32-Bit RISC-V CPU Project

Sean A. U. Grant, *seangrant39800@gmail.com*

August 1, 2025

1 Introduction

1.1 Motivation

In the fall term of my junior year at OSU, I took my first 400-level electrical engineering class, ECE 471 Energy-efficient VLSI Design. This class introduced me to the basic ideas of VLSI and RTL design, timing, power, and industry tools, i.e., Cadence. Before taking this class, I knew that I was passionate about electrical engineering, but I had yet to determine which industry under the electrical engineering umbrella I wanted to pursue. After completing this class, I knew that I wanted to pursue a career in computer architecture and RTL design. The ability to break down something as complex as a computer into individual logic gates and transistors was both fascinating and extremely satisfying. The next term I took my second RTL class, based around synthesizing logic into logic gates and using place and route tools to create a physical design. Again, I found every part of the design process completely captivating. During this course I decided that I wanted to pursue a more complex RTL design as a personal project. I felt that this would be the best way to dive into computer architecture and get the hands-on experience that I wanted. Knowing that I wanted to get my hands dirty and develop a deep understanding of how computer architecture as well as digital logic design, I settled on designing a simple CPU, in the style of the microcontrollers I had used in past projects. Although this design is extremely simple when compared to modern CPUs, my goal was to learn the RTL design process and foster my interest in computer architecture.

1.2 Initial Goals

Before starting this project, I laid out rough goals for my design. These were based on my limited knowledge of computer architecture; however, they provide good insight into where I started. My CPU was to be an 8-bit design with an architecture based on a diagram from an assembly class (Figure 1).

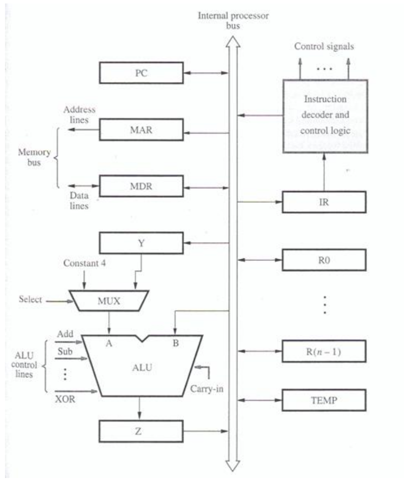


Figure 1: Initial CPU Architecture

In this design, all data is transferred to and from all elements using a single data bus. This design has some obvious issues and limitations, based around how it was designed to teach assembly language; however, at the time it seemed like a robust design to me.

The flaws in my design became apparent after I completed my Arithmetic Logic Unit (ALU) and began to address the interconnect and control logic. It was at this point that I realized my knowledge of how computer architecture translates machine code into executable operations and outputs was very limited. Extensive research led to new design goals:

1. 32-bit word size
2. RISC-V RV32I compatible
3. In-order pipelined datapath

A 32-bit word size and RISC-V compatibility were chosen for several reasons. The idea of my design supporting an Instruction Set Architecture (ISA) was particularly compelling, given that the RISC-V ISA is open-source and widely adopted in industry designs. Additionally, because RISC-V is well supported, using the RISC-V ISA in my CPU would allow me to write, compile, and then run assembly programs on my CPU. The decision to move from an 8-bit to a 32-bit architecture was necessary to support the RISC-V ISA. Finally, I decided on an in-order pipeline for my Datapath as it presented a more significant challenge when compared to a single-cycle or multi-cycle design. Starting with an in-order pipeline will also allow me to make future design improvements, such as implementing an out-of-order pipeline. With these design choices, I felt confident that my CPU would serve as a strong entry-level design, incorporating elements of real CPU architecture (even if outdated), and providing valuable insight into microarchitecture and RTL design.

The RISC-V instruction set was chosen for its streamlined instruction count, which nonetheless provides comprehensive coverage of all instruction types. Additionally, when I expand my CPU in the future, I can easily add new instructions, such as multiplication or division, without needing to rework my entire design. I chose not to include the U-type instructions as they are not used as readily as other instruction types; however, this is an area targeted for future improvement.

Table 1: RISC-V Instruction Set Summary

Inst	Name	FMT	Opcode	funct3	funct7
add	ADD	R	0110011	0x0	0x00
sub	SUB	R	0110011	0x0	0x20
xor	XOR	R	0110011	0x4	0x00
or	OR	R	0110011	0x6	0x00
and	AND	R	0110011	0x7	0x00
sll	Shift Left Logical	R	0110011	0x1	0x00
srl	Shift Right Logical	R	0110011	0x5	0x00
sra	Shift Right Arith*	R	0110011	0x5	0x20
slt	Set Less Than	R	0110011	0x2	0x00
sltu	Set Less Than (U)	R	0110011	0x3	0x00
addi	ADD Immediate	I	0010011	0x0	
xori	XOR Immediate	I	0010011	0x4	
ori	OR Immediate	I	0010011	0x6	
andi	AND Immediate	I	0010011	0x7	
slli	Shift Left Logical Imm	I	0010011	0x1	
srli	Shift Right Logical Imm	I	0010011	0x5	
srai	Shift Right Arith Imm	I	0010011	0x5	
slti	Set Less Than Imm	I	0010011	0x2	
sltiu	Set Less Than Imm (U)	I	0010011	0x3	
lb	Load Byte	I	0000011	0x0	
lh	Load Half	I	0000011	0x1	
lw	Load Word	I	0000011	0x2	
lbu	Load Byte (U)	I	0000011	0x4	
lhu	Load Half (U)	I	0000011	0x5	
sb	Store Byte	S	0100011	0x0	
sh	Store Half	S	0100011	0x1	
sw	Store Word	S	0100011	0x2	
beq	Branch ==	B	1100011	0x0	
bne	Branch !=	B	1100011	0x1	
blt	Branch $<$	B	1100011	0x4	
bge	Branch $\geq$	B	1100011	0x5	
bltu	Branch $<$ (U)	B	1100011	0x6	
bgeu	Branch $\geq$ (U)	B	1100011	0x7	
jal	Jump And Link	J	1101111		
jalr	Jump And Link Reg	I	1100111	0x0	

2 CPU Design

2.1 Datapath

The Datapath for my CPU is based on a traditional 5-stage pipeline. The first stage, Instruction Fetch, handles the program counter and retrieves the current instruction word. The next stage, Instruction Decode, takes the current instruction and generates the appropriate control signals for its execution. This stage also handles the Register File and immediate decoding based on the instruction. Next, the Execute stage performs the desired operation on the specified data. Then, the Memory stage handles any necessary data writes to the Data Memory. Finally, the Write-back stage stores data into the Register File if specified. All clocked modules in the design are synchronized with the global CPU clock. These modules all feature an asynchronous, active-low reset signal that initializes their state and sets all outputs to zero.

