ENGIN 341 – Advanced Digital Design

Engineering Department

University of Massachusetts Boston

Semester – Spring 2021

Instructor – Dr. Michael Rahaim

Lab Report for

Lab 5: RISC Architecture

By

Name: Tyler McKean, Vulindsky Fanfan, Zachary Garnes

Student ID: 01098154, 01538113, 01353567

Date: May 14, 2021

I pledge to uphold the governing principles of the Code of Student Conduct of the University of Massachusetts Boston. I will refrain from any form of academic dishonesty or deception, cheating, and plagiarism. I pledge that all the work submitted here is my own, and that I have clearly acknowledged and referenced other people's work. I am aware that it is my responsibility to turn in other students who have committed an act of academic dishonesty; and if I do not, then I am in violation of the Code. I will report to formal proceedings if summoned.

Signed:

Tyler McKean, Vulindsky Fanfan, Zachary Garnes

CONTENTS

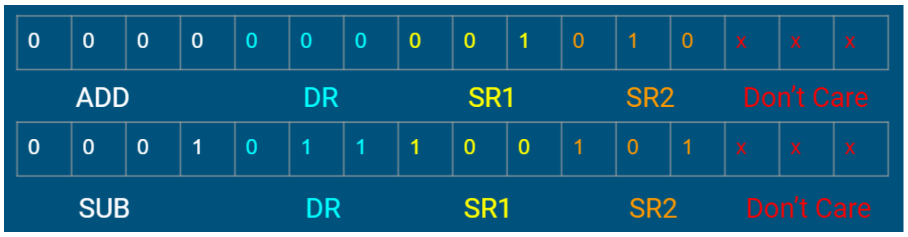
OVERVIEW

For the Final Project, our group was tasked with designing a RISC architecture that included 16-bit instructions, 16-bit words and registers, and 8-bit memory addresses for the Program Memory and Data Memory that were both byte addressable when accessing them. The instructions included four different formats for the opcode, which were: ALU functions, ALU Immediate functions, Memory functions, and Control functions. The ALU functions would perform operations such as Add, Subtract, AND, OR, Shift Logical Left, and Shift Logical Right with 16-bit values that were temporarily stored in two of the eight registers and the results would be saved to a destination register. For the ALU Immediate functions, the value within one selected source register and a 6-bit immediate value, which would be sign extended to 16 bits, would both be performed with Add or an AND operation in the ALU. Again, the results for these ALU Immediate operations would be written back to a destination register. When performing the Memory functions, the functions applicable to our architecture were Load and Store operations. When the Load operation was called in the instruction bits, a given 8-bit memory address within the instructions would be used to access said memory address in the Data Memory and extract those values to be stored into a destination register. When the Store function was used, a value within the specified source register would be put into a given 8-bit memory address in the Data Memory. For the Control operations, we were asked to incorporate a Jump, Set on Less than, and Branch on Equal function. If the Jump operation was called, an 8-bit value would be added or subtracted to the PC+2 signal, which would allow us to either jump back to a previous instruction in the Program Memory or forward. For Set on Less than, two source registers would be evaluated in the ALU by subtracting SR1 by SR2. If the results of the ALU subtraction are less than zero, then a destination register would be stored with the value of 1. Otherwise, if the result of the subtraction is non-zero, then a value of 0 would be stored into the specified register. The final operation of our RISC architecture was the Branch on Equal function. This instruction would evaluate two source registers by subtracting them in the ALU and if the source registers are equal in value, then the results would be zero. If this is the case, then a given 8-bit PC address in the instruction would branch the PC count to that specified location in the Program Memory. Once the RISC architecture for our design was built, our team was asked to translate two different programs into their equivalent machine code, write the instructions into our program memory, and then simulate our design to execute the two programs. The following sections outline the design process and simulation results.

DESIGN DESCRIPTION

Instruction bits breakdown

Since there are four different formats for our RISC architecture, we created bit assignments for the ALU, ALU Immediate, Memory, and Control functions. The bit assignments for the ALU operations can be seen in Fig 1 below.

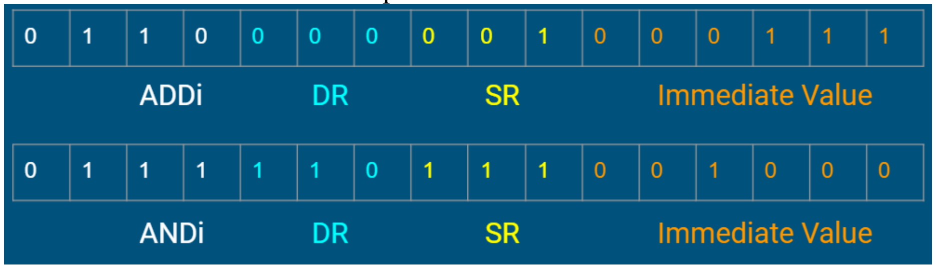


0	0	0	0	0	0	0	0	0	1	0	1	0	x	x	x
ADD				DR			SR1			SR2			Don't Care		
0	0	0	1	0	1	1	1	0	0	1	0	1	x	x	x
SUB				DR			SR1			SR2			Don't Care		

Figure 1 – Bit assignments for ALU operations.

For many of our functions, bits 15 down to 12 of the instructions were assigned as the opcode bits. Since there were 13 functions our architecture needed to do be able to execute, we decided to use the four most significant bits of every instruction to indicate what the modules of our design would be performing. Figure 1 above shows the bit assignments for the ALU and displays the opcode bits for the Add (0000) and Subtract (0001) functions. The remaining functions: AND (0010), OR (0011), Shift

Logical Left (0100), and Shift Logical Right (0101) also followed the same bit assignments listed above. Bits 9 to 11 were preserved for the 3-bit address of the Destination Register, bits 6 to 8 represented the address for Source Register 1, bits 3 to 5 represented the address for Source Register 2 and the remainder were considered as don't cares for the ALU operations.



0	1	1	0	0	0	0	0	0	1	0	0	0	1	1	1

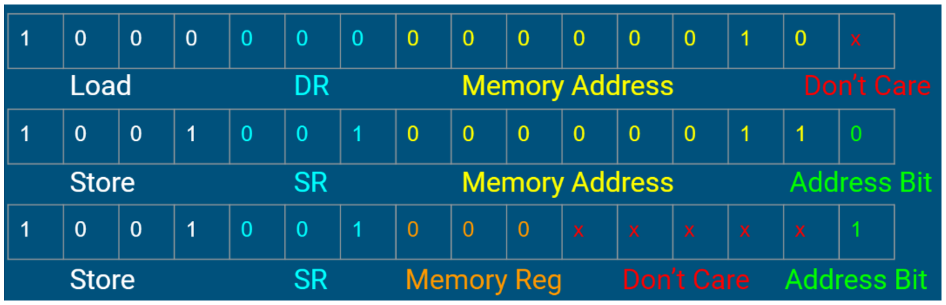
ADDi DR SR Immediate Value

0	1	1	1	1	1	0	1	1	1	0	0	1	0	0	0

ANDi DR SR Immediate Value

Figure 2 – Bit assignments for ALU Immediate operations ADDi and ANDi.

Figure 2 above shows the bit assignments for our ALU Immediate operations, which included functions ADDi (0110) and ANDi (0111). The ALU Immediate bit assignments follows a similar format to the ALU functions with the exception that there is only one Source Register, bits 6 to 8 and a 6-bit Immediate value that were designated for bits 0 to 5. As mentioned previously, the 6-bit Immediate value would need to be extended to a 16-bit value before becoming an input to the ALU module in our design but using the 6 bits in our instruction allowed our design to perform an Add or AND function with a value of 2^6 or 64.



1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	x

Load DR Memory Address Don't Care

1	0	0	1	0	0	1	0	0	0	0	0	0	1	1	0

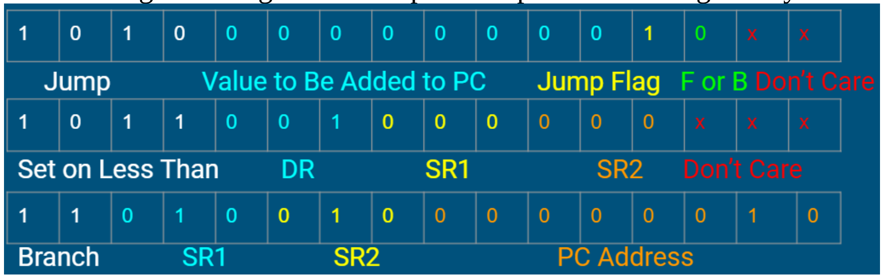
Store SR Memory Address Address Bit

1	0	0	1	0	0	1	0	0	0	x	x	x	x	x	1

Store SR Memory Reg Don't Care Address Bit

Figure 3 – Bit assignments for Memory operations Load and Store.

The bit assignments for our design's Memory functions are depicted above in Figure 3 and showcases the Load (1000) and Store (1001) functions bits. Again, bits 12 to 15 are serving as our opcode bits to determine either a Load or Store function is called, bits 9 to 11 are reserved for the register addresses that a value would either be loaded into or stored into the Data Memory, and bits 1 to 8 are the 8-bit memory addresses used to specify the location in Data Memory we are addressing. We incorporated another version of the Store function that would be necessary when executing the 2nd program we were tasked with implementing. This altered version of the Store function still utilizes the same opcode assignment of 1001 but has a Source Register at bits 9 to 11, a Memory Register at bits 6 to 8, and an Address Bit at bit 0 of the instruction. This instruction would allow our design to store a value in the Source Register using the value in the Memory Register, which would serve as a memory address. The Address Bit would represent a flag that would allow the Data Memory to use the Memory Reg value as an addressable byte value. Our reasons for using this design will be expanded upon in the Design Entry section.



1	0	1	0	0	0	0	0	0	0	0	0	1	0	x	x

Jump Value to Be Added to PC Jump Flag F or B Don't Care

1	0	1	1	0	0	1	0	0	0	0	0	0	x	x	x

Set on Less Than DR SR1 SR2 Don't Care

1	1	0	1	0	0	1	0	0	0	0	0	0	0	1	0

Branch SR1 SR2 PC Address

Figure 4 – Bit assignments for Control operations Jump, Set on Less Than, and Branch on Equal.

The last three functions of our RISC architecture were the Control functions, which included a Jump (1010), Set on Less Than (1011), and Branch on Equal (11) function. Their bit assignments are shown above in Figure 4. The Jump function included an 8-bit value that would be used to increment or decrement the current PC count and was assigned as bits 4 to 11. A Jump Flag was assigned to bit 3 and served as one of the select bits in our control module multiplexer. If the Jump Flag was set high then the multiplexer would allow the altered PC count to pass through the multiplexer and the next PC count would be affected by the 8-bit value in the instruction. Another flag in this instruction was assigned to bit 2 and it would determine whether the Jump Magnitude was meant to be added or subtracted to the current PC count. The "F or B" in the figure refers to forward, meaning addition, or backward, referencing subtraction. The Set on Less Than function included a Destination Register address for bits 9 to 11, Source Register 1 address for bits 6 to 8, and Source Register 2 address for bits 3 to 5. This operation would subtract the value of Source Register 1 from the value of Source Register 2, and if the result was less than zero, than the Destination Register would be assigned a value of 1. Otherwise, if the subtraction result was non-zero, the Destination Register would be set to a value of 0. For the Branch on Equal function there was not enough bits to assign a full 4-bit opcode, but since all the previous opcode values ranged from 0000 to 1011, we could use an opcode value of 11 for bits 14 and 15 of the instruction set. Two Source Registers were assigned to bits 11 to 13 and bits 8 to 10, and the instruction included an 8-bit PC Address for bits 0 to 7. This operation would subtract the value of Source Register 1 from Source Register 2 in the ALU, and if these registers are equal in value, then the ALU result would be zero.
If these zero results occur during this operation, then the PC count would branch to the specified 8-bit PC address in the Program Memory and resume the instruction fetch and decode operations from that new PC address.

Data Path and Modules

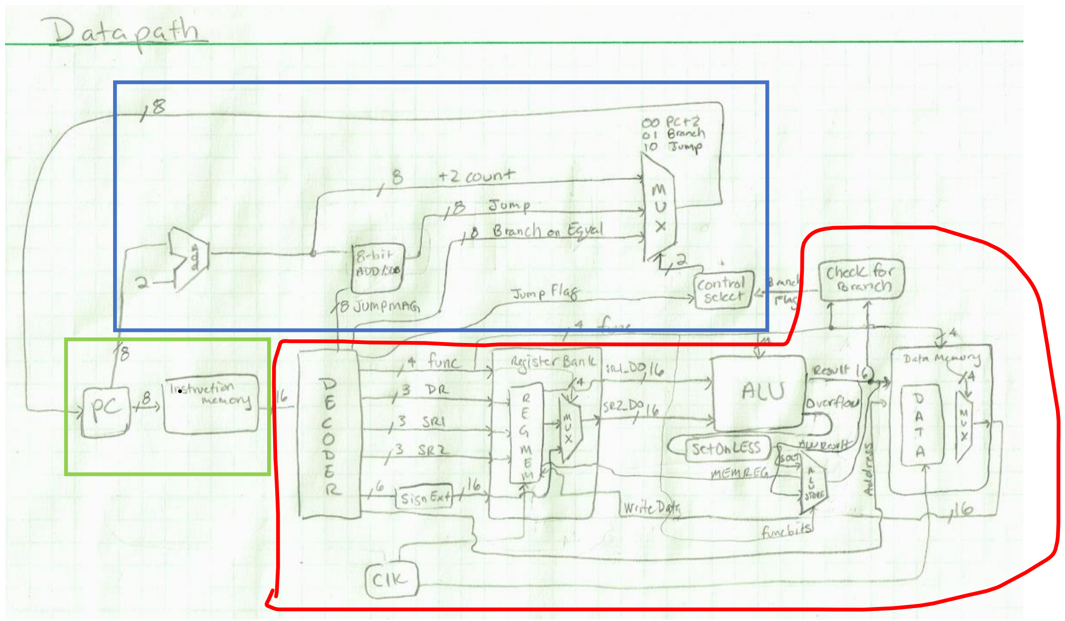


Figure 5 – Overview of Datapath and main modules of RISC architecture.

