Pierce Clark and Max Conine
System on Chip Design 2026
4/30/26

System-On-Chip Design Final Project Report

Abstract

For the 2026 System-On-Chip Design final project, we transformed a single cycle processor into a capable pipelined processor. We optimized our processor by reducing the critical path area, optimizing hazarding, and reducing overall size. The processor achieved a score ten times larger than the original processor, with an execution time 62.4% faster. The project gave the team experience in the process of SOC development. In this report, we will cover processor optimizations, then discuss results and review the progression timeline of the project.

Microarchitecture Specification

Our design features a 5-stage processor microarchitecture based on the RV32IM instruction set, incorporating a dedicated 3-stage multiplication unit and comprehensive hazard unit logic. The design is inspired by the pipelined processor presented in “Digital Design and Computer Architecture, RISC-V Edition” by David and Sarah Harris as well as the design of the Wally RISC-V Processor.

Top Level Block Diagram

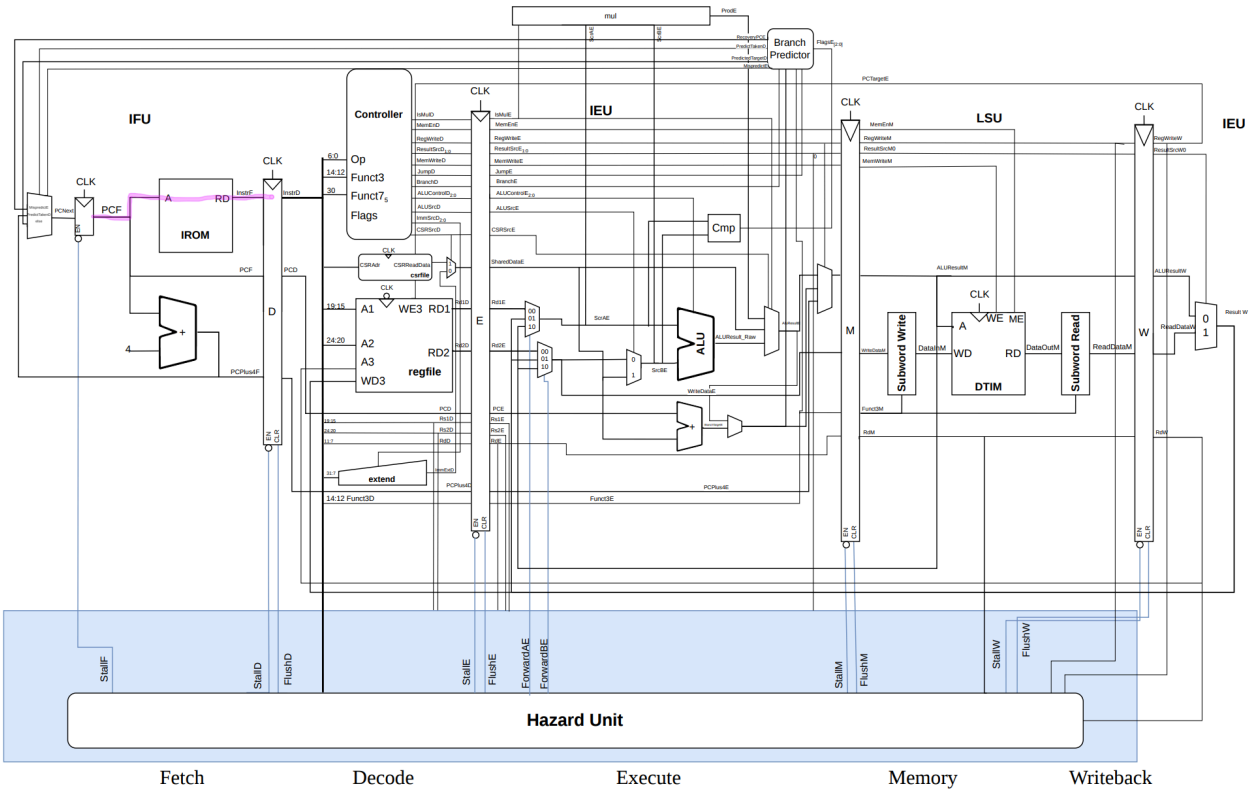

Fig. 1: Top Level Block Diagram with critical path annotated

Processor Optimizations

Pipelining

Our processor is a five stage pipelined processor that increases throughput by breaking instruction execution into multiple, concurrent stages. Our stages include Fetch, Decode, Execute, Memory, Writeback. The pipeline cuts up instructions into multiple stages, which reduces the critical path. Our initial pipeline efforts reduced the 8.24 ns critical path of the single cycle processor to 5.10 ns. Increasing the amount of pipeline stages to six was considered, however due to the slow memory speed, it was impractical.

Fast Mul unit

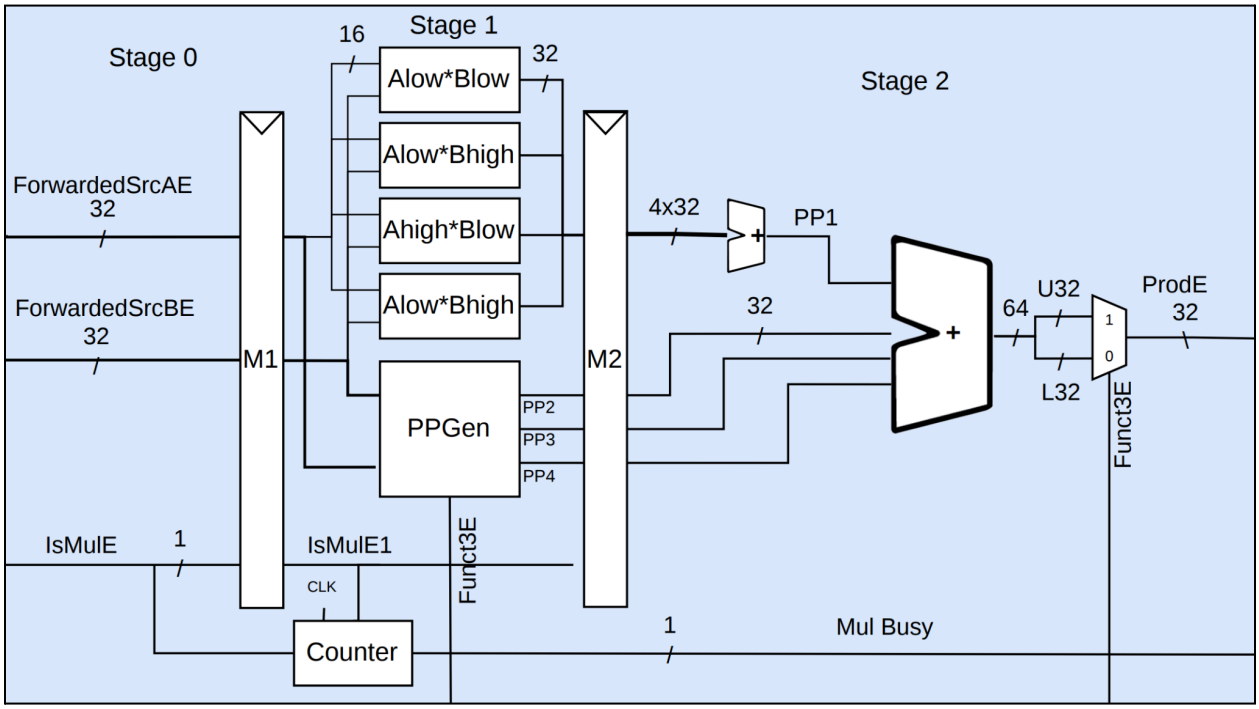

Fig. 2: 3 Stage Multiply unit

Our multiply unit is a modified version of the Wally multiply unit. In the first stage, it calculates the 4 parts of the PP1 by doing 4- 16x16 multiplication calculations. Those 4 are then added together in stage 2 to get the full PP1. All partial products 1-4 are added together in stage 2. The 64 bit results are then mixed to get the lower or upper half based on the instruction. The M1 register keeps the partial product calculation off the critical path. Although it adds two extra cycles to every multiplication, the critical path is much shorter which makes up for the extra cycles. When a multiple enters, the module starts a counter to wait two cycles, then resets. While the mul unit is busy, all other stages are stalled which preserves our hazarding logic. The code for the multiply unit can be found in Appendix A.

Branch Logic

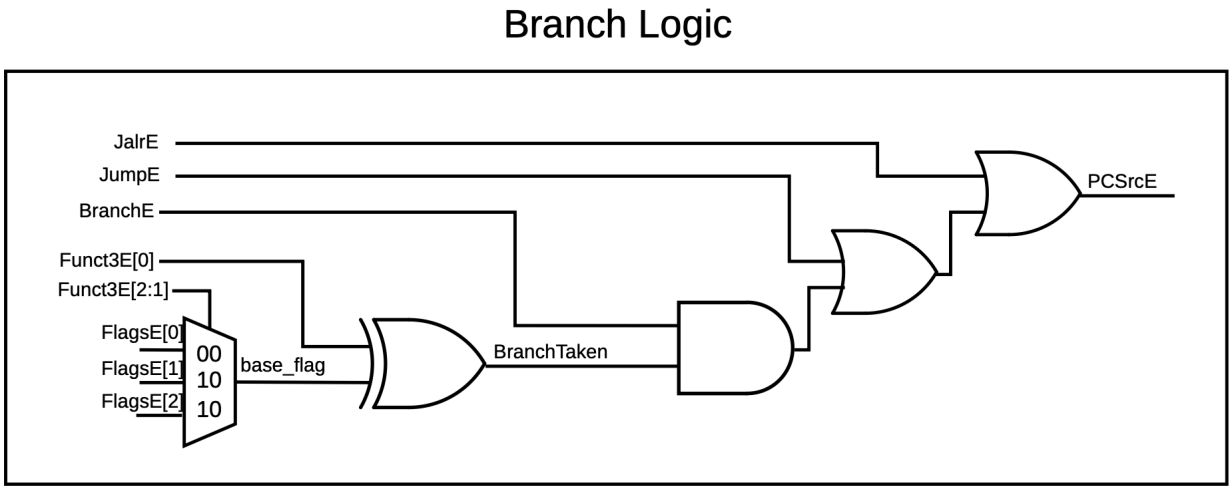

Fig 3. Processor Branch Logic

The branch logic computes whether the PCNext should be PCPlus4 or ALUResultM. It was optimized by exploiting the bit-level symmetry of the RISC-V Funct3 field. Rather than implementing a standard 6-way multiplexer to evaluate each branch individually, we used a factored logic approach to reduce the area and delay of the PCSrcE signal.
The RISC-V ISA encodes the Funct3[2:1] bits as the comparison operation while Funct3[0] encodes the polarity.

