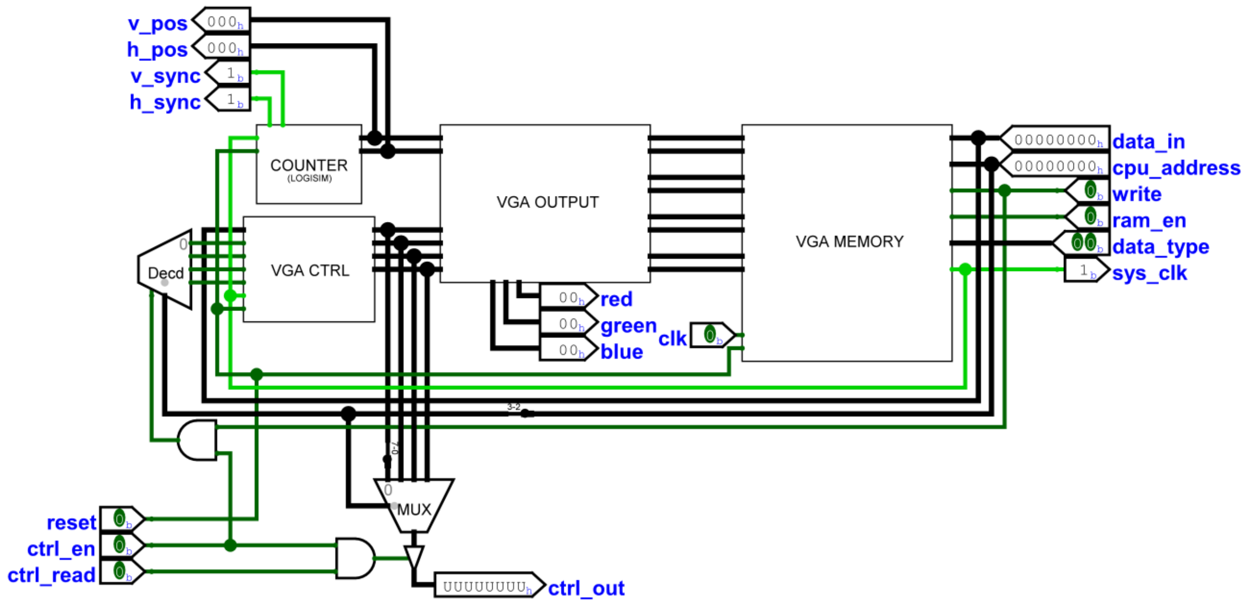


Fig. 32: VGA Logic Diagram

Counter

There is a pixel counter and a line counter. The pixel counter increments the line counter when it finishes a line, and the line counter resets itself once all lines have been completed. Comparators are used to determine where sync pulses must occur.

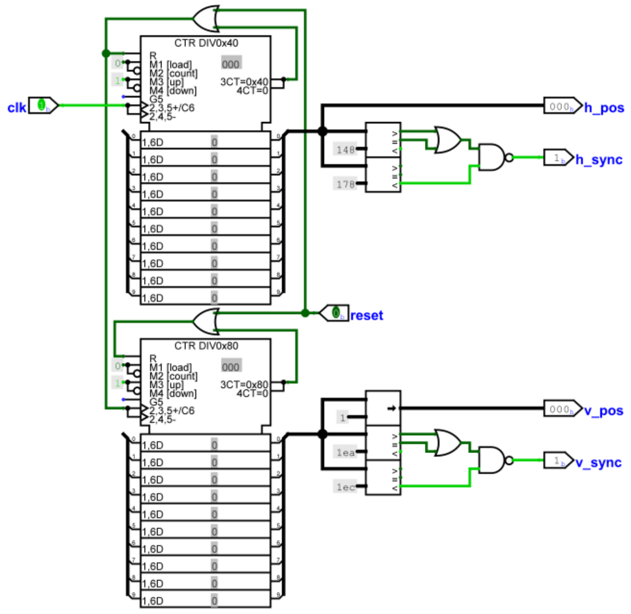


Fig. 33: VGA Counter Diagram

Modes

There are 2 VGA modes supported. One allows 24-bits per pixel, aligned on 4-byte word boundaries. Each and every pixel has 8 red, 8 green, and 8 blue bits. The pixels are consecutive in memory and can be implemented using a two-dimensional array. The second VGA mode is a character-based mode. It uses normal VGA memory to read 1-byte characters. These bytes encode an index for up to 256 programmer defined characters. Each character is 8x8 pixels. Color is set using color RAM. Each 8x8 character space is allotted a background color and a foreground color, each color being 24-bit. This can be implemented using a three-dimensional array.

Control Registers

Control registers are used to determine how the VGA controller will function. It is used to store pointers to VGA specific memory, as well as the VGA mode that is currently in use. The control registers are mapped to memory. Its mapping is as follows:

ADDR	NAME	R/W
0xFFF00	VGA Mode	R/W
0xFFF04	&VGA Mem	R/W
0xFFF08	&Color RAM	R/W
0xFFF0C	&Char ROM	R/W

Fig. 34: VGA Control Register Mapping

Memory

VGA Memory is Dual RAM that has 4 separate sections. It is seen by the CPU as a single RAM, however internally, the VGA card can asynchronously read from any location. The address is calculated based on the VGA counter and the VGA output circuitry.

