ECEN 4243
Lab 3: Pipelined Microarchitecture

Jordan Paul, Ben Sailor

https://github.com/jordanpaul98/ECEN4243_CompARC/tree/main/lab3

Contents

Introduction

Building upon the foundation laid out in Lab 1, Lab 2 diverged by using System Verilog instead of a C-based simulator to construct a single-cycle RISC-V machine. This shift allowed for a hardware implementation of the RISC-V ISA more closely resembling actual hardware, enhancing comprehension of the architecture's hardware components. The objective was to deconstruct the processor into modules, each dedicated to a specific task, and gain insight into instruction decoding and control mechanisms.

For Lab 3 introduces the implementation of a pipelined RISC-V machine, necessitating the construction of a pipeline consisting of five stages. With multiple instructions potentially "in-flight" simultaneously, detecting and handling dependencies between instructions becomes crucial. This lab extends the work from Lab 2, exploring the challenges associated with pipelined hardware implementation. Utilizing the SystemVerilog Hardware Descriptive Language (HDL), this laboratory aims to illustrate the complexities of executing tasks in parallel within hardware, highlighting that despite its seamless nature in human cognition, achieving parallel execution in hardware is non-trivial

Task

Lab 2 single-cycle CPU provides the understanding and backbone for Lab 3. Having completed all instructions for the single-cycle CPU, including testing and integration of additional hardware components, this provides a better understanding of its functioning. However, Lab 3 requires a different approach to the design with the concept of a pipelined CPU. Fortunately, the understanding gained from Lab 2 provides a fundamental foundation and insight into constructing a pipeline CPU.

Like Lab 2, where we built a single-cycle CPU with limited functionality, Lab 3 provides the basics to run a few memfile programs. These include 13 instructions already built into the processor along with Hazard Unit, but the other 25 instructions will need to be built out into the pipeline processor. Despite this new challenge, the previous experience offers a valuable starting point and a clearer idea of how to proceed with the pipeline CPU design. This time, we need to divide the workload across multiple stages and ensure that each stage operates efficiently within the pipeline.

Some of the design with Hazard Unit require additional functionality as well, The hazard Unit only provides enough functionality for the 13 instructions provided but doesn’t account for the additional stages within the pipeline, this will require updating and building into logic to reset and flush the stages beyond the Arithmetic Logic Unit.

The transition to a pipelined architecture introduces complexities such as distributed control and the need for a Hazard Detection Unit (HDU) to manage dependencies between stages. However, with the groundwork laid in Lab 2, the design can be expanded to incorporate the needed control signals and new data paths along the stages.

Processor Components

The processor is broken up into components designed to perform particular tasks. Each component takes input from another component and will output to the next component. Since this is now a pipeline processor, some of the output will be fed into a register so that the output will hold a state between each clock cycle to its next component. Additionally, a new component is introduced which is the Hazard Unit, providing oversight of the stages. It governs which stages need to stall or be flushed, ensuring smooth operation and handling dependencies between stages effectively

- Controller
- Hazard Unit
- PC Source
- PC + 4
- PC Target
- Instruction Memory
- Register File
- ALU source
- ALU
- Sign Extend
- Data Memory
- ALU Result

Pipeline Stages

The process demonstrates the conversion of the single-cycle processor into a five-stage pipeline by inserting registers between stages, facilitating efficient instruction execution. Each stage, including Fetch, Decode, Execute, Memory, and Writeback, performs specific tasks such as instruction fetching, control signal decoding, ALU operation execution, data memory access, and result writing to the register file. To build the Pipeline Processor registers are used within last data path of each stage, these all capture the input pin on rise and output previous inputs into the next stage.

Stages within Pipeline:

1. IF – Instruction Fetch
2. ID – Instruction Decode and register file read
3. EX – Execution or memory address calculation
4. MEM – Memory access
5. WB – Write to register file

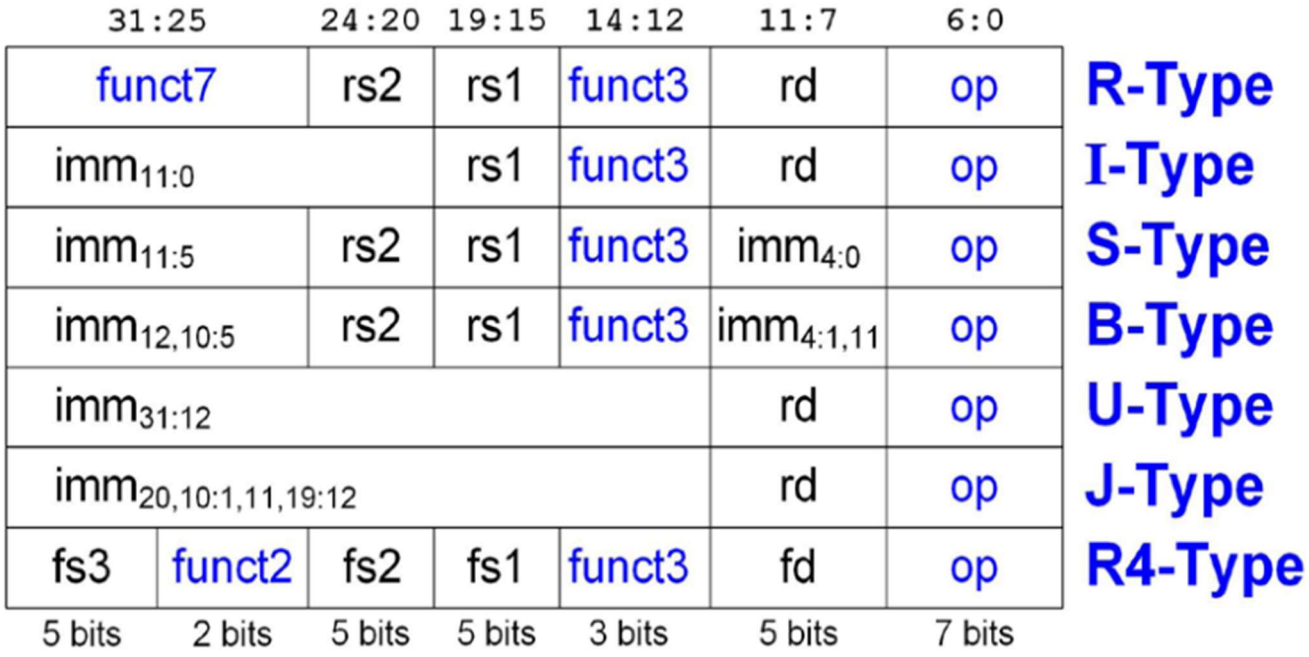
Instructions Set

There are 38 instructions in total, with 37 instructions performing various program operations, and the last instruction, ECALL, effectively terminating the program and returning control back to the operating system. To test the functionality of the pipeline processor, there are 37 memfiles, each corresponding to a specific instruction. Each memfile concludes with an ECALL instruction, ensuring proper program termination and allowing for systematic testing of the processor's capabilities.

Type	Instruction	Description
R	ADD	Add
I	ADDI	Add Immediate
R	AND	and
I	ANDI	And immediate
U	AUPIC	Add upper immediate to PC
B	BEQ	Branch if =
B	BGE	Branch if >=
B	BGEU	Branch if >= unsigned
B	BLT	Branch if <
B	BLTU	Branch if < unsigned
B	BNE	Branch if !=
J	JAL	Jump and Link
I	JALR	Jump and Link Register
I	LB	Load Byte (8 bits)
I	LBU	Load Byte Unsigned (8 bits)
I	LH	Load Half (16 bits)
I	LHU	Load Half Unsigned (16 bits)
I	LW	Load Word (32 bits)
S	SB	Save Byte (8 bits)
S	SH	Save Half (16 bits)
S	SW	Save Word (32 bits)
R	OR	or
I	ORI	or immediate
R	SLL	Shift left logical
I	SLLI	Shift left logical immediate
R	SLT	Set less than
I	SLTI	Set less than immediate
I	SLTIU	Set less than unsigned immediate
R	SLTU	Set less than unsigned
R	SRA	Shift right arithmetic
I	SRAI	Shift right arithmetic immediate
R	SRL	Shift right logical
I	SRLI	Shift right logical immediate
R	SUB	subtract
R	XOR	xor
I	XORI	xor immediate
I	ECALL	Transfer control to OS
U	LUI	Load upper Immediate

Instruction Format

In order to perform actions within the processor, it needs to decode the instruction, and it has this format: After the fetch stage of the processor, the controller will read particular bits within a hexadecimal 32-bit instruction from the loaded Instruction memory. Within this 32-bit format, a structured bit layout enables rapid and accurate decoding by the hardware. Consistent bit fields, including Opcode, Write Address, Read Address, and Functions, are maintained across different instruction types to ensure consistency. Unused fields may exist depending on the instruction, with the immediate field utilized in such cases. This standardized approach facilitates efficient decoding, enabling the hardware to interpret instructions quickly and accurately.



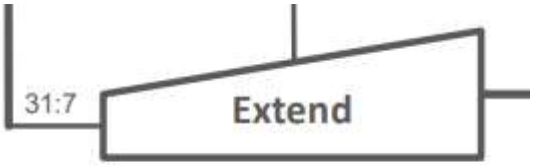
Sign Extend

Given that RISC-V instructions are limited to 32 bits, and many bits are already utilized for the instruction type, the immediate is constrained to 12 bits for most immediate instructions. While this might seem limiting, statistically, constant values are often not large. For cases where a larger constant is required, the Load Upper Immediate (LUI) instruction can be used to create a 32-bit immediate value in registers.

Another challenge arises with signed immediates, as there are both 12-bit and 20-bit immediates. This creates issues when applying 32-bit numbers to signed 12 or 20-bit representations. To address this, sign extension is used, examining the most significant bit and extending it to 32 bits. This ensures that signed values are preserved and correctly interpreted during instruction execution.

The Sign Extend unit is within the second stage of the pipeline along with the Control Unit and is controlled by a 3-bit control signal to toggle between the five instruction types that contain a constant embedded within their instruction. The five different signals determine the bits to read and orientate into the 32-bit output, feeding into both the ALU source Multiplexer and PC Target Adder.

