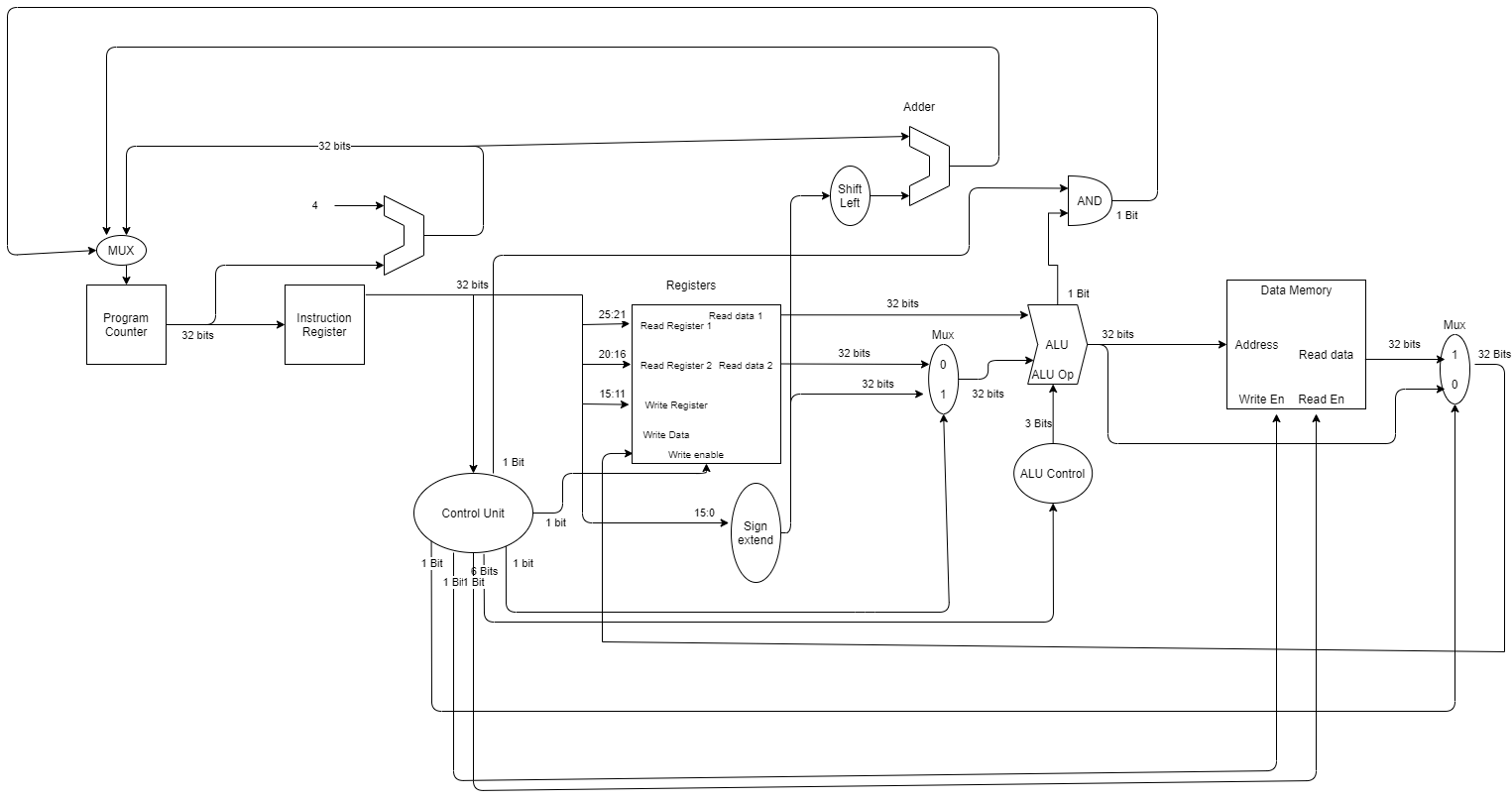
Jacob Johnson

T00237585

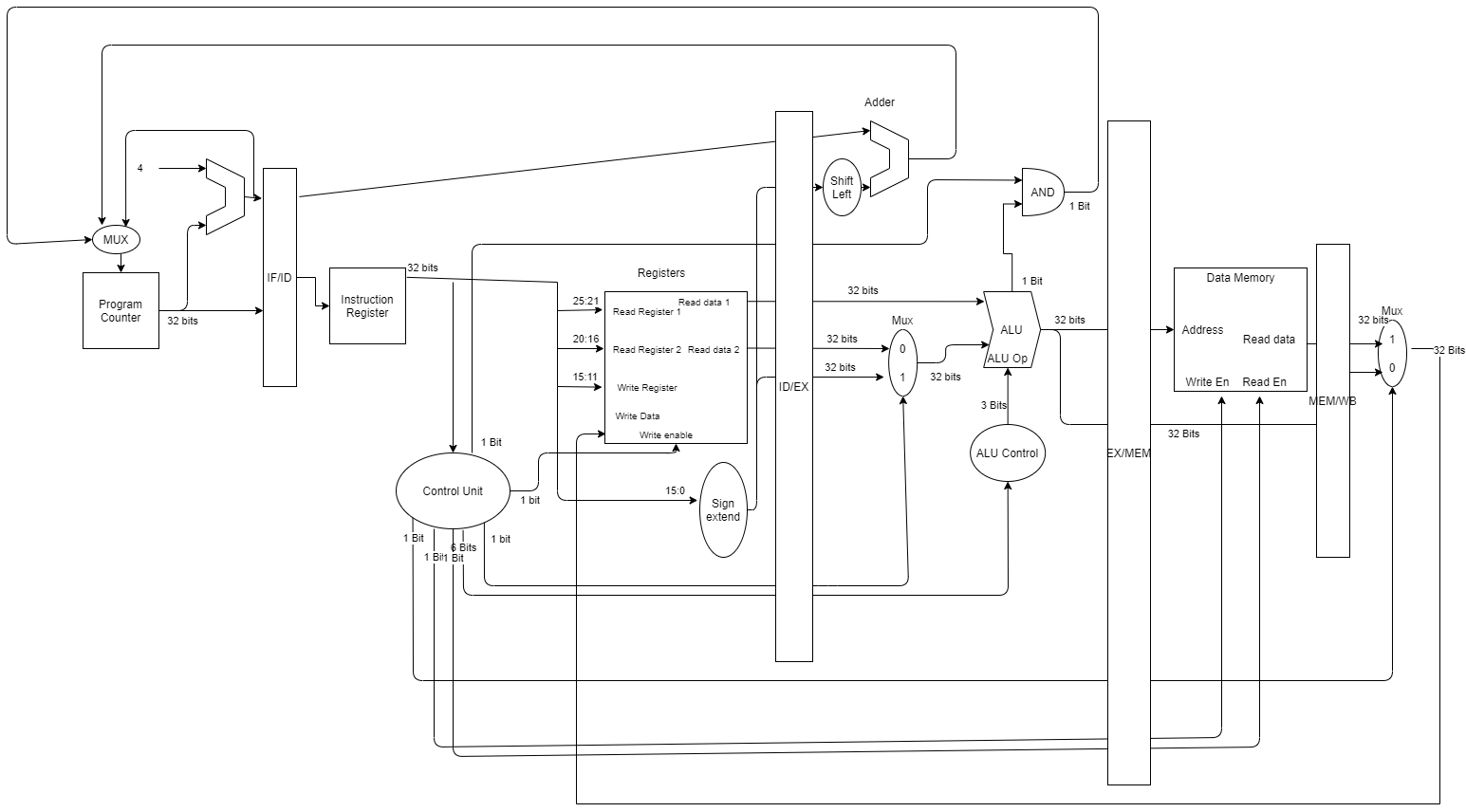
Due: 4/28/2021

Dr. Hasan

**Phase 3 Report**



Single-Cycle



Pipelined Diagram

**Side Note:**

Although not shown (for readability), all blocks except the following use a clock signal:

Add by 4, Control Unit, Sign extend, all MUXes, ALU Control, Shift left, 32 bit adder and AND.

**Objective:**

The objective of phase3 is to integrate all portions of milestone 1 and 2 as well as add data memory, a control unit and hardware for branch instructions. In addition, a single-cycle and pipelined version are both required. The pipeline is 5 stages and is shown above.

**Elaboration of each unit**

**Data Memory:**

This unit was created very similarly to the registers created in phase 2. This creates a user-defined type called memory. This memory is 32 bits wide and 32 bits in length. The length can be variable based upon whatever the application-user sees fit. This memory has a read\_en, write\_en, data in, data out, address and clock. When the clock is asserted high and the write enable is 1, the data in is written to the position set by the output of the ALU. This method is very similar to the registers. The separation of data and registers allows for better security of data so no important information is overwritten.

**Shift Left:**

The shift left unit is provided to assist with the branch instruction. In a normal MIPS processor, this usit will shift by 2 (effectively multiplying by 4) on the output of the signed extension unit. Sinc the memory is in bytes, this will only allow for the addressing of the first byte. For my implementation, memory is only in 32 bit words. This means that you can only address valid memory. In my case the shift left does not shift, but still passes through the unit. This shift is done using the shift instruction created by IEEE standards.

**Control Unit:**

The most complex of the units added in this phase is the control unit. This unit is asynchronous and uses the instruction fetched from memory to decode into control signals. These signals are as follows:

1. RegDst (register destination).
2. Branch.
3. MemRead (for reading from data memory).
4. MemtoReg (multiplexer control for instructio type.
5. ALUOp (see below).
6. MemWrite (for writing to data memory).
7. ALUSrc (register read or immediate value to ALU).
8. RegWrite (For register writeback).