System Clock

The system clock is controlled by the VGA card and generated by the RAM clock circuit in the VGA Memory Dual RAM. It is half the speed of the VGA clock in the Logisim emulation.

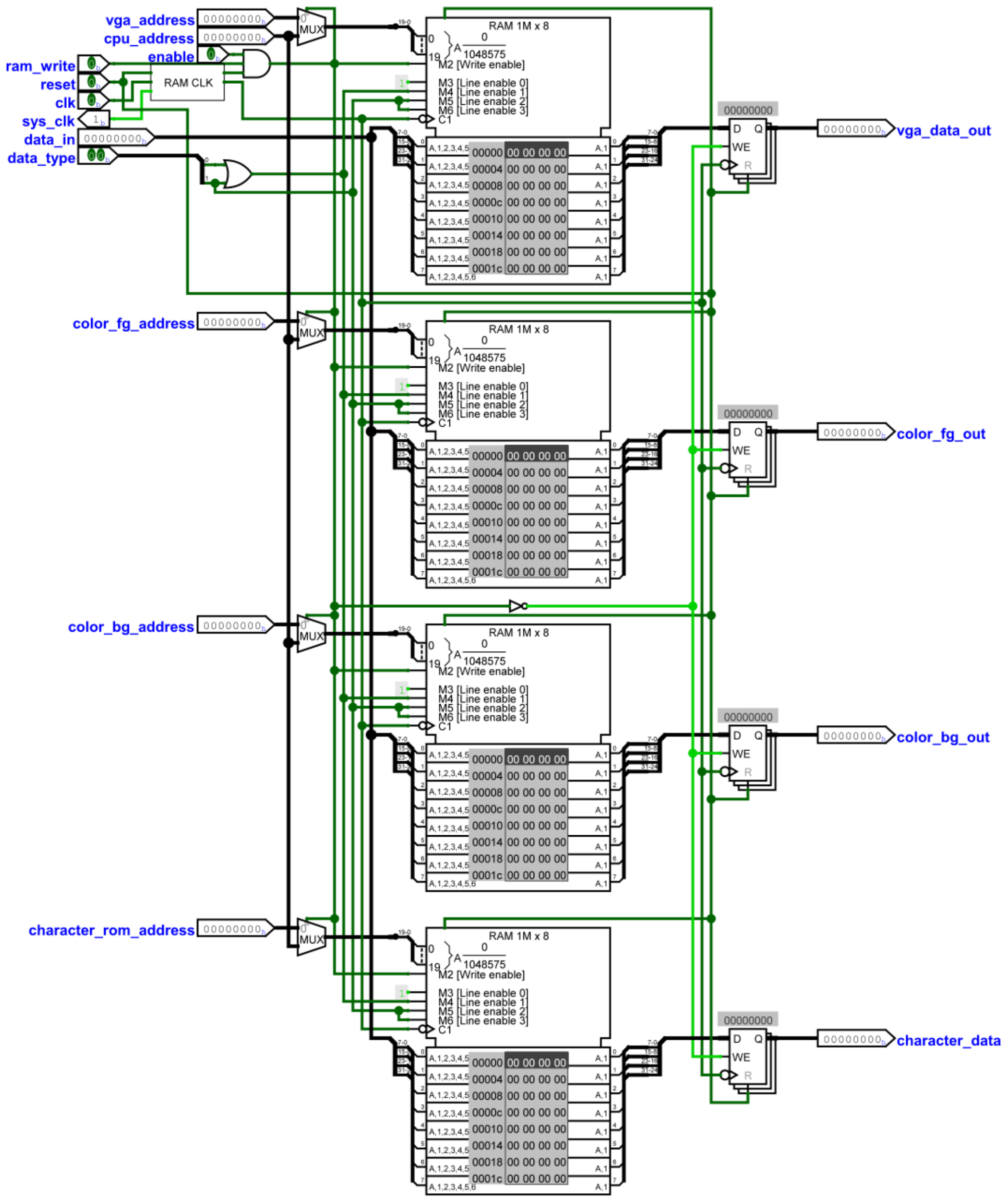
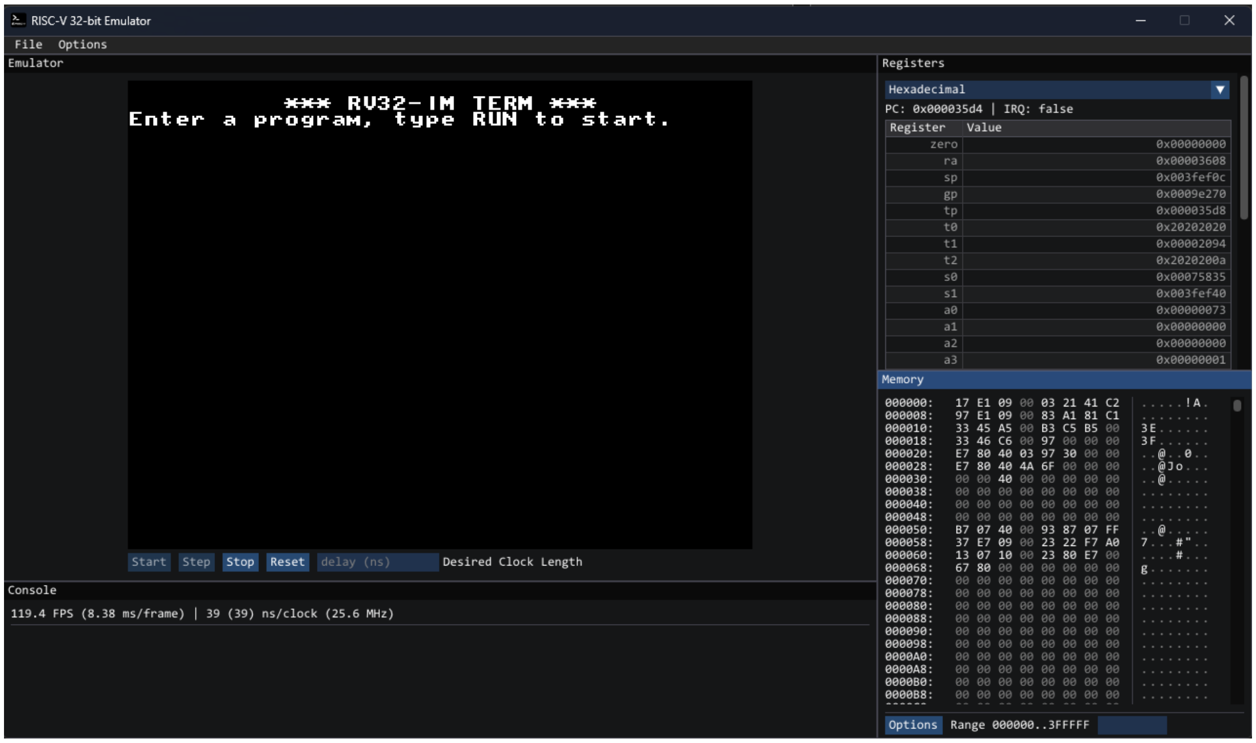