In the figure above you will see the Datapath of the top-level program of the whole project. We divide the entire program into three main modules that you can see boxed off in the figure. The red block is the decode and execute module. It takes in the 16-bit instruction, decodes the bits based on the function that is needed to be performed, executes an operation with the use of an ALU if the function calls for it, loads or stores data from memory to registers and vice versa, and outputs flags for the control block. The control block, blue box, updates the program counter (PC), based on the flags that are set by the decoder and the ALU, the control block will either update by default or based off the jump/branch operation. Finally, the fetch block, green box, which get the instruction that is stored at the memory address of the current PC which is stored in the instruction memory.

Instruction Fetch Block

The instruction fetch unit consists of the instruction/program memory as well as the program counter. To start processing, the CPU needs to fetch the first instruction in the program from the main memory, that being the instruction memory which is 256x8 instruction memory containing 16-bit instructions. The program counter always holds the address of the next instruction, which is an 8-bit address corresponding to a specific instruction in the program memory. The program counter tells the CPU in what order the instruction should be carried out and executed. If a program in the main memory needs to be executed sequentially, the program counter gets incremented by two on every cycle to get to the next instruction. If the flow of execution is changed by a jump flag or a branch flag, the program counter gets incremented accordingly. Thus the next stage fetches the new instruction from the instruction memory according to the modified PC value. Once the instruction is fetched from the instruction memory, it gets sent to the decoder module and eventually gets sent to the execution module to be executed.

Decoder Block

The decoder block's main priority is to translate the 16-bit instructions that enter its input and decipher the function and appropriate actions the instruction execution block and control blocks must take. A VHDL process, which is dependent on the input instruction would take the 16-bit unsigned instruction and chose one of fourteen if statements that will assign the appropriate outputs to the same bit assignments laid out in the previous Instruction Bits breakdown section. The main conditional statement for each if statement is dependent on the four most significant bits that indicate which operation the instruction was written to perform. Since there are thirteen operations in total that are included for the ALU, ALU Immediate, Memory, and Control operations, the function bits for each operation are included in this VHDL process, with the additional STORE operation we added to our design to be able to use a register value to access the Data Memory. The outputs for the Decode block consists of a 3-bit addresses for the DR, SR1, SR2, the 6-bit Immediate value, an 8-bit Memory address for Memory functions, the 8-bit Jump Offset, Jump Flag, and FoB Flag if the value is adding or subtracting to the current PC+2 count, and the Address Flag for our additional Store operation. Each output was assigned to the same bit ranges discussed previously. If an output was not being used for a particular operation the output was given an assignment of don't care.

Instruction Execution Block

The Instruction Execution Block consists of the Decoder module, Register Memory Module, ALU Module, Data Memory Module, Set on Less Than Module, ALU-Store MUX, Check for Branch Module, and the Sign Extension Module. The main purpose of this module is to interpret the 16-bit instructions and perform the appropriate operations specified by the function bits. If an ALU or ALU Immediate operation is decoded from the instructions, then the Register Memory Module will output the specified register values from the decoded register addresses in the instructions. For the ALU Immediate operations, the 6-bit Immediate value in the instruction would first go into the Sign Extension module to convert the value to a 16-bit value. Source Register 1 would become the first input to the ALU, and the second input would be either Source Register 2 or the Immediate value. A Register Memory MUX was implemented into the Register Memory module that checks the func bits for whether they are an ALU or ALU Immediate operation. If the func bits call for just an ALU operation, the Source Register 2 value would become the second input to the ALU. Otherwise, if the func bits specify an ALU Immediate operation, then the 16-bit Immediate value would become the second input to the ALU. The ALU then takes the func bits as an input to select which operation is being performed and the internal MUX would allow the result of that operation to become the output. The ALU result and the 16-bit value that is being stored to the Data Memory during a Store operation both go into another MUX called the ALU-Store MUX. Depending on the func bits, when an ALU or ALU Immediate operation is performed, the ALU result would become the output of the MUX, otherwise, when a Store function is performed, the 16-bit value would output to the Data Input of the Data Memory. If we are storing a value to the Data Memory, the Data Memory Module reads from the 8-bit Address specified in the instructions and saves the DI input to that Address in the memory. The ALU result avoids the Data Memory and instead goes into a Memory MUX that is inside the Data Memory. When an ALU operation is performed, the output of this MUX would send the ALU result back to the Register Memory to be stored in the Destination Register. If a Load operation is performed the output of the Data Memory will be the 16-bit value stored at the specified 8-bit Address in the instruction, which will be saved to the Destination Register in the Register Memory. During a Set on Less Than operation, two Source Registers are subtracted in the ALU and the overflow output of the ALU is used to set the Destination Register value to 1 if the subtraction result was negative, otherwise, it is set to 0 for a non-negative result. A Check for Branch module was also added to check when an ALU subtraction is performed and if the two Source Registers are equal in value, then the ALU result will be zero. This module verifies that the result is zero and would output the 8-bit PC Address that the PC count would branch too in the Instruction Memory. The Register Memory and Data Memory modules were written such that they would execute on the falling edge of the clock. The Instruction Execution Module contains all the appropriate outputs for the Control operations as well. If a Jump operation is decoded from the instructions, then the 8-bit Jump Magnitude is outputted from the Instruction Execution block along with a Jump Flag that will serve as the select bits for a Control MUX. A Forward or Backward Flag is also sent out from the Instruction Execution block to specify whether the Jump Magnitude will be added or subtracted from the current PC+2 count. During a Branch instruction, a Branch Flag will be sent out to the Control module to serve as a select bit for the Control MUX, and the 8-bit PC Address that the Instruction Memory will branch to is written as an output as well.

Control Block

The main function of the control block is to update the PC properly. There are only two operations on which the PC does not update by the default values of 2, which is with a jump or branch operation. First, the default values if set a 2 because the instruction memory is byte accessible meaning that each memory address is 8-bits long. However, the instructions are 16-bit words so two memory address hold one instruction that is why the next instruction location jumps every two memory addresses. The current PC is an input which is then put into an *adder* module with the number 2 in order to get the next default PC. To handle the jump operations, we have 3 inputs: jump flag, back jump flag, and the jump magnitude. The jump flag indicates that jump function was the operation in the instruction. The jump back flag indicates if it is a jump back, which means the magnitude with be subtracted from the current PC instead of added. The jump magnitude is the 8-bit value in the instruction which is the amount the PC is going to jump by. To handle the final operation, branch, there are two inputs: branch flag and branch address. The branch flag is set by the output of the ALU, which will be 1 if source register one is the same value as source register two which indicates that the branch will be performed. The branch address is 8-bit memory address that the PC will branch to.

DESIGN ENTRY

Register Memory

The Register Memory was designed with all the appropriate inputs that would be deciphered from the decode block and output 16-bit values to the ALU when an ALU operation is performed or the Data Memory for Memory operations. The code for the Register Memory can be found in the Appendix. In order to create a temporary storage for the 8 registers, we created a RAMtype that was defined as an 8-bit array made up of 16-bit vectors for the register locations. Using a process tied to the variations in the clk input, we designed the Register Memory with if statements to execute the respective operations depending on the function bits. During operations of an ALU, Load, or Set on Less than operation, the values sent to the DI input of the module would write those 16-bit values to the register location specified by the Destination Register. When a Store operation is being performed, a temporary signal called mem_s would equal the value stored in the register memory we wish to save to the Data Memory. In our special Store operation, if both the Store operation (1001) and the Address Flag is 1, then in addition to the mem_s signal, another signal called address_s would be used to take the 16-bit value stored in SR1 and use this value as a Data memory address we wish to store the mem_s value into. This special operation was written in order to achieve the 2nd program execution, such that we can update the location of storage in the Data Memory by using a value stored temporarily in a register rather than specify the address using an instruction. When this process ends, the mem_s and address_s signals are assigned to their appropriate output ports. When performing an ALU operation, the value of the SR1_DO port is assigned to the value within the address of SR1 and the value stored within the address of SR2 is assigned to a temp2 signal. A Register MUX module was designed and implemented into the Register Memory as a component such that the select bits of this module would either output the temp2 signal for standard ALU operations or the 16-bit sign extended Immediate value during ALU Immediate operations. This Register MUX checks the function bits for cases from 0000 to 1100 and assigns the Immediate value to the output on ADDi (0110) and ANDi (0111). In every other case the output is assigned to the temp2 signal holding the value of the SR2 data.

ALU

Having used the design of our ALU from a previous lab, we altered the ALU to meet the needs of this RISC Architecture project. The code for the altered ALU can be found in the Appendix. The ALU performs 16-bit operations so we adjusted the N-bit values of our previous ALU design to become a 16-bit ALU. The ALU contains an Adder, Subtract, 3-to-1 MUX, a 10-to-1 MUX, and modules instantiated to perform the AND, OR, SLL, and SLR operations. This ALU design includes two additional subtractors apart from the standard subtraction for both the Set on Less Than and Branch on Equal operations. For Set on Less Than, the ALU will use two Source Registers addressed in the instructions and perform a subtraction of input A minus input B. The 3-to-1 MUX inside the ALU checks for when a Borrow out or Overflow occurs, which we could use to check when a subtraction result is negative. This multiplexer also uses the function bits as select bits and will output the SOLT_barrow_out signal to the 10-to-1 MUX that also uses the function bits as select bits. If the result of the subtraction is negative, the SOLT_barrow_out signal will become 1 and would be used as an input into the Set on Less Than module. When this bit is 1, the Set on Less Than module will output a 16-bit value equal to 1, otherwise if the Borrow out is 0, then a 16-bit value equal to 0 will be outputted. One of the 16-bit values would be written back to the DI input of the Register Memory to store either value into a Destination Register. The second subtractor checks for when the Branch on Equal operation (1100) is called and would analyze the result to check if its zero. The Difference output of the subtractor is sent to the 10-to-1 MUX and would pass this result to the output when the Branch on Equal operation is called. Other than these changes, the ALU module is similar to our design in the previous lab.

Data Memory

The data memory is where values from registers can be stored to, as well as value that are currenting in the data memory that can also be loaded to registers. There are 256 memory locations, this is done by creating an array with 256 indices. However, the memory is byte addressable, so the array indices are only 1 byte, and every 16-bit input value is store in two consecutive memory addresses. There are six inputs and one output. The clock and function bits are inputs, the function bits allow the data memory to know if there was a store or load instruction. A memory address is an input as well, this is the memory address at which the load or store will take place. The load function is simpler so let us look at that first. In the clock process, the function input is checked to see if it was a load function, if it was then the values that are stored in data memory of the input address and the input address plus one is stored in a 16-bit temporary signal. This temporary value is put into a multiplexer. The purpose of the multiplexer is based off the function bits; it is to set the output value to the temporary load value if it is a load function, otherwise just store the output values to the input value. The input value is used for the store function, this is the value that is going to be stored into memory. The store function has two different stores, one is when the memory address for storing is directly from the instruction, and the other is when there is a source register, which the value inside it is a memory address, provided in the instruction. That is why there is an input address flag which indicates which store type it is. If the flag is zero, then it is a regular store. The address is taken from the address input and stores the input values in that location of the memory. If the flag is 1, then the address is taken from the register address input. This is the 16-bit values of a register which holds an address. That is the complete functionality of the data memory. There is one thing to note here. The clock process happens on every clock event, rising edges and falling edges. During testing we discovered that values were taking multiple clock ticks to perform the store or load function. We were not too sure why, but making the process execute on any clock event solved the problem.

Decoder Unit

The Decoder block has only one input that would be the 16-bit instruction that would be sent from the Program Memory and contains outputs that would tie to the Register Memory, Data Memory, and the Control block. The code for the Decoder can be found in Appendix, and it was designed using a VHDL process dependent on the changes to the Inst input. As described in the Instruction Bit Breakdown, the Decoder Unit follows the same bit assignments for each operation being performed. When an ALU or Set on Less Than operation is performed, the func bits were assigned to Inst(15 downto 12), the 3-bit addresses for the registers such as the DR became Inst(11 downto 9), SR1 was Inst(8 downto 6), and SR2 became Inst(5 downto 3). The remaining outputs for the Immediate value, 8-bit Memory Address, Jump Offset, FoB Flag, and PC_Address were assigned as don't cares ('-'). For debugging reasons, the Jump Flag and Address Flag outputs were assigned as 0. When an ADDi or ANDi operation was performed, the func bits were assigned again to Inst(15 downto 12), DR was Inst(11 downto 9), SR1 became Inst(8 downto 6), and the 6-bit Immediate value port was assigned to Inst(5 downto 0). For the Memory operations, Store/Load, the Decoder assigns func bits to Inst(15 downto 12), DR to Inst(11 downto 9), and the MemAdd port to Inst(8 downto 1). For the altered Store Function, the Decoder follows a similar format to the standard Store operation but checks if the Inst(0) bit is also equal to 1. For this scenario, the func bits become Inst(15 downto 12), DR is Inst(11 downto 9), SR1 becomes Inst(8 downt 6), and the Address Flag becomes Inst(0) of the instruction. For our Jump operation, the func bits are Inst(15 downto 12), the Jump Offset is Inst(11 downto 4), the Jump Flag is Inst(3), and the FoB Flag was Inst(2). When the Branch on Equal operation is performed the func bits are set to Inst(15 downto 14) and then are padded with two extra zeroes to keep the 4-bit function bit format. SR1 becomes Inst(13 downto 11), SR2 is assigned to Inst(10 downto 8) and the PC_Address is set to Inst(7 downto 0). The outputs of the Decoder would then need to be connected to the appropriate ports within the Instruction Execution Block.

