COMP_ENG 362: Computer Architecture Project

By:
Daniel Shaver
Sebastian Phemister
3/13/24

Instructor:
Panitan Wongse-Ammat

Table of Contents

List of Figures

List of Tables

Executive Summary

Design Problem

The design problem at hand involves the development of a high-performance RISC-V CPU architecture adhering to specific constraints and requirements. Central to this endeavor is the implementation of a 5-stage RISC-V pipeline utilizing Verilog or SystemVerilog, with options for structural or behavioral modeling approaches. The pipeline architecture encompasses key stages—Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back—each performing essential functions in the instruction execution process.

A primary focus of the design problem is the effective resolution of hazards within the pipeline through the implementation of forwarding and stall mechanisms. Forwarding paths must be fully provided to ensure seamless data flow and prevent pipeline stalls, enhancing overall performance and efficiency. Furthermore, the CPU design must integrate support for RISC-V integer multiply instructions to expand its computational capabilities and versatility.

Synthesis of the pipeline processor using Cadence's Genus tool is a key requirement, facilitating efficient hardware synthesis and optimization processes.

The design problem also entails meeting specific performance metrics, including a minimum operating frequency of 800MHz and constraints on chip area relative to frequency. Additionally, the CPU implementation must pass provided test cases, validating its functionality and performance in executing complex computational tasks.

Finally, the CPU must also contain a surprise factor that provides an increase of performance on one of the following metrics: cycles taken, cycle time, power consumption, or number of gates

By addressing these design challenges and requirements, the aim is to develop a robust and efficient RISC-V CPU architecture capable of meeting the demands of modern computing applications while ensuring compatibility with industry-standard design practices and synthesis tools.

Design Approach

The main approach that was taken to build the processor was to use Test Driven Development (TDD) in order to add full forwarding and the surprise factor, which was unanimously decided to be a branch predictor. For a deeper understanding of TDD, refer to the Engineering Standards section.

During the development of the initial 5-stage pipelined CPU, a primary focus was placed on maximizing the simplicity of debugging by keeping all components loosely coupled. This approach led to the creation of separate modules within the top-level architecture of our pipelined CPU. The design incorporated one module dedicated to each of the five pipeline stages, four modules dedicated to facilitating seamless data transfer between the stages, and a module each for detecting and handling stalls as well as managing forwarding mechanisms. This modular design strategy not only enhances the clarity of the overall architecture but also streamlines the debugging process, enabling efficient identification and resolution of any potential issues.

While there are multiple methods of implementing full forwarding, the method that was chosen was to forward data from the end of EX and MEM stages into a multiplexor (MUX) at the end of the ID stage, as shown in Figure 1. This MUX chooses either the register file's data, the data outputted from the ALU, or the data retrieved from memory. The signal that controls the MUX depends on the potential data hazards and is determined by the Forwarding Unit. This logic is present for both registers.

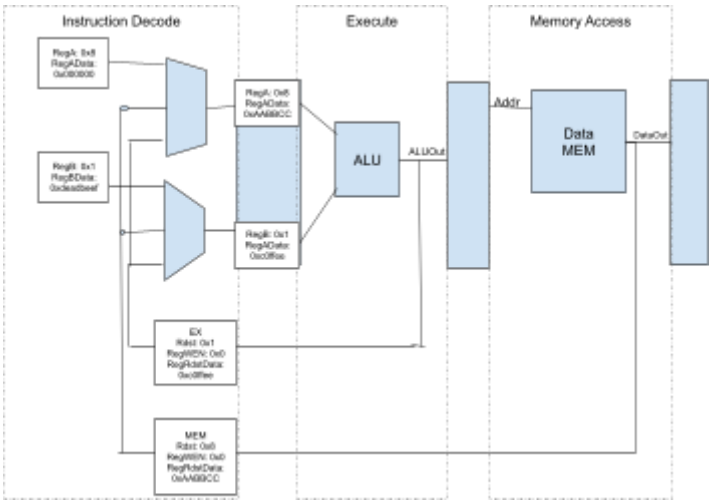


Figure 1: An Image Describing the Forwarding Path Used in the CPU

There are many types of branch predictors; however, we chose to implement a 2-bit branch predictor because it is relatively simple to implement for the increase in speed it provides. The finite state machine (FSM) for a 2-bit predictor only involves four states, and the logic can be seen in Figure 2 below.

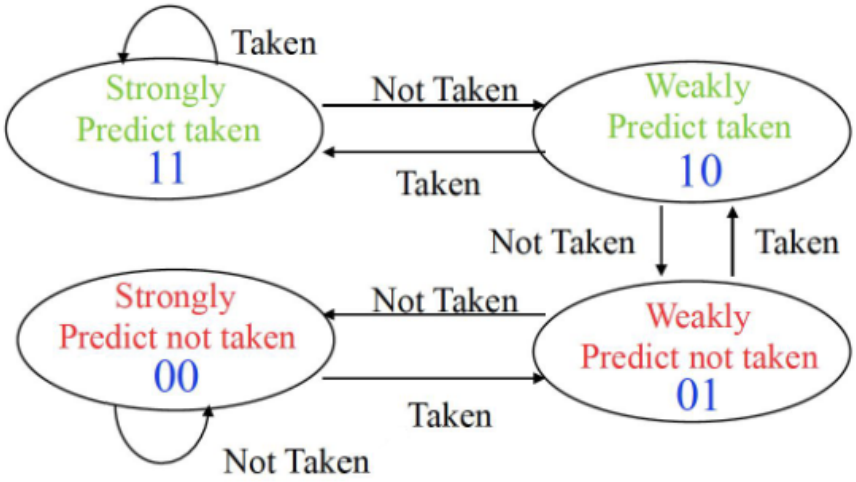


Figure 2: Finite State Machine of 2-Bit Branch Prediction

While our design achieves the specified requirements for area and power, there are areas where further improvement could enhance its performance.

The benefits of our design include:

- Fulfillment of all specified requirements, including support for required instructions and synthesizability with Cadence's Genus.
- Incorporation of full forwarding, enhancing data flow efficiency.
- Implementation of a 2-bit branch predictor, contributing to increased processor speed.

However, it's essential to acknowledge certain limitations:

- Possibility of optimization in certain design aspects (ALU, branch predictor, etc) for further performance enhancement.
- Lack of I/O ports, sensors, or physical design hinders practical applicability for various computing environments.

Potential improvements are discussed further in the Broader Considerations section.

Body

Introduction

Considering the initial constraints, our first task consisted of constructing a five-stage RISC-V CPU using Verilog. Both members of our team had prior experience from COMP_ENG 361 in the fall, where one of the labs involved transitioning a single-cycle CPU into a five-stage pipelined CPU. Consequently, this task proved comparatively straightforward, merely requiring a straightforward git pull from the previous quarter's code repository. Given that Daniel and Sebastian had worked in separate groups during the course, Daniel's code was selected for its clarity and ease of modification.

From this point onward, Daniel's Verilog code, referred to interchangeably as the processor or CPU, shares similarities with the following two designs:

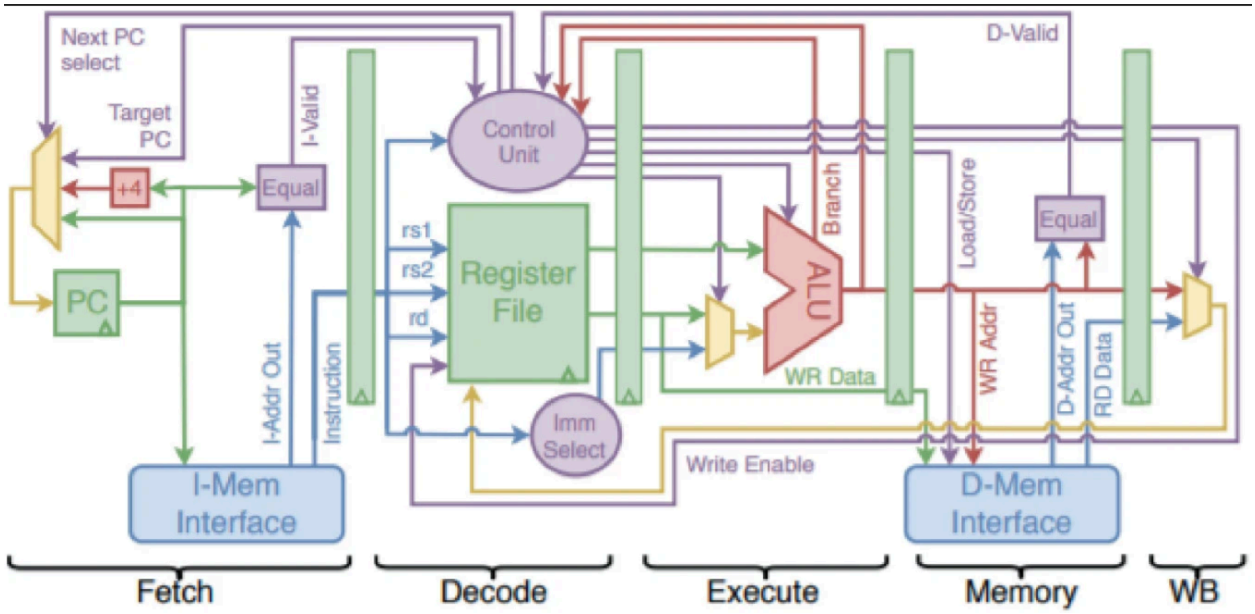


Figure 3: A Figure showing the layout of an non-pipelined five stage Pipelined CPU (Mallidu and Siddamal)