Fig. 35: VGA Memory and System Clock Origin

Emulator



```
*** RV32-IM TERM ***
Enter a program, type RUN to start.
```

```
119.4 FPS (8.38 ms/frame) | 39 (39) ns/clock (25.6 MHz)
```

PC: 0x000035d4 | IRQ: false

Register	Value
zero	0x00000000
ra	0x00003608
sp	0x003fef0c
gp	0x0009e270
tp	0x000035d8
t0	0x20202020
t1	0x00002094
t2	0x2020200a
s0	0x00075835
s1	0x003fef40
a0	0x00000073
a1	0x00000000
a2	0x00000000
a3	0x00000001

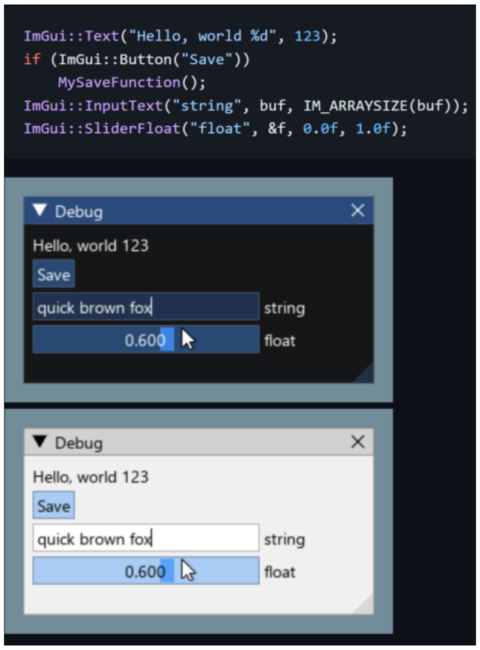
```
000000:   17 E1 09 00 03 21 41 C2   ......!A.
000008:   97 E1 09 00 83 A1 81 C1   .........
000010:   33 45 A5 00 B3 C5 B5 00   3E.......
000018:   33 46 C6 00 97 00 00 00   3F.......
000020:   E7 80 40 03 97 30 00 00   ..@..0..
000028:   E7 80 40 4A 6F 00 00 00   ..@Jo...
000030:   00 00 40 00 00 00 00 00   ..@.....
000038:   00 00 00 00 00 00 00 00   ........
000040:   00 00 00 00 00 00 00 00   ........
000048:   00 00 00 00 00 00 00 00   ........
000050:   B7 07 40 00 93 87 07 FF   ..@.....
000058:   37 E7 09 00 23 22 F7 A0   7...#"..
000060:   13 07 10 00 23 80 E7 00   ....#...
000068:   67 80 00 00 00 00 00 00   g.......
000070:   00 00 00 00 00 00 00 00   ........
000078:   00 00 00 00 00 00 00 00   ........
000080:   00 00 00 00 00 00 00 00   ........
000088:   00 00 00 00 00 00 00 00   ........
000090:   00 00 00 00 00 00 00 00   ........
000098:   00 00 00 00 00 00 00 00   ........
0000A0:   00 00 00 00 00 00 00 00   ........
0000A8:   00 00 00 00 00 00 00 00   ........
0000B0:   00 00 00 00 00 00 00 00   ........
0000B8:   00 00 00 00 00 00 00 00   ........
```

