

EECS 151 RISC-V CPU

Team 6: Reza Sajadiany, Peyrin Kao

December 2021

Contents

1	Project Functional Description and Design Requirements	2
2	High Level Organization	3
2.1	Diagram	3
3	Detailed Description of Submodules	4
3.1	Decoder module	4
3.2	Address space	4
3.3	Write enable	4
3.4	Forwarding module	4
3.5	Stall	4
3.6	Jal	4
3.7	FIFO	5
3.8	Audio synthesizer	5
4	Pipelining Structure	5
4.1	Forwarding	5
4.2	Stalling	5
5	Verification	5
5.1	Debugging approach	6
6	Status and Results	6
6.1	Implementation status	6
6.2	Performance	6
6.3	Optimization	7
6.3.1	Base design	7
6.3.2	Four stage pipeline	8
6.3.3	Removing the forwarding path	8
6.3.4	Microarchitectural optimizations	9
6.3.5	Delaying the UART transactions	9
6.4	Design trade-offs	9
6.5	FPGA Utilization	10
7	Conclusion	10

1 Project Functional Description and Design Requirements

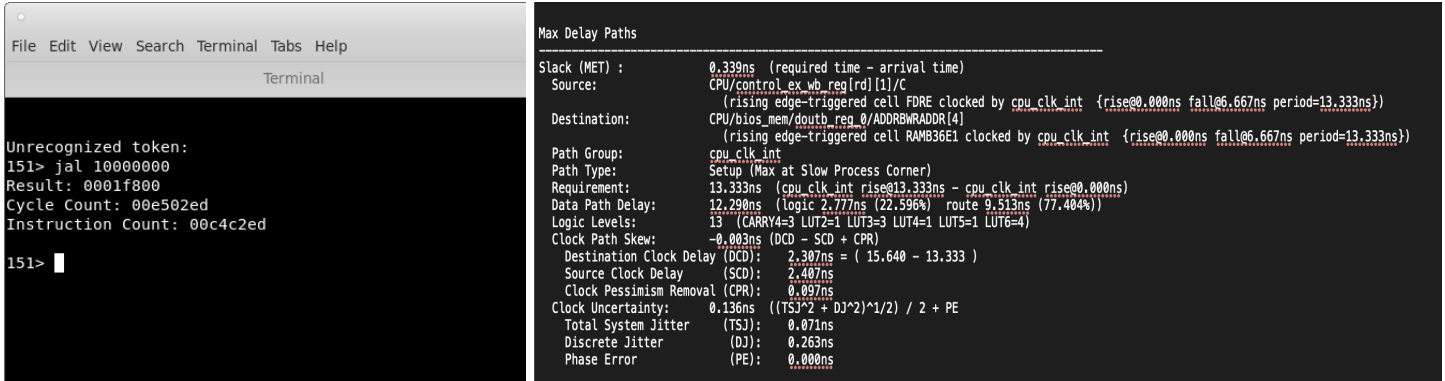
This particular implementations of the RISC-v CPU that we implemented on FPGA boards is capable of executing RISC-V base integer instructions with additional functionality for an audio accelerator. A pipelined RISC-V CPU should be able to accept instructions through BIOS and instructions transferred through UART directly into the instruction memory (IMEM). This scheme allows the FPGA board to be booted in the bios mode and be able to receive compiled hex code as a stored program into the instruction memory. In addition, the CPU should be able to read and write from the provided DMEM, BIOS, IMEM, and memory mapped IO. The CPU can read directly from I/O using the memory-mapped I/O scheme.

For performance, our CPU started as a 3 stage pipelined processor, and forwarding paths were implemented to resolve data hazards where it was possible. We considered different designs and analyzed the trade-offs in optimization step. Different micro-architectural designs had direct impact on the CPI and the frequency of our CPU. However, considering every aspect of performance and functionality, we were able to achieve 75 Mhz frequency on a 3 stage pipeline with a CPI of 1.163 for the matrix multiplication program as shown below.

In addition to the base RISC-V CPU, we also implemented an audio accelerator that can perform FM synthesis to play sounds of different tones and timbres. This part of the CPU is using a different clock frequency which is 150 Mhz. As a result, to communicate between these two clock domains, a 4-way hand shake was needed so that the data would not be corrupted.

The 75 Mhz critical path timing was met and the matrix multiply CPI:

$$CPI = \frac{15008493}{12894957} = 1.16$$



```
File Edit View Search Terminal Tabs Help
Terminal

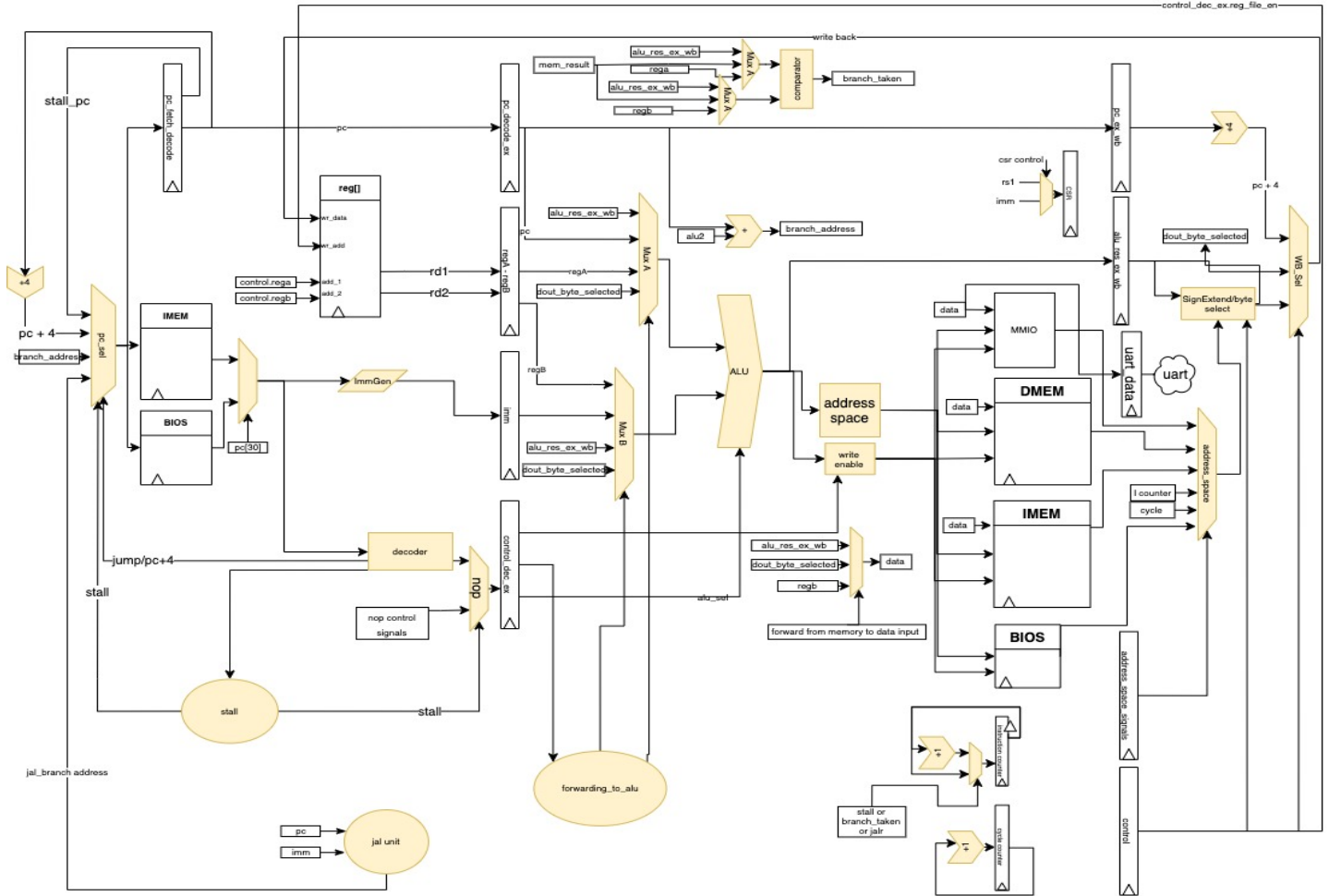
Unrecognized token:
151> jal 10000000
Result: 0001f800
Cycle Count: 00e502ed
Instruction Count: 00c4c2ed

151>

Max Delay Paths
-----
Slack (MET) : 0.339ns (required time - arrival time)
Source: CPU/control_ex_wb_reg[rd][1]/C
(rising edge-triggered cell FDRE clocked by cpu_clk_int {rise@0.000ns fall@6.667ns period=13.333ns})
Destination: CPU/bios_mem/doutb_reg_0/ADDRBWRADDR[4]
(rising edge-triggered cell RAMB36E1 clocked by cpu_clk_int {rise@0.000ns fall@6.667ns period=13.333ns})
Path Group: cpu_clk_int
Path Type: Setup (Max at Slow Process Corner)
Requirement: 13.333ns (cpu_clk_int rise@13.333ns - cpu_clk_int rise@0.000ns)
Data Path Delay: 12.290ns (logic 2.777ns (22.596%) route 9.513ns (77.404%))
Logic Levels: 13 (CARRY4=3 LUT2=1 LUT3=3 LUT4=1 LUT5=1 LUT6=4)
Clock Path Skew: -0.003ns (DCD - SCD + CPR)
Destination Clock Delay (DCD): 2.307ns = ( 15.640 - 13.333 )
Source Clock Delay (SCD): 2.407ns
Clock Pessimism Removal (CPR): 0.097ns
Clock Uncertainty: 0.136ns ((TSJ^2 + DJ^2)^1/2) / 2 + PE
Total System Jitter (TSJ): 0.071ns
Discrete Jitter (DJ): 0.263ns
Phase Error (PE): 0.000ns
```

2 High Level Organization

2.1 Diagram



To make our design modular and to make testing more reliable, we divided the CPU into several smaller modules:

- The arithmetic logic unit (ALU) performs different arithmetic operations on 32-bit values.
- An address space decoder determined whether an instruction should be writing or reading, and whether it should interact with DMEM, MMIO, IMEM, or BIOS memory.
- A byte select module interacted with word-aligned memory in order to support loading bytes and half-words from the memory.
- A comparator helped determine whether branches should be taken and evaluated the address for a taken branch.
- A decoder calculated the control logic signals for a given instruction, assigned the values to their corresponding fields of a structure.
- A forwarding module determined whether forwarding was needed.
- The immediate generator determined the correct immediate from a given instruction.
- A module calculated the address for **jal** instructions in the decode stage.
- A register file that contains 32-bit registers with two asynchronous read ports and one synchronous write port.
- A stalling module that determined whether a stall is needed to resolve a hazard.
- A write-enable module that calculated the correct write mask to use to interact with word-aligned DMEM on store instructions, and the correct value from DMEM to write back into the register on read instructions.

3 Detailed Description of Submodules

Organizing our RTL code using a CPU level package allowed us to have a cleaner RTL and made debugging easier.

3.1 Decoder module

The control signals were packed into a struct whose fields were used in the decode and execute stage. By packing the control signals into a struct, the entire set of control signals could easily be pipelined or passed around the CPU as a group without managing many independent wires and flops. The fields in this struct are listed below. Note we had another struct that packed the signals needed for write-back stage only:

```
typedef struct packed {
    reg [3:0] alu_sel;           // selects ALU operation
    reg      alu_a_sel;         // selects first input to ALU
    reg      alu_b_sel;         // selects second input to ALU
    reg [2:0] wb;               // selects whether to write back ALU, PC+4, or memory data
    reg [1:0] mem_rw;           // selects memory read, write or neither
    reg      reg_file_wen;      // selects whether to write to register
    reg      csr_en;            // selects whether instruction is CSRW
    reg      csr_reg_imm;       // selects whether instruction is CSRW or CSRWI
    reg [2:0] byte_half_full;    // byte == 100, half == 010, full word == 001
    reg      unsigned_load;      // selects whether to perform a signed or unsigned load
    reg [2:0] imm_sel;          // selects which bits of the instructions are used in the immediate
    reg      branch_unsign;     // selects whether the branch comparison is unsigned
    reg      branch;            // selects whether the branch is taken
    reg [2:0] branch_sel;       // selects which type of branch instruction to check for
    reg [4:0] rega;              // selects the first register rs1
    reg [4:0] regb;              // selects the second register rs2
    reg [4:0] rd;                // selects the destination register
    reg      jal;                // selects whether the instruction is jal
    reg      jalr;               // selects whether the instruction is jalr
} Decoder_t;
```

3.2 Address space

The address space uses the last nibble of the address to determine whether the CPU should read or write from DMEM, IMEM, or I/O. Note that IMEM is write-only and BIOS is read-only, so reads and writes respectively had to be disabled in those cases.

3.3 Write enable

The write-enable module checked the bottom 2 bits of the address and a control signal indicating what type of load or store was being performed (word, half-word, or byte, and unsigned or signed) and selected a write mask for writes and a 32-bit data value to write into the corresponding mmemory unit.

3.4 Forwarding module

To make our debugging process easier, we implemented the logic needed for forwarding path to ALU inputs into a separate module. This module decided whether or not forwarding is needed and outputs the select bits for the ALU input mux.

3.5 Stall

The logic for stalls resides in the corresponding module. The only stalls that occurs for our data path is the stall due to a load from memory and using the data from that load to execute a branch instruction or a `jalr` instruction.

3.6 Jal

The `jal` module skips the execute stage to add the PC and the immediate earlier in the pipeline. This helps to jump immediately after fetching the `jal` instruction.

3.7 FIFO

The FIFO design is based on the conventional read/write pointers. Using an extra bit for the pointers allowed us to differentiate between empty and full status of the FIFO. The usage of FIFO in our design was to store the button presses on the FPGA board.

3.8 Audio synthesizer

Our implementation of the audio accelerator is capable of up to 4 voices to be used for the synthesizer. The synthesis module was pipelined to 4 stages to meet the 150 Mhz timing requirement. The module is designed with the use of `N_VOICES` parameter such that 1-4 voices can be synthesized.

4 Pipelining Structure

We implemented a 3-stage pipeline. The three stages are:

- **Instruction fetch**
The instruction fetch stage gets the appropriate instruction based on the address from IMEM or BIOS. We store the PC that was just used to fetch an instruction to propagate it down the pipeline.
- **Instruction decode**
The instruction decode stage uses the decoder module to calculate the control logic signals for the given instruction. We also would resolve the address for `jal` instructions in this stage and jump immediately. immediates are generated and the register file access occurs in this stage. In addition, if there is a need to stall the CPU and re-do the same instruction, we deliver `NOP` signals rather than the actual signals to the control register.
- **Execute and memory**
The execute and memory read/write stage contains the ALU, the comparator, the modules to interact with memory, and connections to the DMEM, IMEM, and BIOS sections for memory interaction. The memory mapped IO registers such as LED, FCW's (for audio synthesis), and instruction and cycle counters reside in the execute/memory stage.