Funct3[2:1]	Selected Base Flag	Instructions Covered
00	FlagsE[0] (Zero)	BEQ, BNE
10	FlagsE[1] (Less Than)	BlT, BGE
11	FlagsE[2] (Less Than Unsigned)	BLTU, BGEU

We use an XOR gate as a conditional inverter to handle both “true” and “inverted” conditions. For example, if Funct3E[0] is 0 (e.g., BEQ), the output is just the base_flag. If Funct3E[0] is 1 (e.g., BNE), the output is !base_flag. This logic encoding reduces the area of the processor from ~60 transistors in a 6-way Mux to ~36 transistors in for 3-way Mux + XOR gates.

Optimized Synthesis

We optimized the synthesis of our processor by adjusting settings and applying flags. We applied the maxopt flag, and a target clock speed of 3ns. We also constrained the synthesis to try to get the smallest area possible, since we effectively hit our limit of 3ns speed due to path length.
Since our memory has a speed of 3ns, our critical path cannot be less than this. We gave the synthesis a goal frequency of 333 MHz to try to exactly hit the target.

```
if { $maxopt == 1 } {
   ungroup -all -simple_names -flatten
   compile_ultra -retime
   optimize_registers
```

The max opt flag provides multiple different capabilities to the design compiler. The three abilities are described below:

ungroup -all -simple_names -flatten: Removes all module hierarchy and flattens the entire design into a single flat netlist before compilation. This gives DC maximum freedom to optimize across module boundaries
compile_ultra -retime: Uses the full suite of optimization algorithms including datapath optimization, constant propagation, and logic restructuring. Allows the design compiler to move registers across combinational logic to balance path delays.
optimize_registers: A second pass that runs after compile_ultra specifically focused on register placement. This includes moving registers to reduce setup time, merging redundant registers, and pipelining registers more efficiently across the flattened netlist.

A copy of the real synthesis flags used in the final processor can be found in Appendix B.

Minimizing Area

To reduce the area of the processor, we focused on removing registers. An unoptimized synth of a 32 bit register is 1300 μm^2, which is around a percent of our processor's area. Thus each removal represents a large improvement in score. There are two areas we focused on during this optimization.

Firstly, we lowered the area of the control and status register by removing unnecessary registers. Our initial process had registers that tracked the amount of each instruction (R-type, Jumps, Branches, Loads, etc.). We removed everything except instructions retired and clock cycles. This lowered the size of the CSR file by 50000 μm^2, which is a significant improvement.

Secondly, we also found how to compress multiple pipelined signals into one. We found that we were writing back signals PC+4 and PCtarget. However we found we could route this into ALUresult, then select the ALU result in the writeback mux. This removed 4 32-bit registers and reduced the size of the writeback mux. We also realized we can mux the CSRReadData with ImmExtD with CSRSrcD control signal select to a SharedDataE signal, which cuts down another 32 bit register.

Another interesting way we reduced the area is by removing the (RdE != 5'b0) check from our hazard unit to force it not to stall if we're writing to r0 which bumped up our score by 1%.

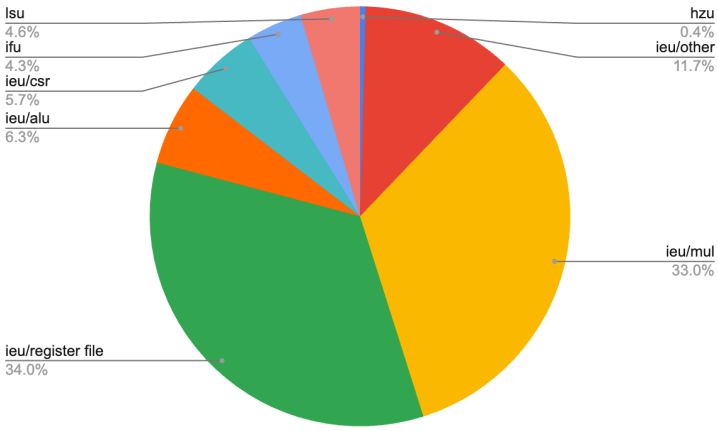

Fig 4. Per Module Area Break of the processor.

We found the register file and multiply unit take up a majority of the processor area as shown in the figure above.

Hazarding

We detect data hazards and load hazards. The hazard unit flushes when a branch is taken or a load introduces a bubble. We optimized our hazard unit to only stall or flush when necessary in order to reduce the total number of cycles. Our hazard unit stalls the Execute, Write and Memory stages if there is a multiplication in progress. Hazard (hazard.sv) unit verilog code can be found in the appendix.

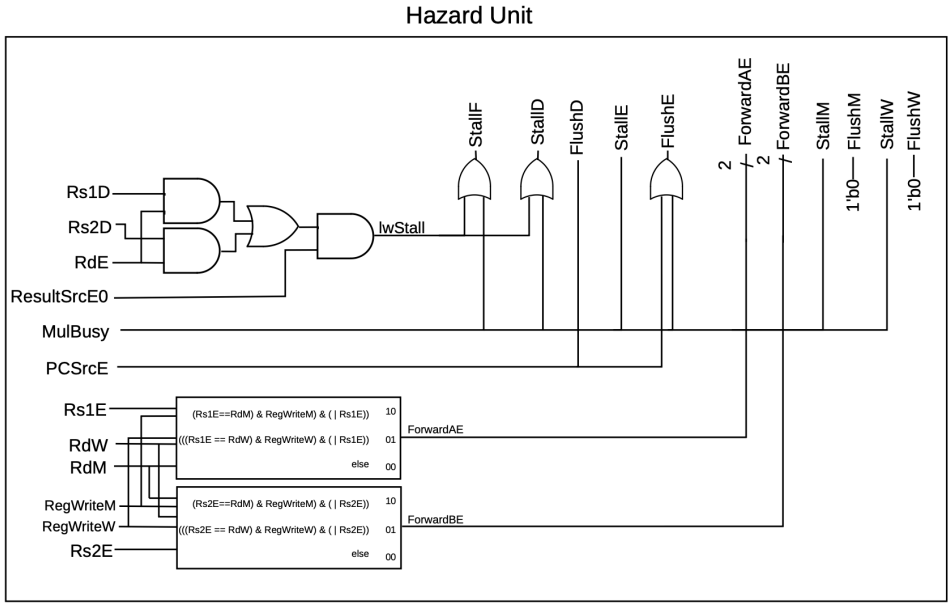

Figure 5. Hazard Unit Logic

Branch Prediction

We experimented with branch prediction and implemented a Branch Target Buffer (BTB) and 2-bit branch predictor into our processor after the second checkpoint with a goal of decreasing our total cycles. We implemented the BTB in the Fetch stage of our processor as shown in the diagram below.

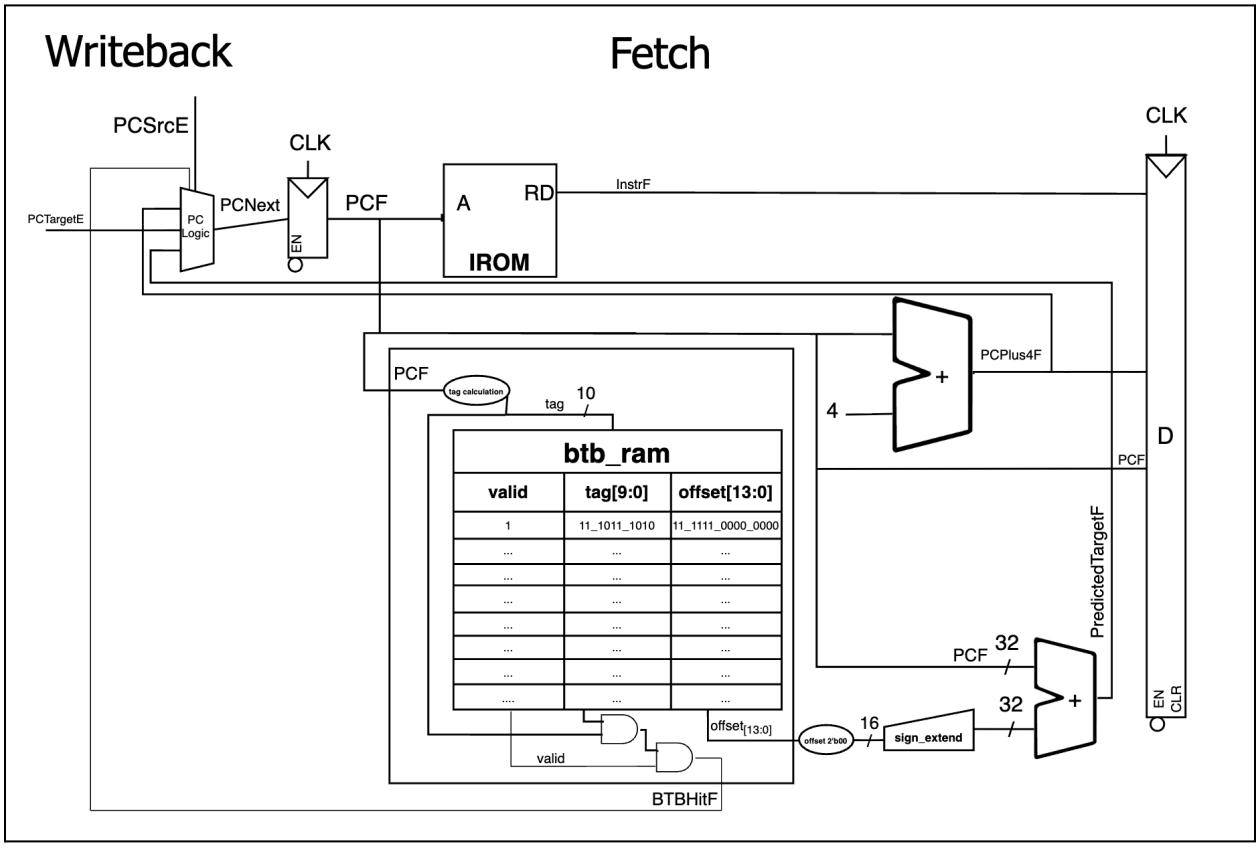

Figure 6. Unused Branch Target Buffer

The BTB works by taking PCF, generating a 10 bit tag to check the btb_ram with. If the tag matches the one already in the btb_ram and is valid, it produces a BTBHitF signal which toggles the PC logic to make the next instruction be the predicted target address. The predicted target address generated by shifting offset over by 2'b00 and sign extending it to 32 bits. Then this offset is added to the PCF to generate the predicted target address. We can change the size of the btb_ram and the size of the tag and how the tag is constructed to optimize performance. Currently the 3 bit index is generated by PCF[4:2] and the 10 bit tag is PCF[14:5]. With a few synthesis runs, we collected the Coremark/MHz score and FOM for each size of the btb_ram depending on the number of index bits.

