Illinois Institute of Technology

ECE485 Project #3

Pipelined stripped-down MIPS design and implementation in VHDL

Robert John Soler
Gabriel Roskowski

Due date: December 4, 2022

Acknowledgement: *I acknowledge that all works including figures, codes and writings belong to me and/or persons who are referenced. I understand if any similarity in the code, comments, customized program behavior, report writings and/or figures are found, both the helper (original work) and the requestor (duplicated/modified work) will be called for academic disciplinary action.*

Abstract

A Central Processing Unit, more commonly known as a CPU, is the electronic circuitry that performs all the instructions that make a computer program. It is responsible for all the calculations, arithmetic, logic, controlling, and input/output operations done by the computer. This project will go over the implementation of a 32-bit RISC architecture MIPS CPU, from the behavioral code of its components, to the structural implementation of its datapath and its testing and results.

Today, multiple large-scale companies are battling over the superiority of CPUs, like Intel and AMD. This battle has been going on for a few decades now, and this has caused CPUs to get more and more advanced and complex as time passes. This project cannot focus on CPUs of this caliber for two reasons: first, datapath information about these CPUs are not open to the public, and second, these CPUs are far too complex to be done by two students for their final project. Instead, the CPU model that will be followed is of the MIPS architecture. This is a very old model and has been open sourced to the public. It is also a lot simpler than the CPUs being used today, making it perfectly feasible for a final project.

This project is done in VHDL, a Hardware Descriptive Language (HDL) used to facilitate the modeling of complex hardware. The CPU will follow a combination of both behavioral and structural model style of implementation. This means that the code will focus on both the instructions and logic behind each of the individual components, as well as the architecture and structure of the entire CPU.

Introduction

Every computer has a CPU, whether it be an embedded system or a personal computer. Over the past few decades, companies have developed their own models of CPUs, whether because they have machines that serve specialized functions, or because they want to dominate the CPU market. This has led to the development of different architectures of CPUs, which usually fall into kinds of architecture: the Complex Instruction Set Computer (CISC) and the Reduced Instruction Set Computer (RISC). Although these CPUs get the same job done, they are very different in architecture, and hold different advantages and disadvantages as well.

The CISC architecture is designed for CPUs that can execute a task in as few lines of code as possible with fewer registers. However, the instruction set of the CPU must be vast and complex in order to compensate for the reduced amount of hardware. This leads to its instructions being more than one word long. In addition, since the instructions are more complex, the decoding process is also complex. This causes some instructions to have long execution times. This architecture is cheaper and can complete a task in lesser lines of code. However, it is slow compared to the RISC.

On the other hand, the RISC architecture is designed for CPUs that maximize throughput so that a task is performed in the least amount of time possible. However, for this to be possible, the instruction set and its instructions have to be a lot simpler than that of the CISC, so that it can be decoded and performed faster. To compensate for a smaller and simpler instruction set, the CPU needs more general purpose registers. This, in turn, increases the production cost. This architecture maximizes throughput and execution speed. However, it is expensive and requires the programmer to write many lines of code to execute a single task.

As mentioned in the abstract, this project will implement the MIPS CPU. MIPS stands for Microprocessor without Interlocked Pipeline Stages. It is an architecture family of RISC architecture. What makes it unique from other CPUs is its “load/store” characteristic. Basically, no instruction other than “store word” and “load word” can directly manipulate memory. All instructions’ operands have to be in registers. Therefore, before an operation is performed, the operands have to be loaded into registers first. The result is always placed in a register too, and to place it into memory, the “store word” instruction has to be used. Hence the characteristic name, “load/store”.

This project aims to implement a functioning 32-bit RISC CPU using both behavioral and structural models. Its functionality will be tested in a testbench where it will be fed different instructions, and the output will be observed.

This project is done in VHDL, and Hardware Descriptive Language (HDL) used to facilitate the modeling of complex hardware. These models are implemented using VHDL in the Xilinx Vivado IDE.

System Design

i. Project Design

The CPU built in this project will be based on the MIPS architecture. However, for the sake of feasibility, it will not be the complete MIPS CPU, but a stripped-down version of it, with a smaller version of its datapath and fewer instructions. It will be built as a combination of behavioral and structural models. Behavioral models will be used for the individual subcomponents of the CPU (Full adder, Register File, Memory Units, etc.), while the structural model will be used for connecting all these subcomponents to form the CPU itself and the ALU.

Another limitation imposed is that there will be no user interface where the user types the assembly instructions and compiler that compiles these instructions. Rather, the hexadecimal value of these instructions will just be manually fed to the instruction memory of the CPU.

ii. Datapath Design

Since this project's CPU is based off of the MIPS architecture, its datapath will also follow that of the MIPS architecture. However, since it is just a stripped-down version, it will not be the complete MIPS datapath with the hazard detection and exception hardware, or the data forwarding hardware, or instruction level parallelism. It also does not include the "shift left 2" and Full Adder hardware used by branch instructions as this architecture does not support branch instructions.

The most advanced feature of this CPU will be its five-stage pipelining feature. These stages are: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write-back (WB). First, the IF stage is where the CPU fetches the instructions from memory. Second, the ID is where the CPU decodes these instructions into data that it can process. Next, the EX stage is where the CPU executes the correct operations on the recently converted data based on the instruction given. Then, the MEM stage is where the memory access or write happens. This does not happen for all instructions, so sometimes it is just bypassed. Lastly, the WB stage is where the result of whatever operation happened in the EX stage is written into the destination register mentioned in the original instruction. This stage also does not happen for some instructions so this stage sometimes gets ignored. These stages exist to maximize the throughput in order to get the most work done in the least amount of time possible. It does this by making all stages work on different instructions at the same time. Therefore, ideally, the CPU would be able to process five instructions at the same time in one clock cycle. All features that are required for this mechanism to work will be included as well. These include the multicycle mechanism, the buffer registers, and the control and ALU control hardware. More of this will be covered in the "Simulation Results and Discussions" section.

iii. Control Logic Design

The control logic determines the actions and operations the CPU does, and for this CPU, it is determined by the control hardware. For this project, the control hardware is not a component

with a proper structure of sub-components inside. Rather, it is just a set of conditional codes that determine what the control signals will be with respect to the op code. These control signals are the write-back, memory, and execution signals, and they go to the ALU control, MUXes the data memory and register file. The ALU control and the MUXs use these signals to choose what operation to do or which output to choose, respectively. Also, since this datapath is a pipelined one, these signals also go into the buffer registers so that they can be used in the future, when needed.

iv. System Flowchart

As mentioned in the datapath design, this CPU will just be a stripped down version of the original MIPS architecture. The original system flowchart used as the basis for this project's CPU is shown in figure 1.

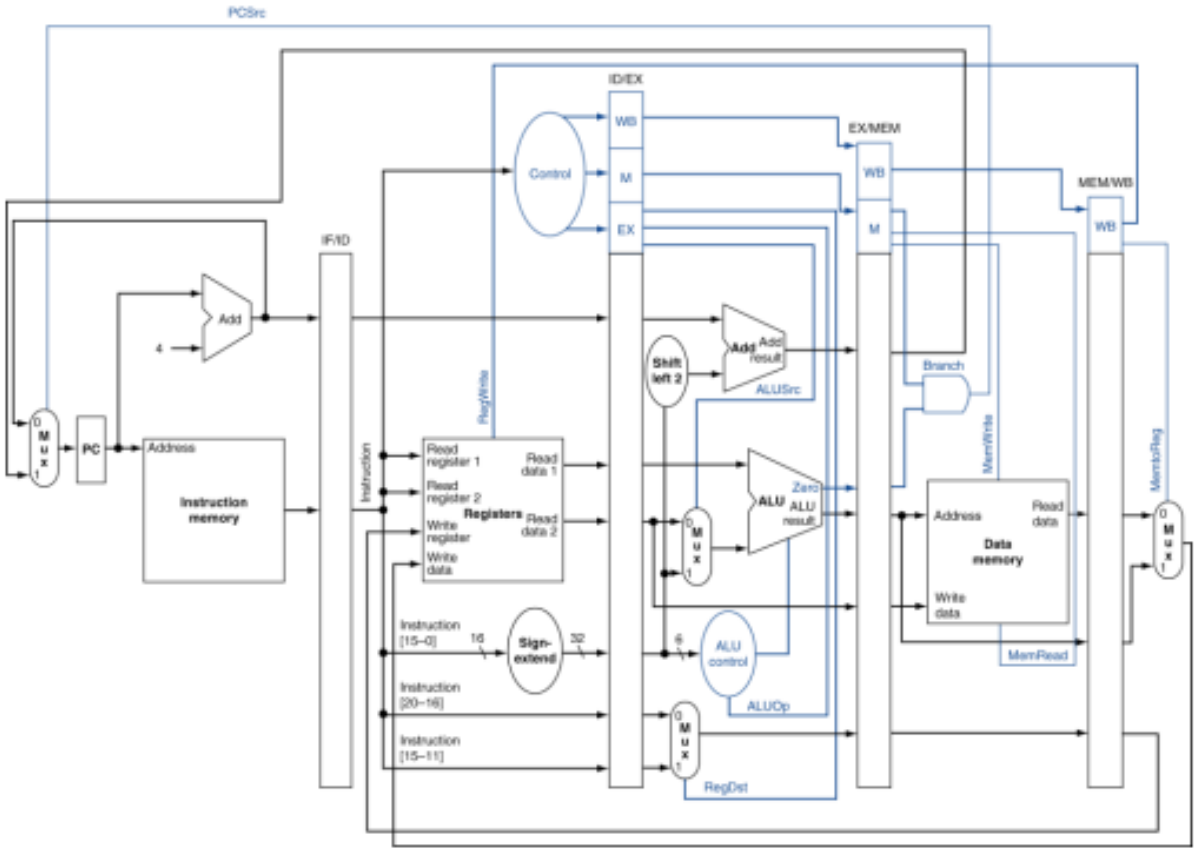

Figure 1: Datapath with Pipelined Control

However, since branch instructions are outside the scope of this project, the datapath in figure 1 was further simplified to remove hardware dedicated to branch instruction only. Figure 2 in the next page shows the datapath in figure 1, but with labels that show which components/connections have been implemented, and which ones have not.

In figure 2, there are labels for the components and the lines that connect them. The ones with checks above them mean that they have been finished. The ones encircled with orange are the

ones that were not implemented due to the CPU not supporting branch instructions. The lines that are highlighted in yellow are the ones that have been implemented. The components that these lines connect are implemented as well. Lastly, the words in red are the labels used for the signals.

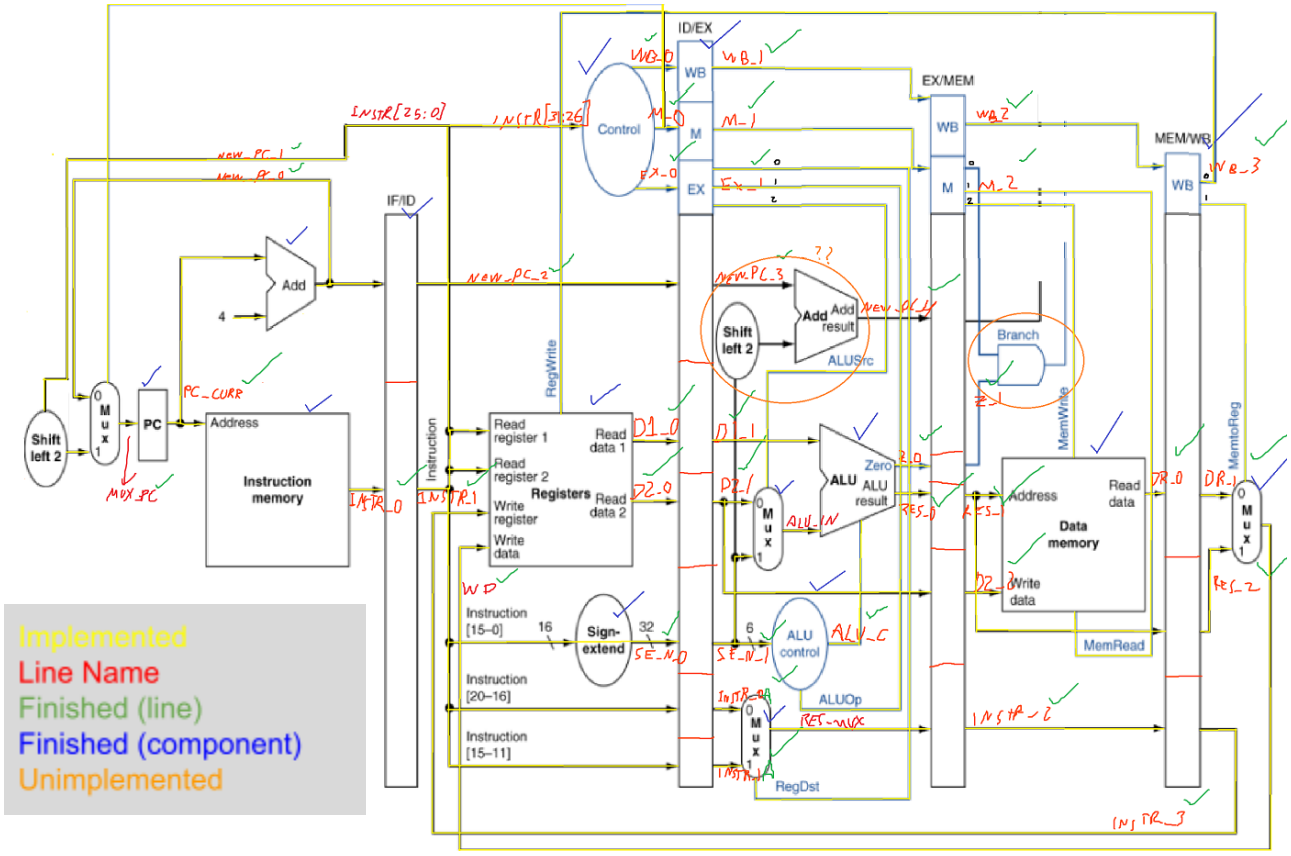

Figure 2: The modified version of the datapath in figure 1

It is important to note that although this CPU is already a significantly simplified one, its datapath still strongly resembles that of the original MIPS CPU architecture.

