Spring 2022 EECS 151 LB FPGA Lab: Final Project Report

Meng-Hui Chou, Viraj Ramakrishnan

April 23, 2022

Contents

1 Project Functional Description and Design Requirements

Our design objective was to create a three-stage pipelined CPU that implements the RISC-V ISA. Because the memories have synchronous reads and writes, we had to place pipeline registers for other signals that would cross from one side of these memories to the other. This causes three stages to emerge: the Fetch, Execute, and Writeback stage.

1.1 Three-Stage Pipeline Structure

To make a three-stage pipelined CPU, we decided to pipeline at IMEM block and DMEM block to account for the synchronous read and write operations. To divide the first and the second stage, registers are put between all the wires between the IMEM/BIOS block and BIFshMux block. For the division of the second and the third stage, registers are placed between all the wires between the MEM block and the DBMux block.

To account for the delay of data coming from the previous stage, some signals such as, but not limited to, BISel, FlushSel, and RegWEn coming from the control logic are stored in registers before they reach the target components. All control signals that are affected by this implementation can be found in 2.1 Block Diagram, whose paths to target components are blocked by a register in the midway.

1.2 Memory Hierarchy

The memory was divided into four separate parts: BIOS, IMEM, DMEM and I/O memory. Using the upper bits of the memory address, the memory that was to be written to / read from was decided. UART was then used to read and write from the I/O Memory, allowing for interaction with devices outside of the CPU.

2 High Level Organization

When designing a big project, it is nice to kick off with a high-level road map so as to maintain abstraction, coherency, and hierarchy. As so, we started the project with drawing out a dedicated block diagram. After making a draft diagram, we then start to write up codes in Verilog. The diagram played an important role not only in debugging but also communication between partners as it could cleanly show what we had in mind and what we had on table so far.

2.1 Block Diagram

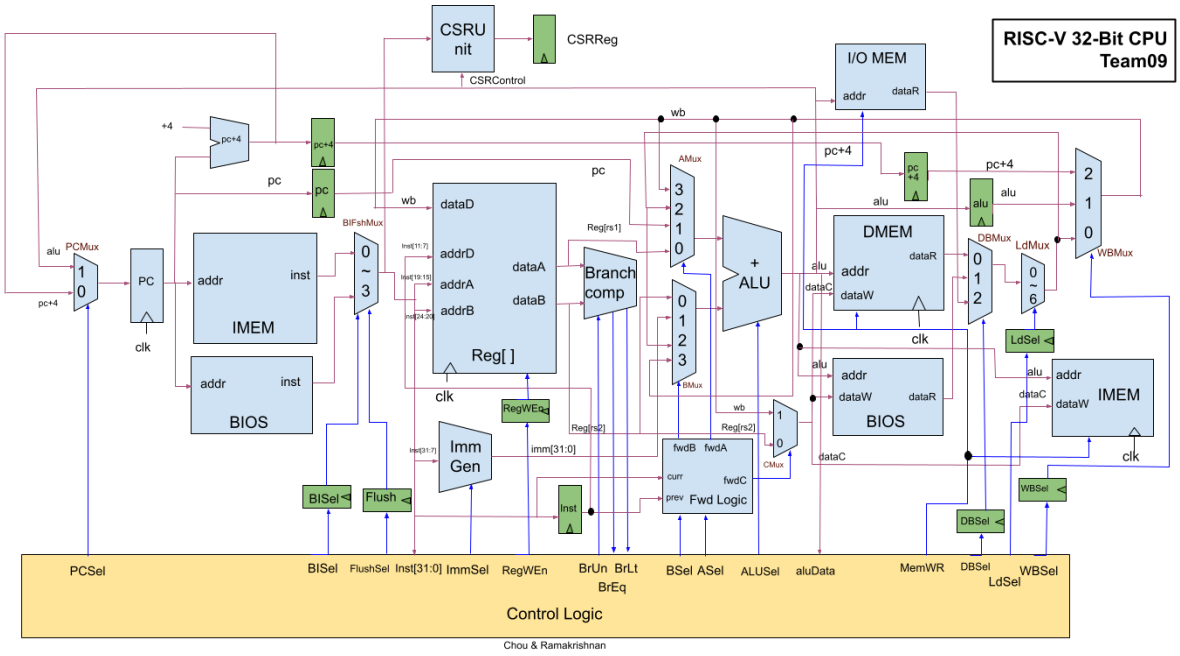

Figure 1: 32-Bit Three-Stage Pipelined RISC-V CPU

A block diagram of our designed CPU is shown in fig. 1. The diagram shows all details including modules decomposition, wiring, multiplexer, input and output signals of each module, name of each multiplexer and pipelined registers. The pipelined stages may not be clear by the first glance, but the division of the first and second stage is between IMEM/BIOS and BIFshMux and the division of the second and third stage is between the second part of MEMs and DBMux. More details can be found in 1.1 Three-Stage Pipeline Structure.