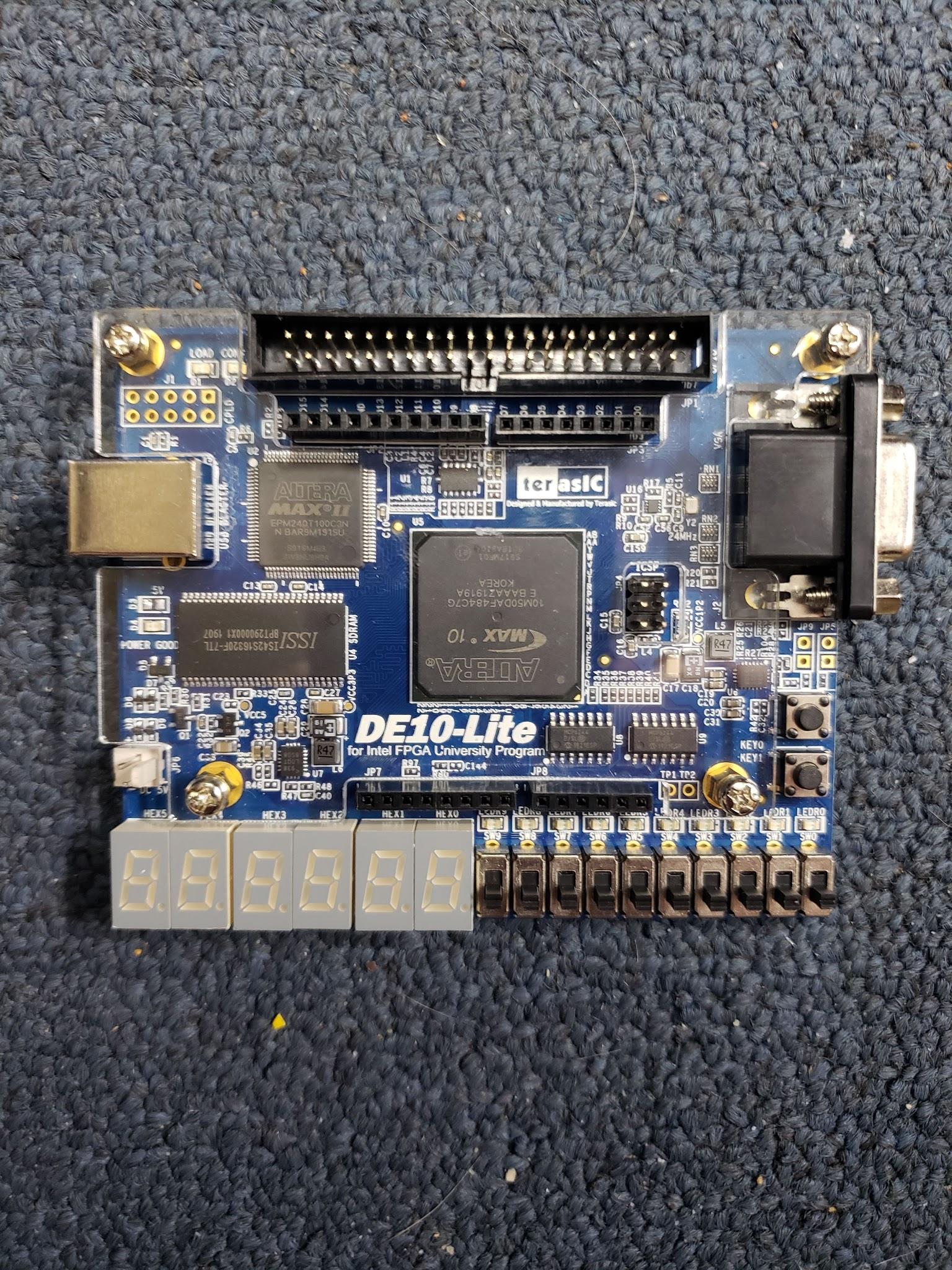
The ALUOp is used to send to the ALU control unit to determine which operation to execute. This operation is the last 6 of the instruction out of fetch. No further logic is performed. The unit takes in the instruction and uses if else statements to determine which signals should be high and which should be low. This is only done for the instructions used in the project and not for all. To implement a true MIPS processor, all instructions would need to be accounted for.

**ALU Control Unit:**

This unit receives data from the control unit and passes on the correct code to the ALU. This code is converted inside the ALU to perform the desired operation. The ALU control unit simply acts as a conversion from function code to ALU code.

**Device Selection:**

The device used for this project is an Altera DE10-Lite. This was the selection due to the reusability from the previous course ECE 4110. An image of this device is below:

