EECS 151 FPGA Project Report

Nikhil Jain, Jianzhi Wang (Team: RISC Takers)

May 8, 2023

Design Objectives

Our primary design objective is to build a multi-stage RISC-V processor, which is to be run on a FPGA. Our goal is to maximize the figure of merit FOM $= \frac{f}{\text{CPI} \cdot \text{cost}}$, which evaluates our design based on performance (dependent on both frequency (f) and cycles per instruction (CPI)) and cost. To do so, we initially implemented the minimal 3 stage processor as detailed by the project specifications, which achieved a low CPI at the expense of a capped frequency. To continue increasing the frequency, we decided to add extra pipeline stages, eventually settling at a traditional 5 stage processor design that has a higher CPI, but a significantly improved frequency.

The memory hierarchy we used is consistent with the project specifications. The IMEM, DMEM, and BIOS modules allow for synchronous read and write, and the RegFile module allows synchronous write with asynchronous read.

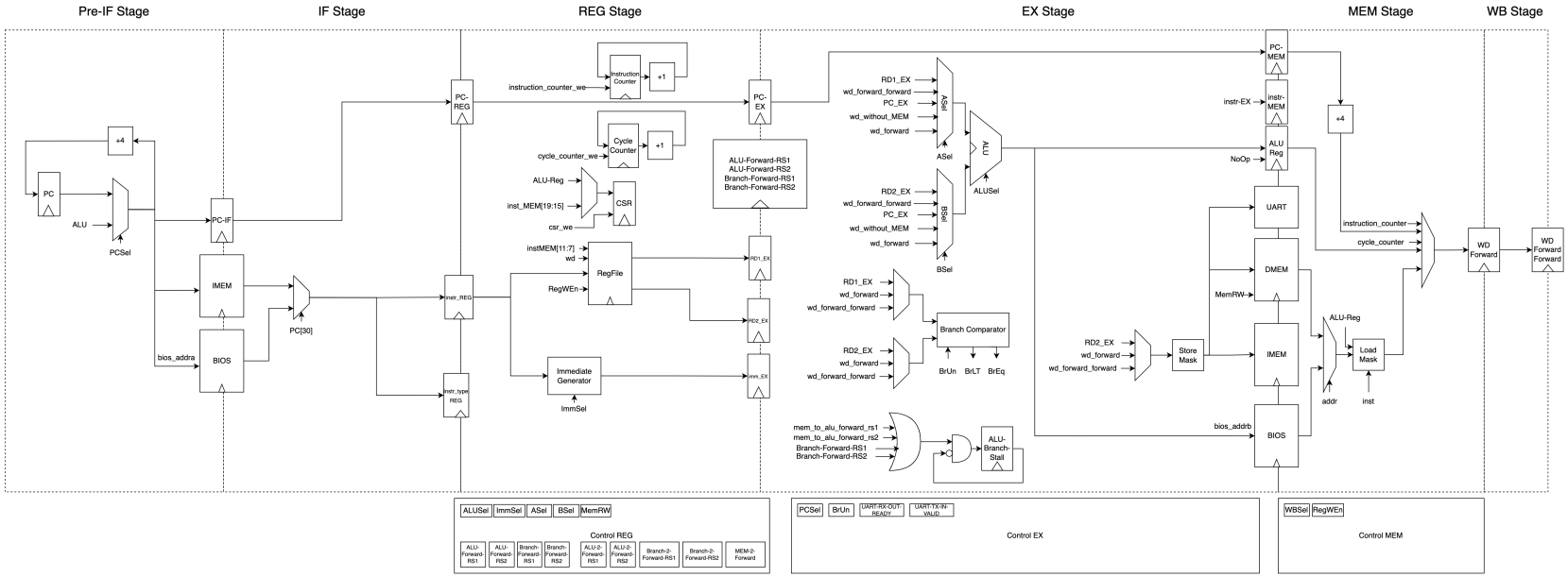
High-Level Organization

Our final product is a 5 stage pipelined processor, with the stages being (1) Pre-IF: Pre-instruction Fetch Stage, (2) IF: Instruction Fetch Stage, (3) REG: Register File Stage, (4) EX: Execute Stage, and (5) MEM: Memory Stage. We also have two delayed write back register stages that are used during forwarding and stalls to minimize CPI and the critical path delay.