Range 000000..3FFFFF

Fig. 36: Emulator Main Screen

The Emulator was originally an event-based program that was tightly coupled to the implementation of the Logic Diagrams. As time progressed, in the name of efficiency, the application became a threaded sequential program as event-based programming had more overhead. We were able to go from 3 microseconds per clock to 35 nanoseconds per clock, almost an 8500% improvement. With the implementation of threading/thread pools, the system clock is no longer tied to the VGA clock. Other improvements in the software emulator include a 2-bit dynamic branch predictor, as well as moving the interrupt hardware to the decode stage, as the instructions are static, meaning no need to fetch. This also prevents interrupts from interfering with the branch predictor algorithm.

The Emulator itself was written with SDL3, SDLRenderer, and ImGui. SDL3 is a frontend framework/window manager for SDLRenderer, while SDLRenderer is like a simplified interface to OpenGL. ImGui is a graphics framework for abstracting UI/UX components regardless of the backend, allowing for quick development of a UX.



```
ImGui::Text("Hello, world %d", 123);
if (ImGui::Button("Save"))
    MySaveFunction();
ImGui::InputText("string", buf, IM_ARRAYSIZE(buf));
ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
```

Fig. 37: ImGui Demo (from ImGui repository)

ImGui allows taking very simple code to generate windows in a real time (immediate mode) manner. This allows us to display the status of the RV32IM core in real time. We also have to push the frame buffer to a GPU texture in order to render the contents of the emulator efficiently. ImGui and SDL3 make this very easy to do.

The emulator also implements a similar UART console to that of Logisim, as well as allowing the size, and map of memory to be dynamically specified by either the programmer or compiler.

Software Library

The software library for this architecture allows for common system operations to be called. It allows for C and C++ to be compiled to the bare metal architecture. Most of the C++20 standard template library is functional on this architecture.

Kernel

Our “kernel” is just a collection of common software tasks for interacting with the hardware. This includes tasks such as writing to the screen, handling interrupts, or handling peripherals.

Compiler

All code for this architecture is compiled with a gcc-based RISC-V cross compiler. The compiler uses a custom CMake toolchain that was specifically built for this architecture. A custom linker script is used to allow memory to be allocated by the linker in a way that matches our memory map. This custom linker script allows for the stack, the heap, and all other labels to be implemented by the programmer how they see fit.

CMake

A CMake custom toolchain, as well as custom post-process scripts allows for custom targets to easily be compiled without having to manually interact with the toolchain.

Project Plan Summary

The final goal of this project is to learn more about the inner workings of a computer, but to also provide a tool that easily shows how digital logic can turn into an architecture that can play games, compile code, and perform anything else that a computer must.

The current plan is to implement all of this in Logisim first, then to implement the emulator. Once the emulator is working, a codebase can be created for the architecture. A sort of "kernel" or software library can be used to simplify menial tasks. This "kernel" will be some sort of header file that will be included in any compiled code. It will be the programmer's responsibility to include the kernel if they choose to use it. There will be no software stored in any sort of ROM. After testing and the software library working, it will be time to start moving towards converting the emulator to something that can be run on an FPGA.

It is critical to the success of this project that the timeline is manage efferently, and in a way that allows time for iterative changes. This is why our phases will be executed in an almost Agile like methodology.

Timeline

Our timeline is shown using a Gantt chart. It is necessity that this architecture is implemented in the following fashion to allow for proper testing.

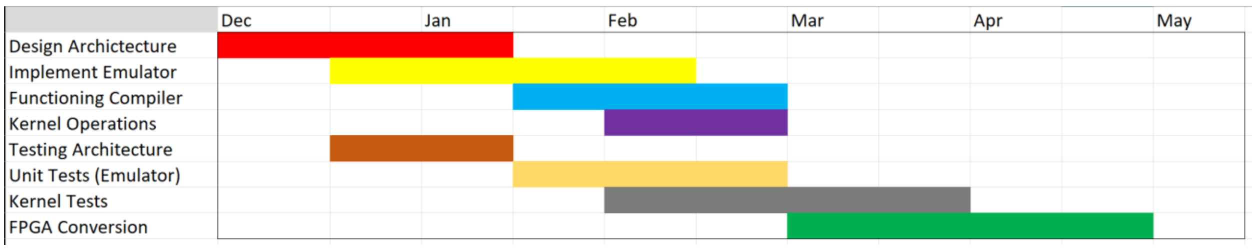


Fig. 38: Project Timeline Gannt Chart

Development Pattern

This project is very iterative in the way it is built. It can be split into many functional steps; however, these functional steps must be completed in a certain order since they build on each other so heavily. For example, the emulator cannot be started until the architecture design and logic diagrams have been sorted out. Compiler operations cannot occur until there is an architecture to test on, and so forth.