BIT	Description	Alignment
3’b000	I type instructions (12 bits)	31:20
3’b001	S type instructions (12 bits)	31:25 11:7
3’b010	B type instructions (12 bits)	7 30:25 11:8 {0}
3’b011	J type instructions (20 bits)	19:12 20 30:21 {0}
3’b100	U type instructions (20 bits)	31:12 12{0}

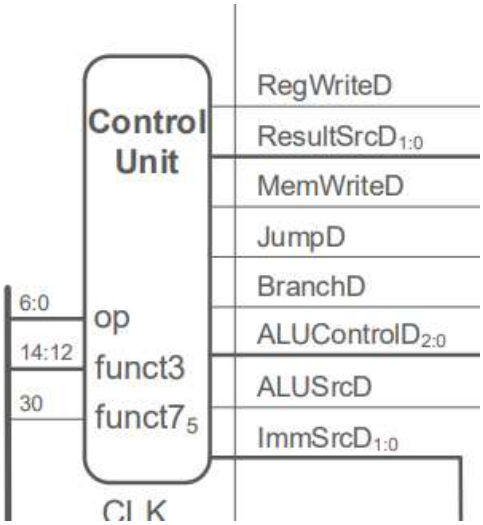


Controller

The most important component is the controller, which plays a crucial role in guiding the processor through the execution of instructions. Positioned within the second stage of the pipeline processor, the controller functions very similarly to that of Lab 2 but with some additional controls. Its primary function involves decoding the instruction type, determining the operation to be performed by the Arithmetic Logic Unit (ALU), selecting the source of the Program Counter (PC), specifying the source of the result, and managing the switching of multiplexers (MUX) to control the data path

In the final schematic, the Controller establishes connections with 8 key processor components, facilitating seamless communication and coordination between them

Component	Bits	Description
PC Source	1	Choose between PC+4 or PC-Address
Sign Extend	3	Sign Extend 4 5 different Instruction types
ALU source	2	Choose PC or Source A and Immediate or Source B
ALU Control	4	ALU control
PC target Select	1	Choose between PC Target or PC + Register 1
Mem Write	1	Enable Memory Writing
Result Source	2	Choose between ALU Result, Memory Data, PC+4
Register Write	1	Enable Register File Writing



To manage data path control, the controller configures its 8 outputs to either a high or low state based on the requirements of each component. With a total of 14 bits available for control, the controller precisely directs the operation of the data path components. The controller data path provides an additional bit compared to that in Lab 2, resembling the PC jump source. However, instead of being read off of the ALU operation anded with the jump bit, it has its own control signal from the controller. This control bit is named PC Target Select and is used for Jump and Link Register operations

OP Code	Reg Write	Imm Source	ALU source	Mem write	Result Source	Branch	ALU OP	PC Target Select	Jump	Short Description
0000011	1	000	01	0	01	0	00	0	0	Load
0100011	0	001	01	1	00	0	00	0	0	Save
0110011	1	xxx	00	0	00	0	10	0	0	R-Type
1100011	0	010	00	0	00	1	01	0	0	B-Type
0010011	1	000	01	0	00	0	10	0	0	I-Type
1101111	1	011	xx	0	10	0	xx	0	1	JAL
1100111	1	000	01	0	10	0	00	1	1	JALR
0010111	1	100	11	0	00	0	00	0	0	AUIPC
0110111	1	100	10	0	00	0	11	0	0	LUI

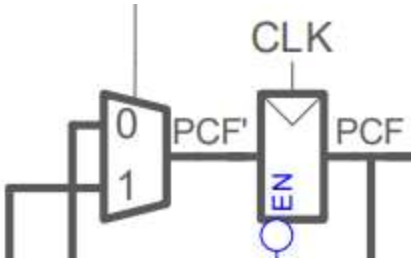
Additionally, because it's a pipeline processor, the control bits will need to maintain their state across the next three stages: Execution (EX), Memory (MEM), and Write Back (WB). As the control signals are not used beyond any further stages, they don't need to cascade down and can stop within the stage they are used in

PC Source Select

To switch between PC+4 or PC Address, a Mux is utilized. The majority of instructions will use the PC+4 signal. However, for conditions where the instruction needs to jump in memory, the controller can use a signal from a multiplier that outputs a Program Counter added with an Immediate for Branch or Jump types, or from a register with Jump and Link Registers. The output is then stored in a register which holds the state of the Program Counter until the rising edge of the clock signal.

The PC Source Select is positioned in the first stage of the pipeline. Once on a positive edge clock, the Program Counter goes into the Instruction Memory and outputs the current instruction. At the output of the First stage, the Program Counter will also need to maintain a held state which will be used in the third stage for Jump and Link Register operations.

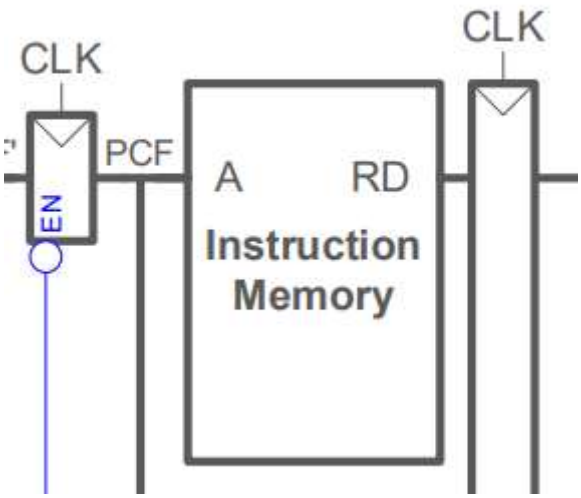
BIT	Description
1'b0	Select PC + 4
1'b1	Select PC Target Address



Instructions Memory

The Instruction Memory serves as a non-modifiable, preloaded memory component at the initiation of the System Verilog simulation, operating as a mapping reference for the Pipeline CPU. With each clock cycle, it retrieves the next instruction from the memory address specified by the Program Counter. This instruction is then passed to the Controller for decoding, enabling the configuration of the processor's pathways accordingly.

Positioned within the first stage of the pipeline, the output of the Instruction Memory goes into a register, which holds state until the next clock cycle. At this point, the instruction is sent to the next stage of the pipeline. This is where the term “In-Flight” originates. Unlike the Single Cycle CPU, where each clock cycle pushes the instruction through for a full processor operation in one cycle, in the Pipeline, instructions are “In-Flight” through each stage of the processor. Consequently, the process can have multiple instructions happening in parallel within the processor, with different stages working on different instructions at a time.



Flushing Instructions

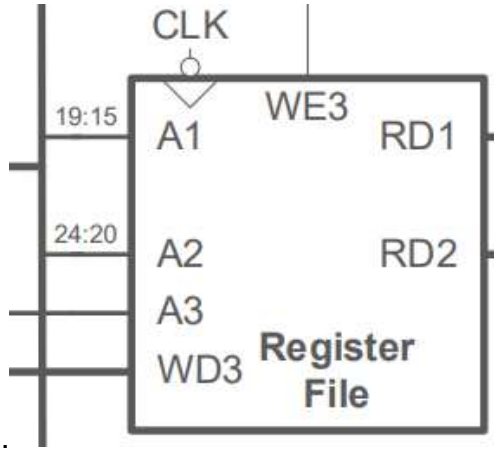
Because the pipeline processor has “in-flight” instructions, it can encounter issues when the processor needs to perform a branch. In the case of a branch instruction, the pipeline stages will continue to compute the proceeding instruction without knowing the outcome of the branch prediction. If the branch isn't taken, then the pipeline can continue as usual. However, if the branch is taken within the Execution stage, the pipeline processor will need to flush the Fetch and Decode stages. This is because the instructions within those stages are following the PC + 4 instructions when they actually need to branch to a different portion of memory. This is known as Branch Misprediction penalty, where the processor loses two clock cycles by flushing the pipeline and reloading instructions at the beginning of the branch. To perform a flush, the registers of each stage have a reset or clear pin which is controlled by the hazard unit.

Register File

The register file plays a vital role in the processor, providing a fast read and write means for short-term memory. It has two register read inputs and two register read outputs, along with a register write input and write data from the ALU Source Mux from the fifth stage.

Positioned in the Decode Stage of the pipeline, the register file ensures that instruction read and write register addresses maintain consistent orientation between instructions. This consistency results in faster output compared to accessing data memory. With 32 32-bit registers, the register file provides the necessary resources for fast and seamless program operation.

Similar to other processor components, the outputs of the register file, Register Read 1 and 2, maintain state between the Decode and Execution stages. To enable register writing, a Write enable signal reads from the Write Data input into the register set by the Write Address, which originates from the fifth stage of the pipeline. Since the Write Address comes from the Write Back stage, it must traverse through three registers: from the Decode stage through the Execution stage, Memory Stage, and Write Back. The Write Back operation takes 3 clock cycles to perform after the initial decode stage for write back operation



read.

Stalling logic

Mentioned in the Register File, due to the pipelined nature of the processor, the operation between the decode stage and the write back stage takes 3 clock cycles. This presents challenges for certain operations, such as reading from Data Memory. In a single-cycle CPU, this operation is seamless and completed within one cycle. However, in a pipelined architecture, instructions are "in-flight," leading to potential issues. If an instruction following a load instruction depends on the data loaded from memory, the data won't be available until 3 cycles later when it's written back into the register.

