3-stage Pipelined RISC-V CPU with UART/MMIO, BHT-based Branch Predictor and Polyphonic Audio Synthesizer

Animesh Agrawal, Raghav Gupta

December 2021

1 Project Design

The core project requirement involved creating a 3-stage, in-order, RISC-V (RV32I) CPU. The CPU avoids stalls on data hazards via forwarding and uses a static branch prediction scheme to minimize stalling on control hazards. Instructions can be fetched from both a BIOS, which is initialized with instructions that allow the CPU to receive user programs over the UART, and the quintessential instruction memory (IMEM), into which the BIOS stores the received program.

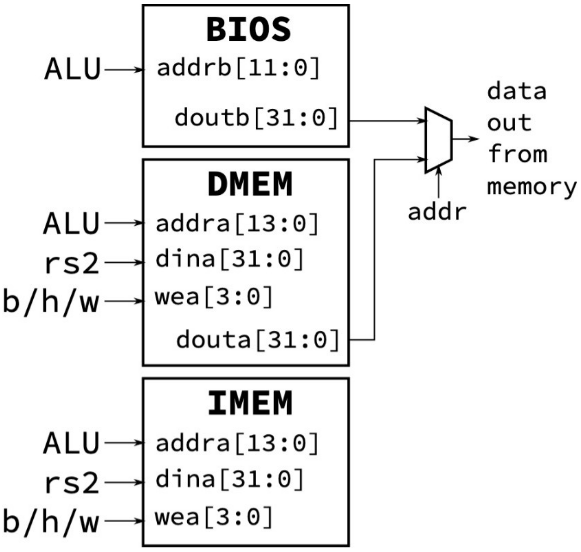


Figure 1: CPU Data Memory stage. The DMEM block refers to the quintessential data-memory available for user programs.