Note: The initial 3 stage pipeline from the project specifications does not include the Pre-IF stage since its path delay is minimal compared to the other stages, but we choose to include this for our 5 stage pipeline definition since a clock cycle is required for an instruction to move from the input of the PC register to the input of the IMEM and BIOS memory blocks.

We have controller blocks for the REG, EX, and MEM stages. The REG stage is where most control values are initially calculated (to be propogated down the pipeline) and therefore has the most complex controller. This design choice is because the control logic can occur in parallel with the RegFile operations and thus will not contribute to the critical path, unlike the EX and MEM stages which use control signals on their respective critical paths. We chose to keep most of the design code in the `cpu.v` file to allow for design adjustments to be made relatively easily, though some important submodules were created, which include those for the ALU, controllers, memory input and output masks, immediate generator, branch comparator, and advanced branch predictor (which we ultimately decided not to use). Our 5 stage pipeline design block diagram is on the next page.

Five-Stage Pipeline Processor Schematic



Detailed Description

Pre-IF Stage

The purpose of the Pre-IF stage (which is not considered a stage for the 3 stage pipeline, but we consider to be a stage for the 5 stage pipeline) is to set up the correct `PC` that will be used to fetch instructions from IMEM or BIOS. The sole register for `PC` in this stage keeps track of the current `PC` value, which is the input for IMEM and BIOS. For instructions that do not change `PC`, `PC` simply increments by 4 on the next cycle. For instructions that do change `PC`, the input to IMEM and BIOS is instead set to `ALUOut` (which contains the updated `PC` value) and the input to the `PC` register is set to `ALUOut+4` since this represents the next instruction in this case. The standard branch prediction, which assumes every branch is not taken, is used, with a pipeline flush procedure present in the case of an incorrect prediction.

IF Stage

The purpose of the IF stage is to fetch the instruction using the `PC` value input from the previous stage. This stage does not contain much logic because most operations require the instruction to be fetched first. If we added logic that depended on the instruction to the IF stage, this would lengthen the delay path from the IMEM block, which is undesirable.

REG Stage

The purposes of the REG stage (which is the primary addition in the move from the 3 stage pipeline to the 5 stage pipeline) are to (1) fetch register values from RegFile and (2) generate the immediate. We also precompute most control values here to decrease the complexity of the controller in EX stage. The CSR, cycle, and instruction registers should be seen as an extension of the RegFile. The REG stage is the location of 2-cycle forwarding logic, which checks if the MEM stage instruction will overwrite the value fetched by the REG stage instruction (see the Forwarding section for more information). At one point, we moved the other forwarding and stall logic to the REG stage as well to reduce the critical path for the processor, though we decided to revert this change due to the unexpected rise in cost.

EX Stage

The purposes of the EX stage are to (1) find the correct ALU inputs and perform the ALU calculations, (2) find the correct branch comparison inputs and perform the branch comparison using the branch comparator module, and (3) prepare the inputs to the memory blocks, which includes a memory mask module to adjust the inputs. To find the correct inputs to the ALU, branch comparator, and memory, the EX stage calculates 1 cycle forwarding logic (which checks whether the MEM stage instruction will overwrite the register value being used by the EX stage instruction) and uses the 2 cycle forwarding logic pipelined from the REG stage to determine which value should be used by the instruction in the EX stage (see the Forwarding section for more information). Due to the presence of a long delay path through memory and the ALU / branch comparator due to forwarding, for instructions that use this specific critical path, we implemented a stall operation that halts the entire processor except for some write back registers to break up the critical path (see the Stalls section for more information).

MEM Stage

The purposes of the MEM stage are to (1) fetch any data from the memory blocks and (2) determine the value to write back to the RegFile (or the other extended registers). The write back data, `wd`, can come from the instruction counter, cycle counter, `PC+4`, ALU, or memory (either BIOS or DMEM).

(Delayed) Write Back Stages

The write back data values for past instructions (or the current instruction in the case of a stall) are stored in two back-to-back registers after the MEM stage to allow for 2 cycle forwarding (since the value to be forwarded would be overwritten by the instruction entering the MEM stage on the cycle when the instruction that needs it enters the EX stage), as well as for a stall if called for by the EX stage instruction (since the output of a register will avoid the time delay of the write data being fetched from the memory blocks, reducing the critical path delay).