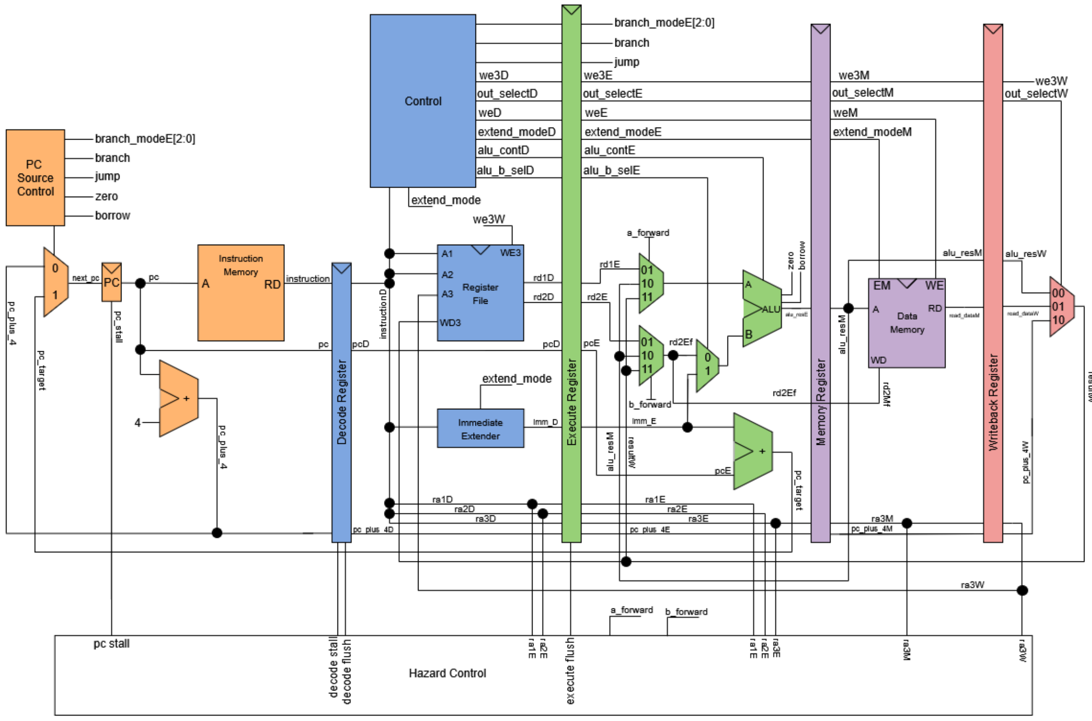


Figure 2: Global CPU Architecture

2.2 Fetch Stage

The first stage of my CPU is the Fetch stage. This stage handles the program counter and Instruction Memory, and consists of 5 major components: the Program Counter (PC), the PC mux, the PC source control, the PC adder, and the Instruction Memory.

The PC is a simple register that updates its output with the input value on every rising clock edge. The PC has two control inputs: an asynchronous, active-low reset, and a stall. The reset input, when active, sets the PC output to zero, thereby resetting the program flow to the first instruction (address 0). The stall input prevents the PC from updating its value. This input is used during pipeline hazards that require the program counter to hold its current value for one or more cycles.

The PC mux determines which of two values is multiplexed to the PC input. One input to the mux comes from the PC adder, while the other is taken from the PC target adder for jumps and branches.

The PC source control module receives five inputs: borrow, zero, jump, branch, and branch_mode. It then determines which value the PC mux should select. Zero indicates if the two values sent to the ALU in the Execute stage are equal, while borrow goes high if operand A is less than operand B in the same ALU. Finally, branch_mode specifies the branching condition to the unit. If the branch condition is met or jump is active, the PC mux selects the branch target address, directing the program to the new instruction.

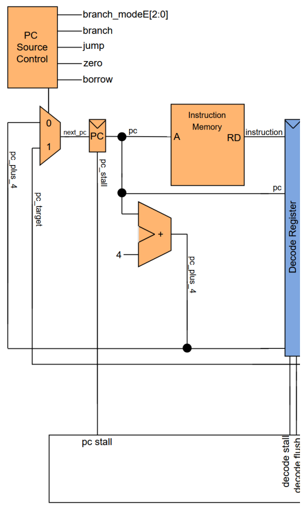


Figure 3: Fetch Stage Architecture

The PC adder is an extremely simple block with one input, pc, and one output, pc_plus_4, representing the next sequential instruction address. This block increments the current pc by 4 to point to the next sequential instruction. The PC increments by 4 because each instruction is a 32-bit word, requiring the pc to advance by 4 bytes (32 bits) to correctly address the next full instruction. My primary focus for this project was the overall CPU structure and functionality; therefore, the Verilog implementation of this block simply uses a procedural assignment, `out = pc + 4`. Although a more efficient or faster design could have been achieved with a ripple-carry or carry-lookahead adder, in the interest of time, I chose to let the synthesizer implement this block rather than designing it at the gate-level. If the critical timing path of the CPU were found to travel through this block, reworking the adding logic could prove beneficial.