Instruction Execution Unit

The code for the Instruction Execution Block can be seen in the Appendix and it ties together and includes the Decoder, Register Memory, the ALU, the Data Memory, ALU Store Mux, Check for Branch Module, the Sign Extension Module, and the Set on Less Than Module as components within this block. We chose to include the Decoder into this block because it helped our design validation when creating a testbench for this module. With the decoder inside, we could create a testbench and check that the ALU, Registers, and Data Memory were all performing the correct operations. Temporary signals were created to interconnect all the ports of the modules. The Decoder takes in the Inst input that would come from the Program Memory. A signal called Func_bits is decoded and used in the Register Memory, ALU, Set on Less Than Module, ALU Store Mux, Check for Branch Module, and the Data Memory. All three of the register's signals tie to the appropriate ports of the Register Memory for the two source registers and destination register. The 6-bit Immediate port ties to the Sign Extension Module, which extends the 6-bit Immediate value to an unsigned 16-bit value. The output of this module is tied to the Register Mux within the Register Memory. The Memory Address port of the Decoder ties to the Address port of the Data Memory block. The JumpOffset, JumpFlag, FoB Flag, and PC_Address ports of the Decoder are tied to the output ports of the

Instruction Execution Block so that they could be tied to the appropriate ports in the Control block in the RISC Architecture Structural module. The Address Flag from the Decoder ties to both the Register Memory and Data Memory for when an altered Store function occurs. The Register Memory and Data Memory are both tied to the clk input and operate during any clk'EVENT. We experimented with both having them operate during a rising and falling edge but ran into some problems that will be later explained, which left us just using a clk'EVENT occurrence. The Register Memory ties together the two Source Register data outputs and 16-bit Immediate value to the ALU, and ties to the Data Memory for the MEMREG port that would store a 16-bit value during the Store operation, and has a Write_Data signal tied to its DI input to save values back to a register. The ALU takes inputs from the two source data ports, and outputs a ALU_Result signal and overflow signal that would be used for the Set on Less Than Module. The MEMREG, SetOnLessThan, and ALU_Result signals all tie into the ALU Store Mux before reaching the Data Memory. This multiplexer was used with the function bits as select bits to determine which input to pass through to the Data Memory according to the operation performed. The MEMREG input would pass during a Store operation, the SetOnLessThan input passes through on a Set On Less Than operation, and the ALU_Result passes through when the operation performed is an ALU or ALU Immediate operation. The CheckForBranch Module also takes in the ALU_Result and checks when the function bits are equal to 1100 and the result was a 16-bit value equal to zero. If these statements are both true, then the output of the CheckForBranch Module, which is tied to the BranchFlag output of the Instruction Execution block, would be set to 1, otherwise it is set to 0. The Data Memory takes inputs from the clk, Func_bits, Address_Flag, and Memory_Address signals from the Decoder. The AddressReg is used to for the altered Store function that would use a 16-bit value in a register as a memory address. This port only examines bits 0 to 7 of this 16-bit value when accessing the Data Memory. The ALU Store MUX output is also tied to the input of the Data Memory, and it outputs a Write_Data signal that is tied back to the Register Memory.

Program Memory

For the program memory, 2 main programs were to be executed sequentially, which the goal was to show that our architecture was working as intended. The first program's goal was to showcase a sequence of load/store and ALU operations, while program 2's goal was to showcase some of the control commands such as "jump", "branch on equal" as well as "set on less than". First, we start with a fresh memory of type ROM, which is an array that starts from 0 to 255, meaning that it contains 256 cells with 8-bits per cell. Since each instruction is 16-bit long, two 8-bit cells are used for a single instruction in the memory. The program memory contains one 8-bit input of type unsigned which is used as our input address. It also contains one 16-bit output which is our actual fetched instruction that will be sent to the decoder to be decoded and executed. Below is showing the first program to be executed.

Program 1

```
Int A = 7;
Int B = 8;
Int my_arr[4]
my_arr[0] = A+B;
my_arr[1] = A or B;
my_arr[2] = A and B;
my_arr[3] = my_arr[1] - my_arr[2];
```

As can be seen in the appendix section, the first instruction in memory is "0110 000 111 000111", which the first 4 bits ("0110") is performing an "add immediate" function with register 7 and the number 7 and saves it into register 0. The second instruction in memory is "0110 001 111 001000", which the first 4 bits ("0110") is also performing an "add immediate" function with register 7 and the number 8 and saves it into register 1. The next instruction is "0000 010 000 001 000", the first 4 bits ("0000") is performing an "add" function with the content of register 0 which is 7 and with the content of register 1 which is 8 and the result into register 2, and the last three bits are "don't care" bits. The next instruction is "1001 010

00000000 0", which the first 4 bits ("1001") is performing a "store" function, the content of register 2 which is now 15 is stored into the data memory at address 0x00. Next instruction in memory is "0011 011 000 001 000", the first 4 bits indicates the "OR" is to be performed on the content of register 0 and the content of register 1 and saves the result into register 3. Next instruction is "1001011000000100" which is performing a "store" function to store the content of register 3 into data memory at the address 0x02. Next instruction is "0010 100 000 001 000", which performing an "AND" operation on the register 0 and register 1 and saves the result into register 4. Next instruction "1001 100 00000100 0", stores the content of register 4 into the data memory at address 0x04. Next instruction "1000 101 00000010 0", is loading the content of data memory at the memory address 0x02 and saves it into register 5. Next instruction "1000 110 00000100 0", is loading the content of data memory at the memory address 0x04 and saves it into register 6. Next instruction "0001 111 101 110 000", is subtracting the content of register 5 from the content of register 6 and saves the result into register 7. The last instruction of the first program "1001 111 00000110 0", stores the content of register 7 into data memory at memory address 0x06.

Program 2

```
Int range = 10;
Int Fib [range];
Fib[0]=0;
Fib[1]=1;
for (int i = 2; i < range; i++)
{
Fib[i] = Fib[i-1] + Fib[i-2];
}
```

Once the execution of the first program is completed, all 8 registers are cleared before loading the second program. First instruction of the 2nd program is "0110 000 101 001010", which is adding number 10 to register 0 representing the range. Next instruction "0110 001 101 000010" is adding number 2 to register 1 and saves value back into register 1, now contains the value 2 for i=2. Next instruction is "0110 010 101 000001", adds 1 to register 5 and saves it into register 2, which now contains the indexing increment. Next instruction "0110 011 101 000010", adds 1 to register 5 and saves it into register 3 that now contains the value of 2 for the address increment. Next instruction "0110 110 101 000001", adds 1 to register 5 and saves it into register 6. Next instruction is "0000 100 100 011 000", which adds register 4 to register 3, and saves in back into register 4, updating our address. Next instruction "1001 110 100 000001", stores the content of register 6 into the memory address in register 4. Next instruction "0000 100 100 011 000", adds register 4 to register 3, saves it back into register 4, updating our address. Instruction "0000 111 101 110 000", adds register 5 to register 6 and saves it into register 7 which contains the Fibonacci sequence "Fib[i]". Next instruction stores the content of register 7 to the memory address in register 4. On the next instruction, the address in register 4 is updated again. Instruction "0110 101 110 000000", adds 0 to register 6 and saves it into register 5. Next instruction "0110 110 111 000000", adds 0 to register 7 and saves it into register 6 switching the values. Next instruction "0000 001 010 010 000", adds register 1 to register 2 and saves it into register 1, incrementing our index. Next instruction "11 00000101001000", which is our branch instruction, which compares register 1 which is our counter to register 0 which contains the range value 10. If they are not equal, then the program goes to the next instruction, which is the jump instruction, looping back to a specific address in the instruction memory. Once the counter in register 1 is equal to range 10 in register 0, then the program branches out of the loop.

Update PC Module

The Update PC Module code can be seen in the Appendix and it contains an Adder, Subtractor, two Multiplexers, and the Control Select MUX. This module would take inputs from the Instruction Execution block such as the Jump Magnitude, the FoB Flag (called jump_back_flag in this module), the Jump Flag, the Branch Flag, and Branch PC Address. It also takes the PC input which would be the current PC count added with two. The 3-to-1 MUX then would have inputs of the PC+2 signal, the Jump signal result, and Branch PC Address. The Control Select module outputs the appropriate select bits for this 3-to-1 MUX depending on if the Jump Flag or Branch Flag are high. For a Jump Operation, the PC+2 Signal would either be added or subtracted from the Jump Magnitude value, the Jump Flag would be set to 1, thus the 3-to-1 MUX would allow the Jump result to pass through to the Instruction Fetch Block to go to the next instruction. When the Branch Flag is set high, the Branch PC Address value would pass through the multiplexer and the Instruction Fetch would resume the program count from the specified PC Address from the Branch operation. If neither of these operations are being called, then the MUX passes through the PC+2 count, and the program memory loads up the next instruction for the Decoder to decipher. The 2-to-1 MUX is used in the Control Block to determine whether the Add or Subtract result from the PC+2 and Jump Magnitude should become the Jump result for Jump operations. The output of the Update PC Module is the Address_out port and it is instantiated as the temporary signal PC_s that is the output of the 3-to-1 MUX.

RISC Architecture

The RISC Architecture ties the entire system design together and the code for it can be seen in the Appendix. This Structural design includes the Inst_Mem block, Ins_Execution_Block, and the Update_PC Module. The Inst_Mem block uses a temporary signal called PC_current that will serve as the current count the program is at. The output of the Inst_Mem uses a signal called Instruction that ties to the input of the Inst_Execution_Block. The clk is only tied to the Instruction Execution Block so that we can perform the necessary operations within the Register Memory and Data Memory, while everything else moves data asynchronously. The Instruction Execution Block is tied to the Update_PC block by the jumpMag, jumpFlag, branchFlag, fobFlag, and branchAddress signals. This ties all the modules together so that Control operations will be performed, and the entire design will Jump to a new PC count, or Branch to a new PC address. The Update_PC module is connected to the Inst_Mem by the PC_current signal as well. The Address_out port of the Update_PC module is assigned to a temp signal called PC_next. A VHDL process that is dependent on the clk signal is written into the RISC Architecture that instantiates the PC_current signal to the PC_next signal when a rising edge of the clock occurs. We initially implemented a state machine design, but soon realized we designed our architecture to be able to execute everything within one clock cycle, so the state machine did nothing except repeat certain instructions for an unnecessary number of times. We decided to comment out our state machine and assign the PC_current signal equal to the PC_next signal instead.

Results and Observations

This section discusses the simulated results of testbenches we ran to validate the operation of our individual models as we were stepping through the design process of this project. It starts off by creating testbenches for the modules within the Instruction Execution block, then the Update PC Module, and finally a testbench running the two programs we were tasked with executing within our architecture.

Simulation Results

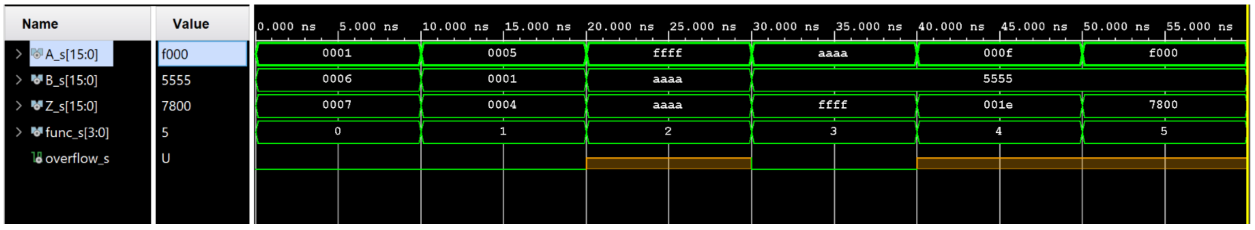

Name
Value
0.000 ns
5.000 ns
10.000 ns
15.000 ns
20.000 ns
25.000 ns
30.000 ns
35.000 ns
40.000 ns
45.000 ns
50.000 ns
55.000 ns
A_s[15:0] f000 0001 0005 ffff aaaa 000f f000
B_s[15:0] 5555 0006 0001 aaaa 5555
Z_s[15:0] 7800 0007 0004 aaaa ffff 001e 7800
func_s[3:0] 5 0 1 2 3 4 5
overflow_s U

Figure 6 – Results from ALU Testbench.

The figure above shows the ALU module testbench as it performs as expected when given certain operations. It performs an addition, subtraction, AND, OR, SLL, and SLR functions. The results match the expectations given the inputs for each combination of operations.

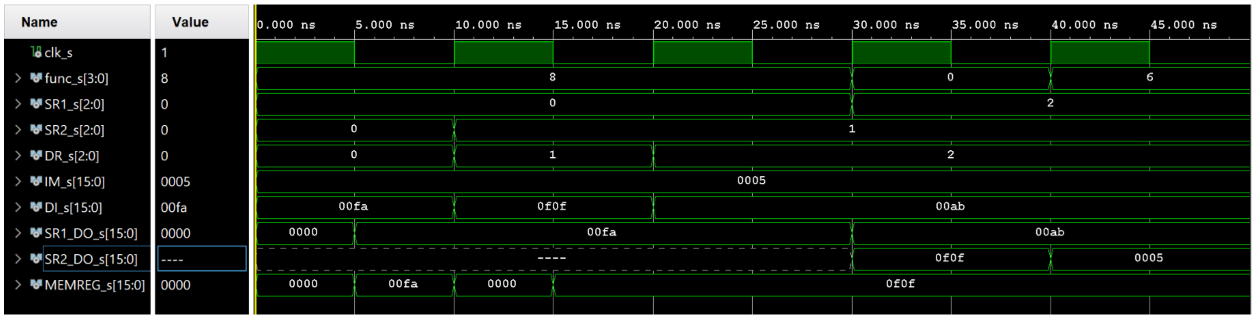


Figure 7 – Results from Register Memory Testbench.

The Register Memory testbench in Fig 7 above shows the module loading in values to source registers 0 and 1 and then calling an ALU Add and ADDi function. Here, the data memory was set to operate on the falling edge of the clock, so the first two clock ticks show the values of 00FA being loaded into R0 and value of 0F0F loaded into R1. The third clock tick shows the value of 00AB being loaded into R2 as well. When the ADD function is called the SR1_DO value reads as 00AB since SR address is R2 and SR2_DO reads 0F0F since SR2 is calling the address of R1. The last clock tick shows an ADDi function being called and again SR1_DO shows 00AB for R2 and the IM value of 0005 is displayed in SR2_DO port, which is what we expected due to the internal Register Mux.

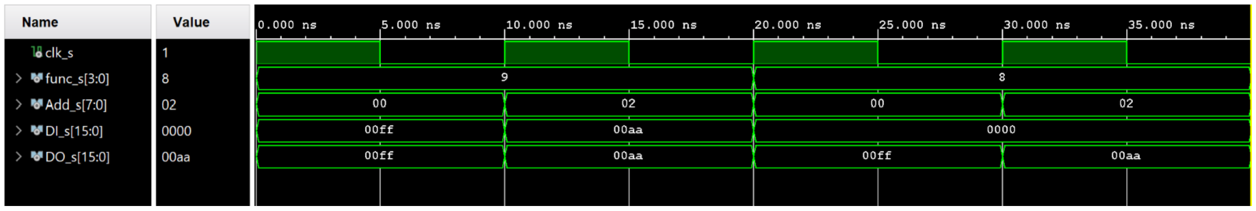


Figure 8 – Results from Data Memory Testbench.