4.1 Forwarding

Forwarding paths were implemented from the stored result of the ALU in the register between the execute and write-back to the input of the ALU in case of a data dependency between two instructions. Forwarding paths were also needed from the output of the memory (`dout_bytes_selected`) to the input of the ALU and comparator. To improve on our critical path we decided to avoid forwarding paths from memory as an effort to optimize our frequency. However, we later decided on including those paths for the benefit of lower stalls (based on the performance of our `matmult`).

4.2 Stalling

There are only two cases that we had to stall the CPU by 1 cycle: a `jalr` after a `lw/lh/lb` with the `jalr` needing the data from the load. And a branch after a load where the branch depends the data from the load. In those two cases, we would stall for 1 cycle for the memory read to finish.

5 Verification

We began by writing Verilog testbenches to test the smaller modules (e.g. ALU, immediate generator, decoder, etc.). After those were passing, we moved on to the provided Verilog testbench for the entire CPU. This helped us find bugs in our CPU and submodules (despite modularization, the CPU was still the most complicated module of the design). When those were passing, we wrote a few additional tests in the CPU testbench to test additional hazards, and using those tests, we fixed our CPU further.

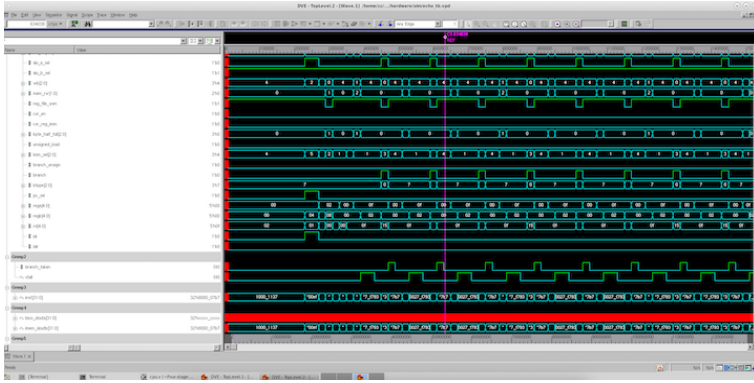
Once all Verilog testbenches passed, we moved to the provided RISC-V ISA tests. Some of these ISA tests passed immediately after the CPU testbench passed, while others failed. For the instructions that failed, we wrote custom assembly tests to further investigate why they were failing, as the provided ISA tests were difficult to debug.

Once the ISA tests passed, we moved to the provided C tests. Because the ISA tests were fairly comprehensive, no debugging of the C tests was necessary.

The most helpful debugging tools were viewing waveforms in DVE and using `$display` statements in the Verilog testbenches to log values during testbench execution.

5.1 Debugging approach

To debug our design, we mainly used the waveform tool (DVE) and traced down the signals that were causing the issues.



One of the most important key observations that we realized throughout the debugging cycles is to avoid having logic that creates X values intentionally. The use of X values became mainly for the cases such as a case statement default case, which we knew should never happen. For instance, the input of the ALU has a mux that should never become an X value. As a result, at any opint that we saw X signals in our waveform, we knew there had to be a bug causing it.

6 Status and Results

6.1 Implementation status

Our CPU passes all the required functionality listed below:

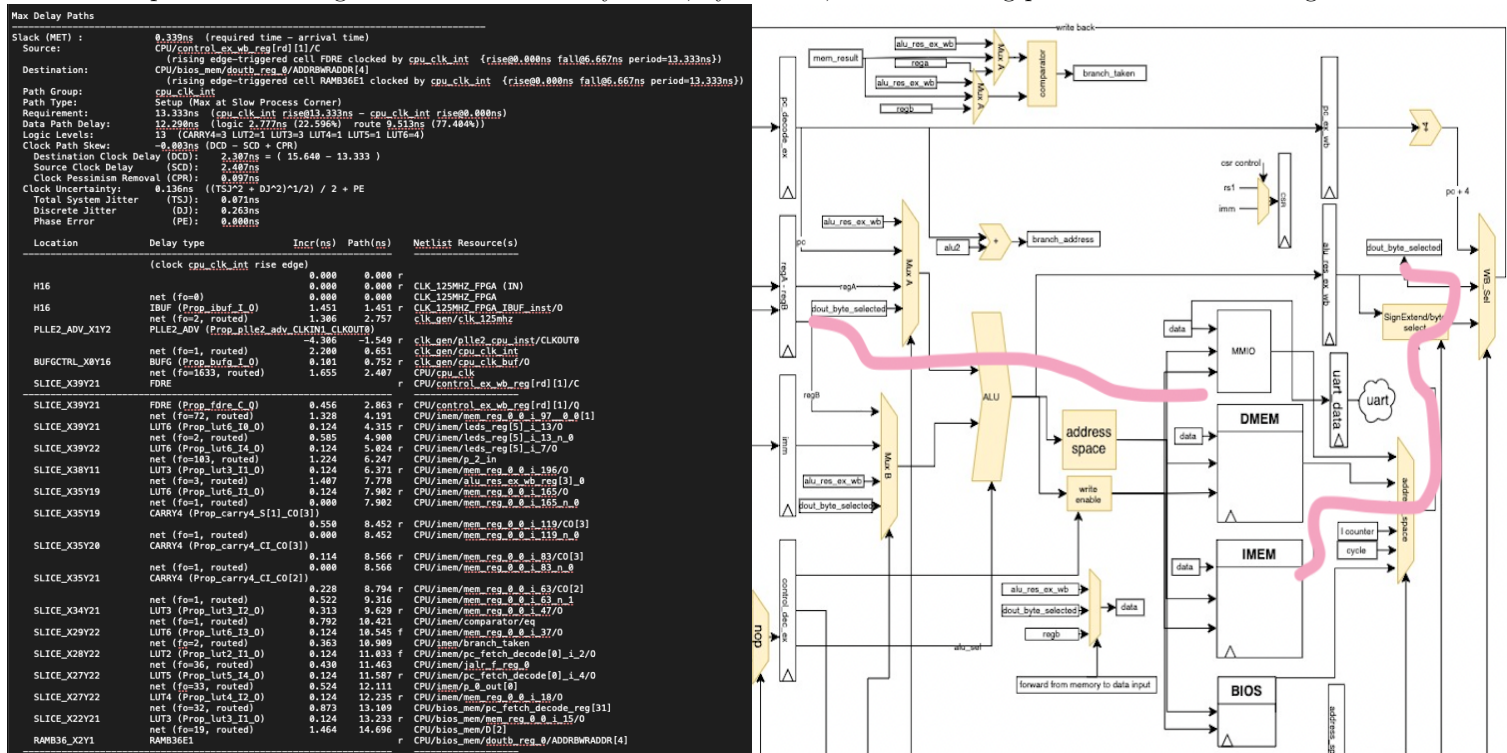
cpu_tb, asm_tb, isa-tests, tests, echo_tb, bios_tb, bios on the FPGA, mmult on the FPGA, user_io_test on the FPGA, piano on the FPGA.