v. The ALU

Among all the components present in this project, the 32-bit ALU is the most unique one. It is the only component that doesn't use a fully behavioral model. Due to its unique behavior and the design requirements, this is the only component that is done with a structural model implementation.

This ALU is constructed as a composition of multiple single-bit ALUs inside. However, since there has already been existing code for the 32-bit RCA prior to this project, the single-bit ALUs were made without the addition function. So, this project does not make a 32-bit ALU from 32 single-bit ALUs that can perform all functions, rather, this project makes a 32-bit ALU from 32 single-bit ALUs with all functions except addition and a single 32-bit RCA that connects all of

them. That way, the 32-bit ALU would still be capable of doing all functions including addition, even when the single-bit ALUs were not capable of addition. The diagrams that describe these are shown below.

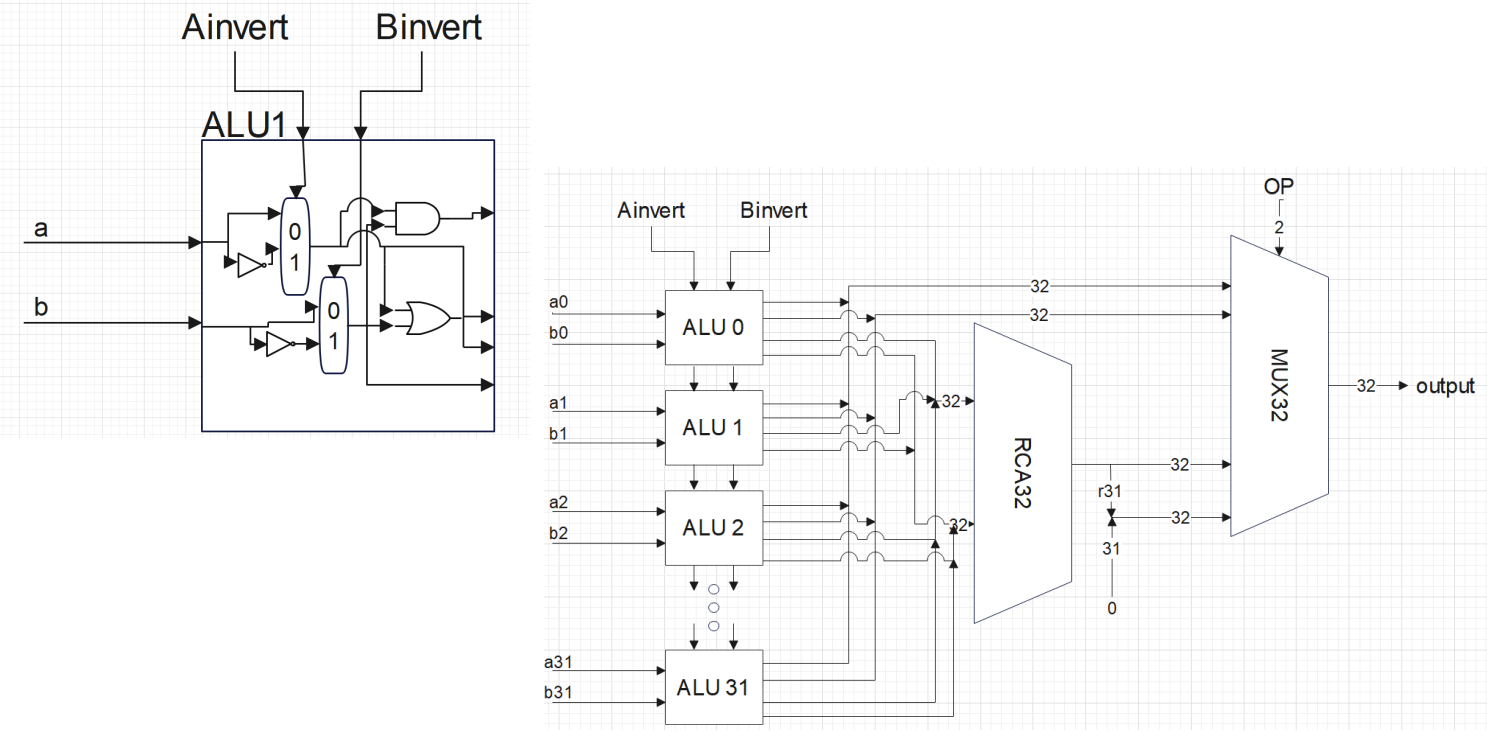

Figure 3: single-bit ALU (top-left) and structure of 32-bit ALU (right)

Note that the structure of the ALU was done like this for two main reasons. First, there has already been existing code on the 32-bit RCA prior to this project. Second, it would be more efficient to reuse that existing RCA code and do the rest as a parallel connection of single-bit ALUs.

Simulation Results and Discussion

i. Test Cases

This project uses multiple test cases to gauge the performance of the CPU. Each test case has its own purpose. The first test case attempts to demonstrate the pipelining mechanism that occurs within the CPU. To do this, only four subsequent instructions will be fed to the CPU. The second and third test cases run a different and longer set of assembly instructions, but with varying intervals between each instruction to show the use of “no-operations”, or NOPs. In addition, the fourth and fifth test cases are sets of assembly instructions that generate the fibonacci sequence and calculate the nth power of two, respectively, to show the capabilities of our CPU.

ii. Testing Results

The first test demonstrates the pipelining mechanism within the CPU. The assembly code used for this is as follows:

```
00000000 02339022 SUB  $s2, $s1, $s3
00000004 024D5024 AND  $t2, $s2, $t5
00000008 020A5025 OR   $t2, $s0, $t2
0000000C 01289820 ADD1 $s3, $t1, $t0
```

Code 1: assembly code to demonstrate the pipelining mechanism in the CPU

In this sample, and in all the future sample assembly codes, the first column is the memory location where the instruction is located. The second column is the hex representation of the assembly instruction, and the next column is just the assembly instruction.

This sample code used for this test case only uses four subsequent lines of assembly code. The test is done in this manner so that it becomes easier to see how all stages of the pipeline are capable of processing different sets of data at the same time. This will be elaborated in the next few paragraphs.

The CPU's output signals and results for this test case are shown in the next page. Note that the signals and outputs that are no longer relevant to the pipelining demonstration are removed so that the figure becomes simpler and easier to understand (these are the empty, black spaces surrounding the output signals).

Figure 4: Output for code 1

Notice how the signal outputs in figure 4 are divided into five different sections. These sections represent the five pipelining stages of the MIPS architecture, namely: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write-back (WB). In the image, these stages are stages 1 through 5, respectively.

To aid in the discussion of the pipelining mechanism, each instruction is outlined with a different color, so their location in the pipeline can be tracked. In figure 4, the data associated with the first instruction, `SUB $s2,$s1,$s3`, are bound by the red outline. The data associated with the second instruction, `AND $t2,$s2,$t5`, are bound by the yellow outline. The data associated with the third instruction, `OR $t2,$s0,$t2`, are bound by the orange outline. And lastly, the data associated with the fourth instruction, `ADD1 $s3,$t1,$t0`, are bound by the pink outline. In addition, white boundaries have been added to separate each clock cycle. This is done to show the alignment of all the five pipelining stages in every single clock cycle.

Now we can talk about the pipelining mechanism. At the first clock cycle, the first instruction, `SUB $s2,$s1,$s3,` enters the first stage. This is where the CPU fetches the instruction from memory. At this time, this is the only instruction being managed by the CPU.

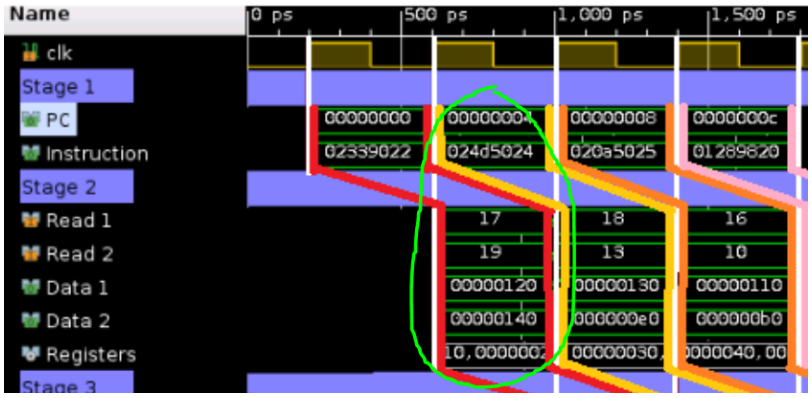

Figure 5: Pipelining in the second clock cycle

In the second clock cycle, this instruction advances to the ID stage, where the CPU decodes this instruction and determines the control logic. At the same time, the second instruction, `AND $t2,$s2,$t5`, enters stage 1. Notice how this is shown in figure 5. The data for the first instruction (outlined in red) is in stage 2, and the data for the second instruction (outlined in yellow) is in stage 1. However, they are both within the same set of white boundaries, meaning these two instructions are being processed in the same clock cycle. In other words, the CPU is working on these two instructions at the same time, only they are in different stages of the pipeline.

As the CPU advances to the third clock cycle, both the first and second instructions advance one stage. The first instruction goes from stage 2 to stage 3, and the second instruction goes from stage 1 to stage 2. At the same time, the third instruction, `OR $t2,$s0,$t2`, enters the first stage. Just like in the previous clock cycle, the data associated with each of the three instructions are aligned under the same clock cycle while they are in different stages. This means that the CPU is now processing three different instructions at the same time.

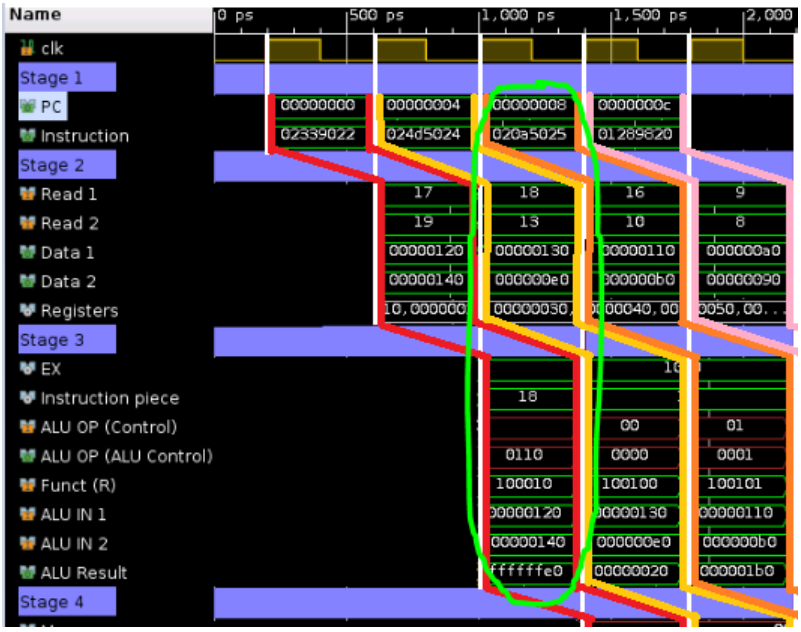

Figure 6: Pipelining in the third clock cycle

This same behavior keeps on repeating every clock cycle. All the instructions being processed in the CPU move to the next stage, and a new instruction enters the first stage.