The Data Memory testbench in Fig 8 above shows a Store operation being performed in the first two clock ticks. The value of 00FF and 00AA are being stored to addresses 0x00 and 0x02 in the Data Memory. When the Load function is called in the last two clock ticks, the value of 00FF appears at the output when using the 0x00 address. The last clock ticks calls the memory address of 0x02 and the value of 00AA appears at the output, which verified that this module is working correctly.

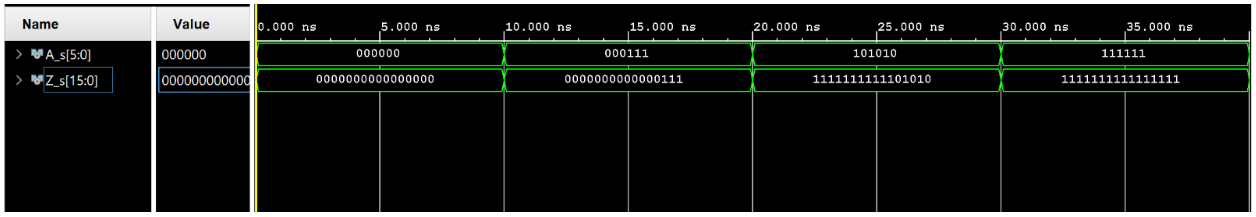


Figure 9 – Results from Sign Extension Testbench.

As can be seen from the sign extension test bench result, it is taking a 6-bit immediate value as an input which will be coming from the decoder and output a 16-bit unsigned value. If the most the significant is 1, then the 6-bit immediate value gets padded with 10 1's, if it is a 0, then it gets padded with 10 0's, as can be seen it's working as intended.

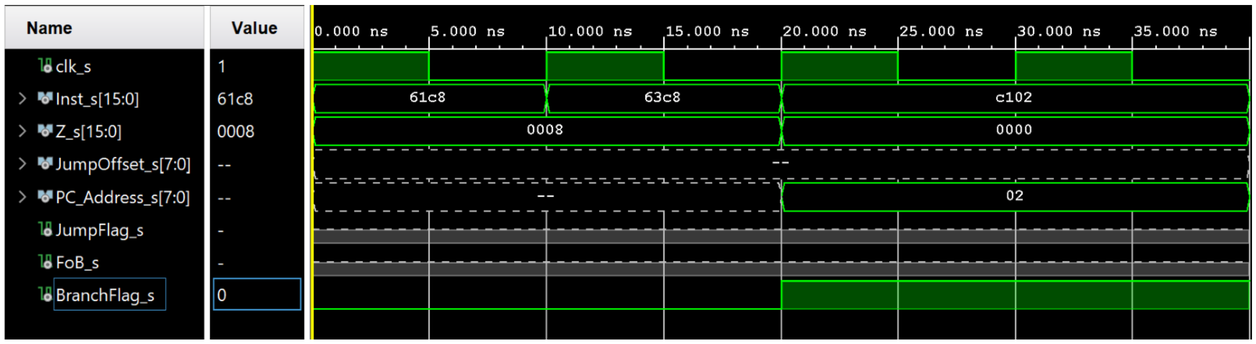


Figure 10 – Results from Check for Branch Testbench.

The results of the Check for Branch Testbench is shown above where the values of 8 is the Immediate value being added to two separate registers. When calling the Branch function in the third clock tick, the result of the ALU is shown to be all zeroes because the subtraction of both registers contained the same value. When this occurs the Branch Flag was set high and the PC Address to branch the program count was defined as 0x02 instead of the don't care bits. This demonstrates the proper operation of this module.

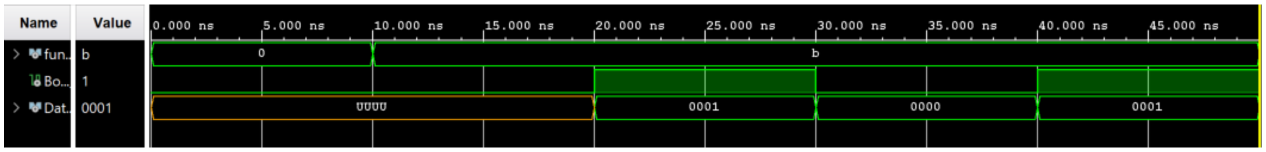


Figure 11 – Results from Set on Less Than Testbench.

The Set on Less Than function compares two values from two registers if the first value is less than the second, then the value in third register is set to 1, which would mean that when subtracting a number from another number, if the first number is greater than the second, then the resulting value would be a negative value, which would mean a borrow out in binary. And as can be seen from the test bench result, if the borrow out is 1, then then 3rd register is set to 1, otherwise it is set to 0.

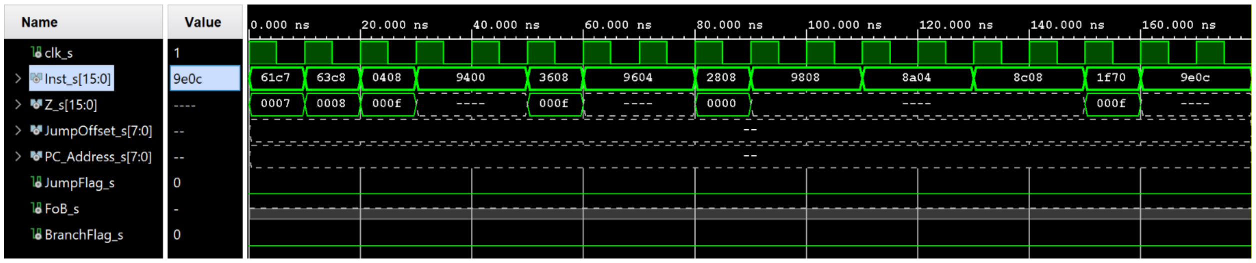


Figure 12 – Results from Instruction Execution Testbench for 1st Program.

Here we can see in Fig 12 the results of the Instruction Execution block successfully running the 1st program we were asked to execute. Since the first program did not involve any Control operations, we could use just the Instruction Execution block to check whether our design could perform the necessary operations for a successful execution. The first two clock ticks show the value of 7 and 8 being Added to registers R0 and R1 from the ADDi function. The third clock tick shows the result of A+B, where R0 was added to R1 and the result is stored to R2. This ALU result of 000F in R2 is then stored to memory address 0x00 in the Data Memory. Our design experienced a clock delay only on the Memory Store functions where it took an extra clock tick to save values to our Memory Data, which became a problem later when trying to execute the 2nd program. The next clock tick at 50ns shows A or B, where R0 and R1 are performed together by an OR operation and saves the result to R3. The result, again, is 000F, which is saved into Data Memory address 0x02 from R3 between 60ns-80ns. At 80ns, A AND B is performed, which results in a value of 0000 saved into R4. At 90ns-100ns, this value of 0000 inside R4 is stored into Memory address 0x04 of the Data Memory. From 110ns-150ns, the value in memory address 0x02 and the value in memory address 0x04 are loaded into registers R5 and R6. At 150ns, a subtraction of R5 from R6 is performed and results in the value of 000F saved into R7. The value in R7 is then saved to memory address 0x06 in the Data Memory and the testbench concludes. The results of each operation in this testbench were the exact expected values, which verifies our Instruction Execution block works properly.

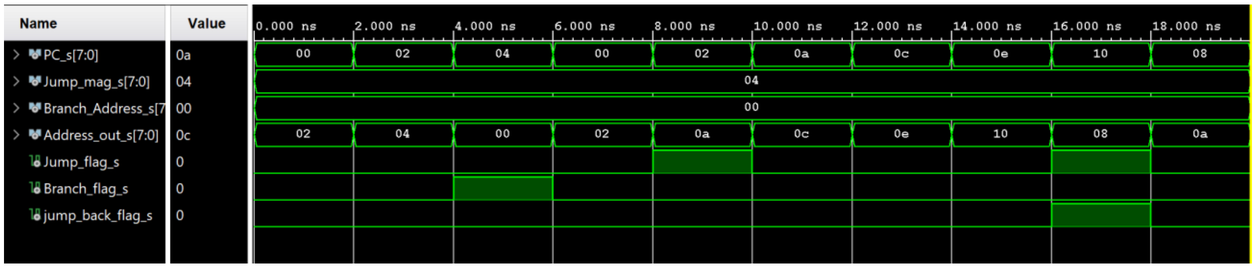


Figure 13 – Results from Update_PC Testbench.

The results from the Update PC Testbench can be seen in Fig 13 above and it shows the module performing as expected. Without the Jump Flag or Branch Flag set high, the PC counts in steps of two since the program memory would hold the next instruction every two bytes. When the Branch Flag is set high at 4ns, the PC branches to the Branch_Address value of 00 and in the next clock tick the PC starts at this address. In the next clock tick at 8ns, the Jump Flag is set high so the current PC is added together with Jump Magnitude value. The Jump Mag value is set to 0x04, but the Jump operation was designed such that this Magnitude value would be twice this amount. The Address_out value reads 0x0A because it is the result of 0x02, the current PC value, added together with two times 0x04, or 0x08, which results in a decimal value of 10 or A in hexadecimal. The PC count resumes from this address value and counts all the was until 0x10 or decimal value of 16. The jump_back_flag is set high indicated that the PC count will be subtracted from the Jump Magnitude and results in a value of 0x08, since this would mean 16 minus 8 in decimal. The testbench concludes proving that the module runs successfully and will allow the proper address to update the PC count when implemented into the Structural design.

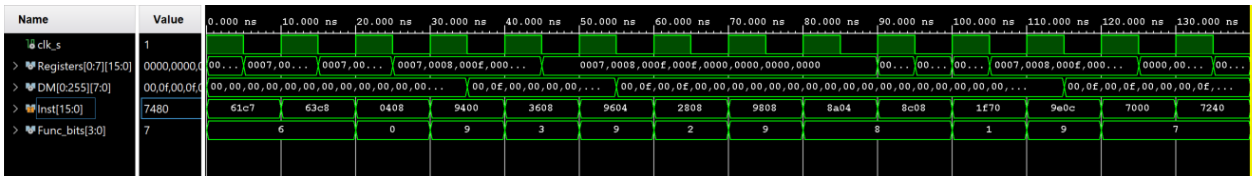


Figure 14 – Results from RISC Architecture Testbench running the 1st Program.

The results of the 1st program's execution using our entire RISC Architecture design is shown in Fig 14 above. The code for this testbench can also be found in Appendix. The 1st program is successfully executed as we read the instructions from the Program Memory and perform the necessary operations in the Instruction Execution Module. Values of 7 and 8 are added as immediate values into R0 and R1 in the first two clock ticks. At 20ns, the A+B operation is performed, and the result is saved into the Data Memory at the address of 0x00. On the falling edge of the clock tick at 30ns, the Data Memory updates with this value in the first address position. At 40ns, A OR B is performed, and the result is stored into Memory address 0x02, which the falling edge of the 50ns clock tick shows the Data Memory update with this value of 000F in the correct position. At 60ns, A AND B is performed and is stored into Memory Address of 0x04. We noticed that since we initialized the Data Memory will all zeroes to begin with, when we try to store the result of 0000 from the A AND B operation, the Data Memory doesn't update in our testbench. We assume this is because the value of 0000 is already in the address of 0x04. Loading in the values at memory locations 0x02 and 0x04 and then performing my_array[1] – my_array[2], the value of 000F is then stored into memory location 0x06, which successfully confirms our design can execute the 1st program.

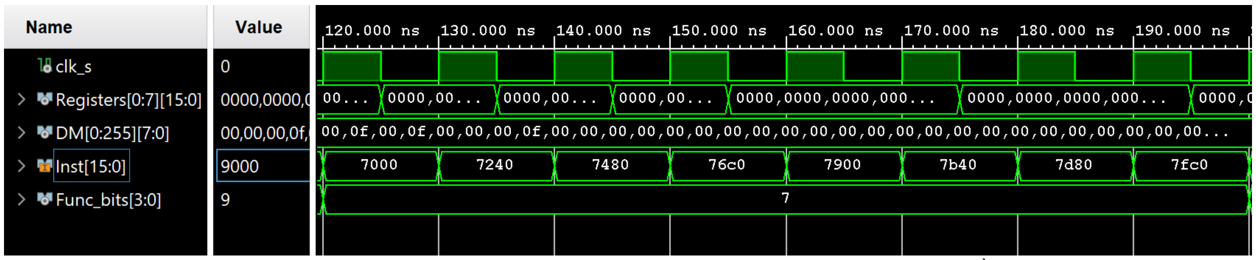


Figure 15 – Results from RISC Architecture Testbench clearing Registers before 2nd Program.

The testbench in Fig 15 above shows eight instructions we wrote into the Program Memory so that we can clear all the registers before starting the 2nd program. All the register values in the Register Memory are cleared by performing an ANDi operation with R6 which contained a value of 0000 inside it. All the register memory is cleared the end of this sequence of instructions and is ready to start the 2nd program.

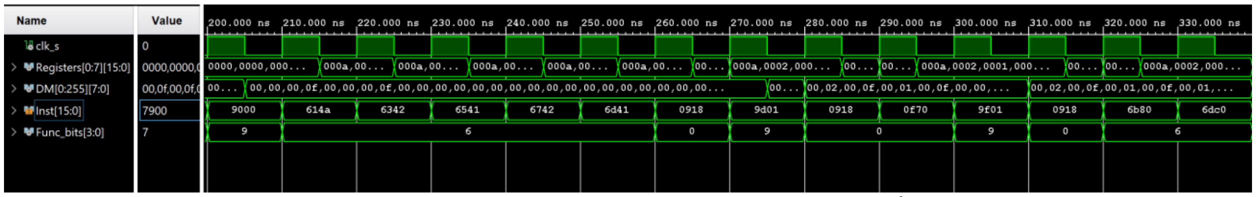


Figure 16 – Results from RISC Architecture Testbench running 2nd Program.

