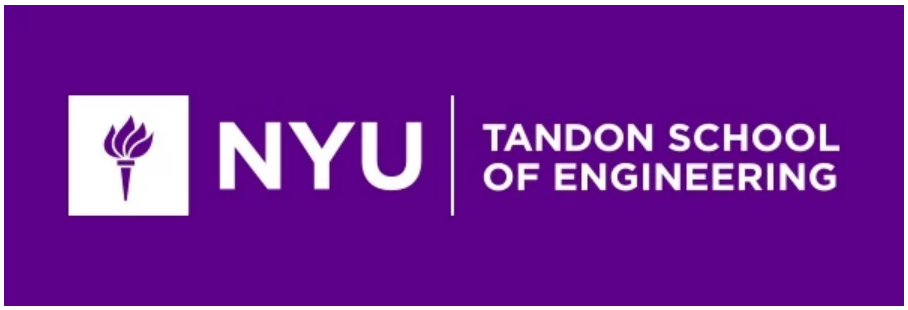
32-bit CPU Implemented on an FPGA

Wyatt Lien

Jesse Coma

Fahim Ghani

Sundeep Kaler



EE 4323 Computer Engineering Design Project

NYU Tandon School of Engineering

Contents

Abstract

An FPGA (Field Programmable Gate Array) was programmed to work as a 32-bit-processor loosely based on the R2000 microprocessor chip set. This processor is a CPU (Central Processing Unit) that interprets instructions from the MIPS (Microprocessor without Interlocked Pipeline Stages) I instruction set. The processor is fully pipelined and supports some data forwarding. The program is written in VHDL (Very high speed IC Hardware Description Language). The processor was implemented on a Diligent Nexys 4 FPGA. The processor was tested using a MIPS assembly program that counts instances of a number in an array. The test program was successful, and made use of various branch, load, store, arithmetic, and jump type instructions.

1 Introduction

VHDL is a hardware description language used to design electronic systems such as FPGAs or ICs (Integrated Circuits). The language provides a text model to describe logic circuits and simulate an interface with the logic. This project was done with the intention of helping model the MIPS I architecture in a way that can help visualize a 32-bit processor from a higher level of programming. The code, written in VHDL, translates directly to logic circuits.

VHDL is also useful in it being a parallel programming language, meaning the logic works as it would in parallel computer architecture. If desired, actions within the same process can performed at the same time, especially useful for Boolean logic. The other advantage to VHDL is that the simulations, like mentioned earlier, benefit the debugging and testing process. A testbench file can be created to streamline the process of synthesizing code

without directly routing to an FPGA.

MIPS is a reduced instruction set computer(RISC) instruction set architecture, developed and implemented on the R2000 32-bit processor in 1986. RISC allows for fewer cycles per instruction than otherwise. This means that the instructions themselves are made simple, rather than larger sets of more complex instructions. In this project, we have implemented instructions from the MIPS I instruction set.

The instructions are processed through the technique of pipelining, which maintains instruction parallelism by introducing stages. Pipelining attempts to divide the work involved with every instruction into sequential steps. As a result, different steps of different instructions can be worked on at the same time, effectively increasing the throughput of the CPU. The RISC architecture consists of five steps:

- Instruction Fetch (IF)
- Instruction Decode and Register Fetch (ID)
- Execute (EX)
- Memory Access (MEM)
- Register Write Back (WB)

Together, the operations are known as the datapath. In the Instruction Fetch (IF) stage, the Program Counter (PC), a register, holds the address of the current instruction being read from memory. While that instruction is sent to the Instruction Cache, the PC predicts the address of the next instruction by incrementing the PC by 4. All instructions are 4

bytes (32 bits) long. Each instruction contains a 6-bit opcode that labels the signal, which is read during the Instruction Decode (ID) stage. The opcode is read and the registers being pointed to by the instructions are read by the CPU. After the registers are accessed, during the Execution (EX) stage, the ALU performs a logical or arithmetic operation, depending on the task. If there is a jump or branch instruction given, in which the PC is incremented to set amounts in order to fetch instructions further in instruction memory. If needed, during the Memory (MEM) stage, memory is accessed to perform operations. In the final stage, the Writeback (WB) stage, the register files are updated with the new information computed during the pipelined cycle. Once this operation is completed, the cycle repeats and the CPU moves back to IF, ready to compute the next instruction.

The datapath is created by connecting multiple blocks in hardware, connected by signals carrying instructions and data. The blocks include the ALU, the GPR, the memory unit, the control unit, as well as many other registers. In this project, a simple program is written that can be executed by the CPU. The program can be stored in the memory unit as a set of multiple instructions which will result in an output that can be displayed on an FPGA. Our memory unit consists of two separate memory areas: a program instruction rom, and a data memory ram.

2 Specifications

2.1 Instruction Set

The instructions supported by our processor are presented in the following tables, separated into their respective functions. This is not the full MIPS I Architecture instruction set, but rather a shortened version, given this project's time frame:

Instruction	Mnemonic	Opcode/Function	Syntax	Operation
Add	add	100000	f $d, $s, $t	$d = $s + $t
Add Unsigned	addu	100001	f $d, $s, $t	$d = $s + $t
Add Immediate	addi	001000	f $d, $s, i	$d = $s + Imm
Add Immediate Unsigned	addiu	001001	f $d, $s, i	$d = $s + Imm
And	and	100100	f $d, $s, $t	$d = $s & $t
And Immediate	andi	001100	f $d, $s, i	$t = $s & Imm
Nor	nor	100111	f $d, $s, $t	$d = !($s — $t)
Or	or	100101	f $d, $s, $t	$d = $s — $t
Or Immediate	ori	001101	f $d, $s, i	$t = $s — Imm
Shift Left Logical	sll	000000	f $d, $t, a	$d = $t << a
Shift Left Logical Variable	sllv	000100	f $d, $t, $s	$d = $t << $s
Shift Right Arithmetic	sra	000011	f $d, $t, a	$d = $t >> a
Shift Right Arithmetic Variable	srav	000111	f $d, $t, $s	$d = $t >> $s
Shift Right Logical	srl	000010	f $d, $t, a	$d = $t >>> a

Shift Right Logical Variable	srlv	000110	f $d, $t, $s	$d = $t >>> $s
Subrtract	sub	100010	f $d, $s, $t	$d = $s - $t
Subtract Unsigned	subu	100011	f $d, $s, $t	$d = $s - $t
Xor	xor	100110	f $d, $s, $t	$d = $s ^ $t
Xor Immediate	xori	001110	f $d, $s, i	$d = $s ^ Imm

Table 1: Arithmetic and Logical Instructions

Instruction	Mnemonic	Opcode/ Function	Syntax	Operation
Set on <	Slt	101010	f $d, $s, $t	$d = ($s ¡ $t)
Set on < Unsigned	sltu	101001	f $d, $s, $t	$d = ($s < $t)
Set on < Immediate	slti	001010	f $d, $s, i	$t = ($s < Imm)
Set on < Immediate Unsigned	sltiu	001001	f $d, $s, i	$t = ($s < Imm)

Table 2: Comparison Instructions

Instruction	Mnemonic	Opcode/ Function	Syntax	Operation
Branch on Equal	beq	000100	o $s, $t, label	if ($s == $t) pc += i << 2
Branch on > Zero	bgtz	000111	o $s, label	if ($s > 0) pc += i << 2
Branch on ≤ Zero	blez	000110	o $s, label	if ($s <= 0) pc += i << 2
Branch on Not Equal	bne	000101	o $s, $t, label	if ($s != $t) pc += i << 2

Table 3: Branch Instructions

Instruction	Mnemonic	Opcode/ Function	Syntax	Operation
Jump	j	000010	o label	pc += i << 2

Table 4: Jump Instructions

Instruction	Mnemonic	Opcode/ Function	Syntax	Operation
Load Word	lw	100011	o $t, i ($s)	$t = MEM [$s + i]:4
Store Word	sw	101011	o $t, i ($s)	MEM [$s + i]:4 = $t

Table 5: Load/Store Instructions

2.2 Datapath

This section will discuss the details of the datapath architecture used. The pipelined implementation is based off a basic bare-bones MIPS pipeline architecture that uses partial data forwarding and delayed branches. The forwarding unit was added to the pipelined datapath, which is shown below.

As can be seen (and as was specified earlier), there are five stages in the datapath. These are: instruction fetch, instruction decode, execute, memory, and write back. The main

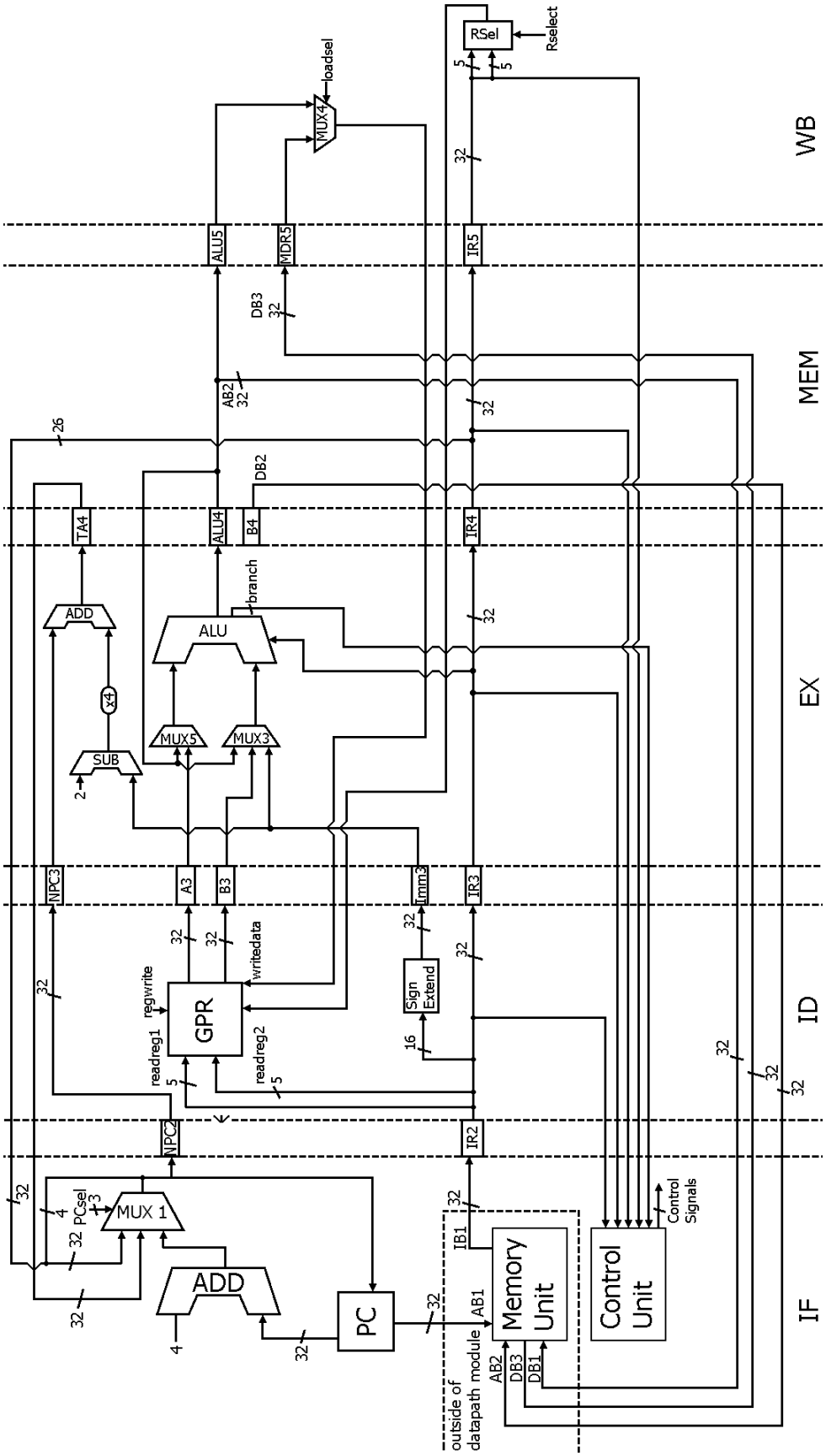


Figure 1: Pipelined Datapath Diagram

difference between the unpipelined and this pipelined architecture is that more than one instruction is being processed at a time. This allows an instruction to be fetched every clock cycle. The registers at each stage retain information about the instruction at each stage. These registers are enumerated, i.e. there is an IR2, IR3, etc. The hardware shown in this datapath can execute load words, store words, branches, arithmetic operations, and jumps. The inputs and outputs to the datapath (to and from the memory unit) are shown. The details will be discussed further in the implementation section. Here only an overview is given. The same goes for the control signals to the multiplexers (which are not shown in the datapath). It can be seen that PC goes through AB1 (address bus 1) to the memory unit, and the instruction is fetched and comes back on IB1 (instruction bus 1). DB2 carries the output of the ALU to be stored to memory. DB3 retrieves data from memory at the address provided to the memory unit by AB2. A control signal "memwrite" tells the memory when to write to the address at AB2. All control signals are generated by passing the values of the instruction registers to the control unit. From this data it decides which control signals to output. The ALU similarly takes in the opcode and function bits of IR3 to decide what operation to perform. It has an output "branch" signal that is set after a compare and tells the control unit if it should execute the branch or not in branch instructions.

2.2.1 Branch

Because a branch statement must resolve a condition (i.e. branch if equal to must compare if the contents of two registers are the same), it can only be taken after the EX stage has completed because the Boolean result of the condition is computed in the ALU. The condition

sets the bit that determines whether or not to take the branch, and here that bit is stored in register 4.Zero. Thus, because this take three clock periods, and instructions are being fetched every clock period, instructions that might affect the contents of certain registers, the branch must be delayed. Two no-ops (NOP instructions) or unrelated instructions must be executed after each branch, hence a delayed branch. The NPC registers are used to store the program counter at the time the branch is calculated, and the offset is added to this value, which is the purpose of the adder in the EX stage. There is also a subtractor in the EX stage. The purpose of this subtractor is to subtract 2 from the offset, because due to the implementation, the offset itself was offset by 2 from what it was needed to be. This allows for branches to execute exactly as they are described in the MIPS instruction set.

2.2.2 Jump

Jump does not use the same registers that branching does because there is no condition. The jump must compute the address to jump to using the instruction register and the top bits of the program counter. Hence MUX1 chooses between signals based on whether the instruction is jump, branch and other. For consistency we used the IR4 register to generate the control signal for this, as that is where branch is decided. Jumps could in theory be done without any no-op instructions following them. to keep a logical control signal, this is kept to that format. This requires two no-ops following a jump. The 26 LSBs of IR4 are shifted right twice (multiplied by 2) and the 4 MSBs of PC are used. This is how it is implemented in the MIPS architecture. Hence jumps are also delayed and four NOPs are used before the jump is taken.

2.2.3 Load/Store Word

Load word and store word both contain an offset in the instruction format which is used to compute the address of the memory address to store to or load from. This address is computed in the ALU, which chooses the register value in the Imm3 register through MUX3. For store words, DB2 (data bus 2) carries the data to be stored, and AB2 (address bus 2) carries the address to store to. For load words, only AB2 is used, and the value in memory comes back on DB3 which goes to MDR5 (the memory data register). Because load words load a value to a register, and so can results of ALU operations, MUX4 is needed, and this register write data is carried to the GPR. Because an instruction that uses the register that the load word loads data into might be used by the instructions being fetched while load word executes, and forwarding was not implemented for loads, the pipeline is stalled while the load word executes. This means, just as with branching, four NOPs are executed after a load word. This is not the case with store word as store word just writes data into memory and doesn't change register values.

2.2.4 ALU Instructions with Forwarding

Any instruction that uses the ALU uses the A and B registers (shown as A3 and B3 because EX is the third stage) which get their values based on the registers specified in the instruction (Rs and Rt fields of the instruction). This is handled by the GPR. All ALU instructions (not the same as those that use the ALU), such as add, subtract, shift, etc. use data forwarding. Observing the datapath, it can be seen that MEM to EX forwarding is

accomplishing by connecting ALU4 to MUX3 and MUX5, which supply the A and B values to the ALU. The logic for the forwarding is straightforward, as the register fields of 3.IR and 4.IR need to be compared, namely the register that 4.IR stores to (either Rd or Rt), and the ones 3.IR uses (Rs or Rt). Again, this isn't a full implementation of forwarding (as this forwarding doesn't include forwarding for loading and storing), but due to the circumstances, this was what was able to be implemented. The forwarding for loading and storing would also be straightforward, and would require WB to MEM forwarding and WB to EX forwarding.