The CPU then advances to the fourth clock cycle, where the maximum number of stages of the pipeline are occupied. Here, the first instruction is being processed in stage 4, the second instruction is being processed in stage 3, the third instruction is being processed in stage 2, and the most recent instruction just entered the pipeline in stage 1. All four instructions are being processed by the CPU at this moment, so we can say that for the set of instructions in code 1, the CPU has reached its maximum possible throughput. Of course, this CPU can handle five instructions at the same time because there are five stages. However, for this example, there are only four instructions, so only a maximum of four stages will be occupied at the same time.

As the CPU advances to the fifth clock cycle, the instructions once again advance by one stage. However, no more instruction enters the first stage because the last instruction of code 1 was fetched in the previous clock cycle. The CPU has run out of instructions to fetch. It is important to note that the first instruction's associated data has already reached the last stage of the pipeline in this clock cycle.

Once the CPU goes from the fifth clock cycle to the sixth, the data associated with the first instruction no longer has another stage to go to, since it has just finished the last stage, or the write-back stage. The result of the **SUB** instruction has just been written into the destination register `$s2`, so this instruction is officially done. Meanwhile, since no more instructions are being fetched by the CPU, and another one just finished, the number of instructions that the CPU is processing at the same time is now decreasing. This trend will continue from here on out.

Fast forward to the last clock cycle, the eighth one. The number of instructions being managed at the same time in the pipeline have been constantly decreasing from the fifth stage all the way to the eighth stage because instructions keep getting finished, but no new ones are being added (see previous paragraph). In this clock cycle, we are left with one last instruction. The three previous instructions have already been finished, and now the last instruction is already at the last stage. Once this clock cycle switches to the next, the last instruction will be finished as well, marking the end of the code. The CPU has finished running the code and is now at rest.

However, this pipelining demonstration is not the only observation that can be taken away from this test case. Return to code 1 and notice that the destination register, `$s2`, of the first instruction is actually one of the operands for the second instruction. This creates a problem with the data being processed in the pipeline. Take a look at figure 7.

Figure 7: Data dependencies gone wrong

Notice the two 18's encircled. 18 is the register number that corresponds to `$s2`, and it is present in the first and second instructions. This is where the problem comes in. Recall that in code 1, the second instruction depends on the final result of the first instruction. The CPU accesses register 18 in the third clock cycle to get the value stored in it and uses this as an operand for the second instruction. But, nothing has been written into this register yet! Why? Because the first

instruction's write-back stage doesn't happen until the fifth clock cycle. Nothing gets written into register 18 until the fifth clock cycle, so the final result that the second instruction is depending on is actually not there yet in the third clock cycle, where the CPU tries to fetch it. This causes the second instruction to execute with the wrong operand values, and thus lead to the wrong result. This is called a data hazard. and an effective solution to it is to introduce the concept of "no-operation" or "**NOP**" in short. This concept will be elaborated further in the next few paragraphs.

To go deeper into this concept, let's explore code 2.

```
00000000 02339022 SUB  $s2, $s1, $s3
00000004 024D5024 AND  $t2, $s2, $t5
00000008 020A5025 OR   $t2, $s0, $t2
0000000C 01289820 ADD1 $s3, $t1, $t0
00000010 8E4B0064 LW   $t3, 100($s2)
00000014 217400C8 ADDI $s4, $t3, 200
00000018 AD490064 SW   $t1, 100($t2)
0000001C 02284827 NOR  $t1, $s1, $t0
00000020 0252482A SLT  $t1, $s2, $s2
00000024 08000940 J    $2500
```

Code 2: test assembly code (without NOPs)

Before discussing the outputs of this set of code, the initial contents of the registers must be discussed first. In this project, the registers in the register file are initialized in a specific manner that gives them unique values inside. The initialized value follows the function `x = 16(n+1)`, where n is the register number and x is the value that will be stored in that register when it gets initialized. For example, `$at` is register #1. This means that the value stored in it when the CPU initializes will be 16(1+1) or 32. However, this is in decimal. In hex, it would be **20**. The hex value of the register's contents is what will be shown in the output. Here is a table of registers, their corresponding register number, and the value that will be stored in them at initialization:

Register	Register number	Value stored in initialization
$s0	16	0x00000110
$s1	17	0x00000120
$s2	18	0x00000130
$s3	19	0x00000140
$s4	20	0x00000150
$t0	8	0x00000090
$t1	9	0x000000a0
$t2	10	0x000000b0
$t3	11	0x000000c0

Table 1: Initial values for registers used in code 2

When the sample code is code 2 was fed to the CPU, the resulting behaviors and outputs were as follows:

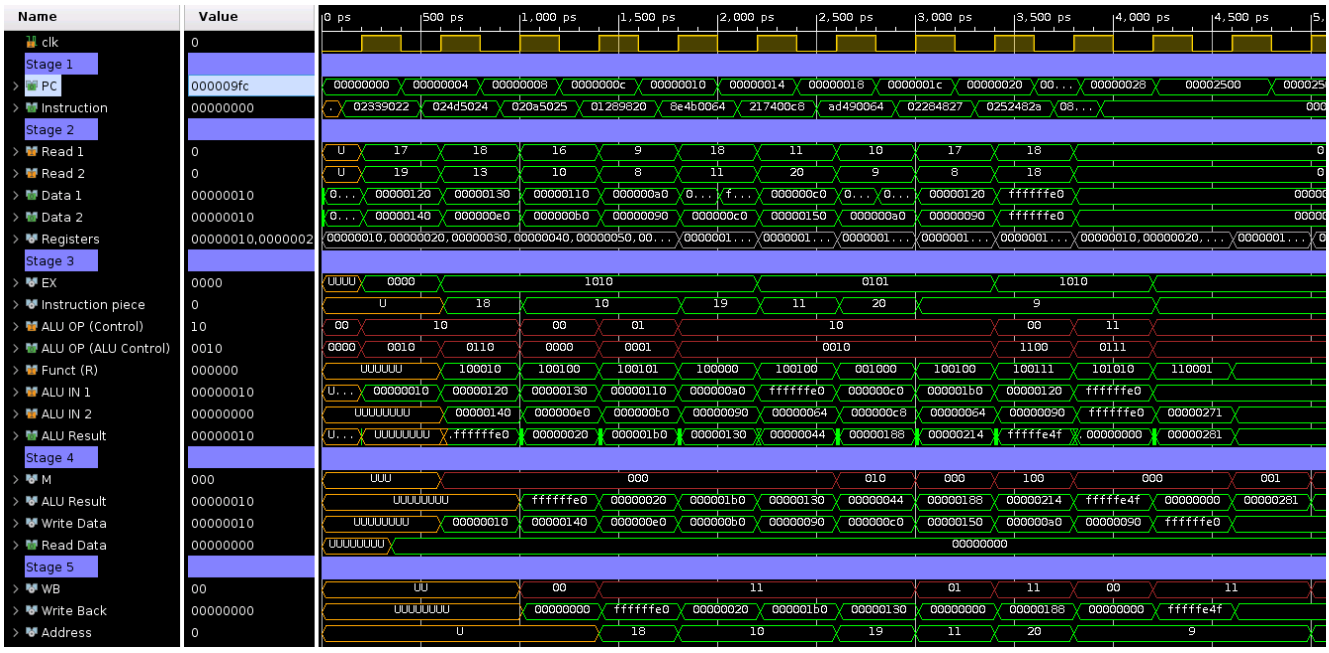

Figure 8: Output of code 2

It will not be a good idea to try and understand what the code is actually doing by looking at the output in figure 8. Rather, we will make a commented version of code 2, where the comments state what each line is actually doing with respect to the resources and data being used by the CPU.

```
1  02339022  SUB  $s2, $s1, $s3 # $s2 <= $s1 - $s3; $s1=0x120, $s3=0x140
2  024D5024  AND  $t2, $s2, $t5 # $t2 <= $s2 & $t5; $s2=0x130, $t5=0x0e0
3  020A5025  OR   $t2, $s0, $t2 # $t2 <= $s0 | $t2; $s0=0x110, $t2=0x0b0
4  01289820  ADD1 $s3, $t1, $t0 # $s3 <= $t1 + $t0; $t1=0x0a0, $t0=0x090
5  8E4B0064  LW   $t3, 100($s2) # $t3 <= 100($s2);  $t3=0x0c0, $s2= -32
6  217400C8  ADDI $s4, $t3, 200 # $s4 <= $t3 + 200; $t3=0x0c0
7  AD490064  SW   $t1, 100($t2) # $t1 => 100($t2);  $t1=0x0a0, $t2=0x1b0
8  02284827  NOR  $t1, $s1, $t0 # $t1 <= $s1 ↓ $t0; $s1=0x120, $t0=0x090
9  0252482A  SLT  $t1, $s2, $s2 # $t1 <= 1 if $s2 < $s2, else $t1 <= 0
10 08000940  J    $2500
```

Code 3: commented test assembly code (without NOPs)

Note the first two lines of the code. Notice how the result in line 1 is used as an operand in line 2. Line 1 performs the operation **0x120 - 0x140**, so the result stored in $s2 must be **0xfffffe0**. However, that is not what happens when the code is run. In line 2, $s2 still contains **0x130**. It hasn't been changed to **0xfffffe0**, which is the result of the previous operation. This is because of the pipelining design. The first instruction's result doesn't get written into the destination register until stage 5, which is still three clock cycles away. The instruction in line 2 starts before this write-back happens, so it ends up fetching the wrong value. This can be verified by referring to figure 8 and seeing that the write-back stage (stage 5) for the first instruction happens in the fourth clock cycle,

but the values used for the second instruction are fetched in the second clock cycle. Notice that the value fetched from register 18 in the second instruction is **0x00000130** instead of **0xffffffe0.**

Then in line 5, $s2 is used as an operand again. However, this time, it contains **0xffffffe0**, or -32. This is because it happens in the fifth clock cycle already, and the result for the first instruction got stored into its destination in the fourth clock cycle.

This phenomenon is called a "data hazard", and it happens when an operation is done incorrectly because of data dependencies that were not properly resolved. It happens when the values used for one instruction depend on the result of the previous instruction.

This hazard is present not only between the first and second instructions. It also happens between the second and third instructions, as well as the fifth and sixth instruction. The one between the fifth and sixth instruction is actually not a data hazard, but a "load-use" hazard. This happens when a register is used right after a load word instruction. In the complete MIPS architecture, the data hazard and the load-use hazard would have different effects, with the load-use hazard requiring one more NOP than the data hazard. However, since this CPU only uses a stripped-down MIPS datapath with no forwarding or hazard detection hardware like the complete one has, the effects of both the data hazards and the load-use hazard are just the same.

Now that we know where the hazards are and how to fix them, let's take a look at code 4.

```
0   02339022  SUB $s2, $s1, $s3 # $s2 <= $s1 - $s3; $s1=0x120, $s3=0x140
1   00000000  NOP
2   00000000  NOP                 # stall
3   00000000  NOP
4   024D5024  AND $t2, $s2, $t5 # $t2 <= $s2 & $t5; $s2 = -32, $t5=0x0e0
5   00000000  NOP
6   00000000  NOP                 # stall
7   00000000  NOP
8   020A5025  OR   $t2, $s0, $t2 # $t2 <= $s0 | $t2; $s0=0x110, $t2=0x0e0
9   01289820  ADD1 $s3, $t1, $t0 # $s3 <= $t1 + $t0; $t1=0x0a0, $t0=0x090
10  8E4B0064  LW   $t3, 100($s2) # $t3 <= 100($s2);  $t3=0x0c0, $s2= -32
11  00000000  NOP
12  00000000  NOP                 # stall
13  00000000  NOP
14  217400C8  ADDI  $s4, $t3, 200     # $s4 <= $t3 + 200; $t3=0x000
15  AD490064  SW $t1, 100($t2)  # $t1 => 100($t2);  $t1=0x0a0, $t2=0x1f0
16  02284827  NOR $t1, $s1, $t0 # $t1 <= $s1 ↓ $t0; $s1=0x120, $t0=0x090
17  0252482A  SLT $t1, $s2, $s2 # $t1 <= 1 if $s2 < $s2, else $t1 <= 0
18  08000940  J      $2500          # jumps to instruction address 2500
```

Code 4: commented test assembly code (with NOPs)

Code 4 contains the same instructions as code 3 does, but this time, notice that they have empty lines between some of them. These empty lines are called "no operations", or in short, "NOPs". They are placed before an instruction that has a dependency on the previous instruction in

an effort to stall it and wait for the result of the previous instruction to be stored first, so that the next instruction can fetch it.