Similar to the design depicted in Figure 3, our CPU aims to minimize cycle time while optimizing each stage of the processor for maximum efficiency. Although it's feasible to move branch calculations within the ID (Instruction Decode) stage, doing so would necessitate fetching data from the register file, comparing values, and determining branch outcomes, all in the same cycle. This duplication of effort in ID basically mirrors the function of the ALU in the EX (Execute) stage, potentially exacerbating our critical path. After careful consideration of the tradeoffs, we opted for an additional stall cycle for mispredicted branches over risking a global

slowdown. The IF (Instruction Fetch), ID, and MEM (Memory) stages are primarily constrained by data access times, so it is unwise to introduce additional computations beyond essential functions like MUXes for full forwarding and basic adders for target address calculations. In contrast, the EX and WB stages require minimal adjustment or optimization, resulting in functionality akin to that depicted in Figure 3.

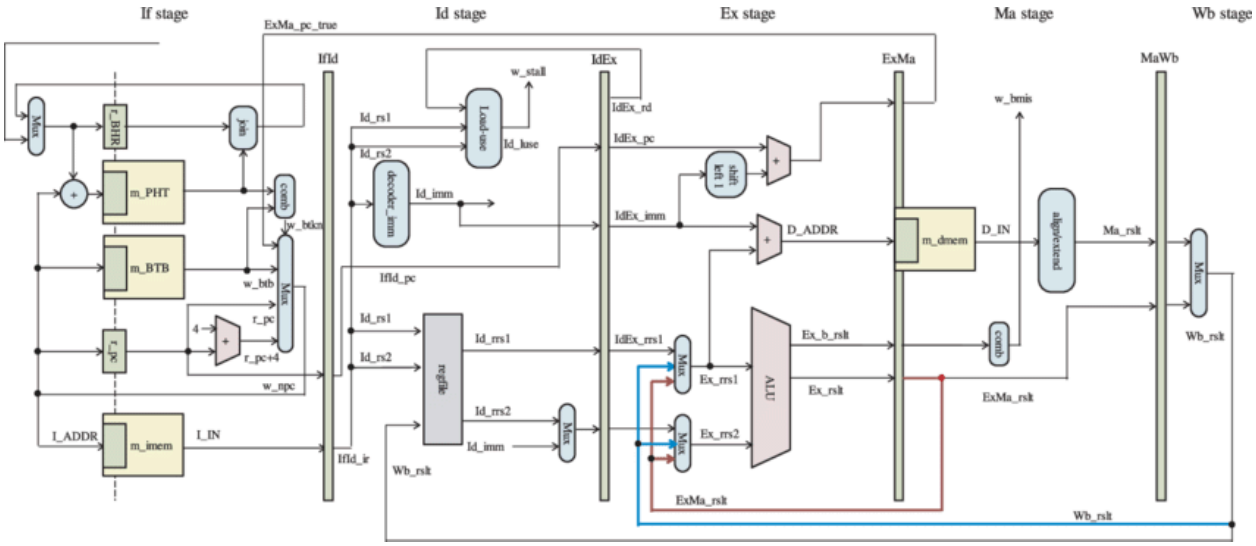


Figure 4: A figure showing the layout of a pipelined five stage CPU ("A Block Diagram of Typical Five-Stage Pipelined Processor...")

As mentioned in the executive summary, the processor is defined by its 5 stages (IF, ID, EX, MEM, and WB), with four pipeline registers between to facilitate the flow of data and control signals throughout the stages. However, one difference is that the processor depicted in Figure 3 includes a branch predictor that contains a prediction cache for every branch instruction inside of the CPU. While this would make branching incredibly fast, it would have slowed down our critical path and in turn made our clock frequency, currently at 800MHz, fail to meet the minimum requirements. This was one of the main reasons we opted for a 2-bit branch predictor instead, which provided a notable increase in performance over the previous method of handling branches, which was to predict always not taken.

The final key difference between our design and Figure 3 was the placement of the full forwarding module, which was chosen to be at the edge of ID in order to simplify the debugging process for data flowing beyond the EX stages.

Design Constraints and Requirements

In developing the RISC-V CPU, a set of stringent constraints and requirements were established to guide the design process. These constraints ensure the adherence to specific design principles while meeting essential performance metrics and compatibility criteria. Below is a comprehensive overview of the design constraints and requirements imposed on the RISC-V CPU:

1. **Implementation Language and Models:**
    - The RISC-V CPU must be implemented using either Verilog or SystemVerilog.
    - The design can be realized through either structural or behavioral modeling paradigms, providing flexibility in the design approach while maintaining compatibility with the target hardware description languages.
2. **Pipeline Architecture:**
    - The CPU architecture must adhere to a 5-stage RISC-V pipeline model, comprising Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB) stages. This architectural framework ensures efficient instruction execution and data processing. The behaviors of each stage are defined below, in Table 1.

Pipeline Stage	Short Name	Function
Instruction Fetch	IF	Fetch the instruction that is being pointed to by PC
Instruction Decode	ID	Decode the instruction, get register data from the GPR file, generate control signals
Execute	EX	Choose data sources to enter into the ALU, an ALU with support for all Register-Type instructions
Memory Access	MEM	For load and store instructions, store/load the related information into memory
Write Back	WB	Write back the data into the Reg-File

Table 1: A Table Describing All of the Stages of the Pipeline According to the Design Problem

3. **Forwarding Paths:**
    - The CPU design should fully provide all forwarding paths to ensure seamless data flow and prevent pipeline stalls. By enabling data forwarding between pipeline stages, the processor minimizes latency and maximizes throughput, enhancing overall performance.
4. **Hazard Resolution:**
    - The design must employ forwarding and stalls to resolve all hazards effectively. Forwarding mechanisms enable the forwarding of data directly from the execution stage to the instruction decode stage, mitigating potential data hazards and enhancing pipeline efficiency.
5. **Instruction Set Support:**
    - The RISC-V CPU must incorporate support for RISC-V integer multiply instructions, expanding the instruction set architecture to accommodate diverse computational requirements and facilitate efficient algorithm execution. Table 2 details all of the required instructions that must be supported.

Instruction Type	Required Instructions
Register-Type	add, sub, sll, slt, sltu, xor, srl, sra, or, and
Register-Type (Multiply)	mul
Immediate-Type	addi, slti, xori, ori, andi, slli, srli, srai
Immediate-Type (Load)	lb, lh, lw, lbu, lhu
Store-Type	sb, sh, sw
Branch-Type	beq, bne, blt, bge, bltu, bgeu
Jump-Type	jal, jalr
U-Type	lui, auipc

Table 2: A Table Displaying All of the Instructions that need to be supported

6. Synthesis:

- Cadence's Genus tool must be utilized for synthesizing the processor. This ensures compatibility with industry-standard synthesis tools and facilitates efficient hardware synthesis and optimization processes.

7. Performance Metrics:

- The CPU implementation must meet specific performance metrics to ensure optimal functionality and compatibility with target applications:

$$fmin = 800MHz$$

$$Areamax|(f = 800MHz) = 30,000\ \mu m^2$$

$$Areamax|(f > 800MHz) = 16193.25e^{0.000778f}\ \mu m^2$$

  - Where $fmin$ is the minimum operating frequency, and $Areamax$ is the maximum area @ $fmin$.
  - Alternatively, if the processor is run at a higher frequency, the third equation is provided to calculate the maximum area at that frequency.
- Imposing a constraint on chip area optimizes resource utilization and minimizes manufacturing costs. Meeting these performance metrics is crucial for ensuring the competitiveness and viability of the CPU architecture in diverse computing environments.