To clearly distinguish data signals and control signals, wires were labeled in blue if they are control signals while in red if they are data signals. The arrows show the direction of signals. Input signals of a module can be found if arrows point toward the module itself while output signals are arrows pointing outwards.

An overview of the block diagram, please continue reading 2.2 An Instruction's Life Cycle.

2.2 An Instrcution's Life Cycle

Based on the block diagram in fig. 1, an instruction's life cycle starts from pc reg, data outputted by PCMux will first stored here before going into the main part of the CPU, and then IMEM/BIOS will fetch the instruction based on the pc that pc register just feeds in. BIFshMux will, then, take care of choosing the right instruction for the later part or flushing the instruction due to a jumping from previous

instruction. After that, the instruction will be decomposed into pieces and be fed into different ports in the Register File, Immediate Generator, Control Logic, and CSR Unit for evaluation.

After all the hard work, Branch Comparator will take whatever dataA and dataB that are outputted from the register file and do evaluation even if the instruction is not a B type. Meanwhile, forwarding logic determines if there is a need for forwarding data due to possible hazards happening. AMux and BMux will choose the right data based on selection bits outputted by forwarding logic and send the data to ALU for further calculations. After data passes the execution stage, memories will see if there is a need for pulling or updating memory data based on instruction and data calculated by ALU. After that, DBMux will mux out the right data from memories and LdMux will mux out the right portion of data based on instruction's functionality. Finally, WBMux chooses the right data from either registers or memory and puts them back to the register file for future use.

For detailed information on what each module does, please continue reading 2.3 Module Hierarchy or 3 Detailed Description of Sub-Pieces.

2.3 Module Hierarchy

We divided the pipeline into several modules, the most important of which we have enumerated here. The most important of these will be expanded on in further detail in 3 Detailed Description of Sub-Pieces.

- *The Control Module* takes as input the instruction in the Execute stage. It sets the control flow for the rest of the datapath.
- *The Forwarding Module* takes as input the instruction from the Execute stage, and the Writeback stage, and the outputs that the Control Module supplies the AMux and BMux to select the correct inputs for the ALU. It then outputs the control logic for the AMux, BMux and CMux (necessary for controlling the data input to memories). It allows for the forwarding of data from the Writeback stage that would not otherwise have been accessible in the Execute Stage.
- *The Arithmetic Logic Unit* takes as input two numbers, and performs a mathematical operation on them.
- *The Immediate Generator* processes the instruction in the Execute stage, and outputs the correct immediate for a given instruction.
- *The Multiplexer* was an abstraction barrier we created - instead of using case statements or if statements, we encapsulated this in a module for better readability.

These submodules provided abstraction barriers that maintained the code's readability, and allowed us to easily test the functionality of different components. Furthermore, creating these modules gave us flexibility to change implementation while optimizing the CPU. More descriptions on sub-modules can be found in 3 Detailed Description of Sub-Pieces of the report.

3 Detailed Description of Sub-Modules

As mentioned in 2.3 Module Hierarchy, we decomposed our CPU into a few modules so as to maintain code readability and abstractions. More details on sub-modules are provided below.

3.1 Control Logic

This is one of the key modules in the pipeline, as it has effects across all stages, and orchestrates the operation of the CPU. In terms of our abstraction barrier, the control module provides control signals to other sub-modules. The control logic was implemented combinationally, so as to respond to an input instruction immediately. We distinguished different signals based on the *funct* and *opcode* fields of the input instruction and then fed the appropriate control signals into appropriate ports of sub-modules. For details on input and output ports of control logic, please refer to the block diagram, descriptions below for each port, and Appendix A: The Implementation of Control Signals.