To ensure that the data is available when needed, the pipeline needs to be stalled on a load instruction if the proceeding register address matches that of the load instruction. Stalling the first two stages halts those instructions until the data is loaded into the register file, additionally the execution stage will need to be flushed to ensure when the pipeline resumes it synced correctly, preventing data hazards and maintaining correct program execution

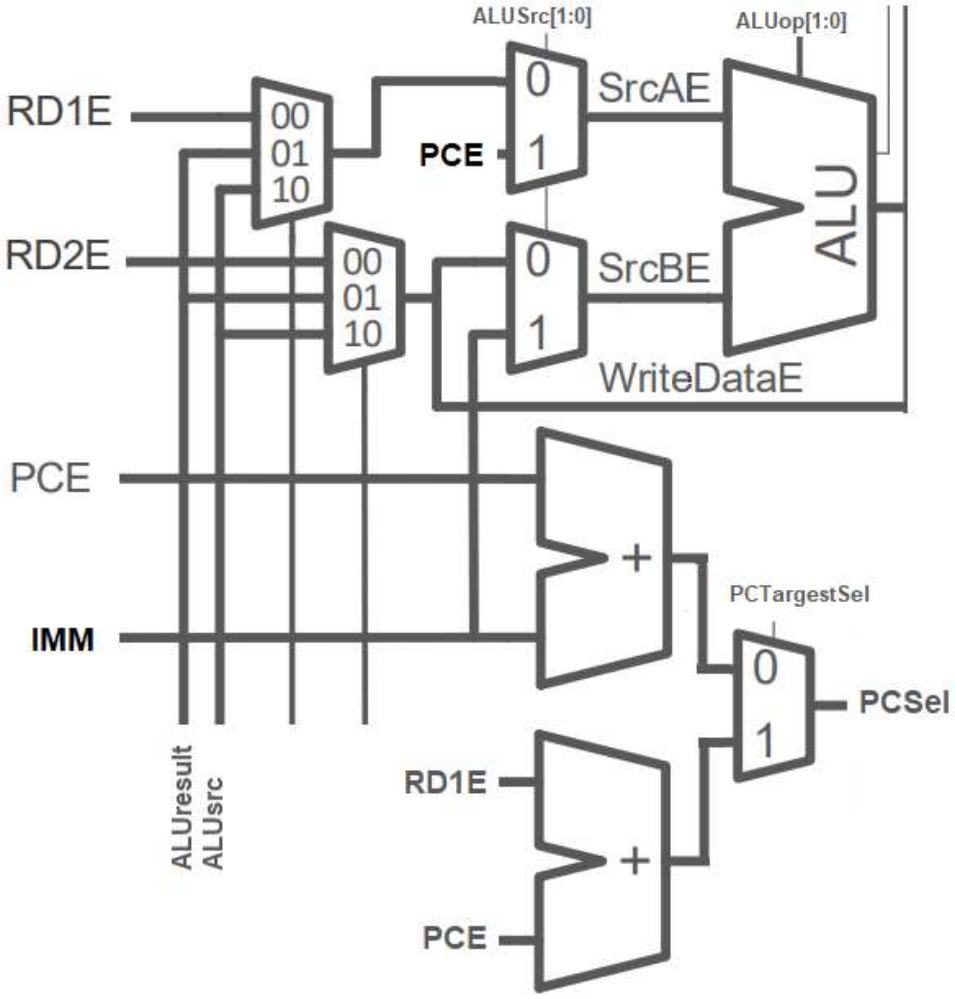
```
// stalls and flushes
assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
assign StallD = lwStallD;
assign StallF = lwStallD;
```

Execution Stage

The execution stage encompasses several components, including the Arithmetic Logic Unit (ALU) and the PC Target Adder. It also includes the ALU Source Registers, where ALU source 1 can be either Register 1 or PC, while source 2 can be either Register 2 or the Immediate source. Additionally, there are two 3-input multiplexers controlled by the Hazard Unit. These assist in expediting pipeline instruction execution by enabling the stage to bypass later stages by utilizing the ALU output as input.

Additionally, to facilitate the correct functioning of Jump and Link Register operations, another Adder, the PC + Register 1, has been integrated. Its output is directed into a MUX alongside the output of the PC Target, enabling selection between PC Target and PC + Reg1 for Branch & Jump or Jump and Link Register Operations.

The controller will change the MUX input selection to control the data paths depending on the instruction type, the controls were set from previous clock cycle along with the data from registers. By allowing the Hazard Unit to control the signal input to the ALU source multiplexers, when instructions are “In-Flight,” the ALU output might be utilized in the subsequent instruction to be executed. Rather than writing the ALU output back to memory (which takes time), the pipeline can instead route the output directly to the input of the ALU corresponding to the register that was intended to use the instruction within the Memory stage. This saves significant time, as a sequence of arithmetic operations can be performed without needing to write back to memory.



ALU Source Multipliers

Two MUXs are utilized to control the ALU's two inputs. MUXa selections determine whether to route PC or Read Address A into ALU input Source A. Additionally, MUXb selects either the signed Immediate or Read Address B for ALU Input Source B. MUXa is governed by Bit 1, while Bit 0 controls MUXb. However, due to the pipeline nature of the processor, data could also come from stages further into the pipeline if the addresses of the registers matches those of the later stages

BIT	Source Pathing
2'b00	Address A and Address B into ALU
2'b01	Address A and Signed Immediate into ALU
2'b10	PC Counter and Address B into ALU
2'b11	PC Counter and Signed Immediate into ALU

PC Target Select

To manage Jump and Link registers effectively, an extra MUX has been incorporated. This MUX accepts inputs from PC Target Results and another Adder, which adds PC and Register 1. The controller sets a PCTargetSel signal when a jump and link Register instruction is used; otherwise, PC Target will be used for normal jumps and branches. This arrangement gives us control over whether the PC Next, during a jump operation, retrieves its address from the PC Target or from the sum of Read Address and Immediate obtained from the ALU.

This arrangement differs from Lab 2, where the PC target selection input came from the ALU instead of an adder specified for this operation. In addition, the Controller provides a control bit for this operation, instead of requiring reading from two control signals through an AND gate to gain control. If more time was available, the ALU could have been redesigned to have essentially two outputs, similar to the diagram of the pipelined CPU from the RISC-V System on Chip text, where the PC-Target originates from the ALU and no additional Adders are needed. However, this solution provides the necessary components to facilitate Jump and Link Register operations without the need to reconfigure the whole execution stage and Arithmetic Logic Unit.

BIT	Description
1'b0	Select PC Target
1'b1	Select ALU Result

Forward Multiplexers

As mentioned previously, there are multiplexers that take input from the Register files, ALU Results in the Memory Stage, and Results selection in the Write Back Stage, which feed into the ALU source multiplexers. These multiplexers are controlled by the Hazard Unit and are used to provide faster instructions processing. In the event where the register address in the Execution Stage currently resides in the Memory or Write Back Stage, the Hazard Unit can switch the data path to pull from those stages to prevent the need to write those results into register file. which can be costly and may stall the processor while waiting for those results to be available in registers.

```
mux3   #(32)  faemux(RD1E, ResultW, ALUResultM, ForwardAE, WriteDataAE);
mux3   #(32)  fbemux(RD2E, ResultW, ALUResultM, ForwardBE, WriteDataBE);
```

ALU Control

A 4-bit signal is used to control the operation of the Arithmetic Logic Unit (ALU). This control signal dictates a range of operations based on the function and instruction type being executed. Additionally, the ALU is responsible for setting the Zero flag, which is used in Branch instructions.

In Lab 2, four flags were sent back to the controller: Zero, Carry, Overflow, and Negative. These flags were used with Function 3 to determine the instruction and flags to check for correct branch operation. However, for Lab 3, only the Zero flag is sent back to the controller. This is because the Zero flag is the only one used to determine if a branch needs to be taken. Depending on the branch type, different flags are used. During Branch Execution, the Zero flag is overwritten depending on the function being performed. Function 3 is passed to the ALU, which sets the Zero Flag accordingly. Originally in Lab 2, there were 15 different control types the ALU needed to distinguish. For Lab 3, this was reduced down to 11. This reduction was achieved by sharing control bit 4'b0001 to be used for SUB and all branches, as this is the operation performed during branching.

Although the ALU still needs to compute the flags for the operations, only the Zero flag will leave and go to the logic for branching. The output of the ALU results goes into a register that is available on the next clock signal for the next stage.

BIT	Operation	Instruction
4'b0000	add	ADD, ADDI
4'b0001	Sub and Branches	SUB, BEQ, BNE, BLT, BGE, BLTU, BGEU
4'b0010	and	AND, ANDI
4'b0011	or	OR, ORI
4'b0100	Exclusive or	XOR
4'b0101	Set less than	SLT
4'b0110	Shift right	SRL, SRLI
4'b0111	Shift Right Arithmetic	SRA
4'b1000	Shift left	SLL, SLLI
4'b1001	Shift Left unsigned	SLTU, SLTIU
4'b1111	Load Upper Immediate	LUI

Zero Flag – indicates that the two inputs into the ALU subtracted equal to Zero, unless during branch operation, the zero flag will be overwritten depending on the branch type and set accordingly

```
// sub with extra carry bit
assign carried = a - b;

assign carry = carried[32];
assign negative = sum[31];
assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;

always_comb
  case (funct3E)
            3'b000:  assign zeroB = (result == 32'b0);      // beq =
            3'b001:  assign zeroB = (result == 32'b0);      // bne !=
            3'b100:  assign zeroB = (negative ^ v);         // blt <
            3'b101:  assign zeroB = (negative ^ v);         // bge >=
            3'b110:  assign zeroB = carry;                          // bltu < unsigned
            3'b111:  assign zeroB = carry;                          // bgeu >= unsigned
            default: assign zeroB = (result == 32'b0);
  endcase

assign zero = zeroB ^ funct3E[0];
```

Data Memory

Data Memory serves as the storage medium for data outside of the register file in the processor, and it operates within the Memory stage, which is the 4th stage of the pipeline. When writing to memory, the "in-Flight" instruction follows the same order as the previous cycle, with the calculated result from the Execution stage, this lets the pipeline continue without need to stall. However, loading from memory adds complexity. As instructions are "In-Flight," when a load instruction occurs, it takes 3 clock cycles to begin fetching from memory, followed by 1 clock cycle to write it to the register file. This poses a problem when the following "In-Flight" instruction requires the value of the address being fetched from memory. The output of the data memory goes to the final register which will be available for Write Back Stage

To solve this issue, the Hazard Unit will stall the Fetch and Decode Stage and flush the Execution stage upon a load instruction. It will then take a clock cycle to write the loaded value back to the register file before lifting the stall to resume the instructions.

The instruction set enables storing and loading of various data types, including Words (4 bytes), Half-words (2 bytes), and Bytes (1 byte), each aligned based on the first two bits of the data address. However, this alignment alone doesn't account for the three different load and store types or loading unsigned values.