3 Implementation

3.1 Design Structure

The top module is the implementation file, which takes all inputs as switches and buttons and displays all outputs on the LEDs of the Nexys board. The inputs for the top module are the clock and pins the FPGA board. The clock is defined by a 10ns constraint; adequate for each cycle to take enough time to complete its stage in the pipeline. The bottom button is used as a reset. The last 32 bits written to memory will be displayed on the 16 LEDs on the FGPA. This conveniently allows for any program output to be displayed by using a single store word instruction. The top button chooses whether to display the leftmost or rightmost 16 bits. A special condition was hardwired into the design for use in the demonstration and test program. When the reset (down) button is pressed, the value encoded by the switches is loaded to the position 0x1c in data memory. This will be discussed further in the test

program section.

3.2 GPR

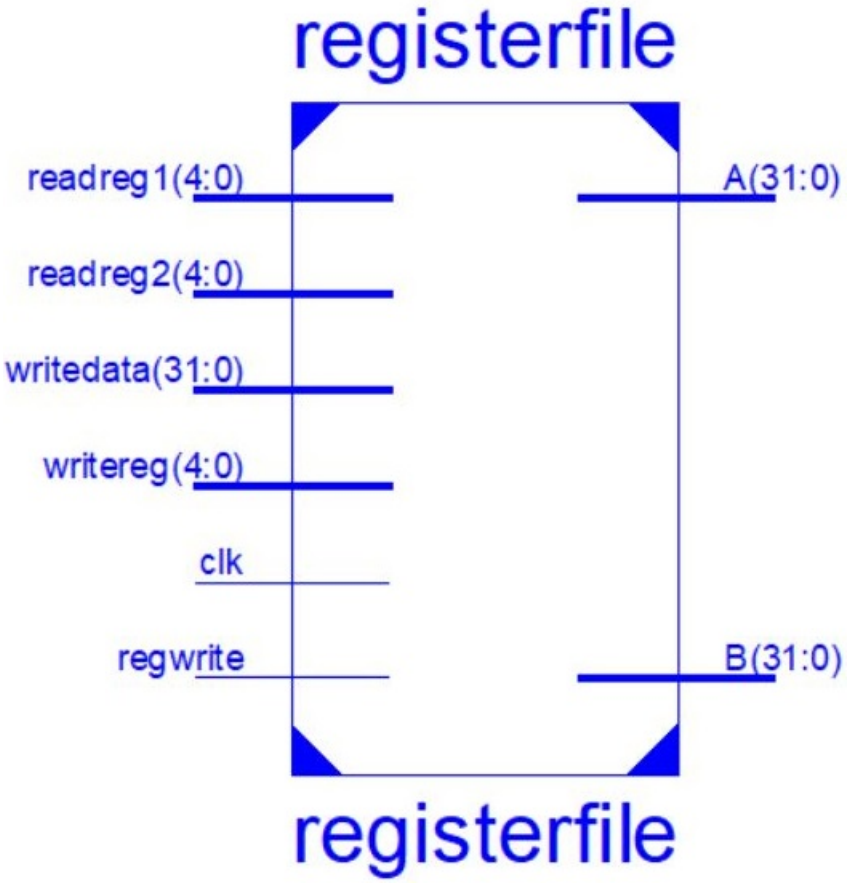


Figure 2: GPR Block Diagram

The GPR is the general purpose registers block, which holds all data needed for the program to hold while it is running. The registers will be in an array of 32, each holding 32 bits. The first two registers, R0 and R1, cannot be written to and hold the values for 0 and 1, respectively. The inputs for this file are the clock, read/write signals, and a data signal. Two registers can be read from, each indicated by a 5-bit signal, readreg1 and readreg2. One register can be written to, indicated by a 5-bit signal, writereg. The data being written to

a register is defined by the input signal writedata. To know if the register is being read or written to, we can read the 1-bit input signal regwrite. The outputs of this block are two 32-bit signals, A and B. Each of these signals are data accessed from registers, depending on the instruction called by the ALU.

3.3 ALU

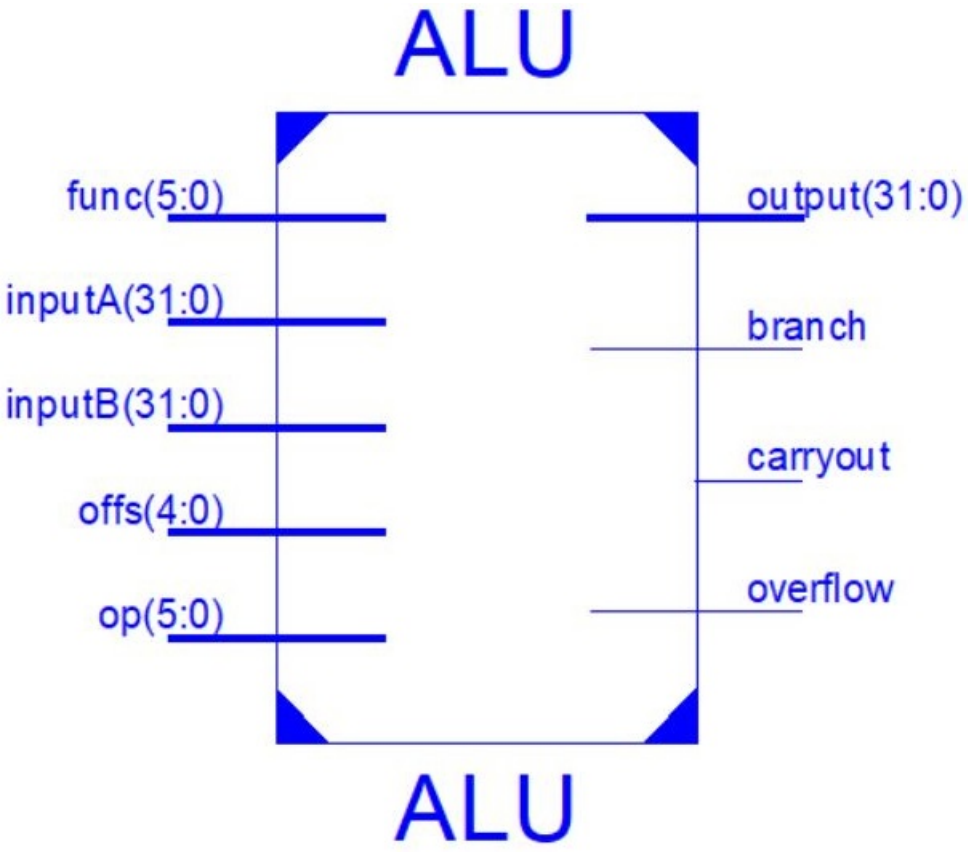


Figure 3: ALU Block Diagram

The ALU is the arithmetic logic unit, which takes care of all arithmetic instruction computing. The ALU will read the opcode (and function bits for r format instructions) for each input and perform a process based on the instruction. These include arithmetic, logical, shift, branch, jump, and load/store functions. The ALU does the bulk of the work of the

program for each type of instruction. Referring back to our instruction sets, the ALU will read the opcode to know which instruction set to look at. After reading the opcode, the funct code will tell the ALU which instruction to look at. Depending on the instruction, the ALU will work with unsigned or signed arithmetic and even account for overflow. All instructions are carried out within the ALU. Other inputs to the ALU are the A and B 32-bit input data signals. The ALU will read from each of these, depending on the instruction, but will always need to load and store. The outputs from the ALU include the branch signal. In the case of a branch statement this signal tells the control unit to either perform the branch and change PC or continue operation as usual. This method was chosen rather than setting a zero flag as most implementations do. The overflow bit was initially programmed into the ALU but we did not make use of it in the datapath. The carryout bit is similar with arithmetic operations that exceed the upper limit of 32 bits. It is also not used by the datapath.

3.4 Control Unit

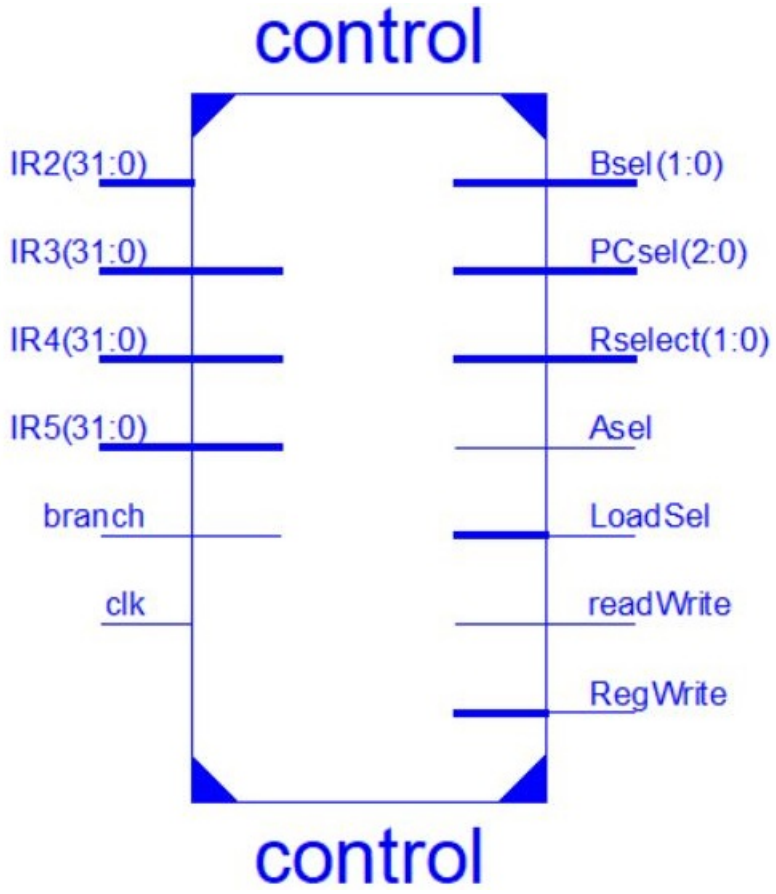


Figure 4: Control Unit Block Diagram

The control unit is necessary for gathering all necessary signals that connects the ALU to the memory and the registers. Once the CPU reads the inputs, the control unit breaks up the 32-bit signals into the several codes, which will be sequentially taken care of in the pipeline. The input signals for the control unit are the instruction signals that will be read from memory, IR2 to IR5, as well as the branch bit from the ALU. The outputs for the control unit are loadsel, regwrite, asel, readwrite, pcsel, bsel, and reselect. Each of these signals is given a value depending on the instructions being read. For example, if a register needs to be written to, we would assign a value of 1 to regwrite. Not every instruction needs

to a signal to be selected, so the control unit only turns on the signal for those instructions that are necessary.

3.5 Datapath

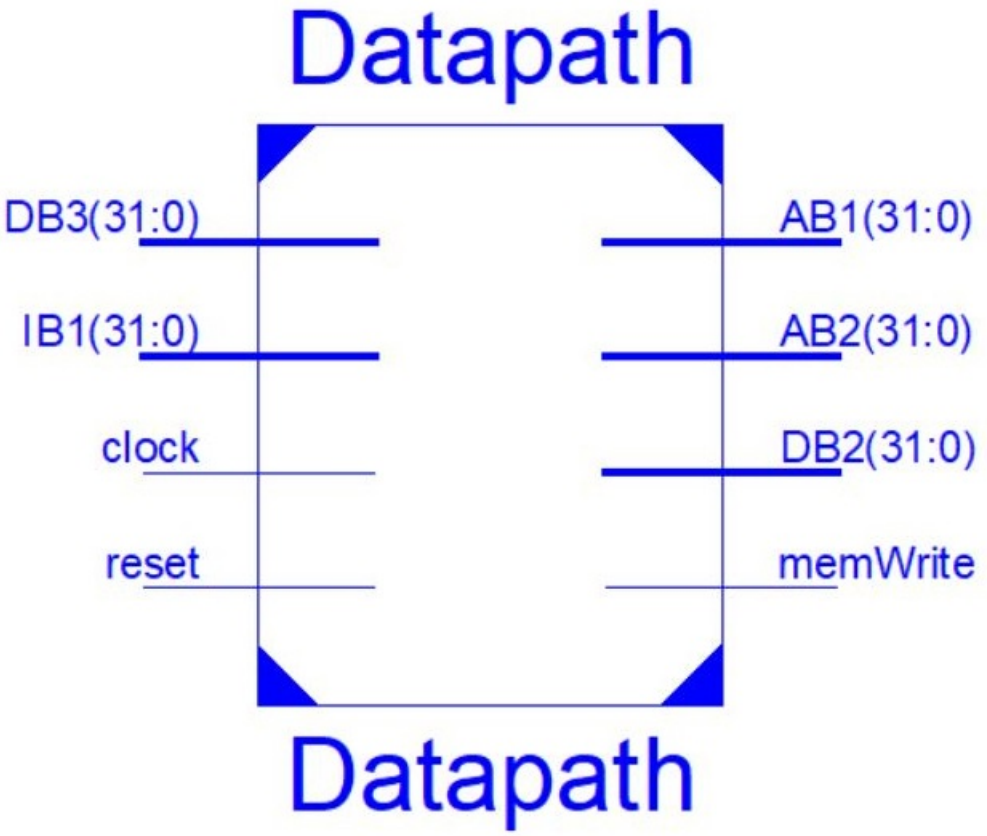


Figure 5: Datapath Block Diagram

The datapath is the upper module for the ALU, the GPR, and the Control Unit. The datapath takes the inputs from the top module and communicates it to the other parts of the CPU. The datapath itself is the entire diagram outline in the pipeline diagram, connecting different blocks of the CPU and serves a similar purpose in the hardware and code here, as well. The datapath holds the Program Counter register as well, which will hold the instructions in place, allowing the data to move throughout the pipeline at the correct times,

according to the clock. The datapath represents the entire pipeline and the data forwarding that occurs during the instruction cycle. The datapath also holds the reset for the program, which would clear all registers to the default values.

3.6 Memory Unit

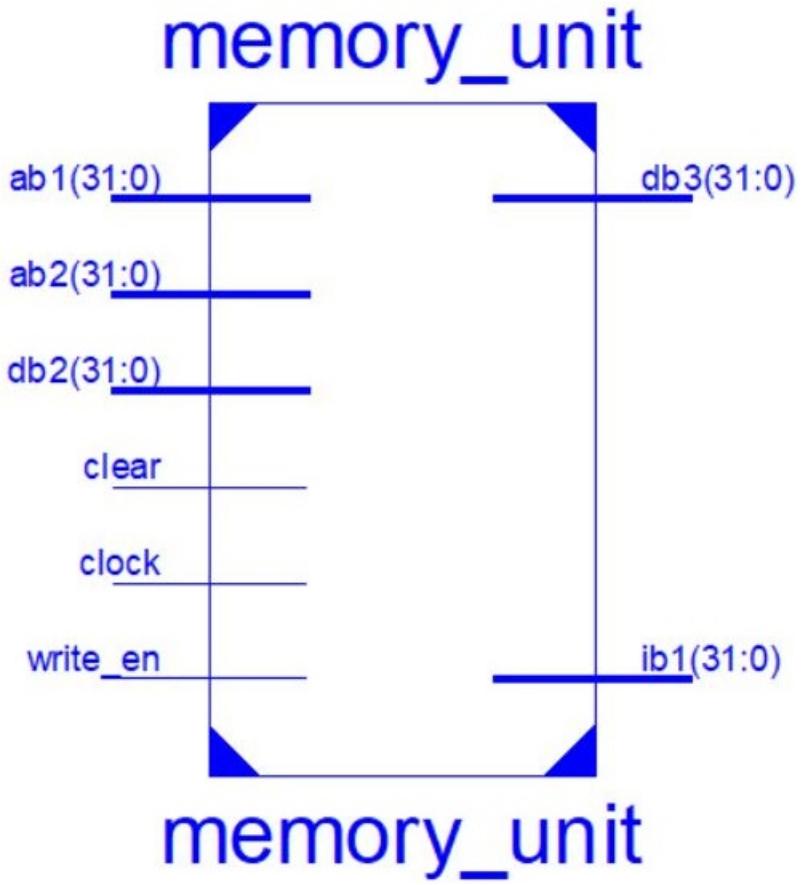


Figure 6: Memory Unit Block Diagram

The memory unit holds the instruction memory and the data memory of the program. The instruction memory is a hard-coded ROM that holds the instructions that will be fed into the pipeline. Data memory is a separate RAM that can be written to by programs. The instructions are written by the user and stored in the memory unit and can be used as

a simple program. The memory unit holds all registers for which the ALU will be storing data that is processed after an instruction has been carried out. The last piece of data written to memory will be displayed on the FPGA's LEDs. The memory unit itself does not process any of the data, but outputs data to and takes it back in from the datapath. The memory unit inputs are the instruction address (IB1), the address of the data that needs to be accessed (AB2), the data being written (DB2), a bit to enable the write, and a bit to enable a clear. The clear was disconnected when it became clear that it was wiping the data memory when the program begins. For programs that actually do need a clear, it can be reconnected. Because our test program requires a hard-coded array, the clear must remain disconnected. The outputs of the memory unit are the instruction being fetched (IB1) and the data being accessed (DB3).