Finally, the Instruction Memory holds all programmed instructions. The block contains 64 words of 32-bit memory, implemented as a Verilog 2D array, which can be initialized from a hex file containing the compiled RISC-V program. During synthesis, initial statements are disregarded; thus, to create a functional design, I replaced the memory array with a case statement that provides the same instruction set functionality. A future improvement will involve adding an initialization mode where this block can be programmed from the testbench, enabling the entire memory content to be included in the synthesized design as intended.

2.3 Decode Stage

The second stage of my CPU is the Decode stage. This stage takes the instruction and decodes it into control signals and data. There are 3 main elements within this stage: Control, Register File, and the Immediate Extender. The instruction is parsed into its component fields based on the RISC-V instruction formats:

Table 2: RISC-V Instruction Formats

Bit Fields	**31–25**	**24–20**	**19–15**	**14–12**	**11–7**	**6–0**
R-type	funct7	rs2	rs1	funct3	rd	opcode
I-type	imm[11:0]	N/A	rs1	funct3	rd	opcode
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B-type	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U-type	imm[31:12]	N/A	N/A	N/A	rd	opcode
J-type	imm[20|10:1|11|19:12]	N/A	N/A	N/A	rd	opcode

The register source addresses (rs1, rs2) are sent to the Register File. The entire instruction, excluding the opcode, is sent to the Immediate Extender. Finally, the opcode, along with funct7 and funct3, are routed to the Control block.

Handling the primary decoding, the Control block is one of the most important elements of the CPU. The Control block receives data from the current instruction, the opcode, funct3, and funct7, and outputs the necessary control signals to execute the instruction as intended. Outputs for the block can be seen in Table 3 and are passed through the pipeline, breaking off where needed.

The decoding is achieved through a series of case statements. The primary case statement determines how funct3 and funct7 are interpreted based on the opcode. Different RISC-V instructions require different interpretations of funct7 and funct3, even when the instruction type is the same; thus, the block must interpret this auxiliary information based on the opcode. Within each opcode selection, the block then interprets funct3, as most function types utilize funct3, while only R-type instructions use funct7. Finally, if necessary, a third-layer case statement handles funct7. Implemented in this way, the Control block accurately determines the exact instruction type. Once the block identifies the instruction, the relevant controls are set based on this information. The multi-layered approach to interpretation was chosen over a singular, large, lookup table-style case statement for two main reasons. Firstly, execution controls that apply to an entire instruction type, or subtype (based on funct3), can be set once, as opposed to setting them individually for every possible instruction. Secondly, when designing the CPU, I hypothesized that the large gate sizes and fan-out required to create a case statement for all possible instructions (e.g., a huge output mux and high-input AND gates for selection) would add significant timing delays. Therefore, a multi-layered approach would enable smaller gates and thus faster speeds for the large number of cases required.

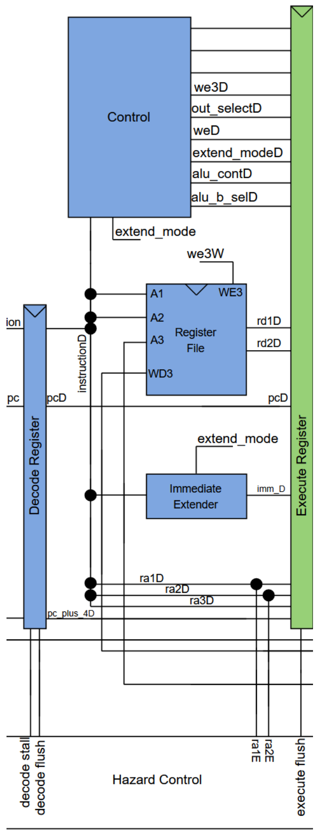


Figure 4: Decode Stage Architecture

After completing my CPU design, I revisited this assumption for verification. I created two different 64-case case statements: one implemented as a single switch case, and the other with three layers, each having four cases. I placed both of these case statements between two D-flip-flop registers and synthesized my design. I found that, for the same functionality, the multi-layered case statement was 66 ps slower. Although my design is slower than it could be, the critical path does not pass through the Control block; thus, I am content with its current state. A future improvement would involve redesigning any large case statements in my code to utilize the faster single-statement method.

The second major element in the Decode stage is the Register File. Containing 32 32-bit registers, the Register File is responsible for storing the data currently being operated on. Data reading is performed combinatorially based on the address inputs, with the outputs being updated on the negative edge of the clock (Verilog: `negedge clk`). Two read addresses, ra1 and ra2, are input, and the data held at these locations is read out as rd1 and rd2, respectively. Similarly, data write is performed synchronously when the write enable signal, we3, is high. During a data write, the data present on the wd3 input is written to the register address specified by wa3. Although the entire Register File block resides conceptually within the Decode stage, data writes to the Register File are performed as part of the final Write-back stage. In order to make this block work with the pipeline, the read operation is performed on the negative edge of the clock while the write is performed on the rising edge of the clock. If either ra1 or ra2 matches the wa3 (write address), the wd3 (write data) is forwarded directly to the corresponding output register (rd1 or rd2). Through this configuration, the Register File can function correctly across both the Decode and Write-back stages while avoiding data hazards.

Finally, the Immediate Extender completes the instruction decode. The sole function of the Immediate Extender is to extract and format the immediate value from the current instruction for use elsewhere in the CPU. As seen in the RISC-V instruction format table above, the immediate can be stored anywhere between the 8th and 32nd bits of the instruction; however, there are only five distinct immediate formats, each corresponding to an instruction type that employs an immediate. Based on this information, the extender can be implemented using a single case statement. In the Control block, a 3-bit signal, imm_select, is generated, which identifies the current instruction's immediate type. This signal is then passed to the Immediate Extender, where it serves as the input to the case statement. Each case then formats the input immediate in accordance with RISC-V guidelines, and based on the imm_select signal, the correctly formatted immediate is output and sent to the next stage.