To address this, the Funct3 field is utilized by the Data Memory to determine the type of instruction to be performed. By examining the Funct3 field, the Data Memory can ascertain whether a Word, Half-word, or Byte alignment is required and appropriately sign-extend the data for signed load operations before loading it into the register file.

Mask

Data_address[1:0]	Word	Half	Byte
2’b00	0xFFFFFFFF	0x0000FFFF	0x000000FF
2’b01			0x0000FF00
2’b10		0xFFFF0000	0x00FF0000
2’b11			0xFF000000

To ensure that writing memory from registers to Data Memory that are not full word length does not overwrite existing memory in the same word slot, a masking technique is used during read and write operations. This allows the Data Memory component to restrict writing or loading data within the appropriate alignment boundaries.

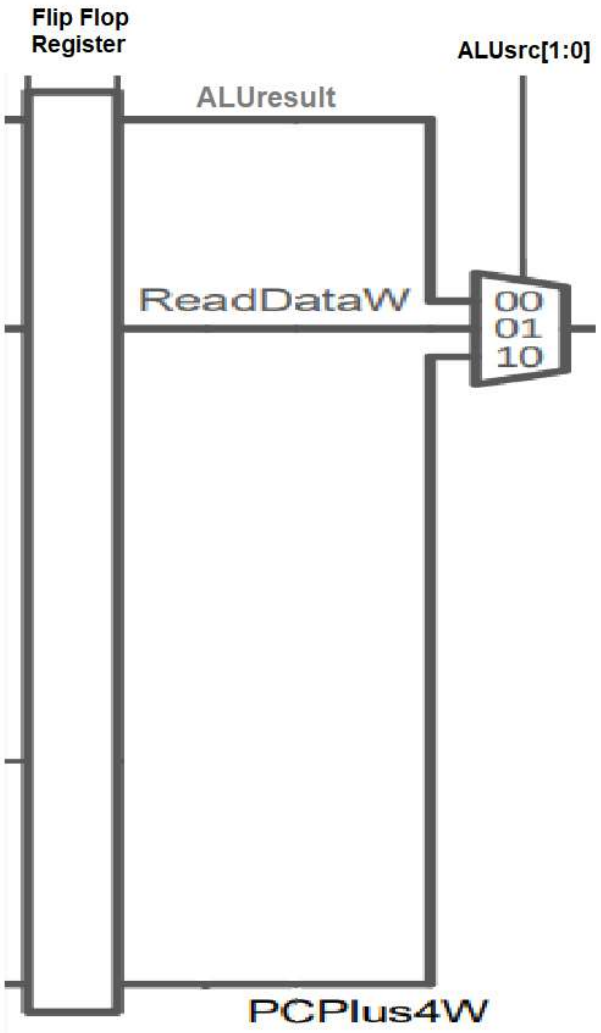
- **Write Masking:** When writing data from registers to Data Memory, the processor applies a mask to the data before storing it. This mask ensures that only the relevant bits of the word slot in memory are modified, while preserving the contents of the remaining bits.
- **Read Masking:** Similarly, when loading data from Data Memory into registers, the processor applies a mask to the loaded data. This mask ensures that only the relevant bits of the word slot in memory are extracted and loaded into the register, while discarding any extraneous bits

Write Back Stage

Similar to Lab 2 where the final component is the Result Source, the Write back stage contains the same Result Source Mux however the inputs come from registers holding the state from Data Memory, ALU Result and PC + 4. The Controller provides a 2 bit control Signal that selects output of Multipliexer from the three inputs and output it back out to the register file or the Forward multiplexer

- ALU Result: The output from the Arithmetic Logic Unit (ALU), representing the result of arithmetic and logical operations performed by the processor.
- Memory Read: Data read from the Data Memory
- PC + 4: save current Program Counter (PC) + 4 to register file

BIT	Source
2'b00	ALU result
2'b01	Data memory
2'b10	PC + 4



Hazard Unit

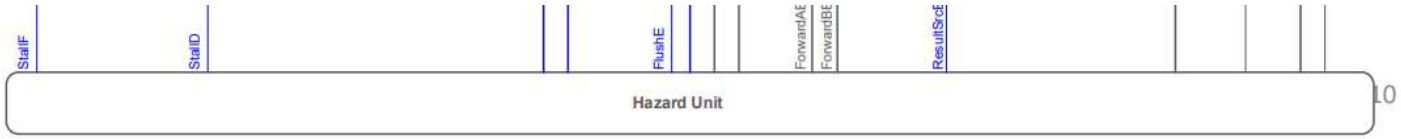
To govern the Pipelined processor, the hazard unit ensures that stages maintain the true order of the program. Certain instructions, such as Load Word and Branches, can cause "Hazards" in the pipeline processor, potentially leading to unintended register overwrites.

For branch operations, the branch condition isn't known until the execution stage. As the execution stage is third in line, the fetch and decode stage will contain instructions for both PC + 4. However, if the branch condition is met, the following "In-Flight" instruction becomes out of order. The Hazard unit recognizes this and flushes the fetch and decode stage, forcing the pipeline to restart at the fetch stage with the new PC Target. This is known as branch misprediction penalty, as two clock cycles of instructions are flushed, but necessary to prevent issues. If the branch condition isn't met, the pipeline continues without penalty.

For Load Word instructions, "In-Flight" instructions might require the data from the loaded word from memory. Therefore, the pipeline needs to be stalled to prevent "in-flight" instructions from accessing incorrect memory from the current register file state and wait until the word is loaded into the register.

The hazard unit also improves performance by allowing the pipeline to forward results from the 4th and 5th stages into the execution stage. This speeds up the processor, as "in-flight" instructions might be a sequence of instructions that are updating the same register. Instead of waiting for the instruction and stalling the pipeline for it to be written back, a few clock cycles can be saved by forwarding the results from the Memory stage to the Execution Stage or from the Writeback Stage to the Execution stage.

In Lab 3, the Hazard Unit plays a crucial role in the pipeline. It provides the necessary functionality for branching, load words, and forwarding. However, when adding new hardware to the process, which may require more clock cycles during the execution stage or any other stage, the hazard unit can then stall the pipeline to ensure that the "in-flight" instructions maintain the correct order according to the program control flow.




```
// forwarding logic
always_comb begin
   ForwardAE = 2'b00;
   ForwardBE = 2'b00;
   if (Rs1E != 5'b0)
     if      ((Rs1E == RdM) & RegWriteM) ForwardAE = 2'b10;
     else if ((Rs1E == RdW) & RegWriteW) ForwardAE = 2'b01;

   if (Rs2E != 5'b0)
     if      ((Rs2E == RdM) & RegWriteM) ForwardBE = 2'b10;
     else if ((Rs2E == RdW) & RegWriteW) ForwardBE = 2'b01;
end

// stalls and flushes
assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
assign StallD = lwStallD;
assign StallF = lwStallD;
assign StallE = 0;
assign StallM = 0;
assign StallW = 0;
assign FlushD = PCSrcE;
assign FlushE = lwStallD | PCSrcE;
assign FlushM = 0;
assign FlushW = 0;
```

Pipelined Microarchitecture

With all these components integrated, the pipelined CPU becomes a versatile tool capable of handling a wide range of instructions and operations efficiently. Similar to Lab Two, the CPU operates continuously until it encounters an ECALL instruction, which serves as a halt signal for the processor.

To validate the correctness of the memory file (.memfile), we inspect the last register written to. If the value written is 0x0000000A, it indicates the successful completion of all tests. Conversely, if the value written is 0x00000011, it signifies a failed test.

The tests consist of various instruction sets, each interdependent on the others to complete the entire test suite. By executing these tests, the CPU showcases its ability to effectively handle a diverse array of instructions and operations, thereby affirming its functionality and performance in a pipelined architecture

Performance

With a pipeline processor, performance considerations become more nuanced compared to a single-cycle CPU. In a single-cycle CPU, the minimum time needed for the clock period is determined by the critical path from the PC counter register to the Result Multiplier. However, in a pipeline CPU, each stage executes in one clock cycle. This implies that an instruction requires multiple clock cycles to traverse the processor, depending on the number of stages. In this lab, with five stages, it would necessitate four cycles from the fetch to write-back points.

However, since each stage is segmented, the minimum clock period is only constrained by the maximum of the critical path through stages. Consequently, with a pipeline CPU, a smaller clock period can be achieved, resulting in faster clock speeds. But because the pipeline introduces complexities such as branch misprediction and stalling, this will lead to an average Cycle Per Instruction (CPI) greater than one. This means that, on average, more than one clock cycle is required to complete an instruction, impacting overall performance.

$$T_{c\_pipelined} = \max \text{ of}$$

$t_{pcq} + t_{mem} + t_{setup}$	**Fetch**
$2(t_{RFread} + t_{setup})$	**Decode**
$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$	**Execute**
$t_{pcq} + t_{mem} + t_{setup}$	**Memory**
$2(t_{pcq} + t_{mux} + t_{RFwrite})$	**Writeback**

Pipeline processor critical path

National Instruments Elvis III Board

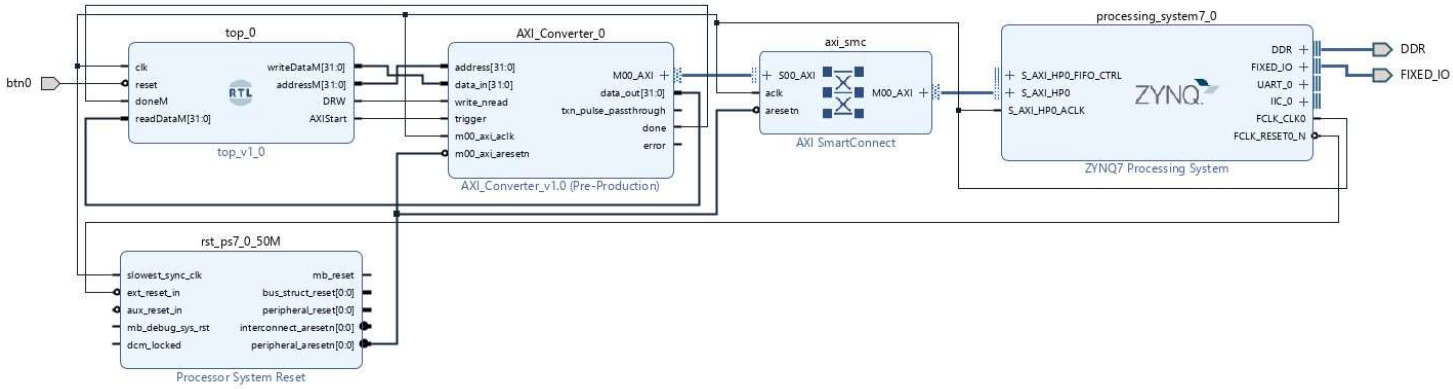
Part of the project involved deploying the SystemVerilog code onto the Elvis III board, configuring the logic equivalence gates onto an FPGA board. This setup aimed to utilize the onboard RAM to simulate the processor on actual hardware, offering a more realistic environment than software simulation. However, this process encountered issues similar to those in Lab 2.