4 Test Program, Simulations, and Results

4.1 Assembly Code

The test program that was implemented on the FPGA is an algorithm to count the number of instances of a particular number in an array. For example, if the array hypothetically holds 4, 5, 4, 1, 9, 5, 0, 0, , 0 and the target value is 4, the program will output 2, the number of instances it found. When implemented on the FPGA, the value in memory is shown using the LED array. The assembly code for the test program is shown below.

PC	Instruction	Fields

0	LW	R2, 1C(R0)
4	ADD	R6, R0, R0
8	ADD	R3, R0, R0
C	ADDI	R4, R0, R15
10	LW	R5, 0(R3)
14	NOP	
18	NOP	
1C	NOP	
20	NOP	
24	ADDI	R3, R3, R1
28	BEQ	R5, R2, F
2C	NOP	
30	NOP	
34	NOP	
38	NOP	
3C	BNE	R3, R4, B
40	NOP	
44	NOP	
48	NOP	
4C	NOP	

50	J	7C
54	NOP	
58	NOP	
5C	NOP	
60	NOP	
64	ADD	R6, R6, R1
68	J	10
6C	NOP	
70	NOP	
74	NOP	
78	NOP	
7C	SW	R6, 20(R0)

Table 6: Assembly Code Program

First, the value to count the occurrences of is loaded into R2 from location 0x1c in the data memory (data ROM). Then, R6 and R3 are initialized with 0. The R6 register is used to actually count the occurrences of the of the value in R2 (the desired match) in the array. The R4 register is initialized with the size of the array (minus one). In this case the array goes from data memory position 0x0 to 0x14. Because memory is organized by words and not bytes, this amounts to an array of size 21. The R3 register holds the current location in the array. It keeps getting incremented. The next value in the array gets loaded into R5.

It gets compared with the target value (R2), and if the same R6 gets incremented. It gets compared with R4 to check if the end of the array has been reached. When the end of the array is reached, the value of R6 is stored to position 0x20. Since PC keeps incrementing even after the program has ended, an extra jump instruction was added past the end of the program that jumps back to a no-op to create an infinite loop so there is no memory access error. This jump instruction is not shown, but was used when testing. The implementation has been programmed to output to the LEDs of the FPGA. the last value that has been stored to memory. Because the program ends by storing the value of R6 to memory, the number of instances of the target value in the array gets output to the LEDs. Pressing the top button displays the higher 16 bits of the value to be output, but because the array is not huge, the higher bits are always zero. In the program two no-ops are necessary after a branch/jump, and up to four could be necessary after a load word if the loaded word is used immediately following the load. For consistency, we placed four no-op instructions after each. All of data memory begins with junk random values. A few are larger than 16 bits so cannot be searched for. The ones that can be searched for and found are listed below. The inputs are the values to be searched for. The outputs are the number of occurrences in the array.

Input	Output
1	4
3	1
5	1

6	3
7	1
c	1
f	1

Table 7: Program Input and Output

4.2 Simulation Results

4.2.1 Functional Simulation

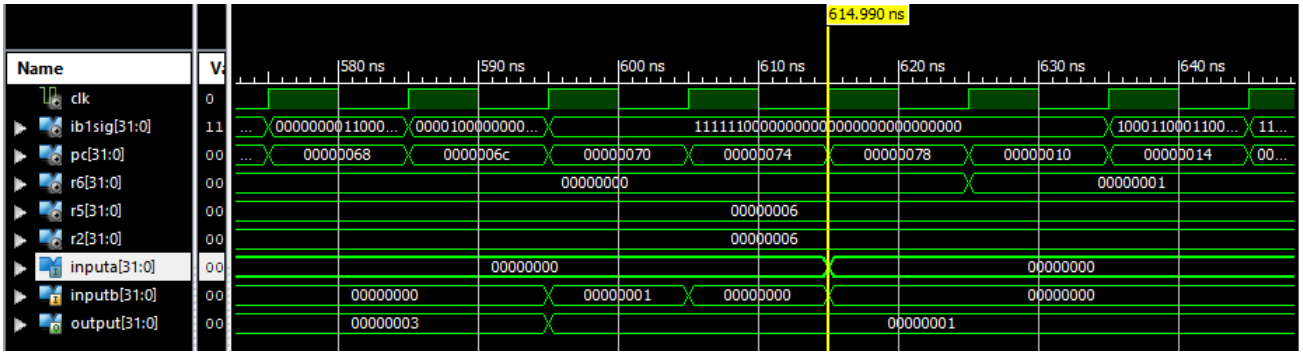


Figure 7: Execution of addition

The program is looking for a match for our target number which is stored in R2. The values of the array are loaded into R5 as we progress through it. When R2 and R5 match, that means we have found an instance of our target number. Once we find an instance of our target number, we branch to an addition to increment R6 which holds the number of instances of the target number in the array. As can be seen, the contents of R2 and R5 are both 6, which means the branch is taken, which is why the addition is executed, and R6

changes increments from 0 to 1.

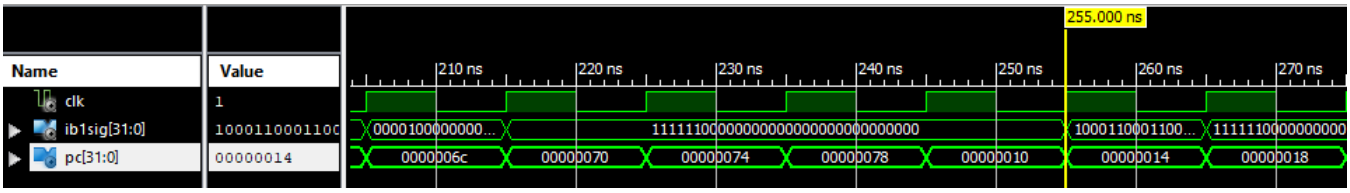


Figure 8: Execution of jump in test program

After we decide whether the number we are looking at in the array matches our target number (and increment R6 as stated previously if it does match) we have to jump back to the part of our program where we traverse through the array. As can be seen, the jump is executed when PC is 68, and then four NOPs are executed before PC returns to 10, which is when the next data value is loaded from the array (data memory).

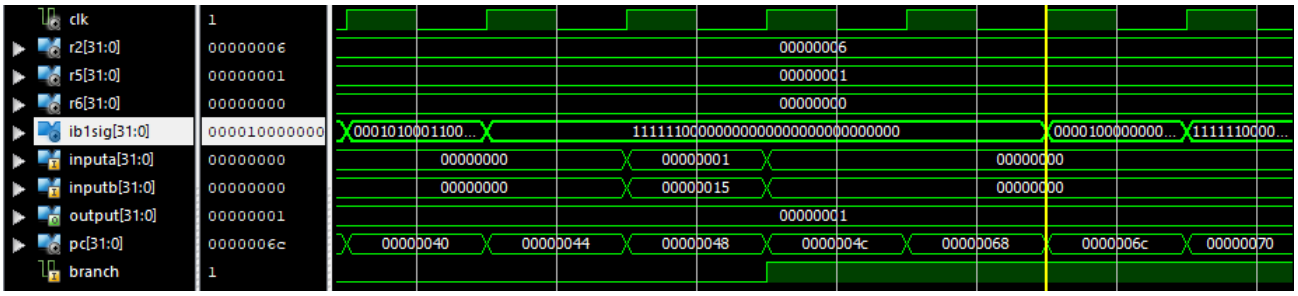


Figure 9: Execution of branch-not-equal-to

Here we are checking whether or not we have reached the end of the array. R4 contains the final location in the array and R3 contains the current location. When they are not equal we continue to traverse through the array. Otherwise we continue to the end of the program where we store the value of R6 in memory. As can be seen from the inputA and inputB (which get R3 and R4 when executing the BNE), which are not equal, the branch

should be taken (again since a BNE in executed), which is the case. Hence PC branches from 3C to 64 after executing 4 NOPS. So in this case in the simulation we are not at the end of the array so we continue to traverse it.

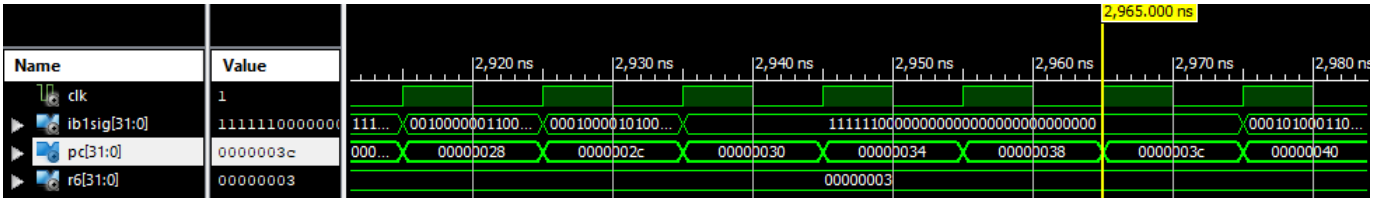


Figure 10: Results of test program

After traversing through the entire array we can view the final result of our program. As can be seen, R6 contains 3 when the test program completes execution. This is the correct result based on the data that was in the array and the desired match. The contents of memory can be seen in the code snippets section.

4.3 FPGA Implementation

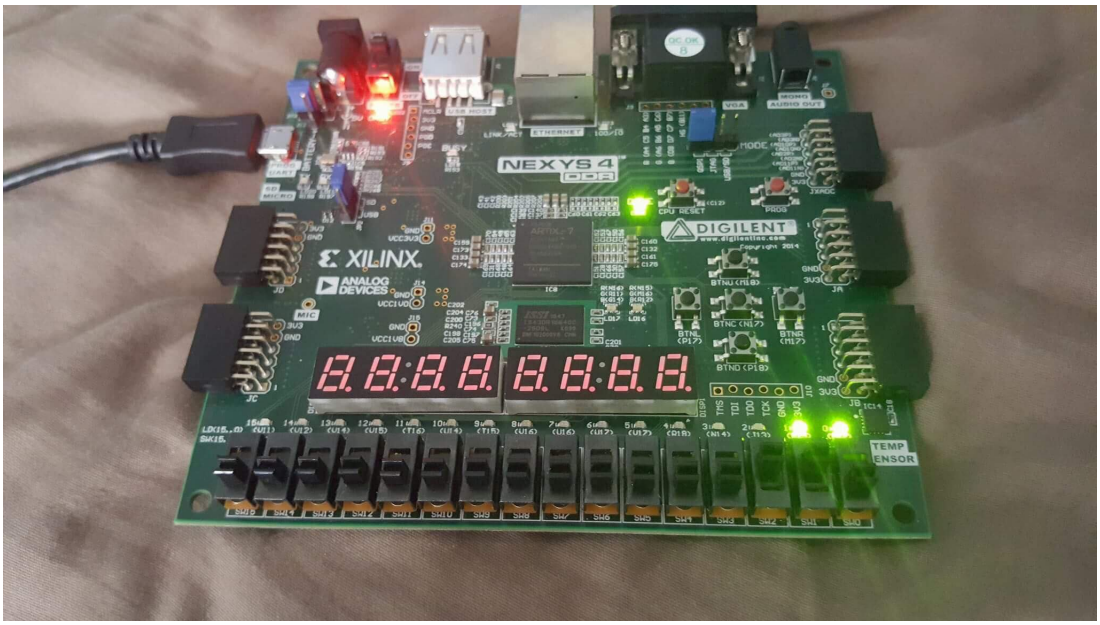


Figure 11: Processor implemented on FPGA

The CPU is shown here running the test program described in the "Assembly Code" section. The initial values in data memory locations 0 through 0x14 are being searched by this program, and the number of times the number 6 appears in the array are being counted. The program has found three instances, and that is what is output. The FPGA switches represent a 16 bit unsigned number, starting with the left-hand side being the most significant bits. When the reset is pushed, the target number to search for in the array is set according to the switches, which in this case was set to 110 in binary or 6 in decimal. The FPGA then loops through the array that is in our memory unit until it finds the number 6. Once the number 6 is found, the register R6 is incremented up by 1. Once the end of the array is reached, the value in R6 is stored into our memory unit. The LEDs correspond to that value stored in the memory unit and in this case, after running through the array, there were 3 instances of the value 6 found so 3 (11 in binary) is shown on the LEDs.