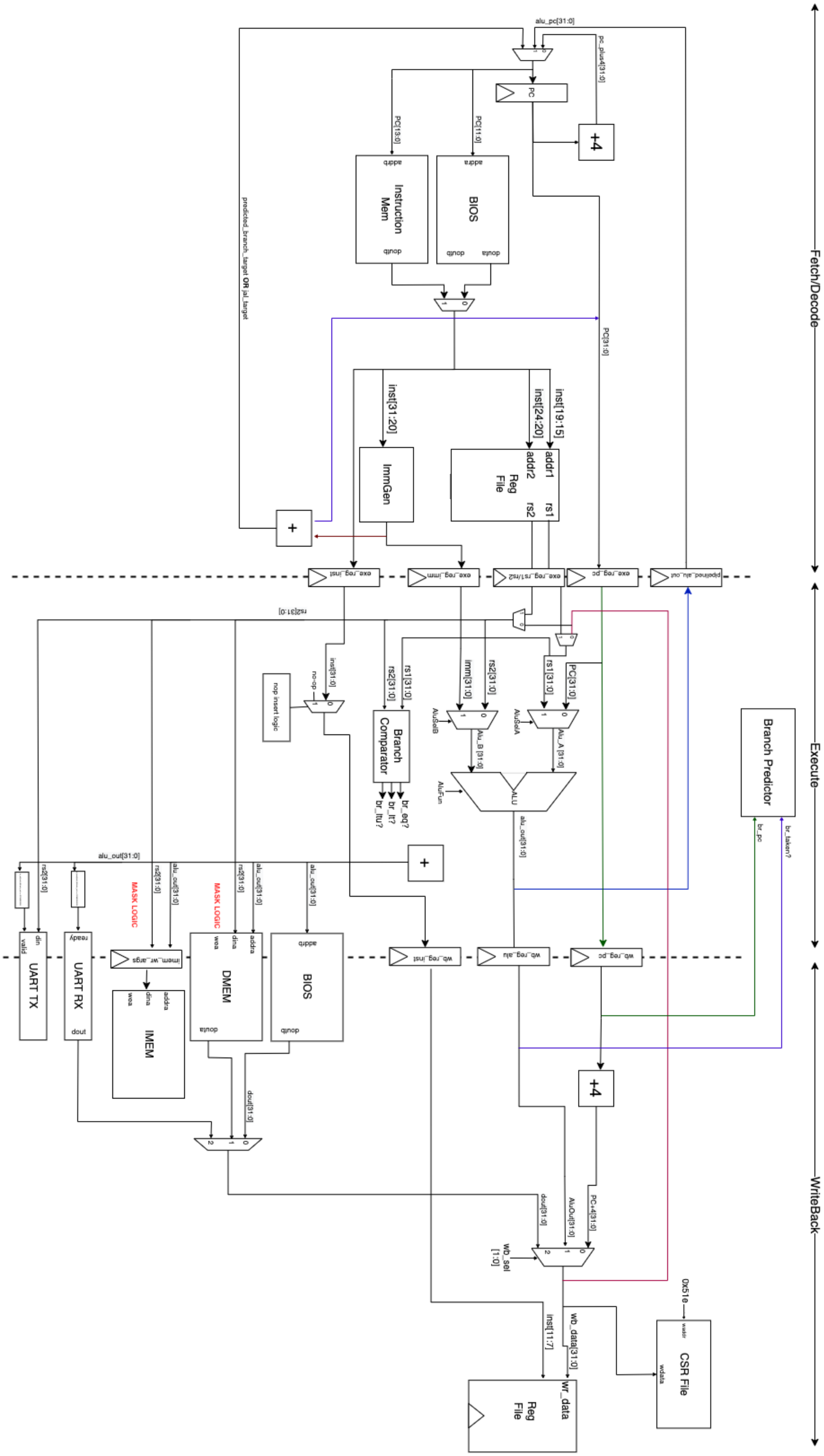
To enable this functionality, the CPU allows for writes into the IMEM by exposing an IMEM write port in the data memory (DMEM) portion of the pipeline. To allow the BIOS program to access its static data, a read port for the BIOS memory is also exposed in the DMEM portion of the CPU pipeline. All memories are synchronous for both reads and writes, and the UART is exposed as a memory-mapped-io (MMIO) device. With this design, the BIOS instructions can poll the UART for data via the UART control signals, writing the received data to the instruction memory via the write port exposed in DMEM. Once all instructions for the user program have been received, a jump instruction can set the program counter (PC) to the start of the IMEM to begin user program execution.