As shown in code 4, the operand $s2 in the second instruction is already the result of the operation in the first instruction. This is because the three NOPs inserted between them have stalled enough clock cycles for the result of the subtraction to get stored into its destination register, before the second instruction fetches it. The same effect applies to the second and third instructions, as well as the fifth and sixth instructions. This can be verified in the output shown in figure 9. Here, the NOPs are those long intervals that are just zeroes. Now, the code runs as it was meant to.

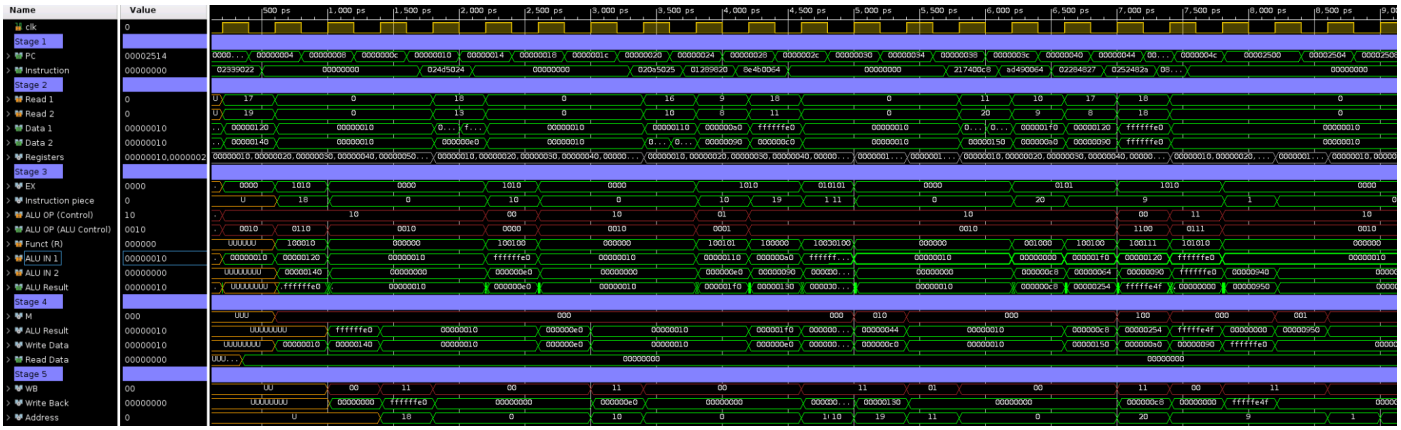

Figure 9: Output for code 4

One key takeaway from this test case is that this code's execution time is a lot longer than the code without NOPs. This is because inserting NOPs mean inserting intervals where the CPU does nothing at all. As an effect, the code takes longer to finish. However, it delivers the correct results, unlike the first one. There is a tradeoff here between throughput and correctness of results.

Of course, any interval where the CPU doesn't do anything is a waste of time. The CPU should, as much as possible, maximize its throughput. This is where hazard detection and forwarding come in. However, this concept is no longer part of the scope of this project, as it will require more hardware and signals that go beyond the datapath being used. They are, however, part of the original, full MIPS datapath.

The fourth test case is a code that calculates the Fibonacci Sequence. This sequence is one where each element is the sum of the two elements before it, with the exception of the first two elements, 0 and 1. Unfortunately, since the CPU has no hardware for branch instructions, conditional code is impossible. This limits it to an infinite loop that will keep on calculating the nth element of the sequence. The code for this is in the appendix. The testing results for this are shown in the next page. Note that they are all in hexadecimal format.

```
0, 1, 1, 2, 3, 5, 8, d, 15, 22, 37, 59, 90, e9, 179, 262, 3db, 63d, a18, 1055,
1a6d, 2ac2, 452f, 6ff1, b520, 12511, 1da31, 2ff42, 4d973, 7d8b5, cb228, 148add,
213d05, 35c7e2, 5704e7, 8cccc9, e3d1b0, 1709e79, 2547029, 3c50ea2, 6197ecb,
9de8d6d, ff80c38, 19d699a5, 29cea5dd, 43a53f82, 6d73e55f, b11924e1, 11e8d0a40,
1cfa62f21
```

Figure 10: Output of the Fibonacci Sequence generator

The fifth and last test case is a code that calculates the nth power of 2. The code for this is also in the appendix.

iii. Improvements and Additional Features

Due to the lack of available priority time we decided not to implement additional features. We thought of many improvements that could add to the system. Initially, we intended to implement conditional branching to allow for complex behavior and make an assembly code to do multiplication and division on the very simple hardware that only supported elementary operations.

iv. Minor notes

- The $zero register should have been made an exception and set to **0x00000000** due to its main purpose.
- All registers, independent of their type, are treated equally in our architecture, without validating anything.
- Most of the clock delays were removed or reduced for debugging purposes, but the original clock of 400ps was kept, as this was the longest stage delay, from the data memory block.
- Most signals and variables in the CPU were left uninitialized due to their lack of importance. The ones that needed to be initialized were set to their required values.

v. Optimizations

There are uncountable optimizations that could be made to make the assembly code perform better and more efficiently, as well as trying to reduce wasted clock cycles where the processor does not compute anything. Part of the optimizations could be done in hardware, part could be performed in a smarter assembler that applies a few concepts that can optimize, especially loops and repeated statements.

An important hardware optimization is the forwarding of calculated data directly to the following instructions that require it, without needing to stall the CPU. This reduces a wasted stalled clock cycle for instructions that can have forwarded register results.

Another relevant optimization could be to reorder the instructions in real time to optimize wasted stalled time, by putting instructions that don't depend on resources right before it. This would require some special hardware to account for jumps and, in case we had implemented, conditional branches.

Branch prediction would be an interesting optimization to implement in the hardware, as well as detection and optimization of loops, due to it being the most weight in most programs.

In terms of gate count and area reduction, almost no optimization was applied. Though, we can recognize a few minor and other major improvements. For the controller, Karnaugh maps could have been used to reduce the amount of required logic gates, however, a simple and readable version was implemented. For the PC+4 adder, most gates would vanish in an optimization that would consider a constant number 4, with many zeroes annihilating AND and OR gates or turning into constant values a few lines after gates. The most area of the die would be consumed by the instruction register and data memory. No effort was put into trying to optimize each 4 byte memory block, as it was done behaviorally and the VHDL synthesizer would take care of it, probably not as efficiently as a human could do, especially if not involving many specific features that were not needed.

vi. Issues Faced

There were several issues faced during the development of the stripped-down RISC hardware, including server access instabilities, excess verbose VHDL, code translating, and putting everything together.

The Endeavour server caused several issues to both of us due to major delays in some sessions. There was also a problem with keyboard encoding that was never fixed and reduced life quality by an extreme amount due to the incapability of using arrow and delete keys. One of us was also locked out of the system for several hours due to the password being reseted constantly while trying to login in the machine, after getting past the server login.

The unprecedented verbosity of VHDL and its rustic and primitive syntax was a minor issue, but still reduced our productivity. Its lack of smartness in assumptions also amounted to several hours of debugging, especially when trying to do type conversion or report making to validate the results. We tried SystemVerilog extra-officially and it was a much better experience, as it also includes object oriented capabilities.

The translation of code by hand was a major challenge due to zero error tolerance. Uncountable hours were wasted translating code and retranslating something because a single bit or two were wrong. We decided to create an assembler that took around 30 minutes to make and validate, which remarkably improved our code translation.

Putting everything together was a complicated process due to the amount of internal signals that needed to be perfectly connected between components. Even with a lot of organization and keeping visual track of what was already implemented and the name of all the internal signals, there were still a few flaws that took debugging to fix.

The assembler

The assembler used for this project was constructed in Java, through the same process of converting an instruction to hexadecimal by hand. For each line of the program, it separates the instruction into fragments delimited by whitespace. The first fragment is the instruction itself. The other fragments are determined based on the type of the instruction. If it is an R-type instruction it is supposed to expect three registers. If it is an I-type instruction, it expects two registers and a 16-bit immediate value. For a J-type instruction, it expects only a single 26-bit jump target. After this, the registers names are processed into their binary forms, as well as the binary opcodes of the instruction and the immediate value or target. Everything is concatenated and the final result is converted to hexadecimal, which is then fed directly to the instruction register.

As a way of preventing hazards, the assembler can introduce NOPs when resources that are used in the next few instructions depend on current instructions. All the resources that the current instruction will modify are put in a fixed-size FIFO queue. For all the instructions, it is checked whether its resources are in the queue or not. If they are, the assembler introduces NOP instructions and inserts a null item into the queue, until the queue does not contain the resource that is going to be used. The assembler code can be checked in appendix II.

Conclusion

After finishing this project, it is safe to say that this is one of the most, if not the most, complex and challenging projects we have ever done. With this comes a great deal of experience and a long list of learnings and conclusions.

First of all, this project was a complete success not only because it was able to function as it was expected to (based on the sample code), but also because this CPU is already capable of doing tasks that have significant applications in the real world, like the Fibonacci generator and the nth power of 2 generator, even when it was just a third of the original model it was based on.

Another thing to point out is that even though VHDL is still popular, it is immensely outdated due to the verbosity and lack of modern programming language features. Because of this, other organizations have developed better hardware description languages that are less verbose than VHDL, but still flexible and even have more features, such as object oriented structuring.

Multiple conclusions were drawn from the process of creating the CPU as well. First, starting with something simple that works and fills the requirements and then evolving and refining it was an interesting and effective practice that we applied in the development. Next, keeping a diagram of the entire datapath with the block diagrams and the connections between them was crucial for organizing and putting together the architecture without getting lost. For bigger and more complex architectures, organization plays an even bigger role in not getting lost. In addition, translating assembly code to hexadecimal is too time consuming and it is a valid point to make an

assembler that is capable of doing it automatically. Lastly, testing the individual components and applying the concept of modularity to make sure that they work before putting them together to create bigger parts is an important idea that reduces the amount of problems by a lot.

Though most of the learnings were from the experience of making the CPU itself, several relevant ones also came from general realizations while in the process of completing the project. For instance, though this hardware was only a third of the original model, and most of its code were straightforward, its physical hardware would have been ridiculously complex for college students to assemble from just transistors, gates, and wires. This only leads to the realization that modern CPUs, like Intel's i9 series or AMD’s Ryzen 9 series, are absolute pinnacles and masterpieces of electrical and computer engineering. However, the vast majority of people don’t realize this because they don’t see what’s happening within their computers, nor do they understand the complexity behind the seemingly simple tasks that their devices do. This is the exact reason why civil and mechanical engineers get far more credit than computer engineers do. The people can see how a well-done building can withstand a magnitude 7 earthquake. They can also see how a car can reach speeds of up to 240 mph. But they can never see how their lives were made a lot, lot easier with the help of gracefully forged 80 billion transistors.

List of References

Forums utilized:

arstechnica.com, thecodingforums.com, stackoverflow.com, electronics.stackexchange.com.

These websites were used for fixing bugs and glitches in VHDL, and learning better practices in implementation and validation.

MIPS information websites:

chortle.ccsu.edu/assemblytutorial/index.html,

d.umn.edu/~gshute/mips/instruction-types.xhtml

ntu.edu.sg/home/smitha/fyp_gerald/index.html,

cs.umd.edu/~meesh/cmsc411/website/projects/outer/memory/align.htm

cs.kzoo.edu/cs230/Resources/MIPS/MachineXL/InstructionFormats.html

These websites were used for reviewing MIPS concepts and the assembly language that is used to program the MIPS, as well as the encoding and decoding process of each instruction format and their life cycles.

VHDL tutorial websites:

surf-vhdl.com, vhdlwhiz.com, cs.sfu.ca/~ggbaker/reference/std_logic/

fpga4student.com/p/vhdl-project.htmlw, nandland.com/generate/

web.cs.ucla.edu/Logic_Design/vhdlintro.html

These websites were mostly used to learn VHDL and better practices, while implementing a few features. They were also used to understand the VHDL standard logic library and its capabilities, as well as type conversion.

