Jacob Johnson
T00237585
Due: 3/23/2021

MIPS Processor Milestone 1

The block diagram in figure 1 shows a modified block diagram on the implementation I created.

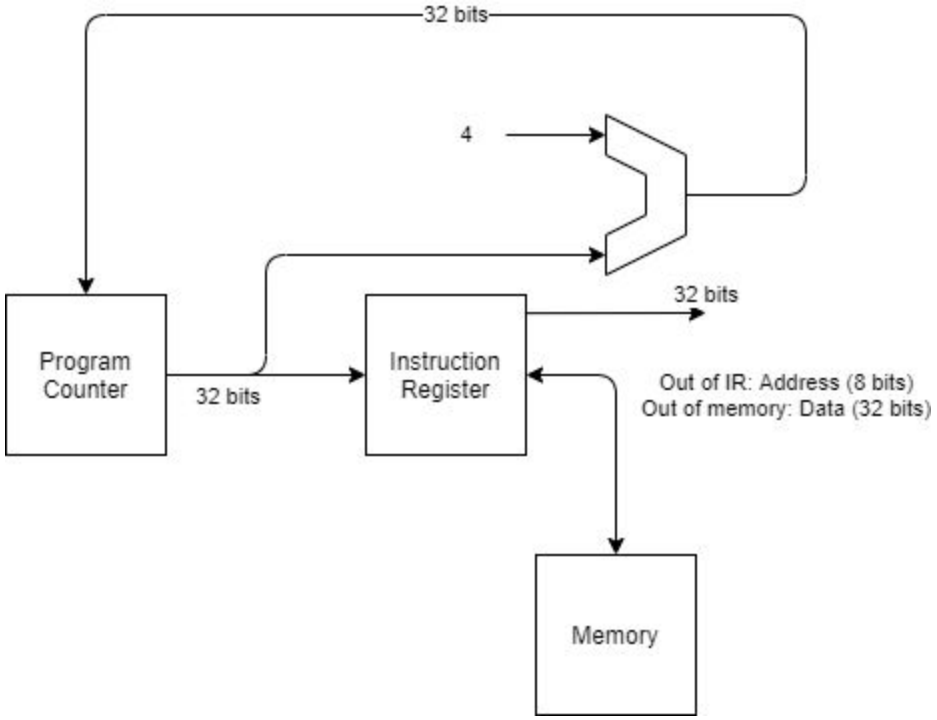

Figure 1

The objective of this project is to recreate a MIPS processor "fetch" function, This includes the program counter (PC), instruction register (IR), adder and memory. As part of this recreation, each "block" is its own file. The Top_level file allows for the components to be connected together using a structural implementation.

For the project, I wanted to break the problem into the smallest pieces possible, then synthesizing into larger functions. Each piece has a fairly simple function and is broken down below.

The program counter is a rising-edge triggered register, storing the data to be fed into the IR and adder. The adder uses the current PC value and adds 1 since each instruction is 4 bytes and memory is 4 bytes in size. The result is then fed back into the PC for a recursive function. The instruction register receives the PC, takes the lowest 8 bits and feeds it into the memory. The memory then returns the data at that location to the IR, where it is fed out of the system.

The device used for this project is the Altera DE10-Lite. Specifically, the chip used is 10M50DAF484C7G. Figure 3 has an image of the device. This is the device used for ECE 4110 and was specified for use for this project.