To get to an MVP, there must be a functional emulator that can at least run some machine code for the designed architecture, as well as show the user what is going on. The final goal is to surpass this with a full-fledged, cycle-accurate emulation of how this architecture was designed to be implemented. The MVP for each phase will be listed further.

Functional Steps (Phases)

According to the Gantt chart, there are a few overlapping phases that are required to complete this project. If these phases are followed, even slightly, we will hit our MVP.

Logic Design

The first part of our MVP is to design the actual architecture. This is done using Logisim Evolution. This is a very iterative process. Many small building blocks must come together in order to complete the task at hand. An MVP would include a RISC-V core that can run the RV32-I instruction set, however this phase's main goal is to complete implement the computer, including memory, peripherals, and graphics.

Individual Components

The individual components that make up the logic diagram can be abstracted in a way that allows the person making logic diagrams to be able to combine logic in a readable manner. For example, the ALU is made up of smaller blocks, which come together to form the larger ALU, which will be implemented in the RISC-V core, which then will be implemented in a top-level architecture implementation.

Top-Level Implementation

The "top-level" implementation is not required for the MVP, although this will be the overall architecture logic simulation when it is combined. A CPU core will be able to interface with a VGA controller, peripherals, and external memory.

Emulator

The emulator phase MVP will be a simple emulator that can implement the basic instruction set defined for the logic diagram. The final goal will be for it to be a full, cycle accurate emulation of the designed architecture written in C++.

Graphics

The graphics section of the emulator will hit its MVP threshold once anything from the core can actually be displayed to the user from the internal logic implementation. The final goal will be to display a 24-bit VGA buffer "emulation".

Logic Implementation

The logic implementation for the emulator will have an MVP of simply being able to implement the core instruction set. No output or peripherals or separate control units are needed for MVP, although the ability to be able to fully implement the logic from the diagram's top-level implementation would be the final goal.

Software Library

The software library is critical in testing the architecture, both in the emulator and the Logisim diagram. It's MVP will be a small set of assembly instructions to speed up programming for the programmer. The final goal will be a C or C++ library that the programmer can include in his code to be added in at compilation. The software library phase consists of compiler setup and programming system functions.

Compiler Setup

The MVP for the compiler setup is for the ability to assemble assembly code for this architecture. The actual end goal is for the C++ standard template library to be able to be compiled with this architecture as a target, with the custom system functions as an addition.

System Functions

The MVP for the system functions section of the software library phase is for there to be a successful collection of assembly code that can be used to speed up and simplify programming on this architecture. The final goal would be for an entire "kernel" to allow the programmer to easily interface with the hardware functions of the architecture.

Source Control

Source control is a critical point of being able to successfully manage this project. It is best used when writing the emulator and will be utilized as such. Git and GitHub will be used to track changes and phases over time.

Using the Software

This section is specifically for documenting how the program is run, compiled, and tested both in the backend, and by the user.

Compilation

Compilation of the emulator will be performed directly inside of the Visual Studio solution. Logic diagram simulations will be run inside of Logisim Evolution 3.9.0. Compilation of the software library will be performed in WSL Ubuntu using the riscv-gnu-toolchain. The toolchain will be compiled for rv32im with abi ilp32 (software float). The toolchain will be implemented using a custom CMake toolchain.

Software Use

The emulator is the only section meant to have a real user interface (aside from the Logisim built in interface). It will be used by loading in a binary file containing the desired contents of memory, and then will function as a normal emulator should. This means it will grab keyboard inputs and display the contents of the video buffer to the screen.

Testing

In terms of testing, the emulator will be the basis for the entire architecture’s viability. The emulator will be end-to-end tested using white box testing. Every single operation of the interface of this architecture will be tested using GoogleTest. It uses simple C++ macros to generate quick and easy large-scale testing.

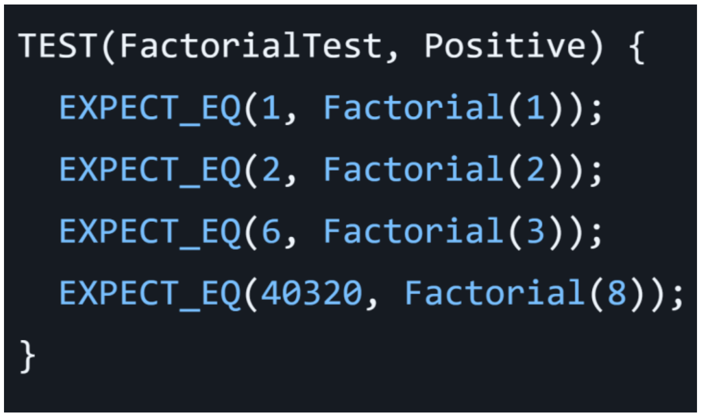
Every instruction, code path, data path, and compiler operation should be tested in this project. Any testing of the ISA will be very rudementry, since it has been tried and true over the past few years. RISC-V might have room to grow via custom CSRs, however there needs to be a working implemention first, which is what this project aims to do. All architecture logic testing will be done in Logisim Evolution to make sure that the logic diagrams to be implemented match the ISA spec.