To prepare the SystemVerilog program for the FPGA Board, several adjustments were necessary. Firstly, the Top module utilized in the simulation for Questa had to be replaced with the provided Top.v file. Additionally, the risv module in the file needed to be modified to accept two new control signals: PCReady and MemStrobe. These signals aided in controlling Memory Control on the Elvis Board, as the board utilized onboard RAM instead of simulated memory.

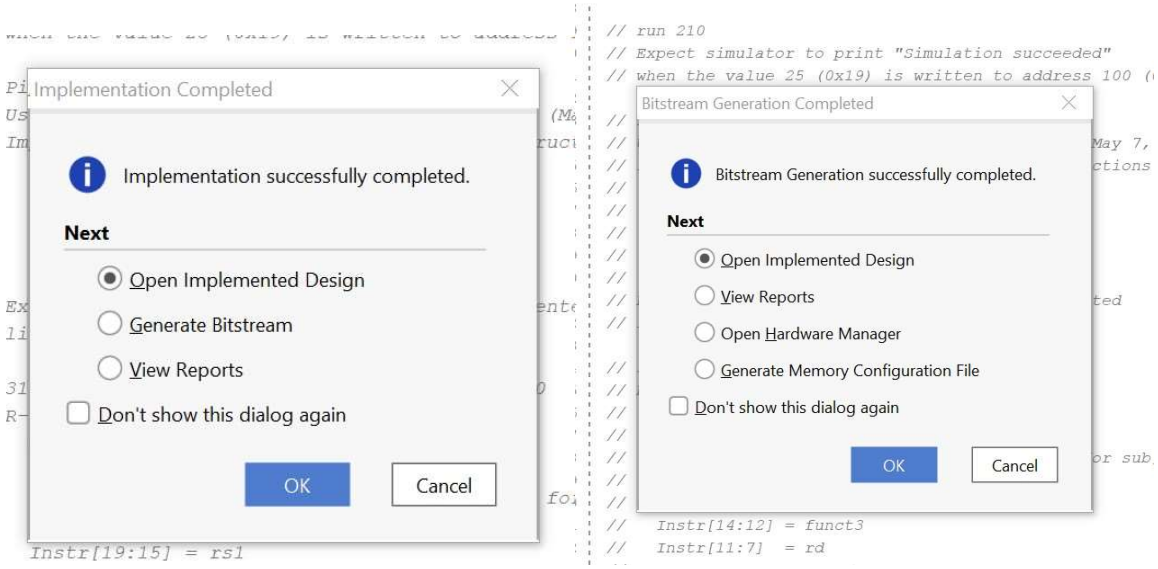
```
(* mark_debug = "true" *) wire [31:0] PC;
(* mark_debug = "true" *) wire [31:0] Instr;
                          wire        MemWrite;
                          wire        PCReady;
                          wire        MStrobe;
                          wire [2:0]  Funct3M;

// instantiate processor and memories
riscv riscv (.clk(clk),
                 .reset(reset),
                 .PCF(PC),
                 .InstrF(Instr),
                 .MemWriteM(MemWriteM),
                 .ALUResultM(addressM),
                 .WriteDataM(writeDataM),
                 .ReadDataM(readDataM),
                 .MemStrobe(MStrobe),
                 .PCReady(PCReady),
                 .funct3M(Funct3M));
```

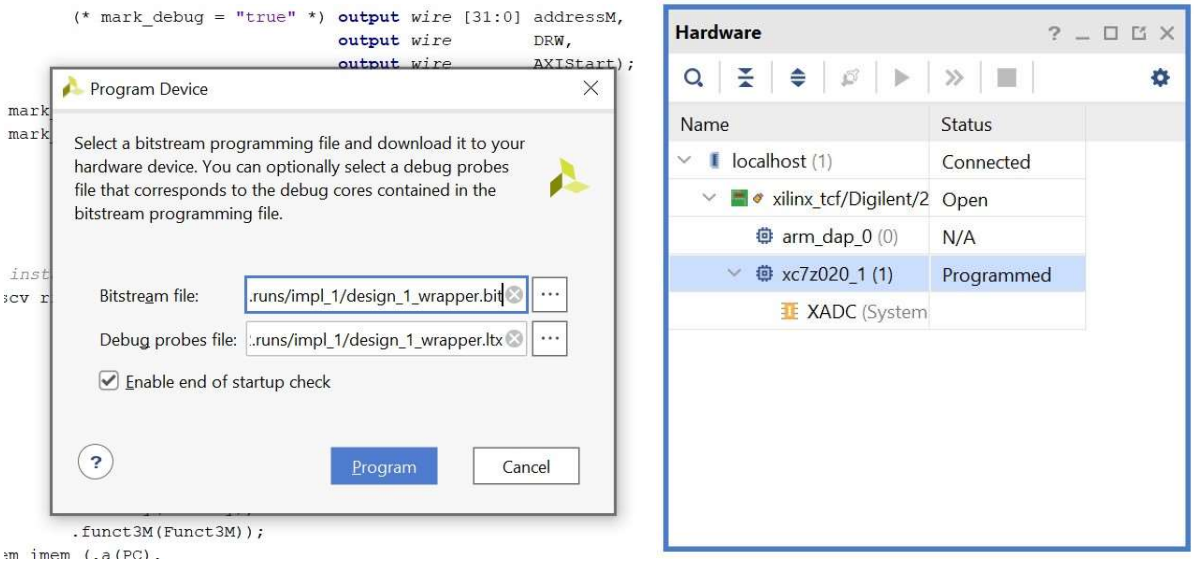
These modifications were already incorporated into the design, with the Top and testbench modules removed from the riscv_pipelined.sv file, which now contained the necessary inputs for Top.v. However, Top.v required changes to accept input from the risc module, as funct3 needed to be routed to the memcontroller.v file to accommodate different types of load and store instructions.



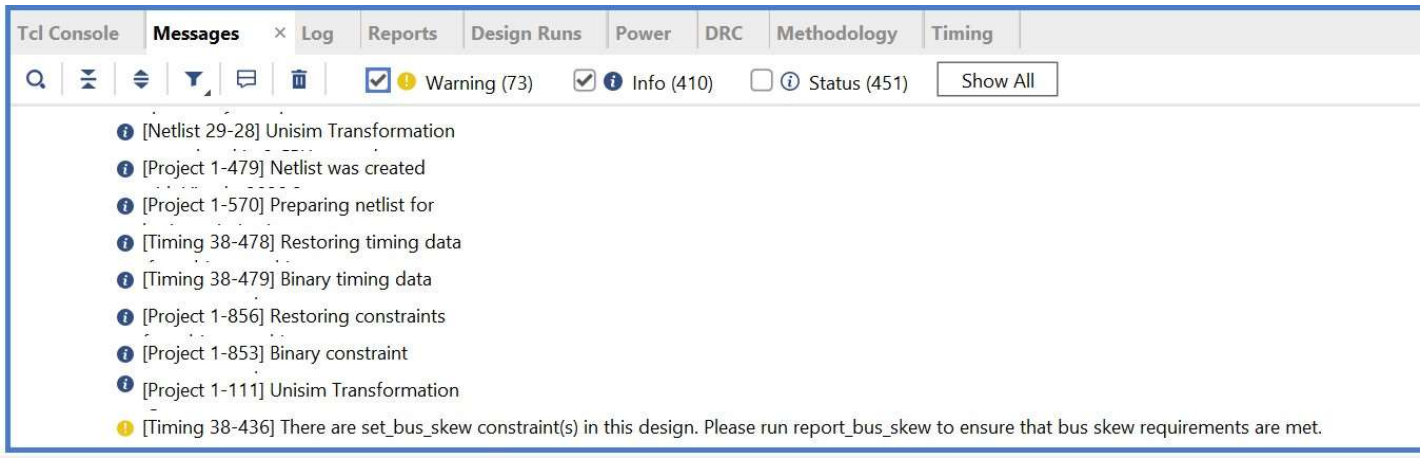
These modifications were saved in a separate file from the one simulated and imported into the Vivado software. After everything was uploaded, we could implement the design on the board. The synthesis, implementation, and bitstream generation were successful, allowing us to proceed with programming the board.