3.1.1 Input Ports

- Inst: A 32-bit wide instruction decoded from the second stage.
- BrEq, BrLt: A 1-bit wide signal sent from subcircuit Branch Comparator where branch conditions are evaluated.
- aluData: A 32-bit wide data coming out from ALU. This is used to determine which memory is being written to, as well the offset of the memory access.

3.1.2 Output Ports

- PCSel: A 1-bit wide signal that is sent to PCMux to choose data between pc adder or ALU based on Inst's instruction type or the branch conditions.
- BISel & FlushSel: BISel is determined by the 30th bit of *Inst* (inst[30]). If inst[30] == 1, BISel is 1 and thus an instruction is read from BIOS. If inst[30] == 0, BISel is 0 and the IMEM instruction is chosen.

  FlushSel is determined by if there is a jump instruction. Jump instructions are B type instructions, as well as *jal* and *jalr* instructions. For B type instructions, if a branch is taken, FlushSel turns into 1 and a no-op is muxed into the instruction. As *jal* and *jalr* are mandatory jumps, a no-op is automatically muxed in.
- ImmSel: A 3-bit wide signal which is sent based on *Inst*'s instruction type.
- RegWEn: A 1-bit wide signal that is determined by the functionality of *Inst*. CPU performs a write if RegWEn is 1 while a read if it is 0.
- BrUn: A 1-bit wide signal that is determined if *Inst* is an unsigned version.
- ASel, BSel: A 2-bit wide signal that is used as selection bits for A/BMuxs, however, the implementation is slightly different from the course's. Instead of going directly to A/BMuxs, it goes to forwarding logic first so as to check if we have to modify the signal if forwarding happens.
- ALUSel: A 4-bit wide signal sent to ALU sub-circuit based on the *Inst*'s calculation requirement.

- MemWR: This 4-bit wide signal is sent in a format xxxx to determine which bytes in port *dataW* or data stored in I/O MEM we have to write into memory or UART transmitter. For example, 0000 means we do not perform any write while 1100 means we write the upper 16-bit data to the destination address.
- DBSel: A 2-bit wide signal that is sent to DBMux which allows CPU to choose data between DMEM, BIOS, and I/O MEM based on *aluData,* the destination address in this case.
- LdSel: This 3-bit wide signal is sent to choose which bytes of data at the *aluData* address we should load into the destination register based on instruction.
- WBSel: A 2-bit wide data that is used as a selection bit for WBMux to choose between data outputted from DMEM, alu register, and pc+4 register.

For the implementation of control signals, please refer to Appendix A: The Implementation of Control Signals at the end of report.

3.2 Forwarding Logic

The forwarding logic is specifically designed to eliminate all Data hazards. Structural Hazards are handled by separate memories, and sufficient ports to and from memories. Data Hazards are handled by stalling (in this current, unoptimized pipeline). After much experimentation, a few design decisions were made, listed here.

Firstly, the decision was made to have the forwarding logic take the outputs of the control logic as inputs, as opposed to controlling separate muxes in the pipeline. At a block diagram level, this allows us to interpret the Forwarding Logic as simply modifying the control logic signals to correctly handle data hazards.

Secondly, the forwarding actually only draws from one place - the wire that writes back to the Register File. Whereas some implementations of a 4 or 5-stage pipeline have separate forwarding wires for DMEM outputs and ALU outputs, our implementation only needs to pull from one place. This is a luxury afforded to us by the fact that we have a 3-stage pipeline. We are ready with the value to write back to the register in the next cycle after an instruction is in the execute stage, meaning that this is the only value we need to pull in. This value is therefore added as an option that can be muxed into the inputs of the ALU, which the Forwarding Logic can then control. This design decision was advantageous over previous attempts, because it meant that the handling for any Data Hazard was basically identical (save for *store* instructions). Anytime there was a register that had been written to in the previous cycle, we simply fed the value bound for it into the appropriate input of the ALU or memory. Special care had to be taken to identify that no-ops were not perceived as a write to the register file.

3.3 I/O Memory

We chose to implement I/O memory as a set of registers. Because of the relatively unsophisticated methods of data storage and access, it seemed simpler to maintain a set of registers to represent the

memory at the desired addresses in I/O memory. This was a useful simplification, as instead of repeated memory accesses, we only needed to write basic sequential logic to achieve the effect of an I/O memory.