4.4 Post-Place-and-Route Report

```
Device Utilization Summary:

Slice Logic Utilization:
  Number of Slice Registers:                1,437 out of 126,800    1%
    Number used as Flip Flops:              1,338
    Number used as Latches:                    99
    Number used as Latch-thrus:                 0
    Number used as AND/OR logics:               0
  Number of Slice LUTs:                     2,517 out of  63,400    3%
    Number used as logic:                   2,458 out of  63,400    3%
      Number using O6 output only:          2,290
      Number using O5 output only:             28
      Number using O5 and O6:                 140
      Number used as ROM:                       0
```

```
    Number used as Memory:                       47 out of  19,000    1%
      Number used as Dual Port RAM:               0
      Number used as Single Port RAM:             0
      Number used as Shift Register:             47
        Number using O6 output only:             47
        Number using O5 output only:              0
        Number using O5 and O6:                   0
    Number used exclusively as route-thrus:      12
      Number with same-slice register load:      11
      Number with same-slice carry load:          1
      Number with other load:                     0

Slice Logic Distribution:
  Number of occupied Slices:                    919 out of  15,850    5%
  Number of LUT Flip Flop pairs used:         2,660
    Number with an unused Flip Flop:          1,236 out of   2,660   46%
    Number with an unused LUT:                  143 out of   2,660    5%
    Number of fully used LUT-FF pairs:        1,281 out of   2,660   48%
    Number of slice register sites lost
      to control set restrictions:                0 out of 126,800    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                        163 out of     210   77%
    IOB Flip Flops:                              32

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:                   0 out of     135    0%
  Number of RAMB18E1/FIFO18E1s:                   0 out of     270    0%
  Number of BUFG/BUFGCTRLs:                       4 out of      32   12%
    Number used as BUFGs:                         4
    Number used as BUFGCTRLs:                     0
  Number of IDELAYE2/IDELAYE2_FINEDELAYs:         0 out of     300    0%
  Number of ILOGICE2/ILOGICE3/ISERDESE2s:         0 out of     300    0%
  Number of ODELAYE2/ODELAYE2_FINEDELAYs:         0
  Number of OLOGICE2/OLOGICE3/OSERDESE2s:        32 out of     300   10%
    Number used as OLOGICE2s:                    32
    Number used as OLOGICE3s:                     0
```

```
    Number used as OSERDESE2s:                    0
  Number of PHASER_IN/PHASER_IN_PHYs:             0 out of      24    0%
  Number of PHASER_OUT/PHASER_OUT_PHYs:           0 out of      24    0%
  Number of BSCANs:                               0 out of       4    0%
  Number of BUFHCEs:                              0 out of      96    0%
  Number of BUFRs:                                0 out of      24    0%
  Number of CAPTUREs:                             0 out of       1    0%
  Number of DNA_PORTs:                            0 out of       1    0%
  Number of DSP48E1s:                             0 out of     240    0%
  Number of EFUSE_USRs:                           0 out of       1    0%
  Number of FRAME_ECCs:                           0 out of       1    0%
  Number of IBUFDS_GTE2s:                         0 out of       4    0%
  Number of ICAPs:                                0 out of       2    0%
  Number of IDELAYCTRLs:                          0 out of       6    0%
  Number of IN_FIFOs:                             0 out of      24    0%
  Number of MMCME2_ADVs:                          0 out of       6    0%
  Number of OUT_FIFOs:                            0 out of      24    0%
  Number of PCIE_2_1s:                            0 out of       1    0%
  Number of PHASER_REFs:                          0 out of       6    0%
  Number of PHY_CONTROLs:                         0 out of       6    0%
  Number of PLLE2_ADVs:                           0 out of       6    0%
  Number of STARTUPs:                             0 out of       1    0%
  Number of XADCs:                                0 out of       1    0%


Overall effort level (-ol):   High
Router effort level (-rl):    High


Starting initial Timing Analysis.  REAL time: 18 secs
Finished initial Timing Analysis.  REAL time: 18 secs

Starting Router


Phase  1  : 13105 unrouted;      REAL time: 20 secs

Phase  2  : 11920 unrouted;      REAL time: 20 secs

Phase  3  : 3011 unrouted;      REAL time: 26 secs

Phase  4  : 3079 unrouted; (Par is working to improve performance)     REAL time:
 ↪  33 secs

Updating file: Datapath.ncd with current fully routed design.
```

```
104
105 Phase  5  : 0 unrouted; (Par is working to improve performance)     REAL time: 42
    ↪  secs
106
107 Phase  6  : 0 unrouted; (Par is working to improve performance)     REAL time: 43
    ↪  secs
108
109 Phase  7  : 0 unrouted; (Par is working to improve performance)     REAL time: 43
    ↪  secs
110
111 Phase  8  : 0 unrouted; (Par is working to improve performance)     REAL time: 43
    ↪  secs
112
113 Phase  9  : 0 unrouted; (Par is working to improve performance)     REAL time: 45
    ↪  secs
114 Total REAL time to Router completion: 45 secs
115 Total CPU time to Router completion: 44 secs
116
117 Partition Implementation Status
118 -------------------------------
119
120   No Partitions were found in this design.
121
122 -------------------------------
123
124 Generating "PAR" statistics.
125 INFO:Par:459 - The Clock Report is not displayed in the non timing-driven mode.
126 Timing Score: 16726 (Setup: 16726, Hold: 0)
127
128 Asterisk (*) preceding a constraint indicates it was not met.
129    This may be due to a setup or hold violation.
130
131 ----------------------------------------------------------------------------------
132   Constraint    |    Check    | Worst Case |  Best Case | Timing |   Timing
133                 |             |    Slack   | Achievable | Errors |    Score
134 ----------------------------------------------------------------------------------
135   Autotimespec  | SETUP       |         N/A|     3.609ns|     N/A|       16726
136  constraint for | HOLD        |     0.066ns|            |       0|           0
137    clock net    |             |            |            |        |
138   clock_BUFGP   |             |            |            |        |
139 ----------------------------------------------------------------------------------
```

5 Conclusion

The processor we set out to make is a 32-bit processor with purpose of interpreting assembly instructions from the MIPS I instruction set written in VHDL and implemented on a Diligent Nexys 4 FPGA. After making a high level design of the processor, the processor was then coded using the Xilinx ISE and Vivado design suites. After having been coded, we proceeded to go through unit testing of the individual pieces of our processor. Once we were confident in the ability of our processor to run instructions, it was then programmed onto an FPGA with a test program to search through an array and count the number of instances of a target number. The program was completed successfully, showing that our processor worked properly. Since the processor is a general purpose CPU, it can be run for a variety of tasks, depending on what is loaded into the instruction memory. While the processor does work, there are a number of improvements that can be made to improve the efficiency such as improving the data forwarding so fewer no operation instructions would required. More instructions can be added also which would make our processor closer to some of the more current implementation of the MIPS architecture, such as MIPS V.

6 Bibliography

MIPS architecture. (2018, May 16). Retrieved from https://en.wikipedia.org/wiki/MIPS_architecture

Hadimioglu, H. (2017, November 06). Pipelined EMY CPU [Pdf]. Brooklyn: Tandon School of Engineering.

MIPS Instruction Reference. (n.d.). Retrieved from

http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

7 Appendix

All of the code can be found at https://github.com/lienwyatt/32-bit-MIPS-processor

7.1 Implementation.vhd

```
----------------------------------------------------------------------------------
--code for top module
----------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY CPU IS
PORT(
      clk    : IN    STD_LOGIC;--clock
      btnD    : IN    STD_LOGIC; --left bits/right bits
      btnU    : IN    STD_LOGIC; --reset
      led    : OUT    STD_LOGIC_VECTOR(15 DOWNTO 0);
  sw : in std_logic_vector(15 downto 0) := "1111111111111111"
);
END CPU;

ARCHITECTURE behavioral OF CPU IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT Datapath
    PORT(
         AB1 : OUT  std_logic_vector(31 downto 0);
         IB1 : IN  std_logic_vector(31 downto 0);
         AB2 : OUT  std_logic_vector(31 downto 0);
         DB2 : OUT  std_logic_vector(31 downto 0);
         DB3 : IN  std_logic_vector(31 downto 0);
         memWrite : OUT std_logic;
         reset : IN  std_logic;
         clock : IN  std_logic
```

```
35          );
36      END COMPONENT;
37
38      component memory_unit
39          port(
40          ab1: in std_logic_vector(31 downto 0); -- pc (address of instruction)
41          ib1: out std_logic_vector(31 downto 0); -- instruction fetched from
             ↪  memory
42
43      switches: in std_logic_vector(15 downto 0);
44
45          ab2: in std_logic_vector(31 downto 0); --  address of data (to be fetched
             ↪  or written to)
46          db2: in std_logic_vector(31 downto 0); -- carries the data to be written
             ↪  to memory
47          write_en : in std_logic; -- write enable
48          db3: out std_logic_vector(31 downto 0); -- data out
49
50          clear: in std_logic; -- clear bit (for data memory)
51          clock : in std_logic -- clock signal
52          );
53          end component;
54
55
56      --Inputs
57      signal IB1sig : std_logic_vector(31 downto 0) := (others => '0');
58      signal DB3 : std_logic_vector(31 downto 0) := (others => '0');
59      signal reset : std_logic;
60      signal clock : std_logic := '0';
61
62        --Outputs
63      signal AB1 : std_logic_vector(31 downto 0);
64      signal AB2 : std_logic_vector(31 downto 0);
65      signal DB2 : std_logic_vector(31 downto 0);
66       signal MemWrite: std_logic;
67       signal led_sig : std_logic_vector(31 downto 0);
68       signal output  : std_logic_vector(31 downto 0);
69      -- Clock period definitions
70      constant clock_period : time := 10 ns;
71
72  BEGIN
73  clock<=clk;
74  reset<=btnU;
75       -- Instantiate the Unit Under Test (UUT)
76      uut: Datapath PORT MAP (
```

```
            AB1 => AB1,
            IB1 => IB1sig,
            AB2 => AB2,
            DB2 => DB2,
            DB3 => DB3,
            memWrite => MemWrite,
            reset => reset,
            clock => clock
          );

     memory: memory_unit PORT MAP(
     ab1 => AB1,
     ib1 => IB1sig,
   switches => sw,
     ab2 => AB2,
     db2 => DB2,
     db3 => DB3,
     write_en => MemWrite,
     clear =>reset,
     clock => clock
     );
process(memWrite, clk)--THIS PRINTS TO LEDS THE LAST THING TO BE WRITTEN TO
 ↪  MEMORY
begin
if (memWrite='1' AND clk' event AND clk = '1') then
    output<=DB3;
end if;
end process;


process(btnU)
begin
if (btnD='1') then
    led<=output(31 downto 16);
else
    led<=output(15 downto 0);
end if;
end process;

END;
```

7.2 Datapath.vhd

```
----------------------------------------------------------------------------
-- Data Path
```

```
-- connects the GPR, ALU and control unit together
----------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Datapath is

    Port (
    AB1: out std_logic_vector(31 downto 0);
    IB1: in std_logic_vector(31 downto 0);
    AB2: out std_logic_vector(31 downto 0);
    DB2: out std_logic_vector(31 downto 0);
    DB3: in std_logic_vector(31 downto 0);
  memWrite: out std_logic;
  reset:in std_logic; -- used for resetting pc to first address
    clock: in std_logic);
end Datapath;

architecture Behavioral of Datapath is
    signal NPC2: std_logic_vector(31 downto 0);--PC2 register
    signal NPC3: std_logic_vector(31 downto 0);--PC3 register

    signal TA4: std_logic_vector(31 downto 0);-- used for branches

    signal TA: std_logic_vector(31 downto 0);-- used for calculating branches

    signal A3: std_logic_vector(31 downto 0);--holds the value for A

    signal B3: std_logic_vector(31 downto 0);--holds the value of B
    signal B4: std_logic_vector(31 downto 0);

    signal IMM3: std_logic_vector(31 downto 0);--hold the immediate value

    signal IR2: std_logic_vector(31 downto 0); --instruction registers
    signal IR3: std_logic_vector(31 downto 0);
    signal IR4: std_logic_vector(31 downto 0);
    signal IR5: std_logic_vector(31 downto 0);

    signal ALU4: std_logic_vector(31 downto 0); --alu output
    signal ALU5: std_logic_vector(31 downto 0);

  signal PC: std_logic_vector(31 downto 0);

```

```
47      signal MDR5: std_logic_vector(31 downto 0);-- values from muxes and control
        ↪  signals
48      signal branch4: std_logic;
49      signal readWrite: std_logic;
50
51      signal pc_count: std_logic_vector (31 downto 0) :=
        ↪  "00000000000000000000000000000000";
52
53
54
55      signal Rs: std_logic_vector(4 downto 0);--gpr signals
56      signal Rd: std_logic_vector(4 downto 0);
57      signal Rt: std_logic_vector(4 downto 0);
58      signal A: std_logic_vector(31 downto 0);
59      signal B: std_logic_vector(31 downto 0);
60      signal regwrite: std_logic;
61
62      component registerfile
63       port(
64      clk: in std_logic;
65          readreg1: in std_logic_vector(4 downto 0);
66          readreg2: in std_logic_vector(4 downto 0);
67          writereg: in std_logic_vector(4 downto 0);
68          writedata: in std_logic_vector(31 downto 0);
69          regwrite: in std_logic;
70          A: out std_logic_vector(31 downto 0);
71          B: out std_logic_vector(31 downto 0)
72          );
73      end component;
74
75       -- alu signals
76       signal aluoutput: std_logic_vector(31 downto 0);--alu signals
77       signal aluoverflow: std_logic;
78       signal alubranch: std_logic;
79       signal alucarry: std_logic;
80
81       component ALU
82       port(
83       inputA: in std_logic_vector(31 downto 0);
84       inputB: in std_logic_vector(31 downto 0);
85       op: in std_logic_vector(5 downto 0);
86      offs: in std_logic_vector(4 downto 0);--should this exist
87       func: in std_logic_vector(5 downto 0);
88       output: out std_logic_vector(31 downto 0);
89       overflow: out std_logic;
```

```
90       carryout: out std_logic;
91       branch: out std_logic
92       );
93      end component;
94
95      --muxes for use in conjunction with control signals
96      signal mux1: std_logic_vector(31 downto 0);--muxes
97      signal mux2: std_logic_vector(31 downto 0);
98      signal mux3: std_logic_vector(31 downto 0);
99      signal mux4: std_logic_vector(31 downto 0);
100     signal mux5: std_logic_vector(31 downto 0);
101
102
103
104     signal PCsel: std_logic_vector(2 downto 0);--control unit signals
105     signal Bsel: std_logic_vector(1 downto 0);
106     signal Rselect: std_logic_vector(1 downto 0);
107     signal Loadsel: std_logic;
108     signal Rsel: std_logic_vector(4 downto 0);
109     signal Asel: std_logic;
110
111
112
113   component control
114   port(
115   branch : in std_logic;
116   clk : in std_logic;
117   IR2 : in std_logic_vector(31 downto 0);
118   IR3 : in std_logic_vector(31 downto 0);
119   IR4 : in std_logic_vector(31 downto 0);
120   IR5 : in std_logic_vector(31 downto 0);
121   PCsel : out std_logic_vector (2 downto 0);
122   Bsel : out std_logic_vector(1 downto 0);
123   Asel : out std_logic;
124   LoadSel : out std_logic;
125   Rselect : out std_logic_vector(1 downto 0);
126   RegWrite: out std_logic;
127   readWrite: out std_logic
128  );
129  end component;
130
131 begin
132
133 gpr: registerfile port map(
134   clk=>clock,
```

```
135      readreg1=>Rs,
136      readreg2=>Rt,
137      writereg=>Rsel,
138      A=>A,
139      B=>B,
140      writedata=>mux4,
141      regwrite=>regwrite
142      );
143
144
145  Arithmetic_logic_unit: ALU port map(
146      inputA=>mux5,
147      inputB=>mux3,
148      op=>IR3(31 downto 26),
149      func=>IR3(5 downto 0),
150    offs=>IR3(10 downto 6),--should this exist
151      output=>aluoutput,
152      overflow=> aluoverflow,
153      carryout=> alucarry,
154      branch=>alubranch
155      );
156
157  Control_unit: control port map(
158      branch=>branch4,
159   clk=>clock,
160   IR2=>IR2,
161   IR3=>IR3,
162   IR4=>IR4,
163   IR5=>IR5,
164   PCSel=>PCSel,
165   Asel=>Asel,
166   Bsel=>Bsel,
167   Rselect=>Rselect,
168   LoadSel=>LoadSel,
169   RegWrite=>regwrite,
170   readWrite => readWrite
171  );
172
173  DB2<=B4;
174  AB2<=ALU4;
175  AB1<=PC;
176  Rs<=IR2(25 downto 21);
177  Rt<=IR2(20 downto 16);
178  memWrite<= readWrite;
179  pc_count <= PC + "00000000000000000000000000000100";
```

```
180
181 with PCsel select mux1<=
182 pc_count when "001",
183 --jump to address
184 TA4 when "010",
185 --branch
186 PC(31 downto 28) & IR4(25 downto 0) & "00" when "011",
187 pc_count when others;
188 TA(31 downto 2)<=(IMM3(29 downto 0) - "10");--pc = imm x 4
189 TA(1 downto 0)<= "00";
190 process(clock, reset)
191 begin
192 if(clock' event and clock='1') then
193    branch4<=alubranch;
194    A3<=A;
195    B3<=B;
196    B4<=B3;
197    NPC2<=mux1;
198    MDR5<=DB3;
199    NPC3<=NPC2;
200    ALU4<=aluoutput;
201    ALU5<=ALU4;
202    IR2<=IB1;
203    IR3<=IR2;
204    IR4<=IR3;
205    IR5<=IR4;
206    TA4<=TA+NPC3;
207
208    IMM3(15 downto 0)<=IR2(15 downto 0);--sign extend
209    if (IR2(15)='1') then
210        IMM3(31 downto 16)<="1111111111111111";
211    else
212        IMM3(31 downto 16)<="0000000000000000";
213    end if;
214 end if;
215 end process;
216
217 --pc reg
218 process (clock, reset)
219 begin
220  if(reset = '0') then
221   if(clock' event and clock='1') then
222    PC<=mux1;
223   end if;
224  else -- reset = 1
```

```
  PC<= x"00000000";
 end if;
end process;

with Loadsel select mux4 <=
ALU5 when '1',
MDR5 when others;

with Rselect select Rsel <=
    IR5(15 downto 11) when "01",
    IR5(20 downto 16) when "00",
  "00000" when others;


with Asel select mux5<=
    ALU4 when '1',
    A3 when others;


with Bsel select mux3 <=
    B3 when "01",
    ALU4 when "11",
    IMM3 when others;

end Behavioral;
```