Number of index bits	Coremark / MHz score	FOM
2 (4 slots)	2.88	3,749
3 (8 slots)	2.95	4,085.3
4 (16 slots)	3	4,000
5 (32 slots)	3.03	3227

Unfortunately while adding the BTB did increase our Coremark/MHz score, it did come at some area costs. The total area of the processor with the BTB and 2-bit predictor implemented was 214931 μm^2 and the critical path was 3.170150 ns. We found that the branch_buffer area was 6.7% of our total processor area. The critical path and area makeup are shown below.

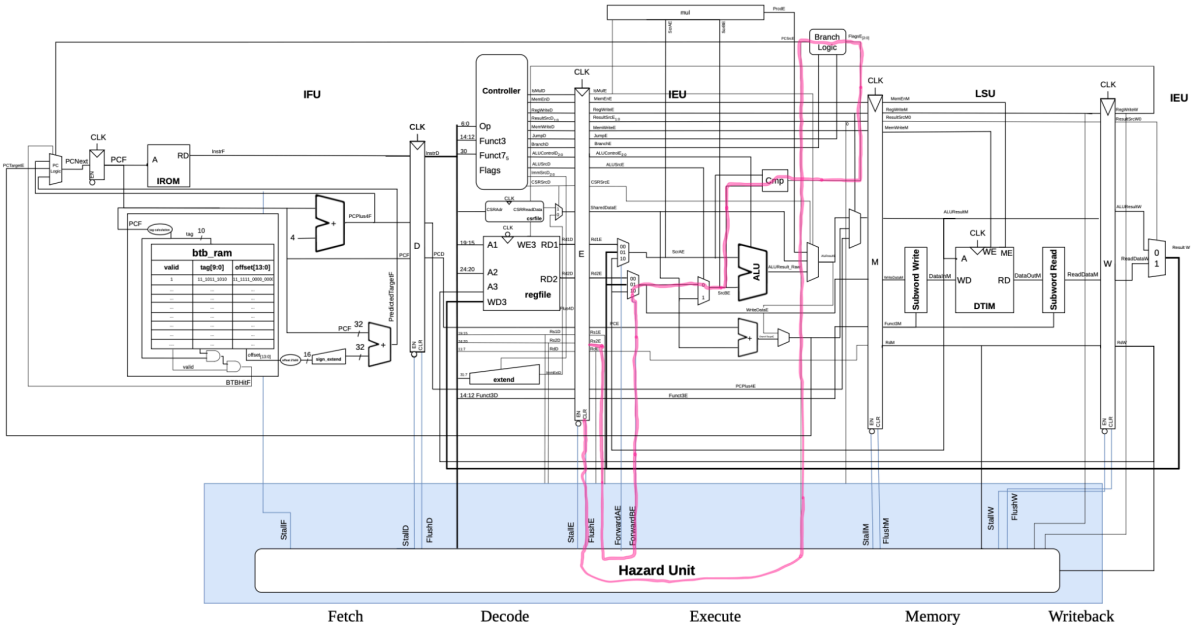

Figure 7. BTB Critical path

As you can see, the critical path is slowed down because of the combinational logic needed to produce the FlushE signal. This is because we need to use the PredictTakenE and PCSrcE signals to determine whether or not we need to flush the execution stage.

Learning from these lessons, we decided to design a branch predictor that takes minimal overhead area and doesn't affect our critical path all while producing a high Coremark/MHz score. To do this, we implemented a branch predictor that always takes backwards branches (Fig. 8).

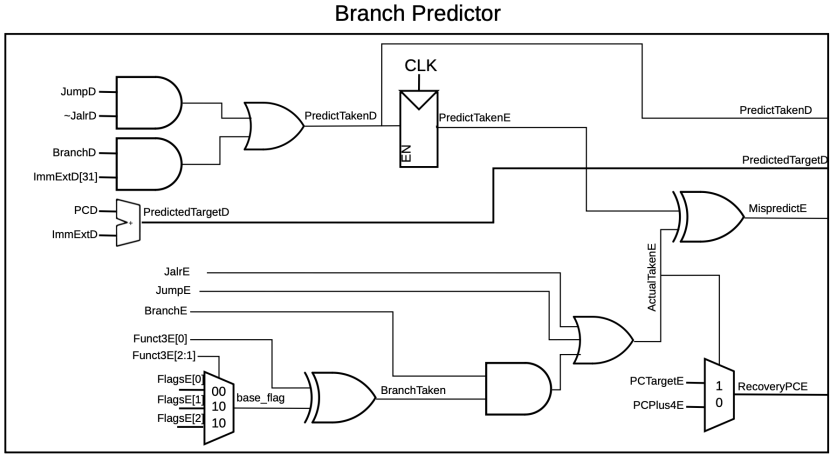

Figure 8. Backwards Branch Predictor

The predictor logic predicts the branch taken if the offset is negative. We can check if the offset is negative if ImmExtD[31] is 1. Thus, if the instruction is a Jump or Branch and the 31st bit of the ImmExt is 1, we predict a branch taken. Our PCNext is then chosen to be the PredictedTarget. There is a 1-cycle penalty if the predicted taken is correct and a 2-cycle penalty if the prediction is wrong. When testing it against Coremark, we found the Coremark/MHz score increased from 2.83 to 2.91. This is the branch prediction logic we used in our final processor (ieu.sv in appendix).

List of standard extensions implemented

RV32I

We started with the base integer instruction set for 32-bit RISC-V processors. RV32I provides all the fundamental instructions needed for a complete, general-purpose computer. RV32I instructions include:

- Integer arithmetic: ADD, SUB, AND, OR, XOR, Shifts
- Load and Store: Memory access for bytes, halfwords, and words
- Conditional branches: (BEQ, BNE, BLT, BGE, etc.) and jumps (JAL, JALR)
- Set-less-than instructions: (SLT, SLTU)
- Immediate operations: ADDI, ANDI, etc.

Zmmul

Zmmul provides just the multiplication instructions from the M extension, without division or remainder operations.

- MUL: Multiply (lower 32 bits of result)
- MULH: Multiply high (upper 32 bits, signed × signed)
- MULHSU: Multiply high (signed × unsigned)
- MULHU: Multiply high (unsigned × unsigned)

This extension significantly cuts down the amount of instructions because it can be done in one cycle rather than in a software loop.

Zicsr and Zicntr

Zicsr and Zicntr were implemented in the control and status register to allow us to benchmark the processor using coremark. These two instructions allow us to write and read the CSR registers to calculate our coremark score.

Figure of Merit

$$Score = \frac{10^{30}}{(CoreMark_cycles * clock_period)^3 * Area}$$

To gauge the worthiness of our processor, we used a score provided by the SOC teaching team that factors in execution time, and area. The emphasis is on execution time rather than Area. Our primary focus in this

project was first on clock period through pipelining, next through cycles by branch predictions, then area by architecture reformations.

We predicted that our final clock speed would be 3.86, and we surpassed this with our fast multiplication unit. Our estimation of the final area was nearly twice as large as our final. Since we did not implement any new extensions, the amount of cycles did not decrease. We also predicted that we could use software to better compile coremark, however we found its initial state was already maximally optimized. In conclusion, our final score of 4814 was not as good as our predicted figure of merit prediction 7190. The table below demonstrates the improvements between checkpoints during the project, which represent our cumulative project during the semester.

	Initial	Checkpoint 1	Checkpoint 2	Final
Figure of Merit	456.99	777.19 (+41.19%)	4,208.96 (+441.57%)	4814 .14 (+12.588%)
Crit Path Length	8.24 ns	5.101549 (-38.1%)	3.017745 (-40.98%)	3.017745 (No Change)
Area	221,069.87 μm^2	282,589.71 (+27.82%)	196,409.57 (-30.497%) μm^2	188,099.94 (-4.231%) μm^2
Coremark Score	3.79	3.07 (-18.997%)	2.83 (-7.818%)	2.91 (+2.827%)
CPI	1	1.140 (+14%)	1.24 (+8.77%)	1.190 (-4.032%)
Execution Time	27,494,271	16,574,458.26 (-39.71%)	10,655,014.82 (-35.714%)	10,336,280.59 (-2.991)

Final Area Report

Hierarchical Module	Absolute Total (μm^2)	Percent of Total	Combinational (μm^2)	Non-combinational (μm^2)	Design Module
riscvsingle	188,099.94	100.0%	3,847.22	1,256.88	riscvsingle
c (controller)	24,682.58	13.1%	22,418.64	2,263.94	controller

dp (datapath)	159,569.32	84.8%	15,287.42	11,683.20	datapath
dp/alu	34,256.78	18.2%	34,256.78	0.00	alu
dp/extend	8,947.34	4.8%	8,947.34	0.00	extend
dp/mul	52,841.68	28.1%	7,653.89	46,396.71	mul
dp/mux2_1 (ForwardAE)	1,847.29	1.0%	1,847.29	0.00	mux2
dp/mux2_2 (ForwardBE)	1,836.48	1.0%	1,836.48	0.00	mux2
dp/mux3_1 (ResultSrcMux)	3,124.79	1.7%	3,124.79	0.00	mux3
dp/mux3_2 (ALUSrcAMux)	2,956.18	1.6%	2,956.18	0.00	mux3
dp/mux4 (ALUSrcBMux)	4,187.34	2.2%	4,187.34	0.00	mux4
dp/pcreg	1,247.84	0.7%	0.00	1,247.84	flopenr
dp/rf (register file)	33,084.36	17.6%	6,687.55	26,396.81	regfile
Total	**188,099.94**	100.0%	126,499.99	61,599.94	

Table of time breakdown on each aspect of the project, with a comparison to the proposal estimate

During the project, we tracked our time spent and progress each session in a work log. Below is an overview of what parts of the project we spent time on during each work session.