The 2nd program being executed on our RISC Architecture is shown in the testbench of Fig 16. It starts by storing the value of 0 from R0 into the Data Memory since Fib[0] is equal to 0. The next clock tick we start using the ADDi instruction to save some necessary values to our Register Memory. From 210ns to 260ns, we add an Immediate value of 10 to R0, which will keep track of the range of the for loop iterations in the program. We add an Immediate value of 2 to R1, which will keep track of the current value of i in the for loop. An Immediate value of 1 is added to R2 so that we can use this register to increment R1, or i, by one after every iteration of the loop. We then add an Immediate value of 2 to R3 because this register will serve as our address increment register. So, the altered Store function we created could be utilized by saving a value to the memory address stored in R3. After each iteration of the loop, this address register would be increased by 2 so that we could just store values to the next available Data Memory location without specifying a memory address. At 250ns, we add an Immediate value of 1 to R6 because Fib[1] will be equal to 1. At 260ns, we add R4 with R3 which will update our address register, but soon after we started to experience problems. We noticed when calling the altered Store function, that the values were not saving into the correct Memory Addresses. After performing some extensive debugging, we noticed our store function would need an extra clock tick to store a value to the Data Memory. We could not find the reason why this happening after carefully looking over every module of our code. Not being able to store a value within the same clock tick lead us to unsuccessfully execute the 2nd program.

SUMMARY

For this project, the goal was to design, test and implement a 16-bit RISC architecture that was able load, store and execute a simplified set of instructions using VHDL. The first thing that was done was defining and specifying the instruction machine code that would be used to implement each instruction set. Second, a flowchart was built that oversaw the whole design, it indicated all the required steps that would be taking place, all the different modules and the flow of executions. Once the overall design was well understood, codes as well as testbenches were written for all the different modules to make sure they were working properly. Each module was tested individually to make sure that they worked as intended. The ALU, which was used in a previous project was modified to fit our design and tested for all 6 operations which were, “Add”, “Subtract”, “And”, “OR”, “Shift left logical” and “Shift right logical”. Once tested, it was proven that the ALU met design specifications. The same design pattern was continued for all the other modules, such as the instruction fetch module, we made sure that right instruction could be fetched given a specific address. For the register module, we made sure that values could be uploaded to all 8 registers and that operations could be performed among them. Same thing was done for the data memory, all the multiplexers used, the decode module etc. At the end were able to successfully execute the first program, and as far as the second program, as mentioned in the second program testbench, we ran into an issue with the store function which was not being executed on the first rising edge of clock but was taking an extra clock tick to be executed, that in turn resulted in the 2nd program not being executed properly. At the end of this project, a lot was learned, a better understanding of the RISC architecture as well as how a CPU work was gained, we had a better grasp on how instructions are executed inside a CPU.

APPENDIX

RISC Architecture

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

entity RISC_Architecture is

  Port (

        clk : in std_logic );

end RISC_Architecture;

architecture Structural of RISC_Architecture is

component Inst_Mem is

    port(

        address : in unsigned(7 downto 0); --takes pc as input

        Inst_Set : out unsigned(15 downto 0)

        );

end component;

component Inst_Execution_Block is

    Port (

            clk : in std_logic;

            Inst : in unsigned(15 downto 0); -- 16-bit Instruction from Instruction Memory

            Z : out unsigned(15 downto 0); -- 16- bit Output from ALU

            JumpOffset : out unsigned (7 downto 0);

            JumpFlag : out std_logic;

            BranchFlag : out std_logic;

            FoB : out std_logic;
```

```
          PC_Address : out unsigned(7 downto 0) );
end component;
component Update_PC is
   Port ( PC : in unsigned(7 downto 0);
         Jump_mag : in unsigned(7 downto 0);
         Branch_Address : in unsigned(7 downto 0);
         Jump_flag : in STD_LOGIC;
         jump_back_flag: in STD_LOGIC;
         Branch_flag : in STD_LOGIC;
         Address_out : out unsigned(7 downto 0));
end component;
signal jumpFlag, branchFlag, fobFlag : std_logic := '0';
signal Instruction, Z_out : unsigned(15 downto 0) := "0000000000000000";
signal jumpMag, branchAddress, PC_current, PC_next : unsigned (7 downto 0) := "00000000";
type statetype is (S0, S1, S2);
signal state: statetype:= S0;
begin
   Program_Memory: Inst_Mem
   port map(
      address => PC_current,
      Inst_Set => Instruction );

   Decode_Execution_Block: Inst_Execution_Block
   port map (
   clk => clk,
   Inst => Instruction,
   Z => Z_out,
   JumpOffset => jumpMag,
   JumpFlag => jumpFlag,
   BranchFlag => branchFlag,
   FoB => fobFlag,
   PC_Address => branchAddress );

   UpdatePC: Update_PC
```

```
    port map (
    PC => PC_current,
    Jump_mag => jumpMag,
    Branch_Address => branchAddress,
    Jump_flag => jumpFlag,
    jump_back_flag => fobFlag,
    Branch_flag => branchFlag,
    Address_out => PC_next );

    FSM: process(clk)
    begin
    if (clk = '1' AND clk'EVENT) then
        PC_current <= PC_next;
--        case state is
--            when S0 =>
--                state <= S1;
--            when S1 =>
--                state <= S2;
--            when S2 =>
--                state <= S0;
--                PC_current <= PC_next;
--            end case;
        end if;
    end process;
end Structural;
```

RISC Architecture Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity RISC_tb is
end RISC_tb;
architecture stimulus of RISC_tb is
component RISC_Architecture is
  Port (
        clk : in std_logic );
end component;
signal clk_s : std_logic := '1';
begin
    UUT: RISC_Architecture
    port map (
    clk => clk_s );
clk_s <= not clk_s after 5 ns;
end stimulus;
```

Instruction Fetch Unit

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_unsigned.all;
Use ieee.std_logic_arith.all;
entity Inst_Fetch is
                  port ( clk : in std_logic;
                                      Reset : in std_logic;
                                      PC_in : in unsigned(7 downto 0);
                                      pc_out2 : out unsigned(7 downto 0);
                                      instruction : out unsigned (15 downto 0));
end Inst_Fetch;
architecture Arc_Fetch of Inst_Fetch is
signal PC_out1_s :  unsigned (7 downto 0);
  component program_counter is
                                    port(
                                    clk : in std_logic;
                                    PC_in : in unsigned (7 downto 0);
                                    Reset: in std_logic;
                                    PC_out1:  out unsigned(7 downto 0);
                                    PC_out2: out unsigned(7 downto 0));
  end component  program_counter;
  component Inst_Mem is
        port(
      address : in  unsigned(7 downto 0);---takes pc as input

                              Inst_Set  : out unsigned(15 downto 0));

  end component Inst_Mem;

begin

    P_C : program_counter

    port map(

    clk=> clk,

    PC_in=> PC_in,

    Reset=>Reset,

    PC_out1=> PC_out1_s,

    PC_out2 => PC_out2

    );

    I_M : Inst_Mem

    port map(

    address=> PC_out1_s,

    Inst_Set=> instruction

 );

end architecture Arc_Fetch ;
```

Program Memory

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity Inst_Mem is
port(
address : in unsigned(7 downto 0); --takes pc as input
Inst_Set : out unsigned(15 downto 0)
);
end entity Inst_Mem;
architecture RTL of Inst_Mem is
type ROMtype is array (0 to 255) of unsigned(7 downto 0);
signal ROM : ROMtype;
begin
  ------PROGRAM #1------
ROM(0)<=  "01100001";
ROM(1)<=  "11000111"; -- Addi #7 to R0(Contained 0's) Now contains A
ROM(2)<=  "01100011";
ROM(3)<=  "11001000";-- Addi #8 to R1(Contains 0's) Now contained B
ROM(4)<=  "00000100";
ROM(5)<=  "00001000"; -- A+B save to R2
ROM(6)<=  "10010100";
ROM(7)<=  "00000000";-- store result in R2 to Data Mem 0x00
ROM(8)<=  "00110110";
ROM(9)<=  "00001000"; -- A or B save to R3
ROM(10)<= "10010110";
ROM(11)<= "00000100";-- store result in R3 to Mem 0x02
ROM(12)<= "00101000";
ROM(13)<= "00001000";-- A and B save to R4
ROM(14)<= "10011000";
ROM(15)<= "00001000";-- store result in R4 to Data Mem 0x04
ROM(16)<= "10001010";
ROM(17)<= "00000100";-- load value in Data Mem 0x02 to R5
ROM(18)<= "10001100";
ROM(19)<= "00001000"; -- load value in Data Mem 0x04 to R6
ROM(20)<= "00011111";
ROM(21)<= "01110000";-- Subtract R5 to R6 and saves it into R7
ROM(22)<= "10011110";
ROM(23)<= "00001100";-- store result in R7 to Data Mem 0x06
--------CLEAR REGISTERS----------------------------
ROM(24)<= "01110000";
ROM(25)<= "00000000";----Andi #0 to R0 (Contained #7) put value back into R0(Reset R0)
ROM(26)<= "01110010";
ROM(27)<= "01000000";----Andi #0 to R1  put value back into R1(Reset R1)
ROM(28)<= "01110100";
ROM(29)<= "10000000";----Andi #0 to R2  put value back into R2(Reset R2)
ROM(30)<= "01110110";
ROM(31)<= "11000000";----Andi #0 to R3  put value back into R3(Reset R3)
ROM(32)<= "01111001";
ROM(33)<= "00000000";----Andi #0 to R4  put value back into R4(Reset R4)
ROM(34)<= "01111011";
ROM(35)<= "01000000";----Andi #0 to R5  put value back into R5(Reset R5)
ROM(36)<= "01111101";
ROM(37)<= "10000000";----Andi #0 to R6  put value back into R6(Reset R6)
ROM(38)<= "01111111";
ROM(39)<= "11000000";----Andi #0 to R7  put value back into R7(Reset R7)
ROM(40)<= "10010000";
ROM(41)<= "00000000";
------PROGRAM 2-----------
ROM(42)<= "01100001";
ROM(43)<= "01001010";-----Addi #10 to R0(Contained 0's), store value back into (R0), Now contains the value of 10 for int range=10
ROM(44)<= "01100011";
ROM(45)<= "01000010";----Addi #2 to R1(Contained 0's), store value back into (R1), Now contains the value of 2 for i=2
ROM(46)<= "01100101";
ROM(47)<= "01000001";----Addi #1 to R2(Contained 0's), store value back into (R2), Now contains the value of 1 for index increment
ROM(48)<= "01100111";
ROM(49)<= "01000010";----Addi #2 to R3(Contained 0's), store value back into (R3), Now contains the value of 2 for address increment
--ROM(42)<= "01101001";
--ROM(43)<= "00001000"; -- Addi 0x08 to R4, should be last address (Data Mem 0x08 in our case)
--ROM(42)<= "00001111";
--ROM(43)<= "01110000";---- Add R5 to R6 and store value in R7 (Contains Fib[i])
ROM(50)<= "01101101";
ROM(51)<= "01000001";---- Addi #1 to R6(Contained 0's), store value back into (R6)
ROM(52)<= "00001001";
ROM(53)<= "00011000";---- Add R4 to R3 (Updating our address) put it back into R4
ROM(54)<= "10011101";
ROM(55)<= "00000001";-----Store R6 to the memory address in R4
ROM(56)<= "00001001";
ROM(57)<= "00011000";---- Add R4 to R3 (Updating our address) put it back into R4
ROM(58)<= "00001111";
ROM(59)<= "01110000";---- Add R5 to R6 and store value in R7 (Contains Fib[i])
ROM(60)<= "10011111";
ROM(61)<= "00000001";--------   Store R7 to the memory address in R4
ROM(62)<= "00001001";
ROM(63)<= "00011000";---- Add R4 to R3 (Updating our address) put it back into R4
ROM(64)<= "01101011";
ROM(65)<= "10000000";----Addi #0 to R6 Store in R5
ROM(66)<= "01101101";
ROM(67)<= "11000000";----Addi #0 to R7 and save s into R6
ROM(68)<= "00000010";
ROM(69)<= "10010000";---- Add R1 to R2, stores back into R1 (Increment the index)
ROM(70)<= "11000001";
ROM(71)<= "01001000"; ----Branch( Compare R1(Indexing) to R0(#10)
ROM(72)<= "10100000";
ROM(73)<= "01111100";---- Jump
ROM(74)<= "00001001";
ROM(75)<= "00011000";---- Add R4 to R3 (Updating our address) put it back into R4
--Address input is used to access data in ROM
Inst_Set(15 downto 8) <= ROM(to_integer(address));
Inst_Set(7 downto 0) <= ROM(to_integer(address)+1);
end architecture RTL;
```

Update PC (Control Block) Module

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

entity Update_PC is
   Port ( PC : in unsigned(7 downto 0);
          Jump_mag : in unsigned(7 downto 0);
          Branch_Address : in unsigned(7 downto 0);
          Jump_flag : in STD_LOGIC;
          jump_back_flag: in STD_LOGIC;
          Branch_flag : in STD_LOGIC;
          Address_out : out unsigned(7 downto 0));
end Update_PC;
architecture Behavioral of Update_PC is
component Adder is
   generic(N : INTEGER := 8);
   port(
      A : in unsigned (N-1 downto 0);
      B : in unsigned (N-1 downto 0);
      Cin : in STD_LOGIC;
      S : out unsigned (N-1 downto 0);
      Cout : out STD_LOGIC);
end component;
component Subtractor is
   generic(N : INTEGER := 8);
   port(
      A : in unsigned (N-1 downto 0);
      B : in unsigned (N-1 downto 0);
      Barrow_in : in STD_LOGIC;
      Difference : out unsigned (N-1 downto 0);
      Barrow_out : out STD_LOGIC);
end component;
component MUX2to1_1bit is
   port(
      A : in unsigned(7 downto 0);
      B : in unsigned(7 downto 0);
      SEL : in STD_LOGIC;
      Z : out unsigned(7 downto 0));
end component;

component MUX3to1_2bit is
   Port ( Default_PC : in unsigned(7 downto 0);
          Jump : in unsigned(7 downto 0);
          Branch : in unsigned(7 downto 0);
          SEL : in std_logic_vector(1 downto 0);
          PC_out : out unsigned(7 downto 0));
end component;
component Control_select is
   Port ( Jump : in STD_LOGIC;
          Branch : in STD_LOGIC;
          SEL : out std_logic_vector(1 downto 0));
end component;
signal default_add_2 : unsigned(7 downto 0) := "00000010";
signal default_PC_s, jump_mag_s, jump_PC_s, jumpF_PC_s, jumpB_PC_s, branch_PC_s, PC_s : unsigned(7 downto 0) := "00000000";
signal control_s : std_logic_vector(1 downto 0) := "00";
signal low_logic : STD_LOGIC := '0';
signal two_carry_out, jump_carry_out, barrow_out_s : STD_LOGIC := '0';
begin
controller : Control_select
   port map(
      Jump => Jump_flag,
      Branch => Branch_flag,
      SEL => control_s);
 default_adder : Adder
   generic map (N=>8)
      port map (
      A => PC,
      B => default_add_2,
      Cin => low_logic,
      S => default_PC_s,
      Cout => two_carry_out);