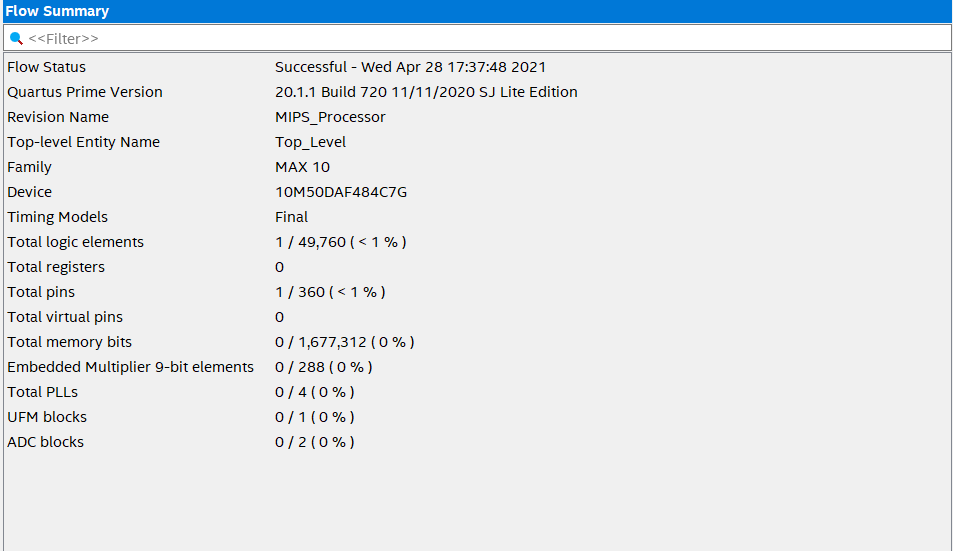


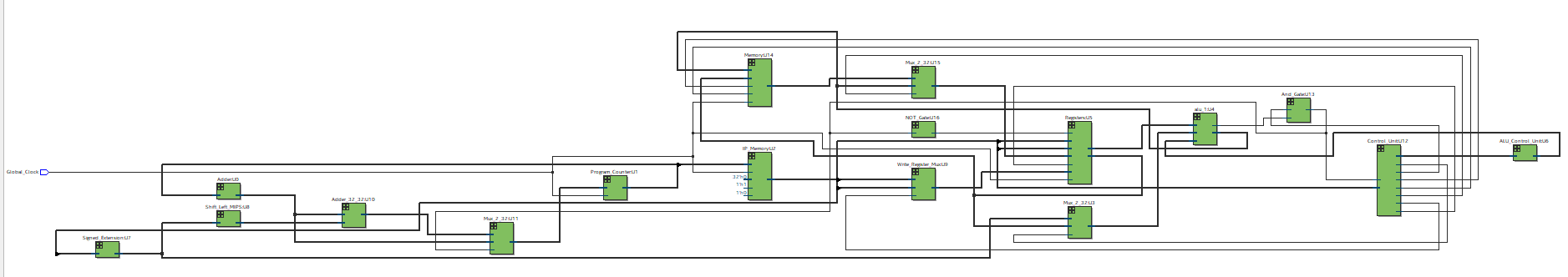
DE10-Lite FPGA

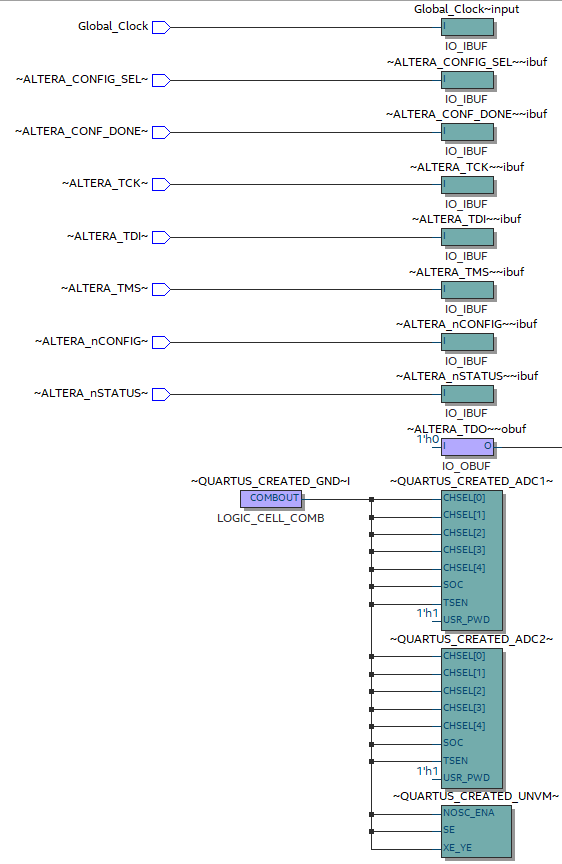
**Design Compilation**

**Single Cycle**

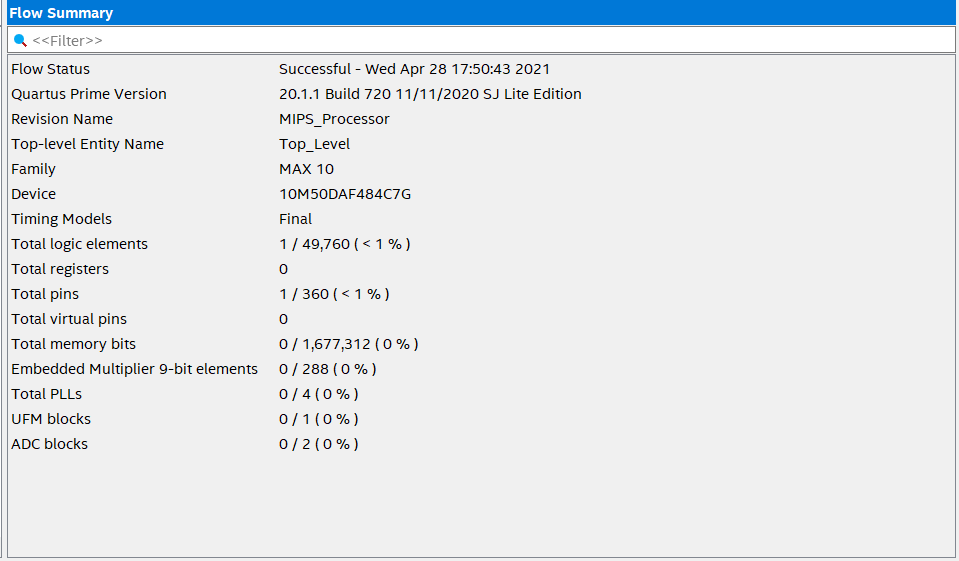
Here is the single cycle results for flow summary, RTL view and technology mapping (in order).