Description of work	**Date**	**Time Spent Together**	**Coremark Score**	**Individual time**

Pipelining	3/23	1.5	N/A	
Pipelining	3/24	2	N/A	
Pipelining	3/30	2.5	N/A	
Pipelining	3/30	1.5	N/A	
Pipelining	3/31		N/A	3
Pipelining	3/31	2		
Pipelining	4/1	7	777.19	
Checkpoint 1 prep	4/2	2	777.19	5
mmu	4/5	2.25	750	
mmu	4/7	2	750	
mmu	4/11	2	1300	
mmu	4/12		1106	3
Design Review	4/13	1.5	2,000.00	
Diagram	4/14	1	4,208.96	
Checkpoint 2 prep	4/16	3	4,208.96	
Compile flags + BTB	4/20	2	4,208.96	
BTB / BPred	4/21	3	4,208.96	
BTB / BPred	4/23	1	4,208.96	
BTB / BPred	4/26	2	4,208.96	
Area Optimization	4/27	4	4504.41	
Final Report	4/29	5	4504.41	4
Final Report	4/30	4	4504.41	3

Time Spent per feature:

Pipelining	36 hr
MMU	15.5 hours
BTB / BPred	12 hours
Final Report	25 hours
Total Man Hours	120.5 hours

Comparison to Estimate

In the project proposal, we estimated the project would take us 60-80 hours by spending a minimum of 4 hours per week made up of 2.5 and 1.5 hour pair programming work sessions averaging 5-6 hours per week. This estimate was roughly 65 hours under the actual time spent.

Git commit log (git log)

```
commit 48d97a13a76306240af8481093f27422448c8c57 (HEAD -> main, origin/main, origin/HEAD)
Author: Max Conine [redacted]
Date:   Thu Apr 30 00:21:25 2026 -0700

    cleaned up csr file

commit 8305daceb8e162074b0a74d2a5901dc7632d8dd7
Author: Max Conine [redacted]
Date:   Thu Apr 30 00:04:04 2026 -0700

    code cleanup

commit f2b95fedb232756fa2562261a4747ed8ceb2ffb2
Merge: ea54cda71 fe0e012b2
Author: maxconine <58146795+maxconine@users.noreply.github.com>
Date:   Wed Apr 29 12:25:32 2026 -0700

    Merge pull request #1 from SystemOnChipTeam/optimizeMax

commit fe0e012b220adf2cc7b042a123207c2ecbc1be1c (origin/optimizeMax, optimizeMax)
Author: Max Conine [redacted]
Date:   Wed Apr 29 12:21:49 2026 -0700

    removed moduleparamid

commit b6b9ac15830cbf9e96dd2826dad5cffd7bc1ab1b
Author: Pierce Clark [redacted]
Date:   Mon Apr 27 17:38:27 2026 -0700

    New writeback logic

commit b147edfe0ea0e48c534f249b1ae2c8a5bc1406fe
Author: Max Conine [redacted]
Date:   Mon Apr 27 17:02:15 2026 -0700

    set_max_area 0, score:4,336.24

commit 703d215a4856e2bac156162ac1a7a7a2c9912dc4
Author: Max Conine [redacted]
Date:   Mon Apr 27 16:20:21 2026 -0700

    fixed formatting

commit 6e9a15bfa3ff8071d56fe732b3d76d9af11c13df
Author: Max Conine [redacted]
Date:   Mon Apr 27 15:38:31 2026 -0700

    4,327.48 score

commit de591ab7f91a7bdec42631763204cc71f47af0c1
Author: Max Conine [redacted]
Date:   Mon Apr 27 14:56:50 2026 -0700

    4,269.48 score

commit adb0aa63e278de6797a3f22db3f0bed43133f58c
Author: Max Conine [redacted]
Date:   Mon Apr 27 14:24:58 2026 -0700

    4250.22 score achieved

commit 635a84f4e64e9a8e9644a0cab66aa8345dd3ee39
Author: Max Conine [redacted]
Date:   Mon Apr 27 14:14:30 2026 -0700

    updated memory to 3ns delay

commit 0456ada9f29d896c69eb74a80a1c7eaf01d58044 (origin/StalledMul, StalledMul)
Merge: 1f5d03714 e669fa018
Author: Max Conine [redacted]
Date:   Mon Apr 20 16:34:31 2026 -0700

    increased irom speed, set critical range 0.1

commit 1a048dd0a96a300842b9a8fe922fd974f08a7a91
Merge: 522e1b3f1 fbad7eeae
Author: Pierce Clark [redacted]
Date:   Thu Apr 2 16:00:54 2026 -0700

    Merge branch 'main' of https://github.com/SystemOnChipTeam/SOC_Final_Proj

commit fbad7eeaed66ea152177c43b19577d263ef524d7
Author: Max Conine [redacted]
Date:   Thu Apr 2 16:00:46 2026 -0700

    ifu cleanup

commit 522e1b3f1c6ea14e897e8afe18dd3cbbac90b22f
Author: Pierce Clark [redacted]
Date:   Thu Apr 2 16:00:43 2026 -0700

    Cleaned Controller code comments

commit ce0ea8d3dfc55ea2599582410f4934a46dc4a0c9
Author: Pierce Clark [redacted]
Date:   Thu Apr 2 15:57:07 2026 -0700

    CSR file instantiation

commit 2a8b4ac1777dc69ba1db475e356f050c0eeeb2cd
Author: Max Conine [redacted]
Date:   Thu Apr 2 15:56:28 2026 -0700

    hazard unit cleanup

commit 955216dc69edcefcbf558327462c9dc6d6910508
Merge: 9ff01e33e ea0834aad
Author: Max Conine [redacted]
Date:   Thu Apr 2 15:51:47 2026 -0700

    Merge branch 'main' of https://github.com/SystemOnChipTeam/SOC_Final_Proj

commit 9ff01e33eb455750a88474e538d2e70d6985a153
Author: Max Conine [redacted]
Date:   Thu Apr 2 15:51:28 2026 -0700

    controller cleanup

commit ea0834aadba60d381c80df9d984cbe6e7b0972f8
Author: Pierce Clark [redacted]
Date:   Thu Apr 2 15:45:17 2026 -0700

    Clean Up ALU Comments

commit 7cccfa567e3ffd3b024d875cdb11994db193a60e
Author: Max Conine [redacted]
Date:   Thu Apr 2 03:04:20 2026 -0700

    Our first score!

commit 478b984480530fd0924abc1e6b71857a81fccba4
Author: Max Conine [redacted]
Date:   Thu Apr 2 01:24:02 2026 -0700

    42/45 act4 tests passing!

commit 9763569477e4d9c945bf4a4d3baf4ae187a8b66a
Author: Max Conine [redacted]
Date:   Thu Apr 2 01:08:28 2026 -0700

    passed add test

commit 87c392140a1150bd73dfed81fb0706e151e2eb14
Author: Max Conine [redacted]
Date:   Thu Apr 2 00:45:00 2026 -0700

    cleared compilation errors
```

commit 1f5d03714f1ef1b99c908f5d7f2ea2368a6e2c17
Author: Max Conine [redacted]
Date: Mon Apr 20 16:33:42 2026 -0700

increased irom speed, set critical range 0.1

commit e669fa018e4231e69d7af98d4abb1fa97dafda1f
Merge: 6353f454a 9f28f1e66
Author: Pierce Clark [redacted]
Date: Mon Apr 20 16:16:08 2026 -0700

Small changes

commit 6353f454a49cddfcb5530766a8d682985fe2dff7
Author: Pierce Clark [redacted]
Date: Mon Apr 20 16:14:02 2026 -0700

Small changes

commit 9f28f1e66fecd0bc1921c03785b0984da490d366
Author: Max Conine [redacted]
Date: Mon Apr 20 16:10:57 2026 -0700

formatted code

commit ef47da1c6763ba85b7818c6e2828fffa5b8c5924
Author: Pierce Clark [redacted]
Date: Tue Apr 14 21:36:43 2026 -0700

New mul unit fsm

commit d7430dc8cc894e8d0ae391db9403f3687b3abd0b
Author: Pierce Clark [redacted]
Date: Mon Apr 13 16:34:53 2026 -0700

Synth settings

commit b3295a9f7099e99b8044b915e8ed881a6b856793
Author: Pierce Clark [redacted]
Date: Mon Apr 13 11:53:11 2026 -0700

CSRfile size reduced

commit ce90c593b09f21e6cbaa2277f59deec43b515f60
Author: Pierce Clark [redacted]
Date: Mon Apr 13 10:51:19 2026 -0700

Removed PCtarget and PCplus4 from memory pipline and writeback mux

commit c7005047d300ab0cacbb8c444838405ed7bfc121
Author: Pierce Clark [redacted]
Date: Mon Apr 13 10:26:37 2026 -0700

Cleaned up ieu muxing code for the alu result, renamed variable in the mul unit to match drawing

commit 351bdc1a2a526947d94f0b526f8ca4b960aa813f
Author: Pierce Clark [redacted]
Date: Sun Apr 12 20:47:04 2026 -0700

Partial product multiplier synthed

commit 6886bd1a2390492159941fc22441beec4d3dce23
Author: Pierce Clark [redacted]
Date: Sun Apr 12 20:20:35 2026 -0700

Four partial product multiplier

commit 97368f98fe647a131c114ceb194940c4071b65a4
Author: Pierce Clark [redacted]
Date: Sun Apr 12 20:10:03 2026 -0700

Split multiplication into partial products

commit eaa56942cd45059d4c9c55db59df1382deeb3fbd
Author: Pierce Clark [redacted]
Date: Sun Apr 12 16:53:43 2026 -0700