Shift_left_func : Jump_mag_s <= jump_mag(6 downto 0) & '0';
jump_adder : Adder
   generic map (N=>8)
   port map (
      A => PC,
      B => Jump_mag_s,
      Cin => low_logic,
      S => jumpF_PC_s,
      Cout => jump_carry_out);
Subtract : Subtractor
   generic map (N=>8)
   port map (
   A => PC,
   B => Jump_mag_s,
   Barrow_in => low_logic,
   Difference => jumpB_PC_s,
   Barrow_out => barrow_out_s);
Mux2 : MUX2to1_1bit
   port map(
      A => jumpF_PC_s,
      B => jumpB_PC_s,
      SEL => jump_back_flag,
      Z => jump_PC_s);
Mux1 : MUX3to1_2bit
   port map(
      Default_PC => default_PC_s,
      Jump => jump_PC_s,
      Branch => Branch_Address,
      SEL => control_s,
      PC_out => PC_s );
Address_out <= PC_s;
end Behavioral;
```

Adder Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Adder is
    generic (N : positive := 16);
    Port ( A, B : in unsigned (N-1 downto 0);
           Cin : in STD_LOGIC;
           S : out unsigned (N-1 downto 0);
           Cout : out STD_LOGIC);
end Adder;
architecture Behavioral of Adder is
component Full_Adder is
    port(
        A : in STD_LOGIC;
        B : in STD_LOGIC;
        Carry_In : in STD_LOGIC;
        Sum : out STD_LOGIC;
        Carry_Out : out STD_LOGIC);
end component;
signal Cc : std_logic_vector(N downto 0);
begin
    Cc(0) <= Cin;
    Full_Adders: for i in 0 to N-1 generate
                FAx: Full_Adder port map (A(i), B(i), Cc(i), S(i), Cc(i+1));
            end generate;
     Cout <= Cc(N);
end Behavioral;
```

Subtractor Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Subtractor is
   generic (N : positive := 16);
   Port ( A, B : in unsigned (N-1 downto 0);
          Barrow_in : in STD_LOGIC;
          Difference : out unsigned (N-1 downto 0);
          Barrow_out : out STD_LOGIC);
end Subtractor;
architecture Behavioral of Subtractor is
component Full_Subtractor is
   port(
      A : in STD_LOGIC;
      B : in STD_LOGIC;
      Barrow_In : in STD_LOGIC;
      Difference : out STD_LOGIC;
      Barrow_Out : out STD_LOGIC);
end component;
signal Cc : unsigned(N downto 0);
begin
   Cc(0) <= Barrow_in;
   Full_Subtractors : for i in 0 to N-1 generate
               FAx: Full_Subtractor port map (A(i), B(i), Cc(i), Difference(i), Cc(i+1));
            end generate;
   Barrow_out <= Cc(N);
end Behavioral;
```

Controller Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Control_select is
   Port ( Jump : in STD_LOGIC;          Branch : in STD_LOGIC;
          SEL : out STD_LOGIC_VECTOR(1 downto 0));
end Control_select;
architecture Behavioral of Control_select is
begin
SEL <= Jump & Branch;
end Behavioral;
```

2 to 1 Multiplexer (1 Bit Select) Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity MUX2to1_1bit is
   Port ( A, B : in unsigned(7 downto 0);
          SEL : in STD_LOGIC;
          Z : out unsigned(7 downto 0));
end MUX2to1_1bit;
architecture dataflow of MUX2to1_1bit is
begin
   with SEL select Z <=
      A when '0',
      B when '1',
      "UUUUUUUU" when others;
end dataflow;
```

3 to 1 Multiplexer (2 Bit Select) Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity MUX3to1_2bit is
   Port ( Default_PC : in unsigned(7 downto 0);
        Jump : in unsigned(7 downto 0);
        Branch : in unsigned(7 downto 0);
        SEL : in STD_LOGIC_VECTOR(1 downto 0);
        PC_out : out unsigned(7 downto 0));
end MUX3to1_2bit;
architecture Behavioral of MUX3to1_2bit is
begin
    with SEL select PC_out <=
        Default_PC when "00",
        Branch when "01",
        Jump when "10",
        "UUUUUUUU" when others;
end Behavioral;
```

Register Memory Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity Register_Memory is
port(
    clk : in std_logic;
    func : in STD_LOGIC_VECTOR(3 downto 0);
    AddressFlag : in std_logic;
    SR1 : in unsigned(2 downto 0); -- SR1 Address
    SR2 : in unsigned(2 downto 0); -- SR2 Address
    DR : in unsigned(2 downto 0); -- DR Address
    IM : in unsigned(15 downto 0); -- 16-bit Immediate Value
    DI : in unsigned(15 downto 0); -- Data In
    SR1_DO: out unsigned(15 downto 0); -- SR1 Data Out
    SR2_DO: out unsigned(15 downto 0); -- SR2 Data Out
    MEMREG : out unsigned(15 downto 0); -- Memory Address Register
    AddressReg : out unsigned(15 downto 0)
);
end Register_Memory;
architecture Behavorial of Register_Memory is
component RegisterMUX is
    Port (  func : in STD_LOGIC_VECTOR(3 downto 0);
            SRD2 : in unsigned(15 downto 0);
            Immediate : in unsigned (15 downto 0);
            Z : out unsigned(15 downto 0));
end component;
type RAMtype is array(0 to 7) of unsigned (15 downto 0); -- 8 Register locations
signal Registers: RAMtype := (others => (others => '0'));
signal temp1, temp2 : unsigned(15 downto 0);
signal mem_s, address_s : unsigned(15 downto 0):= "0000000000000000";
signal what_is_happening : unsigned(2 downto 0) := "000";
begin
    process (clk) is
    begin
        if (clk'event)  then
            if func(3) = '0' or func = "1000" or func = "1011" then
                Registers(to_integer(DR)) <= DI;
                mem_s <= Registers(to_integer(DR));
                what_is_happening <= "001";
            elsif func = "1001" AND AddressFlag = '0' then
                mem_s <= Registers(to_integer(DR));
                what_is_happening <= "010";
            elsif func = "1001" AND AddressFlag = '1' then
                address_s <= Registers(to_integer(SR1));
                mem_s <= Registers(to_integer(DR));
                what_is_happening <= "011";
            end if;
        end if;
    end process;
    MEMREG <= mem_s;
    AddressReg <= address_s;
    SR1_DO <= Registers(to_integer(SR1)); -- read from SR1 address and place values to SR1_DO
              temp2 <= Registers(to_integer(SR2)); -- read from SR2 address and place values to temp2
    MUX: RegisterMUX
    port map(
            func => func,
            SRD2 => temp2,
            Immediate => IM,
            Z => SR2_DO );
end Behavorial;
```

Register MUX

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity RegisterMUX is
    Port (
            func : in STD_LOGIC_VECTOR(3 downto 0);
            SRD2 : in unsigned(15 downto 0);
            Immediate : in unsigned (15 downto 0);
            Z : out unsigned(15 downto 0));
end RegisterMUX;
architecture Behavioral of RegisterMUX is
begin
    with func select Z <=
        SRD2 when "0000", -- SRD2 when ADD Operation performed
        SRD2 when "0001", -- SRD2 when SUBTRACT Operation performed
        SRD2 when "0010", -- SRD2 when AND Operation Performed
        SRD2 when "0011", -- SRD2 when OR Operation Performed
        SRD2 when "0100", -- SRD2 when SHIFT LEFT Operation Performed
        SRD2 when "0101", -- SRD2 when SHIFT RIGHT Operation Performed
        Immediate when "0110", -- IMMEDIATE when ADDi Operation Performed
        Immediate when "0111", -- IMMEDIATE when ANDi Operation Performed
        SRD2 when "1011", -- SET ON LESS THAN
        SRD2 when "1100", -- BRANCH ON EQUAL
        "----------------" when others;
end Behavioral;
```

Data Memory Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.numeric_std.all;
entity Data_memory is
port(
    clk : in std_logic; -- clock signal
    func : in STD_LOGIC_VECTOR(3 downto 0); -- func select bits from instruction
    AddressFlag: in std_logic;
    Address: in unsigned(7 downto 0); -- Address
    AddressReg: in unsigned(15 downto 0);
    DI: in unsigned (15 downto 0); -- Data In
    DO: out unsigned(15 downto 0) -- Data Out
    );
end Data_memory;
architecture Behavorial of Data_memory is
component MemoryMUX is
    Port (
            func : in STD_LOGIC_VECTOR(3 downto 0);
            Mem : in unsigned(15 downto 0);
            A : in unsigned(15 downto 0);
            Z : out unsigned(15 downto 0));
end component;
type DMtype is array(0 to 255) of unsigned (7 downto 0); -- 256 Memory locations due to 8-bit address
signal DM : DMtype := (others => (others => '0'));
signal temp : unsigned(15 downto 0) := "0000000000000000";
signal address_tmp : unsigned(7 downto 0) := "00000000";
signal data1, data2, data3, data4 : unsigned(7 downto 0):= "00000000";
begin
    process (clk) is
    begin
```

```
        if  clk'event then
            if func = "1001" AND AddressFlag ='0' then -- if STORE instruction
                DM(to_integer(Address)) <= DI(15 downto 8);
                DM(to_integer(Address)+1) <= DI(7 downto 0);
            elsif func = "1001" AND AddressFlag = '1' then
                DM(to_integer(AddressReg(7 downto 0))) <= DI(15 downto 8);
                DM(to_integer(AddressReg(7 downto 0)+1)) <= DI(7 downto 0);
            elsif func = "1000" then -- if LOAD instruction
                temp(15 downto 8) <= DM(to_integer(Address));
                temp(7 downto 0) <= DM(to_integer(Address)+1);
            end if;
      end if;
end process;
    MUX: MemoryMUX
    port map(
            func => func,
            Mem => temp,
            A => DI,
            Z => DO );
end Behavorial;
```

Data Memory MUX

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity MemoryMUX is
    Port (
            func : in STD_LOGIC_VECTOR(3 downto 0); -- func select bits from instruction
            Mem : in unsigned(15 downto 0); -- Memory stored values
            A : in unsigned(15 downto 0); -- ALU result
            Z : out unsigned(15 downto 0)); -- Output of MUX to Write back to Register Memory
end MemoryMUX;
architecture Behavioral of MemoryMUX is
begin
    Z <= Mem when func = "1000" else A;
end Behavioral;
```

ALU Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Arithmetic_Logic_Unit is
    Port ( A : in unsigned (15 downto 0);
           B : in unsigned (15 downto 0);
           Function_Select : in STD_LOGIC_VECTOR (3 downto 0);
           Function_Out : out unsigned (15 downto 0);
           Overflow_Out : out STD_LOGIC);
end Arithmetic_Logic_Unit;
architecture Behavioral of Arithmetic_Logic_Unit is
component Adder is
    port(
        A : in unsigned (15 downto 0);
        B : in unsigned (15 downto 0);
        Cin : in STD_LOGIC;
        S : out unsigned (15 downto 0);
        Cout : out STD_LOGIC);
end component;
component Subtractor is
    port(
        A : in unsigned (15 downto 0);
        B : in unsigned (15 downto 0);
        Barrow_in : in STD_LOGIC;
        Difference : out unsigned(15 downto 0);
        Barrow_out : out STD_LOGIC);
end component;

component MUX3to1 is
```

```
    port(
        A : in STD_LOGIC;
        B : in STD_LOGIC;
        C : in STD_LOGIC;
        SEL : in STD_LOGIC_VECTOR (3 downto 0);
        Z : out STD_LOGIC);
end component;
component MUX10to1 is
    port(
        A : in unsigned (15 downto 0);
        B : in unsigned (15 downto 0);
        C : in unsigned (15 downto 0);
        D : in unsigned (15 downto 0);
        E : in unsigned (15 downto 0);
        F : in unsigned (15 downto 0);
        G : in unsigned (15 downto 0);
        H : in unsigned (15 downto 0);
        I : in unsigned (15 downto 0);
        J : in unsigned (15 downto 0);
        SEL : in STD_LOGIC_VECTOR (3 downto 0);
        Z : out unsigned(15 downto 0));
end component;
signal A_out, S_out, And_gate_out, Or_gate_out, SOLT_out, Branch_out : unsigned (15 downto 0);
signal Shift_left_out, Shift_right_out : unsigned (15 downto 0);
signal A_carry_out, S_barrow_out, SOLT_barrow_out: STD_LOGIC := '0';
signal low_logic, Branch_low : STD_LOGIC := '0';
begin
Add: Adder
    port map (
    A => A,
    B => B,
    Cin => low_logic,
    S => A_out,
    Cout => A_carry_out);
Subtract : Subtractor
    port map (
    A => A,
    B => B,
    Barrow_in => low_logic,
    Difference => S_out,
    Barrow_out => S_barrow_out);
SOLT_Subtract: Subtractor
    port map (
    A => A,
    B => B,
    Barrow_in => low_logic,
    Difference => SOLT_out,
    Barrow_out => SOLT_barrow_out );
Branch_Subtract: Subtractor
    port map (
    A => A,
    B => B,
    Barrow_in => low_logic,
    Difference => Branch_out,
    Barrow_out => Branch_low );
Mux2 : MUX3to1 port map(
    A => A_carry_out,
    B => S_barrow_out,
    C => SOLT_barrow_out,
    SEL => Function_Select,
    Z => Overflow_Out);
And_Gate : And_gate_out <= (A AND B);
Or_Gate : Or_gate_out <= (A OR B);
Shift_left_func : Shift_left_out <=  A(16-2 downto 0) & '0';
Shift_right_func : Shift_right_out <= '0' & A(16-1 downto 1);
Mux11: MUX10to1 port map(
    A => A_out,
    B => S_out,
    C => And_gate_out,
    D => Or_gate_out,
    E => Shift_left_out,
    F => Shift_right_out,
    G => A_out,
    H => And_gate_out,
    I => SOLT_out,
    J => Branch_out,
    SEL => Function_Select,
    Z => Function_Out);
end Behavioral;
```

10 to 1 MUX

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity MUX10to1 is
    Port ( A : in unsigned (15 downto 0);
           B : in unsigned (15 downto 0);
           C : in unsigned (15 downto 0);
           D : in unsigned (15 downto 0);
           E : in unsigned (15 downto 0);
           F : in unsigned (15 downto 0);
           G : in unsigned (15 downto 0);
           H : in unsigned (15 downto 0);
           I : in unsigned (15 downto 0);
           J : in unsigned (15 downto 0);
           SEL : in STD_LOGIC_VECTOR (3 downto 0);
           Z : out unsigned (15 downto 0));
end MUX10to1;
architecture dataflow of MUX10to1 is
begin
 with SEL select Z <=
        A when "0000", -- ADD
        B when "0001", -- SUBTRACT
        C when "0010", -- AND
        D when "0011", -- OR
        E when "0100", -- SHIFT LEFT
        F when "0101", -- SHIFT RIGHT
        G when "0110", -- ADDI
        H when "0111", -- ANDI
        I when "1011", -- SET ON LESS THAN
        J when "1100", -- BRANCH ON EQUAL
        "----------------" when others;
end dataflow;
```