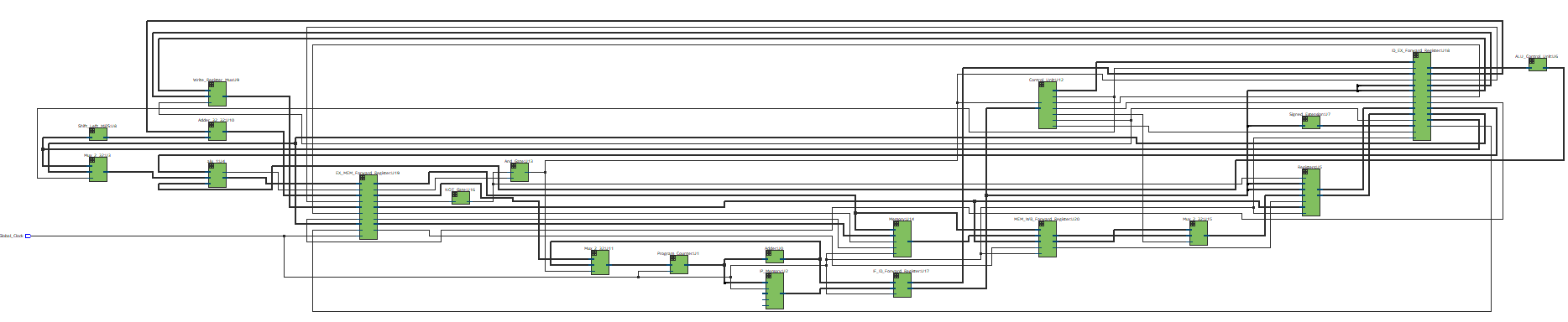


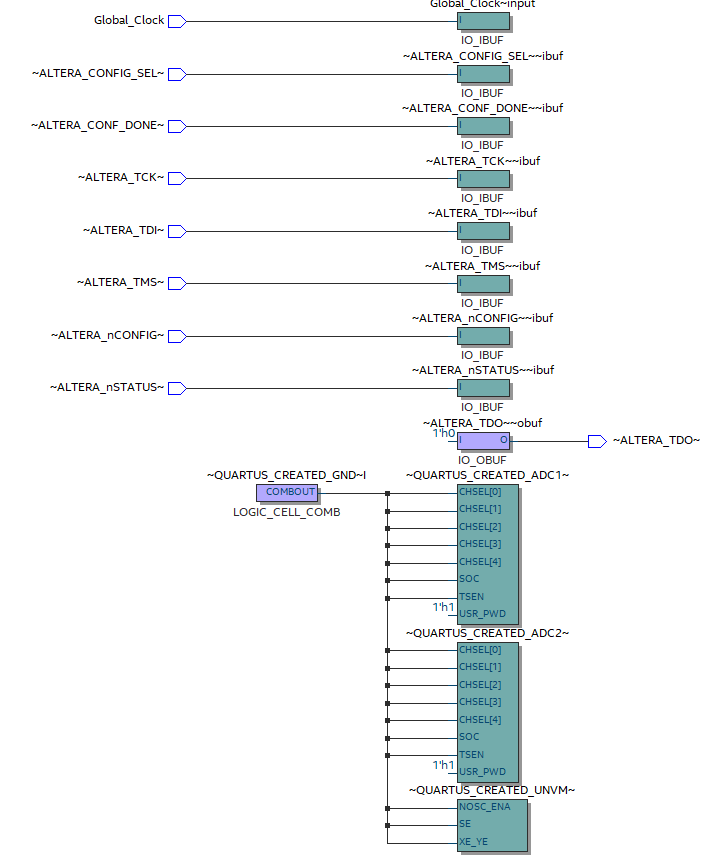




Here is the same information for the pipelined version.







**Testbench Elaboration:**

To preload values into registers, I was able to start the design with default values. I did the same with data memory (all being 0s in this case). To load the instruction memory, I used a .mif file. This file is used as a read-only file for the instruction memory to parse as the PC increments. The compiled code was created by using some test values as well as the required code as part of phase 3. These lines are as follows:

add $t3, $t0, $t1

sub $t4, $s0, $s1

and $t5, $t3, $t4

sw $t5, 0($zero)