Tools:

https://www.calculator.net/hex-calculator.html

This tool was constantly used when converting assembly code to hexadecimal by hand and to validate the initial assembler conversions.

Appendix

I. Source Codes

```
1  ----------------------------------------------------------------------------------------------
2  -- University: Illinois Institute of Technology
3  -- Engineers: Robert Soler and Gabriel Roskowski
4  -- Module Name: 2 inputs 5 bit MUX
5  ----------------------------------------------------------------------------------------------
6  library IEEE;
7  use IEEE.STD_LOGIC_1164.ALL;
8  -- port definition. Accepts 2 inputs and outputs one of them based on the selector
9  entity MUX2_5 is
10     Port ( i0 : in STD_LOGIC_VECTOR(4 downto 0);
11            i1 : in STD_LOGIC_VECTOR(4 downto 0);
12            s : in STD_LOGIC;
13            z : out STD_LOGIC_VECTOR(4 downto 0));
14 end MUX2_5;
15 -- behavioral model of the mux
16 architecture Behavioral of MUX2_5 is
17 begin
18     process (i0, i1, s) is
19     begin
20         if s = '0' then z <= i0 after 15ps;    -- first choice (s=0)
21         elsif s = '1' then z <= i1 after 15ps; -- second choice (s=1)
22         end if;
23     end process;
24 end Behavioral;
```

Code 5: Behavioral model for a 5-bit 2x1 MUX

```
1  ----------------------------------------------------------------------------------------------
2  -- University: Illinois Institute of Technology
3  -- Engineers: Robert Soler and Gabriel Roskowski
4  -- Module Name: 2 inputs 32 bit MUX
5  ----------------------------------------------------------------------------------------------
6  library IEEE;
7  use IEEE.STD_LOGIC_1164.ALL;
8  -- port definition. Accepts 2 inputs and outputs one of them based on the selector
9  entity MUX2_32 is
10     Port ( i0 : in STD_LOGIC_VECTOR(31 downto 0);
11            i1 : in STD_LOGIC_VECTOR(31 downto 0);
12            s : in STD_LOGIC;
13            z : out STD_LOGIC_VECTOR(31 downto 0));
14 end MUX2_32;
15 -- behavioral model of the mux
16 architecture Behavioral of MUX2_32 is
17 begin
18     process (i0, i1, s) is
19     begin
20         if s = '0' then z <= i0 after 15ps;    -- first choice (s=0)
21         elsif s = '1' then z <= i1 after 15ps; -- second choice (s=1)
22         end if;
23     end process;
24 end Behavioral;
```

Code 6: Behavioral model for a 32-bit 2x1 MUX

```vhdl
----------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Behavioral model of 32 bit MUX with 4 inputs
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- port description. Receives 4 inputs and chooses one to output to z based on selectors s0 and s1
entity MUX4_32 is
    port(in0, in1, in2, in3 : in std_logic_vector(31 downto 0); s0, s1 : in std_logic; z : out std_logic_vector(31 downto 0));
end MUX4_32;
-- behavioral model of the MUX
architecture Behavioral of MUX4_32 is
begin
    process(in0, in1, in2, in3, s0, s1)
        variable temp : std_logic_vector(1 downto 0);        -- auxiliary variable
    begin
        temp := s1 & s0;                                     -- assigns temp variable
        if temp = "00" then z <= in0;                        -- and chooses the input based on temp to put into z
        elsif temp = "01" then z <= in1;
        elsif temp = "10" then z <= in2;
        else z <= in3 after 100 ps;
        end if;
    end process;
end Behavioral;
```

Code 7: Behavioral model for a 32-bit 4x1 MUX

```vhdl
----------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineer: Gabriel Felipe Roskowski
-- Module Name: FA1 - Behavioral
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--defining full adder ports
entity FA1 is
    port(a,b,c: in std_logic;q, carry: out std_logic);
end FA1;
--modelling full adder behavioraly
architecture Behavioral of FA1 is
begin
    fa: process (a,b,c) is
        variable qv, cv : std_logic := '0';         -- temporary variables
        begin
            qv := a xor b xor c;                    -- output
            cv := (a and b) or (c and (a xor b));   -- carry output
            q <= qv;
            carry <= cv;
        end process fa;
end Behavioral;
```

Code 8: Behavioral model for a 1-bit full-adder

```vhdl
----------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Behavioral adder (simplified)
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- port mapping. Accepts two inputs and adds
entity Adder is
    port(a, b: in std_logic_vector(31 downto 0);
        cin: in std_logic;
        q: out std_logic_vector(31 downto 0);
        cout: out std_logic);
end Adder;
-- behavioral model of adder
architecture Behavioral of Adder is
begin
    q <= a + b after 50ps;                  -- uses IEEE std logic unsigned library to add the std_logic vectors.
    cout <= '0' after 50ps;                 -- ignores cout (outputs 0 so that it isn't floating)
end Behavioral;
-- this adder could have been simplified further by removing the carry output and the need for carry input and a second input
-- chosen not to simplify due to other portions of the processor that might use it
```

Code 9: Modified adder for PC+4

```vhdl
----------------------------------------------------------------------------------
-- University:  Illinois Institute of Technology
-- Engineer: Gabriel Felipe Roskowski
-- Module Name: RCA4 - Structural
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- defining the entity
entity RCA4 is
    port(
        a_v: in std_logic_vector(3 downto 0);
        b_v: in std_logic_vector(3 downto 0);
        cin: in std_logic;
        q_v: out std_logic_vector(3 downto 0);
        cout: out std_logic);
end RCA4;
-- defining the structural model
architecture Structural of RCA4 is
    -- declaring components that are going to be used
    component FA1 is
        port(a, b, c: in std_logic; q, carry: out std_logic);
    end component;
    -- declaring a few signals that are going to be used
    signal c0, c1, c2: std_logic;
begin
    -- mapping signals to intputs and outputs
    fa_0: FA1 port map(a_v(0), b_v(0), cin, q_v(0), c0);
    fa_1: FA1 port map(a_v(1), b_v(1), c0, q_v(1), c1);
    fa_2: FA1 port map(a_v(2), b_v(2), c1, q_v(2), c2);
    fa_3: FA1 port map(a_v(3), b_v(3), c2, q_v(3), cout);
end Structural;
```

Code 10: Structural model for a 4-bit RCA made of 1-bit FAs

```vhdl
----------------------------------------------------------------------------------
-- University:  Illinois Institute of Technology
-- Engineer: Gabriel Felipe Roskowski
-- Module Name: RCA32 - Structural
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- defining entity with all port mapping
entity RCA32 is
    port(
        a_v: in std_logic_vector(31 downto 0);
        b_v: in std_logic_vector(31 downto 0);
        cin: in std_logic;
        q_v: out std_logic_vector(31 downto 0);
        cout: out std_logic);
end RCA32;
-- defining architecture model of RCA32
architecture Structural of RCA32 is
-- declaring components that are going to be used and mapping them
component RCA4 is
    port(
        a_v: in std_logic_vector(3 downto 0);
        b_v: in std_logic_vector(3 downto 0);
        cin: in std_logic;
        q_v: out std_logic_vector(3 downto 0);
        cout: out std_logic);
end component;
    -- signals that are going to be used
    signal c0, c1, c2, c3, c4, c5, c6: std_logic;
begin
    -- mapping all inputs to outputs or signals
    rca_0: RCA4 port map(a_v(3 downto 0), b_v(3 downto 0), cin, q_v(3 downto 0), c0);
    rca_1: RCA4 port map(a_v(7 downto 4), b_v(7 downto 4), c0, q_v(7 downto 4), c1);
    rca_2: RCA4 port map(a_v(11 downto 8), b_v(11 downto 8), c1, q_v(11 downto 8), c2);
    rca_3: RCA4 port map(a_v(15 downto 12), b_v(15 downto 12), c2, q_v(15 downto 12), c3);
    rca_4: RCA4 port map(a_v(19 downto 16), b_v(19 downto 16), c3, q_v(19 downto 16), c4);
    rca_5: RCA4 port map(a_v(23 downto 20), b_v(23 downto 20), c4, q_v(23 downto 20), c5);
    rca_6: RCA4 port map(a_v(27 downto 24), b_v(27 downto 24), c5, q_v(27 downto 24), c6);
    rca_7: RCA4 port map(a_v(31 downto 28), b_v(31 downto 28), c6, q_v(31 downto 28), cout);
end Structural;
```

Code 11: Structure model for the 32-bit RCA

```vhdl
----------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Single-bit ALU behavioral model
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- port description.
entity ALU1 is
    port(a, b: in std_logic;
         an, bn: in std_logic;
         andOut, orOut, aOut, bOut : out std_logic);
end ALU1;
-- behavioral description of the ALU bit
architecture Behavioral of ALU1 is
begin
    process (a, b, an, bn)
        variable ar, br : std_logic;                    -- temporary variables after inversion or not
    begin
        ar := a;
        br := b;
        if an = '1' then ar := not a; end if;           -- invert a if an
        if bn = '1' then br := not b; end if;           -- invert b if bn
        andOut <= ar and br;                            -- and output
        orOut <= ar or br;                              -- or output
        aOut <= ar;                                     -- inverted or not a
        bOut <= br;                                     -- inverted or not b
    end process;
end Behavioral;
```

Code 12: Behavioral model for the 1-bit ALU

```vhdl
----------------------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Structural model of the ALU using single bit ALUs and the RCA32 from P2
----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- port description.
entity ALU32 is
    port(a, b: in std_logic_vector(31 downto 0);
         op: in std_logic_vector(1 downto 0);
         an, bn: in std_logic;
         q: out std_logic_vector(31 downto 0);
         ov, zero: out std_logic);
end ALU32;
architecture Structural of ALU32 is
    -- declaration of components that are going to be used (normal single bit ALU and MSB ALU)
    component ALU1 is
        port(a, b: in std_logic;
         an, bn: in std_logic;
         andOut, orOut, aOut, bOut : out std_logic);
    end component;
    component RCA32 is
        port(a_v: in std_logic_vector(31 downto 0);
        b_v: in std_logic_vector(31 downto 0);
        cin: in std_logic;
        q_v: out std_logic_vector(31 downto 0);
        cout: out std_logic);
    end component;
    component MUX4_32 is
        port(in0, in1, in2, in3 : in std_logic_vector(31 downto 0);
            s0, s1 : in std_logic;
            z : out std_logic_vector(31 downto 0));
    end component;
    -- declaration of a few temporary signals, such as carry and set
    signal andOut, orOut, aOut, bOut, z, less: std_logic_vector(31 downto 0) := x"00000000";
    -- used to check if all elements of the 32 bit vector are zero
    function all_zero(v: std_logic_vector) return std_logic is begin
        if v = x"00000000" then return '1';
        else return '0';
        end if;
    end function;
    -- port mapping between compontents
begin
    -- generates all 31 mini ALUs required
    alus: for i in 0 to 31 generate
        alu_i : ALU1 port map(a(i), b(i), an, bn, andOut(i), orOut(i), aOut(i), bOut(i));
    end generate alus;
    -- maps the modified a and b outputs from the ALU to the RCA
    rca : RCA32 port map(aOut, bOut, bn, z, open);
    less <= "00000000000000000000000000000000" & z(31); -- creates the slt output
    -- chooses out of the 4 possible operations
    mux : MUX4_32 port map(andOut, orOut, z, less, op(0), op(1), q);
end Structural;
```

Code 13: Structural model for the 32-bit ALU

```vhdl
----------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Sign-Extender
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-- port definition. Gets the immediate part of the instruction and sign-extends it
entity SignExtender is
    port(instr : in std_logic_vector(15 downto 0);
        extended : out std_logic_vector(31 downto 0));
end SignExtender;
--behavioral model of the sign extender
architecture Behavioral of SignExtender is
begin
    process(instr)
        begin
            extended <= std_logic_vector(resize(signed(instr), 32)) after 10ps;
                                    -- converts the input to signed number and rezises
        end process                 -- it to a 32 bit vector
end Behavioral;
-- could have been done in the main module, but was chosen to be separated
```