New synth flag allows for retiming

commit c3ae6ea3d40887dc60f0634a86c3c0a51ee10865
Author: Pierce Clark [redacted]
Date: Sun Apr 12 16:24:05 2026 -0700

Act4 passes with optimized crit path

commit 600951345b5a35a8c24e502731bc2c15be33dedd
Author: Pierce Clark [redacted]
Date: Sun Apr 12 16:06:56 2026 -0700

Pierce branch passes act4

commit 8e6043429da9e35b40dbad35537a3525ce0dcb79
Author: Pierce Clark [redacted]
Date: Wed Apr 1 23:10:47 2026 -0700

Pipelined PCTarget for AUIPC Instruction

commit 6bb4c745ac015e70faf4bf6ac261e2669dad3ae7
Author: Pierce Clark [redacted]
Date: Wed Apr 1 22:45:15 2026 -0700

updated top level instantiations, removed writeen logic

commit 855957ae1b7966643912cb155d188be622ce9e34
Author: Pierce Clark [redacted]
Date: Wed Apr 1 22:03:09 2026 -0700

Hazard Unit Verilog syntax fixes

commit aed560fd84235d20d2a1066dc9a26a54062462a2
Author: Pierce Clark [redacted]
Date: Wed Apr 1 21:59:04 2026 -0700

updated file headers

commit 6314db1852cba2fa60846cd1fccc22c9c83d16db
Author: Pierce Clark [redacted]
Date: Wed Apr 1 21:54:40 2026 -0700

Organized signals, updated instantiations, add pipeline registers

commit f75b60f8ae0003415d2362f26e28b5a6fe79c541
Author: Pierce Clark [redacted]
Date: Wed Apr 1 20:30:43 2026 -0700

Fixed signal widths and updated file headers

commit 1a9e93ecee2a7259c19af87aa13e52f1125502b9
Author: Pierce Clark [redacted]
Date: Wed Apr 1 20:12:36 2026 -0700

fixed signal instantiations, sizes, added comments

commit e9883859f4b02ede45b363f6ef8b1c44e0111c5e
Merge: 8300d558a 6c9ffb47c
Author: Pierce Clark [redacted]
Date: Tue Mar 31 22:00:39 2026 -0700

Merge branch 'main' of https://github.com/SystemOnChipTeam/SOC_Final_Proj

commit 8300d558a178f190a1bcbaa15f41ae42cda38f86
Author: Pierce Clark [redacted]
Date: Tue Mar 31 22:00:24 2026 -0700

IEU interfaces with ifu, hazard, and lsu units. IEU still needs significant rewiring work

commit 6c9ffb47c90e7b824235f129e98fa290b80af2ce
Merge: 960a519b2 80b3a6485
Author: Max Conine [redacted]
Date: Tue Mar 31 21:59:19 2026 -0700

Merge branch 'main' of https://github.com/SystemOnChipTeam/SOC_Final_Proj

commit 960a519b26e5fc3d413388e384de89921a34d6bf
Author: Max Conine [redacted]
Date: Tue Mar 31 21:59:03 2026 -0700

controller in good condition

commit 80b3a6485258571796905454c79473647d1ad75a
Author: Pierce Clark [redacted]
Date: Tue Mar 31 20:47:38 2026 -0700

Updated ifu and lsu to match block diagrams, added memory interfaces and hazard unit interface

commit f5381286846afb7895e0628930c8ffd93f615732
Author: Max Conine [redacted]
Date: Tue Mar 31 20:30:06 2026 -0700

working on controller

commit 9d96e2b1c749eda9abc439378921483f3b120d69
Author: Pierce Clark [redacted]
Date: Mon Mar 30 09:45:32 2026 -0700

new lsu completed

commit cb7b9ace53d0b86edbed6a39b29fbb6677928d31
Author: Max Conine [redacted]
Date: Mon Mar 30 09:44:42 2026 -0700

implemented hazard unit

```
commit 8b6bc0b2080d960aa0562b53857c460a70613583
Author: Pierce Clark [redacted]
Date:   Sun Apr 12 15:44:01 2026 -0700

    Pierce multiply unit C test passes

commit d0a6b3aea6faec255e95fa3ab221d5a35da0527c
Author: Max Conine [redacted]
Date:   Sun Apr 5 17:05:57 2026 -0700

    pipelining mul progress

commit ea54cda71f460ba996e11b1047d6d4ab9a6533fc
Merge: b833f61f2 5d25cf205
Author: Pierce Clark [redacted]
Date:   Thu Apr 2 16:05:58 2026 -0700

    Merge branch 'main' of https://github.com/SystemOnChipTeam/SOC_Final_Proj

commit b833f61f2395455ace20181b118539a874ebe5e3
Author: Pierce Clark [redacted]
Date:   Thu Apr 2 16:05:53 2026 -0700

    Deleted subword read and write modules

commit 5d25cf205f48e44b7bf9a2f7ad828ebff95191e2
Author: Max Conine [redacted]
Date:   Thu Apr 2 16:05:45 2026 -0700

    lsu cleanup
```

```
commit 942b07f70cff04fd28b3509d5f08f316226f078e
Author: Max Conine [redacted]
Date:   Tue Mar 24 21:58:04 2026 -0700

    Created NEW Ieu and Lsu, layed out structurally, added pipeline registers, wires routed in
Lsu

commit 16d4facb136c3d63bf4544b73b9b6e57f4dbd18b
Author: Pierce Clark [redacted]
Date:   Mon Mar 23 17:33:27 2026 -0700

    Working on risc-v single module

commit 259686b06f47538b56fc475094e11c77c4537044
Author: Pierce Clark [redacted]
Date:   Mon Mar 23 16:30:06 2026 -0700

    Added processor verilog and deleted examples

commit 08668610c4fd2e62b932b6bafbd428e729d1613c
Merge: c3cf76249 b2a540d49
Author: Jordan Carlin <jordanmcarlin@gmail.com>
Date:   Mon Mar 23 07:04:30 2026 -0700

    Merge pull request #1688 from
openhwgroup/dependabot/github_actions/actions/upload-artifact-7

    Bump actions/upload-artifact from 6 to 7
```

Reflection on teamwork

We both shared a genuine interest in this project, which motivated us to work hard throughout. Our communication was frequent, with discussions and design reviews that helped us align on architectural decisions. We collaborated well on the code itself, working together on implementation details, and keeping a comprehensive log of progress and goals.

Looking ahead to future projects, we see two key areas for improvement. First, we would invest more time in high-level planning before going into technical implementation. This would have helped us avoid working without a clear plan and make better architectural decisions upfront. Second, we need to use our time together more efficiently. Near the end of the project, we were just running synthesis, which consumed valuable time that could have been better spent. Overall, we are happy with the teamwork aspect of this project and both learned valuable lessons that will be useful in the future.

Verification

We verified our processor using act4 and coremark. Act 4 is a good act comprehensively testing if operations give the correct results, while coremark provides a good idea of how a real program would perform. We found that act4 was not comprehensive enough to catch pipelining bugs that later popped up in coremark.

Act4

```
=== Test Log Summary ===
Total logs:              45
Passed:                  45
Failed:                  0
Missing 'Completed':     0
Unreadable:              0
```

✅ Passed tests:
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-add-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-addi-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-and-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-andi-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-auipc-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-beq-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-bge-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-bgeu-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-blt-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-bltu-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-bne-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-fence-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-jal-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-jalr-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-lb-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-lbu-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-lh-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-lhu-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-lui-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-lw-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-nop-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-or-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-ori-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-sb-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-sh-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-sll-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-slli-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-slt-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-slti-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-sltiu-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-sltu-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-sra-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-srai-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-srl-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-srli-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-sub-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-sw-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-xor-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/I/I-xori-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/Zicntr/Zicntr-csrrs-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/Zihpm/Zihpm-csrrs-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/Zmmul/Zmmul-mul-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/Zmmul/Zmmul-mulh-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/Zmmul/Zmmul-mulhsu-00.elf.sim.log
/home/pclark/SOC_Final_Proj/examples/exercises/lynn/tests/act4/work/lynn-rv32I/elfs/rv32i/Zmmul/Zmmul-mulhu-00.elf.sim.log

Coremark Verification

```
# [TB] ENTRY_ADDR = 0x80000000
# [TB] TOHOST_ADDR = 0x8000d000
# [10000] INFO: Starting simulation.
#    WALLY CoreMark Results (from get_time)
#     Elapsed MTIME: 3425167
#     Elapsed MINSTRET: 2864107
```

```
#    CoreMark/MHz Score: 10,000,000 / 3425167 = 2.91
#    CPI: 3425167 / 2864107 = 1.19
# 2K performance run parameters for coremark.
# CoreMark Size    : 666
# Total ticks      : 3425167
# Total time (secs): 34
# Iterations/Sec   : 0
# Iterations       : 10
# Compiler version : GCC15.2.0
# Compiler flags   : -g -mabi=ilp32 -march=rv32i_zicsr_zmmul -static -falign-functions=16 -mbranch-cost=1
-DSKIP_DEFAULT_MEMSET -mtune=sifive-3-series -O3 -finline-functions -falign-jumps=4 -fno-delete-null-pointer-checks
-fno-rename-registers --param=loop-max-datarefs-for-datadeps=0 -funroll-all-loops --param=uninlined-function-insns=8 -fno-tree-vrp
-fwrapv -fipa-pta -nostdlib -nostartfiles -ffreestanding -mstrict-align -DTOTAL_DATA_SIZE=2000 -DMAIN_HAS_NOARGC=1
-DPERFORMANCE_RUN=1 -DITERATIONS=10 -DXLEN=32
# INFO: Test Completed!
# [35372831000] INFO: Program Finished! Ending simulation.
# ** Note: $finish    : /home/pclark/SOC_Final_Proj/examples/exercises/lynn/tb/testbench.sv(165)
#    Time: 35372831 ns  Iteration: 0  Instance: /testbench
# End time: 09:22:30 on Apr 30,2026, Elapsed time: 0:00:34
```

Coremark executed with zero errors in the final version of the processor. The final processor has a Coremark/MHz score of 2.91, and a CPI of 1.19. The processor also passes all 45 act4 tests for rv32IM