To ensure the functionality of our CPU and sub-modules, we also have written test benches dedicated to sub-modules. We generated some assembly tests and we pass all the tests.

6.2 Performance

Our latest design runs at 75 MHz. Three or more attempts at optimization changed the frequency, but 75 MHz is the highest frequency we were able to obtain without sacrificing the performance of our `matmult`.

The critical path of our design is located at memory reads, byte select, and forwarding paths to the execute stage.



6.3 Optimization

We considered three different designs for optimization purposes after the first (base) design:

6.3.1 Base design

To optimize our CPU in a way that would be apparent and result in a better performance, we opted to minimize our branch and jump penalties. This approach landed itself to a design that might not be considered a usual RISC-V pipeline core. We decided to place the branch comparator in the decode stage so that the branch result is determined as soon as decode stage. Having a jump unit in the decode stage was another idea to minimize the number of cycles that we have to stall in order to branch/jump to the appropriate PC. The jump unit is responsible for adding the PC to the immediate value to compute the PC where we jump. This 3 stage pipeline, was designed with the purpose of making the Instruction Per Cycle as high as possible.

Some of the most interesting aspect of this design came into play after the realization that in order to be able to jump and branch correctly, we need to resolve the hazards that might exist for branch and jalr instructions.

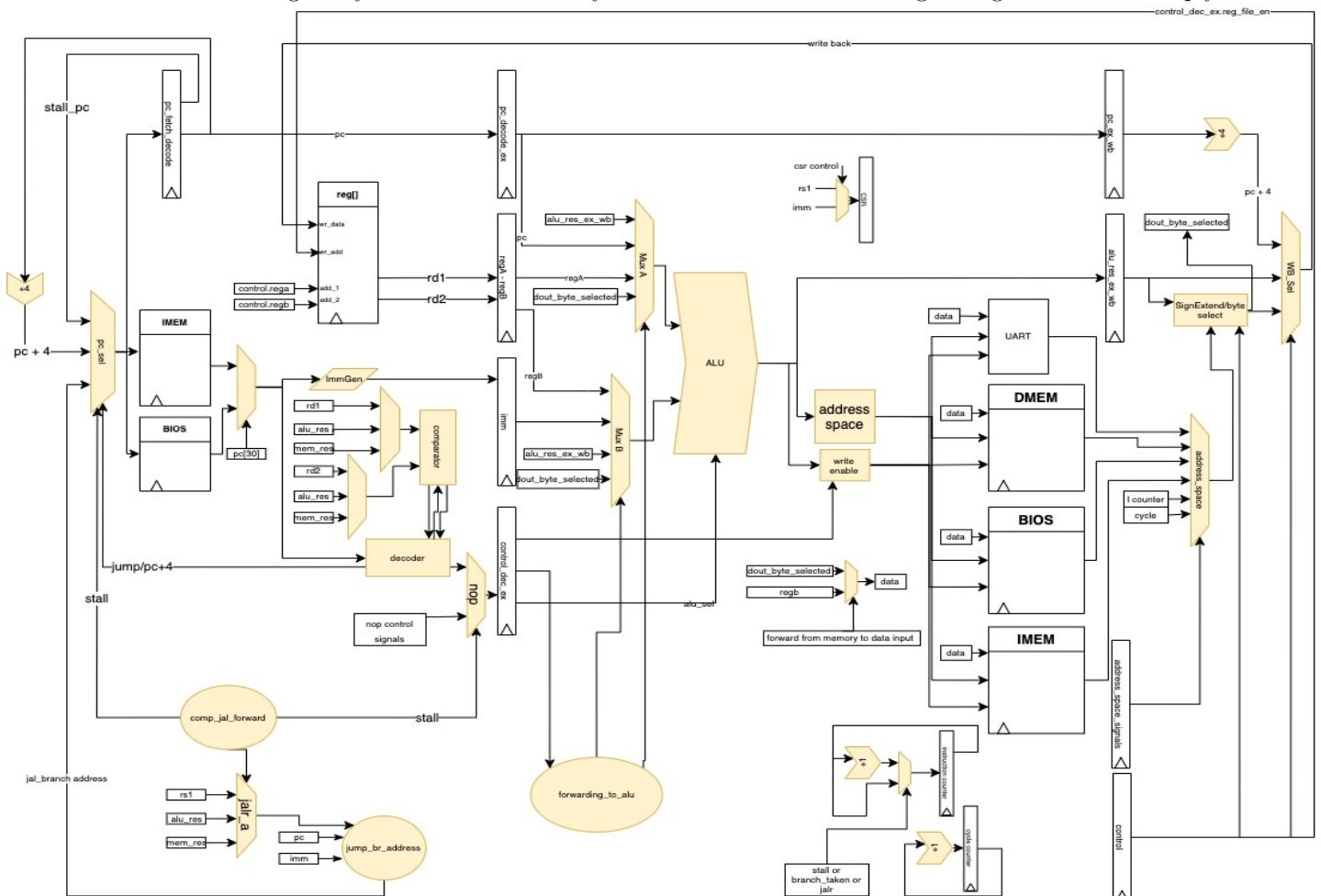
For example:

```
addi x1 x2 x3
jalr rd x1 0
```