8. Test Case Requirements:

- The CPU implementation must pass two provided test cases, consisting of microbenchmarks designed to evaluate the processor's performance in calculating a Fibonacci number. These test cases serve as benchmarks for assessing the CPU's functionality, efficiency, and accuracy in executing complex computational tasks.

By adhering to these design constraints and requirements, the development of the RISC-V CPU aims to achieve a robust, efficient, and high-performance processor architecture that meets the demands of modern computing applications while ensuring compatibility with industry-standard design practices and synthesis tools.

Engineering Standards

Test Driven Development

From the advice of Prof. Panitan, we followed the Test Driven Development guidelines for this project. This approach simplified and accelerated bug detection, enabling us to quickly identify the source of errors from a concise list of potential causes.

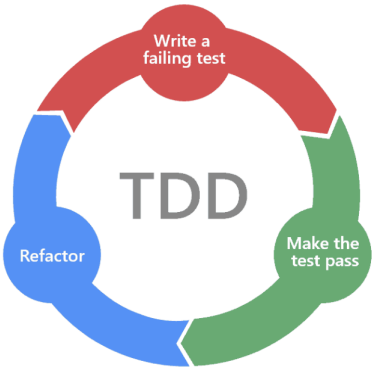


Figure 5: A Figure Explaining Test Driven Development (Hanes)

In TDD, tests are written before the actual code. The methodology involves writing a test for each piece of desired functionality, running the test (which initially fails), implementing the code to pass the test, rerunning the test (which should now pass), and finally refactoring the code if necessary. By following TDD, we ensured that our code met specific requirements outlined by tests, resulting in cleaner, more modular code with fewer bugs.

```
makeMemoryFile("mem_in.hex",
noOp,
noOp,
noOp,
noOp,
noOp,
HexBigToLilEndian("0x00418133"),
haltInst
)
```

Figure 6: An Figure Showing an Example of A Simple Test Case to Pass

Before diving into implementing new features, we needed to ensure that the original functionality of the CPU remained intact. The test case depicted in Figure 6, a basic add instruction (add x2, x3, x4), served as our initial litmus test, guaranteeing the correct execution of fundamental operations before proceeding. Subsequently, we expanded the scope of testing by executing sequences of add instructions in succession, introducing branching statements between

instructions, and finally integrating a single instance of forwarding. Only after rigorously completing these preliminary tests did we proceed to evaluate each type of forwarding—EX-EX, MEM-EX, and MEM-MEM—both individually and in concert. Following thorough testing, encompassing both our basic and advanced benchmarks (detailed in the Benchmarks section), we can confidently assert that our CPU is devoid of bugs for the most common use cases. However, it's important to note that while extensive testing significantly reduces the likelihood of encountering bugs, absolute bug-free assurance remains elusive, despite our exhaustive efforts spanning thousands of lines of assembly code.

Example of Test Driven Development

When developing full forwarding, we ran into three interesting problems with our first attempt:

```
always@(*)begin
    //mem-ex and mem-mem forwarding
    //are only concerned about loads
    if(opcode_MEM == `OPCODE_LOAD) begin
        RegA_ID_Data = (Rdst_MEM_Name == RegA_ID) ? Rdst_MEM_Data : RegA_ID_cur;
        RegB_ID_Data = (Rdst_MEM_Name == RegB_ID) ? Rdst_MEM_Data : RegB_ID_cur;
    end
    //ex-ex forwarding
    //works with everything but loads, we don't need to forward data in stores
    if(opcode_EX != `OPCODE_LOAD && opcode_EX != `OPCODE_STORE) begin
        RegA_ID_Data = (Rdst_EX_Name == RegA_ID) ? Rdst_EX_Data : RegA_ID_cur;
        RegB_ID_Data = (Rdst_EX_Name == RegB_ID) ? Rdst_EX_Data : RegB_ID_cur;
    end
end
```

Figure 7: A Figure Showing the First Attempts at Full Forwarding

According to the testing framework outlined earlier, this code does work up until forwarding stacked add instructions. However, when attempting to perform MEM-MEM forwarding and EX-EX forwarding at the same time, there are 3 main errors. Firstly, we overlooked the scenario where the register being forwarded was x0. Although it might seem illogical to write back to the x0 register, the design never validated whether the instructions had the write-back permissions. Compounding this issue, the default write-back register for a store instruction is x0, leading to both intentional and unintentional writes to x0 being improperly forwarded. This oversight became glaringly obvious once fixed, but resulted in numerous test bench failures. After addressing these issues, another complication emerged: when a load instruction in the MEM stage wrote to the same register as the second input register in the EX stage, and the instruction in EX did not write to that register, the CPU erroneously forwarded the data from MEM to ID,

only to overwrite it back to its original value in EX. In resolving these three challenges, Seb devised a solution that, while effective, was far from elegant.

```
always@(*)begin
    if(opcode_EX != `OPCODE_LOAD && opcode_EX != `OPCODE_STORE && opcode_EX != `OPCODE_BRANCH && opcode_EX != `OPCODE_JAL && opcode_EX != `OPCODE_JALR && RegWREN_EX == 1'b0) begin
        RegA_ID_Data = (Rdst_EX_Name == RegA_ID && RegA_ID != 5'b0) ? Rdst_EX_Data : RegA_ID_cur;
        RegB_ID_Data = (Rdst_EX_Name == RegB_ID && RegB_ID != 5'b0) ? Rdst_EX_Data : RegB_ID_cur;
        if((opcode_MEM == `OPCODE_LOAD ||  opcode_MEM == `OPCODE_COMPUTE ||  opcode_MEM == `OPCODE_ICOMPUTE ||  opcode_MEM == `OPCODE_JAL ||  opcode_MEM == `OPCODE_JALR) && RegWREN_MEM == 1'b0) begin
            RegA_ID_Data = (Rdst_MEM_Name == RegA_ID && Rdst_EX_Name != RegA_ID && RegA_ID != 5'b0) ? Rdst_MEM_Data : RegA_ID_Data;
            RegB_ID_Data = (Rdst_MEM_Name == RegB_ID && Rdst_EX_Name != RegB_ID && RegB_ID != 5'b0) ? Rdst_MEM_Data : RegB_ID_Data;
        end
    end
    else begin
        if((opcode_MEM == `OPCODE_LOAD ||  opcode_MEM == `OPCODE_COMPUTE ||  opcode_MEM == `OPCODE_ICOMPUTE ||  opcode_MEM == `OPCODE_JAL ||  opcode_MEM == `OPCODE_JALR) && RegWREN_MEM == 1'b0) begin
            RegA_ID_Data = (Rdst_MEM_Name == RegA_ID && Rdst_EX_Name != RegA_ID && RegA_ID != 5'b0) ? Rdst_MEM_Data : RegA_ID_cur;
            RegB_ID_Data = (Rdst_MEM_Name == RegB_ID && Rdst_EX_Name != RegB_ID && RegB_ID != 5'b0) ? Rdst_MEM_Data : RegB_ID_cur;
        end
        else begin
            RegA_ID_Data = RegA_ID_cur;
            RegB_ID_Data = RegB_ID_cur;
        end
    end
end
```