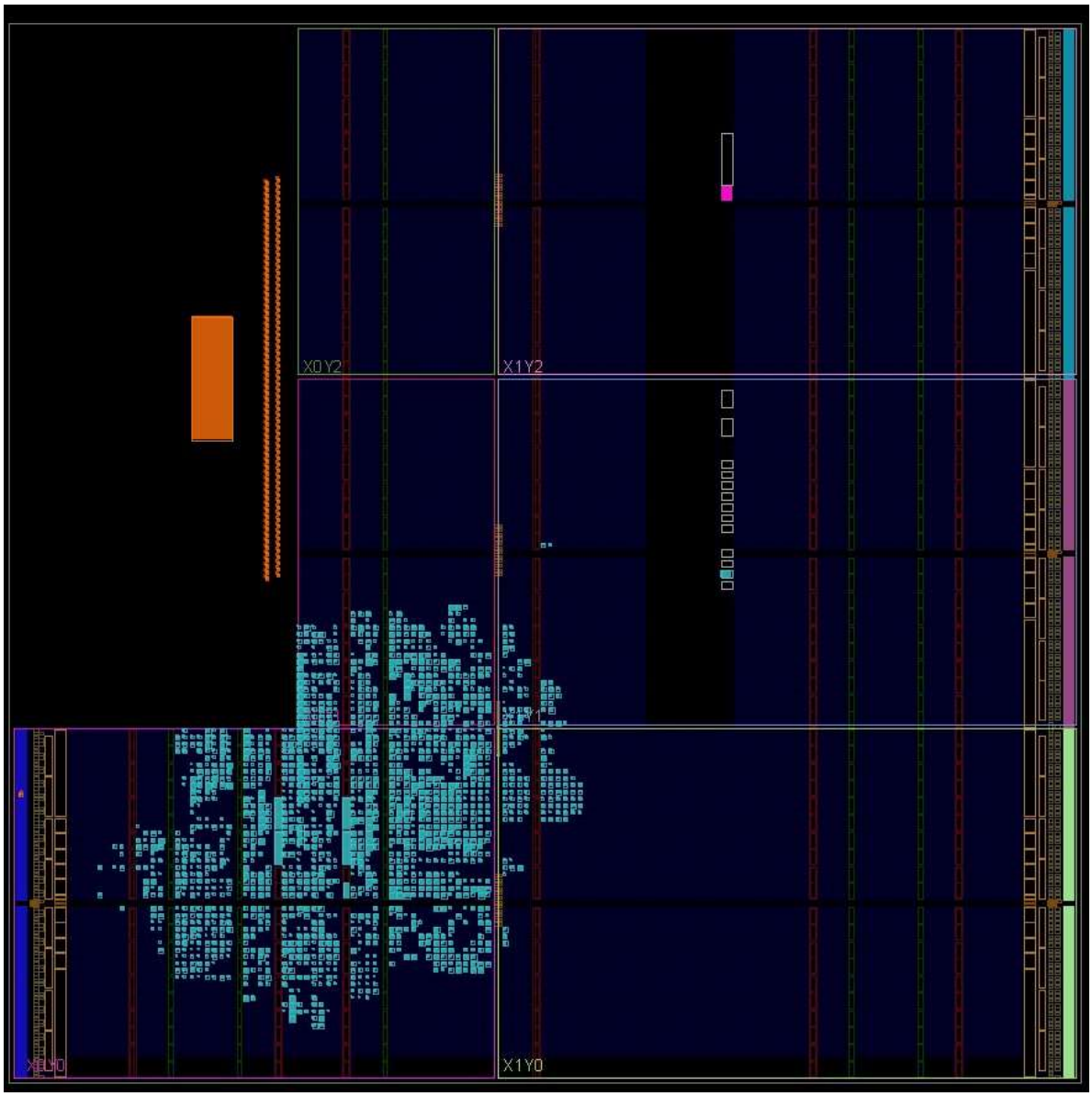
Unlike in Lab 2, the bitstream was successfully generated, and programming the board was completed without any issues. However, we encountered difficulty in getting the board to output anything on the computer. According to the instructions for using the Elvis Board, holding down Button-0 while programming and successfully programming the board should prompt a signal analyzer view, which would allow pressing the play button. However, in our case, this view or play button never appeared.

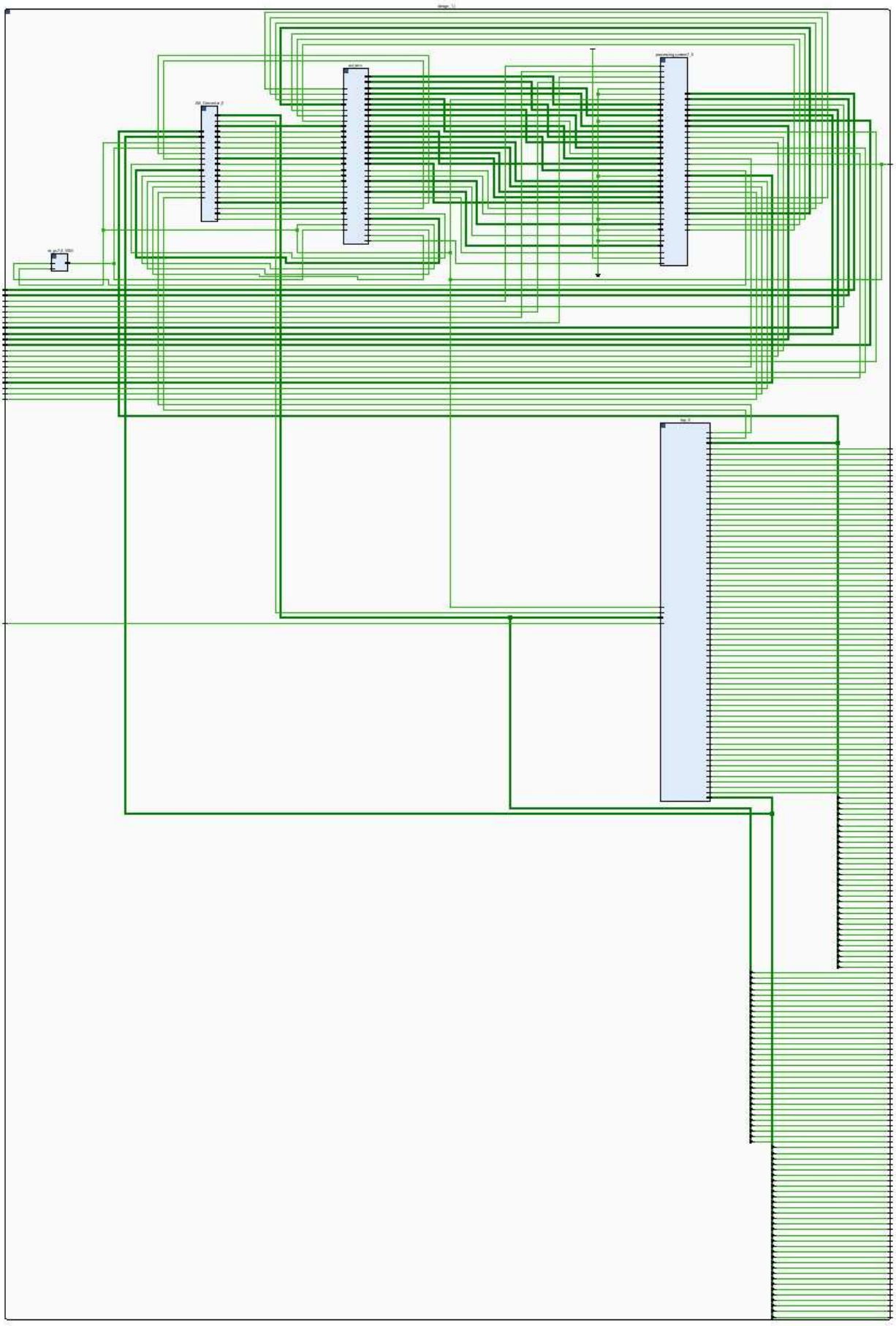


As a result, we were unable to view the output stream of the board, even though the board was successfully programmed.



Message Window, No error after programmed





FPGA Schematic

Conclusion

Lab 3 builds upon the foundation laid by the single-cycle CPU, demonstrating the capabilities and advantages of a pipeline processor. While dividing the processor into stages might initially seem to slow down the total execution time due to the need for more clock cycles to push an instruction through the processor, implementing the pipeline process in SystemVerilog reveals the significant improvements possible. By allowing a hazard unit to control the flow of data across the pipeline and parallelizing multiple instructions "in-flight," a pipeline processor can outperform a single-cycle CPU.

Expanding upon the existing code posed challenges, requiring the integration of new components and enhancements to ensure the seamless functionality of all 37 instruction sets. This involved incorporating two new instructions to fully utilize the capabilities of the 4-bit ALU control. Additionally, the pipeline processor required additional hardware to ensure the correct functioning of the stages. Some stage registers would require a 178 bits wide flip-flop, enabling instructions "in-flight" to retain their states as the processor cycles.

Incorporating the hazard unit also highlighted challenges related to memory. In a single-cycle CPU, memory can become the slowest part of the critical path. However, in the pipeline, the critical path shifts to the execution stage. While reading and writing from memory in a pipeline processor isn't as problematic, issues arise when loaded memory is needed within an "in-flight" instruction, requiring the processor to stall.

Through this lab, we've gained a clearer understanding of how a processor interprets instructions from memory and executes a sequence of operations, including loading and storing data, to efficiently run functional programs. Overall, this experience has deepened our comprehension of processor architecture and operation, providing valuable insights into fundamental computer systems concepts.

However, despite successfully programming the FPGA board, we encountered Issues preventing the expected output. The signal analyzer view, crucial for verifying the board's functionality, failed to appear as anticipated, highlighting the complexities of hardware integration and the need for further troubleshooting and refinement