Finally, the CPU supports two CSR instructions (`csrw` and `csrwi`) that enable interaction with the `tohost` CSR address. This limited CSR functionality is sufficient to allow the CPU to run the RISC-V ISA test suite.

2 High-Level Organization

The CPU datapath, including the CPU's interface with memory and MMIO devices, is implemented directly in the `cpu` submodule in `cpu.v`. The control logic signals for each of the three cpu stages (instruction fetch and decode, execute, and writeback) are implemented in separate submodules for each stage. The immediate generator, branch comparator, and ALU are also implemented as separate modules to enable easier unit testing of these error-prone components. The branch predictor is also implemented in a separate submodule, `branch_predictor.v` and is connected to the fetch/decode and writeback stages of the CPU. A detailed block diagram of the CPU can be found on the next page.

Throughout the report, two different CPUs may be referenced – one with a branch predictor and one without a branch predictor. Both CPUs have the same core datapath design, but the CPU without branch prediction can reach a higher maximum clock frequency at the expense of CPI. Since adding the branch predictor increased the critical path, we introduced additional optimizations after adding the branch predictor to either further decrease CPI or to attempt to decrease the (now increased) critical path. For comprehensiveness, we discuss both designs and indicate where they differ throughout the report.


Fetch/Decode
Execute
WriteBack
alu_pc[31:0]
pc_plus4[31:0]
PC
+4
PC[13:0]
PC[11:0]
Instruction Mem
addrb
doutb
BIOS
addra
douta
doutb
predicted_branch_target OR jal_target
PC[31:0]
inst[31:20]
inst[24:20]
inst[19:15]
ImmGen
addr1
addr2
Reg File
rs1
rs2
+
exe_reg_inst
exe_reg_imm
exe_reg_rs1/rs2
exe_reg_pc
pipelined_alu_out
rs2[31:0]
inst[31:0]
no-op
nop insert logic
rs1[31:0]
rs2[31:0]
Branch Comparator
br_eq?
br_lt?
br_ltu?
imm[31:0]
rs2[31:0]
rs1[31:0]
PC[31:0]
AluSelB
AluSelA
Alu_B [31:0]
Alu_A [31:0]
AluFun
ALU
alu_out[31:0]
Branch Predictor
br_pc
br_taken?
+
alu_out[31:0]
rs2[31:0]
alu_out[31:0]
MASK LOGIC
rs2[31:0]
alu_out[31:0]
MASK LOGIC
alu_out[31:0]
din
valid
UART TX
ready
UART RX
dout
imem_wr_args
IMEM
addra
dina
wea
DMEM
addra
dina
wea
douta
BIOS
addrb
doutb
wb_reg_inst
wb_reg_alu
wb_reg_pc
+4
dout[31:0]
dout[31:0]
AluOut[31:0]
PC+4[31:0]
wb_sel [1:0]
inst[11:7]
wb_data[31:0]
wr_data
Reg File
0x51e
waddr
wdata
CSR File

3 Description of Submodules

3.1 Pipeline Stages

The CPU is pipelined into 3 conceptual stages: Instruction Fetch and Decode, Execute, and Writeback.

3.1.1 Instruction Fetch and Decode Stage

Since all memories are synchronous-read, the instruction memories are parallel with the program counter register; in other words, the logic to drive the program counter (represented with the variable `next_pc` in `cpu.v`) also drives the input addresses for the instruction memories. This differs from the quintessential RISC-V CPU, in which the output of the PC register serves as the input for the instruction memories. While this may be more conceptually difficult to reason about, this modification is necessary to balance the delay of each stage of the pipeline – a pipeline stage starting at the program counter register and ending at the instruction memory would have virtually no delay, increasing the size of the remaining two stages and therefore greatly increasing the CPU's critical path.

The instruction fetch and decode (IFID) stage also, as the name implies, decodes instructions fetched from the instruction memory by reading from the registers required for the instruction and generating the immediate required for the instruction (if any). In the version of our CPU with a branch predictor, the IFID also computes the target for `jal` instructions and the predicted target for branch instructions based on the recommendation of the branch predictor (either taken or not taken). This means that, in the CPU with branch prediction, jal instructions and branches with a correct prediction (and no data hazards) are fully resolved in the IFID stage, and despite propagating through the remaining two stages of the CPU, do not perform meaningful work in said stages.

3.1.2 Execute Stage

The Execute stage performs calculations via the ALU and drives the input for most synchronous-read memories (see Writeback Stage). A dedicated adder is used to drive the input addresses of the memories to reduce the fanout of the ALU, and forwarding paths from the Writeback stage help resolve RAW hazards without stalls. The inputs for various memory-mapped-io (MMIO) components are also driven in the Execute stage. Such components include the FPGA's LEDs (which can be turned on or off via a store instruction, the FPGA's buttons and switches (which can be read using load instructions), and instruction and cycle counters to calculate CPI (which again can be read using load instructions).

In the CPU with branch prediction, the ALU and branch comparators' output is compared against the branch prediction from the IFID stage to determine if the branch was correctly predicted or not. In the case of an incorrect prediction, the Execute stage appropriately sets the `pc_sel` control signal to fix the mispredict.

Finally, stalls are inserted in the Execute stage of the CPU. When an instruction needs to be nop'd (for example, due to branch prediction), the relevant control bits are overwritten in this stage since the Execute stage is the first stage in which modifications to state elements (excluding pipeline registers) can be made by the instruction in flight.

3.1.3 Writeback Stage

The outputs of data memory are received in this stage – since the memories are synchronous, their outputs arrive one cycle after the inputs are driven in the execute stage – and along with the output of the ALU, are multiplexed onto the register file's write port. The CSR Register file is also updated in this stage.

In the CPU with branch prediction, for branch instructions, the output of the ALU and the instruction's program counters are paseed into the branch predictor so the relevant bimodal counters can be updated.