3.4 Wires/Muxes/Registers

The wiring, multiplexers and registers of a CPU are essential to its functionality, and several design decisions were made here, both to improve the functionality of the system, and also to handle the complexity of the system most efficiently. The most important decisions are listed below.

3.4.1 CSR External Wiring

Since a CSR (control status register) is some state that is stored independently of the register file and the memory, we decided to handle the CSR instruction separately. The CPU takes the output instruction of BIFshMux and determines if it is a CSR instruction. If the instruction is a CSR, data will be written into a separate register (storing it into the *tohost* address) instead of ones in the register file.

3.4.2 BIFshMux

We eventually combined the BISel and FlushSel signals together for optimization and organizational purposes. This mux essentially resolves which option we are going to take: an instruction from BIOS, from IMEM, or a Flush.

3.4.3 CMux

To implement forwarding of store instructions, it was necessary to forward the value being written back into the input of the memories. Thus, we created a mux to decide whether to pass register or forwarded values into memories. The selection bit of this mux is computed from forwarding logic instead of control logic because the forwarding logic block decides whether forwarding is occurring - it makes the most sense to therefore have that control this mux, as it already controls the select bits for AMux and BMux.

3.4.4 Inst Register

This register was added to help the forwarding logic check for data hazards.We have to know at the time where the current instruction is being executed if the current instruction creates hazards with the previous instruction. With this implementation, we can get a copy of previous instruction and use it as a reference to compare with the current instruction.

3.5 Optimizations

To discover the pros and cons of different optimization choices, there are three optimizations that we made to our CPU: No-NOP, No-Forwarding, and Parallel Memory Address Calculation.

3.5.1 No-NOP

The reference diagram is shown in fig. 2. The concept of this optimization was to push CPI as low as possible (while potentially sacrificing some clock speed. The concept is as follows: because there is no separate Fetch and Register Read stage, the result of the branch comparison (and therefore whether a branch is taken or not, as well as the address we'd be jumping to) are available in the next cycle. We can therefore avoid all stalls by simply passing in the correct address into the memories.