or in some cases, it is actually not possible to forward with our design and a stall is inevitable:

```
lw  x2, 0(x3)
jalr rd, x2, 0
```

The same kind of hazards would also occur for branch instructions. In our design we would know the result of the ALU for any kind of ALU dependency, but the result of memory reads are not available at that point in the decode stage. In those cases, we would stall and wait for the memory read to be available and forward that result to the jump unit and comparator. We were able to make this design fully functional with a very low number of stalls occurring during the matrix multiply workload.



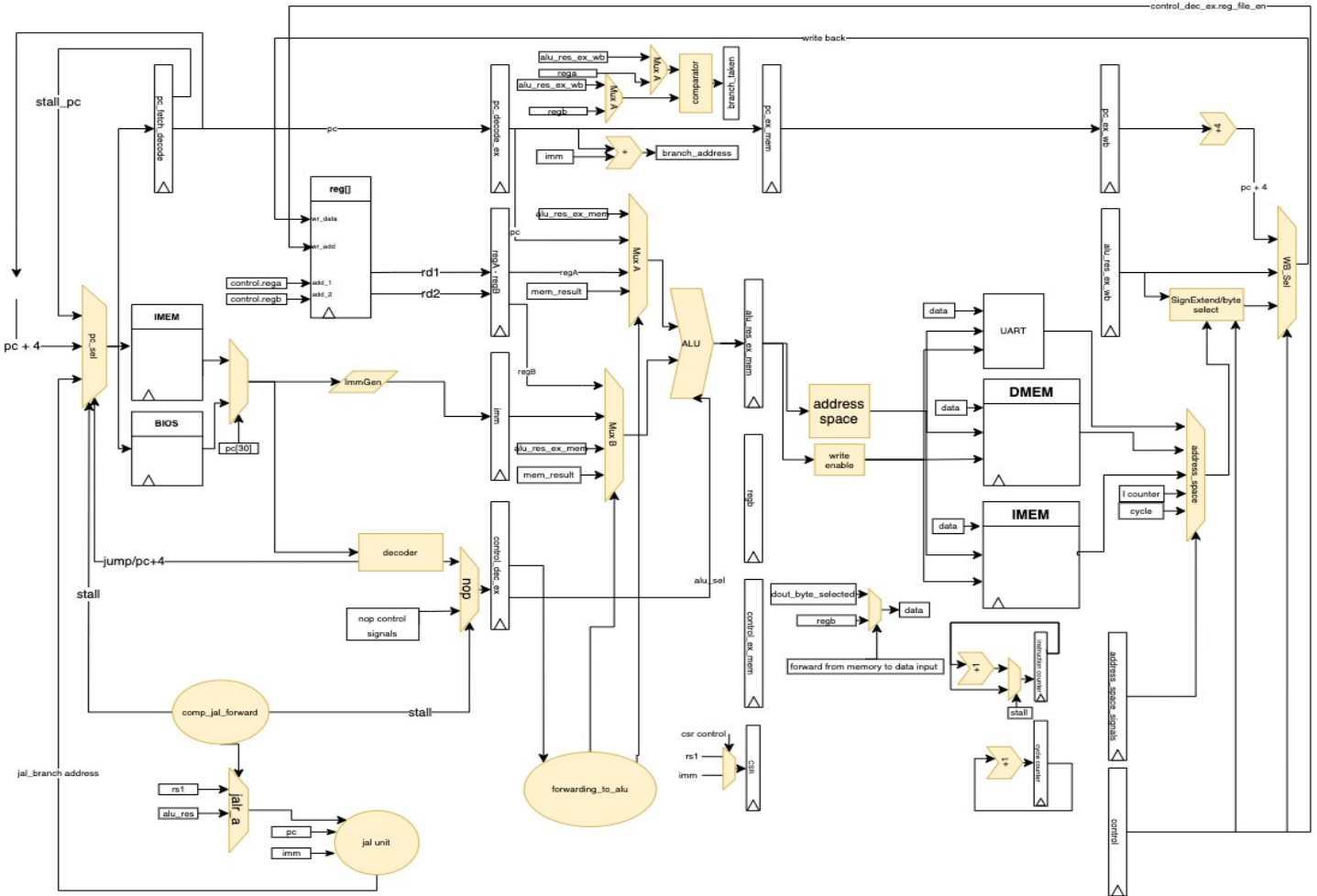
The base design performance was limited due to the critical path created by the forwarding lines.