3.2 FIFOs

The UART transmitter and receiver interface with the host machine using a set of FIFOs since the host and FPGA may not supply/request data entirely in sync. Our FIFOS are implemented using a circular buffer of parameterizable size. `full` and `empty` signals can be driven based on the status of the FIFO to indicate whether data can be read or written to the FIFO.

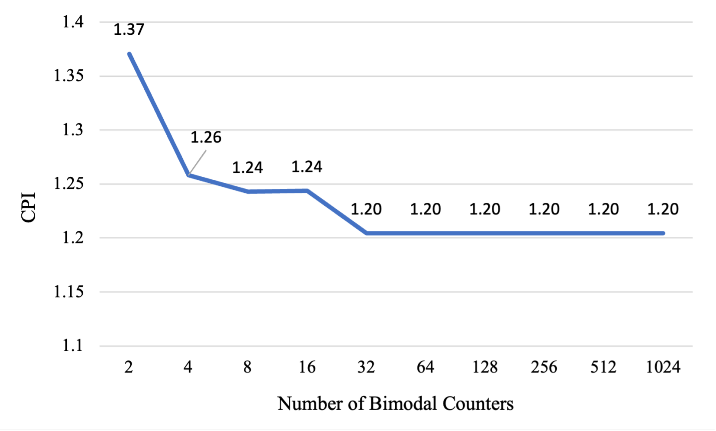
3.3 Audio Synthesizer

The pipelined Frequency Modulation Synthesizer supports 4 different voices, and uses a ready-valid interface to communicate with the rest of the modules. Each voice is synthesized using 2 NCOs, where the second (carrier) NCO is modulated by a shifted result of the first (modulator) NCO. Finally, the outputs of all voices are summed.

In order to operate at `150 MHz`, and provide slack to Vivado so the rest of the optimized CPU meets timing, this module was aggressively pipelined at each step of the process. The output summation uses a pipelined tree adder.

3.4 The Branch Predictor

The branch predictor features a parameterizable number of bimodal counters arranged in a BHT that are indexed into by the instruction's address (the program counter value associated with the instruction). The bimodal counters use asynchronous reads and synchronous writes to provide predictions in the fetch stage and update using the instruction that's committing in the writeback stage. We update the branch predictor in the writeback stage and not the execute stage, when the branch's target is first known, since the instruction may be NOP'd in the Execute stage.



When graphing the CPI of `mmult` against the number of counters used in the branch predictor, we see that the best CPI is reached at 32 counters; although 7 bits are necessary to uniquely identify every instruction in `mmult`, the subset of addresses defined by branch instructions seems to only require 5 address bits for unique identification. It is important to note

4 Verification

We wrote testbenches for the branch predictor, immediate generator and ALU. We also used the provided testbenches with some additional tests to verify our design and debug issues. In particular, we added tests for the following behaviors to `asm_tb.v`:

- Forwarding into a branch, i.e., branch in execute and register write in writeback
- Forwarding into a branch, i.e., jalr in execute and register write in writeback

- jal followed by jalr
- chains of dependent instructions, such as 4 RAWs in a row
- jal and then overwrite link register
- not forwarding writes to x0
- nested branches to check total iteration count with branch prediction control logic

The last test was particularly useful in tracing down conditions where our branch predictor was first picking incorrect branch targets and then replaying certain instructions resulting in fewer than expected iterations. Many of the bugs we caught were caused by using component outputs directly instead of the appropriate pipeline register (for example by passing the output of the immediate generator into the ALU instead of using the pipeline register for the immediate between the IFID and EX stage). This could have been more easily avoided by implementing every stage of the CPU as a separate submodule and connecting pipeline registers to the outputs of each submodule, but would likely have complicated implementation as well.

5 Status and Results

The CPU is fully functional, passing all benchmarks and produces the correct checksum for `mmult`. Precise statistics about the CPU's performance are more nuanced due to the inclusion of the branch predictor; as a result, discussion of said metrics is deferred to later sections.

6 Optimizations

6.1 Move Inst/Cycle Counters and IMEM to Writeback Stage

6.1.1 The Critical Path

The critical path begins at the output port of the Data Memory whose data, via a forwarding path, enters the execute stage and ends at the Instruction Memory input in the Execute stage. Similar paths with a delay smaller than the previously mentioned (but still in violation of the timing requirement) ended at the Instruction/Cycle counters.