Code 14: Behavioral model for the Sign-extender hardware

```vhdl
----------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: PC Register
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-- port definition. Buffers a 32 bit program counter with the clock
entity PC is
    port (pcin : in std_logic_vector (31 downto 0);
          clk : in std_logic;
          pcout : out std_logic_vector (31 downto 0));
end PC;
-- behavioral model of the PC register
architecture Behavioral of PC is            -- the storage of the current program counter. Starts at zero
    signal pcmem : std_logic_vector(31 downto 0) := X"00000000";
begin
    process(clk, pcin)
        begin
            if rising_edge(clk) then pcmem <= pcin;     -- consumes the input at the rising edge of the clock
            end if;
            pcout <= pcmem;                             -- always outputs the register
        end process;
end Behavioral;
```

Code: 15: Behavioral model for the PC register

```vhdl
----------------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Behavioral model of the register file
----------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-- port description. Reads registers addressed by readReg1 and readReg2 into readData1 and readData2.
-- Writes writeData to register addressed by writeReg. Only writes when the clock is rising and regWrite = 1
entity RegFile is
    port (readReg1 : in std_logic_vector (4 downto 0);
          readReg2 : in std_logic_vector (4 downto 0);
          writeReg : in std_logic_vector (4 downto 0);
          writeData : in std_logic_vector (31 downto 0);
          regWrite, clk : in std_logic;
          readData1 : out std_logic_vector (31 downto 0);
          readData2 : out std_logic_vector (31 downto 0)
          );
end RegFile;

-- behavioral model of the register file
architecture Behavioral of RegFile is
    -- data structure that contain 32 registers of 32 bits each
    type rftype is array(0 to 31) of std_logic_vector(31 downto 0);

    -- this function serves to precisely initialize the regFile with desired values
    function initialize return rftype is
    -- variable of custom structure to be set and returned (currently filled with zeroes)
        variable reg : rftype := (others=>(others=>'0'));
    begin
        for i in 0 to 31 loop
            -- arbitrary initialization (chosen like this because it is possible to track the register from its value
            reg(i) := std_logic_vector(to_unsigned(16 * (i + 1), 32));
        end loop;
        return reg;
    end function initialize;
    signal registArr : rftype := initialize;                                    -- uses custom initialization function

begin
    process(regWrite, clk, writeReg, readReg1, readReg2)
    begin
        if rising_edge(clk) and regWrite = '1' then         -- if clock is rising and regWrite is 1,
            -- writes writeData to the register file addressed by writeReg
            registArr(to_integer(unsigned(writeReg))) <= writeData after 5ps;
        end if;
        readData1 <= registArr(to_integer(unsigned(readReg1))) after 10ps;    -- read register 1
        readData2 <= registArr(to_integer(unsigned(readReg2))) after 10ps;    -- read register 2
    end process;

end Behavioral;
```

Code 16: Behavioral model for the register file

```
----------------------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Behavioral model of the instruction memory
----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-- port mapping (just receives program counter and outputs the instruction)
entity InstrMEM is
    port (pc : in std_logic_vector (31 downto 0);
          instr : out std_logic_vector (31 downto 0));
end InstrMEM;

-- the model of the instruction register
architecture Behavioral of InstrMEM is

    -- structure that holds 64k instructions, or, 256kB
    type imtype is array(0 to 65536) of std_logic_vector(31 downto 0);

    -- this function serves to precisely initialize the instruction with the desired values and everything else NOP
    function initialize return imtype is
        -- variable of custom structure to be set and returned (currently filled with zeroes)
        variable reg : imtype := (others=>(others=>'0'));
    begin
        -- demo program to generate Fibonacci sequence
        reg(0) := x"00000022";
        reg(1) := x"00000000";
        reg(2) := x"00000000";
        reg(3) := x"00000000";
        reg(4) := x"20080000";
        reg(5) := x"20090001";
        reg(6) := x"00000000";
        reg(7) := x"00000000";
        reg(8) := x"00000000";
        reg(9) := x"01205020";
        reg(10) := x"01094820";
        reg(11) := x"00000000";
        reg(12) := x"00000000";
        reg(13) := x"01404020";
        reg(14) := x"08000006";
        return reg;
    end function initialize;
signal registArr : imtype := initialize;  -- the container of the instructions that utilizes the custom initializer
begin
    process(pc)
        -- at all times, outputs the instruction at pc/4 (word-aligned)
        begin instr <= registArr(to_integer(unsigned(pc)/4)) after 100ps;
    end process;
end Behavioral;
```

Code 17: Behavioral model for the instruction memory

```vhdl
----------------------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Behavioral model of the data memory
----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--port mapping
entity DataMemory is
    port(clk, write, read: in std_logic;
        addr, writeData: in std_logic_vector(31 downto 0);
        readData: out std_logic_vector(31 downto 0));
end DataMemory;

-- behavioral description of the memory
architecture Behavioral of DataMemory is
    -- the memory has 2^16 4-byte words of storage (can be modified to have 4GB)
    -- initializing everything with zeroes
    type memory is array (0 to 65536) of std_logic_vector(31 downto 0);
    signal dataMemory : memory := (others=>(others=>'0'));
begin
    process(clk, addr, writeData, write, read)
    begin
        -- if the clock is rising and it is receiving writing signal, write into addr/4 (word-aligned) the value in writeData
        if rising_edge(clk) and write = '1' then
            dataMemory(to_integer(unsigned(addr))/4) <= writeData after 350ps;
        end if;
        if read = '1' then                              -- if it receives signal to read, read address addr/4 into readData
            readData <= dataMemory(to_integer(unsigned(addr))/4) after 350ps;
        else readData <= X"00000000" after 350ps;               -- othewise, readData becomes zero
        end if;
    end process;
end Behavioral;
```

Code 18: Behavioral model for the data memory

```vhdl
----------------------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: IFID buffer
----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- port definition. Buffers the instruction and the program counter with a clock. 64 bits total
entity IF_ID_Buffer is
    Port ( IF_Instruction : in STD_LOGIC_VECTOR (31 downto 0);
           IF_PC : in STD_LOGIC_VECTOR (31 downto 0);
           clk : in STD_LOGIC;
           ID_Instruction : out STD_LOGIC_VECTOR (31 downto 0);
           ID_PC : out STD_LOGIC_VECTOR (31 downto 0)
           );
end IF_ID_Buffer;
-- behavioral model of the buffer
architecture Behavioral of IF_ID_Buffer is
begin
    process is
        variable inst, PC : std_logic_vector(31 downto 0); -- stores both inputs in variables
    begin
        wait until rising_edge(clk);            -- when the clock raises, the instruction and program are
            inst := IF_Instruction;             -- passed from the input to the internal variables
            PC := IF_PC;
        ID_Instruction <= inst;                 -- and then to the outputs of the circuit.
        ID_PC <= PC;
    end process;
end Behavioral;
-- we attempted this different structure and it appears to be an advantage to define everything in terms of variables
-- and assign outputs only in the end of the process
```

Code 19: Behavioral model for the IF/ID buffer register

```vhdl
----------------------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: IDEX buffer
----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- this buffer uses the same concept as the IF_ID buffer. Check it for more detailed commenting.
-- port definition. Takes control logic commands (WB, M, EX), the program counter, the data fetched
-- from the register file, the sign extended immediate portion of the instruction and fragments of
-- the instructions, buffering it with the clock. 147 total bits.
entity ID_EX_Buffer is
    Port (ID_WB : in STD_LOGIC_VECTOR(1 downto 0);
          ID_M : in STD_LOGIC_VECTOR(2 downto 0);
          ID_EX : in std_logic_vector (3 downto 0);
          ID_PC : in std_logic_vector (31 downto 0);
          ID_data1 : in std_logic_vector (31 downto 0);
          ID_data2 : in std_logic_vector (31 downto 0);
          ID_signExtend : in std_logic_vector (31 downto 0);
          ID_instruction1 : in std_logic_vector (4 downto 0);
          ID_instruction2 : in std_logic_vector (4 downto 0);
          clk : in std_logic;
          EX_WB : out STD_LOGIC_VECTOR(1 downto 0);
          EX_M : out STD_LOGIC_VECTOR(2 downto 0);
          EX_EX : out std_logic_vector (3 downto 0);
          EX_PC : out std_logic_vector (31 downto 0);
          EX_data1 : out std_logic_vector (31 downto 0);
          EX_data2 : out std_logic_vector (31 downto 0);
          EX_signExtend : out std_logic_vector (31 downto 0);
          EX_instruction1 : out std_logic_vector (4 downto 0);
          EX_instruction2 : out std_logic_vector (4 downto 0)
          );
end ID_EX_Buffer;
architecture Behavioral of ID_EX_Buffer is
begin
    process is
    variable PC, data1, data2, signExtend : std_logic_vector(31 downto 0);
    variable M : STD_LOGIC_VECTOR(2 downto 0);
    variable WB : STD_LOGIC_VECTOR(1 downto 0);
    variable EX : std_logic_vector(3 downto 0);
    variable inst1, inst2 : std_logic_vector (4 downto 0);
    begin
        wait until rising_edge(clk);
            WB := ID_WB;
            M := ID_M;
            EX := ID_EX;
            PC := ID_PC;
            data1 := ID_data1;
            data2 := ID_data2;
            signExtend := ID_signExtend;
            inst1 := ID_instruction1;
            inst2 := ID_instruction2;
        EX_WB <= WB;
        EX_M <= M;
        EX_EX <= EX;
        EX_PC <= PC;
        EX_data1 <= data1;
        EX_data2 <= data2;
        EX_signExtend <= signExtend;
        EX_instruction1 <= inst1;
        EX_instruction2 <= inst2;
    end process;
end Behavioral;


```

Code 20: Behavioral model for the ID/EX buffer register

```vhdl
----------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: EXMEM buffer
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- this buffer uses the same concept as the IF_ID buffer. Check it for more detailed commenting.
-- port definition. Takes control logic commands (WB, M, EX), the program counter, the results
-- from the ALU, the second register and the 5 bit chosen piece of instruction. 107 total bits.
entity EX_MEM_Buffer is
    Port ( EX_WB : in STD_LOGIC_VECTOR(1 downto 0);
           EX_M : in STD_LOGIC_VECTOR(2 downto 0);
           EX_PC : in STD_LOGIC_VECTOR (31 downto 0);
           EX_zero : in STD_LOGIC;
           EX_aluResult : in STD_LOGIC_VECTOR (31 downto 0);
           EX_data2 : in STD_LOGIC_VECTOR (31 downto 0);
           EX_chosenInstruction : in STD_LOGIC_VECTOR (4 downto 0);
           clk : in std_logic;
           MEM_WB : out STD_LOGIC_VECTOR(1 downto 0);
           MEM_M : out STD_LOGIC_VECTOR(2 downto 0);
           MEM_PC : out STD_LOGIC_VECTOR (31 downto 0);
           MEM_zero : out STD_LOGIC;
           MEM_aluResult : out STD_LOGIC_VECTOR (31 downto 0);
           MEM_data2 : out STD_LOGIC_VECTOR (31 downto 0);
           MEM_chosenInstruction : out STD_LOGIC_VECTOR (4 downto 0)
           );
end EX_MEM_Buffer;
architecture Behavioral of EX_MEM_Buffer is
begin
    process is
    variable zero : std_logic;
    variable M : STD_LOGIC_VECTOR(2 downto 0);
    variable WB : STD_LOGIC_VECTOR(1 downto 0);
    variable PC, aluRes, data2 : std_logic_vector(31 downto 0);
    variable chinst : std_logic_vector (4 downto 0);
    begin
        wait until rising_edge(clk);
            WB := EX_WB;
            M := EX_M;
            PC := EX_PC;
            zero := EX_zero;
            aluRes := EX_aluResult;
            data2 := EX_data2;
            chinst := EX_chosenInstruction;
        MEM_WB <= WB;
        MEM_M <= M;
        MEM_PC <= PC;
        MEM_zero <= zero;
        MEM_aluResult <= aluRes;
        MEM_data2 <= data2;
        MEM_chosenInstruction <= chinst;
     end process;
end Behavioral;
```