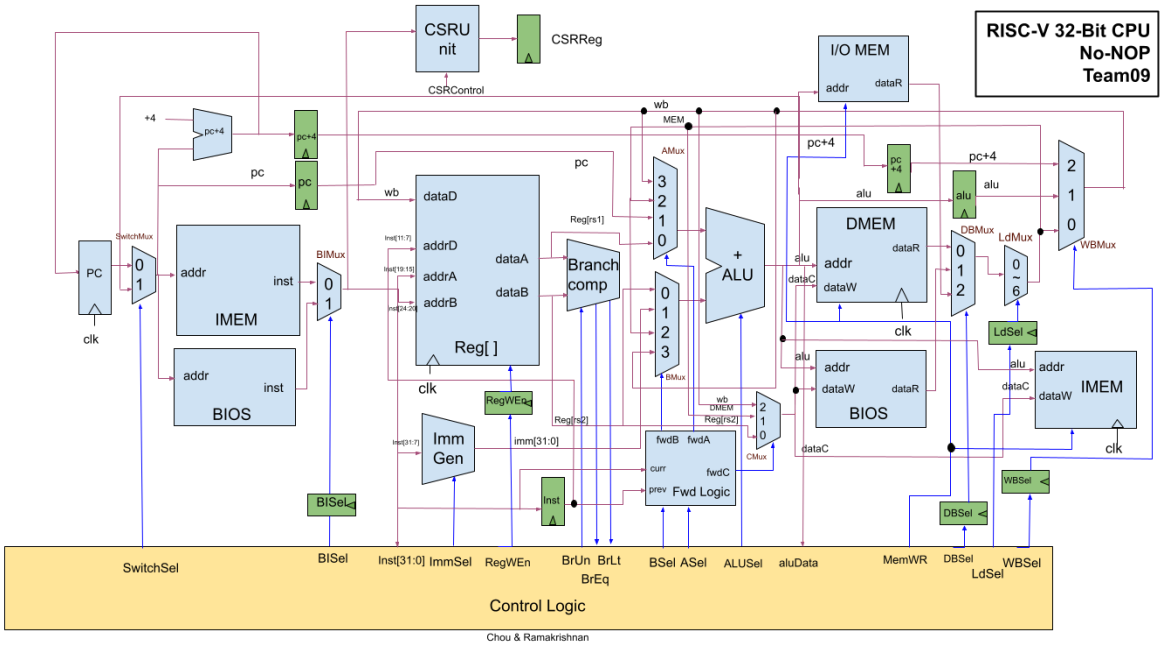

Figure 2: Our Designed CPU with No-NOP Edition

Several things changed in the pipeline to achieve this. The major difference is that now we have SwitchSel, which decides whether to use the value from the PC register, or the calculated new location from the ALU. An additional optimization was that we eliminated the flush mux, and effectively moved it into the Fetch stage. Because the Execute stage is the longest (see 4 States and Results for more details), this load balanced the stages somewhat, and improved clock frequency.

3.5.2 No-Forwarding

The reference diagram is shown in fig. 3. The concept of this optimization was to push CPI in the other direction, to increase clock speed and perhaps suffer a hit to CPI. In this optimization, we experimented with eliminating forwarding. The forwarding logic improves CPI, but it also makes paths somewhat longer: additional logic performed by the forwarding logic delays the decision of the inputs into the ALU, slowing down the Execute stage. In addition, the control signals are not ready until the writeback mux has decided the correct value to write back. By eliminating forwarding (and just stalling whenever we encounter a data hazard), we eliminate this issue.

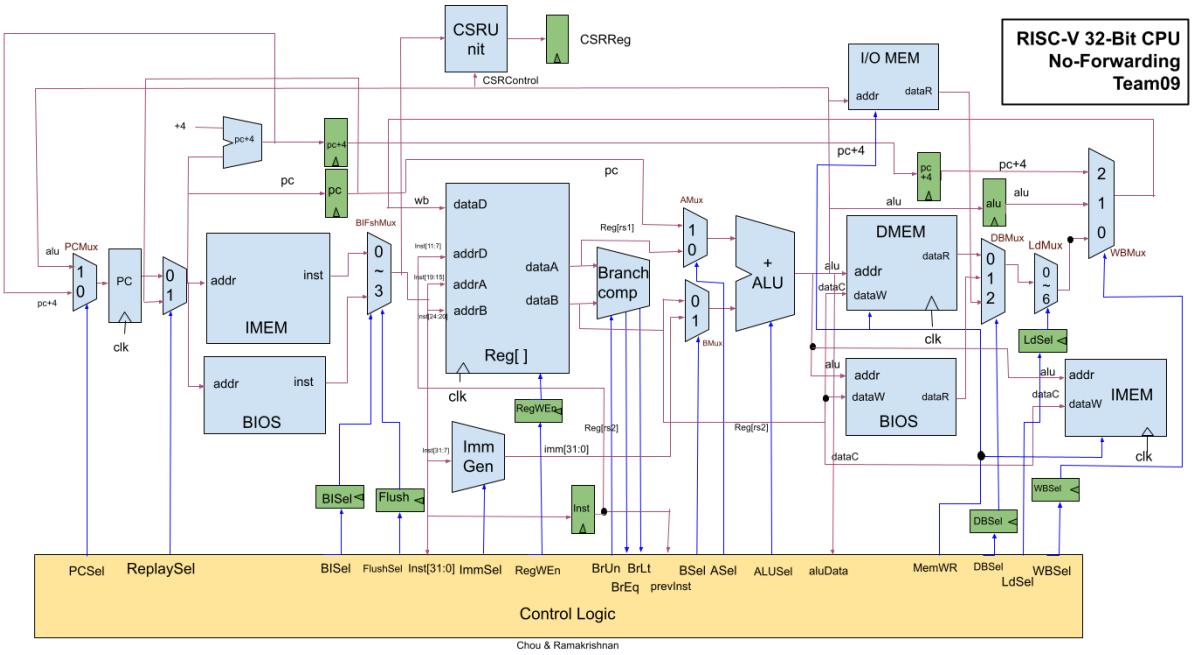

Figure 3: Our Designed CPU with No Forwarding Edition

Several things changed as a result of this design decision. Several Muxes (AMux, BMux) were shortened, and CMux was eliminated. The inputs to the AMux and BMux are also now smaller, and are available sooner. To make the system work we needed to stall on data hazards, which proved difficult to implement. The usual method of muxing in a no-op would not work, because it would lead to an infinite loop, as it is the instruction in Execute that determines whether there is a data hazard: changing this instruction removes the conflict, which un-flushes the instruction, repeating the cycle. To resolve this, we came up with a new way to no-op: defanging the instruction at the memories and Register file. By preventing the instruction from writing to anything, we discovered, it was as if the instruction never happened. We also realized that we needed to 'replay' the instruction in cases where a data hazard had been detected (and make sure that a hazard was not being created by this replay). This was achieved with a ReplaySel control signal from control logic, which overrides the PC with the value from the previous cycle.