6.1.2 Solution and Performance

Since the instruction memory is only written to in the DMEM stage, it can be safely moved to the Writeback stage. Although they are read from, the same can also be done for the instruction and cycle counters; since the counters are implemented using registers, they support asynchronous reads and therefore data does not need to be "requested" from them one cycle in advance.

It is important to note that this modification does not significantly reduce the critical path in terms of propagational delay through logical elements; DMEM and other MMIO devices are still present in the Execute Stage. The rationale behind this optimization is reducing the fanout of the signals driving the IMEM and Inst/Cycle Counters as opposed to reducing the "length" of the critical path.

6.1.3 Alternate Solution: Dedicated Memory Adder

In the same spirit as the optimization described above, a dedicated adder for calculating the memory address for memory and MMIO components reduces fanout of the ALU and therefore decreases delay. Our RISC-V core leverages this optimization and includes the result in the section about Overall Performance after optimizations. However, for the sake of brevity, we did not give this optimization an independent section.

Implementation	Max Clock Frequency	Critical Path	CPI (mmult)
Initial	60 MHz	17.38 ns	1.18
IMEM/Ctrs in WB	60 MHz	16.068 ns	1.18

6.2 Move NOP Insertion to Execute Stage from Fetch

6.2.1 Existing NOP Insertion Logic

In previous iterations of the CPU, instructions were converted to NOPs in the IFID stage via a multiplexer that selects between the instruction and an instruction-encoded no-op. This MUX is placed between the conceptual IF (instruction fetch) and ID (instruction decode) stages, meaning the result of the MUX is passed in as the instruction for the register file and the immediate generator. In this optimization, we discuss why this placement is sub-optimal and can lead to an unexpected critical path.

6.2.2 The Critical Path

Before this optimization, the critical path of the CPU started from the DMEM's `douta` (data out) in the writeback stage and ended at `ex_reg_imm` (the pipeline register for the immediate between the IFID and Execute stages). Data fetched from the DMEM is selected via the memory select mux in the writeback stage and send to the Execute stage via a forwarding path. In the Execute stage, the forwarding path is selected when a data hazard is present, allowing it to be passed into the branch comparator. Since instructions may have to be converted to NOPs in the case of an incorrectly predicted branch, the output of the branch comparator is used to drive the NOP select MUX which, before this optimization, was in the IFID stage. As mentioned before, the result of the NOP select MUX is passed into the Register File and Immediate Generator.

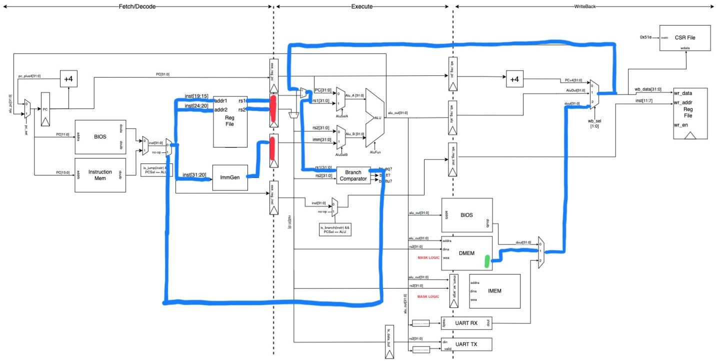


Figure 2: The critical path before the optimization. Green indicates the start of the critical path and red indicates the end. Control logic not depicted.

6.2.3 Solution: Move NOP Insertion to EX

Instead of inserting a NOP in the IFID stage, we delayed the NOP insertion logic by 1 clock cycle and moved the NOP insert mux to the EX stage. By pipelining the input of the NOP select MUX, we break up the critical path by adding a clocked component; the decision to move NOP insertion to the EX stage is then a necessary modification since the select signal is delayed one clock cycle.

6.2.4 Performance Impact

While this optimization alone was not sufficient to increase our clock frequency, it decreased the critical path of the CPU by 1.247ns. `MMULT` CPI remains unchanged at 1.18cpi before and after this optimization since, although this modification changes the location at which we insert stalls in our CPU, it does not change the number of stalls inserted for each control hazard.