Figure 8: A Figure Displaying The Current Full Forwarding Module

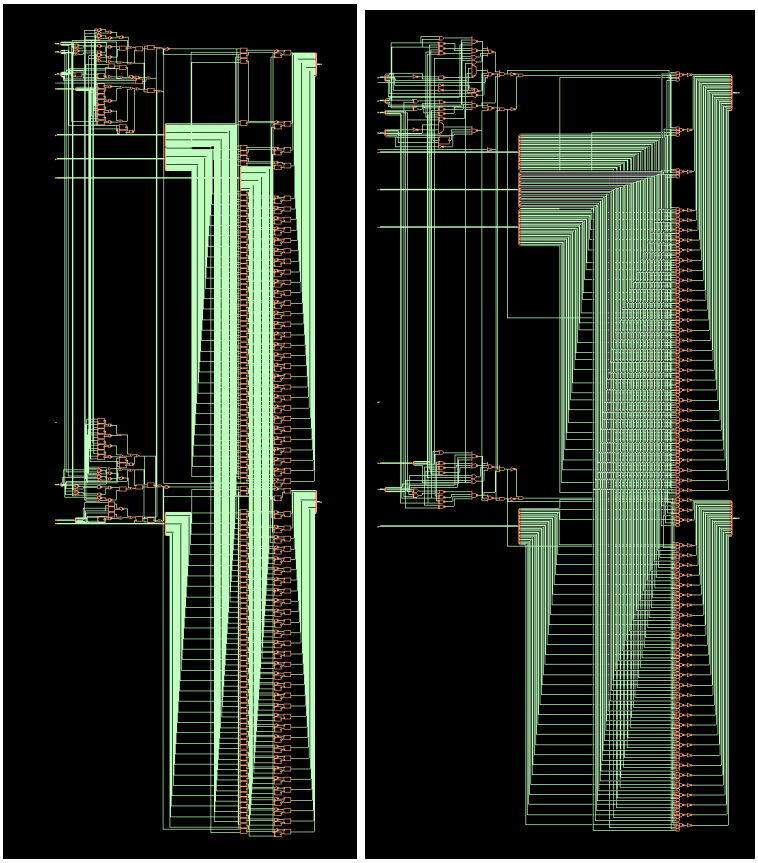


Figure 9: A Comparison of the Forwarding Without Optimizations (Left) and With High Optimizations (Right)

Upon reviewing Genus' optimizations of the Forwarding Unit in Figure 9, it looked as if it had optimized out/removed most of the gates that were redundant in the logic of the Forwarding Unit. The logical checks determining which data to forward accounted for approximately 25% of the unit's overall logic. Consequently, we made the decision to maintain the Forwarding Unit in

its current state, prioritizing the synthesis of the CPU and ensuring compliance with design requirements. This decision was further reinforced by our adherence to TDD principles, which guided us to focus on addressing immediate concerns while maintaining the integrity of existing functionality.

Broader Considerations

Commercial Feasibility

While the processor does work for base RV32I and RV32M instruction sets, there are areas in which the CPU falls short. Primarily, there are no I/O ports, sensors, or physical design. This would make the practical application of this CPU highly limited, as it cannot be used for desktop computers, embedded systems, or FPGA purposes.

However, on the other hand, because it is a virtual representation of a CPU, it is possible to test the functionality of an algorithm written in RISC-V for the potential speedup over x86-64 and ARM machines. Given that companies like TensTorrent are using RISC-V to develop CPUs, this product does have some commercial feasibility for testing the potential cycle speedup between the two assembly languages.

Now consider the CPU as a commercial product ready to take off in the market. Adding I/O ports becomes essential to improving its usability and attractiveness. These ports would allow for easy integration with depth sensors or cameras, meeting the needs of embedded devices that don't require Wi-Fi connectivity and need quick storage and single-threading capabilities.

Ethical Considerations

As a side effect of manufacturing this CPU, we would need to ensure that the suppliers of the materials are sourcing their silicon and other raw materials ethically, there are fair labor laws in place and the transportation of the materials doesn’t generate unsustainable amounts of GHG emissions.

Design description

Instruction Fetch Stage

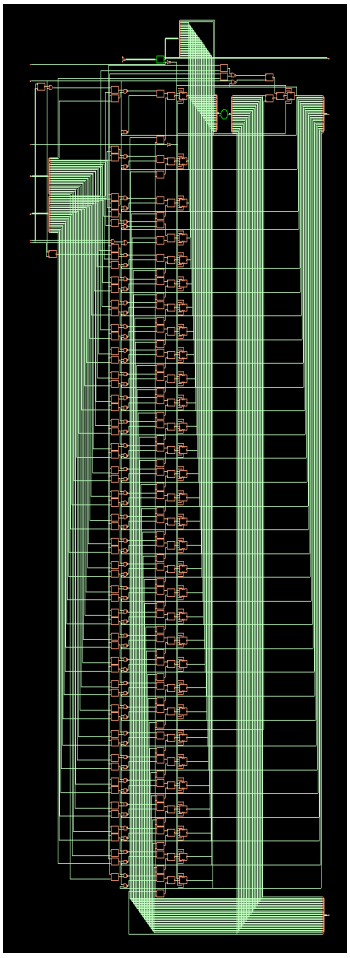


Figure 10: A Figure Displaying the Instruction Fetch Module With No Optimizations

The instruction fetch of our CPU is very similar to that of the diagram in Figure 3. It contains the Instruction Memory, program counter (PC), PC select, and the target address for the next PC. Most of the area inside of the module is logic that determines which PC source to use—the target address from EX, the predicted branch location from ID, or the current PC+4 (the next instruction in memory). Each signal is 32 bits long, seen in Figure 10.

Instruction Decode Stage

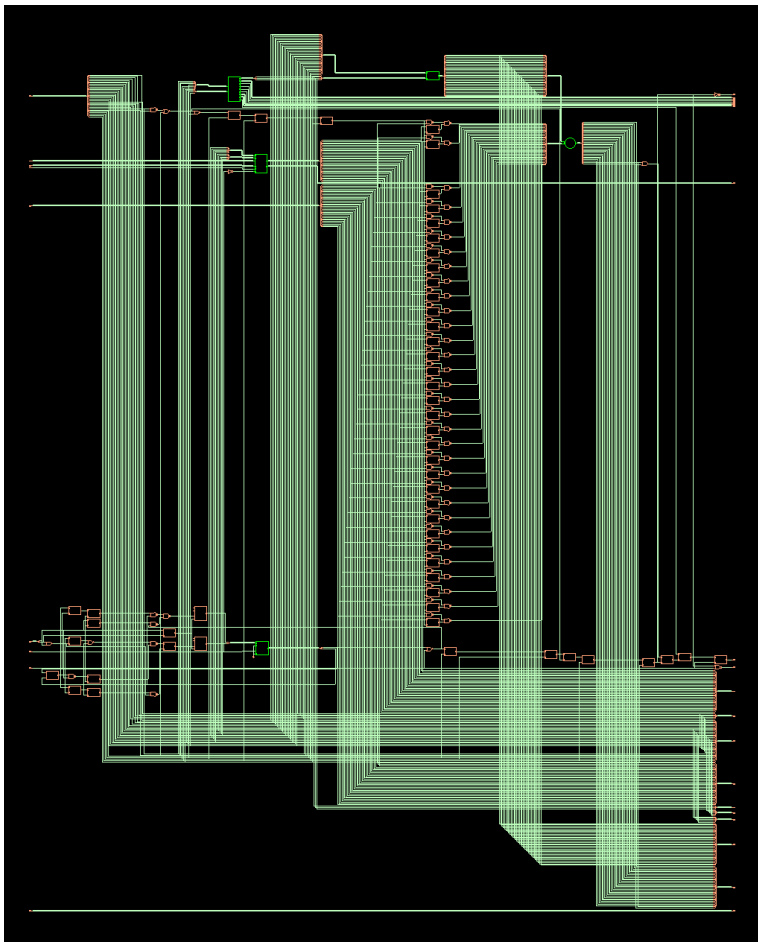


Figure 11: A Figure Displaying the Instruction Decode Module With No Optimizations

In Instruction Decode, the biggest stage, we have the Register File, the Control Signal Generator, the Immediate Generator, and the Branch Predictor. Also included is the PC Align module, which raises an exception (by setting the global halt signal) if the target address calculated from the branch predictor is misaligned.

Inside of the Control Signal Generator, the following signals were generated based on the current instruction in the ID stage (set them to 0 if not applicable):

Control Signal	Purpose
ImmSel	Identifying the form of the immediate (I-Type, B-Type)
RWrEn	Low enabled Register File Write Signal
ALUSrcA	Chooses the Source into the ALU between RegA and PC
ALUSrcB	Chooses the Source into the ALU between RegB and Imm
MemWrEn	Low enabled DMEM (data memory) Write Signal
WBSel	Selects the source of data to be written back between ALU out and DMEM out
MemSize	Determines the amount of data to read from DMEM

Table 3: A Table describing the Global Control Signals used in the Processor

These signals are passed through each of the pipeline registers to the EX, MEM, and WB stages.