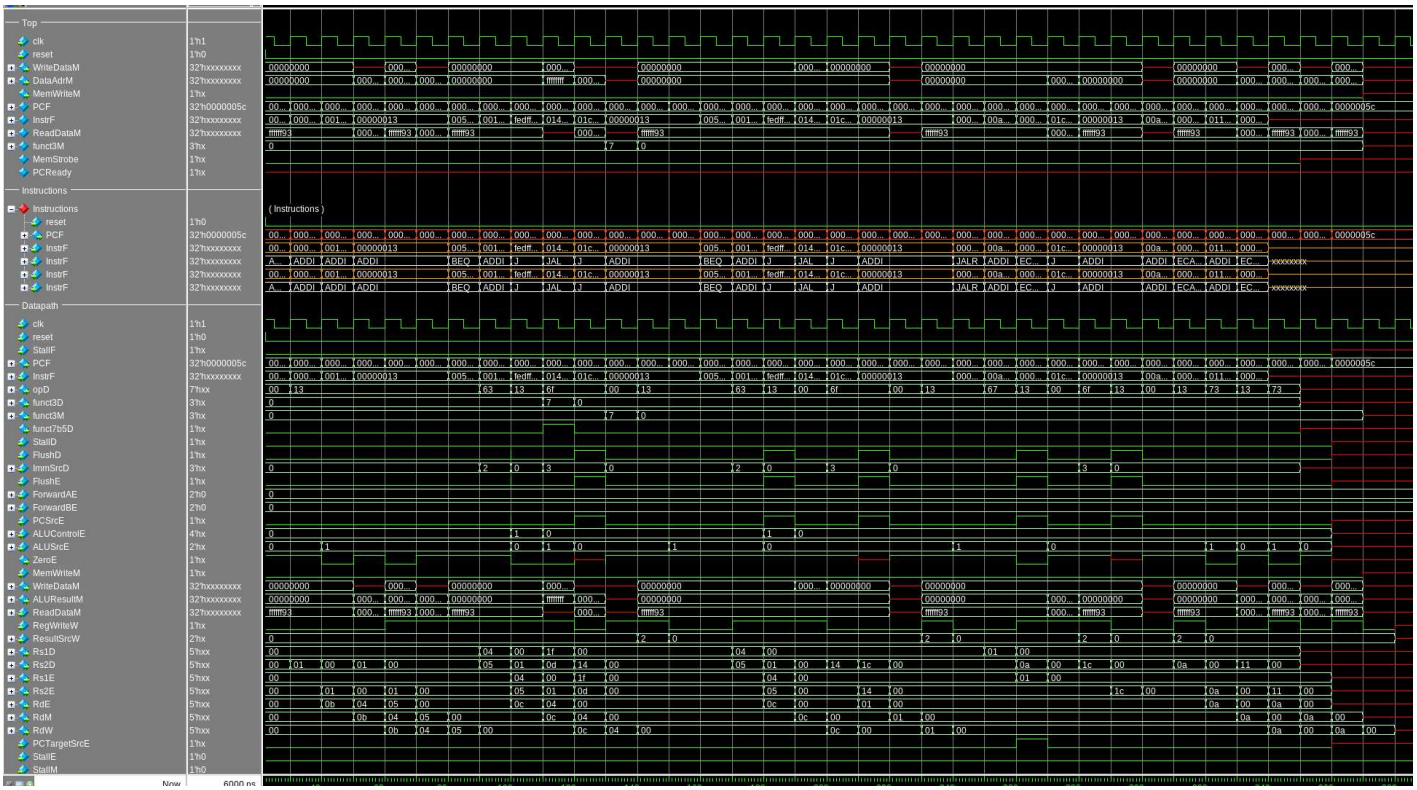
Questa Simulation

To validate the correctness of the pipeline processor, Questa can be utilized for hardware implementation simulation. Memfiles, which contain instructions, are loaded into the simulation environment to run individual tests for each instruction and ensure proper functionality.

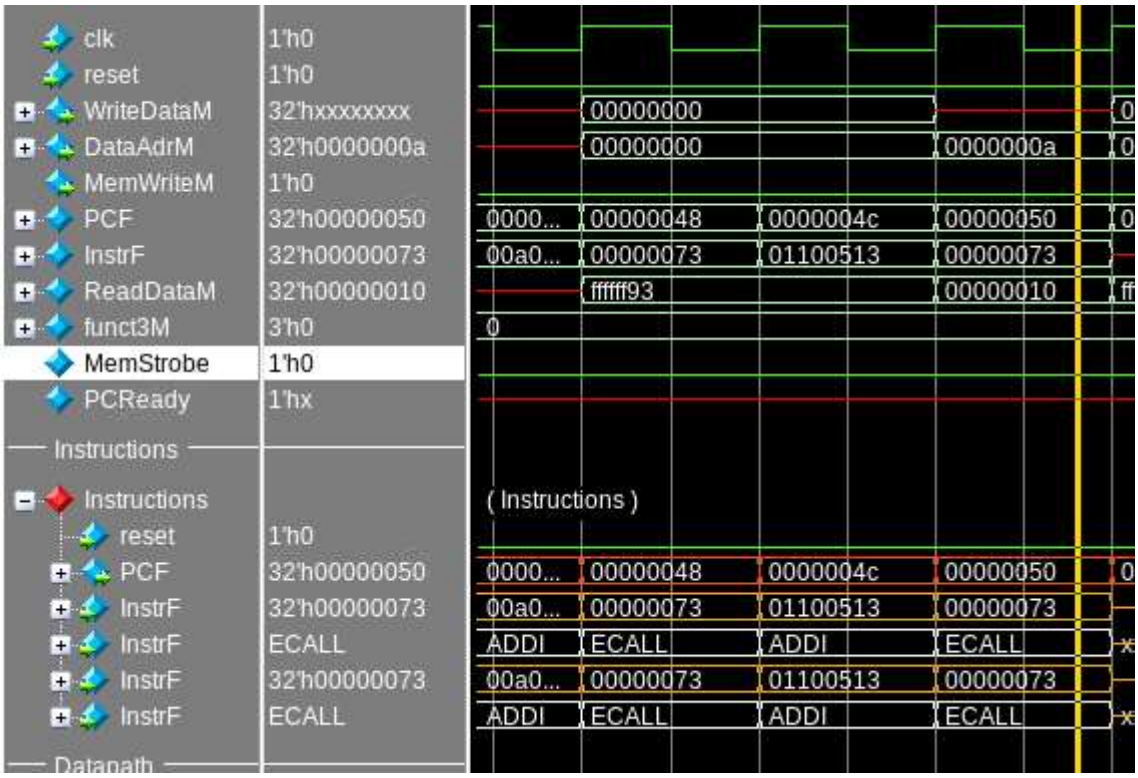
To confirm the successful completion of a memfile, two checks can be performed. First, reading the register to verify if 0x0000000A was written to it after the program's completion indicates a successful test. Alternatively, a simple verification can be conducted by confirming if ECALL was called twice.

ECALL returns control to the operating system. As mentioned earlier, because the branch condition isn't determined until the execution stage, there are two possible final paths, resulting in either a pass or fail outcome. ECALL is invoked in both scenarios, with the difference being the value written to the register, either 0x0A or 0x11.

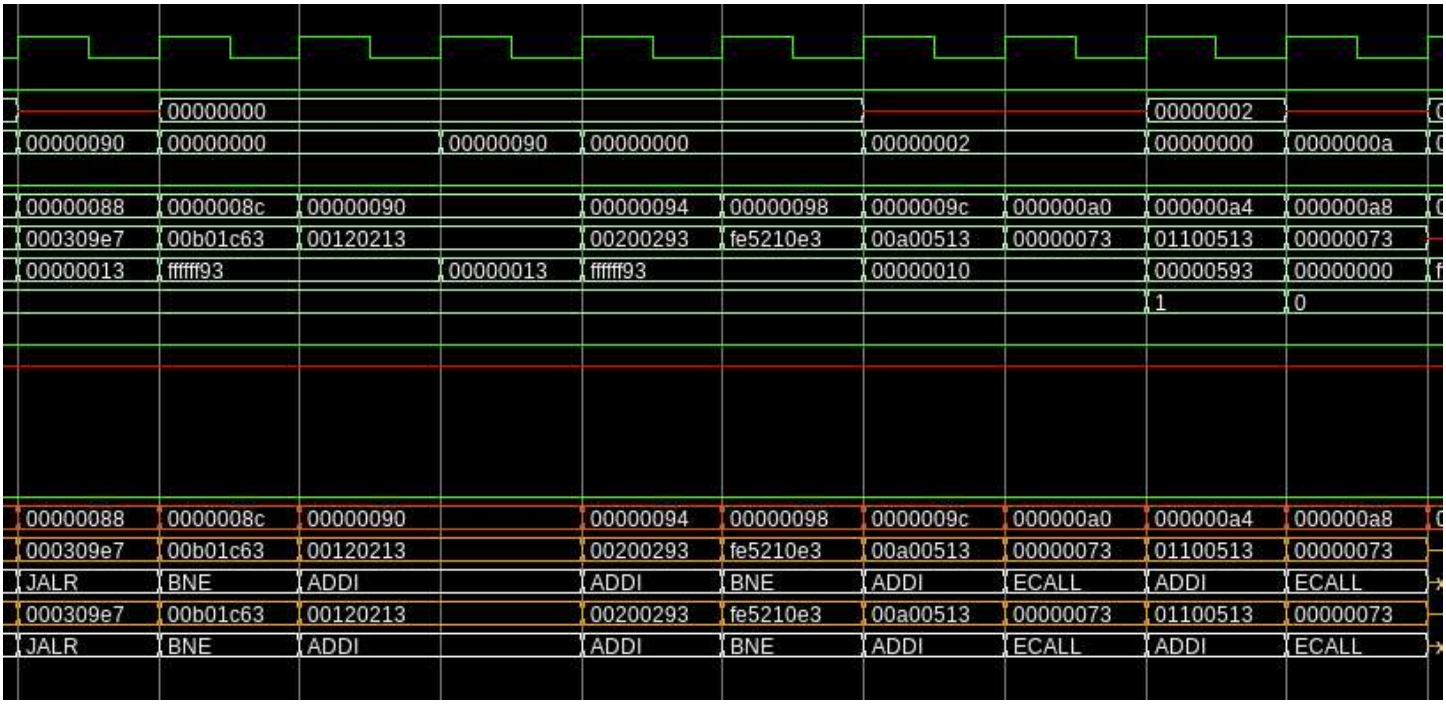
In both cases, ECALL is encountered and processed before the branch prediction is concluded. This allows us to observe two ECALL instructions "In-Flight" and also highlights the issue of "Branch Misprediction Penalty."