Verification Issues

```
# [0]ERROR! list crc 0xec70 - should be 0xe714
# [0]ERROR! matrix crc 0xb9d6 - should be 0x1fd7
# [0]ERROR! state crc 0xb215 - should be 0x8e3a
```

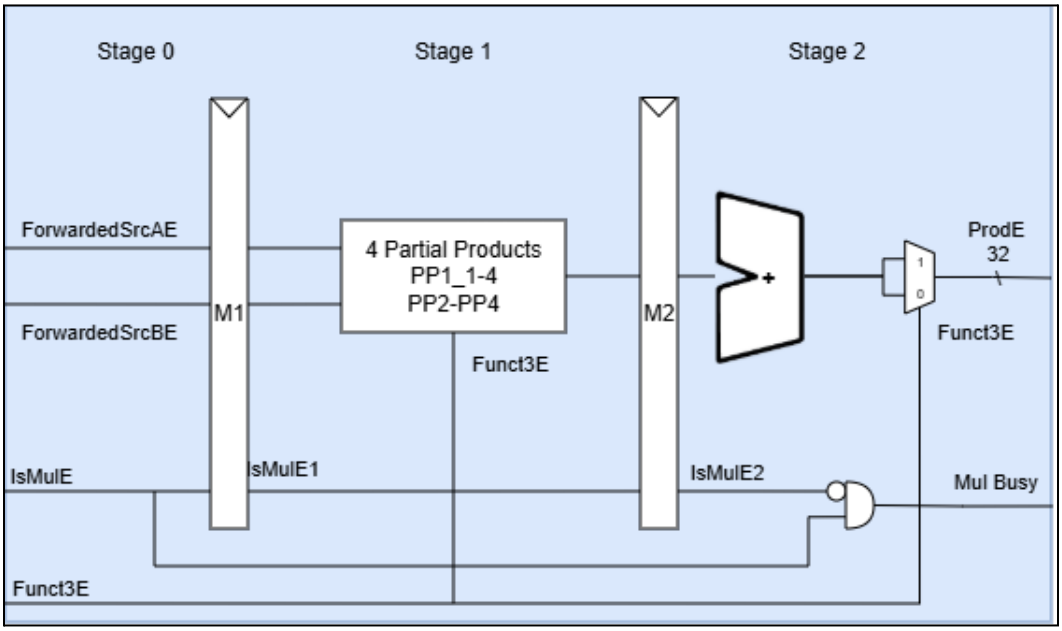

Fig. 9: Version of Mul unit that failed to pass Coremark test.

During testing, we found that the processor was giving wrong calculations despite passing the act 4 testbench. This was caused by improper stall logic that failed to stall a second multiply in back to back multiplications. This is a very niche corner case that only happens in pipelined processors, because act4 doesn't test if a processor can do back to back multiplication, just that it can do one at a time right. We found this to be a very challenging, but rewarding bug to solve in relation to pipelining.

Summary of how you have applied AI to your project

AI was most effective in debugging our code and suggesting synthesis improvements. When connecting units together, it's easy to mess up signal logic. It was very effective at recognizing bugs and combinational verilog for errors before compilation. There was some hallucination in the copilot autofill feature of it making signals up when we were connecting modules. We also got led astray in debugging the hazard unit logic. Some lessons we learned are to debug on your own before asking AI to help out so you can get a better understanding of possible bugs. We also found giving the AI model a diagram of the processor, helped it understand our code better and helped it suggest better solutions to the bugs we were having. Anything from help with git to verifying ideas, we found AI helpful at giving advice on.

Future improvements

We plan to further reduce the critical path length by adding another stage to our processor in the execute stage as shown in Fig. 5. This will allow for less stalls in the multiply unit and reduced critical path length. We plan to have the muxes selecting srcA and srcB for the ALU computation to be in the first stage as they rely on signals that travel from other parts of our processor.
In lab 5, we found that the coremark benchmark test has 173933 mul instructions. Since we currently stall every time we multiply, we are adding an additional 173933 stalls. This decreases the performance of our processor by ~5% as the total number of cycles was 3,248,907 when we didn't pipeline our mul unit.

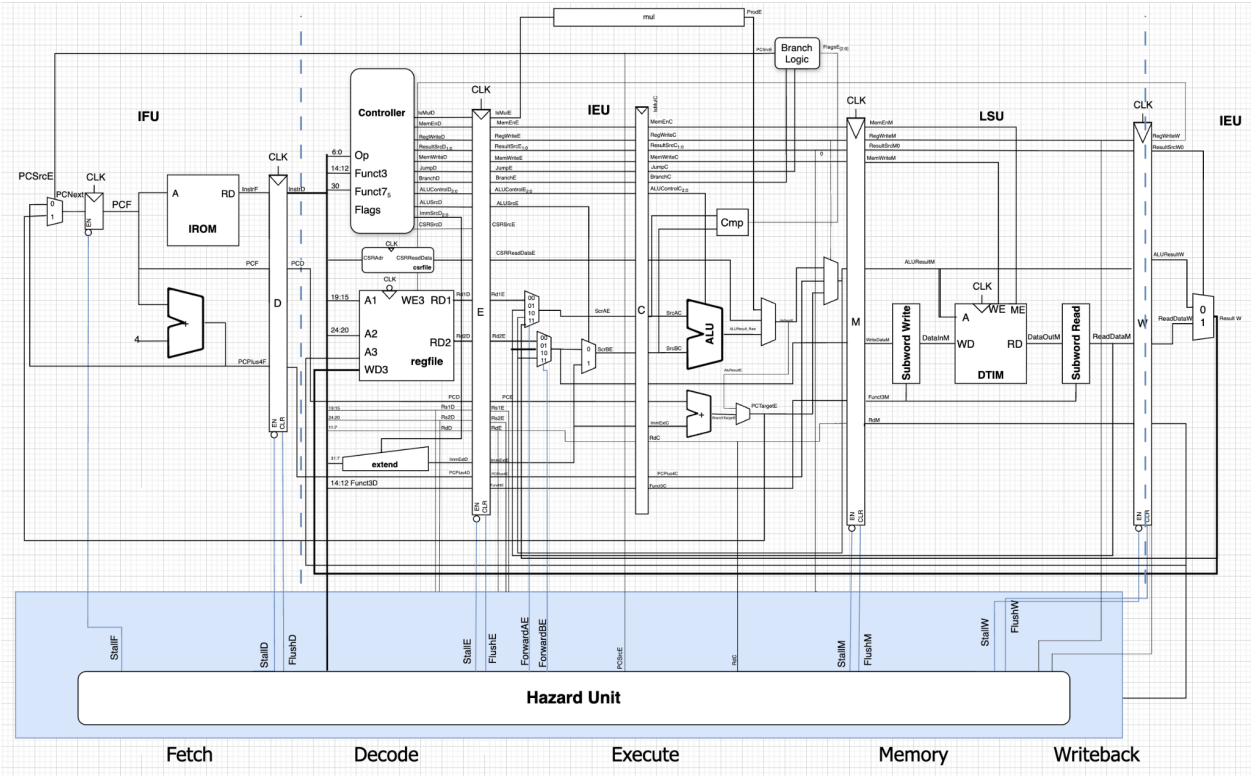

Fig. 10: [6-Stage Block Diagram](). Currently not implemented yet.