Also relevant, but not necessarily part of the ID stage, are the Stall and Forwarding Units. They take register data from both the EX and MEM stage and alter the data, if needed, going into the EX stage. These modules work together in tandem to resolve read after write (RAW), write after read (WAR), and write after write (WAW) data hazards. The Forwarding Unit handles RAW hazards by forwarding the result of a previous instruction directly to an instruction that requires it, avoiding the need to wait for it to be written back to the register file. WAW hazards are also resolved by forwarding, ensuring that the most recent result is available to subsequent instructions.

WAR hazards occur when a write to a register follows a read from the same register in another instruction, potentially causing incorrect behavior if not handled properly. Full forwarding doesn't eliminate WAR hazards because the data being written can't be forwarded to the instruction that already read the data earlier. In this case, the Stall Unit will stall until the write operation that is causing the hazard has completed.

While there is a global halt signal to inform the user of the CPU that something has gone wrong, the CPU is intended to finish all instructions before asserting the halt globally. This means that every stage in the pipeline has its own local halt signal that can be triggered either when a previous stage in the CPU has raised an exception or when an exception has been raised inside of the current stage of the processor.

Execute Stage

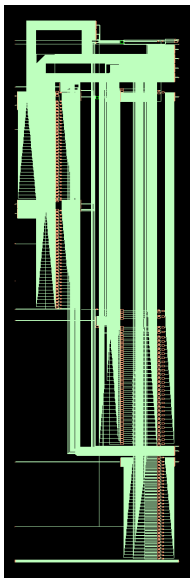


Figure 12: A Figure Displaying the Execute Module With No Optimizations

The EX stage contains our Branch Unit (BU), which verifies whether the prediction in the previous ID stage was correct or not, and the ALU, which handles all required arithmetic.

If there is a branch in EX, and it was predicted correctly as taken, the BU sets control signals to squash the instruction directly after the branch instruction in ID, taking a total of two cycles. If the branch was predicted correctly as not taken, nothing happens, and the pipeline continues loading instructions from the PC+4. If the branch was taken incorrectly, the BU squashes both the instruction after the branch in ID and the branched-to instruction in IF, resetting the next instruction to PC+4. Finally, if the branch was not taken incorrectly, the BU will squash the same two instructions, except this time setting the next PC to where the branch points to. In both the incorrect cases, the branch instruction takes three cycles to execute.

Memory Access/Writeback Stage

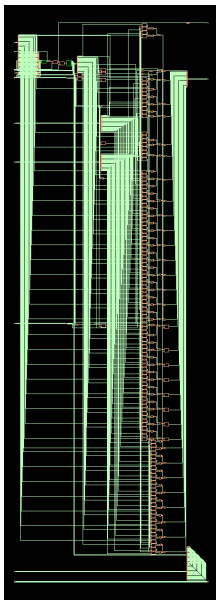


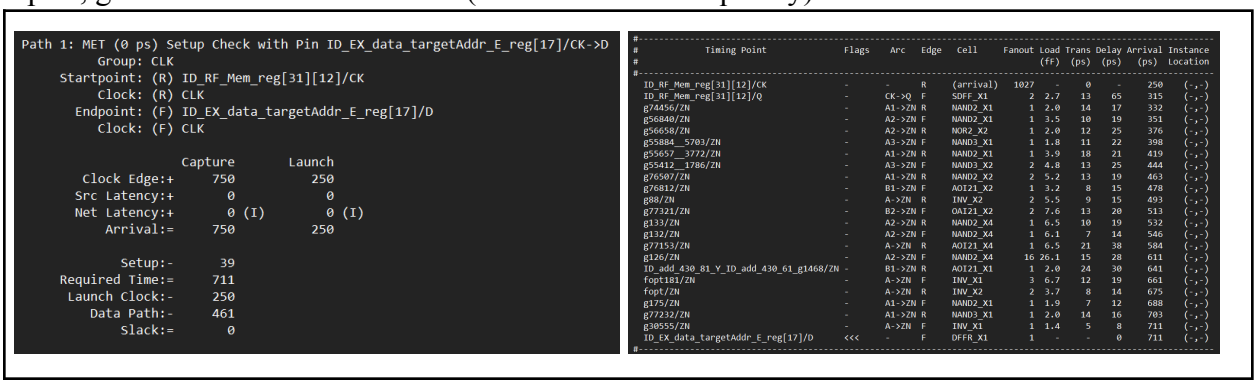
Figure 13: A Figure Displaying the Memory Access Module With No Optimizations

In Memory Access, we have a Size Check module (a memory size & alignment checker), an interface to Data Memory, and a sign extender module for the data that was read in. The function of these modules is relatively self explanatory: the data is accessed, it's asserted that nothing went wrong while doing so, and it's propagated into the WB stage for writing to the register file as well as into the Forwarding Unit. In WB, we have a MUX that determines the source of the data between DMEM out and ALU out and checks that it has permission to write back to the register file. If all conditions are met, we do a successful writeback to memory. Also in this stage, the local halt signal is finally turned into the global halt that stops the processor if it was enabled in any of the stages.

Performance/Testing

Timing Report

Our CPU meets the required clock timing with 0 slack. Displayed in Figure 13 is the timing report, given a clock time of 1.25ns (800MHz clock frequency).



```
Path 1: MET (0 ps) Setup Check with Pin ID_EX_data_targetAddr_E_reg[17]/CK->D
        Group: CLK
   Startpoint: (R) ID_RF_Mem_reg[31][12]/CK
        Clock: (R) CLK
     Endpoint: (F) ID_EX_data_targetAddr_E_reg[17]/D
        Clock: (F) CLK

                Capture       Launch
   Clock Edge:+     750          250
   Src Latency:+      0            0
   Net Latency:+      0 (I)        0 (I)
       Arrival:=    750          250

         Setup:-     39
  Required Time:=    711
   Launch Clock:-    250
      Data Path:-    461
          Slack:=      0
```

```
#-------------------------------------------------------------------------------------------------------------------
#            Timing Point                Flags    Arc   Edge   Cell      Fanout Load Trans Delay Arrival Instance
#                                                                          (fF)  (ps)  (ps)  (ps)  Location
#-------------------------------------------------------------------------------------------------------------------
  ID_RF_Mem_reg[31][12]/CK                -      -     R     (arrival)   1027    -     0     -     250    (-,-)
  ID_RF_Mem_reg[31][12]/Q                 -      CK->Q F     SDFF_X1        2  2.7    13    65    315    (-,-)
  g74456/ZN                               -      A1->ZN R    NAND2_X1       1  2.0    14    17    332    (-,-)
  g56840/ZN                               -      A2->ZN F    NAND2_X1       1  3.5    10    19    351    (-,-)
  g56658/ZN                               -      A2->ZN R    NOR2_X2        1  2.0    12    25    376    (-,-)
  g55884__5703/ZN                         -      A3->ZN F    NAND3_X1       1  1.8    11    22    398    (-,-)
  g55657__3772/ZN                         -      A1->ZN R    NAND2_X1       1  3.9    18    21    419    (-,-)
  g55412__1786/ZN                         -      A3->ZN F    NAND3_X2       2  4.8    13    25    444    (-,-)
  g76507/ZN                               -      A1->ZN R    NAND2_X2       2  5.2    13    19    463    (-,-)
  g76812/ZN                               -      B1->ZN F    AOI21_X2       1  3.2     8    15    478    (-,-)
  g88/ZN                                  -      A->ZN  R    INV_X2         2  5.5     9    15    493    (-,-)
  g77321/ZN                               -      B2->ZN F    OAI21_X2       2  7.6    13    20    513    (-,-)
  g133/ZN                                 -      A2->ZN R    NAND2_X4       1  6.5    10    19    532    (-,-)
  g132/ZN                                 -      A2->ZN F    NAND2_X4       1  6.1     7    14    546    (-,-)
  g77153/ZN                               -      A->ZN  R    AOI21_X4       1  6.5    21    38    584    (-,-)
  g126/ZN                                 -      A2->ZN F    NAND2_X4      16 26.1    15    28    611    (-,-)
  ID_add_430_81_Y_ID_add_430_61_g1468/ZN  -      B1->ZN R    AOI21_X1       1  2.0    24    30    641    (-,-)
  fopt181/ZN                              -      A->ZN  F    INV_X1         3  6.7    12    19    661    (-,-)
  fopt/ZN                                 -      A->ZN  R    INV_X2         2  3.7     8    14    675    (-,-)
  g175/ZN                                 -      A1->ZN F    NAND2_X1       1  1.9     7    12    688    (-,-)
  g77232/ZN                               -      A1->ZN R    NAND3_X1       1  2.0    14    16    703    (-,-)
  g30555/ZN                               -      A->ZN  F    INV_X1         1  1.4     5     8    711    (-,-)
  ID_EX_data_targetAddr_E_reg[17]/D       <<<    -      F    DFFR_X1        1    -     -     0    711    (-,-)
#-------------------------------------------------------------------------------------------------------------------
```

Figure 14: A Figure Displaying the Timing Report from Genus

As shown on the right side of Figure 14, the critical path is based on getting the data from the register file into the ID/EX register in time for the next clock cycle. This is expected, as the signals coming from the start of ID have to traverse through large amounts of transistors to grab the correct data in the register file and propagate this information to the registers into EX.