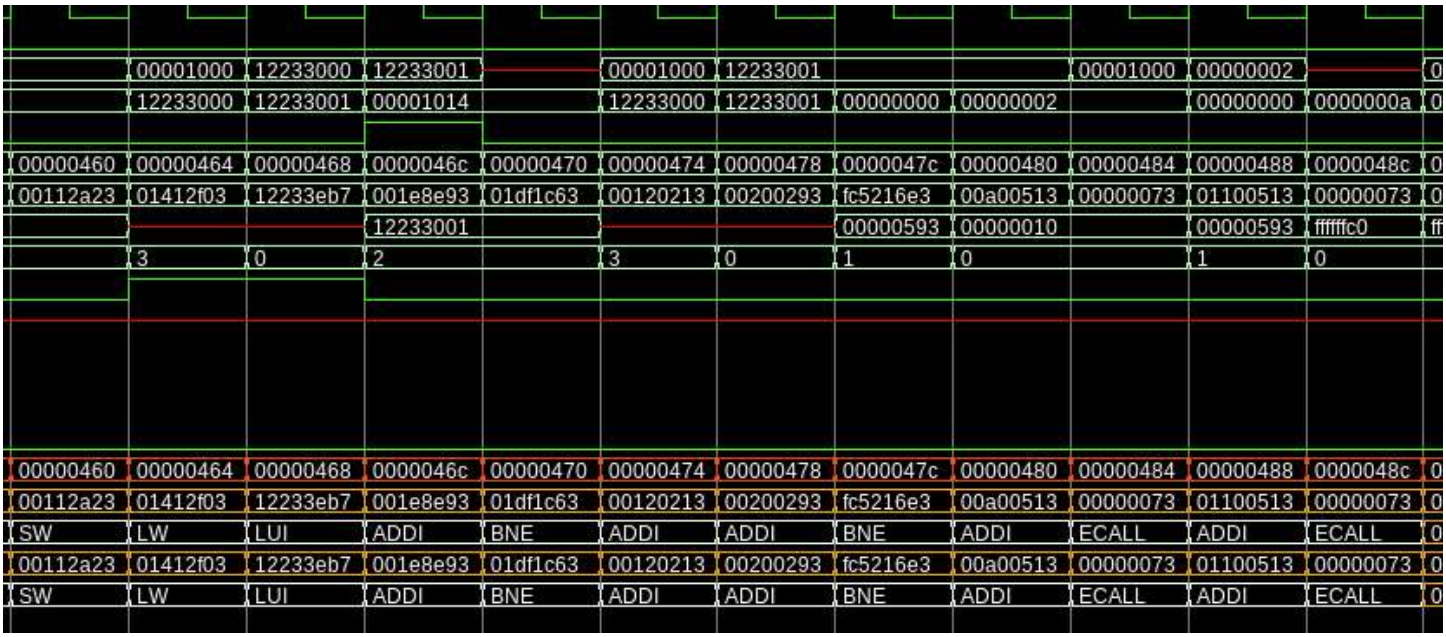
Add Upper Immediate PC auipc.memfile



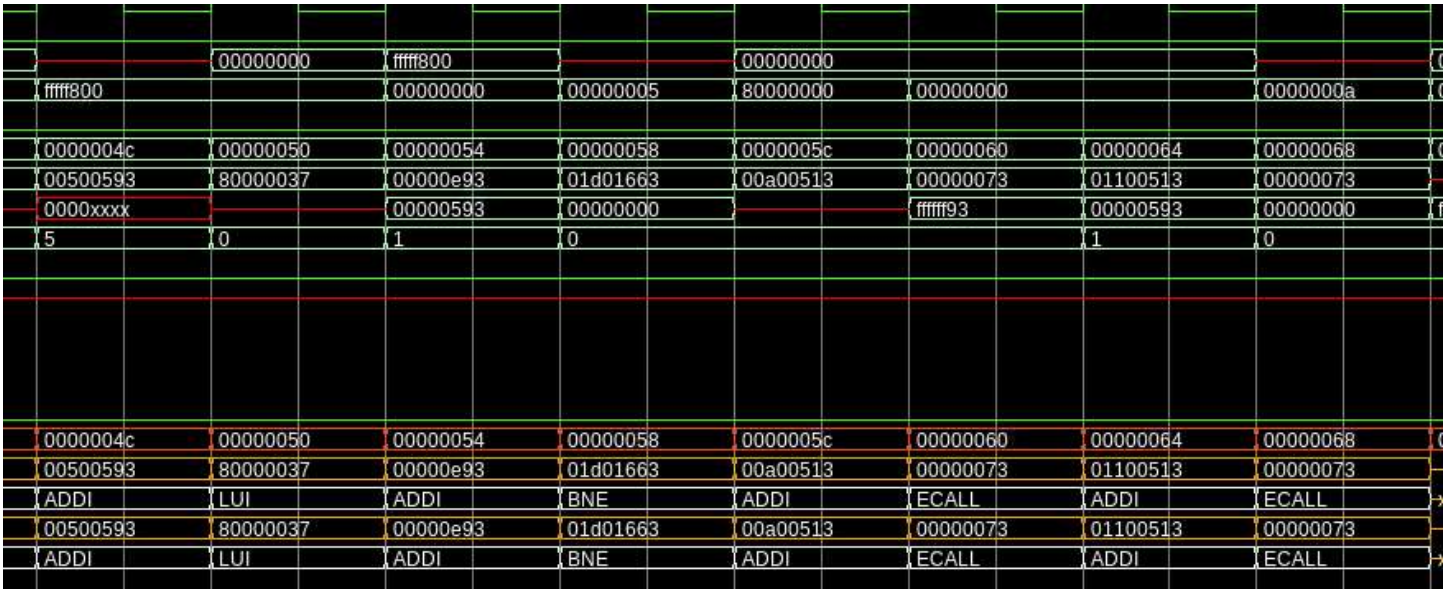
auipc.memfile – Double ECALL with register write of 0x0000000a



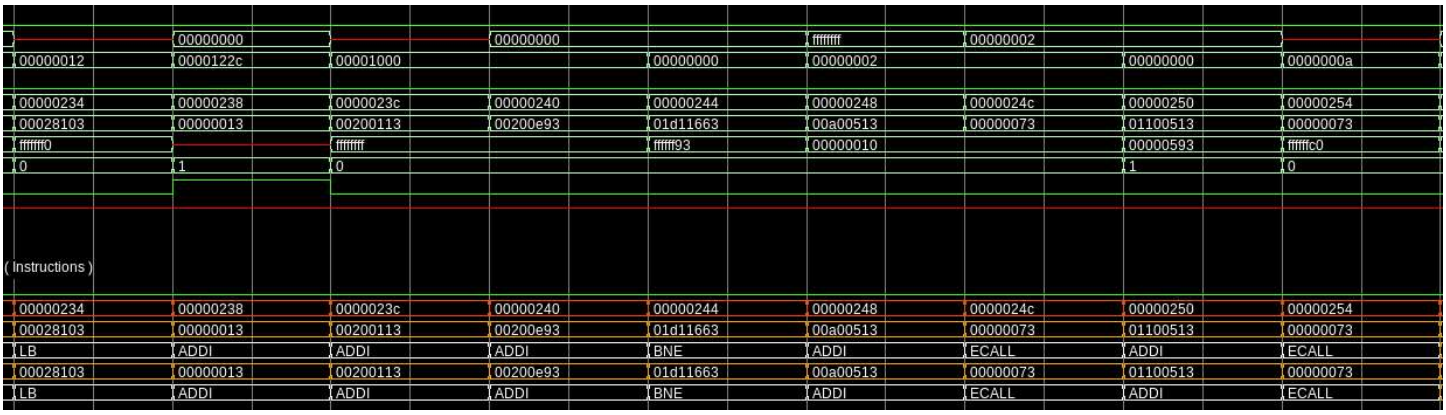
Jump and Link Register jalr.memfile



load word lw.memfile



Load Upper Instruction lui.memfile



Load Byte lb.memfile

How to Run

File included has two folders, one containing the file used for simulation other used for Vidado. Files for Vidado won't run in Questa as necessary modules will be missing

Within the folder for Questa (SystemVerilog Questa simulation) will contain the riscv_pipelined.do and necessary riscv_pipelined.sv and memfiles

To running the simulation in terminal type:

vsim – do riscv_pipelined.do

to modify the memfile being run. uncomment the memfile in testbench module want to run, comment out the others

```
module testbench();

   logic        clk;
   logic        reset;

   logic [31:0] WriteData, DataAdr;
   logic        MemWrite;

   // instantiate device to be tested
   top dut(clk, reset, WriteData, DataAdr, MemWrite);

   initial
     begin
        string memfilename;
        // memfilename = {"programs/riscvtest.memfile"};
                // memfilename = {"programs/fib.memfile"};
                // memfilename = {"programs/bne-test.memfile"};


        // memfilename = {"testing/add.memfile"};    // works
        // memfilename = {"testing/addi.memfile"};   // works
        // memfilename = {"testing/and.memfile"};        // works
        // memfilename = {"testing/andi.memfile"};       // works
        // memfilename = {"testing/auipc.memfile"};      // works
        // memfilename = {"testing/beq.memfile"};        // works
        // memfilename = {"testing/bge.memfile"};        // works
        // memfilename = {"testing/bgeu.memfile"};       // works
        // memfilename = {"testing/blt.memfile"};        // works
        // memfilename = {"testing/bltu.memfile"};       // works
        // memfilename = {"testing/bne.memfile"};        // works
        // memfilename = {"testing/jal.memfile"};        // works
        // memfilename = {"testing/jalr.memfile"};   // works
         memfilename = {"testing/lb.memfile"};   // works
        // memfilename = {"testing/lbu.memfile"};        // works
        // memfilename = {"testing/lh.memfile"};         // works
        // memfilename = {"testing/lhu.memfile"};    // works
        // memfilename = {"testing/lui.memfile"};    // works
        // memfilename = {"testing/lw.memfile"};     // works
        // memfilename = {"testing/or.memfile"};         // works
        // memfilename = {"testing/ori.memfile"};        // works
        // memfilename = {"testing/sb.memfile"};         // works
        // memfilename = {"testing/sh.memfile"};         // works
        // memfilename = {"testing/sll.memfile"};        // works
        // memfilename = {"testing/slli.memfile"};       // works
        // memfilename = {"testing/slt.memfile"};        // works
        // memfilename = {"testing/slti.memfile"};   // works
        // memfilename = {"testing/sltiu.memfile"};  // works
        // memfilename = {"testing/sltu.memfile"};       // works
        // memfilename = {"testing/sra.memfile"};        // works
        // memfilename = {"testing/srai.memfile"};   // works
        // memfilename = {"testing/srl.memfile"};        // works
        // memfilename = {"testing/srli.memfile"};       // works
        // memfilename = {"testing/sub.memfile"};        // works
        // memfilename = {"testing/sw.memfile"};         // works
        // memfilename = {"testing/xor.memfile"};        // works
        // memfilename = {"testing/xori.memfile"};       // works
        $readmemh(memfilename, dut.imem.RAM);
        $readmemh(memfilename, dut.dmem.RAM);
     end
```