Forwarding

For both REG and EX stages, there is logic required to determine whether forwarding is required. To define the forwarding logic, it is convenient to partition

the opcodes into the following sets:

- Type α = {R TYPE, I TYPE, J TYPE, U TYPE}
- Type β = {R TYPE, I TYPE, S TYPE, B TYPE, CSR TYPE}

Intuitively, type α instructions are those which may cause data hazards, while type β instructions are those which may suffer from data hazards. Below, RSX refers to either RS1 or RS2. The forwarding conditions for RS1 and RS2 are almost the same, with slight adjustments depending on which instructions actually affect their values. Additionally, if an instruction attempts to write to register `x0`, the value will not be forwarded since the value will not be written to the register. Our forwarding logic can then be stated as follows:

- Forward: inst_{MEM} is of type α, inst_{EX} is of type β, and $\text{RSX}_{EX} = \text{RSD}_{MEM}$.
- MEM to ALU Forward or Branch Comparator: inst_{MEM} is a load instruction, inst_{EX} is a type β, and $\text{RSX}_{EX} = \text{RSD}_{MEM}$.
- MEM to MEM Forward: inst_{MEM} is of type α, inst_{EX} is a store instruction, and $\text{RS2}_{EX} = \text{RSD}_{MEM}$.

In the case of MEM to ALU or Branch Comparator forwarding, a stall will occur by design to reduce the critical path time.

There is also 2-cycle forwarding logic present for 2-cycle hazards, which make use of the write back register after the MEM stage that stores the `wd` value of the instruction that just completed.

- Forward 2: inst_{MEM} is of type α, inst_{REG} is of type β, and $\text{RSX}_{REG} = \text{RSD}_{MEM}$.
- MEM to MEM 2 Forward: inst_{MEM} is of type α, inst_{REG} is a store instruction, and $\text{RSX}_{REG} = \text{RSD}_{MEM}$.

Stalls

A stall is used when a value returned by a memory block in the MEM stage is required for the EX stage's ALU or branch comparator inputs since without a stall, there would be a path through a memory block and ALU or the branch

comparator, which effectively removes the benefit of pipelining between the EX and MEM stages. This is handled by a `stall` signal that is true when either RS1 or RS2 requires MEM to ALU or Branch Comparator forwarding, and the processor has not already stalled:

$$\text{stall}_{t+1} = (\text{MEM to ALU or Branch Comparator}) \wedge \neg\text{stall}_t$$

In this case, the entire processor halts operation while the `wd` value is stored in the write back register present after the MEM stage (and the previous instruction's write data is stored in the second write back register after that for any 2 cycle hazards that persist). Thus, on the following cycle (when the processor is no longer stalled), the forwarded value is simply the output of a register without any delay time.

Other Optimizations

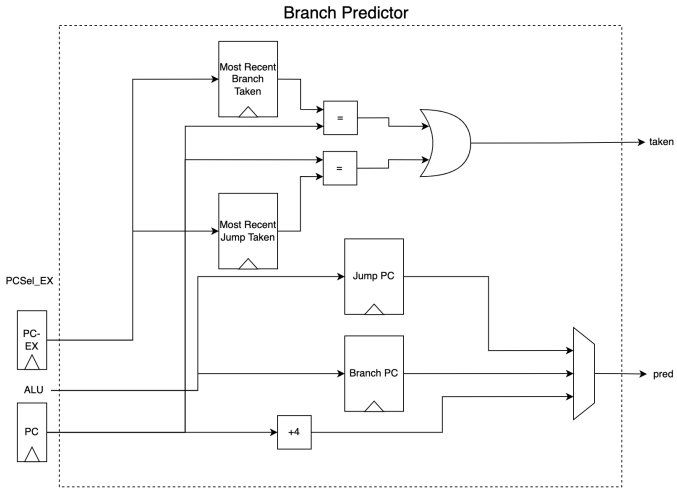
Beyond the decision to go with a 5 stage pipeline to optimize frequency and standard optimizations such as forwarding and cost optimization by removing repetitive logic, we also tried other optimization ideas, many of which exist within the final design.

Controls

Our critical path often exists within the execute (EX) stage, and at one point went through the EX stage controller. Hence, we sought to minimize the size of this control block. To do so, we decided to precompute whatever control signals we can in the REG stage, and pass those through pipeline registers to the EX stage.