Area Report

Given the clock frequency of 800 MHz, our CPU must be under 30,000 square micrometers.

```
====================================================================
  Generated by:           Genus(TM) Synthesis Solution 18.14-s037_1
  Generated on:           Mar 07 2024  04:43:04 pm
  Module:                 PipelinedCPU
  Technology library:     NangateOpenCellLibrary revision 1.0
  Operating conditions:   typical
  Interconnect mode:      global
  Area mode:              physical library
====================================================================

  Instance   Module  Cell Count  Cell Area  Net Area   Total Area
--------------------------------------------------------------------
PipelinedCPU              7987  17345.328 11359.220    28704.548
```

Figure 15: A Figure Displaying the Area report from Genus

As shown in Figure 15, our CPU does meet the requirements with an extra 1296^2 micrometers to spare. This was achieved via using high optimizations for syn_generic_effort, syn_map_effort, syn_opt_effort.

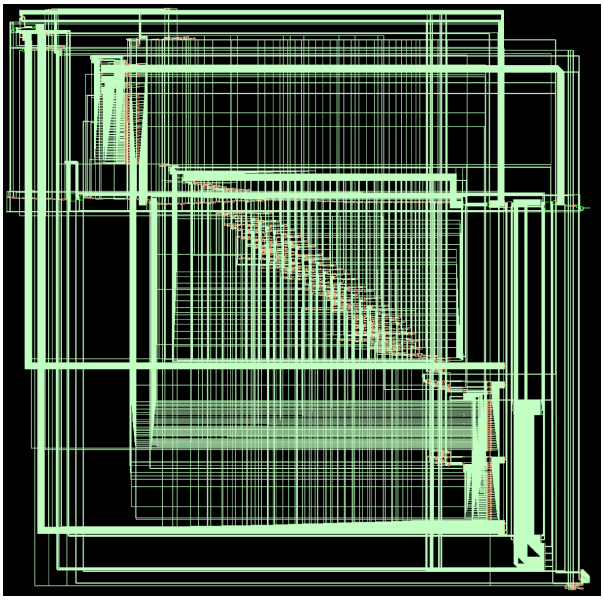


Figure 16: A Figure Displaying the PipelinedCPU module in Genus

As shown in Figure 16, this is our entire CPU without separations between the different submodules. It is very interesting to see that most of the area inside of the is empty. There can potentially be higher levels of optimizations to use less control signals in between stages in an attempt to make the design use less vertical area and therefore more area efficient, however, the CPU does meet the requirements and runs without bugs.

Benchmarks

In order to test the speed and correctness of our processor, we created two basic benchmarks and two advanced benchmarks. Written in RISC-V assembly, some are verifiable through comparison of register file or memory values, while others serve as a speed test.

For example, basic benchmark 1 is a two-part test that verifies the handling of data hazards and the correctness of register and immediate type instructions. The first part includes edge cases and hazards for these instructions, and the second part has the ability to act as a speed test through writing bytes to memory. Bytes are initialized in memory as a string of ASCII characters; and, when run, the algorithm will capitalize and write them back to memory. With a long enough string of bytes, the algorithm can take many cycles; however, basic benchmark 1 was generally used to test for register/immediate type correctness (see Appendix n: Basic Benchmark 1).

Conversely, basic benchmark 2 is a speed test with many branches, data hazards and memory hazards, but it doesn't have anything to verify. The algorithm increments a register by a custom amount and loads and stores from memory at a custom frequency until the value in the register reaches a custom total number. For consistency, we have kept the increment at 10,000, the

load/store frequency every 10 iterations (or until the increment reaches 100,000), and the total number at 166,191,000, which, coincidentally, is the annual profit in USD that Northwestern University makes from student tuition and housing, as of 2023. Effectively, basic benchmark 2 is an algorithm that branches 9 out of 10 times, many many times (See Appendix n: Basic Benchmark 2).

In order to see how effective the addition of full forwarding as well as a 2-bit branch predictor is, basic benchmark 2 was selected.

After running basic benchmark 2 on the unmodified processor with its default values of 10,000, 100,000, and 166,191,000, it took **118,004** cycles to finish running.

With the addition of full forwarding and a 2-bit branch predictor, the same code was run, and it took **63,161** cycles to execute.

Calculating the Speedup:

$$speedup = \frac{118004}{63161} = 1.868$$

Because of its frequent data and memory hazards, as well as the sheer volume of branches, it makes sense that this benchmark has such a dramatic speedup from the unmodified pipelined processor—nearly cutting the execution time in half.

Compare this speedup to the same two versions of the processor running advanced benchmark 2, which has no data or memory hazards:

Without full forwarding and branch prediction, advanced benchmark 2 took **1,408,980** cycles to execute. With the modifications, it took **1,277,910** cycles.

Calculating the Speedup Again:

$$speedup = \frac{1408980}{1277910} = 1.103$$

Without hazards, this speedup is exclusively due to branch prediction. Yet, opposed to basic benchmark 2, advanced benchmark 2 is a recursive algorithm, and does not have any loops. Assuming that the processor predicts the branch instructions correctly half of the time, then the only place where cycles are saved is through jal and $jalr$ instructions, which are always predicted correctly and save a cycle each compared to the unmodified processor.

If we count the number of regular, jump, and branch instructions, then we could calculate the speedup without executing the benchmark. However, with fifteen levels of recursion, counting

becomes a little difficult. Instead, we can approximate the speedup using one level of recursion, as seen below.

The number of cycles advanced benchmark 2 takes to execute without branch prediction:

$$\begin{aligned} cycles &= (\#\ of\ regular\ instr\ *\ 1) + (\#\ of\ jumps\ *\ 3) + (\#\ of\ branches\ *\ 3) \\ &= 32\ *\ 1 + 5\ *\ 3 + 2\ *\ 3 \\ &= 53 \end{aligned}$$

Now, with the branch predictor:

$$\begin{aligned} cycles &= (\#\ of\ regular\ instr\ *\ 1) + (\#\ of\ jumps\ *\ 2) + (\#\ of\ branches\ *\ 2.5) \\ &= 32\ *\ 1 + 5\ *\ 2 + 2\ *\ 2.5 \\ &= 47 \end{aligned}$$

$$projected\ speedup = \frac{53}{47} = 1.13$$

The projected speedup of 1.13 is slightly higher than the actual speedup of 1.103, but this variance is within an acceptable margin of error given the approximation method used and the complexity of the benchmark. As a surprise factor, the branch predictor accomplished its task of increasing the speed of the processor—through cycles taken.

The benchmarks provided invaluable insights into the tangible impact of our implemented changes and additions to the processor, demonstrating their effectiveness in enhancing both speed and correctness under varying computational conditions.

Functionality Testing/Demo Day

For the final test of functionality of our CPU, we ran two testbench simulations written by Prof. Panitan, fibo10 and fibo10_dp. As witnessed by the Professor, both of these testbenches, when run on our CPU, had the correct output memory and registers.

Conclusion

The development of a high-performance RISC-V CPU architecture has been a challenging yet rewarding endeavor. By adhering to specific constraints and requirements, we have crafted a robust and efficient processor architecture capable of meeting the demands of modern computing applications while ensuring compatibility with industry-standard design practices and synthesis tools.

Through a meticulous design approach, incorporating Test Driven Development (TDD) principles, we have navigated through the complexities of developing a 5-stage RISC-V pipelined CPU, emphasizing modularity and simplicity to facilitate debugging and optimization processes.

Performance evaluation through timing and area reports, along with extensive benchmarking, provides tangible evidence of the processor's capabilities and effectiveness. The successful completion of functionality testing, validated by comprehensive testbench simulations, reaffirms the reliability and correctness of our CPU architecture. The integration of full forwarding and a 2-bit branch predictor has significantly enhanced the processor's performance, reducing cycle time and increasing overall efficiency by almost double in some programs.