7.3 GPR.vhd

```
----------------------------------------------------------------------------------
-- GPR
-- contains all 32 general purpose registers that can be written to as well as
 ↪  read
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity registerfile is
    port(
  clk: in std_logic;
    readreg1: in std_logic_vector(4 downto 0);
    readreg2: in std_logic_vector(4 downto 0);
    writereg: in std_logic_vector(4 downto 0);
    writedata: in std_logic_vector(31 downto 0);
```

```
17      regwrite: in std_logic;
18      A: out std_logic_vector(31 downto 0);
19      B: out std_logic_vector(31 downto 0)
20      );
21
22  --  Port ( );
23  end registerfile;
24
25  architecture Behavioral of registerfile is --contains 32 registers
26      signal R0 : std_logic_vector(31 downto 0);
27      signal R1 : std_logic_vector(31 downto 0);
28      signal R2 : std_logic_vector(31 downto 0);
29      signal R3 : std_logic_vector(31 downto 0);
30      signal R4 : std_logic_vector(31 downto 0);
31      signal R5 : std_logic_vector(31 downto 0);
32      signal R6 : std_logic_vector(31 downto 0);
33      signal R7 : std_logic_vector(31 downto 0);
34      signal R8 : std_logic_vector(31 downto 0);
35      signal R9 : std_logic_vector(31 downto 0);
36      signal R10 : std_logic_vector(31 downto 0);
37      signal R11: std_logic_vector(31 downto 0);
38      signal R12 : std_logic_vector(31 downto 0);
39      signal R13 : std_logic_vector(31 downto 0);
40      signal R14 : std_logic_vector(31 downto 0);
41      signal R15 : std_logic_vector(31 downto 0);
42      signal R16 : std_logic_vector(31 downto 0);
43      signal R17 : std_logic_vector(31 downto 0);
44      signal R18 : std_logic_vector(31 downto 0);
45      signal R19 : std_logic_vector(31 downto 0);
46      signal R20 : std_logic_vector(31 downto 0);
47      signal R21 : std_logic_vector(31 downto 0);
48      signal R22 : std_logic_vector(31 downto 0);
49      signal R23 : std_logic_vector(31 downto 0);
50      signal R24 : std_logic_vector(31 downto 0);
51      signal R25 : std_logic_vector(31 downto 0);
52      signal R26 : std_logic_vector(31 downto 0);
53      signal R27 : std_logic_vector(31 downto 0);
54      signal R28 : std_logic_vector(31 downto 0);
55      signal R29 : std_logic_vector(31 downto 0);
56      signal R30 : std_logic_vector(31 downto 0);
57      signal R31 : std_logic_vector(31 downto 0);
58  begin
59  --R0 and R1 are constant values
60  R0<="00000000000000000000000000000000";
61  R1<="00000000000000000000000000000001";
```

```
process(readreg1, readreg2, regwrite)
begin
--gets the value for the A input for the ALU
case readreg1 is
    when "00000"=> A<=R0;
    when "00001"=> A<=R1;
    when "00010"=> A<=R2;
    when "00011"=> A<=R3;
    when "00100"=> A<=R4;
    when "00101"=> A<=R5;
    when "00110" =>A<=R6;
    when "00111"=> A<=R7;
    when "01000"=> A<=R8;
    when "01001"=> A<=R9;
    when "01010"=> A<=R10;
    when "01011"=> A<=R11;
    when "01100"=> A<=R12;
    when "01101"=> A<=R13;
    when "01110"=> A<=R14;
    when "01111"=> A<=R15;
    when "10000"=> A<=R16;
    when "10001"=> A<=R17;
    when "10010"=> A<=R18;
    when "10011"=> A<=R19;
    when "10100"=> A<=R20;
    when "10101"=> A<=R21;
    when "10110"=> A<=R22;
    when "10111"=> A<=R23;
    when "11000"=> A<=R24;
    when "11001"=> A<=R25;
    when "11010"=> A<=R26;
    when "11011"=> A<=R27;
    when "11100"=> A<=R28;
    when "11101"=> A<=R29;
    when "11110"=> A<=R30;
    when "11111"=> A<=R31;
    when others =>
end case;
-- gets the value for the B input of the ALU
case readreg2 is
        when "00000"=> B<=R0;
        when "00001"=> B<=R1;
        when "00010"=> B<=R2;
        when "00011"=> B<=R3;
        when "00100"=> B<=R4;
```

```
107         when "00101"=> B<=R5;
108         when "00110"=> B<=R6;
109         when "00111"=> B<=R7;
110         when "01000"=> B<=R8;
111         when "01001"=> B<=R9;
112         when "01010"=> B<=R10;
113         when "01011"=> B<=R11;
114         when "01100"=> B<=R12;
115         when "01101"=> B<=R13;
116         when "01110"=> B<=R14;
117         when "01111"=> B<=R15;
118         when "10000"=> B<=R16;
119         when "10001"=> B<=R17;
120         when "10010"=> B<=R18;
121         when "10011"=> B<=R19;
122         when "10100"=> B<=R20;
123         when "10101"=> B<=R21;
124         when "10110"=> B<=R22;
125         when "10111"=> B<=R23;
126         when "11000"=> B<=R24;
127         when "11001"=> B<=R25;
128         when "11010"=> B<=R26;
129         when "11011"=> B<=R27;
130         when "11100"=> B<=R28;
131         when "11101"=> B<=R29;
132         when "11110"=> B<=R30;
133         when "11111"=> B<=R31;
134         when others =>
135     end case;
136 end process;
137
138 process(writereg, regwrite, writedata, clk)
139 begin
140 --synchronous write to the GPRs
141 if (regwrite='1' and clk' event and clk='1') then
142 case writereg is
143     when "00010"=> R2<=writedata;
144     when "00011"=> R3<=writedata;
145     when "00100"=> R4<=writedata;
146     when "00101"=> R5<=writedata;
147     when "00110"=> R6<=writedata;
148     when "00111"=> R7<=writedata;
149     when "01000"=> R8<=writedata;
150     when "01001"=> R9<=writedata;
151     when "01010"=> R10<=writedata;
```

```
152      when "01011"=> R11<=writedata;
153      when "01100"=> R12<=writedata;
154      when "01101"=> R13<=writedata;
155      when "01110"=> R14<=writedata;
156      when "01111"=> R15<=writedata;
157      when "10000"=> R16<=writedata;
158      when "10001"=> R17<=writedata;
159      when "10010"=> R18<=writedata;
160      when "10011"=> R19<=writedata;
161      when "10100"=> R20<=writedata;
162      when "10101"=> R21<=writedata;
163      when "10110"=> R22<=writedata;
164      when "10111"=> R23<=writedata;
165      when "11000"=> R24<=writedata;
166      when "11001"=> R25<=writedata;
167      when "11010"=> R26<=writedata;
168      when "11011"=> R27<=writedata;
169      when "11100"=> R28<=writedata;
170      when "11101"=> R29<=writedata;
171      when "11110"=> R30<=writedata;
172      when "11111"=> R31<=writedata;
173      when others =>
174  end case;
175  end if;
176  end process;
177  end Behavioral;
```

7.4 ALU.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.numeric_std.all;
4  use ieee.std_logic_unsigned.all;
5
6  -- Uncomment the following library declaration if using
7  -- arithmetic functions with Signed or Unsigned values
8  --use IEEE.NUMERIC_STD.ALL;
9
10 -- Uncomment the following library declaration if instantiating
11 -- any Xilinx primitives in this code.
12 --library UNISIM;
13 --use UNISIM.VComponents.all;
14
15 entity ALU is
16 port(
```

```
17  inputA, inputB: in std_logic_vector(31 downto 0);
18  op: in std_logic_vector(5 downto 0);
19  offs: in std_logic_vector(4 downto 0);
20  func: in std_logic_vector(5 downto 0);
21  output: out std_logic_vector(31 downto 0);
22  overflow: out std_logic;
23  carryout: out std_logic;
24  branch: out std_logic
25  );
26  end ALU;
27
28  architecture Behavioral of ALU is
29  signal offset: std_logic_vector(4 downto 0);
30  signal opcode: std_logic_vector(5 downto 0);
31  signal funct: std_logic_vector(5 downto 0);
32  signal Aunsigned: std_logic_vector(32 downto 0);
33  signal Bunsigned: std_logic_vector(32 downto 0);
34  signal Asigned: std_logic_vector(32 downto 0);
35  signal Bsigned: std_logic_vector(32 downto 0);
36  signal output_sig: std_logic_vector(32 downto 0);
37  signal Asigned_tmp, Bsigned_tmp :std_logic_vector(32 downto 0);
38
39  begin
40  offset<= offs;
41  opcode <= op;
42  funct <= func;
43  Aunsigned <= '0' & inputA;
44  Bunsigned <= '0' & inputB;
45  Asigned <= '0' & inputA;
46  Bsigned <= '0' & inputB;
47  -- determines what instruction is being run based on its opcode
48  process(opcode, Aunsigned, Asigned, Bsigned, Bunsigned, funct, offset)
49  begin
50  case opcode is
51  when "000000" => -- arithmetic operations (opcode = 00)
52  case funct is --in the case the opcode is 000000, funct bits are used to
    ↪  determine the instruction
53  when "100000" =>  --(signed addition)
54  output_sig <= std_logic_vector(signed(Asigned) + signed(Bsigned));
55  if Asigned(31) /= Bsigned(31) then -- adding a positive and negative number, cant
    ↪  have overflow
56  overflow <= '0';
57  else
```

```
58 if output_sig(31) /= Asigned(31) then --both numbers are positive or both numbers
    ↪ are negative (output MSB must match input MSBs which are the same since boeth
    ↪ are of same sign)
59  overflow <= '1';
60 else overflow <= '0';
61 end if;
62 end if;
63 when "100001" => --(unsigned addition)
64 output_sig <= std_logic_vector(unsigned(Aunsigned) + unsigned(Bunsigned));
65 if output_sig(32) = '1' then
66 overflow <= '1';
67 else
68 overflow <= '0';
69 end if;
70 when "100010" => --(subtraction)
71 output_sig <= std_logic_vector(signed(Asigned) - signed(Bsigned));
72 if(Asigned(31) = Bsigned(31)) then--overflow handling. A and B have different
    ↪ signs
73 overflow <= '0';
74 elsif(Asigned(31) = '1') then -- A is negative, so B must be positive according
    ↪ to previous if (above the else)
75 Asigned_tmp <= not(Asigned) + '1'; -- turns A positive
76  if(Asigned_tmp > Bsigned) then --if absolute value of A is > B, and A is
     ↪ negative, output must be negative
77   if(output_sig(31) /= '1') then
78    overflow <= '1';
79   else
80    overflow <= '0';
81   end if;
82  end if;
83 else -- A is positive, B is negative (meaning we did A + B, where both A and B
    ↪ are positive)
84  if(output_sig(32) /= '1') then--checking the carry-out bit
85   overflow <= '1';
86  else
87   overflow <= '0';
88  end if;
89 end if;
90
91 when "100011" => --(unsigned subtraction)
92 output_sig <= std_logic_vector(unsigned(Aunsigned) + unsigned(Bunsigned));
93 if (Aunsigned >= Bunsigned) then
94 overflow <= '0';
95 else
96 overflow<= '1';
```

```
97  end if;
98
99  when "100100" => -- (and)
100 output_sig <= (Aunsigned) AND (Bunsigned);
101 when "100101" => -- (or)
102 output_sig <= (Aunsigned) OR (Bunsigned);
103 when "100110" => --(xor)
104 output_sig <= (Aunsigned) XOR (Bunsigned);
105 when "100111" => --(nor)
106 output_sig <= (Aunsigned) NOR (Bunsigned);
107
108 when "101010" => --(set on less than)
109 if(Asigned(31) = Bsigned(31)) then
110 if(Asigned(31)='0') then
111  if(Asigned<Bsigned) then
112   output_sig<= "00000000000000000000000000000001";
113  else
114   output_sig<= "00000000000000000000000000000000";
115  end if;
116 else -- both A and B are negative, abs values must be compared
117  Asigned_tmp  <= not(Asigned) + '1';
118  Bsigned_tmp  <= not(Bsigned) + '1';
119  if(Asigned_tmp > Bsigned_tmp) then -- if A is a smaller negative
     ↪  number,althought abs(A) < abs(B), A is > B.
120   output_sig<= "00000000000000000000000000000001";
121  else
122   output_sig<= "00000000000000000000000000000000";
123  end if;
124 end if;
125 end if;
126
127 when "101011"=> --(set on less than unsigned)
128 if(Aunsigned < Bunsigned) then
129    output_sig<= "00000000000000000000000000000001";
130 else
131    output_sig<= "00000000000000000000000000000000";
132 end if;
133 when "000000"=>--(shift left logical)
134 --left unsigned shift using an immediate value
135 case offset is
136 when "00000" => output_sig<= Bunsigned;
137 when "00001" => output_sig<= Bunsigned(31 downto 0) & "0";
138 when "00010" => output_sig<= Bunsigned(30 downto 0) & "00";
139 when "00011" => output_sig<= Bunsigned(29 downto 0) & "000";
140 when "00100" => output_sig<= Bunsigned(28 downto 0) & "0000";
```

```
141  when "00101" => output_sig<= Bunsigned(27 downto 0) & "00000";
142  when "00110" => output_sig<= Bunsigned(26 downto 0) & "000000";
143  when "00111" => output_sig<= Bunsigned(25 downto 0) & "0000000";
144  when "01000" => output_sig<= Bunsigned(24 downto 0) & "00000000";
145  when "01001" => output_sig<= Bunsigned(23 downto 0) & "000000000";
146  when "01010" => output_sig<= Bunsigned(22 downto 0) & "0000000000";
147  when "01011" => output_sig<= Bunsigned(21 downto 0) & "00000000000";
148  when "01100" => output_sig<= Bunsigned(20 downto 0) & "000000000000";
149  when "01101" => output_sig<= Bunsigned(19 downto 0) & "0000000000000";
150  when "01110" => output_sig<= Bunsigned(18 downto 0) & "00000000000000";
151  when "01111" => output_sig<= Bunsigned(17 downto 0) & "000000000000000";
152  when "10000" => output_sig<= Bunsigned(16 downto 0) & "0000000000000000";
153  when "10001" => output_sig<= Bunsigned(15 downto 0) & "00000000000000000";
154  when "10010" => output_sig<= Bunsigned(14 downto 0) & "000000000000000000";
155  when "10011" => output_sig<= Bunsigned(13 downto 0) & "0000000000000000000";
156  when "10100" => output_sig<= Bunsigned(12 downto 0) & "00000000000000000000";
157  when "10101" => output_sig<= Bunsigned(11 downto 0) & "000000000000000000000";
158  when "10110" => output_sig<= Bunsigned(10 downto 0) & "0000000000000000000000";
159  when "10111" => output_sig<= Bunsigned(9 downto 0) & "00000000000000000000000";
160  when "11000" => output_sig<= Bunsigned(8 downto 0) & "000000000000000000000000";
161  when "11001" => output_sig<= Bunsigned(7 downto 0) & "0000000000000000000000000";
162  when "11010" => output_sig<= Bunsigned(6 downto 0) &
     ↪  "00000000000000000000000000";
163  when "11011" => output_sig<= Bunsigned(5 downto 0) &
     ↪  "000000000000000000000000000";
164  when "11100" => output_sig<= Bunsigned(4 downto 0) &
     ↪  "0000000000000000000000000000";
165  when "11101" => output_sig<= Bunsigned(3 downto 0) &
     ↪  "00000000000000000000000000000";
166  when "11110" => output_sig<= Bunsigned(2 downto 0) &
     ↪  "000000000000000000000000000000";
167  when "11111" => output_sig<= Bunsigned(1 downto 0) &
     ↪  "0000000000000000000000000000000";
168  when others =>
169  end case;
170
171  when "000001"=>--(shift left arithmetic)
172  --left signed shift using an immediate value
173  case offset is
174  when "00000" => output_sig<= Bunsigned;
175  when "00001" => output_sig<= Bunsigned(31 downto 0) & "0";
176  when "00010" => output_sig<= Bunsigned(30 downto 0) & "00";
177  when "00011" => output_sig<= Bunsigned(29 downto 0) & "000";
178  when "00100" => output_sig<= Bunsigned(28 downto 0) & "0000";
179  when "00101" => output_sig<= Bunsigned(27 downto 0) & "00000";
```