$$\frac{\text{cycle}}{\text{instruction}} = \frac{12894957}{12894956} = 1.00$$

The most important flaw of this design was its long critical path of 20 nanoseconds. This critical path could only achieve a frequency of 50 Mhz.

6.3.2 Four stage pipeline

After realizing our critical path, we designed a 4-stage pipeline so that our critical path is broken into two stages. We split the execute/memory stage to two separate stages. The 4-stage implementation should be available to checkout in our git history:



```

Max Delay Paths
-----
Slack (MET) : 0.026ns (required time - arrival time)
Source:      cpu/control_mem_wb_reg[rd1][0]/C
              (rising edge-triggered cell FDRE clocked by cpu_clk_int {rise@0.000ns
Destination:  cpu/imem/mem_reg_2_3/ADDRBWRADDR[8]
              (rising edge-triggered cell RAMB36E1 clocked by cpu_clk_int {rise@0.000ns
Path Group:   cpu_clk_int
Path Type:    Setup (Max at Slow Process Corner)
Requirement:  16.667ns (cpu_clk_int rise@16.667ns - cpu_clk_int rise@0.000ns)
Data Path Delay: 15.880ns (logic 4.105ns (25.850%) route 11.775ns (74.150%))
Logic Levels: 20 (CARRY4=6 LUT2=1 LUT3=1 LUT4=2 LUT5=2 LUT6=8)
Clock Path Skew: -0.063ns (DCD - SCD + CPR)
Destination Clock Delay (DCD): 2.355ns = ( 19.022 - 16.667 )
Source Clock Delay (SCD): 2.415ns
Clock Pessimism Removal (CPR): -0.004ns
Clock Uncertainty: 0.132ns ((TSJ*2 + DJ*2)^1/2) / 2 + PE
Total System Jitter (TSJ): 0.071ns
Discrete Jitter (DJ): 0.253ns
Phase Error (PE): 0.080ns

```

```

File Edit View Search Terminal Tabs Help
Terminal
Unrecognized token:
151> jal 10000000
Result: 0001f800
Cycle Count: 00e502ed
Instruction Count: 00c4c2ed
151>

```

This design did not help with the critical path since at that point we noticed that the critical path is after the reads from the memory and throughout the forwarding paths. As a result, we decided on the 3-stage pipeline.

6.3.3 Removing the forwarding path

To reduce the critical path length, we removed the forwarding path from the output of the memory to the input of the ALU and the comparator. This way we did not have any load on the output of the byte select other than the write back mux. This optimization increased our frequency from 70 to 75 Mhz. The critical path was indicating a path that existed in the

decode stage. The path included the stall logic, and the mux that was placed to choose between instruction memory and bios output from the fetch stage.

Critical path: 13.1 ns (paths from imem, to stall and back to bios and imem).

$$CPI = \frac{\text{cycle}}{\text{instruction}} = \frac{12894957}{15216070} = 1.19$$

6.3.4 Microarchitectural optimizations

To avoid high fanouts that were present as part of netlists in the timing report, we decided to use duplicate units and flops and divide the load between the copies. As an example, we added a separate address adder to avoid the mux that is selecting in the output of the ALU. The address adder would only feed into the memory units.

In addition, we duplicated the flops that were used throughout the datapath. Using the first copy for half of the load and the second copy for the rest of the load. This way we lowered the capacitance load that one particular flop or combinational logic had to load.

We also searched through the Xilinx user guide for optimization flags that optimizes for low fanout. In the case that we missed a net with a high fanout, the routing and placement optimization would duplicate the element appropriately. The flag used was `-fanout_opt` in the `impl.tcl` script.

We also realize that some of the `if/else if` statements were implemented as priority blocks, which is not the intended design. Therefore, we went through many of our `if/else if` statements and changed them to be `unique case` blocks that would infer a multiplexor.