Appendix A: SystemVerilog code

mul.sv

```
////////////////////////////////////////////
// mul.sv
//
// Written: David_Harris@hmc.edu 16 February 2021
// Modified: Pierce Clark and Max Conine 12 April 2026
//
// Purpose: Integer multiplication
//
// Documentation: RISC-V System on Chip Design
//
// A component of the CORE-V-WALLY configurable RISC-V project.
// https://github.com/openhwgroup/cvw
//
// Copyright (C) 2021-23 Harvey Mudd College & Oklahoma State University
//
// SPDX-License-Identifier: Apache-2.0 WITH SHL-2.1
//
// Licensed under the Solderpad Hardware License v 2.1 (the “License”); you may not use this
file
// except in compliance with the License, or, at your option, the Apache License version 2.0.
You
// may obtain a copy of the License at
//
// https://solderpad.org/licenses/SHL-2.1/
//
// Unless required by applicable law or agreed to in writing, any work distributed under the
// License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
// either express or implied. See the License for the specific language governing permissions
// and limitations under the License.
```

```
////////////////////////////////////////////////////////////////////////////////////////////////
/

module mul #(parameter XLEN) (
   input logic clk, reset,
   input logic [XLEN-1:0] ForwardedSrcAE, // source A from after Forwarding mux
   ForwardedSrcBE,                       // source A and B from after Forwarding mux
   input logic [2:0]       Funct3E,      // type of multiply
   input logic             IsMulE,       // whether we are doing a multiply instruction
   output logic [XLEN-1:0] ProdE,        // double-widthproduct
   output logic            MulWorking    // whether we are currently processing a multiply
instruction in execute stage (for hazard unit)
);

 // Number systems
 // Let A' = sum(i=0, XLEN-2, A[i]*2^i)
 // Unsigned: A = A' + A[XLEN-1]*2^(XLEN-1)
 // Signed:   A = A' - A[XLEN-1]*2^(XLEN-1)

 // Multiplication: A*B
 // Let P' = A' * B'
 //     PA = (A' * B[XLEN-1])
 //     PB = (B' * A[XLEN-1])
 //     PP = A[XLEN-1] * B[XLEN-1]
 // Signed * Signed     = P' + (-PA - PB)*2^(XLEN-1) + PP*2^(2XLEN-2)
 // Signed * Unsigned   = P' + ( PA - PB)*2^(XLEN-1) - PP*2^(2XLEN-2)
 // Unsigned * Unsigned = P' + ( PA + PB)*2^(XLEN-1) + PP*2^(2XLEN-2)

 logic [XLEN-1:0] ForwardedSrcAE1, ForwardedSrcBE1;
 logic [XLEN-1:0] Aprime, Bprime;  // lower bits of source A and B
 logic MULH, MULHSU;  // type of multiply
 logic [XLEN-2:0] PA, PB;  // product of msb and lsbs
 logic PP;  // product of msbs
 logic [XLEN*2-1:0] PP2E, PP3E, PP4E;  // partial products
 logic [XLEN*2-1:0] PP1E2, PP2E2, PP3E2, PP4E2;  // registered partial proudcts
 logic IsMulE1;  // registered version of IsMulE to track which stage of multiplication we are
in

 ////////////////////////////
 // Stage1: Compute partial products
 ////////////////////////////

 flopr #(XLEN) ForwardAReg (clk, reset, ForwardedSrcAE, ForwardedSrcAE1);
 flopr #(XLEN) ForwardBReg (clk, reset, ForwardedSrcBE, ForwardedSrcBE1);
 flopr #(1) IsMulReg1 (clk, reset, IsMulE, IsMulE1);

 assign Aprime = {1'b0, ForwardedSrcAE1[XLEN-2:0]};
 assign Bprime = {1'b0, ForwardedSrcBE1[XLEN-2:0]};

 localparam HALF = XLEN / 2;
 logic [HALF-1:0] Aprime_lo, Aprime_hi;
 logic [HALF-1:0] Bprime_lo, Bprime_hi;
 logic [XLEN-1:0] PP1E_ll, PP1E_lh, PP1E_hl, PP1E_hh;

 assign Aprime_lo = Aprime[HALF-1:0];
 assign Aprime_hi = Aprime[XLEN-1:HALF];
 assign Bprime_lo = Bprime[HALF-1:0];
 assign Bprime_hi = Bprime[XLEN-1:HALF];

 // Four 16x16 multiplies — true HALF x HALF critical path
 assign PP1E_ll = Aprime_lo * Bprime_lo;
 assign PP1E_lh = Aprime_lo * Bprime_hi;
 assign PP1E_hl = Aprime_hi * Bprime_lo;
 assign PP1E_hh = Aprime_hi * Bprime_hi;

 assign PA = {(XLEN - 1) {ForwardedSrcAE1[XLEN-1]}} & ForwardedSrcBE1[XLEN-2:0];
 assign PB = {(XLEN - 1) {ForwardedSrcBE1[XLEN-1]}} & ForwardedSrcAE1[XLEN-2:0];
 assign PP = ForwardedSrcAE1[XLEN-1] & ForwardedSrcBE1[XLEN-1];

 // flavor of multiplication
```

```
 assign MULH = (Funct3E == 3'b001);
 assign MULHSU = (Funct3E == 3'b010);

 // Select partial products, handling signed multiplication
 assign PP2E = {2'b00, (MULH | MULHSU) ? ~PA : PA, {(XLEN - 1) {1'b0}}};
 assign PP3E = {2'b00, (MULH) ? ~PB : PB, {(XLEN - 1) {1'b0}}};
 always_comb
   if (MULH) PP4E = {1'b1, PP, {(XLEN - 3) {1'b0}}, 1'b1, {(XLEN) {1'b0}}};
   else if (MULHSU) PP4E = {1'b1, ~PP, {(XLEN - 2) {1'b0}}, 1'b1, {(XLEN - 1) {1'b0}}};
   else PP4E = {1'b0, PP, {(XLEN * 2 - 2) {1'b0}}};

 ////////////////////////////
 // Stage2: Sum partial proudcts
 ////////////////////////////
 logic [XLEN-1:0] PP1E_ll2, PP1E_lh2, PP1E_hl2, PP1E_hh2;
 flopr #(XLEN) PP1E_llReg (clk, reset, PP1E_ll, PP1E_ll2);
 flopr #(XLEN) PP1E_lhReg (clk, reset, PP1E_lh, PP1E_lh2);
 flopr #(XLEN) PP1E_hlReg (clk, reset, PP1E_hl, PP1E_hl2);
 flopr #(XLEN) PP1E_hhReg (clk, reset, PP1E_hh, PP1E_hh2);
 flopr #(XLEN * 2) PP2Reg (clk, reset, PP2E, PP2E2);
 flopr #(XLEN * 2) PP3Reg (clk, reset, PP3E, PP3E2);
 flopr #(XLEN * 2) PP4Reg (clk, reset, PP4E, PP4E2);

 // Combine
 assign PP1E2 = {{XLEN{1'b0}},  PP1E_ll2}
              + {{HALF{1'b0}},  PP1E_lh2, {HALF{1'b0}}}
              + {{HALF{1'b0}},  PP1E_hl2, {HALF{1'b0}}}
              + {               PP1E_hh2, {XLEN{1'b0}}};

 // add up partial products; this multi-input add implies CSAs and a final CPA
 logic [XLEN*2-1:0] ProdFull;  // internal full-width product
 assign ProdFull = PP1E2 + PP2E2 + PP3E2 + PP4E2;  //ForwardedSrcAE * ForwardedSrcBE;

 assign ProdE = (Funct3E == 3'b000) ? ProdFull[XLEN-1:0] : ProdFull[XLEN*2-1:XLEN];
 // mul working logic: used to give the multiply unit an extra cycle to compute the result

 // mul working logic: used to give the multiply unit 2 extra cycles to compute the result
 logic [1:0] count;
 always_ff @(posedge clk) begin
   if (IsMulE & ~IsMulE1)  // first cycle of a new multiply
     count <= 2'b00;
   else if (count == 2'b10)  // finished, reset for next
     count <= 2'b00;
   else if (IsMulE)  // count while multiply active
     count <= count + 1;
 end

 assign MulWorking = IsMulE & (count < 2'b10);

endmodule
```

hazard.sv

```
// hazard.sv
// RISC-V pipelined processor
// Max Conine and Pierce Clark
// pclark@hmc.edu mconine@hmc.edu 2026

module hazard (
   // Decode Stage Inputs
   input  logic [4:0]  Rs1D, Rs2D,           // Source registers
   input  logic        PredictTakenD,        // Branch predicted taken in Decode

   // Execute Stage Inputs
   input  logic [4:0]  Rs1E, Rs2E,           // Source registers
   input  logic [4:0]  RdE,                  // Destination register
```

```
    input  logic        MispredictE,          // Branch mispredicted in Execute
    input  logic        ResultSrcE0,          // Load instruction flag
    input  logic        MulBusy,              // Multi-cycle multiply active flag

    // Memory Stage Inputs
    input  logic [4:0]  RdM,                  // Destination register
    input  logic        RegWriteM,            // Register write enable

    // Writeback Stage Inputs
    input  logic [4:0]  RdW,                  // Destination register
    input  logic        RegWriteW,            // Register write enable

    // Forwarding Controls
    output logic [1:0]  ForwardAE, ForwardBE, // ALU operand mux selects

    // Pipeline Flow Controls
    output logic        StallF,               // Fetch stall
    output logic        StallD, FlushD,       // Decode stall/flush
    output logic        StallE, FlushE,       // Execute stall/flush
    output logic        StallM, FlushM,       // Memory stall/flush
    output logic        StallW, FlushW        // Writeback stall/flush
);

    logic lwStall;
    assign lwStall = ResultSrcE0 & ((Rs1D == RdE) | (Rs2D == RdE)); //& (RdE != 5'b0)

    assign StallF  = lwStall | MulBusy;
    assign StallD  = lwStall | MulBusy;

    // Flush Decode if we mispredict in execute OR if we predict taken in decode
    assign FlushD  = MispredictE | PredictTakenD;
    // Flush Execute if there's a load stall OR if we mispredict in execute
    assign FlushE  = lwStall | MispredictE;

    // forward to solve data hazards whenever possible
    always_comb begin
        if (((Rs1E == RdM) & RegWriteM) & (|Rs1E)) ForwardAE = 2'b10;
        else if (((Rs1E == RdW) & RegWriteW) & (|Rs1E)) ForwardAE = 2'b01;
        else ForwardAE = 2'b00;
    end

    always_comb begin
        if (((Rs2E == RdM) & RegWriteM) & (|Rs2E)) ForwardBE = 2'b10;
        else if (((Rs2E == RdW) & RegWriteW) & (|Rs2E)) ForwardBE = 2'b01;
        else ForwardBE = 2'b00;
    end

    // tie off unused pipeline control signals
    assign StallE = MulBusy;
    assign StallM = MulBusy;
    assign FlushM = 1'b0;
    assign StallW = MulBusy;
    assign FlushW = 1'b0;

endmodule
```