Despite the notable achievements in meeting performance metrics and functionality requirements, it's essential to acknowledge areas for future improvement and development in our CPU architecture. While our CPU functions correctly, there exists potential for enhancing its performance and scalability.

One avenue for improvement lies in reducing the critical path in the ID stage to achieve a higher clock frequency, ideally reaching 1 GHz. By minimizing delays, we could enhance the processor's speed and responsiveness.

A potential development in the future is the implementation of a cache for Data Memory (DMEM). Introducing a cache system would mitigate memory access latency, particularly when scaling the processor's working memory size into the hundreds of megabytes or gigabytes. By incorporating a cache mechanism, we could optimize memory access times and improve overall system performance, enabling our CPU to efficiently handle larger and more complex computational tasks.

Developing our CPU has been a difficult task, and it would not have been possible without the incredible support of our Professor, Classmates, and most importantly, Ran.

References

"Fig. 1 A Block Diagram of Typical Five-Stage Pipelined Processor..." *ResearchGate*, https://www.researchgate.net/figure/A-block-diagram-of-typical-five-stage-pipelined-processor-baseline_fig1_347538569. Accessed 11 Mar. 2024.

Hanes, Mirela. "Why Test-Driven Development (TDD)." *Marsner Technologies*, 10 Dec. 2019, https://marsner.com/blog/why-test-driven-development-tdd/.

Mallidu, Jayashree, and Saroja V. Siddamal. "A Survey on In-Order 5-Stage Pipeline RISC-V Processor Implementation." *Information and Communication Technology for Competitive Strategies (ICTCS 2021)*, Springer, Singapore, 2023, pp. 777–84. *link.springer.com*, https://doi.org/10.1007/978-981-19-0098-3_73.

"Tower of Hanoi." *Wikipedia*, 11 Mar. 2024. *Wikipedia*, https://en.wikipedia.org/w/index.php?title=Tower_of_Hanoi&oldid=1213256832.

Appendix 1: Basic Benchmark 1

Basic benchmark 1 has two parts in its code: Rainbow and SHOUT. The rainbow section runs through every single register and immediate type instruction to check for correctness, and the SHOUT section takes in a sequence of ASCII characters and capitalizes them, hence the names.

To verify correctness, run this benchmark with the given input registers and compare x0-x15 with the output registers.

In the DATA section in memory, a sequence of bytes can be placed, and they will be capitalized by the code. The input characters are located at memory address 0x70, and the capitalized characters will be stored directly after the end of the input characters. The character length of the sequence must be set in x16 before running the code.

As the default example, the characters "65 78 61 6d 70 6c 65" are located in the DATA section in memory, and x16 is equal to 7; the number of bytes in the sequence. When the benchmark is run, the data right after the above sequence becomes "45 58 41 4d 50 4c 45". Converting to ASCII, the input "65 78 61 6d 70 6c65" is "example" and the output "45 58 41 4d 50 4c 45" is "EXAMPLE". This sequence can be customized.

<table>
<tr><th>Memory</th><th>Registers</th></tr>
<tr><td rowspan="2">// Rainbow:

b3 80 10 00 // add x1, x1, x1
33 01 11 40 // sub x2, x2, x1
b3 c1 30 00 // xor x3, x1, x3
33 e2 30 00 // or x4, x1, x3
b3 f2 12 00 // and x5, x5, x1
33 13 13 00 // sll x6, x6, x1
b3 53 33 00 // srl x7, x6, x3
33 a4 44 00 // slt x8, x9, x4
b3 b4 44 00 // sltu x9, x9, x4
93 80 00 22 // addi x1, x1, 0x220
13 41 31 33 // xori x2, x2, 0x333
93 e1 01 11 // ori x3, x3, 0x110
13 72 11 32 // andi x4, x4, 0x321
93 92 10 00 // slli x5, x1, 0x1
13 d3 07 01 // srli x6, x15, 0x10
93 d3 07 41 // srai x7, x15, 0x1
13 24 f4 ff // slti x8, x8, -0x1
93 34 f5 ff // sltiu x9, x10, -0x1</td><td>Register inputs:

x0: 00000000
x1: 00000001
x2: 00000002
x3: 00000003
x4: 00000000
x5: ffffffff
x6: 00000002
x7: 00000003
x8: 00000000
x9: ffffffff
x10: 00000000

x15: 88880000
x16: 00000007 // length of chars
x17: 00000070</td></tr>
<tr><td>Register outputs:

x0: 00000000</td></tr>
</table>

// SHOUT:
 	
33 08 18 01 // add x16, x16, x17
b3 89 09 01 // add x19, x19, x16
 	
6f 00 80 01 // jal x0, cond
// body:
03 89 08 00 // lb x18, 0(x17)
13 79 f9 0d // andi x18, x18, 0xdf
93 88 18 00 // addi x17, x17, 1
23 80 29 01 // sb x18, 0(x19)
93 89 19 00 // addi x19, x19, 1
// cond:
e3 16 18 ff // bne x16, x17, body
 	
// halt
00 00 00 00
 	
////////////////////////////////
// data
65 78 61 6D 70 6C 65
////////////////////////////////
	x1: 00000222
	x2: 00000333
	x3: 00000111
	x4: 00000321
	x5: 00000444
	x6: 00008888
	x7: ffff8888
	x8: 00000000
	x9: 00000001
	x10: 00000000
	
	x15: 88880000
	x16: 00000077
	x17: 00000077

Appendix 2: Basic Benchmark 2

Basic Benchmark 2 iteratively adds a custom number (x2) up to a custom total (x5), and will store and load the value from memory at a custom milestone. It also stores the sum calculated in memory at address 0x20.

These inputs are customizable, but are set at default values. The total is set as the yearly revenue made by Northwestern from student tuition and housing ($166,191,000), the iterative number is $10,000, and you load and store from memory every $100,000.

Be careful with using really small iterative numbers and really large totals, as you could end up taking hundreds of millions of cycles.

The store and load value can be set to the total to test pure addition and branching, and it can be set to the iterative number to store and load at every iteration.

Although this benchmark is primarily used as a speed test, to check for correctness with the default values, the word at address 0x20 should be equal to 0x09e802c0 after execution.

Memory	Registers

// code:
33 01 00 00 // add x2, x0, x0
33 01 11 00 // add x2, x2, x1
e3 6e 31 fe // bltu x2, x3, -4
03 22 00 02 // lw x4, 0x20(x0)
33 02 22 00 // add x4, x4, x2
23 20 40 02 // sw x4, 0x34(x0)
e3 64 52 fe // bltu x4, x5, -24
// halt
00 00 00 00
// data:
00 00 00 00
	x0: 00000000
	x1: 00002710 // 10,000
	x2: 00000000
	x3: 000186A0 // 100,000
	x4: 00000000
	x5: 09e7df98 // 166,191,000
	x6: 00000000

Appendix 3: Advanced Benchmark 1

Advanced Benchmark 1 tests the functionality of forwarding during WAR data hazards, branching and branch predictors. The test takes the square value of a number via the multiply instruction, and then manually calculates the square of x via adding the value of x onto itself x times. This can be used as both a correctness check as well as a speed test. Because it heavily involves the multiply instruction, it can also be used to compare different implementations of multiplication support.

For its default values, with no forwarding and a Branch Predictor of not taken, the entire program takes 28,703 cycles.

This benchmark can be run with dynamically long number of words and dynamic values to be squared, for testing the short or long term speedup when implementing new features on the CPU.