Decoder Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Decoder is
    Port (
        Inst : in unsigned (15 downto 0);
        func : out std_logic_vector(3 downto 0);
        SR1 : out unsigned(2 downto 0);
        SR2 : out unsigned(2 downto 0);
        DR : out unsigned(2 downto 0);
        IM : out unsigned(5 downto 0);
        MemAdd : out unsigned (7 downto 0);
        JumpOffset : out unsigned (7 downto 0);
        JumpFlag : out std_logic;
        FoB : out std_logic;
        PC_Address : out unsigned(7 downto 0);
        AddressFlag : out std_logic
    );
end Decoder;
architecture Behavioral of Decoder is
signal Branch : std_logic_vector (1 downto 0);
signal zeros : std_logic_vector (1 downto 0) := "00";
```

```
begin
Decode: process(Inst)
   begin
      if Inst(15 downto 12) = "0000" then -- ADD Operation
         func <= std_logic_vector(Inst(15 downto 12));
         DR <= Inst(11 downto 9);
         SR1 <= Inst(8 downto 6);
         SR2 <= Inst(5 downto 3);
         IM <= "------";
         MemAdd <= "--------";
         JumpOffset <= "--------";
         JumpFlag <= '0';
         FoB <= '-';
         PC_Address <= "--------";
         AddressFlag <= '0';
      elsif Inst(15 downto 12) = "0001" then -- SUBTRACT Operation
         func <= std_logic_vector(Inst(15 downto 12));
         DR <= Inst(11 downto 9);
         SR1 <= Inst(8 downto 6);
         SR2 <= Inst(5 downto 3);
         IM <= "------";
         MemAdd <= "--------";
         JumpOffset <= "--------";
         JumpFlag <= '0';
         FoB <= '-';
         PC_Address <= "--------";
         AddressFlag <= '0';
      elsif Inst(15 downto 12) = "0010" then -- AND Operation
         func <= std_logic_vector(Inst(15 downto 12));
         DR <= Inst(11 downto 9);
         SR1 <= Inst(8 downto 6);
         SR2 <= Inst(5 downto 3);
         IM <= "------";
         MemAdd <= "--------";
         JumpOffset <= "--------";
         JumpFlag <= '0';
         FoB <= '-';
         PC_Address <= "--------";
         AddressFlag <= '0';
      elsif Inst(15 downto 12) = "0011" then -- OR Operation
         func <= std_logic_vector(Inst(15 downto 12));
         DR <= Inst(11 downto 9);
         SR1 <= Inst(8 downto 6);
         SR2 <= Inst(5 downto 3);
         IM <= "------";
         MemAdd <= "--------";
         JumpOffset <= "--------";
         JumpFlag <= '0';
         FoB <= '-';
         PC_Address <= "--------";
         AddressFlag <= '0';
      elsif Inst(15 downto 12) = "0100" then -- SHIFT LEFT Operation
         func <= std_logic_vector(Inst(15 downto 12));
         DR <= Inst(11 downto 9);
         SR1 <= Inst(8 downto 6);
         SR2 <= Inst(5 downto 3);
         IM <= "------";
         MemAdd <= "--------";
         JumpOffset <= "--------";
         JumpFlag <= '0';
         FoB <= '-';
         PC_Address <= "--------";
         AddressFlag <= '0';
      elsif Inst(15 downto 12) = "0101" then -- SHIFT RIGHT Operation
         func <= std_logic_vector(Inst(15 downto 12));
         DR <= Inst(11 downto 9);
         SR1 <= Inst(8 downto 6);
         SR2 <= Inst(5 downto 3);
         IM <= "------";
         MemAdd <= "--------";
```

```
    JumpOffset <= "--------";
    JumpFlag <= '0';
    FoB <= '-';
    PC_Address <= "--------";
    AddressFlag <= '0';
elsif Inst(15 downto 12) = "0110" then -- ADD Immediate Operation
    func <= std_logic_vector(Inst(15 downto 12));
    DR <= Inst(11 downto 9);
    SR1 <= Inst(8 downto 6);
    SR2 <= "---";
    IM <= Inst(5 downto 0);
    MemAdd <= "--------";
    JumpOffset <= "--------";
    JumpFlag <= '0';
    FoB <= '-';
    PC_Address <= "--------";
    AddressFlag <= '0';
elsif Inst(15 downto 12) = "0111" then -- AND Immediate Operation
    func <= std_logic_vector(Inst(15 downto 12));
    DR <= Inst(11 downto 9);
    SR1 <= Inst(8 downto 6);
    SR2 <= "---";
    IM <= Inst(5 downto 0);
    MemAdd <= "--------";
    JumpOffset <= "--------";
    JumpFlag <= '0';
    FoB <= '-';
    PC_Address <= "--------";
    AddressFlag <= '0';
elsif Inst(15 downto 12) = "1000" then -- LOAD Operation
    func <= std_logic_vector(Inst(15 downto 12));
    SR1 <= "---";
    SR2 <= "---";
    DR <= Inst(11 downto 9);
    IM <= "------";
    MemAdd <= Inst(8 downto 1);
    JumpOffset <= "--------";
    JumpFlag <= '0';
    FoB <= '-';
    PC_Address <= "--------";
    AddressFlag <= '0';
 elsif Inst(15 downto 12) = "1001" AND Inst(0) = '1' then -- STORE Operation
    func <= std_logic_vector(Inst(15 downto 12));
    SR1 <= Inst(8 downto 6);
    SR2 <= "---";
    DR <= Inst(11 downto 9);
    IM <= "------";
    MemAdd <= "--------";
    JumpOffset <= "--------";
    JumpFlag <= '0';
    FoB <= '-';
    PC_Address <= "--------";
    AddressFlag <= std_logic(Inst(0));
elsif Inst(15 downto 12) = "1001" then -- STORE Operation
    func <= std_logic_vector(Inst(15 downto 12));
    SR1 <= "---";
    SR2 <= "---";
    DR <= Inst(11 downto 9);
    IM <= "------";
    MemAdd <= Inst(8 downto 1);
    JumpOffset <= "--------";
    JumpFlag <= '0';
    FoB <= '-';
    PC_Address <= "--------";
    AddressFlag <= std_logic(Inst(0));
elsif Inst(15 downto 12) = "1010"  then -- JUMP Operation
    func <= std_logic_vector(Inst(15 downto 12));
    SR1 <= "---";
    SR2 <= "---";
    DR <= "---";
```

```
            IM <= "------";
            MemAdd <= "--------";
            JumpOffset <= Inst(11 downto 4);
            JumpFlag <= std_logic(Inst(3));
            FoB <= std_logic(Inst(2));
            PC_Address <= "--------";
            AddressFlag <= '0';
        elsif Inst(15 downto 12) = "1011" then -- SET ON LESS THAN Operation
            func <= std_logic_vector(Inst(15 downto 12));
            SR1 <= Inst(8 downto 6);
            SR2 <= Inst(5 downto 3);
            DR <= Inst(11 downto 9);
            IM <= "------";
            MemAdd <= "--------";
            JumpOffset <= "--------";
            JumpFlag <= '0';
            FoB <= '-';
            PC_Address <= "--------";
            AddressFlag <= '0';
        elsif Inst(15 downto 14) = "11" then -- BRANCH ON EQUAL Operation
            func <= std_logic_vector(Inst(15 downto 14) & "00");
            SR1 <= Inst(13 downto 11);
            SR2 <= Inst(10 downto 8);
            DR <= "---";
            IM <= "------";
            MemAdd <= "--------";
            JumpOffset <= "--------";
            JumpFlag <= '0';
            FoB <= '-';
            PC_Address <= Inst(7 downto 0);
            AddressFlag <= '0';
        end if;
        end process;
end Behavioral;
```

Sign Extension Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Sign_Extend is
    Port ( A : in unsigned(5 downto 0);
           Z : out unsigned(15 downto 0));
end Sign_Extend;
architecture Behavioral of Sign_Extend is
begin
    Z <= unsigned(resize(signed(A), Z'length)); -- Extends 6-bit immediate value to unsigned 16-bit
end Behavioral;
```

Check for Branch Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity CheckForBranch is
  Port (
        func_bits : in std_logic_vector(3 downto 0);
        ALU_Result : in unsigned(15 downto 0);
        BranchFlag : out std_logic );
end CheckForBranch;
architecture Behavioral of CheckForBranch is
signal Branch_tmp : std_logic;
begin
    process(func_bits, ALU_Result)
        begin
        if (func_bits = "1100") AND (ALU_Result = "0000000000000000") then
            Branch_tmp <= '1';
        else
            Branch_tmp <= '0';
```

```
      end if;
   end process;
   BranchFlag <= Branch_tmp;
end Behavioral;
```

Instruction Execution Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Inst_Execution_Block is
   Port (
         clk : in std_logic;
         Inst : in unsigned(15 downto 0); -- 16-bit Instruction from Instruction Memory
         Z : out unsigned(15 downto 0); -- 16- bit Output from ALU
         JumpOffset : out unsigned (7 downto 0);
         JumpFlag : out std_logic;
         BranchFlag : out std_logic;
         FoB : out std_logic;
         PC_Address : out unsigned(7 downto 0) );
end Inst_Execution_Block;
architecture Behavioral of Inst_Execution_Block is
component Decoder is
Port (
      Inst : in unsigned (15 downto 0);
      func : out std_logic_vector(3 downto 0);
      SR1 : out unsigned(2 downto 0);
      SR2 : out unsigned(2 downto 0);
      DR : out unsigned(2 downto 0);
      IM : out unsigned(5 downto 0);
      MemAdd : out unsigned (7 downto 0);
      JumpOffset : out unsigned (7 downto 0);
      JumpFlag : out std_logic;
      FoB : out std_logic;
      PC_Address : out unsigned(7 downto 0);
      AddressFlag : out std_logic
      );
end component;
component Register_Memory is
port(
   clk : in std_logic;
   func : in STD_LOGIC_VECTOR(3 downto 0);
   AddressFlag : in std_logic;
   SR1 : in unsigned(2 downto 0); -- SR1 Address
   SR2 : in unsigned(2 downto 0); -- SR2 Address
   DR : in unsigned(2 downto 0); -- DR Address
   IM : in unsigned(15 downto 0); -- 16-bit Immediate Value
   DI : in unsigned(15 downto 0); -- Data In
   SR1_DO: out unsigned(15 downto 0); -- SR1 Data Out
   SR2_DO: out unsigned(15 downto 0); -- SR2 Data Out
   MEMREG : out unsigned(15 downto 0); -- Memory Address Register
   AddressReg : out unsigned(15 downto 0)
   );
end component;
component Arithmetic_Logic_Unit is
   Port ( A : in unsigned (15 downto 0);
        B : in unsigned (15 downto 0);
        Function_Select : in STD_LOGIC_VECTOR (3 downto 0);
        Function_Out : out unsigned (15 downto 0);
        Overflow_Out : out STD_LOGIC
        );
end component;
component SetOnLessThan is
  Port (
      func : in std_logic_vector(3 downto 0);
      Borrow : in std_logic;
      DataOut : out unsigned(15 downto 0)
       );
end component;
component ALU_STORE_MUX is
```

```
  Port (
        func : in std_logic_vector(3 downto 0);
        MemReg : in unsigned(15 downto 0);
        ALU_Out : in unsigned(15 downto 0);
        SetLessThan : in unsigned(15 downto 0);
        DataOut : out unsigned(15 downto 0));
end component;
component CheckForBranch is
  Port (
        func_bits : in std_logic_vector(3 downto 0);
        ALU_Result : in unsigned(15 downto 0);
        BranchFlag : out std_logic );
end component;
component Data_memory is
port(
    clk : in std_logic; -- clock signal
    func : in STD_LOGIC_VECTOR(3 downto 0); -- func select bits from instruction
    AddressFlag : in std_logic;
    Address : in unsigned(7 downto 0); -- Address
    AddressReg : in unsigned(15 downto 0);
    DI: in unsigned (15 downto 0); -- Data In
    DO: out unsigned(15 downto 0) -- Data Out
    );
end component;
component Sign_Extend is
    Port ( A : in unsigned(5 downto 0);
           Z : out unsigned(15 downto 0)
           );
end component;
signal overflow, Address_Flag : std_logic:= '0';
signal Func_bits : std_logic_vector(3 downto 0):= "0000";
signal SourceReg1_Address, SourceReg2_Address, DestReg_Address : unsigned(2 downto 0):= "000";
signal Immed_6_bits : unsigned (5 downto 0):= "000000";
signal Immed_16_bits, SR1_Data, SR2_Data, Address_Reg : unsigned (15 downto 0):= "0000000000000000";
signal Memory_Address : unsigned (7 downto 0):= "00000000";
signal Write_Data : unsigned (15 downto 0):= "0000000000000000";
signal ALU_Result : unsigned (15 downto 0):= "0000000000000000";
signal MemRegAdd : unsigned (15 downto 0) := "0000000000000000";
signal MuxOut : unsigned (15 downto 0) := "0000000000000000";
signal SetOnLess : unsigned (15 downto 0) := "0000000000000000";
begin
    DECODE: Decoder
    port map (
    Inst => Inst,
    func => Func_bits,
    SR1 => SourceReg1_Address,
    SR2 => SourceReg2_Address,
    DR => DestReg_Address,
    IM => Immed_6_bits,
    MemAdd => Memory_Address,
    JumpOffset => JumpOffset,
    JumpFlag => JumpFlag,
    FoB => FoB,
    PC_Address => PC_Address,
    AddressFlag => Address_Flag );

    SIGN_EXT: Sign_Extend
    port map (
    A => Immed_6_bits,
    Z => Immed_16_bits );

    REGISTERS: Register_Memory
    port map(
    clk => clk,
    func => Func_bits,
    AddressFlag => Address_Flag,
    SR1 => SourceReg1_Address,
    SR2 => SourceReg2_Address,
    DR => DestReg_Address,
    IM => Immed_16_bits,
    DI => Write_Data,
    SR1_DO => SR1_Data,
    SR2_DO => SR2_Data,
    MEMREG => MemRegAdd,
    AddressReg => Address_Reg );

    ALU: Arithmetic_Logic_Unit
    port map (
    A => SR1_Data,
    B => SR2_Data,
    Function_Select => Func_bits,
    Function_Out => ALU_Result,
    Overflow_Out => overflow );

    SetLessThan : SetOnLessThan
    port map (
    func => Func_bits,
    Borrow => overflow,
    DataOut => SetOnLess );

    STORE_MUX: ALU_STORE_MUX
    port map (
    func => Func_bits,
    MemReg => MemRegAdd,
    SetLessThan => SetOnLess,
    ALU_Out => ALU_Result,
    DataOut => MuxOut );

    Branch: CheckForBranch
    port map (
    func_bits => Func_bits,
    ALU_Result => ALU_Result,
    BranchFlag => BranchFlag );

    DATA_MEM: Data_Memory
    port map (
    clk => clk,
    func => Func_bits,
    AddressFlag => Address_Flag,
    AddressReg => Address_Reg,
    Address => Memory_Address,
    DI => MuxOut,
    DO => Write_Data );

    Z <= ALU_Result;

end Behavioral;
```