ieu.sv

```
// ieu.sv
// RISC-V pipelined processor
// Max Conine and Pierce Clark
// pclark@hmc.edu mconine@hmc.edu 2026

module ieu(
        // Decode Stage Inputs
        input   logic           clk, reset,
        input   logic [31:0]    InstrD,     // 32-bit instruction
        input   logic [31:0]    PCD,        // Program Counter
```

```
        input   logic [31:0]    PCPlus4D,   // PC + 4

        // Execute Stage Outputs
        output  logic           MemEnE,     // Memory access enable
        output  logic           RegWriteE,  // Register file write enable
        output  logic           MemWriteE,  // Data memory write enable
        output  logic [31:0]    ALUResultE, // ALU result or memory address
        output  logic [31:0]    WriteDataE, // Data to store in memory
        output  logic [2:0]     Funct3E,    // funct3 for memory/branch ops

        // Memory Stage Inputs
        input logic [31:0]      ALUResultM, // Forwarded ALU result from M

        // Writeback Stage Inputs
        input   logic           RegWriteW,  // Writeback register write enable
        input   logic [31:0]    ALUResultW, // Forwarded ALU result from W
        input   logic [31:0]    ReadDataW,  // Forwarded memory data from W
        input   logic [4:0]     RdW,        // Writeback destination register

        // Hazard Unit Decode Stage Interface
        output  logic [4:0]     Rs1D, Rs2D, // Source regs for stall detection

        // Hazard Unit Execute Stage Interface
        input   logic           StallE, FlushE, StallM, FlushM, StallW, FlushW, // Stall and
flush signals from hazard unit
        input   logic [1:0]     ForwardAE, ForwardBE, // Forwarding mux selects
        output  logic [4:0]     Rs1E, Rs2E, RdE,      // Registers for forwarding logic

        // Branch Prediction Interface
        output  logic           PredictTakenD,        // Prediction made in Decode
        output  logic [31:0]    PredictedTargetD,     // Predicted target address in Decode
        output  logic           MispredictE,          // Flag if prediction was wrong in
Execute
        output  logic [31:0]    RecoveryPCE,          // Correct PC to recover to if
mispredicted

        output  logic           ResultSrcE0,          // Load instruction flag (ResultSrc[0])
        output  logic           MulWorking            // Whether we are currently processing a
multiply instruction in execute stage
    );

    // Decode Stage internal signals
    logic        MemEnD, RegWriteD, MemWriteD;
    logic [1:0]  ResultSrcD, ResultSrcE;
    logic        JumpD, BranchD, ALUSrcD;
    logic [3:0]  ALUControlD;
    logic [2:0]  ImmSrcD;
    logic [31:0] PCPlus4E;

    // Datapath (D-stage) wires
    logic [31:0] Rd1D, Rd2D, ImmExtD;
    logic        CSRSrcD;
    logic [31:0] CSRReadDataD;
    logic        IsMulD;

    // Detect JALR in decode stage based on opcode
    logic        JalrD;
    assign JalrD = (InstrD[6:0] == 7'b1100111);

    // Combinational assignments
    assign ResultSrcE0 = ResultSrcE[0];
    assign Rs1D = InstrD[19:15];
    assign Rs2D = InstrD[24:20];

    // Shared Decode Data (Multiplex ImmExtD and CSRReadDataD)
    logic [31:0] SharedDataD;
    assign SharedDataD = CSRSrcD ? CSRReadDataD : ImmExtD;

    // Predict branches taken if offset is negative (ImmExtD[31]).
    // Predict JAL (JumpD & ~JalrD) as always taken.
```

```
   assign PredictedTargetD = PCD + ImmExtD;
   assign PredictTakenD = (BranchD & ImmExtD[31]) | (JumpD & ~JalrD);

   // Pipeline Register for Prediction
   logic PredictTakenE;
   flopenrc #(1) PredictTakenEReg (clk, reset, FlushE, ~StallE, PredictTakenD,
PredictTakenE);

   // Execute Stage internal signals
   logic [31:0] Rd1E, Rd2E, PCE, SharedDataE;
   logic        JumpE, BranchE, ALUSrcE, JalrE;
   logic [3:0]  ALUControlE;
   logic [31:0] SrcAE, SrcBE;
   logic [2:0]  FlagsE;
   logic        BranchTaken;
   logic [31:0] BranchTargetE;
   logic [31:0] PCTargetE; // Moved from output port to internal logic
   logic [31:0] ProdE;
   logic        IsMulE;
   logic        CSRSrcE;

   // Writeback
   logic [31:0] ResultW;

   // Decode Stage Modules
   controller c(.clk, .reset, .InstrD, .MemEnD, .RegWriteD, .ResultSrcD, .MemWriteD, .JumpD,
.BranchD, .ALUControlD, .ALUSrcD, .ImmSrcD, .CSRSrcD, .IsMulD);

   csrfile csr(.clk(clk), .reset(reset), .CSRAdr(InstrD[31:20]), .RetiredInstr(~StallE &
~FlushE), .CSRReadData(CSRReadDataD));

   regfile rf(.clk, .WE3(RegWriteW), .A1(InstrD[19:15]), .A2(InstrD[24:20]), .A3(RdW),
.WD3(ResultW), .RD1(Rd1D), .RD2(Rd2D));

   extend ext(.Instr(InstrD[31:7]), .ImmSrc(ImmSrcD), .ImmExt(ImmExtD));

   // Pipeline Registers (D -> E)
   flopenrc #(1)  IsMulEReg       (clk, reset, FlushE, ~StallE, IsMulD,       IsMulE);
   flopenrc #(1)  MemEnEReg       (clk, reset, FlushE, ~StallE, MemEnD,       MemEnE);
   flopenrc #(1)  RegWriteEReg    (clk, reset, FlushE, ~StallE, RegWriteD,    RegWriteE);
   flopenrc #(2)  ResultSrcEReg   (clk, reset, FlushE, ~StallE, ResultSrcD,   ResultSrcE);
   flopenrc #(1)  MemWriteEReg    (clk, reset, FlushE, ~StallE, MemWriteD,    MemWriteE);
   flopenrc #(1)  JumpEReg        (clk, reset, FlushE, ~StallE, JumpD,        JumpE);
   flopenrc #(1)  BranchEReg      (clk, reset, FlushE, ~StallE, BranchD,      BranchE);
   flopenrc #(4)  ALUControlEReg  (clk, reset, FlushE, ~StallE, ALUControlD,  ALUControlE);
   flopenrc #(1)  ALUSrcEReg      (clk, reset, FlushE, ~StallE, ALUSrcD,      ALUSrcE);
   flopenrc #(1)  JalrEReg        (clk, reset, FlushE, ~StallE, JalrD,        JalrE);
   flopenrc #(1)  CSRSrcEReg      (clk, reset, FlushE, ~StallE, CSRSrcD,      CSRSrcE);

   // Datapath registers
   flopenrc #(32) RD1EReg         (clk, reset, FlushE, ~StallE, Rd1D, Rd1E);
   flopenrc #(32) RD2EReg         (clk, reset, FlushE, ~StallE, Rd2D, Rd2E);
   flopenrc #(32) PCEReg          (clk, reset, FlushE, ~StallE, PCD, PCE);
   flopenrc #(5)  Rs1EReg         (clk, reset, FlushE, ~StallE, InstrD[19:15], Rs1E);
   flopenrc #(5)  Rs2EReg         (clk, reset, FlushE, ~StallE, InstrD[24:20], Rs2E);
   flopenrc #(5)  RdEReg          (clk, reset, FlushE, ~StallE, InstrD[11:7], RdE);

   // The new shared 32-bit register replaces ImmExtEReg and CSRReadDataEReg
   flopenrc #(32) SharedDataEReg (clk, reset, FlushE, ~StallE, SharedDataD, SharedDataE);

   flopenrc #(32) PCPlus4EReg     (clk, reset, FlushE, ~StallE, PCPlus4D, PCPlus4E);
   flopenrc #(3)  Funct3EReg      (clk, reset, FlushE, ~StallE, InstrD[14:12], Funct3E);

   // Datapath Forwarding & Execute Stage
   mux3 #(32) ForwardmuxA(Rd1E, ResultW, ALUResultM, ForwardAE, SrcAE);
   mux3 #(32) ForwardmuxB(Rd2E, ResultW, ALUResultM, ForwardBE, WriteDataE);
   mux2 #(32) srcbmux(WriteDataE, SharedDataE, ALUSrcE, SrcBE);

   // Execute stage modules
   logic [31:0] ALUResult_Raw;
```

```
    cmp comparator(.SrcA(SrcAE), .SrcB(SrcBE), .Flags(FlagsE));
    alu alu(SrcAE, SrcBE, ALUControlE, ALUResult_Raw);
    mul #(32) mul(.clk, .reset, .ForwardedSrcAE(SrcAE), .ForwardedSrcBE(SrcBE),
.IsMulE(IsMulE), .Funct3E(Funct3E), .ProdE(ProdE), .MulWorking(MulWorking));

    // Select ALU result:
    logic [31:0] ALUOutE;
    always_comb
        if      (IsMulE)    ALUOutE = ProdE;          // MUL/MULH/MULHSU/MULHU
        else if (CSRSrcE)   ALUOutE = SharedDataE;    // CSR
        else                ALUOutE = ALUResult_Raw;  // normal ALU

    // Branch/Jump Target Logic
    adder pcadder(PCE, SharedDataE, BranchTargetE);
    logic [31:1] TargetUpper;
    assign TargetUpper = JalrE ? ALUOutE[31:1] : BranchTargetE[31:1];
    assign PCTargetE = {TargetUpper, 1'b0};

    // Output the final ALUResultE to the rest of the pipeline
    always_comb
        case (ResultSrcE)
            2'b11:   ALUResultE = PCTargetE;  // AUIPC
            2'b10:   ALUResultE = PCPlus4E;   // JAL / JALR
            default: ALUResultE = ALUOutE;    // LW / SW / R-Type / I-Type / CSR / MUL
        endcase

    // Execute stage actual taken logic
    logic base_flag;
    always_comb begin
        case (Funct3E[2:1])
            2'b00: base_flag = FlagsE[0]; // BEQ / BNE
            2'b10: base_flag = FlagsE[1]; // BLT / BGE
            2'b11: base_flag = FlagsE[2]; // BLTU / BGEU
            default: base_flag = 1'b0;
        endcase
    end

    // XOR with base flag for branch taken logic (BEQ vs BNE, BLT vs BGE, BLTU vs BGEU)
    assign BranchTaken = base_flag ^ Funct3E[0];


    logic ActualTakenE;
    assign ActualTakenE = (BranchE & BranchTaken) | JumpE | JalrE;

    // We mispredicted if our prediction doesn't match reality
    assign MispredictE = ActualTakenE ^ PredictTakenE;

    // If we mispredicted, where do we go?
    // If it was actually taken (but we predicted NT), go to target.
    // If it was actually NOT taken (but we predicted T), go to PC+4.
    assign RecoveryPCE = ActualTakenE ? PCTargetE : PCPlus4E;

    logic IsLoadE, IsLoadM, IsLoadW;
    assign IsLoadE = (ResultSrcE == 2'b01);

    flopenrc #(1) IsLoadMReg(clk, reset, FlushM, ~StallM, IsLoadE, IsLoadM);
    flopenrc #(1) IsLoadWReg(clk, reset, FlushW, ~StallW, IsLoadM, IsLoadW);

    // Writeback mux
    assign ResultW = IsLoadW ? ReadDataW : ALUResultW;

endmodule
```

Appendix B: Synthesis

Synthesis Code

Our processor was synthesized on SkyWater SKY130, an open-source 130nm CMOS process design kit.

```
SYNTH_MAKE_FLAGS          ?= \
            OUTPUTBASEDIR=$(WORK_DIR) \
            NAME=lynn \
            DESIGN=$(SYNTH_TOP) \
            SYNTH_FILES=$(SYNTH_FILES) \
            LIB_FILES=$(LIB_FILES) \
            PROCESSORTOP=$(PROCESSOR_TOP) \
            INCDIR=$(INC_DIR) \
            CONFIG=- \
            MOD=- \
            HDLDIR=$(PROCESSOR_DIR) \
            LYNN=1 \
            TECH=$(SYNTH_TECH) \
            MAXOPT=1 \
            FREQ=333 \
            MAXCORES=4
```

Appendix C: Top Level Diagram