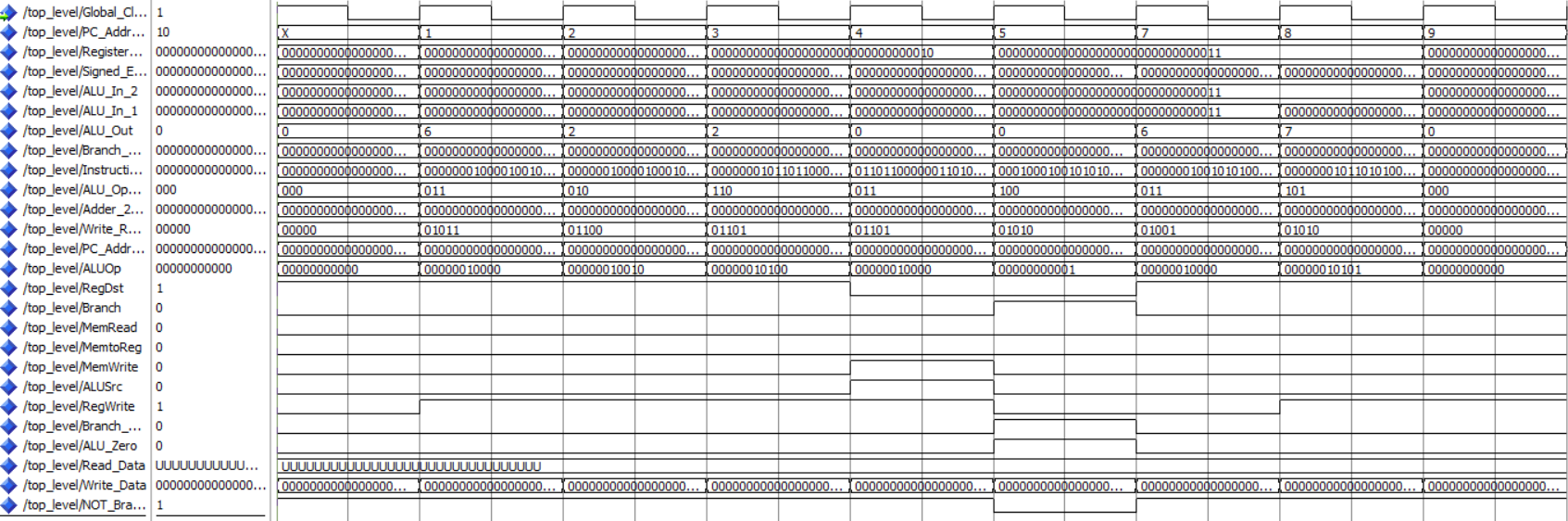
beq $t1, $t2, 07

add $t1, $t1, $t1

sw $t1, 100($t2)

or $t2, $t3, $t2

This allows for a range of instructions to be tested to show the durability of the code. The single-cycle waveforms are as follows:



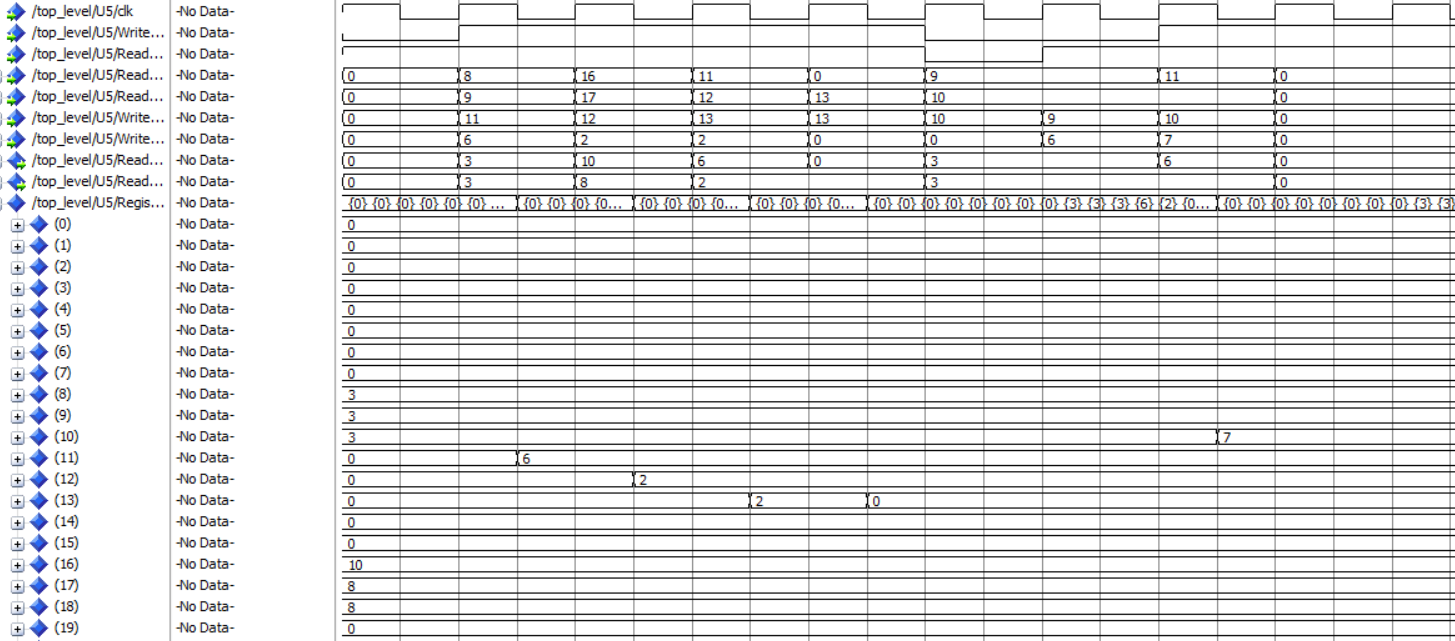
Modelsim Single-Cycle Top Level Results

**Single Cycle Elaboration:**

The main obstacle for phase 3 was implementing the branch instruction. PC\_Address is located beneath the Global\_Clock which drives the whole system. This address is put into unsigned for readability. As you can see, the branch is taken and the new address is 7, therefore skipping over 6. 5 is not skipped, however, there is a manual override in the control unit to account for this. In the case of a branch, no data will be written for the previous instruction. This cannot change the lost clock cycle, but will prevent unwanted changes.