Branch Predictor

We tried using a branch predictor (see the diagram below) that works by storing the previous branch instruction that was taken, as well as the corresponding `PC` value of the destination, so that if the branch appears again, we will predict it to be taken. We also store the most recent jump with its destination, which will always be taken if seen again. The result was a decrease in CPI for `mmult`, but an increase in CPI for `bdd` significantly due to its constantly varying branch decisions. Furthermore, it increased the cost significantly, so we decided that simply assuming every branch is not taken was the best choice.



Bitwise Optimization

We used Karnaugh map optimization on the opcodes, which allowed us to simplify the checking of instruction types to a single bit comparison.

NO-OP	000	U TYPE	100
B TYPE	001	J TYPE	101
CSR TYPE	010	I TYPE	110
S TYPE	011	R TYPE	111

Hence, α type instructions are those with MSB $= 1$ and β type instructions are those with second most significant bit $= 1$.

We tried to perform similar optimization on the load-mask and store-mask modules. However, they increased the cost significantly instead. We suspect this is due to the compiler's modular optimization, which likely prefers behavioural codes.

Non-integer Clock Periods

We also implemented non-integer clock periods with the following table of values.

CLKOUT0_DIVIDE	DIVCLK_DIVIDE	CLKFBOUT_MULT_F	Frequency (MHz)
10	5	63	157.5
10	5	64	160.0
10	5	78	195.0

Status and Results

Our final processor works. The maximum frequency we achieved among all versions of the processor we built was 195MHz, though this did not work consistently on all FPGAs. Hence, we adjusted our final design to balance cost and performance, running at 166.7MHz. To ensure stable performance, we further reduced our frequency for checkoff. During checkoff, we had a frequency of 157.5MHz, CPI of 1.34, and cost of 1780990, for an FOM of 66.10. Here are the FOM values we were able to achieve with different versions of our design:

Processor Versions				
Processor	Frequency	CPI	Cost	FOM
3-Stage Pipeline	135.0	1.15	1882032	62.11
5-Stage Pipeline 1.0	165.0	1.34	1759325	69.99
5-Stage Pipeline 1.1	195.0	1.34	2069699	70.31
5-Stage Pipeline 2.0 (final)	166.7	1.34	1766751	70.51
5-Stage Pipeline 2.1 (safe)	160.0	1.34	1803692	66.31
5-Stage Pipeline Checkoff	157.5	1.34	1780990	66.10

Critical Path

Our final critical path exists within the EX stage and goes from the pipelined instruction register, through the forwarding logic, to the multiplexer at the input of the ALU, through the ALU, and finally to the memory block inputs. We attempted to reduce this critical path further, but the cost increase was too much in comparison to the frequency improvement so we decided to leave this critical path.

Scripting

We tried three flags: `AreaOptimized`, `AreaOptimized_high` and `PerformanceOptimized`. The `PerformanceOptimized` directive worked the best, as shown by the following data:

Processor Version = 5-Stage Pipeline 2.1				
Directive	Frequency	CPI	Cost	FOM
AreaOptimized_medium	160.0	1.34	1873952	63.82
AreaOptimized_high	160.0	1.34	1883815	63.48
PerformanceOptimized	160.0	1.34	1803692	66.31