Bit Width	**Signal Name**	**Controls Bit Position**	**Description**
1 bit	`reg_write`	12	Enables register file writeback
3 bits	`imm_select`	11:9	Selects immediate encoding type: I=001, S=010, B=011, J=100
1 bit	`alu_src_b`	8	Selects ALU operand B: 0 = register, 1 = immediate
1 bit	`data_mem_write`	7	Enables Data Memory write
2 bits	`out_select`	6:5	Selects output source: 00=ALU, 01=Load, 10=PC+4 (JAL)
3 bits	`branch_mode`	4:2	Branch mode: 000=No branch, 001=BEQ, 010=BNE, 011=BLT, 100=BGE
1 bit	`branch_enable`	1	Enables branch evaluation
1 bit	`jump`	0	Jump instruction flag (e.g., JAL/JALR)
6 bits	`alu_funct`	–	ALU operation
3 bits	`data_mem_extend`	–	Data Memory access width

Table 3: CPU Control Signal Breakdown

2.4 Execute Stage

With the instruction decoded into control signals and relevant values, the CPU is ready for the Execute stage. This stage encompasses the ALU and the PC target adder. This stage comprises five blocks: the ALU, the ALU B input select mux, rd1_forward and rd2_forward muxes, and the PC target adder. Through these blocks, the arithmetic operation specified by the instruction is executed. The ALU output is sent to the Memory stage register, while signals relevant to jump and branch instructions, such as zero, borrow, and the imm_pc, are sent back to the Fetch stage. Additionally, this stage handles forwarding hazards, which occur when the instruction in the Execute stage requires the result of a previous instruction currently in the Memory stage or the Write-back stage.

The PC target adder is the only block in this stage not directly related to the primary ALU of this CPU. Two inputs are sent to this combinational block: the current pc and the immediate. These two inputs are summed and output as the signal imm_pc, which is connected to the PC mux in the Fetch stage. This block is necessary for J-type and B-type instructions that require the program to jump to a different location in the Instruction Memory. A simple adder powers the logic of this device. Once again, I allowed the synthesizer tool to generate the addition logic to prioritize project time and focus. If the critical timing path of the CPU were found to travel through this block, reworking the adding logic could prove beneficial.

The rd1_forward and rd2_forward modules are identical blocks with the same purpose: forwarding data from the Memory or Write-back stage to the Execute stage. A simple 3:1, 32-bit mux handles the data forwarding, where the three inputs are the data directly from the Register File, the data from the Memory stage, and the data from the Write-back stage. Both muxes receive the same Memory and Write-back stage data inputs, as seen in the datapath diagram (Figure 2). The control signals for these two muxes originate in the Hazard Control module, which monitors the ra1 and ra2 signals for the instruction currently in the Execute stage, checking for matches with the ra3 signal of instructions currently in the Memory and Write-back stages. If a match is found, the control signal directs the appropriate mux to forward the data, thus avoiding the collision.

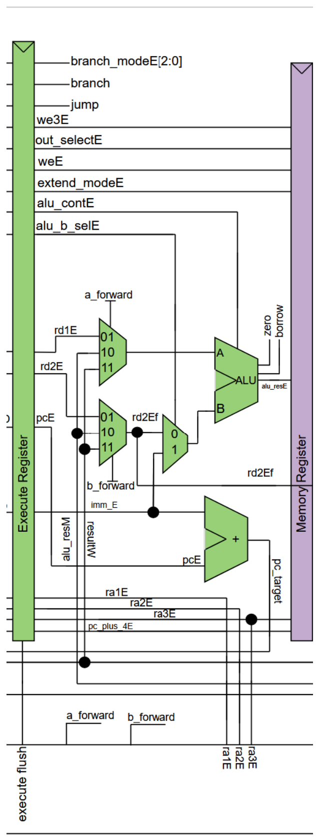


Figure 5: Execute Stage Architecture

Similar to the forwarding blocks, the ALU B input select mux is a 2:1, 32-bit mux. This mux's purpose is to enable the CPU to send either the rd2 (register data) or the immediate to the ALU's B input. This is necessary for instructions utilizing the immediate, either for storing or mathematical operations. The control for this mux is generated by the Control unit in the Decode stage and passed along to the Execute stage.

The final and most crucial module in this stage is the ALU. The ALU performs all arithmetic and logic calculations needed for the CPU, aside from the PC target adder. Currently, the ALU supports 10 different arithmetic and logical operations; however, it has extra logic to support more operations as I plan to expand the CPU's capabilities in the future. (Insert operation table here)

The ALU is built on a combinational multiplexed architecture, where all possible operations are calculated in parallel, and a multiplexer selects which result to send to the output. This means that the ALU's speed, and thus the speed of the CPU, is determined by the slowest possible calculation, in addition to the large 19:1, 32-bit mux. While this is not an efficient design choice, my primary motivation for this CPU project was to learn about computer architecture and CPU functionality; therefore, I opted for this familiar architecture. In future iterations of this CPU, I will modify this ALU in order to increase global CPU clock speed, as will be discussed later. Aside from the 32-bit alu_out result, the ALU generates two additional outputs: zero and borrow. The zero signal is driven high if the alu_out is equal to zero. This is calculated by taking a logical NOR operation across all bits of the ALU output; if all bits are zero, the zero flag is asserted. The borrow signal is generated by the subtract operation. When the b operand is subtracted from a and the result indicates a negative value (i.e., a borrow is needed), the borrow flag is driven high. This flag is used to indicate if a is less than b and most commonly appears in B-type instructions. Finally, the alu_funct signal, which is passed to the ALU mux, controls which operation the ALU "performs" and is defined by the following table:

Similar to the other controls, this signal is generated in the Control block in the Decode stage and passed into the Execute stage along with the rest of the instruction's data. With the instruction executed and the result calculated, all relevant data can now be passed to the next stage.