Code 21: Behavioral model for the EX/MEM buffer register

```vhdl
----------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: MEMWB buffer
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- this buffer uses the same concept as the IF_ID buffer. Check it for more detailed commenting.
-- port definition. Takes control logic commands (WB), the data read from the data memory, the
-- ALU result and the chosen piece of instruction. 71 total bits
entity MEM_WB_Buffer is
    Port ( MEM_WB : in std_logic_vector (1 downto 0);
           MEM_readData : in STD_LOGIC_vector (31 downto 0);
           MEM_aluResult : in STD_LOGIC_vector (31 downto 0);
           MEM_chosenInstruction : in std_logic_vector (4 downto 0);
           clk : in STD_LOGIC;
           WB_readData : out STD_LOGIC_vector (31 downto 0);
           WB_WB : out STD_LOGIC_vector (1 downto 0);
           WB_aluResult : out STD_LOGIC_vector (31 downto 0);
           WB_chosenInstruction : out std_logic_vector (4 downto 0)
           );
end MEM_WB_Buffer;
architecture Behavioral of MEM_WB_Buffer is
begin
    process is
    variable WB : std_logic_vector (1 downto 0);
    variable rdata, aluRes : std_logic_vector (31 downto 0);
    variable chinst : std_logic_vector (4 downto 0);
    begin
        wait until rising_edge(clk);
            WB := MEM_WB;
            rdata := MEM_readData;
            aluRes := MEM_aluResult;
            chinst := MEM_chosenInstruction;
        WB_WB <= WB;
        WB_readData <= rdata;
        WB_aluResult <= aluRes;
        WB_chosenInstruction <= chinst;
     end process;
end Behavioral;
```

Code 22: Behavioral model for the MEM/WB buffer register

```vhdl
----------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Behavioral model of the ALU controller
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- port description. Takes instruction piece (funct) and 2 bits from controller (op) and
-- outputs a 4 bit control for the ALU
entity ALUControl is
    port (instr : in std_logic_vector (5 downto 0);
          op : in std_logic_vector (1 downto 0);
          aluOp : out std_logic_vector(3 downto 0));
end ALUControl;
-- Behavioral model of the ALU controller
architecture Behavioral of ALUControl is
begin
    process (instr, op)
    begin
        if op = "10" then                                   -- if uses funct
            if instr = "100010" then aluOp <= "0110"; end if; -- SUB
            if instr = "100100" then aluOp <= "0000"; end if; -- AND
            if instr = "100000" then aluOp <= "0010"; end if; -- ADD
            if instr = "100101" then aluOp <= "0001"; end if; -- OR
            if instr = "100111" then aluOp <= "1100"; end if; -- NOR
            if instr = "101010" then aluOp <= "0111"; end if; -- SLT
        elsif op = "01" then aluOp <= "0010";               -- ADDI
        elsif op = "00" then aluOp <= "0010";               -- SW and LW
        else aluOp <= "0000"; end if;  -- if nothing matches, defaults to AND (arbitrary choice)
    end process;
end Behavioral;
```

Code 23: Behavioral model for the ALU control unit

```vhdl
----------------------------------------------------------------------------------
-- University: Illinois Institute of Technology
-- Engineers: Robert Soler and Gabriel Roskowski
-- Module Name: Behavioral model of the controller
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- port mapping that receives the instruction and outputs the control bits for writeback,
-- memory and execution management
entity Control is
    port (instr : in std_logic_vector (31 downto 0);
          WB : out std_logic_vector (1 downto 0);
          M : out std_logic_vector(2 downto 0);
          EX : out std_logic_vector (3 downto 0));
end Control;
-- behavioral model of the controller
architecture Behavioral of Control is
    signal opcode : std_logic_vector(5 downto 0);        -- temporary piece of instruction from bit 26 to 31 (op)
begin
    opcode <= instr(31 downto 26) after 20ps;            -- grabs op from instruction
    process(opcode, instr)
    begin
        if instr = x"00000000" then        -- nop
            WB <= "00";
            M <= "000";
            EX <= "0000";
        else
            if (opcode = "000000") then    -- add, sub, and, or, nor, slt,
                WB <= "11";
                M <= "000";
                EX <= "1010";
            elsif (opcode = "100011") then -- lw
                WB <= "01";
                M <= "010";
                EX <= "0101";
            elsif (opcode = "001000") then -- addi
                WB <= "11";
                M <= "000";
                EX <= "0101";
            elsif (opcode = "101011") then -- sw
                WB <= "00";
                M <= "100";
                EX <= "0101";
            elsif (opcode = "000010") then -- j
                WB <= "00";
                M <= "001";
                EX <= "0000";
            else                           -- anything else behaves like a NOP
                WB <= "00";
                M <= "000";
                EX <= "0000";
            end if;
        end if;
    end process;
end Behavioral;
```

Code 24: Structural model for the CPU's main control unit

Code 25: Structural model for the entire 32-bit CPU (continued to the next few pages)

```
1  ----------------------------------------------------------------------------------
2  -- University: Illinois Institute of Technology
3  -- Engineers: Robert Soler and Gabriel Roskowski
4  -- Module Name: Structural model of the processor, putting all the subcomoponents together
5  ----------------------------------------------------------------------------------
6
7  library IEEE;
8  use IEEE.STD_LOGIC_1164.ALL;
9  -- port mapping. Will only input clk. A real processor should have other IOs to interact with external components
10 entity RISC is
11     port(clk : in STD_LOGIC);
12 end RISC;
13 -- declaration of several components that are used in the processor
14 architecture Structural of RISC is
15     component DataMemory is
16         port(clk, write, read: in std_logic;
17             addr, writeData: in std_logic_vector(31 downto 0);
18             readData: out std_logic_vector(31 downto 0));
19     end component;
20     component ALU32 is
21     port(a, b: in std_logic_vector(0 to 31);
22          op: in std_logic_vector(0 to 1);
23          an, bn: in std_logic;
24          q: out std_logic_vector(0 to 31);
25          ov, zero: out std_logic);
26     end component;
27     component Adder is
28     port(a, b: in std_logic_vector(31 downto 0);
29         cin: in std_logic;
30         q: out std_logic_vector(31 downto 0);
31         cout: out std_logic);
32     end component;
33     component ALUControl is
34     port (instr : in std_logic_vector (5 downto 0);
35          op : in std_logic_vector (1 downto 0);
36          aluOp : out std_logic_vector(3 downto 0));
37     end component;
38     component Control is
39     port (instr : in std_logic_vector (31 downto 0);
40          WB : out std_logic_vector (1 downto 0);
41          M : out std_logic_vector(2 downto 0);
42          EX : out std_logic_vector (3 downto 0));
43     end component;
44     component EX_MEM_Buffer is
45     Port ( EX_WB : in STD_LOGIC_VECTOR(1 downto 0);
46            EX_M : in STD_LOGIC_VECTOR(2 downto 0);
47            EX_PC : in STD_LOGIC_VECTOR (31 downto 0);
48            EX_zero : in STD_LOGIC;
49            EX_aluResult : in STD_LOGIC_VECTOR (31 downto 0);
50            EX_data2 : in STD_LOGIC_VECTOR (31 downto 0);
51            EX_chosenInstruction : in STD_LOGIC_VECTOR (4 downto 0);
52            clk : in std_logic;
53            MEM_WB : out STD_LOGIC_VECTOR(1 downto 0);
54            MEM_M : out STD_LOGIC_VECTOR(2 downto 0);
55            MEM_PC : out STD_LOGIC_VECTOR (31 downto 0);
56            MEM_zero : out STD_LOGIC;
57            MEM_aluResult : out STD_LOGIC_VECTOR (31 downto 0);
58            MEM_data2 : out STD_LOGIC_VECTOR (31 downto 0);
59            MEM_chosenInstruction : out STD_LOGIC_VECTOR (4 downto 0)
60            );
61     end component;
62     component ID_EX_Buffer is
63     Port (ID_WB : in STD_LOGIC_VECTOR(1 downto 0);
64          ID_M : in STD_LOGIC_VECTOR(2 downto 0);
65          ID_EX : in std_logic_vector (3 downto 0);
66          ID_PC : in std_logic_vector (31 downto 0);
67          ID_data1 : in std_logic_vector (31 downto 0);
68          ID_data2 : in std_logic_vector (31 downto 0);
69          ID_signExtend : in std_logic_vector (31 downto 0);
```

```
        ID_instruction1 : in std_logic_vector (4 downto 0);
        ID_instruction2 : in std_logic_vector (4 downto 0);
        clk : in std_logic;
        EX_WB : out STD_LOGIC_VECTOR(1 downto 0);
        EX_M : out STD_LOGIC_VECTOR(2 downto 0);
        EX_EX : out std_logic_vector (3 downto 0);
        EX_PC : out std_logic_vector (31 downto 0);
        EX_data1 : out std_logic_vector (31 downto 0);
        EX_data2 : out std_logic_vector (31 downto 0);
        EX_signExtend : out std_logic_vector (31 downto 0);
        EX_instruction1 : out std_logic_vector (4 downto 0);
        EX_instruction2 : out std_logic_vector (4 downto 0)
        );
    end component;
    component IF_ID_Buffer is
    Port ( IF_Instruction : in STD_LOGIC_VECTOR (31 downto 0);
           IF_PC : in STD_LOGIC_VECTOR (31 downto 0);
           clk : in STD_LOGIC;
           ID_Instruction : out STD_LOGIC_VECTOR (31 downto 0);
           ID_PC : out STD_LOGIC_VECTOR (31 downto 0)
           );
    end component;
    component InstrMEM is
    port (pc : in std_logic_vector (31 downto 0);
          instr : out std_logic_vector (31 downto 0));
    end component;
    component MEM_WB_Buffer is
    Port ( MEM_WB : in std_logic_vector (1 downto 0);
           MEM_readData : in STD_LOGIC_vector (31 downto 0);
           MEM_aluResult : in STD_LOGIC_vector (31 downto 0);
           MEM_chosenInstruction : in std_logic_vector (4 downto 0);
           clk : in STD_LOGIC;
           WB_readData : out STD_LOGIC_vector (31 downto 0);
           WB_WB : out STD_LOGIC_vector (1 downto 0);
           WB_aluResult : out STD_LOGIC_vector (31 downto 0);
           WB_chosenInstruction : out std_logic_vector (4 downto 0)
           );
    end component;
    component PC is
    port (pcin : in std_logic_vector (31 downto 0);
          clk : in std_logic;
          pcout : out std_logic_vector (31 downto 0));
    end component;
    component RegFile is
    port (readReg1 : in std_logic_vector (4 downto 0);
          readReg2 : in std_logic_vector (4 downto 0);
          writeReg : in std_logic_vector (4 downto 0);
          writeData : in std_logic_vector (31 downto 0);
          regWrite, clk : in std_logic;
          readData1 : out std_logic_vector (31 downto 0);
          readData2 : out std_logic_vector (31 downto 0)
          );
    end component;
    component SignExtender is
    port(instr : in std_logic_vector(15 downto 0);
        extended : out std_logic_vector(31 downto 0));
    end component;
    component MUX2_32 is
    Port ( i0 : in STD_LOGIC_VECTOR(31 downto 0);
           i1 : in STD_LOGIC_VECTOR(31 downto 0);
           s : in STD_LOGIC;
           z : out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component MUX2_5 is
    Port ( i0 : in STD_LOGIC_VECTOR(4 downto 0);
           i1 : in STD_LOGIC_VECTOR(4 downto 0);
           s : in STD_LOGIC;
           z : out STD_LOGIC_VECTOR(4 downto 0));
    end component;
```

```
139
140     -- the many signals that are used across the processor to connect components
141     signal pc_curr, new_pc0, new_pc1, new_pc2, new_pc3, new_pc4, muxpc, ext : std_logic_vector(31 downto 0);
142     signal instr_0, instr_1 : std_logic_vector(31 downto 0);
143     signal instr_0A, instr_1A, instr_2, instr_3, resmux : std_logic_vector(4 downto 0);
144     signal WB_0, WB_1, WB_2, WB_3 : std_logic_vector(1 downto 0);
145     signal M_0, M_1, M_2 : std_logic_vector(2 downto 0);
146     signal EX_0, EX_1 : std_logic_vector(3 downto 0);
147     signal D1_0, D1_1, D2_0, D2_1, D2_2 : std_logic_vector(31 downto 0);
148     signal SE_0, SE_1 : std_logic_vector(31 downto 0);
149     signal ALU_C : std_logic_vector(3 downto 0);
150     signal Z_0, Z_1 : std_logic;
151     signal ALU_IN, RES_0, RES_1, RES_2 : std_logic_vector(31 downto 0);
152     signal DR_0, DR_1, WD : std_logic_vector(31 downto 0);
153     -- execution of the processor
154 begin
155     -- first stage
156     ext <= "0000" & instr_1(25 downto 0) & "00";                -- shifting left the jump by 2 places
157     mux_0 : MUX2_32 port map(new_pc0, ext, M_0(0), muxpc);      -- choosing between jump target and PC + 4
158     pc_register: PC port map(muxpc, clk, pc_curr);              -- the pc register
159     pc_adder : Adder port map(X"00000004", pc_curr, '0', new_pc0, open);    -- the adder for the pc register (PC+4)
160     instruction_memory : InstrMEM port map(pc_curr, instr_0);               -- the instruction memory
161     first_buffer : IF_ID_Buffer port map(instr_0, new_pc0, clk, instr_1, new_pc2);  -- the first buffer (first stage)
162
163     -- second stage
164     register_file : RegFile port map(instr_1(25 downto 21), instr_1(20 downto 16),
165                 instr_3, WD, WB_3(0), clk, D1_0, D2_0);                                 -- register file
166     control_circuit : Control port map(instr_1, WB_0, M_0, EX_0);                       -- decoding of the instruction
167     sign_extender : SignExtender port map(instr_1(15 downto 0), SE_0);                  -- sign-extends immediate value
168     second_buffer : ID_EX_Buffer port map(WB_0, M_0, EX_0, new_pc2, D1_0, D2_0, SE_0, instr_1(20 downto 16), -- second buffer
169                 instr_1(15 downto 11), clk, WB_1, M_1, EX_1, new_pc3, D1_1, D2_1, SE_1, instr_0A, instr_1A);
170
171     -- third stage
172     mux_1 : MUX2_32 port map(SE_1, D2_1, EX_1(1), ALU_IN);                  -- mux that chooses what to give to the ALU
173     mux_2 : MUX2_5 port map(instr_1A, instr_0A, EX_1(0), resmux);           -- mux that chooses the instruction portion as address for writeback
174     alu_control : ALUControl port map(SE_1(5 downto 0), EX_1(3 downto 2), ALU_C);   -- controller of the ALU (grabs funct and 2 bits from controller)
175     alu : ALU32 port map(D1_1, ALU_IN, ALU_C(1 downto 0), ALU_C(3), ALU_C(2), RES_0, Z_0);  -- the ALU (implements AND, OR, NOR, SLT, SUB, ADD)
176     -- we can still implement add for PC for branching and SL2
177     third_buffer : EX_MEM_Buffer port map(WB_1, M_1, X"00000000", Z_0, RES_0, D2_1, resmux, clk, WB_2, M_2,   -- third buffer (branch unimplemented)
178                 open, Z_1, RES_1, D2_2, instr_2);
179
180     -- forth stage
181     -- the memory of the processor that stores the data (interacts with sw and lw)
182     data_memory : DataMemory port map(clk, M_2(2), M_2(1), RES_1, D2_2, DR_0);
183     -- there is also the and gate for the branch that could be implemented
184     forth_buffer : MEM_WB_Buffer port map(WB_2, DR_0, RES_1, instr_2, clk, DR_1, WB_3, RES_2, instr_3);          -- forth buffer
185
186     -- fifth stage
187     mux_3 : MUX2_32 port map(DR_1, RES_2, WB_3(1), WD);         -- mux that chooses data from the memory or the ALU, depending on the instruction
188 end Structural;
```