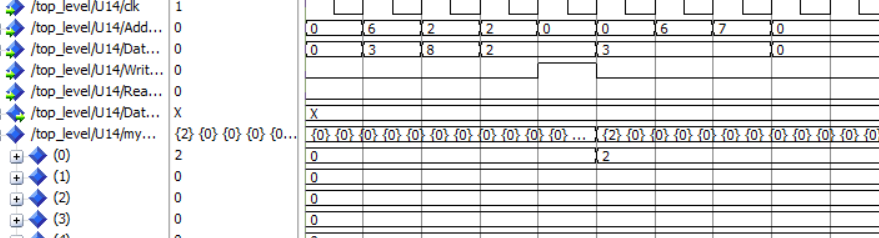
The branch is decided in the control unit, along with an AND gate for the ALU zero. When branch and ALU zero are asserted, the writeback is disabled and the branch address is passed to the PC.

Data is written to the memory when the clock edge and MemWrite are asserted high. This is seen during PC address 4. This is 1 cycle behind the instruction for the store instruction. This is due to the limitations of instruction memory. The macro used for this requires a clock cycle to “lock in” the address for reading. The counteract for this is to allow the waste of a clock cycle and prevent modification. The registers are shown below:



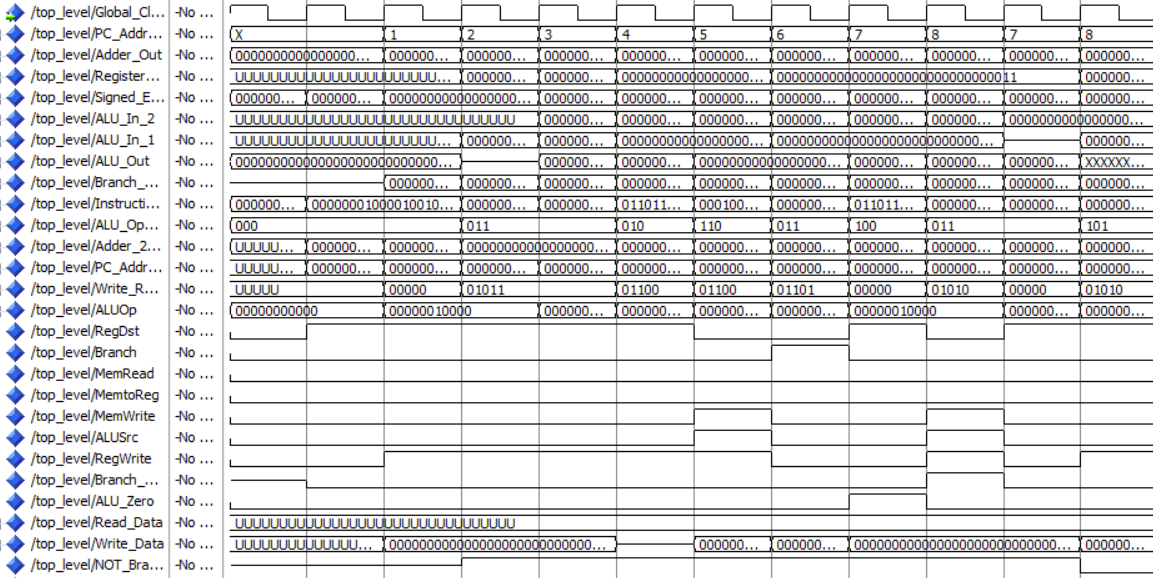
Modelsim Single Cycle Register Results

This diagram shows how the registers respond to the instructions. As you can tell from the assembly instructions provided above, the correct values are computed and stored in the assigned registers.