Memory	Registers

63 8a 00 04 // beq x1, x0, 84
33 ae 00 00 // slt x28, x1, x0
63 12 0e 04 // bne x28, x0, 68
93 04 40 05 // addi x9, x0, 84
33 00 00 00 // nop
 	
03 a5 04 00 // lw x10, 0(x9)
b3 05 a5 02 // mul x11, x10, x10
 	
13 06 05 00 // addi x12, x10, 0
93 06 00 00 // addi x13, x0, 0
b3 86 a6 00 // add x13, x13, x10
33 06 e6 01 // add x12, x12, x30
e3 1c c0 fe // bne x0, x12, -8
 	
33 67 00 00 // or x14, x0, x0
63 92 b6 00 // bne x13, x11, 4
33 67 ef 01 // or x14, x30, x30
13 07 10 00 // addi x14, x0, 1
23 a0 e4 00 // sw x14, 0(x9)
23 91 b4 00 // sh x11, 2(x9)
b3 80 e0 01 // add x1, x1, x30
93 84 44 00 // addi x9, x9, 4
e3 94 00 fc // bne x1, x0, -56
 	
// halt
AB EE FF C0
 	
// INSERT YOUR FIRST VALUE HERE
	x1: 00000008 - The number of words in memory to multiply together

03 00 00 00
//INSERT YOUR SECOND VALUE HERE
05 00 00 00
// ...you get the pattern
10 00 00 00
// another
19 00 00 00
//
00 01 00 00
//
ff 00 00 00
//
ab 05 00 00
//
10 00 00 00

```
makeMemoryFile("mem_in.hex",
  #0x0
  HexBigToLilEndian("0x04008c63"), # beq x1, x0, 74              go to debug opcode (ends)
  HexBigToLilEndian("0x0000ae33"), # slt x28, x1, x0            set 28 to 1 if x1 is less than 0
  HexBigToLilEndian("0x040e1663"), # bne x28, x0, 76             go to debug opcode (ends) (only if negative)
  HexBigToLilEndian("0x05800493"), # addi x9, x0, 88             put pointer in x9
  HexBigToLilEndian("0x00000033"), # nop
  #20
  HexBigToLilEndian("0x0004a503"), # lw x10, 0(x9)
  HexBigToLilEndian("0x02a505b3"), # mul x11, x10, x10
  #Manual multiplyer
  HexBigToLilEndian("0x00050613"), # addi x12, x10, 0    setup the iter val
  HexBigToLilEndian("0x00000693"), # addi x13, x0, 0     setup the tmp val
  HexBigToLilEndian("0x00a686b3"), # add x13, x13, x10   add to itself
  HexBigToLilEndian("0x01e60633"), # add x12, x12, x30  subtract 1
  HexBigToLilEndian("0xfec01ce3"), # bne x0, x12, -8     go up two inst
  #set the memory to be 1 if they match, 0 if not
  HexBigToLilEndian("0x00006733"), # or x14, x0, x0      set the reg to be 0
  HexBigToLilEndian("0x00b69263"), # bne x13, x11, 4     skip if it does match
  HexBigToLilEndian("0x01ef6733"), # or x14, x30, x30    set the reg to be -1
  HexBigToLilEndian("0x00100713"), # addi x14, x0, 1     add 1
  HexBigToLilEndian("0x00e4a023"), # sw x14, 0(x9)       store success bit back into mem
  HexBigToLilEndian("0x00b49123"), # sh x11, 2(x9)       store the lower half of the word into the high part of mem
  #go to the next word in memory
  HexBigToLilEndian("0x01e080b3"), # add x1, x1, x30
  HexBigToLilEndian("0x00448493"), # addi x9, x9, 4
  HexBigToLilEndian("0xfc0092e3"), # bne x1, x0, -60
  HexBigToLilEndian("0xC0FFEEAB"), #debug opcode
  #Words in mem
  HexBigToLilEndian("0x00000003"),
  HexBigToLilEndian("0x00000005"),
  HexBigToLilEndian("0x00000010"),
  HexBigToLilEndian("0x00000019"),
  HexBigToLilEndian("0x00000100"),
  HexBigToLilEndian("0x00000000"),
  HexBigToLilEndian("0x00000000"),
  HexBigToLilEndian("0x00000000"),
  HexBigToLilEndian("0x00000000") )
```

Appendix 4: Advanced Benchmark 2

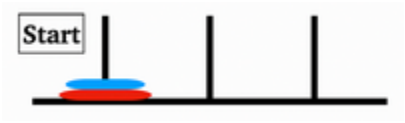
For this benchmark, it is important to be familiar with the Towers of Hanoi problem: ("Tower of Hanoi")

In the input registers, x1 is n, or the number of discs in the Towers of Hanoi problem. There are three pillars, the source, target, and auxiliary pillar, labeled as tower 1, 3, and 2 respectively. These numbers are stored in x2-x4.

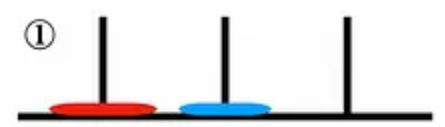
The benchmark has the capacity to record the actual "moves" that are made in the Towers of Hanoi problem. Each move is stored in memory as half words that can be easily read. These halfwords start at memory address 0xb0, and look something similar to 0x0121, or 0x0223. They can be read from left to right, for example:

0x0ABC --> Disc A is moved from tower B to tower C.
0x0121 --> Disc 1 is moved from tower 2 to tower 1.
0x0213 --> Disc 2 is moved from tower 1 to tower 3.

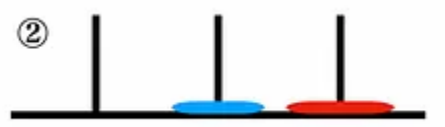
As an example, let n be equal to 2. Below is what the starting state looks like, with 2 discs on tower 1:



The first move is 0x0112:



Next 0x0213:



And finally 0x0123:

Figures 17-21: A graphical representation of the Towers of Hanoi

For the benchmark itself, n is set to 15 discs. There is nothing to verify; however, execution takes over a million cycles—it is used as a speed test.

Memory	Registers

6f 07 40 01 // jal x14, hanoi
 	
00 00 00 00 // code after function call
00 00 00 00
00 00 00 00
00 00 00 00
 	
// hanoi:
 	
93 87 c7 fe // addi x15, x15, -20
23 a0 e7 00 // sw x14, 0(x15)
23 a2 57 00 // sw x5, 4(x15)
23 a4 67 00 // sw x6, 8(x15)
23 a6 77 00 // sw x7, 12(x15)
23 a8 87 00 // sw x8, 16(x15)
 	
93 82 00 00 // addi x5, x1, 0
13 03 01 00 // addi x6, x2, 0
93 83 01 00 // addi x7, x3, 0
13 04 02 00 // addi x8, x4, 0
 	
33 00 00 00 // nop, x9 = 1
63 86 92 04 // beq x5, x9, output
 	
// recur1:
 	
93 80 f2 ff // addi x1, x5, -1
13 01 03 00 // addi x2, x6, 0
93 01 04 00 // addi x3, x8, 0
13 82 03 00 // addi x4, x7, 0
6f f7 1f fc // jal x14, hanoi
 	
6f 00 40 03 // jal x0, output
	x0: 00000000
	x1: 0000000f
	x2: 00000001
	x3: 00000003
	x4: 00000002
	x9: 00000001
	x10: memory storing addr
	x14: stack pointer
	x15: bottom of the stack

```
        // recur2:

        93 80 f2 ff   // addi x1, x5, -1
        13 01 04 00   // addi x2, x8, 0
        93 81 03 00   // addi x3, x7, 0
        13 02 03 00   // addi x4, x6, 0
        6f f7 9f fa   // jal x14, hanoi

        // exithanoi:

        03 a7 07 00   // lw x14, 0(x15)
        83 a2 47 00   // lw x5, 4(x15)
        03 a3 87 00   // lw x6, 8(x15)
        83 a3 c7 00   // lw x7, 12(x15)
        03 a4 07 01   // lw x8, 16(x15)

        93 87 47 01   // addi x15, x15, 20

        // ret
        67 00 07 00   // jalr x0, 0(x14)

        // output:

        13 85 02 00   // addi x10, x5, 0
        13 15 45 00   // slli x10, x10, 4
        33 05 65 00   // add x10, x10, x6
        13 15 45 00   // slli x10, x10, 4
        33 05 75 00   // add x10, x10, x7
        23 90 a5 00   // sh x10, 0(x11)
        93 85 25 00   // addi x11, x11, 2

        e3 84 92 fc  // beq x5, x9, exithanoi
        6f f0 1f fb  // jal x0, recur2
```

Here is the same code in Python, where n is the number of discs:

```python
def hanoi(n, source, target, auxiliary):
        if n > 0:
        # Move n-1 discs from source to auxiliary peg
        hanoi(n-1, source, auxiliary, target)
        # Move the nth disc from source to target peg
        print(f"Move disc {n} from {source} to {target}")
        # Move the n-1 discs from auxiliary peg to target peg
        hanoi(n-1, auxiliary, target, source)
# Example usage:
hanoi(3, 'A', 'C', 'B')
```