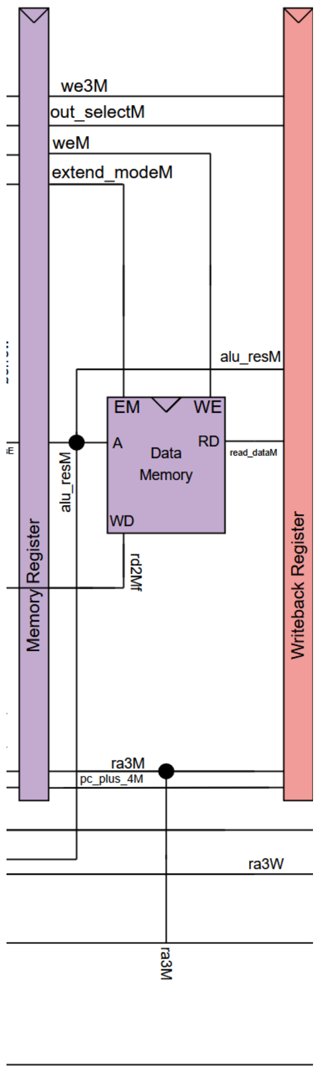
Table 4: ALU Opcode List

Hex Code	Description
0x0	No operation
0x1	Add two operands
0x2	Subtract second operand from first
0x3	Bitwise AND
0x4	Bitwise OR
0x5	Bitwise XOR
0x6	Set if less than (signed)
0x7	Set if less than (unsigned)
0x8	Shift left logical
0x9	Shift right logical
0xA	No op (Formally Multiply signed × unsigned)
0xB	No op (Formally Multiply unsigned)
0xC	No op (Formally Lower 32 bits of multiply)
0xD	No op (Formally Upper 32 bits of multiply)
0xE	Shift right arithmetic
0xF	No op (Formally Divide signed)
0x10	No op (Formally Divide unsigned)
0x11	No op (Formally Remainder (signed))
0x12	No op (Formally Remainder (unsigned))

2.5 Memory Stage

As the penultimate stage, the Memory stage focuses on writing and reading data from the Data Memory, also known as RAM. Only store and load instructions interact with the Data Memory; other instructions pass the ALU result data directly from the Memory stage register to the Write-back stage register. The Memory stage contains only one module: Data Memory; however, its signals are monitored by the Hazard Control in order to check for forwarding opportunities and potential stalls.

The Data Memory serves as the primary data storage for the CPU. In a real CPU, this would most likely be implemented with a significant amount of DRAM. For my CPU, as the overall design is much simpler than a real CPU, it includes only 64 32-bit words of RAM. This memory is implemented as a 2D Verilog array. As a whole, the Data Memory module has four inputs and one output. When the write_enable input is high, the Data Memory writes the data currently at the write_data input (which comes from rd2 in the previous stage) to the memory location specified by the address input. The data_mem_extend signal tells the module how to handle data width as needed; for example, if the instruction is store half word, the Data Memory will write the least significant 16 bits from the write_data input to the specified address. Similar logic applies to load instructions. The data write operation occurs synchronously; however, the read operation is combinational. This means that the read_data is continuously output to the next stage based on the current address.



2.6 Write-back Stage

The final stage of my CPU is the Write-back stage. This stage determines the final output of the instruction and writes the value back to the Register File. There are two modules in this stage: the out_select mux and the Register File, the latter of which is shared with the Decode stage. As with the other stages, control signals and hazard data are passed from the Decode stage through the pipeline until this final stage is reached. After this stage, all data not designated for storage is discarded.

The out_select mux is a 32-bit, 3:1 mux that controls which of the three inputs is selected as the final value to write back. The three inputs are the alu_resW (ALU result, passed unchanged from the Execute stage), read_dataW (data read from the Data Memory), and finally, pc_plus_4W (which is routed through all stages from the Fetch stage). The control signal for this block is based on the instruction and passed through from the Decode stage. This mux has one output, data_out, which connects to the wd3 input of the Register File. In this manner, the data written to the Register File can be chosen based on the instruction, whether it's from the Data Memory for load instructions, the pc_plus_4 for linking instructions, or simply the ALU output for all other instructions.

As discussed earlier, the Register File has separate write controls, allowing it to function within both the Write-back and Decode stages. The architecture of this block remains the same, with the data write function working the same as in the Data Memory module; the register address and write enable are created in the Decode stage and passed through until being used now. With this cycle complete, a single instruction has finished execution.

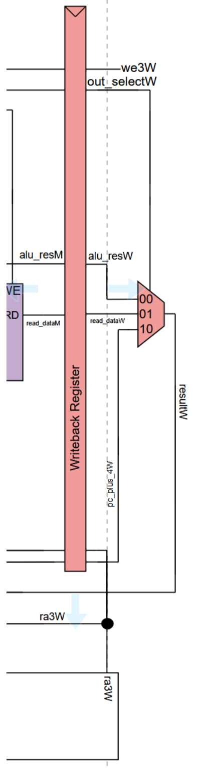


Figure 7: A tall narrow image

3 Testing

Once I had completed the Verilog implementation of my CPU, it was time to test. Testing is an extremely important part of digital logic design, and to no surprise highlighted many errors in my CPU design.

3.1 Methodology

As this is a personal project and not for production, the emphasis on extremely thorough testing was not held to the same stringent level as in a professional environment. Instead of individually testing and verifying each block of my CPU, I opted for an ad hoc approach. While this methodology can complicate debugging and issue resolution, my research and preparation for the design provided me with a strong understanding of my CPU's operation. The main intermediate testing I performed before testing my final design was creating, testing, and verifying a single-cycle CPU. This CPU was easier to implement as it lacked pipelining, and thus presented no hazards or stage registers. I developed and performed my testing plan on this CPU first, which informed my understanding of how my pipelined CPU should function.

The testing plan for my CPU involved first writing a RISC-V program that would utilize all of the main CPU functionality, i.e., all instruction and hazard types. This program would continually modify a single value and then write it to a specific address in the Data Memory. My Verilog testbench monitors this address in the Data Memory and will indicate whether the test is successful. As a precautionary measure, I would step through the Register File in SimVision to ensure that the register values matched the instructions being executed.