Implementation	Max Clock Frequency	Critical Path	CPI (mmult)
NOP in IFID	60 MHz	16.068 ns	1.18
NOP in EX	60 MHz	14.821 ns	1.18

6.2.5 Alternate Solution: Implicit NOP Insertion

This critical path could also have been resolved by implicitly inserting NOPs directly at pipeline registers instead of replacing the entire instruction. For example, the pipeline registers for `rs1` and `rs2` at the boundary of IFID and Execute could have been set to 0 and the pipeline register storing the destination register address could also have been cleared. This has the same effect as inserting the NOP as an instruction but avoids the critical path since the insertion occurs after the register file and immediate generator and directly at a clocked component. Our RISC-V core leverages this optimization to reduce the critical path when the branch predictor is present and includes the result in the section about Overall Performance after optimizations including the branch predictor. However, for the sake of brevity, we did not give this optimization an independent section.

6.3 Pipeline Path from ALU to Program Counter

6.3.1 The Critical Path

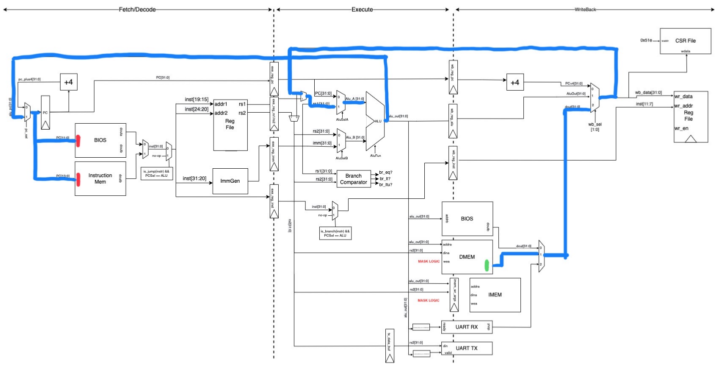


Figure 3: The critical path before the optimization. Green indicates the start of the critical path and red indicates the end. Control logic not depicted.

Similar to the previous optimization, the critical path begins at DMEM in the writeback stage, uses the data forwarding path to enter the execute stage. This time, however, the data from the forwarding path enters the ALU. The target propagates through the PC select MUX and reaches the input ports of the BIOS and Instruction memories, which are clocked components.

6.3.2 Solution: Pipeline ALU -> PC wire

We resolve this critical path by pipelining the path from the ALU to the program counter select MUX. Since the target PC from the ALU is now delayed by 1 clock cycle, we also pipeline the PC select signal so it remains synchronized with the target PC.

6.3.3 Performance Impact

This solution increases the number of stalls inserted for taken branches and jump instructions. Previously, one instruction had to be converted to a NOP since an erroneous instruction was fetched in the IFID stage while the branch/jump instruction was being computed in the Execute stage. With the pipelined design, although the target for taken branch and jump instructions is still computed in the Execute stage, an additional cycle is required for the computed target to reach the program counter mux. This means that two instructions must be converted to NOPs in the case of control hazards. After this change, `mmult` runs at a CPI of 1.36 (from an increase of 1.18). However, the regression in CPI enables a higher clock frequency of 75 MHz. benchmark.

Implementation	Max Clock Frequency	Critical Path	CPI (mmult)
Unpipelined ALU ->PC	60 MHz	14.821 ns	1.18
Pipelined ALU ->PC	75 MHz	12.916 ns	1.36

6.4 Overall Performance Impact

Without optimizations, our CPU was capable at running at a clock frequency of 60 MHz. With optimizations, our CPU stalls an additional cycle at control hazards but can run at an increased clock frequency of 75 MHz.