```
when "00110" => output_sig<= Bunsigned(26 downto 0) & "000000";
when "00111" => output_sig<= Bunsigned(25 downto 0) & "0000000";
when "01000" => output_sig<= Bunsigned(24 downto 0) & "00000000";
when "01001" => output_sig<= Bunsigned(23 downto 0) & "000000000";
when "01010" => output_sig<= Bunsigned(22 downto 0) & "0000000000";
when "01011" => output_sig<= Bunsigned(21 downto 0) & "00000000000";
when "01100" => output_sig<= Bunsigned(20 downto 0) & "000000000000";
when "01101" => output_sig<= Bunsigned(19 downto 0) & "0000000000000";
when "01110" => output_sig<= Bunsigned(18 downto 0) & "00000000000000";
when "01111" => output_sig<= Bunsigned(17 downto 0) & "000000000000000";
when "10000" => output_sig<= Bunsigned(16 downto 0) & "0000000000000000";
when "10001" => output_sig<= Bunsigned(15 downto 0) & "00000000000000000";
when "10010" => output_sig<= Bunsigned(14 downto 0) & "000000000000000000";
when "10011" => output_sig<= Bunsigned(13 downto 0) & "0000000000000000000";
when "10100" => output_sig<= Bunsigned(12 downto 0) & "00000000000000000000";
when "10101" => output_sig<= Bunsigned(11 downto 0) & "000000000000000000000";
when "10110" => output_sig<= Bunsigned(10 downto 0) & "0000000000000000000000";
when "10111" => output_sig<= Bunsigned(9 downto 0) & "00000000000000000000000";
when "11000" => output_sig<= Bunsigned(8 downto 0) & "000000000000000000000000";
when "11001" => output_sig<= Bunsigned(7 downto 0) & "0000000000000000000000000";
when "11010" => output_sig<= Bunsigned(6 downto 0) &
 ↪  "00000000000000000000000000";
when "11011" => output_sig<= Bunsigned(5 downto 0) &
 ↪  "000000000000000000000000000";
when "11100" => output_sig<= Bunsigned(4 downto 0) &
 ↪  "0000000000000000000000000000";
when "11101" => output_sig<= Bunsigned(3 downto 0) &
 ↪  "00000000000000000000000000000";
when "11110" => output_sig<= Bunsigned(2 downto 0) &
 ↪  "000000000000000000000000000000";
when "11111" => output_sig<= Bunsigned(1 downto 0) &
 ↪  "0000000000000000000000000000000";
when others =>
end case;

when "000010"=>--(shift right logical)
--unsigned right shift using an immediate
case offset is
when "00000" => output_sig<= Bunsigned;
when "00001" => output_sig<= "0" & Bunsigned(32 downto 1);
when "00010" => output_sig<= "00" & Bunsigned(32 downto 2);
when "00011" => output_sig<= "000" & Bunsigned(32 downto 3);
when "00100" => output_sig<= "0000" & Bunsigned(32 downto 4);
when "00101" => output_sig<= "00000" & Bunsigned(32 downto 5);
when "00110" => output_sig<= "000000" & Bunsigned(32 downto 6);
```

```
when "00111" => output_sig<= "0000000" & Bunsigned(32 downto 7);
when "01000" => output_sig<= "00000000" & Bunsigned(32 downto 8);
when "01001" => output_sig<= "000000000" & Bunsigned(32 downto 9);
when "01010" => output_sig<= "0000000000" & Bunsigned(32 downto 10);
when "01011" => output_sig<= "00000000000" & Bunsigned(32 downto 11);
when "01100" => output_sig<= "000000000000" & Bunsigned(32 downto 12);
when "01101" => output_sig<= "0000000000000" & Bunsigned(32 downto 13);
when "01110" => output_sig<= "00000000000000" & Bunsigned(32 downto 14);
when "01111" => output_sig<= "000000000000000" & Bunsigned(32 downto 15);
when "10000" => output_sig<= "0000000000000000" & Bunsigned(32 downto 16);
when "10001" => output_sig<= "00000000000000000" & Bunsigned(32 downto 17);
when "10010" => output_sig<= "000000000000000000" & Bunsigned(32 downto 18);
when "10011" => output_sig<= "0000000000000000000" & Bunsigned(32 downto 19);
when "10100" => output_sig<= "00000000000000000000" & Bunsigned(32 downto 20);
when "10101" => output_sig<= "000000000000000000000" & Bunsigned(32 downto 21);
when "10110" => output_sig<= "0000000000000000000000" & Bunsigned(32 downto 22);
when "10111" => output_sig<= "00000000000000000000000" & Bunsigned(32 downto 23);
when "11000" => output_sig<= "000000000000000000000000" & Bunsigned(32 downto
 ↪  24);
when "11001" => output_sig<= "0000000000000000000000000" & Bunsigned(32 downto
 ↪  25);
when "11010" => output_sig<= "00000000000000000000000000" & Bunsigned(32 downto
 ↪  26);
when "11011" => output_sig<= "000000000000000000000000000" & Bunsigned(32 downto
 ↪  27);
when "11100" => output_sig<= "0000000000000000000000000000" & Bunsigned(32 downto
 ↪  28);
when "11101" => output_sig<= "00000000000000000000000000000" & Bunsigned(32
 ↪  downto 29);
when "11110" => output_sig<= "000000000000000000000000000000" & Bunsigned(32
 ↪  downto 30);
when "11111" => output_sig<= "0000000000000000000000000000000" & Bunsigned(32
 ↪  downto 31);
when others =>
end case;

when "000011"=>--(shift right arithmetic)
--signed right shift using an immediate
if(Bsigned(31) = '0') then --in the case it is positive
case offset is
when "00000" => output_sig<= Bsigned;
when "00001" => output_sig<= "0" & Bsigned(32 downto 1);
when "00010" => output_sig<= "00" & Bsigned(32 downto 2);
when "00011" => output_sig<= "000" & Bsigned(32 downto 3);
when "00100" => output_sig<= "0000" & Bsigned(32 downto 4);
```

```
when "00101" => output_sig<= "00000" & Bsigned(32 downto 5);
when "00110" => output_sig<= "000000" & Bsigned(32 downto 6);
when "00111" => output_sig<= "0000000" & Bsigned(32 downto 7);
when "01000" => output_sig<= "00000000" & Bsigned(32 downto 8);
when "01001" => output_sig<= "000000000" & Bsigned(32 downto 9);
when "01010" => output_sig<= "0000000000" & Bsigned(32 downto 10);
when "01011" => output_sig<= "00000000000" & Bsigned(32 downto 11);
when "01100" => output_sig<= "000000000000" & Bsigned(32 downto 12);
when "01101" => output_sig<= "0000000000000" & Bsigned(32 downto 13);
when "01110" => output_sig<= "00000000000000" & Bsigned(32 downto 14);
when "01111" => output_sig<= "000000000000000" & Bsigned(32 downto 15);
when "10000" => output_sig<= "0000000000000000" & Bsigned(32 downto 16);
when "10001" => output_sig<= "00000000000000000" & Bsigned(32 downto 17);
when "10010" => output_sig<= "000000000000000000" & Bsigned(32 downto 18);
when "10011" => output_sig<= "0000000000000000000" & Bsigned(32 downto 19);
when "10100" => output_sig<= "00000000000000000000" & Bsigned(32 downto 20);
when "10101" => output_sig<= "000000000000000000000" & Bsigned(32 downto 21);
when "10110" => output_sig<= "0000000000000000000000" & Bsigned(32 downto 22);
when "10111" => output_sig<= "00000000000000000000000" & Bsigned(32 downto 23);
when "11000" => output_sig<= "000000000000000000000000" & Bsigned(32 downto 24);
when "11001" => output_sig<= "0000000000000000000000000" & Bsigned(32 downto 25);
when "11010" => output_sig<= "00000000000000000000000000" & Bsigned(32 downto
 ↪  26);
when "11011" => output_sig<= "000000000000000000000000000" & Bsigned(32 downto
 ↪  27);
when "11100" => output_sig<= "0000000000000000000000000000" & Bsigned(32 downto
 ↪  28);
when "11101" => output_sig<= "00000000000000000000000000000" & Bsigned(32 downto
 ↪  29);
when "11110" => output_sig<= "000000000000000000000000000000" & Bsigned(32 downto
 ↪  30);
when "11111" => output_sig<= "0000000000000000000000000000000" & Bsigned(32
 ↪  downto 31);
when others =>
end case;
else -- in the case that it is negative
case offset is
when "00000" => output_sig<= Bsigned;
when "00001" => output_sig<= "1" & Bsigned(32 downto 1);
when "00010" => output_sig<= "11" & Bsigned(32 downto 2);
when "00011" => output_sig<= "111" & Bsigned(32 downto 3);
when "00100" => output_sig<= "1111" & Bsigned(32 downto 4);
when "00101" => output_sig<= "11111" & Bsigned(32 downto 5);
when "00110" => output_sig<= "111111" & Bsigned(32 downto 6);
when "00111" => output_sig<= "1111111" & Bsigned(32 downto 7);
```

```
when "01000" => output_sig<= "11111111" & Bsigned(32 downto 8);
when "01001" => output_sig<= "111111111" & Bsigned(32 downto 9);
when "01010" => output_sig<= "1111111111" & Bsigned(32 downto 10);
when "01011" => output_sig<= "11111111111" & Bsigned(32 downto 11);
when "01100" => output_sig<= "111111111111" & Bsigned(32 downto 12);
when "01101" => output_sig<= "1111111111111" & Bsigned(32 downto 13);
when "01110" => output_sig<= "11111111111111" & Bsigned(32 downto 14);
when "01111" => output_sig<= "111111111111111" & Bsigned(32 downto 15);
when "10000" => output_sig<= "1111111111111111" & Bsigned(32 downto 16);
when "10001" => output_sig<= "11111111111111111" & Bsigned(32 downto 17);
when "10010" => output_sig<= "11111111111111111" & Bsigned(32 downto 18);
when "10011" => output_sig<= "111111111111111111" & Bsigned(32 downto 19);
when "10100" => output_sig<= "1111111111111111111" & Bsigned(32 downto 20);
when "10101" => output_sig<= "11111111111111111111" & Bsigned(32 downto 21);
when "10110" => output_sig<= "11111111111111111111" & Bsigned(32 downto 22);
when "10111" => output_sig<= "111111111111111111111" & Bsigned(32 downto 23);
when "11000" => output_sig<= "111111111111111111111" & Bsigned(32 downto 24);
when "11001" => output_sig<= "1111111111111111111111" & Bsigned(32 downto 25);
when "11010" => output_sig<= "11111111111111111111111111" & Bsigned(32 downto
 ↪  26);
when "11011" => output_sig<= "11111111111111111111111111" & Bsigned(32 downto
 ↪  27);
when "11100" => output_sig<= "11111111111111111111111111" & Bsigned(32 downto
 ↪  28);
when "11101" => output_sig<= "11111111111111111111111111" & Bsigned(32 downto
 ↪  29);
when "11110" => output_sig<= "11111111111111111111111111" & Bsigned(32 downto
 ↪  30);
when "11111" => output_sig<= "11111111111111111111111111" & Bsigned(32
 ↪  downto 31);
when others =>
end case;
end if;

when "000100"=> --(shift left logical variable)
--unsigned left shift based on the value in a register
case Aunsigned(4 downto 0) is
when "00000" => output_sig<= Bunsigned;
when "00001" => output_sig<= Bunsigned(31 downto 0) & "0";
when "00010" => output_sig<= Bunsigned(30 downto 0) & "00";
when "00011" => output_sig<= Bunsigned(29 downto 0) & "000";
when "00100" => output_sig<= Bunsigned(28 downto 0) & "0000";
when "00101" => output_sig<= Bunsigned(27 downto 0) & "00000";
when "00110" => output_sig<= Bunsigned(26 downto 0) & "000000";
when "00111" => output_sig<= Bunsigned(25 downto 0) & "0000000";
```

```
334  when "01000" => output_sig<= Bunsigned(24 downto 0) & "00000000";
335  when "01001" => output_sig<= Bunsigned(23 downto 0) & "000000000";
336  when "01010" => output_sig<= Bunsigned(22 downto 0) & "0000000000";
337  when "01011" => output_sig<= Bunsigned(21 downto 0) & "00000000000";
338  when "01100" => output_sig<= Bunsigned(20 downto 0) & "000000000000";
339  when "01101" => output_sig<= Bunsigned(19 downto 0) & "0000000000000";
340  when "01110" => output_sig<= Bunsigned(18 downto 0) & "00000000000000";
341  when "01111" => output_sig<= Bunsigned(17 downto 0) & "000000000000000";
342  when "10000" => output_sig<= Bunsigned(16 downto 0) & "0000000000000000";
343  when "10001" => output_sig<= Bunsigned(15 downto 0) & "00000000000000000";
344  when "10010" => output_sig<= Bunsigned(14 downto 0) & "000000000000000000";
345  when "10011" => output_sig<= Bunsigned(13 downto 0) & "0000000000000000000";
346  when "10100" => output_sig<= Bunsigned(12 downto 0) & "00000000000000000000";
347  when "10101" => output_sig<= Bunsigned(11 downto 0) & "000000000000000000000";
348  when "10110" => output_sig<= Bunsigned(10 downto 0) & "0000000000000000000000";
349  when "10111" => output_sig<= Bunsigned(9 downto 0) & "00000000000000000000000";
350  when "11000" => output_sig<= Bunsigned(8 downto 0) & "000000000000000000000000";
351  when "11001" => output_sig<= Bunsigned(7 downto 0) & "0000000000000000000000000";
352  when "11010" => output_sig<= Bunsigned(6 downto 0) &
      ↪  "00000000000000000000000000";
353  when "11011" => output_sig<= Bunsigned(5 downto 0) &
      ↪  "000000000000000000000000000";
354  when "11100" => output_sig<= Bunsigned(4 downto 0) &
      ↪  "0000000000000000000000000000";
355  when "11101" => output_sig<= Bunsigned(3 downto 0) &
      ↪  "00000000000000000000000000000";
356  when "11110" => output_sig<= Bunsigned(2 downto 0) &
      ↪  "000000000000000000000000000000";
357  when "11111" => output_sig<= Bunsigned(1 downto 0) &
      ↪  "0000000000000000000000000000000";
358  when others =>
359  end case;
360
361  when "000101"=> --(shift left arithmetic variable)
362  --signed left shift based on the value in a register
363  case Aunsigned(4 downto 0) is
364  when "00000" => output_sig<= Bunsigned;
365  when "00001" => output_sig<= Bunsigned(31 downto 0) & "0";
366  when "00010" => output_sig<= Bunsigned(30 downto 0) & "00";
367  when "00011" => output_sig<= Bunsigned(29 downto 0) & "000";
368  when "00100" => output_sig<= Bunsigned(28 downto 0) & "0000";
369  when "00101" => output_sig<= Bunsigned(27 downto 0) & "00000";
370  when "00110" => output_sig<= Bunsigned(26 downto 0) & "000000";
371  when "00111" => output_sig<= Bunsigned(25 downto 0) & "0000000";
372  when "01000" => output_sig<= Bunsigned(24 downto 0) & "00000000";
```