This approach to testing allowed me to verify all CPU functionality with full confidence, while prioritizing my time as the designer. If I were designing this for production or as part of a larger system where my design needed to meet specific integration requirements, detailed, individual module testing would be the superior option.

3.2 Errors

Through testing, I was able to uncover many errors in my Verilog. One of the largest, yet thankfully simplest to resolve, was in my interconnect. Every block in my design has multiple inputs and outputs, often serving the same purpose but across different pipeline stages. As such, the namespace quickly became polluted. At times, I even

```
    addi x1, x0, 5            # x1 = 5
    addi x2, x0, 10           # x2 = 10
    addi x3, x0, 84           # x3 = 84 (target memory address)
    addi x4, x0, 3            # x4 = 3
    addi x5, x0, 7            # x5 = 7
    add   x6, x1, x2          # x6 = 5 + 10 = 15
    sub   x7, x6, x1          # x7 = 15 - 5 = 10
    xor   x8, x7, x1          # x8 = 10 ^ 5 = 15
    or    x9, x7, x1          # x9 = 10 | 5 = 15
    and   x10, x9, x8         # x10 = 15 & 15 = 15
    sll   x11, x4, x5         # x11 = 3 << 7 = 384
    srl   x12, x11, x5        # x12 = 384 >> 7 = 3
    sra   x13, x11, x5        # x13 = 384 >> 7 = 3 (arith)
    slt   x14, x4, x5         # x14 = (3<7)?1:0 = 1
    sltu  x15, x5, x4         # x15 = (7<3 unsigned)?0 = 0
    addi  x16, x2, -5         # x16 = 10 - 5 = 5
    xori  x17, x2, 7          # x17 = 10 ^ 7 = 13
    ori   x18, x2, 0xF0      # x18 = 10 | 0xF0 = 0xFA
    andi  x19, x18, 0x0F      # x19 = 0xFA & 0x0F = 0x0A (10)
    slli  x20, x4, 2          # x20 = 3 << 2 = 12
    srli  x21, x20, 2         # x21 = 12 >> 2 = 3
    srai  x22, x20, 2         # x22 = 12 >> 2 (arith) = 3
    slti  x23, x4, 20         # x23 = (3<20)?1=1
    sltiu x24, x5, 3          # x24 = (7<3 unsigned)?0=0
    sw    x2, 0(x10)          # memory[15] = 10 (writes at address in x10)
    sh    x1, 4(x10)          # memory[19] halfword = 5
    sb    x1, 8(x10)          # memory[23] byte = 5
    lw    x25, 0(x10)         # x25 = memory[15] = 10
    lh    x26, 4(x10)         # x26 = memory[19] = 5
    lb    x27, 8(x10)         # x27 = memory[23] = 5
    lhu   x28, 4(x10)         # x28 = memory[19] = 5 (zero-extended)
    lbu   x29, 8(x10)         # x29 = memory[23] = 5 (zero-extended)
    add   x30, x25, x26       # x30 = 10 + 5 = 15
    beq   x1, x2, skip1       # NOT taken (5 != 10)
    bne   x1, x2, br1         # TAKEN (5 != 10)
    sw    x2, 0(x3)           # NOT executed (branch skipped)

skip1:
    blt   x1, x2, br2
    bge   x2, x1, br3
    bltu  x1, x2, br2
    bgeu  x2, x1, br3

br1:
    addi x31, x0, 1           # executed due to bne
    jal   x6, br3             # jump to br3, also x6 = PC+4

br2:
    addi x31, x0, 2           # NOT executed (branch not reached)

br3:
    addi x31, x0, 3           # final branch target, x31=3
    addi x7, x0, 4            # x7 = 4
    jalr x8, 0(x7)            # jump to PC+4 -> effectively continues normally
    addi x9, x0, 70           # x9 = 70
    addi x9, x9, 1            # x9 = 71
    sw    x9, 0(x3)           # memory[84] = 71 (final expected result)
```

Figure 8: CPU Test Script

created multiple buses for the same interconnect. Testing quickly highlighted these faults, and careful review helped me correct the signal routing issues.

The second major issue I encountered was hazard control in the pipeline. Not only was the organization of hazard control complicated to manage, but meeting timing requirements between Write-back and Register File read proved to be a significant challenge. Initially, my Register File would write data to the registers on the positive edge of the clock and read on the negative edge of the clock. Conceptually, this made sense as the data would be present in the register half a clock cycle before the read operation. However, at the beginning of the clock cycle, the data destined for the Register File write is launched. Before the data can reach the Register File to be written, it has to pass through the out_select mux. Consequently, the data does not arrive in time to be clocked into the Register File and then read. To solve this, I added logic to the Register File to directly forward the output data to either the rd1 or rd2 register as necessary. With this fix, my CPU was fully functional.

4 Synthesis

4.1 Setup

Now that I had verified my CPU design, it was time to synthesize it. Synthesis takes the Verilog HDL I wrote and converts it into a detailed gate-level netlist. This netlist is then used for a detailed simulation of the design, accounting for power, area, heat generation, and most importantly for me, timing. For my Cadence Genus synthesis, I used a pre-existing synthesis script provided during my second RTL class. Using this file provided a functional synthesis script and allowed me to focus on my design; however, in the future, as I advance my design, I will learn how to configure my own synthesis setup to achieve more optimal results.