Testing Methodology

Since white-box testing will be used with GoogleTest, tests will be very simple. We will simply as the emulator to process an input and compare its output to an assumed value. Since the ISA is already specified, it is very simple to calculate what output we should get from a specific input.

Implementation

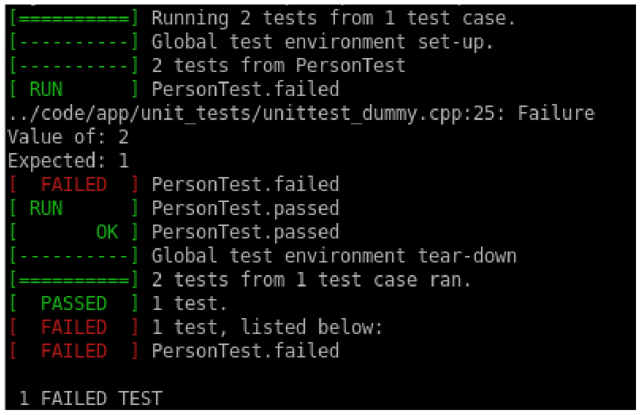
The GoogleTest implementation will be a separate project within the same Visual Studio solution. It will be a set of macros that will generate an executable that will run the tests. Google test will take something like the following:



```
TEST(FactorialTest, Positive) {
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(2, Factorial(2));
  EXPECT_EQ(6, Factorial(3));
  EXPECT_EQ(40320, Factorial(8));
}
```

Fig. 39: GoogleTest Sample Input

And it will generate a test console executable that will give an output that looks like the following:



```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from PersonTest
[ RUN      ] PersonTest.failed
../code/app/unit_tests/unittest_dummy.cpp:25: Failure
Value of: 2
Expected: 1
[  FAILED  ] PersonTest.failed
[ RUN      ] PersonTest.passed
[       OK ] PersonTest.passed
[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran.
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] PersonTest.failed

 1 FAILED TEST
```

Fig. 40: GoogleTest Sample Output

Another benefit of GoogleTest is that it allows for “mock” objects. What this allows us to do is test only a certain component of the architecture. If we are using a standard library to draw a bitmap, we do not want to test the validity of that library, but how our code interacts with that library. We then create a mock bitmap library that will have a custom interface with our test that will only test our code, not the library’s code. This cannot prevent misuse of a library, but it can perform more specific white-box testing.

Conclusion

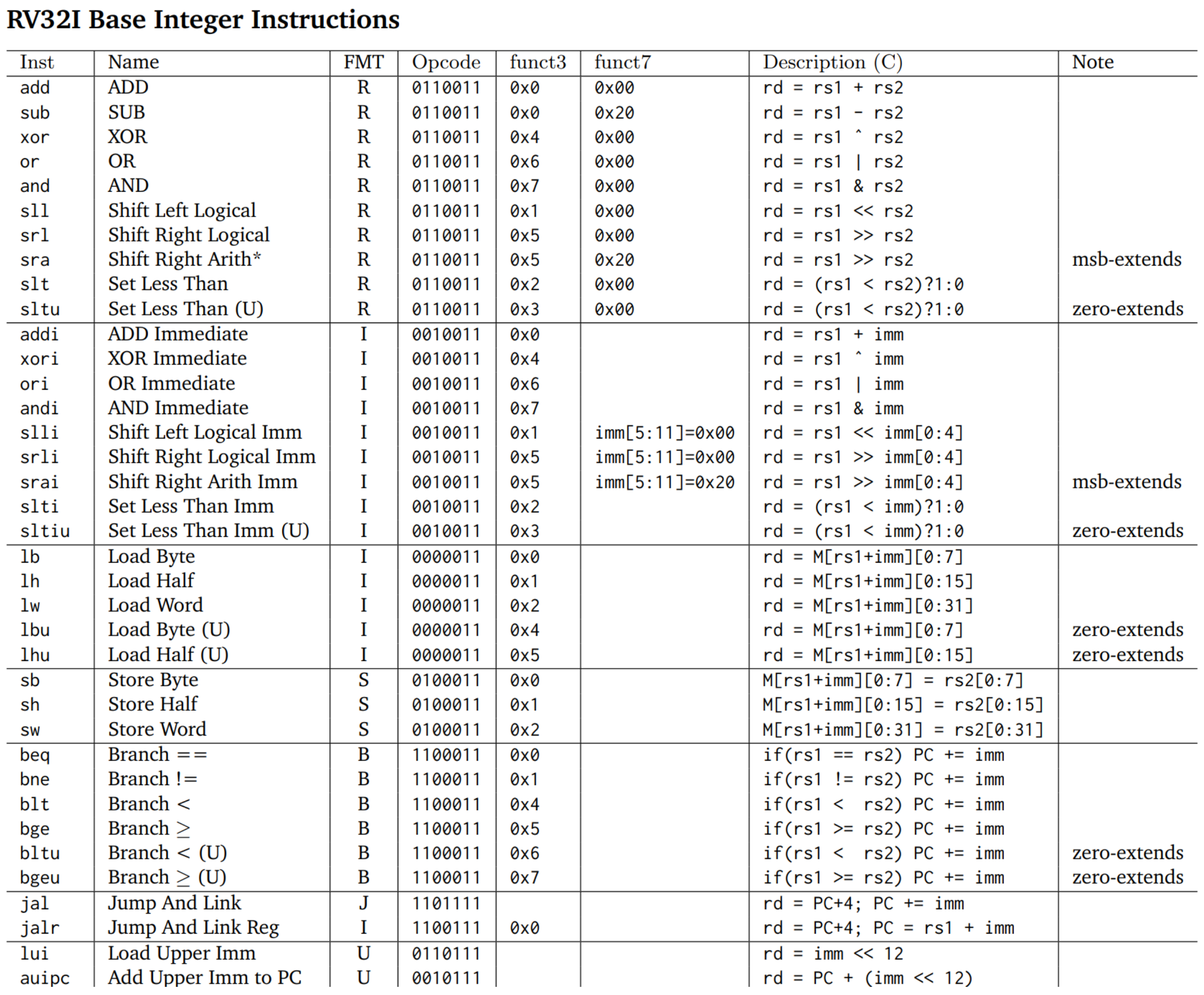
All in all, this architecture is not a groundbreaking discovery in the world of computers, but it is a very good dive into the inner workings of RISC-V, allowing for others to gain a deeper understanding of ISA. This may lead to new discoveries or developments in the future, as well as an extensible platform to test these changes on. Not only can this project be learned from in terms of understanding the ISA, but it is a very good tool to understand how lower-level C++ applications can be written to perform very custom functions.