```
when "01001" => output_sig<= Bunsigned(23 downto 0) & "000000000";
when "01010" => output_sig<= Bunsigned(22 downto 0) & "0000000000";
when "01011" => output_sig<= Bunsigned(21 downto 0) & "00000000000";
when "01100" => output_sig<= Bunsigned(20 downto 0) & "000000000000";
when "01101" => output_sig<= Bunsigned(19 downto 0) & "0000000000000";
when "01110" => output_sig<= Bunsigned(18 downto 0) & "00000000000000";
when "01111" => output_sig<= Bunsigned(17 downto 0) & "000000000000000";
when "10000" => output_sig<= Bunsigned(16 downto 0) & "0000000000000000";
when "10001" => output_sig<= Bunsigned(15 downto 0) & "00000000000000000";
when "10010" => output_sig<= Bunsigned(14 downto 0) & "000000000000000000";
when "10011" => output_sig<= Bunsigned(13 downto 0) & "0000000000000000000";
when "10100" => output_sig<= Bunsigned(12 downto 0) & "00000000000000000000";
when "10101" => output_sig<= Bunsigned(11 downto 0) & "000000000000000000000";
when "10110" => output_sig<= Bunsigned(10 downto 0) & "0000000000000000000000";
when "10111" => output_sig<= Bunsigned(9 downto 0) & "00000000000000000000000";
when "11000" => output_sig<= Bunsigned(8 downto 0) & "000000000000000000000000";
when "11001" => output_sig<= Bunsigned(7 downto 0) & "0000000000000000000000000";
when "11010" => output_sig<= Bunsigned(6 downto 0) &
 ↪  "00000000000000000000000000";
when "11011" => output_sig<= Bunsigned(5 downto 0) &
 ↪  "000000000000000000000000000";
when "11100" => output_sig<= Bunsigned(4 downto 0) &
 ↪  "0000000000000000000000000000";
when "11101" => output_sig<= Bunsigned(3 downto 0) &
 ↪  "00000000000000000000000000000";
when "11110" => output_sig<= Bunsigned(2 downto 0) &
 ↪  "000000000000000000000000000000";
when "11111" => output_sig<= Bunsigned(1 downto 0) &
 ↪  "0000000000000000000000000000000";
when others =>
end case;

when "000110"=> --(shift right logical variable)
--unsigned right shift based on the value in a register
case Aunsigned(4 downto 0) is
when "00000" => output_sig<= Bunsigned;
when "00001" => output_sig<= "0" & Bunsigned(32 downto 1);
when "00010" => output_sig<= "00" & Bunsigned(32 downto 2);
when "00011" => output_sig<= "000" & Bunsigned(32 downto 3);
when "00100" => output_sig<= "0000" & Bunsigned(32 downto 4);
when "00101" => output_sig<= "00000" & Bunsigned(32 downto 5);
when "00110" => output_sig<= "000000" & Bunsigned(32 downto 6);
when "00111" => output_sig<= "0000000" & Bunsigned(32 downto 7);
when "01000" => output_sig<= "00000000" & Bunsigned(32 downto 8);
when "01001" => output_sig<= "000000000" & Bunsigned(32 downto 9);
```

```
412 when "01010" => output_sig<= "0000000000" & Bunsigned(32 downto 10);
413 when "01011" => output_sig<= "00000000000" & Bunsigned(32 downto 11);
414 when "01100" => output_sig<= "000000000000" & Bunsigned(32 downto 12);
415 when "01101" => output_sig<= "0000000000000" & Bunsigned(32 downto 13);
416 when "01110" => output_sig<= "00000000000000" & Bunsigned(32 downto 14);
417 when "01111" => output_sig<= "000000000000000" & Bunsigned(32 downto 15);
418 when "10000" => output_sig<= "0000000000000000" & Bunsigned(32 downto 16);
419 when "10001" => output_sig<= "00000000000000000" & Bunsigned(32 downto 17);
420 when "10010" => output_sig<= "00000000000000000" & Bunsigned(32 downto 18);
421 when "10011" => output_sig<= "000000000000000000" & Bunsigned(32 downto 19);
422 when "10100" => output_sig<= "000000000000000000" & Bunsigned(32 downto 20);
423 when "10101" => output_sig<= "0000000000000000000" & Bunsigned(32 downto 21);
424 when "10110" => output_sig<= "0000000000000000000" & Bunsigned(32 downto 22);
425 when "10111" => output_sig<= "00000000000000000000" & Bunsigned(32 downto 23);
426 when "11000" => output_sig<= "00000000000000000000" & Bunsigned(32 downto
    ↪ 24);
427 when "11001" => output_sig<= "000000000000000000000" & Bunsigned(32 downto
    ↪ 25);
428 when "11010" => output_sig<= "000000000000000000000" & Bunsigned(32 downto
    ↪ 26);
429 when "11011" => output_sig<= "000000000000000000000" & Bunsigned(32 downto
    ↪ 27);
430 when "11100" => output_sig<= "000000000000000000000" & Bunsigned(32 downto
    ↪ 28);
431 when "11101" => output_sig<= "000000000000000000000" & Bunsigned(32
    ↪ downto 29);
432 when "11110" => output_sig<= "000000000000000000000" & Bunsigned(32
    ↪ downto 30);
433 when "11111" => output_sig<= "000000000000000000000" & Bunsigned(32
    ↪ downto 31);
434 when others =>
435 end case;
436
437 when "000111"=> --(shift right arithmetic variable)
438 --signed right shift based on the value in a register
439 if(Bsigned(31) = '0') then --in the case that the value is poisitive
440 case Aunsigned(4 downto 0) is
441 when "00000" => output_sig<= Bsigned;
442 when "00001" => output_sig<= "0" & Bsigned(32 downto 1);
443 when "00010" => output_sig<= "00" & Bsigned(32 downto 2);
444 when "00011" => output_sig<= "000" & Bsigned(32 downto 3);
445 when "00100" => output_sig<= "0000" & Bsigned(32 downto 4);
446 when "00101" => output_sig<= "00000" & Bsigned(32 downto 5);
447 when "00110" => output_sig<= "000000" & Bsigned(32 downto 6);
448 when "00111" => output_sig<= "0000000" & Bsigned(32 downto 7);
```

```
449 when "01000" => output_sig<= "00000000" & Bsigned(32 downto 8);
450 when "01001" => output_sig<= "000000000" & Bsigned(32 downto 9);
451 when "01010" => output_sig<= "0000000000" & Bsigned(32 downto 10);
452 when "01011" => output_sig<= "00000000000" & Bsigned(32 downto 11);
453 when "01100" => output_sig<= "000000000000" & Bsigned(32 downto 12);
454 when "01101" => output_sig<= "0000000000000" & Bsigned(32 downto 13);
455 when "01110" => output_sig<= "00000000000000" & Bsigned(32 downto 14);
456 when "01111" => output_sig<= "000000000000000" & Bsigned(32 downto 15);
457 when "10000" => output_sig<= "0000000000000000" & Bsigned(32 downto 16);
458 when "10001" => output_sig<= "00000000000000000" & Bsigned(32 downto 17);
459 when "10010" => output_sig<= "000000000000000000" & Bsigned(32 downto 18);
460 when "10011" => output_sig<= "0000000000000000000" & Bsigned(32 downto 19);
461 when "10100" => output_sig<= "00000000000000000000" & Bsigned(32 downto 20);
462 when "10101" => output_sig<= "000000000000000000000" & Bsigned(32 downto 21);
463 when "10110" => output_sig<= "0000000000000000000000" & Bsigned(32 downto 22);
464 when "10111" => output_sig<= "00000000000000000000000" & Bsigned(32 downto 23);
465 when "11000" => output_sig<= "000000000000000000000000" & Bsigned(32 downto 24);
466 when "11001" => output_sig<= "0000000000000000000000000" & Bsigned(32 downto 25);
467 when "11010" => output_sig<= "00000000000000000000000000" & Bsigned(32 downto
    ↪ 26);
468 when "11011" => output_sig<= "000000000000000000000000000" & Bsigned(32 downto
    ↪ 27);
469 when "11100" => output_sig<= "0000000000000000000000000000" & Bsigned(32 downto
    ↪ 28);
470 when "11101" => output_sig<= "00000000000000000000000000000" & Bsigned(32 downto
    ↪ 29);
471 when "11110" => output_sig<= "000000000000000000000000000000" & Bsigned(32 downto
    ↪ 30);
472 when "11111" => output_sig<= "0000000000000000000000000000000" & Bsigned(32
    ↪ downto 31);
473 when others =>
474 end case;
475 else -- in the case that the value is negative
476 case Aunsigned(4 downto 0) is
477 when "00000" => output_sig<= Bsigned;
478 when "00001" => output_sig<= "1" & Bsigned(32 downto 1);
479 when "00010" => output_sig<= "11" & Bsigned(32 downto 2);
480 when "00011" => output_sig<= "111" & Bsigned(32 downto 3);
481 when "00100" => output_sig<= "1111" & Bsigned(32 downto 4);
482 when "00101" => output_sig<= "11111" & Bsigned(32 downto 5);
483 when "00110" => output_sig<= "111111" & Bsigned(32 downto 6);
484 when "00111" => output_sig<= "1111111" & Bsigned(32 downto 7);
485 when "01000" => output_sig<= "11111111" & Bsigned(32 downto 8);
486 when "01001" => output_sig<= "111111111" & Bsigned(32 downto 9);
487 when "01010" => output_sig<= "1111111111" & Bsigned(32 downto 10);
```

```
488 when "01011" => output_sig<= "11111111111" & Bsigned(32 downto 11);
489 when "01100" => output_sig<= "111111111111" & Bsigned(32 downto 12);
490 when "01101" => output_sig<= "1111111111111" & Bsigned(32 downto 13);
491 when "01110" => output_sig<= "11111111111111" & Bsigned(32 downto 14);
492 when "01111" => output_sig<= "111111111111111" & Bsigned(32 downto 15);
493 when "10000" => output_sig<= "1111111111111111" & Bsigned(32 downto 16);
494 when "10001" => output_sig<= "11111111111111111" & Bsigned(32 downto 17);
495 when "10010" => output_sig<= "11111111111111111" & Bsigned(32 downto 18);
496 when "10011" => output_sig<= "11111111111111111" & Bsigned(32 downto 19);
497 when "10100" => output_sig<= "11111111111111111" & Bsigned(32 downto 20);
498 when "10101" => output_sig<= "11111111111111111" & Bsigned(32 downto 21);
499 when "10110" => output_sig<= "11111111111111111" & Bsigned(32 downto 22);
500 when "10111" => output_sig<= "11111111111111111" & Bsigned(32 downto 23);
501 when "11000" => output_sig<= "11111111111111111" & Bsigned(32 downto 24);
502 when "11001" => output_sig<= "11111111111111111" & Bsigned(32 downto 25);
503 when "11010" => output_sig<= "11111111111111111" & Bsigned(32 downto
    ↪ 26);
504 when "11011" => output_sig<= "11111111111111111" & Bsigned(32 downto
    ↪ 27);
505 when "11100" => output_sig<= "11111111111111111" & Bsigned(32 downto
    ↪ 28);
506 when "11101" => output_sig<= "11111111111111111" & Bsigned(32 downto
    ↪ 29);
507 when "11110" => output_sig<= "11111111111111111" & Bsigned(32 downto
    ↪ 30);
508 when "11111" => output_sig<= "11111111111111111" & Bsigned(32
    ↪ downto 31);
509 when others =>
510 end case;
511 end if;
512 when others =>
513
514 end case; -- opcodes are no longer x00
515
516 when "001000"=> --(add immediate signed)
517 output_sig <= std_logic_vector(signed(Asigned) + signed(Bsigned));
518 if (Asigned(31) /= Bsigned(31)) then
519 overflow <= '0';
520 else
521 if(output_sig(31) /= Asigned(31)) then
522 overflow <= '1';
523 else
524 overflow<= '0';
525 end if;
526 end if;
```

```
527
528 when "001001"=> --(add immediate unsigned)
529 output_sig <= std_logic_vector(unsigned(Aunsigned) + unsigned(Bunsigned));
530 if(output_sig(32) = '1') then
531 overflow <= '1';
532 else
533 overflow <= '0';
534 end if;
535
536 when "001010"=> --(set on less than immediate)
537 if(Asigned(31) = Bsigned(31)) then
538 if(Asigned(31)='0') then
539 if(Asigned<Bsigned) then
540  output_sig<= "00000000000000000000000000000001";
541 else
542  output_sig<= "00000000000000000000000000000000";
543 end if;
544 else -- both A and B are negative, abs values must be compared
545 Asigned_tmp  <= not(Asigned) + '1';
546 Bsigned_tmp  <= not(Bsigned) + '1';
547 if(Asigned_tmp > Bsigned_tmp) then -- if A is a smaller negative number,althought
    ↪  abs(A) < abs(B), A is > B.
548  output_sig<= "00000000000000000000000000000001";
549 else
550  output_sig<= "00000000000000000000000000000000";
551 end if;
552 end if;
553 end if;
554
555 --Branches
556 when "000100"=>--BEQ
557 if (inputA=inputB) then
558 branch<='1';
559 else
560 branch<='0';
561 end if;
562 when "000101"=>--BNE
563 if (NOT (inputA=inputB)) then
564 branch<='1';
565 else
566 branch<='0';
567 end if;
568
569 when "000001" =>--Branch on less than than zero (we are ignoring the other bits
    ↪  for now)
```

```
570 if (inputA(31)='1') then--if the signed number is negative
571 branch<='1';
572 else
573 branch<='0';
574 end if;
575
576 when "000110"=> --branch on less than or equal to zero
577 if (inputA(31)='1' or inputA(31 downto 0)="00000000000000000000000000000000")
    ↪ then--if the signed number is negative
578 branch<='1';
579 else
580 branch<='0';
581 end if;
582
583 when "000111"=> -- branch on greater than zero
584 if (inputA(31)='0' and inputA(31 downto 0)/= "00000000000000000000000000000000")
    ↪ then
585 branch<='1';
586 else
587 branch<='0';
588 end if;
589
590 when "001011"=> --(set on less than immediate unsigned)
591 if(Aunsigned < Bunsigned) then
592 output_sig<= "00000000000000000000000000000001";
593 else
594 output_sig<= "00000000000000000000000000000000";
595 end if;
596
597 when "001100"=> --(and immediate)
598 output_sig <= (Aunsigned) AND (Bunsigned);
599 when "001101"=> --(or immediate)
600 output_sig <= (Aunsigned) OR (Bunsigned);
601 when "001110"=> --(xor immediate)
602 output_sig <= (Aunsigned) XOR (Bunsigned);
603 when "001111"=> --(load upper immediate)
604 output_sig(15 downto 0)<="0000000000000000";
605 output_sig(31 downto 16)<=Aunsigned(15 downto 0);
606 when "100011"=> --loadword
607 output_sig <= std_logic_vector(signed(Asigned) + signed(Bsigned));
608      if Asigned(31) /= Bsigned(31) then -- adding a positive and negative number,
         ↪ cant have overflow
609        overflow <= '0';
610     else
```