Improvements on the 3 Stage Processor

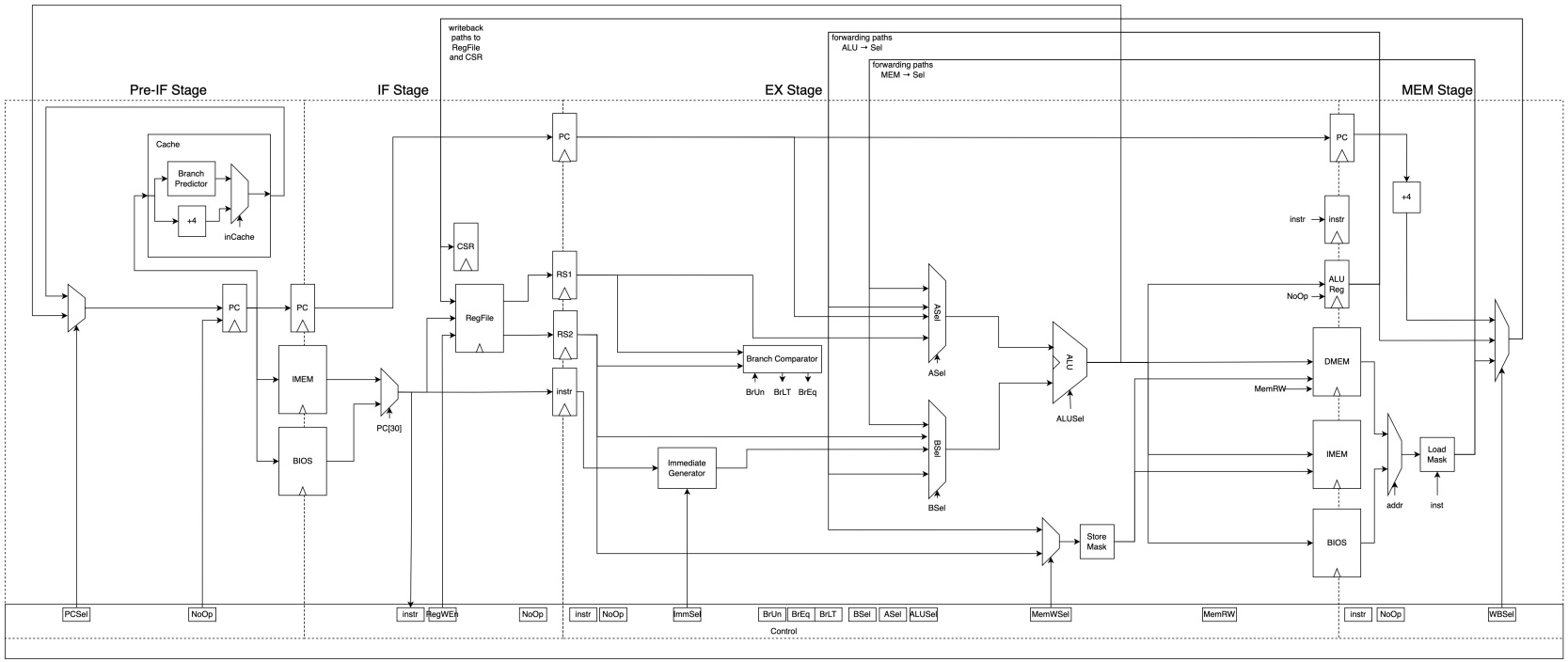
Originally, we designed a 3 stage processor according to the specifications. Here, we compare the two processors.

The main advantages of our 5 stage processor over its 3 stage predecessor are:

- Splitting of IMEM and RegFile: Although the asynchronous read functionality of the RegFile allows it to be placed in the same stage as IMEM, splitting it up reduces a long delay path through both modules.
- Separate Controller Block: The splitting up of the controller blocks to only provide relevant signals for each stage reduces the cost and prevents a long delay path through the controller.
- Precomputation in the REG stage: The REG stage can be used for precomputations in parallel to the RegFile's read operation (such as immediate generation) to be done and passed onto the EX stage since if this logic was present in the EX stage where it is used for the ALU inputs, it would create a long delay path. Also, all control logic signals that can be computed in the REG stage and pipelined forward are done so for the same reason.

Our initial 3 stage pipeline design block diagram is on the next page for reference.

Three-Stage Pipeline Processor Schematic



Conclusions

Overall, we learned that design optimization requires trial and error. Many times, an idea for improvement may not help much. There are too many combinations of ideas to try and since many of them are complex, it is important to understand which ideas are likely to help in order to make consistent progress. We also learned that micro-optimizations for cost are difficult. The compiler abstracts away many internal optimizations and it is difficult to visualize how the modules are translated and placed on the FPGA. Some of our optimizations ended up increasing the cost even though theoretically they should be strictly better (for example, converting all registers from `REGISTER_R_CE` to `REGISTER_R`). Next time, we would look to experiment more with the tcl files to find if there are more options to improve the compiler performance for our specific design, as well as continue increasing the frequency of the 5 stage processor, which we think has a lot of potential for improvement (despite the increased CPI and cost) due to the improved frequency.

Future Extensions

Here are some ideas that we may want to try in the future:

- We could implement a cache for the memory blocks to reduce the critical path delay even more, which if implemented well, should not increase the CPI by much.
- We could adjust our processor to be a superscalar processor that can compute multiple instructions at once, which could ideally decrease the CPI by a significant amount, though it would also increase the cost by a lot.
- We could try to implement an improved branch prediction module that is more accurate and would reduce the CPI sufficiently.