To decide which design is better, we use time per program as a metric for representing the overall performance of the CPU:

$$\frac{\text{Time}}{\text{program}} = \frac{\text{Cycles}}{\text{Instructions}} \times \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Time}}{\text{Cycle}}$$

Our benchmark program of choice is `mmult`, which multiplies two `64x64` matrices together.

$$\frac{\frac{\text{Time}}{\text{program}}_{\text{original}}}{\frac{\text{Time}}{\text{program}}_{\text{optimized}}} = \frac{\frac{\text{Cycles}}{\text{Instructions}} \times \cancel{\frac{\text{Instructions}}{\text{Program}}} \times \frac{\text{Time}}{\text{Cycle}}}{\frac{\text{Cycles}}{\text{Instructions}} \times \cancel{\frac{\text{Instructions}}{\text{Program}}} \times \frac{\text{Time}}{\text{Cycle}}} = \frac{1.18/60}{1.36/75} \approx 1.085$$

This tells us that, despite the increase in CPI, `mmult` runs faster on the optimized CPU than the initial design. It is important to note that `mmult` is comparatively more data-hazard heavy than control-hazard heavy, so additional stalls on control hazards are preferable to stalling on data hazards.

6.4.1 CPU With Branch Predictor

The addition of the Branch Predictor increased the critical path of the CPU and therefore decreased the maximum clock frequency at which it could operate. The addition of a branch predictor with 32 bimodal counters alone reduced the CPI of the CPU from `1.36` to `1.2`. However, the lengthened critical path generated from the branch predictor afforded additional timing slack with which we could introduce new optimizations – most notably resolving jal instructions in the fetch stage – further reducing the CPI of `mmult` to `1.16`.

With branch prediction, the maximum clock frequency the CPU runs at is `70 MHz`. When comparing the CPU with a branch predictor against the optimized CPU without a branch predictor:

$$\frac{1.36/75}{1.16/70} \approx 1.094$$

Indicating that the CPU with branch prediction is faster than the CPU without branch prediction. When comparing the CPU with branch prediction against our original design:

$$\frac{1.18/60}{1.16/70} = 1.186$$

We can see that our original design was nearly 20% slower than the design with branch prediction.

6.5 Area Utilization

Our design uses 2445 LUTs, 36 BRAM cells, 36 flip-flops, and 0 DSPs. We anticipate this value is higher than expected due to the presence of the bimodal counter for the branch predictor. However, given that the CPU, even with a branch predictor, does not come close to using all of the FPGA's resources, we consider this to be an acceptable tradeoff.

7 Conclusion

Ultimately we're happy that our CPU and branch predictor work and are grateful for all the lessons we learned in the process. Some of our lessons and suggestions for the future are listed below:

7.1 Test Often

Most of our bugs were found in modules that could have been individually tested but were not; trying to debug the entire CPU at once was not a particularly viable strategy. After we wrote separate testbenches for individual components, such as the immediate generator and ALU, we found many minor errors that would prevent certain instructions or edge cases from functioning as expected.

7.2 Don't be afraid of Waveforms

Debugging was much easier and faster after we immediately resorted to opening waveforms instead of first trying to re-read our code to look for errors. With a project as large as the CPU, the bug is likely too small to catch during a cursory scroll-through

7.3 Be Together, Not the Same

Working together while coding and debugging made the process both faster and more enjoyable. We found that we could accomplish the same amount if we both worked together on the same thing instead of working on individual tasks and then trying to combine them, since we saved time both on explaining our work to each other and on trying to deal with merge conflicts/overwritten history. At the same time, we found that while it was beneficial to work together, this didn't mean one partner had to code/debug while the other watched. We found it helpful to, for example, have one of us work on the implementation while the other wrote the testbench, so we could instantly check our work while still clarifying our assumption and design details with each other while necessary.

7.4 Avenues of Further Exploration

Despite our CPU's satisfactory level of performance, there always remains room for further optimization. We provide suggestions for this below:

- Optimizing signal use, particularly for comparisons, can contribute significantly to performance.
- Exploring different functions that combine information to index into the branch predictor would be interesting. We modeled our branch predictor in Python and would have like to explore this design space with a trace of `mmult` branch and jump instructions.
- Selective removal of data forwarding paths as certain forwarding paths are used infrequently and the reduction in CPI is easily compensated by an increase in clock frequency. An example of this would be load use instructions.
- Repurposing branch prediction and fixup logic to resolve jalr instructions in the fetch stage.