```
611         if output_sig(31) /= Asigned(31) then --both numbers are positive or both
            ↪  numbers are negative (output MSB must match input MSBs which are the
            ↪  same since boeth are of same sign)
612             overflow <= '1';
613         else overflow <= '0';
614         end if;
615    end if;
616 when "101011"=>--storeword
617 output_sig <= std_logic_vector(signed(Asigned) + signed(Bsigned));
618 if Asigned(31) /= Bsigned(31) then -- adding a positive and negative number, cant
    ↪  have overflow
619 overflow <= '0';
620 else
621 if output_sig(31) /= Asigned(31) then --both numbers are positive or both numbers
    ↪  are negative (output MSB must match input MSBs which are the same since boeth
    ↪  are of same sign)
622     overflow <= '1';
623 else overflow <= '0';
624 end if;
625 end if;
626 when others =>
627 end case;
628 end process;
629
630 --end process;
631    output<=output_sig(31 downto 0);
632
633 end Behavioral;
```

7.5 Control.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Control unit
3  --
4  --sends control signals based on opcodes out to different parts of the processor
5  ----------------------------------------------------------------------------------
6  library IEEE;
7  use IEEE.STD_LOGIC_1164.ALL;
8  use IEEE.numeric_std.all;
9  use ieee.std_logic_unsigned.all;
10
11 entity control is
12 port(
13 clk: in std_logic;
14 branch: in std_logic;
```

```
15  IR2, IR3, IR4, IR5 : in std_logic_vector(31 downto 0);
16  LoadSel, RegWrite, Asel, readWrite: out std_logic;
17  PCsel: out std_logic_vector(2 downto 0);
18  Bsel, Rselect: out std_logic_vector(1 downto 0)
19  );
20  end control;
21
22  architecture Behavioral of control is
23   signal opcode2, opcode3, opcode4, opcode5: std_logic_vector(5 downto 0);
      ↪  --opcodes in each stage
24   signal funct2, funct3, funct4, funct5: std_logic_vector(5 downto 0); --function
      ↪  bits for each stage
25   signal concat3, concat4: std_logic_vector(11 downto 0);
26   signal opResult1, opResult2, functResult, opAndfunct, opOrFunct, storet, stored,
      ↪  bchoose,B, achoose: std_logic;
27
28   begin
29   --sets control signals based on values in the instruction registers
30   opcode2 <= IR2(31 downto 26);
31   opcode3 <= IR3(31 downto 26);
32   opcode4 <= IR4(31 downto 26);
33   opcode5 <= IR5(31 downto 26);
34   funct2 <= IR2(5 downto 0);
35   funct3 <= IR3(5 downto 0);
36   funct4 <= IR4(5 downto 0);
37   funct5 <= IR5(5 downto 0);
38
39   process(clk, IR4)
40   begin
41    ---IF
42    --PCsel is the select for mux1 in the datapath
43    -- determines what the next value of PC will be
44    case opcode4 is
45      when "000010"=>
46              PCsel<="011";--jump
47    when "111111"=>
48    PCsel<="001"; --noop
49    when "000001"|"000100"|"000101"|"000110"|"000111"=>--branch instructions
50         if (branch='1') then
51             PCsel<="010"; --only branches if the branch condition was met
52         else
53             PCsel<="001";
54         end if;
55    when others=>
56    PCsel <="001";--increase pc by 4
```

```
57    end case;
58   end process;
59
60  concat4<=opcode4&funct4;
61
62  --selects depending on if data needs to be forwarded to either the A or B input
     ↪  of the ALU
63  process(IR4, IR3, storet, stored)
64  begin
65      if ((IR4(20 downto 16)=IR3(20 downto 16) AND storet='1')OR(IR4(15 downto
         ↪  11)=IR3(20 downto 16) AND stored='1')) then bchoose<='1';
66      else bchoose<='0';
67      end if;
68      if ((IR4(20 downto 16)=IR3(25 downto 21) AND storet='1')OR(IR4(15 downto
         ↪  11)=IR3(25 downto 21) AND stored='1')) then achoose<='1';
69          else achoose<='0';
70     end if;
71  end process;
72
73  process(B, bchoose)
74  begin
75      if (bchoose='1')then Bsel<="11";--forwarding data
76      elsif(B='1') then Bsel<="01";--adding from reg
77      else Bsel<="00";--adding immediate
78      end if;
79  end process;
80
81  Asel<=achoose;
82
83  -- determines if the last operation stored using rd
84   with concat4 select stored<=
85   '1' when "000000100000" | "000000100001" | "000000100100" | "000000100101" |
      ↪  "000000000000" | "000000000100" | "000000101010" | "000000101011" |
      ↪  "000000000011" | "000000000010" | "000000000110" | "000000100011" |
      ↪  "000000100110",
86   '0' when others;
87  -- determines if the last operation stored using rt
88   with opcode4  select storet<=
89   '1' when "001001" | "001100" | "100000" | "100011" | "001101" | "011010" |
      ↪  "001011" | "001110" |"010000",
90      '0' when others;
91  -- select for which B value we are using for the given opcode
92   with opcode3 select B<=
93   '0' when "001000"| "001001" |"001010" |"001011" | "001100" | "001101" | "001110"
      ↪  | "001111" | "100011" | "101011",--options that use immediates
```

```
 '1' when others;
-- select for determining what data is being written into the GPRs
 with opcode5 select LoadSel <=
 '1' when "001000" | "001001" |"001010" |"001011" | "001100" |"001101" | "001110"
  ↪  | "001111"| "000000",
 '0' when others;
-- select for determining whether we are writing using Rs or Rt
 with opcode5 select Rselect <=
 "01" when "000000",
 "10" when "000100" | "000101",
 "00" when others;
-- select for writing to the GPRs
 with opcode5 select RegWrite <=
    '1' when "000000"|"001000" | "001001" |"001010" |"001011" | "001100"
     ↪  |"001101" | "001110" | "001111"|"100011",
    '0' when others;
-- used for whether we are writing to memory or not
 with opcode4 select readWrite <=
 '1' when "101011",
 '0' when others;

 end Behavioral;
```

7.6 Memory_Unit.vhd

```
----------------------------------------------------------------------------------
-- Memory unit
-- contains all memory for the a 32 bit MIPS processor
-- includes data memory which can be read and written to as well as read only
-- instruction memory
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity memory_unit is
port
(
ab1: in std_logic_vector(31 downto 0); -- pc (address of instruction)
ib1: out std_logic_vector(31 downto 0); -- instruction fetched from memory

ab2: in std_logic_vector(31 downto 0); --  address of data (to be fetched or
 ↪  written to)
```

```
19  db2: in std_logic_vector(31 downto 0); -- carries the data to be written to
    ↪  memory
20  write_en : in std_logic; -- write enable
21  db3: out std_logic_vector(31 downto 0); -- data out
22
23  clear: in std_logic; -- clear bit (for data memory)
24  clock : in std_logic; -- clock signal
25  switches:in std_logic_vector(15 downto 0)
26  );
27  end memory_unit;
28
29  architecture Behavioral of memory_unit is
30
31  signal inst_addr : std_logic_vector(31 downto 0);
32  signal inst : std_logic_vector(31 downto 0); -- instruction that'll go out on ib1
33
34  signal data_addr : std_logic_vector(31 downto 0);
35  signal data_in : std_logic_vector(31 downto 0);
36  signal data_out : std_logic_vector(31 downto 0);
37
38
39  -- size of memory is 64x32 (64 32-bit locations)
40  type rom is array (0 to 63) of std_logic_vector(31 downto 0);
41  type ram is array (0 to 63) of std_logic_vector(31 downto 0);
42
43  --instruction memory
44  --instruction memory (count instances progeam). Remove spaces before
    ↪  implementing. They are for documentation/debug purposes
45  constant inst_mem : rom :=
46  (
47  "10001100000000010000000000011100","00000000000000000011000000100000","00000000000000000000110000
    ↪  add, add, addi,
48  "10001100011001010000000000000000", --load
49  "11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000
50  "00100000011000110000000000000001","00010000101000100000000000001111", --branch
51  "11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000
52  "00010100011001000000000000001011",--instruction 60(decimal) branch
53  "11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000
54  "00001000000000000000000000011111", --instruction 80(decimal) jump
55  "11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000
56  "00000000110000010011000000100000","00001000000000000000000000000100",--instruction
    ↪  104 jump
57  "11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000
58  "10101100000001100000000000011111",
59  "11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000
```

```
"11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000000
"11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000000
"00001000000000000000000000101000","11111100000000000000000000000000","11111100000000000000000000
 ↪  --jump for infintite loop of noops
"11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000000
"11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000000
"11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000000
"11111100000000000000000000000000","11111100000000000000000000000000","11111100000000000000000000
--"11111100000000000000000000000000", "11111100000000000000000000000000",
 ↪  "11111100000000000000000000000000", "11111100000000000000000000000000",
--"11111100000000000000000000000000" "11111100000000000000000000000000",
 ↪  "11111100000000000000000000000000", "11111100000000000000000000000000"
--"11111100000000000000000000000000", "11111100000000000000000000000000",
 ↪  "11111100000000000000000000000000", "11111100000000000000000000000000",
);

--DATA Memory. In this example the array is searched for "110" or 6, the
 ↪  destination is on line 31, location 0x20. It should find 3 instances.
signal data_mem : ram :=
(
--"00000000000000000000000000000001", "00000000000000000000000000001100",
 ↪  "00000000000000000000000000000110", "00000000000000000000000000000110",
--"00000000000000000000000000000001", "00000000000000000000000000000110",
 ↪  "00000000000000000000000000000001", "00000000000000000000000000000101",
--"00000000000000000000000000001111", "00000000000000000000000000000011",
 ↪  "00000000000000000000000000000001", "00000000000000000000000000000011",
--"00000000000000000000000000000000", "00000000000000000000000000000000",
 ↪  "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000111", "00000000000000000000000000000001",
 ↪  "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000000", "00000000000000000000000000000000",
 ↪  "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000000", "00000000000000000000000000000100",
 ↪  "00000000000000000000000000000000", "00000000000000000000000000000001",
--"00000000000000000000000000000110", "00000000000000000000000000001000",
 ↪  "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000000", "00000000000000000000000000000000",
 ↪  "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000000", "00000000000000000000000000000000",
 ↪  "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000000", "00000000000000000000000000000000",
 ↪  "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000000", "00000000000000000000000000000000",
 ↪  "00000000000000000000000000000000", "00000000000000000000000000000000",
```

```
--"00000000000000000000000000000000", "00000000000000000000000000000000", "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000000", "00000000000000000000000000000000", "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000000", "00000000000000000000000000000000", "00000000000000000000000000000000", "00000000000000000000000000000000",
--"00000000000000000000000000000000", "00000000000000000000000000000000", "00000000000000000000000000000000", "00000000000000000000000000000000"


"00000000000000000000000000000001", "00000000000000000000000000001100", "00000000000000000000000000000110", "00000000000000000000000000000110",
"00000000000000000000000000000001", "00000000000000000000000000000110", "00000000000000000000000000000001", "00000000000000000000000000000101",
"00000000000000000000000000001111", "00000000000000000000000000000011", "00000000000000000000000000000001", "00000000000000000000000000000011",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"00000000000000000000000000000111", "00000000000000000000000000000001", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"00000000000000000000000000000110", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000",
"10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000", "10000000000000000000000000000000"

);
```

```
112 --data memory
113 signal data_mem_cleared : ram :=
114 (
115
116 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
117 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
118 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
119 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
120 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
121 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
122 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
123 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
124 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
125 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
126 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
127 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
128 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
129 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
130 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000",
131 "00000000000000000000000000000000", "00000000000000000000000000000000",
     ↪ "00000000000000000000000000000000", "00000000000000000000000000000000"
132 );
133
134
135 begin
136
137 inst_addr <= ab1;
138 data_addr <= ab2;
139 data_in <= db2;
140
```

```
-- process to get next instruction
process(clock)
begin
 if(clock = '1' and clock' event) then
  inst <= inst_mem(to_integer(unsigned(inst_addr))/4);
 end if;
end process;
data_out <= data_mem(to_integer(unsigned(data_addr))); -- read data CHANGE
 ↪  THISSSSS


--process to read/write data, data memory can be cleared as well
process(clock, clear)
begin
 if(clear = '1') then
  if (clock' event and clock='1') then
   data_mem(28)<="0000000000000000" & switches;
  END IF;
 else
  if(clock = '1' and clock' event) then
   -- write data to memory
   if(write_en = '1') then
    data_mem(to_integer(unsigned(data_addr))) <= data_in;
   end if;
  end if;
 end if;
end process;

ib1 <= inst;
db3 <= data_out;


end Behavioral;
```

7.7 Constraints File

```
## Clock signal
#NET "clk"   LOC = "E3" | IOSTANDARD = "LVCMOS33";
NET "clk"   LOC = E3 | IOSTANDARD = LVCMOS33;    #Bank = 35, Pin name =
 ↪  #IO_L12P_T1_MRCC_35,     Sch name = clk100mhz
NET "clk" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100 MHz HIGH 50%;

#
### Switches
```

```
NET "sw<0>"          LOC=J15 | IOSTANDARD=LVCMOS33; #IO_L24N_T3_RS0_15
NET "sw<1>"          LOC=L16 | IOSTANDARD=LVCMOS33; #IO_L3N_T0_DQS_EMCCLK_14
NET "sw<2>"          LOC=M13 | IOSTANDARD=LVCMOS33; #IO_L6N_T0_D08_VREF_14
NET "sw<3>"          LOC=R15 | IOSTANDARD=LVCMOS33; #IO_L13N_T2_MRCC_14
NET "sw<4>"          LOC=R17 | IOSTANDARD=LVCMOS33; #IO_L12N_T1_MRCC_14
NET "sw<5>"          LOC=T18 | IOSTANDARD=LVCMOS33; #IO_L7N_T1_D10_14
NET "sw<6>"          LOC=U18 | IOSTANDARD=LVCMOS33; #IO_L17N_T2_A13_D29_14
NET "sw<7>"          LOC=R13 | IOSTANDARD=LVCMOS33; #IO_L5N_T0_D07_14
NET "sw<8>"          LOC=T8 | IOSTANDARD=LVCMOS18; #IO_L24N_T3_34
NET "sw<9>"          LOC=U8 | IOSTANDARD=LVCMOS18; #IO_25_34
NET "sw<10>"         LOC=R16 | IOSTANDARD=LVCMOS33; #IO_L15P_T2_DQS_RDWR_B_14
NET "sw<11>"         LOC=T13 | IOSTANDARD=LVCMOS33; #IO_L23P_T3_A03_D19_14
NET "sw<12>"         LOC=H6 | IOSTANDARD=LVCMOS33; #IO_L24P_T3_35
NET "sw<13>"         LOC=U12 | IOSTANDARD=LVCMOS33; #IO_L20P_T3_A08_D24_14
NET "sw<14>"         LOC=U11 | IOSTANDARD=LVCMOS33; #IO_L19N_T3_A09_D25_VREF_14
NET "sw<15>"         LOC=V10 | IOSTANDARD=LVCMOS33; #IO_L21P_T3_DQS_14
#
#
### Buttons
##NET "cpu_resetn"     LOC=C12 | IOSTANDARD=LVCMOS33; #IO_L3P_T0_DQS_AD1P_15
#
#NET "control<0>"           LOC=N17 | IOSTANDARD=LVCMOS33; #IO_L9P_T1_DQS_14
NET "btnD"           LOC=P18 | IOSTANDARD=LVCMOS33; #IO_L9N_T1_DQS_D13_14
#NET "control<2>"           LOC=P17 | IOSTANDARD=LVCMOS33; #IO_L12P_T1_MRCC_14
#NET "control<3>"           LOC=M17 | IOSTANDARD=LVCMOS33; #IO_L10N_T1_D15_14
NET "btnU"           LOC=M18 | IOSTANDARD=LVCMOS33; #IO_L4N_T0_D05_14
#
#
### LEDs1
NET "led<0>"         LOC=H17 | IOSTANDARD=LVCMOS33; #IO_L18P_T2_A24_15
NET "led<1>"         LOC=K15 | IOSTANDARD=LVCMOS33; #IO_L24P_T3_RS1_15
NET "led<2>"         LOC=J13 | IOSTANDARD=LVCMOS33; #IO_L17N_T2_A25_15
NET "led<3>"         LOC=N14 | IOSTANDARD=LVCMOS33; #IO_L8P_T1_D11_14
NET "led<4>"         LOC=R18 | IOSTANDARD=LVCMOS33; #IO_L7P_T1_D09_14
NET "led<5>"         LOC=V17 | IOSTANDARD=LVCMOS33; #IO_L18N_T2_A11_D27_14
NET "led<6>"         LOC=U17 | IOSTANDARD=LVCMOS33; #IO_L17P_T2_A14_D30_14
NET "led<7>"         LOC=U16 | IOSTANDARD=LVCMOS33; #IO_L18P_T2_A12_D28_14
NET "led<8>"         LOC=V16 | IOSTANDARD=LVCMOS33; #IO_L16N_T2_A15_D31_14
NET "led<9>"         LOC=T15 | IOSTANDARD=LVCMOS33; #IO_L14N_T2_SRCC_14
NET "led<10>"        LOC=U14 | IOSTANDARD=LVCMOS33; #IO_L22P_T3_A05_D21_14
NET "led<11>"        LOC=T16 | IOSTANDARD=LVCMOS33; #IO_L15N_T2_DQS_DOUT_CSO_B_14
NET "led<12>"        LOC=V15 | IOSTANDARD=LVCMOS33; #IO_L16P_T2_CSI_B_14
NET "led<13>"        LOC=V14 | IOSTANDARD=LVCMOS33; #IO_L22N_T3_A04_D20_14
```

```
52 NET "led<14>"        LOC=V12 | IOSTANDARD=LVCMOS33; #IO_L20N_T3_A07_D23_14
53 NET "led<15>"        LOC=V11 | IOSTANDARD=LVCMOS33; #IO_L21N_T3_DQS_A06_D22_14
```