II. Testbench Codes

Code 26: Testbench for the Data Memory (continued to the next page)

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity TestbenchDataMem is
4  end TestbenchDataMem;
5
6  architecture Behavioral of TestbenchDataMem is
7      component DataMemory is
8          port(clk, write, read: in std_logic;
9          addr, writeData: in std_logic_vector(31 downto 0);
10         readData: out std_logic_vector(31 downto 0));
11     end component;
12     signal clk, write, read: std_logic;
13     signal addr, writeData: std_logic_vector(31 downto 0);
14     signal readData: std_logic_vector(31 downto 0);
```

```
15 begin
16     mem: DataMemory port map(clk, write, read, addr, writeData, readData);
17     process is
18     begin
19         clk <= '0';
20         write <= '0';
21         read <= '0';
22         addr <= X"00000000";
23         writeData <= X"00000000";
24         wait for 1ps;
25 
26         write <= '1';
27         writeData <= X"11111111";
28         clk <= '1';
29 
30         wait for 1ps;
31 
32         clk <= '0';
33 
34         wait for 1ps;
35 
36         writeData <= X"22222222";
37         addr <= X"00000001";
38         clk <= '1';
39 
40         wait for 1ps;
41 
42         write <= '0';
43         clk <= '0';
44         writeData <= X"00000000";
45 
46         wait for 1ps;
47 
48         read <= '1';
49         clk <= '1';
50 
51         wait for 1ps;
52 
53         addr <= X"00000000";
54         clk <= '0';
55 
56         wait for 1ps;
57 
58         clk <= '1';
59 
60         wait for 1ps;
61 
62     end process;
63 end Behavioral;
```

(Code 26 ends here)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TestbenchALU1 is
--  Port ( );
end TestbenchALU1;

architecture Behavioral of TestbenchALU1 is
    component ALU1 is
        port(a, b: in std_logic;
         op: in std_logic_vector(0 to 1);
         an, bn, cin, less: in std_logic;
         o: out std_logic;
         cout: out std_logic);
    end component;
    signal a, b: std_logic;
    signal op: std_logic_vector(0 to 1);
    signal an, bn, cin, less: std_logic;
    signal o, cout: std_logic;
begin
    --note: overflow doesn't happen sometimes. needs further testing
    alu: ALU1 port map(a, b, op, an, bn, cin, less, o, cout);
    sim: process is
        begin
            an <= '0';
            bn <= '1';
            op <= "11";
            cin <= '0';
            less <= '0';
            a <= '1';
            b <= '1';
            wait for 1ps;
            a <= '0';
            b <= '1';
            wait for 1ps;
            less <= '1';
            a <= '1';
            b <= '0';
            wait for 1ps;
            less <= '0';
            cin <= '1';
            wait for 1ps;
            less <= '1';
            wait for 1ps;
            a <= '0';
            wait for 1ps;
            less <= '0';
            cin <= '1';
            wait for 1ps;
            less <= '1';
        end process;
end Behavioral;
```

Code 27: Testbench for the 1-bit ALU

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity TestbenchALU is
--  Port ( );
end TestbenchALU;

architecture Behavioral of TestbenchALU is
    component ALU32 is
        port(a, b: in std_logic_vector(0 to 31);
             op: in std_logic_vector(0 to 1);
             an, bn: in std_logic;
             q: out std_logic_vector(0 to 31);
             ov, zero: out std_logic);
    end component;
    signal a, b, q: std_logic_vector(0 to 31);
    signal op: std_logic_vector(0 to 1);
    signal an, bn, ov, zero: std_logic;
begin
    --note: overflow doesn't happen sometimes. needs further testing
    alu: ALU32 port map(a, b, op, an, bn, q, ov, zero);
    sim: process is
        begin
            report "testing addition ";
            an <= '0';
            bn <= '1';
            op <= "11";

            a <= x"00000140";
            b <= x"00000004";
            wait for 4ps;
            a <= x"00000130";
            b <= x"00000120";
            wait for 4ps;
            a <= "11111111111111111111111111111000";
            b <= "11111111111111111111111111111100";
            wait for 2ps;
            a <= "11111111111111111111111111111100";
            b <= "11111111111111111111111111111000";
            wait for 2ps;

            a <= "00000000000000000000000000000000";
            b <= "11000000000000000000000000001111";
            wait for 2ps;
            a <= "10000000000000000000000000000011";
            b <= "10000000000000000000000000000111";
            wait for 2ps;
            a <= "00000000000000000000000000000000";
            b <= "00000000000000000000000000000000";
            wait for 2ps;
            a <= "11111111111111111111111111111111";
            b <= "00000000000000000000000000000001";
            wait for 2ps;
            a <= "00000000000000000000000000001111";
            b <= "00000000000000000000000000000101";
            wait for 2ps;
            a <= "01111111111111111111111111111111";
            b <= "10111111111111111111111111111111";
            wait for 2ps;
            report "added numbers";
        end process;
end Behavioral;
```

Code 28: Testbench for the 32-bit ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RISC_SIMULATION is
--  Port ( );
end RISC_SIMULATION;

architecture Behavioral of RISC_SIMULATION is
     component RISC is
        port(clk: in std_logic);
    end component;
    signal clk : std_logic;
begin
    proc: RISC port map(clk);
    process is
    begin
        for i in 1 to 10 loop
            clk <= '0';
            wait for 200ps;
            clk <= '1';
            wait for 200ps;
        end loop;
    end process;

end Behavioral;
```

Code 29: Testbench for the 32-bit RISC processor

III. The assembler

Code 30: Java code for the Assembler (continued to the next few pages)

```
import java.io.BufferedReader;

public class Test2 {
    static String regs[] = { "zero", "at", "v0", "v1", "a0", "a1", "a2", "a3", "t0", "t1", "t2", "t3", "t4", "t5", "t6",
            "t7", "s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "t8", "t9", "k0", "k1", "gp", "sp", "fp", "ra" };
    static String instrs[] = { "SUB", "AND", "OR", "ADD1", "LW", "ADDI", "SW", "NOR", "SLT", "J" };
    static String opcode[] = { "000000", "000000", "000000", "000000", "100011", "001000", "101011", "000000", "000000",
            "000010" };
    static String suffix[] = { "00000100010", "00000100100", "00000100101", "00000100000", "", "", "", "00000100111",
            "00000101010", "" };

    public static void main(String[] args) {
        String code = loadCode("code.asm");
        String instructions[] = code.split("\n");
        boolean NOPsafety = true;
        boolean typeReady = true;
        int a = 0;
        List<String> register = new ArrayList<>();
        for (int i = 0; i < instructions.length; i++) {
            String instructionRaw = instructions[i];
            String instruction = instructions[i].replace(",", "").replace("$", "").replace(")", "");
            String segs[] = instruction.split("\\s+"), c[] = null;
            String rs = null, rt = null, rd = null;
            if (segs.length > 1)
                rd = registerToBinary(segs[1]);
            if (segs.length > 2) {
                rs = registerToBinary(segs[2]);
                c = segs[2].replace("(", ",").split(",");
            }
            if (segs.length > 3)
                rt = registerToBinary(segs[3]);
```

```
            if (register.size() >= 4)
                register.remove(0);
            if (rs != null && rt != null && NOPsafety)
                if (register.contains(rs) || (c.length > 1 && register.contains(c[1])) || register.contains(rt)) {
                    if (typeReady)
                        System.out.println("reg(" + (a++) + ") := x\"00000000\";");
                    else
                        System.out
                                .println(Long.toString(0x100000000L | (a++), 16).substring(1) + "    00000000    NOP");
                    register.add("--------");
                    i--;
                    continue;
                }
            int instrIndex = indexOf(segs[0], instrs);
            String t = opcode[instrIndex];
            if (instrIndex < 4 || instrIndex == 7 || instrIndex == 8) {
                t += rs + rt + rd + suffix[instrIndex];
                register.add(rd);
            }
            if (instrIndex == 9)
                t += toBinaryFixed(Long.parseLong(segs[1]) >> 2, 26);
            if (instrIndex == 4 || instrIndex == 6) {
                t += registerToBinary(c[1]) + rd + toBinaryFixed(Long.parseLong(c[0]), 16);
                register.add(rd);
            }
            if (instrIndex == 5) {
                t += rs + rd + toBinaryFixed(Long.parseLong(segs[3]), 16);
                register.add(rd);
            }
            if (t != null) {
                if (typeReady)
                    System.out.println("reg(" + (a++) + ") := x\""
                            + Long.toString(0x100000000L | Long.parseLong(t, 2), 16).substring(1).toUpperCase()
                            + "\";");
                else
                    System.out.println(Long.toString(0x100000000L | (a++) * 4, 16).substring(1).toUpperCase() + "    "
                            + Long.toString(0x100000000L | Long.parseLong(t, 2), 16).substring(1).toUpperCase() + "    "
                            + instructionRaw);
            }
        }
    }

    private static String registerToBinary(String reg) {
        return toBinaryFixed(indexOf(reg, regs), 5);
    }

    private static String toBinaryFixed(long index, int len) {
        return Long.toBinaryString((long) (Math.pow(2, len)) | index).substring(1);
    }

    private static int indexOf(String c, String[] regs) {
        for (int i = 0; i < regs.length; i++)
            if (regs[i].equals(c))
                return i;
        return -1;
    }

    private static String loadCode(String path) {
        StringJoiner c = new StringJoiner("\n");
        try {
            BufferedReader br = new BufferedReader(new FileReader(new File(path)));
            String temp = null;
            while ((temp = br.readLine()) != null)
                c.add(temp);
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return c.toString();
    }
}
```