6.3.5 Delaying the UART transactions

For our final attempt to improve the critical path, we delayed all the UART transactions by one cycle. All the UART writes would happen in the write back stage. This reduced the UART sections from our critical path and achieved the final 75 Mhz frequency without hurting the performance in any way. $CPI = \frac{15008493}{12894957} = 1.16$

```

File Edit View Search Terminal Tabs Help
Terminal

Unrecognized token:
151> jal 10000000
Result: 0001f800
Cycle Count: 00e502ed
Instruction Count: 00c4c2ed

151> █

Max Delay Paths
-----
Slack (MET) : 0.339ns (required time - arrival time)
Source: CPU/control_ex_wb_reg[rd][1]/C
(rising edge-triggered cell FDRE clocked by cpu_clk_int {rise@0.000ns fall@6.667ns period=13.333ns})
Destination: CPU/bios_mem/doutb_reg_0/ADDRBWRADDR[4]
(rising edge-triggered cell RAMB36E1 clocked by cpu_clk_int {rise@0.000ns fall@6.667ns period=13.333ns})
Path Group: cpu_clk_int
Path Type: Setup (Max at Slow Process Corner)
Requirement: 13.333ns (cpu_clk_int rise@13.333ns - cpu_clk_int rise@0.000ns)
Data Path Delay: 12.290ns (logic 2.777ns (22.596%) route 9.513ns (77.404%))
Logic Levels: 13 (CARRY4=3 LUT2=1 LUT3=3 LUT4=1 LUT5=1 LUT6=4)
Clock Path Skew: -0.003ns (DCD - SCD + CPR)
Destination Clock Delay (DCD): 2.307ns = ( 15.640 - 13.333 )
Source Clock Delay (SCD): 2.407ns
Clock Pessimism Removal (CPR): 0.097ns
Clock Uncertainty: 0.136ns ((TSJ^2 + DJ^2)^1/2) / 2 + PE
Total System Jitter (TSJ): 0.071ns
Discrete Jitter (DJ): 0.263ns
Phase Error (PE): 0.000ns

```

6.4 Design trade-offs

Throughout the optimization steps, we realized that we need to consider the trade-offs of our one implementation over the other. Our first design is an example of such scenario where we optimized the CPI, but at the cost of a large critical path. We also got rid of forwarding paths at some stage in our optimization efforts. The frequent stalls after removing the forwarding path did not help the performance of our CPU.

6.5 FPGA Utilization

8. Primitives

Ref Name	Used	Functional Category
FDRE	1889	Flop & Latch
LUT6	1293	LUT
LUT5	382	LUT
LUT2	345	LUT
LUT3	261	LUT
MUXF7	256	MuxFx
LUT4	233	LUT
MUXF8	128	MuxFx
CARRY4	122	CarryLogic
RAMB36E1	34	Block Memory
LUT1	14	LUT
OBUF	9	IO
IBUF	8	IO
RAMD32	6	Distributed Memory
FDSE	4	Flop & Latch
BUFG	4	Clock
RAMS32	2	Distributed Memory
PLLE2_ADV	2	Clock
RAMB18E1	1	Block Memory

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	-	-	-
0	-	-	Set
0	-	-	Reset
0	-	Set	-
0	-	Reset	-
0	Yes	-	-
0	Yes	-	Set
0	Yes	-	Reset
4	Yes	Set	-
1889	Yes	Reset	-

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	1071	0	0	13300	8.05
SliceCL	782	0	0	53200	4.26
SliceCM	289	0	0	53200	4.26
LUT as Logic	2265	0	0	53200	4.26
using 05 output only	0	0	0	53200	4.26
using 06 output only	2802	0	0	53200	4.26
using 05 and 06	263	0	0	53200	4.26
LUT as Memory	4	0	0	17400	0.02
LUT as Distributed RAM	4	0	0	17400	0.02
using 05 output only	0	0	0	17400	0.02
using 06 output only	0	0	0	17400	0.02
using 05 and 06	4	0	0	17400	0.02
LUT as Shift Register	0	0	0	17400	0.02
Slice Registers	1890	0	0	106400	1.78
Register driven from within the Slice	479	0	0	106400	1.78
Register driven from outside the Slice	1411	0	0	106400	1.78
LUT in front of the register is unused	1007	0	0	106400	1.78
LUT in front of the register is used	404	0	0	106400	1.78
Unique Control Sets	67	0	0	13300	0.50

* * Note: Available Control Sets calculated as Slice * 1, Review the Control Sets Report for more information regarding control sets.

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	2269	0	0	53200	4.27
LUT as Logic	2265	0	0	53200	4.26
LUT as Memory	4	0	0	17400	0.02
LUT as Distributed RAM	4	0	0	17400	0.02
LUT as Shift Register	0	0	0	17400	0.02
Slice Registers	1890	0	0	106400	1.78
Register as Flip Flop	1890	0	0	106400	1.78
Register as Latch	0	0	0	106400	0.00
F7 Muxes	256	0	0	26600	0.96
F8 Muxes	128	0	0	13300	0.96

7 Conclusion

Our project was successfully finished. We achieved all of the goals that we set up at the beginning of the phase 1. Our CPU is functionally correct and performs well for the workloads that we tested such as matrix multiply. As a team, we worked on the project together and learned to make design choices and decide on trade-offs.