Modelsim Single Cycle Data Memory Results

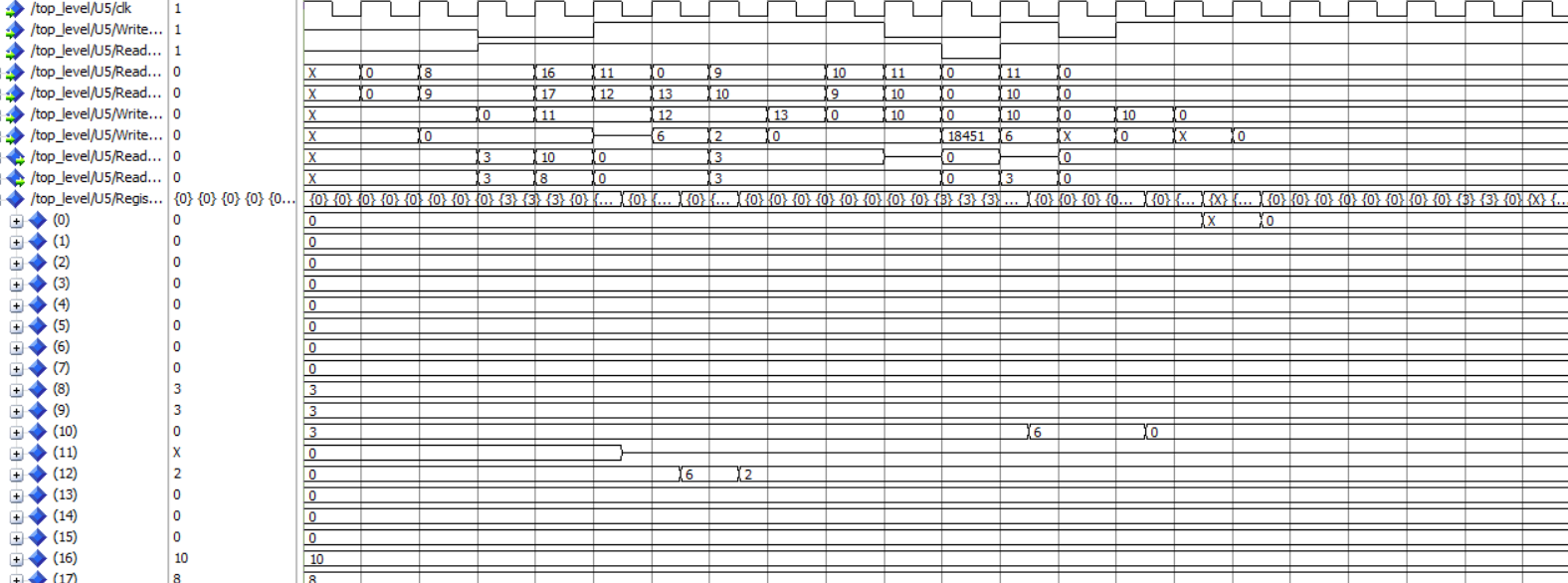
This diagram shows the storing of data. The sw command from the assembly correctly stores the register into data memory 0.



Modelsim Pipelined Top Level Results

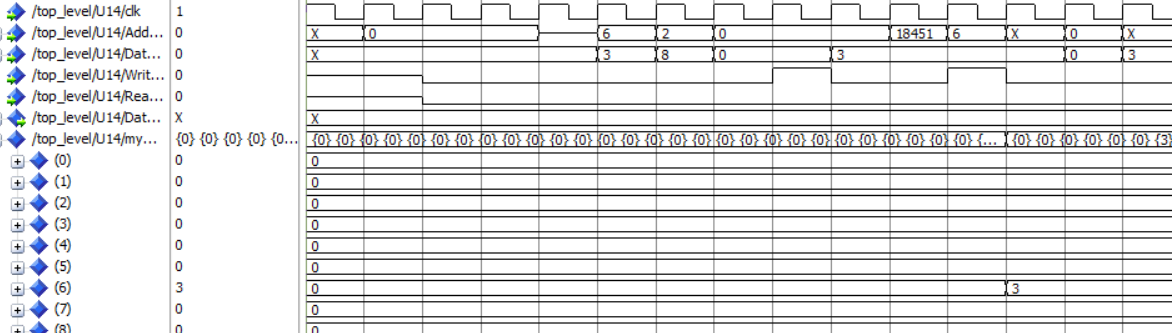
**Pipelined Elaboration:**

As mentioned for the single cycle processor, the branch instruction works here as well. When ALU zero is asserted high, and 1 cycle before, branch is asserted high, the branch is taken and the PC changes, This is different in the case of the pipelined implementation, is when the branch is taken relative to the PC address. In the single cycle, the branch is taken much sooner. When analyzing the pipelined variation, the branch instruction must propagate through the pipeline registers first. This causes a delay, however the branch is still taken.



Modelsim Pipelined Register Results

This diagram shows the register results from the pipelined version of the processor. This result is not correct. Due to time constraints, I am unable to resolve this. This is likely due to a few small logical errors in connecting the pipeline registers to the components instead of the single-cycle signals. The result should match that of the single cycle processor, with some delay form start to end.



Modelsim Pipelined Data Memory Results

This diagram shows the Modelsim output for the data memory. Similar to the register output, this implementation is also not correct. The control signals do propagate through the pipeline registers, however, the data and register number are incorrect. With more time, the issue could be diagnosed and resolved.

**Timing Analysis**

**Single Cycle:**

Despite my best efforts, I was unable to understand why the timing analyzer could not find the path. No registers are inferred and I do not know why this is. After much research, I was unable to find an answer. Therefore setup time, hold time and Fmax cannot be found for phase 3.

**Improvements:**

Unable to use metrics, an exact improvement cannot be calculated. However, it can be assumed that pipelined will greatly improve the speed of the system.

**Overhead:**

Adding the pipelined registers requires 5 at least 5 more registers. Depending on the maximum size, more would need to be added. This would cause a large amount of hardware to be used up. On a smaller device this could cause issues of resource management. However, with the large FPGA we use, it is not an issue.