3.5.3 Parallel Memory Address Calculation

The reference diagram is shown in fig. 4.This optimization was driven by a desire to shorten critical path times by changing resource utilization. We noticed that our critical path involved a memory address calculation, in a write to memory. To eliminate this issue, we devised a way to separately calculate the offsets of memory addresses. Because the ALU takes up so much time, if we made a simple adder in parallel, we could simply use the result of this adder when memory access instructions happen.

Crucially, to shorten the critical path, we can replace the add instruction that usually happens in the ALU when there is a memory instruction with a quicker operation, such as an or.

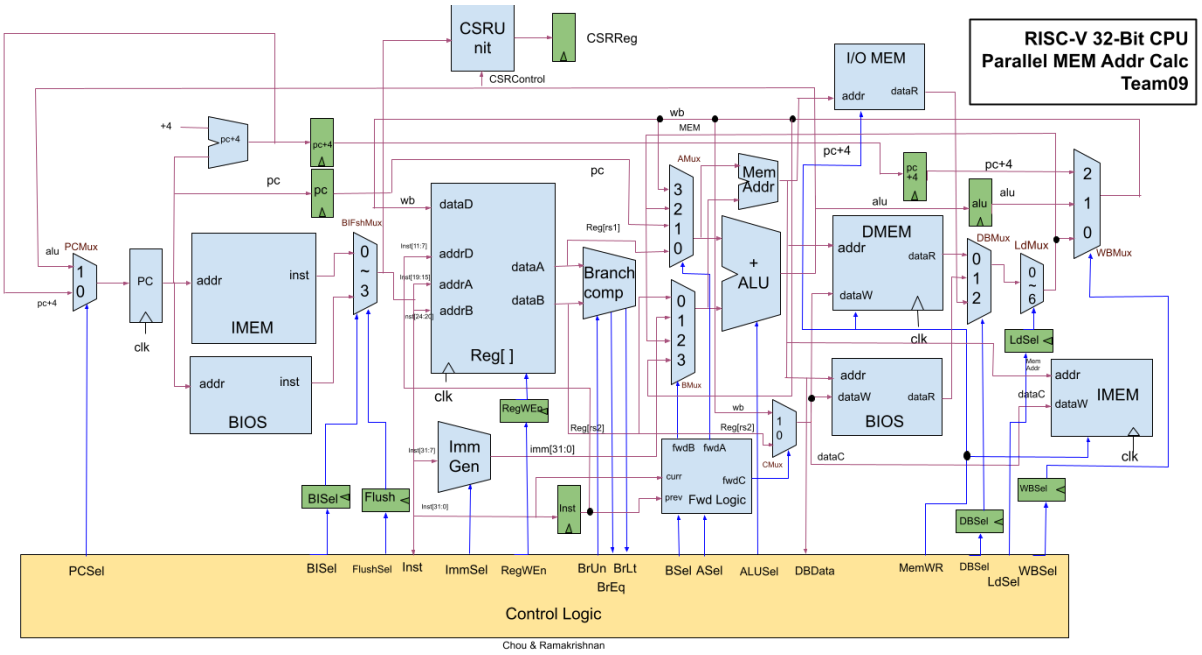

Figure 4: Our Designed CPU with Parallel Memory Address Calculation Edition

As can be seen in fig. 4, there is no wire from ALU to memory - this is entirely handled by the MemAddr, which simply adds two inputs. Care must be taken to ensure that the ALU output is properly divorced from the inputs to memories, to feel any effect of this optimization.

4 Results

In general, our CPU works in that it fulfills our design objectives. It avoids data hazards, and achieves great CPI and speed on matrix multiplication.