4.2 Issues

The first issue I experienced was simply getting my design to synthesize successfully. As mentioned in my Instruction Memory block, initial statements are disregarded by the synthesizer. Due to this, my CPU had no programmed instructions and thus produced no output. By design, the Genus synthesizer is extremely aggressive and will remove all logic it deems unnecessary. However, in this case, it was resulting in a design with disconnected inputs and outputs (effectively dead logic). The solution to this problem was to hardcode my instructions directly into the CPU. This approach obviously comes with significant limitations; however, for the purposes of validating my design, it was adequate. Future iterations of my CPU will feature an input to program the Instruction Memory, which will both resolve this issue and allow my CPU to be programmed and reprogrammed like a real microcontroller. After

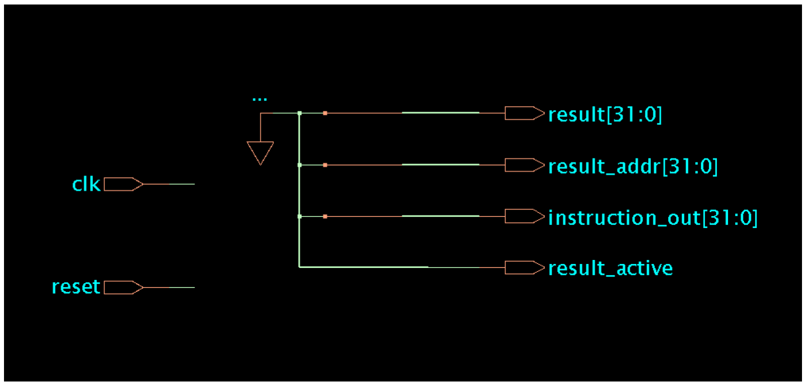


Figure 9: Failed Synthesis

running my simulation, I observed in SimVision that nearly every signal was in a high-impedance (Z) state. After the reset completed, the Fetch and Decode stages operated for a few cycles, and then the entire CPU transitioned into Z-states (unknown values).

Although this turned out to be a frustrating and difficult to track down issue, it allowed me to learn important design considerations and the importance of thinking about the hardware you are designing with Verilog, not just the HDL lines you write. My first approach to solving this bug was based on how I had approached issues in my pre-synthesis simulation: stepping through the simulation cycle by cycle and looking for blocks outputting incorrect results based on the given input. This, along with walking through my Verilog, did reveal a few minor mistakes I had made, but it didn't fix or provide any insight into why my CPU was floating to high-impedance states.

It wasn't until I decided to examine the timing report that I found the real issue. The timing report is based on a 6000 ps clock period; again, this is due to the Genus synthesis setup provided by my class. The synthesis report showed that I was significantly failing my setup timing requirement, specifically by 24719 ps. This is an extraordinarily large violation of my setup requirements. It was so significant that, despite the synthesis basing the report on a clock frequency of 160 MHz, I was failing timing in my post-synthesis simulation, which was clocked at 50 MHz. This is particularly astounding as setup violations can be resolved by de-clocking the design, while hold violations require a redesign. To resolve this violation solely by slowing the clock speed, my CPU would need to run at 32 MHz or slower.

Delving deeper into the timing report, the critical path (the one causing the massive delay) was listed as launching from the Write-back register (specifically the ra3W signal) and landing at the PC. This path would trace the data forward from the output through the Execute stage, through the longest instruction's path in the ALU, and finally to the PC for a jump or branch. Clearly this is a long path; however, when scrolling through the path, I noticed several blocks with a "div" label. Upon seeing this, I realized the root cause of my timing issues. In developing my CPU, I included ALU functionality for multiply and divide instructions despite not fully supporting these instructions elsewhere in my CPU. However, the logic for these operations is extremely time-intensive. As such, when determining the critical path through my CPU, the synthesizer identified the division path through the ALU, significantly increasing the reported critical path delay. Removing the multiplication and division modules, along with some minor troubleshooting, resulted in significantly better results.

4.3 Results

After troubleshooting my design, I was able to achieve a setup violation of only 13 ps. With this data, my maximum possible clock speed is 166 MHz, and my CPU is fully functional in post-synthesis simulation. Along with the timing report, Genus generates power and area reports. These show that my design will consume 10.39 mW and occupy 0.524 mm^2 (based on the Skyworks 130 nm design node provided in my coursework). Combining logic where possible, such as add and subtract operations, will help reduce these numbers as fewer transistors will reduce both static power loss and area. The critical path of my CPU was found to be from the Execute stage register's launch of ra2 (the address of register 2) through the Hazard Control to check for a hazard. The path then runs through the rd2_forward mux, through the ALU, and then back through the Hazard Control to check for a jump or branch. Finally, the Hazard Control will issue a flush to the Execute register, which is the endpoint of this critical path (specifically, at the D-input of a register that holds an ALU related function in the Execute stage).

Table 5: Critical Timing Path Summary

Path Status	VIOLATED (-13 ps)
Check Type	Setup (Pin `DO/ER/alu_functE_reg[3]/CK→D`)
Clock Group	clk
Startpoint	`DO/ER/ra2E_reg[2]/CK`
Start Clock	clk
Endpoint	`DO/ER/alu_functE_reg[3]/D`
End Clock	clk
Capture Edge	6000 ps
Launch Edge	0 ps
Source Latency	0 ps
Net Latency	0 ps (both capture & launch)
Arrival Time	6000 ps
Setup Time	-53 ps
Uncertainty	150 ps
Required Time	5903 ps
Launch Clock	0 ps
Data Path	5916 ps
Slack	-13 ps

Table 6: Power Breakdown (Unit: W)

Category	**Leakage**	**Internal**	**Switching**	**Total**	**Row%**
memory	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
register	3.30636e-05	4.52288e-03	1.18894e-03	5.74489e-03	55.31%
latch	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
logic	5.77978e-06	1.53670e-03	3.04769e-03	4.59017e-03	44.19%
bbox	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
clock	4.86217e-10	1.68556e-06	5.03885e-05	5.20745e-05	0.50%
pad	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
pm	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
Subtotal	3.88439e-05	6.06127e-03	4.28702e-03	1.03871e-02	100.00%
Percentage	0.37%	58.35%	41.27%	100.00%	100.00%

Table 7: Area Report

Instance	**Module**	**Cell Count**	**Cell Area**	**Net Area**	**Total Area**
micro_2	NA	11563	252 729.115	271 167.572	523 896.687
D0	datapath	11563	252 729.115	262 182.754	514 911.869
C0	control	75	651.305	1 151.093	1 802.398
D0	d_mem	4342	122 468.155	107 966.426	230 434.581
E0	i_extender	68	563.702	840.475	1 404.178
ER	execute_reg	376	9 085.892	3 636.704	12 722.596
I0	i_mem	85	816.988	1 330.167	2 147.154
MR	memory_reg	107	4 888.595	0.000	4 888.595
WR	write_reg	111	4 126.835	25.773	4 152.608

5 Place and Route

Place and Route (PNR) is the process of taking a verified, synthesized design and transforming it into physical structures and layers that can be sent to a semiconductor fab for fabrication into a physical chip. Although I am not going to have my CPU design made, I still wanted to perform PNR and view my design as a chip. Similar to synthesis with Cadence Genus, I used a pre-made **.tcl** file for Cadence Innovus from my second RTL class. Another area of future improvement will be learning how to write my own **.tcl** scripts so I can optimize my PNR for my design.