Future Ideas

In the future, it would be ideal to implement this architecture into an FPGA, although the scope of this project does not allow for that. It would also be cool to test the limitations of the architecture in terms of clock speed, additional co-processors, custom CSR extensions, custom control units, etc.

Additional Documents

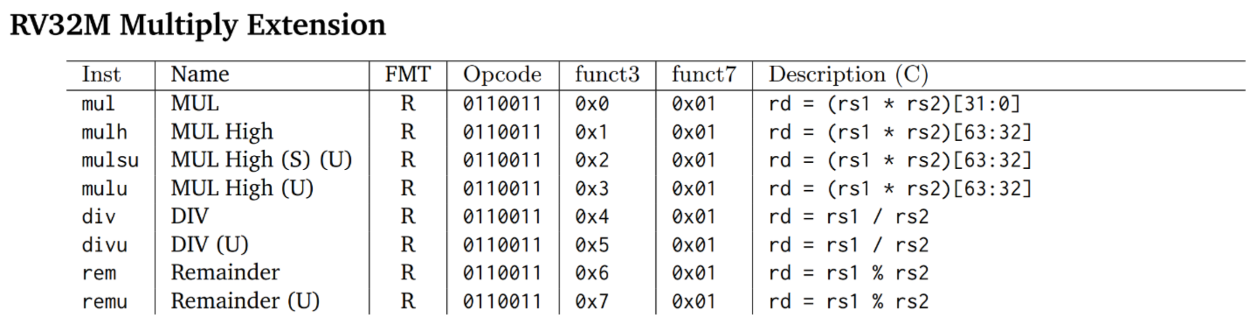
This section contains all useful references that were used during the process of creating this project.



RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	

Doc. 1: Integer Instruction Set (Simon Fraser University)



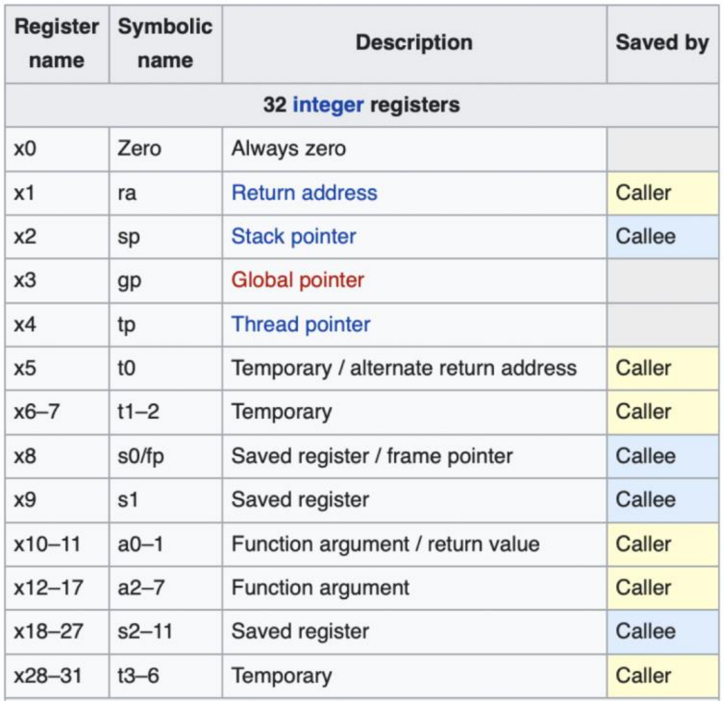
RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
`mul`	MUL	R	0110011	0x0	0x01	`rd = (rs1 * rs2)[31:0]`
`mulh`	MUL High	R	0110011	0x1	0x01	`rd = (rs1 * rs2)[63:32]`
`mulsu`	MUL High (S) (U)	R	0110011	0x2	0x01	`rd = (rs1 * rs2)[63:32]`
`mulu`	MUL High (U)	R	0110011	0x3	0x01	`rd = (rs1 * rs2)[63:32]`
`div`	DIV	R	0110011	0x4	0x01	`rd = rs1 / rs2`
`divu`	DIV (U)	R	0110011	0x5	0x01	`rd = rs1 / rs2`
`rem`	Remainder	R	0110011	0x6	0x01	`rd = rs1 % rs2`
`remu`	Remainder (U)	R	0110011	0x7	0x01	`rd = rs1 % rs2`