4.1 CPI, Clock Period and Area Utilization

For our ‘vanilla’ implementation, we achieved a PERIOD = 18. This translates to 1000000000/18 = 55.6MHz. We achieved a CPI of 1.18 with this implementation. Our utilization of resources was as follows:

- LUT Usage: 1416 (2.66%)
- Register Usage: 402 (0.38%)
- BRAM Usage: 34 (24.29%)
- DSP Usage: 0 (0%)

For our No-nop implementation, we achieved a PERIOD = 17. This translates to 1000000000/17 = 58.8MHz. We achieved a CPI of 1 with this implementation. We managed to achieve a better CPI *and* a better clock speed (due to reorganization), meaning this is a clear improvement.

For our No-forwarding implementation, we achieved a PERIOD = 16. This translates to 1000000000/16 = 62.5MHz. We achieved a CPI of 1.24 with this implementation. Although we did lose some speed to higher CPI, we made up for it with a significantly faster speed: on balance, this is 7% faster than before.

For our Parallel implementation, we achieved a PERIOD = 15. This translates to 1000000000/15 = 66.7MHz. We achieved a CPI of 1.18 with this implementation. Basically, for this one, we were able to leave our CPI unchanged - but we managed to achieve a much lower period - this translates to an improvement of 33%, which is quite significant.

4.2 Critical Path

In the 'vanilla' case, we found that the critical path was from the memory we were reading from (IMEM or BIOS), through the execute stage (ALU, etc) to a memory we were writing to (I/O MEM, DMEM, etc.). This was largely unchanged throughout the optimization process - although, when we added wires in the No-NOP implementation to implement a CPI of 1, the critical path became the memory back to the same memory (possibly because of the longer wires that were required to implement the optimization).

5 Conclusions

We have a lot to say about this project and here is the breakdown of our personal thoughts for now and the future.

5.1 Reflections

We learned a lot from this experience.

(From Viraj's side) This project is the most challenging project I have done in Berkeley so far. It really tested my ability to deal with complexity, as well as my ability to assess how much time a task will take.

It has been the case in the past that the planning phase has defined the project, and I think this was definitely the case for our project. Because both of us had a decent idea of what we wanted to create, and how we were dividing the project up, we were able to make good quality, consistent decisions that helped us build with clarity and consensus. As the project progressed, we feel we grew into the tools that had initially seemed awkward and unintuitive, making the entire process far smoother. Our discovery of how to use GTKWave remotely and access lab machines through X2Go drastically improved our efficiency, as we were able to simulate remotely and quickly.

That being said, there were definitely places for improvement. We may have held back on testing too long - because we didn't want to build out the BIOS until later, we were essentially running off of IMEM the entire time, which hampered our ability to test. Unit testing on the Forwarding and Control Logic were insufficient to find some tricky bugs. Initially, the debugging process was excruciatingly slow. It was only

as we gained experience with what bugs usually looked like that we began to make debugging decisions more quickly.

Aside from the designing and debugging process, we think that writing up a final report helps us organize ideas better and have a chance to think thoroughly if there is a better way to improve performance of the current choice. (From Meng-Hui's side) Under the time pressure, given only two weeks to make the CPU functional, we sometimes did not have time to slow down pace and really understand what our partner had coded up. However, during the write-up, I was able to catch up what I missed from my partner's part and picked up concepts that I used to take for granted.

5.2 What's Next?

If we are given one more chance to reverse our design, we want to try to combine No-NOP optimization with parallel memory address calculation optimization. We expect the CPI will be exactly 1 and the minimum clock frequency will still be 66.7MHz since we expect the critical path will stay the same, data running from IMEM to DMEM.

We think the most we have to improve is to make each stage's path more even so that a stage which completes a task earlier does not have to wait the others that long before proceeding to the next stage. We think that making our design into a four-stage pipelined CPU would be a relatively better choice in terms of speeding up clock frequency since, based on how we pipelined the CPU, there seems to be no way to make the second stage's path even shorter. We expect that a four-stage pipelined CPU which has an additional stage after the register file with parallel memory address calculation optimization could have a better performance on clock frequency.

Regarding CPI, we originally wanted to try branch prediction optimization, but knowing that this choice may not improve clock frequency and CPI and would be awkward in a three-stage pipelined CPU where the fetch stage is back to back with the execution stage, we then stopped to explore other options. However, in a four-stage pipelined CPU, implementing branch prediction is more reasonable as now the CPU must be implemented with an injecting NOP feature so as to have the right behavior when encountering jump type. As such, with branch prediction features in a four-stage pipelined CPU, we expect CPI could be improved even further.