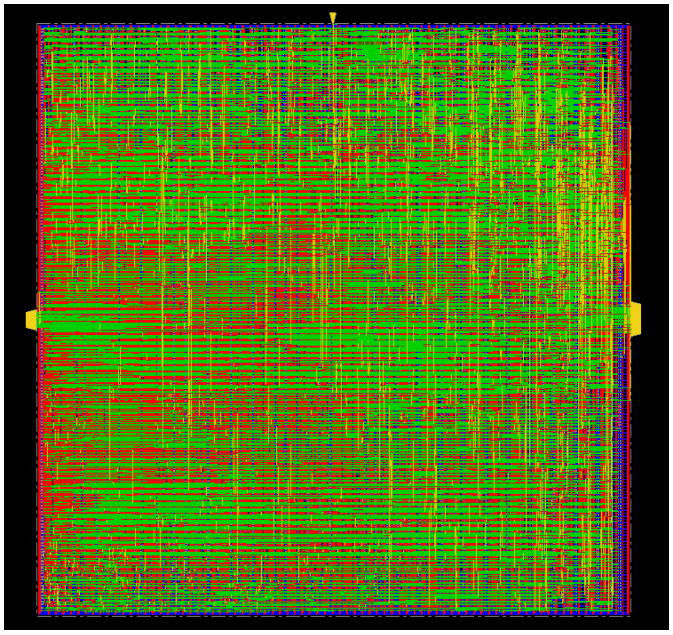


Figure 10: CPU post Innovus PNR

The result was amazing to see. I had never seen such a large design placed before, and I felt a lot of pride in my CPU.

6 Conclusion

6.1 Summary

In this project, I transformed a single ambitious idea, to build a CPU, into a tangible design that could theoretically be sent to a fab for production. Although this CPU has severe limitations and is extremely simple compared to anything available for sale today, I am extremely proud of my design and, more importantly, what I have learned. My CPU is only the first step in my journey to learn about computer architecture and RTL design. I will continue to learn more about CPU design and improve my CPU.

6.2 Takeaways

The most important skill I have learned through this project is how to sufficiently break down a large project until each individual step becomes understandable. When I first began to understand pipelining and CPU stages, I couldn't imagine how to create logic that would function as the theoretical ideas suggested. However, by breaking down a stage into individual blocks, and those blocks further into individual logic steps, I could see a clear path to project completion. For example, a block like the ALU can be very complex and is essentially the heart of the CPU. However, when examining its structure, the ALU just a set of individual operations and a mechanism to select between them. Each operation can then be broken down and addressed as its own module, and to select between them a mux can be used. This method of identifying known logic structures and small modules that needed to be designed, when applied consistently, resulted in a completed CPU.

A significant final takeaway for me was the confirmation of my strong interest in the computer architecture industry. Completing this project has shown me how satisfying and endlessly interesting computer architecture is. Everything from reading about new advanced technologies to seeing a simple Verilog design function as intended gives me the engineer's spark and drives me forward. The ability to understand, on the most basic level, how all the technology I interact with and devices I have written programs for work, is fascinating. I will continue to pursue computer architecture projects, education, and a career in the field.

6.3 Future Improvements

As mentioned throughout this report, there are numerous improvements to be made to my CPU. Thankfully, I plan on continuing to make modifications until this design has reached its scaling limits, at which point it will be time for a new architecture. Beyond the smaller fixes mentioned previously, such as providing a way to program the CPU's Instruction Memory from the testbench and increasing the instruction set, I have plans for more systemic upgrades.

The biggest upgrade I am looking forward to making is implementing an out-of-order pipeline. Currently, instructions are executed in the order they are stored in the Instruction Memory; however, when hazards resolved through stalls occur, CPU efficiency drops as not all stages of the CPU are being utilized. To prevent these drops in efficiency, an out-of-order CPU will execute instructions that do not rely on the currently executing instructions, and therefore are not impacted by the hazard. By inserting out-of-order instructions into the pipeline, all stages of the CPU can maintain constant usage. Out-of-order pipelines are standard in current CPU designs, and gaining experience in their operation appeals to my desire to understand how computers work.

Another upgrade I am considering adding is a dedicated multiplication unit. This unit would execute multiply and divide operations. As discussed previously, both multiplication and division are extremely time-intensive operations. Due to this, I hope that by incorporating dedicated hardware, I can create an alternate path for these operations that spans multiple stages (e.g., from Execute to Write-back). This is feasible as multiplication and division instruction results are stored in registers, not data memory. By spanning multiple stages, the path for these instructions can accommodate more clock cycles while still meeting timing requirements.

Finally, the last optimization I am looking to make to my CPU is increasing the number of pipeline stages. By adding more stages, I can decrease the logic per stage, and thus increase the clock speed of the CPU. If incorporated correctly, adding stages will increase my CPU's instruction throughput.

6.4 Final Thoughts

This project has been absolutely fascinating, and it is only just a start. I would highly recommend this project to anyone interested in computer architecture, as it shows you the true power of hardware and digital logic. Through this project, I have further reinforced my interest in a career in computer architecture. I am greatly looking forward to continuing my project and creating a faster, better, more capable CPU.