Set on Less Than Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity SetOnLessThan is
  Port (
        func : in std_logic_vector(3 downto 0);
        Borrow : in std_logic;
        DataOut : out unsigned(15 downto 0)
         );
end SetOnLessThan;
architecture Behavioral of SetOnLessThan is
begin
    process(Borrow)
    begin
    if func = "1011" then -- if SET ON LESS THAN
        if Borrow = '0' then
            DataOut <= "0000000000000000";
        elsif Borrow = '1' then
            DataOut <= "0000000000000001";
        end if;
    end if;
end process;
end Behavioral;
```

ALU Store MUX

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity ALU_STORE_MUX is
  Port (
        func : in std_logic_vector(3 downto 0);
        MemReg : in unsigned(15 downto 0);
        ALU_Out : in unsigned(15 downto 0);
        SetLessThan : in unsigned(15 downto 0);
        DataOut : out unsigned(15 downto 0));
end ALU_STORE_MUX;
architecture Behavioral of ALU_STORE_MUX is
begin
with func select DataOut <=
    ALU_Out when "0000", -- ADD
    ALU_Out when "0001", -- SUBTRACT
    ALU_Out when "0010", -- AND
    ALU_Out when "0011", -- OR
    ALU_Out when "0100", -- SHIFT LEFT
    ALU_Out when "0101", -- SHIFT RIGHT
    ALU_Out when "0110", -- ADDI
    ALU_Out when "0111", -- ANDI
    MemReg when "1001", -- STORE
    SetLessThan when "1011", -- SET ON LESS THAN
    ALU_Out when "1100", -- BRANCH ON EQUAL
    "----------------" when others;
end Behavioral;
```

ALU Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity ALU_tb is
end ALU_tb;
architecture stimulus of ALU_tb is
component Arithmetic_Logic_Unit is
    Port ( A : in unsigned (15 downto 0);
           B : in unsigned (15 downto 0);
           Function_Select : in STD_LOGIC_VECTOR (3 downto 0);
           Function_Out : out unsigned (15 downto 0);
           Overflow_Out : out STD_LOGIC);
end component;
signal A_s, B_s, Z_s : unsigned (15 downto 0) := "0000000000000000";
signal func_s : std_logic_vector (3 downto 0) := "0000";
signal overflow_s : std_logic := '0';
begin
    UUT: Arithmetic_Logic_Unit
    port map(
            A => A_s,
            B => B_s,
            Function_Select => func_s,
            Function_Out => Z_s,
            Overflow_Out => overflow_s );
func_s <= "0000" after 0ns, "0001" after 10ns, "0010" after 20ns, "0011" after 30ns, "0100" after 40ns, "0101" after 50ns, "0001" after 60ns;
A_s <= "0000000000000001" after 0ns, "0000000000000101" after 10ns, "1111111111111111" after 20ns, "1010101010101010" after 30ns,
"0000000000001111" after 40ns, "1111000000000000" after 50ns, "0000000000001111" after 60ns;
B_s <= "0000000000000110" after 0ns, "0000000000000001" after 10ns, "1010101010101010" after 20ns, "0101010101010101" after 30ns,
"0000000000000000" after 60ns;
end stimulus;
```

Register Memory Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity RegisterMemory_tb is
end RegisterMemory_tb;
architecture stimulus of RegisterMemory_tb is
component Register_Memory is
port(
    clk : in std_logic;
    func : in STD_LOGIC_VECTOR(3 downto 0);
    SR1 : in unsigned(2 downto 0); -- SR1 Address
    SR2 : in unsigned(2 downto 0); -- SR2 Address
    DR : in unsigned(2 downto 0); -- DR Address
    IM : in unsigned(15 downto 0); -- 16-bit Immediate Value
    DI : in unsigned(15 downto 0); -- Data In
    SR1_DO: out unsigned(15 downto 0); -- SR1 Data Out
    SR2_DO: out unsigned(15 downto 0); -- SR2 Data Out
    MEMREG : out unsigned(15 downto 0) -- Memory Address Register
);
end component;
signal clk_s : std_logic := '1';
signal func_s : std_logic_vector(3 downto 0) := "0000";
signal SR1_s, SR2_s, DR_s : unsigned(2 downto 0) := "000";
signal IM_s, DI_s, SR1_DO_s, SR2_DO_s, MEMREG_s : unsigned(15 downto 0) := "0000000000000000";
begin
    UUT: Register_Memory
    port map(
        clk => clk_s,
        func => func_s,
        SR1 => SR1_s,
        SR2 => SR2_s,
        DR => DR_s,
        IM => IM_s,
        DI => DI_s,
        SR1_DO => SR1_DO_s,
        SR2_DO => SR2_DO_s,
        MEMREG => MEMREG_S );
clk_s <= not clk_s after 5ns;
func_s <= "1000" after 0ns, "0000" after 30ns, "0110" after 40ns, "1000" after 50ns, "0001" after 70ns;
DR_s <= "000" after 0ns, "001" after 10ns, "010" after 20ns, "101" after 50ns, "110" after 60ns;
DI_s <= "0000000011111010" after 0ns, "0000111100001111" after 10ns, "0000000010101011" after 20ns, "0000000000001111" after 50ns,
"0000000000000000" after 60ns;
IM_s <= "0000000000000101" after 0ns;
SR1_s <= "000" after 10ns, "010" after 30ns, "101" after 50ns;
SR2_s <= "001" after 10ns, "110" after 60ns;
end stimulus;
```

Data Memory Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity DataMemory_tb is
end DataMemory_tb;
architecture stimulus of DataMemory_tb is
component Data_Memory is
port(
    clk : in std_logic; -- clock signal
    func : in STD_LOGIC_VECTOR(3 downto 0); -- func select bits from instruction
    Address: in unsigned(7 downto 0); -- Address
    DI: in unsigned (15 downto 0); -- Data In
    DO: out unsigned(15 downto 0) -- Data Out
    );
end component;
signal clk_s : std_logic:= '1';
signal func_s : std_logic_vector(3 downto 0) := "0000";
signal Add_s : unsigned(7 downto 0) := "00000000";
signal DI_s, DO_s : unsigned(15 downto 0) := "0000000000000000";
begin
    UUT: Data_Memory
    port map(
            clk => clk_s,
            func => func_s,
            Address => Add_s,
            DI => DI_s,
            DO => DO_s );
clk_s <= not clk_s after 5 ns;
func_s <= "1001" after 0ns, "1000" after 20ns, "1001" after 40ns, "1000" after 60ns, "0000" after 80ns;
DI_s <= "0000000011111111" after 0ns, "0000000010101010" after 10ns, "0000000000000000" after 20ns,"0000000000001111" after 40ns,
"0000000000000000" after 50ns;
Add_s <= "00000000" after 0ns, "00000010" after 10ns, "00000000" after 20ns, "00000010" after 30ns, "00000010" after 40ns, "00000100" after
50ns, "00000010" after 60ns, "00000100" after 70ns;
end stimulus;
```

Sign Extension Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity SignExtend_tb is
end SignExtend_tb;
architecture Behavioral of SignExtend_tb is
component Sign_Extend is
    Port ( A : in STD_LOGIC_VECTOR(5 downto 0);
           Z : out unsigned(15 downto 0));
end component;
signal A_s : STD_LOGIC_VECTOR(5 downto 0) := "000000";
signal Z_s : unsigned(15 downto 0):= "0000000000000000";
begin
    UUT: Sign_Extend
    port map(
            A => A_s,
            Z => Z_s );
    A_s <= "000111" after 10ns, "101010" after 20ns, "111111" after 30ns;
end Behavioral;
```

Set On Less Than Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity SetOnLessThan_tb is
end SetOnLessThan_tb;
architecture Behavioral of SetOnLessThan_tb is
component SetOnLessThan is
  Port (
        func : in std_logic_vector(3 downto 0);
        Borrow : in std_logic;
        DataOut : out unsigned(15 downto 0)
         );
end component;
signal func_s : std_logic_vector(3 downto 0):= "0000";
signal Borrow_s : std_logic:= '0';
signal DataOut_s : unsigned(15 downto 0) := "0000000000000000";
begin
    UUT: SetOnLessThan
    port map(
    func => func_s,
    Borrow => Borrow_s,
    DataOut => DataOut_s );
func_s <= "1011" after 10ns;
Borrow_s <= '0' after 10ns, '1' after 20ns, '0' after 30ns, '1' after 40ns;
end Behavioral;
```

Check For Branch Testbench

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity CheckForBranch_tb is
end CheckForBranch_tb;
architecture stimulus of CheckForBranch_tb is
component CheckForBranch is
  Port (
        func_bits : in std_logic_vector(3 downto 0);
        ALU_Result : in unsigned(15 downto 0);
        BranchFlag : out std_logic );
end component;
signal func_s : std_logic_vector(3 downto 0) := "0000";
signal ALUResult_s : unsigned(15 downto 0):= "0000000000000001";
signal BranchFlag_s : std_logic := '0';
begin
   UUT: CheckForBranch
   port map (
   func_bits => func_s,
   ALU_Result => ALUResult_s,
   BranchFlag => BranchFlag_s );
func_s <= "1100" after 10ns;
ALUResult_s <= "0000000000000000" after 20ns;
end stimulus;
```

Instruction Execution Testbench

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Inst_Exec_tb is
end Inst_Exec_tb;
architecture stimulus of Inst_Exec_tb is
component Inst_Execution_Block is
   Port (
          clk : in std_logic;
          Inst : in unsigned(15 downto 0); -- 16-bit Instruction from Instruction Memory
          Z : out unsigned(15 downto 0); -- 16- bit Output from ALU
          JumpOffset : out unsigned (7 downto 0);
          JumpFlag : out std_logic;
          BranchFlag : out std_logic;
          FoB : out std_logic;
          PC_Address : out unsigned(7 downto 0) );
end component;
signal clk_s : std_logic := '1';
signal Inst_s, Z_s : unsigned (15 downto 0) := "0000000000000000";
signal JumpOffset_s, PC_Address_s : unsigned(7 downto 0):= "00000000";
signal JumpFlag_s, FoB_s, BranchFlag_s : std_logic := '0';
begin
   UUT: Inst_Execution_Block
   port map (
       clk => clk_s,
       Inst => Inst_s,
       Z => Z_s,
       JumpOffset => JumpOffset_s,
       JumpFlag => JumpFlag_s,
       BranchFlag => BranchFlag_s,
       FoB => FoB_s,
       PC_Address => PC_Address_s );
clk_s <= not clk_s after 5 ns;
Inst_s <= "0110000111000111" after 0ns, "0110001111001000" after 10ns, "1001000001000001" after 20ns, "0000010000001000" after 20ns,
"1001010000000000" after 30ns, "0011011000001000" after 50ns, "1001011000000100" after 60 ns, "0010100000001000" after 80ns,
"1001100000001000" after 90ns, "1000101000000100" after 110ns, "1000110000001000" after 130ns, "0001111101110000" after 150ns,
"1001111000001100" after 160ns;
-- "0000010000001000" after 40ns;
-- "0000010000001000" after 20ns, "1001010000000000" after 30ns, "0011011000001000" after 50ns, "1001011000000100" after 60 ns,
"0010100000001000" after 80ns, "1001100000001000" after 90ns, "1000101000000100" after 110ns, "1000110000001000" after 130ns,
"0001111101110000" after 150ns, "1001111000001100" after 160ns;
-- "0000010000001000" after 20ns, "1001010000000000" after 30ns, "0011011000001000" after 50ns, "1001011000000100" after 60 ns,
"0010100000001000" after 80ns, "1001100000001000" after 90ns, "1000101000000100" after 110ns, "1000110000001000" after 130ns,
"0001111101110000" after 150ns, "1001111000001100" after 160ns;
-- "1011010001000000" after 20ns, "1011011000001000" after 30ns;
-- "1100000100000010" after 20ns,
end stimulus;
```