6 Division of Labor

Documents have been submitted separately on Gradescope.

Appendix A: The Implementation of Control Signals

Category	Names of Signals	Control Bits	Actions/Comments
Inputs	Inst	none	32-Bit Instruction from Second Stage
	BrEq	0	Branch is not equal
		1	Branch is equal
	BrLt	0	Branch is not less than
		1	Branch is less than
	aluData	none	32-Bit ALU Data from second Stage
Outputs	BISel	0	IMEM
		1	BIOS
	FlushSel	0	Not flush
		1	Flush
	ALUSel	0000	_add: signedA + signedB
		0001	_sub: signedA - signedB
		0010	_sll: signedA << signedB
		0011	_slt: (signedA < signedB) ? 1: 0
		0100	_sltu: (unsignedA < unsignedB) ? 1: 0
		0101	_xor: signedA ^ signedB
		0110	_srl: signedA >> signedB[4:0]
		0111	_sra: signedA >>> signedB[4:0]
		1000	_or: unsignedA | unsignedB
		1001	_and: unsignedA& unsignedB
		1010	_passA: unsignedA
		1011	_passB: unsignedB
	RegWEn	0	Do Not Write
		1	Write
	BrUn	0	Signed

		1	Unsigned
	ASel	00	Data from [rs1]
		01	Data from pc
		10	Data from Memory
		11	Data from Write Back
	BSel	00	Data from [rs2]
		01	Data from Immediate Generator
		10	Data from Memory
		11	Data from Write Back
	ImmSel	000	R Type
		001	I Type
		010	S Type
		011	B Type
		100	U Type
		101	J Type
	MemWR	0000	Do Not Write Any Bit to Memory
		0001	Write Lower 8 Bits to Memory
		0010	Write Second Lower 8 Bits to Memory
		0100	Write Second Upper 8 Bits to Memory
		1000	Write Upper 8 Bits to Memory
		0011	Write Lower 16 Bits to Memory
		1100	Write Upper 16 Bits to Memory
		1111	Write Every Bit to Memory
	DBSel	00	Data from DMEM
		01	Data from BIOS
		10	Data from I/O MEM
	LdSel	000	Load Lower 8 Bits to Register
		001	Load Second Lower 8 Bits to Register

		010	Load Second Upper 8 Bits to Register
		011	Load Upper 8 Bits to Register
		100	Load Lower 16 Bits to Register
		101	Load Upper 16 Bits to Register
		110	Load Every Bit to Register
	WBSel	00	Data from Memory
		01	Data from ALU Register (2_3)
		10	Data from pc+4 Register (2_3)

(SPECIFICATION)

Project Functional Description and Design Requirements

Describe the design objectives of your project. You don't need to go into details about the RISC-V ISA, but you need to describe the high-level design parameters (pipeline structure, memory hierarchy, etc.) for this version of the RISC-V. (≈ 0.5 page)

High-level organization

How is your project broken down into pieces. Block diagram level-description. We are most interested in how you broke the CPU datapath and control down into submodules, since the code for the later checkpoints will be pretty consistent across all groups. Please include an updated block diagram (≈ 1 page).

Detailed Description of Sub-pieces

Describe how your circuits work. Concentrate here on novel or non-standard circuits. Also, focus your attention on the parts of the design that were not supplied to you by the teaching staff. (≈ 2 pages).

Status and Results

What is working and what is not? At what frequency (50MHz or greater) does your design run? Do certain checkpoints work at a higher clock speed while others only run at 50 MHz? Please also provide the area utilization. Also include the CPI and minimum clock period of running mmult for the various optimizations you made to your processor. This section is particularly important for non-working designs (to help us assign partial credit). (≈ 1-2 pages).

Conclusions

What have you learned from this experience? How would you do it different next time? (≈ 0.5 page).

Division of Labor

This section is mandatory. Each team member will turn in a separate document from this part only. The submission for this document will also be on Gradescope. How did you organize yourselves as a team. Exactly who did what? Did both partners contribute equally? Please note your team number next to your name at the top. (≈ 0.5 page).