Doc. 2: Multiplication Instruction Set (Simon Fraser University)

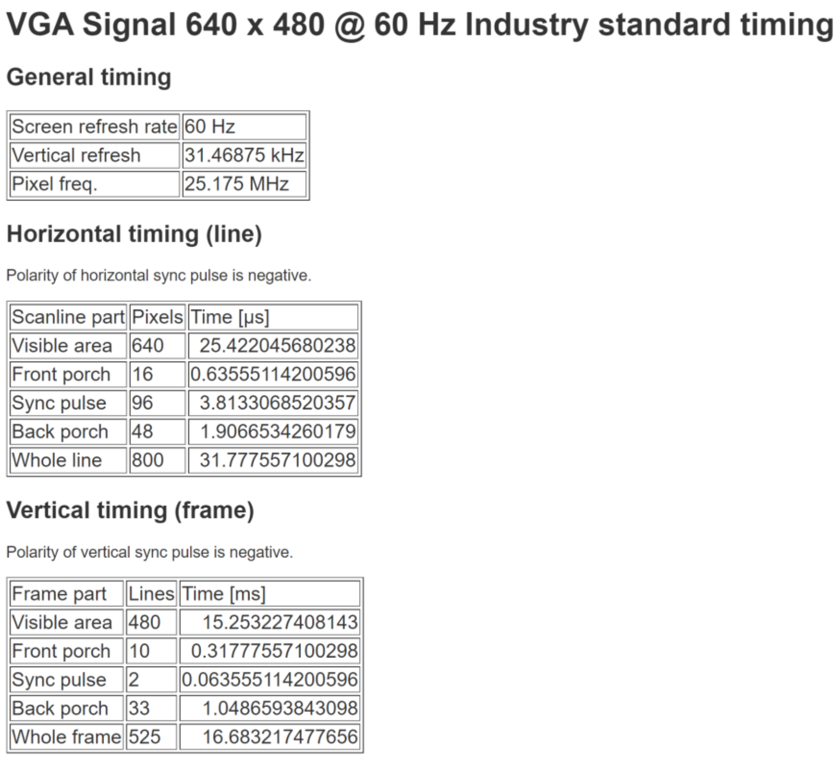
Splitter, Pin, Probe, Tunnel, Pull Resistor, Clock, Power on Reset, Constant, Power/Ground, Do not connect, Transistor, Transmission Gate, Bit Extender, NOT Gate, Buffer, AND/OR/NAND/NOR Gate, XOR/XNOR/Odd Parity/Even Parity Gate, Controlled Buffer/Inverter, Programmable logic array, Multiplexer, Demultiplexer, Decoder, Priority Encoder, Bit Selector, Adder, Subtractor, Multiplier, Divider, Negator, Comparator, Shifter, Bit Adder, Bit Finder, D/T/J-K/S-R Flip-Flop, Register, Counter, Shift Register, Random, RAM, ROM, Button, Dip Switch, Joystick, Keyboard, LED, RGB LED, 7-Segment Display, Hex Digit Display, LED Matrix, TTY, Port I/O, Reptar Local Bus, RGB Video, Switch, Buzzer, Slider, Digital oscilloscope, PLA, VGA screen

Doc. 3: Logisim Available Library



Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

Doc. 4: Register Convention (Wikipedia)



VGA Signal 640 x 480 @ 60 Hz Industry standard timing

General timing

Screen refresh rate	60 Hz
Vertical refresh	31.46875 kHz
Pixel freq.	25.175 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Vertical timing (frame)

Polarity of vertical sync pulse is negative.

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.31777557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.0486593843098
Whole frame	525	16.683217477656

Doc. 5: VGA Timings (tinyvga.com)

Interesting Algorithms used, but not limited to:

- Efficient Moving Average
- Branch Prediction
- GPU Texture Calculations
- VGA Decoding/Encoding
- Keyboard Matrix Decoding
- Stdlib.h style syscalls
- Dynamic Memory Mgmt/Heap Allocation
- Linker Scripting
- Thread pools, concurrency/ synchronization.
- Input Buffering/Streams
- Serial Communication Protocols