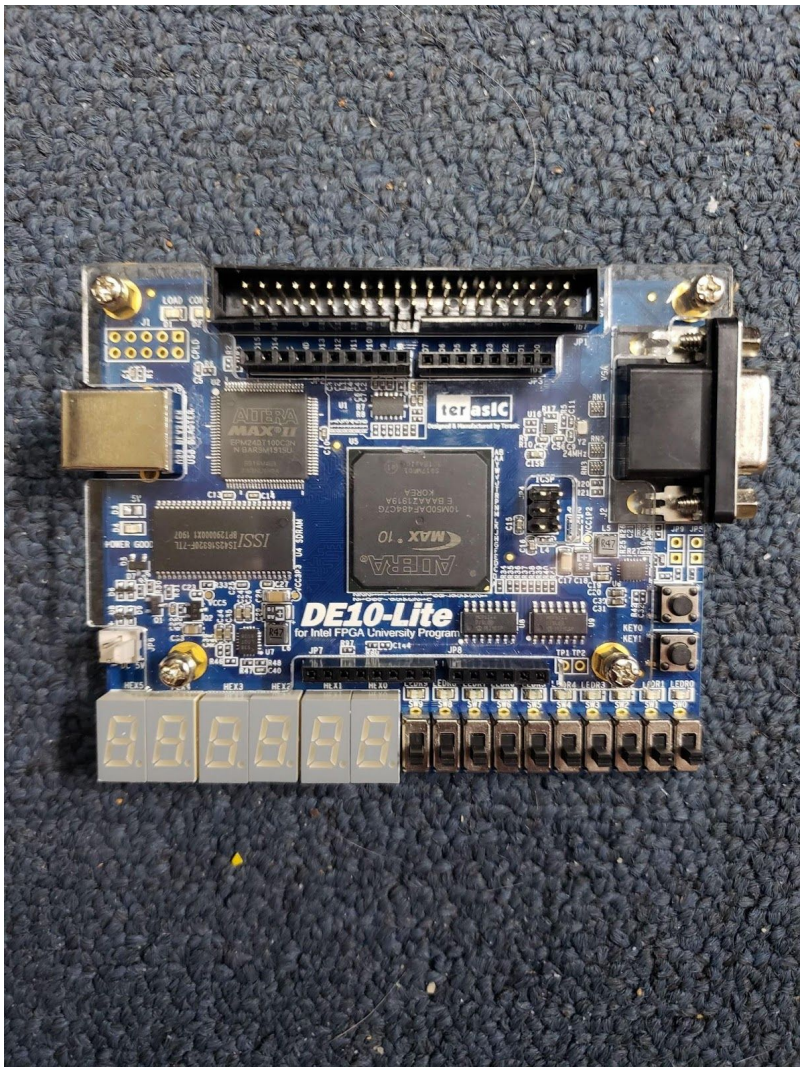

Figure 3

After compilation, the flow summary, RTL and technology map view are in figure 4, 5 and 6 respectively.

Flow Summary

<<Filter>>

Flow Status	Successful - Mon Mar 22 20:19:23 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	MIPS_Processor
Top-level Entity Name	Top_Level
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	44 / 49,760 (< 1 %)
Total registers	43
Total pins	33 / 360 (9 %)
Total virtual pins	0
Total memory bits	8,192 / 1,677,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 4

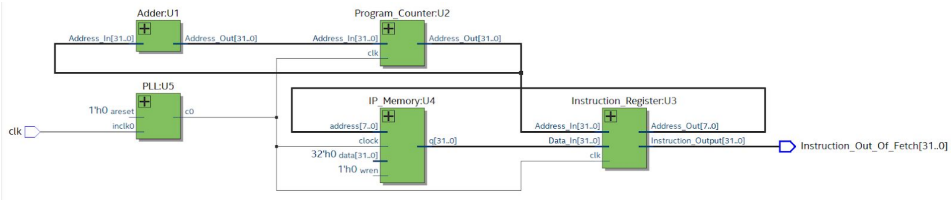

Figure 5

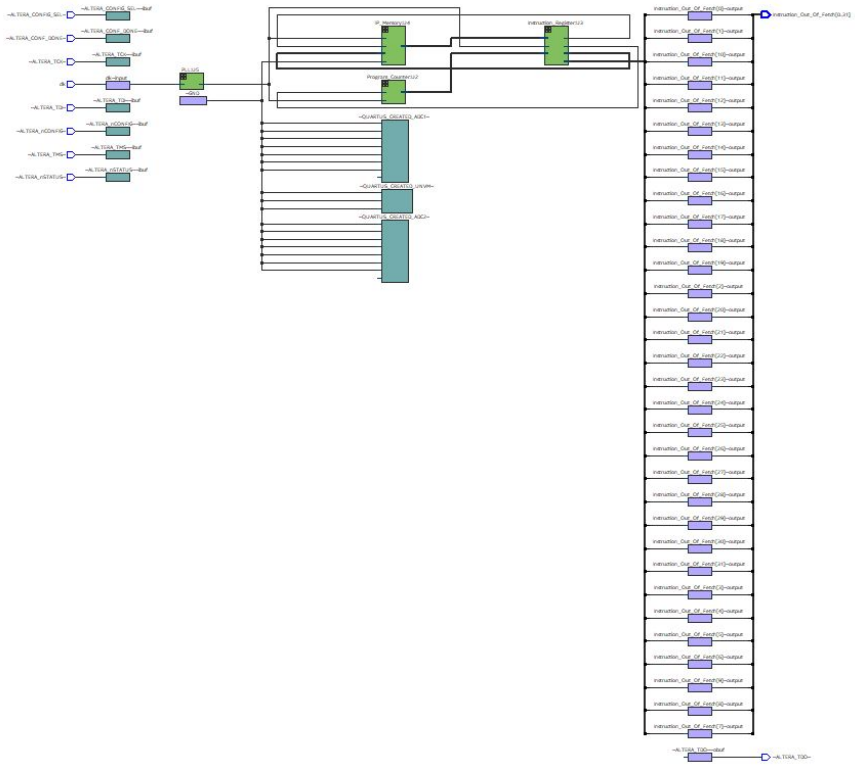

Figure 6

For my synthesis of the project, I did not use a testbench, but opted to use the preloaded memory instead. This method uses a .mif file to program the device. I used 5 lines from location 00_6 to 04_6. These instructions are labeled in my file via comments for easy reading. The instructions are as follows:

1. Loads 0 to $s1
2. Loads 2 to $s2
3. Adds $s2 to $s1 and stores to $t0
4. Adds $t0 to $s2 to $t0
5. Subtracts 1 from $t0 to $t0

The waveform created using Modelsim is located in figure 7. An in-depth analysis will follow.

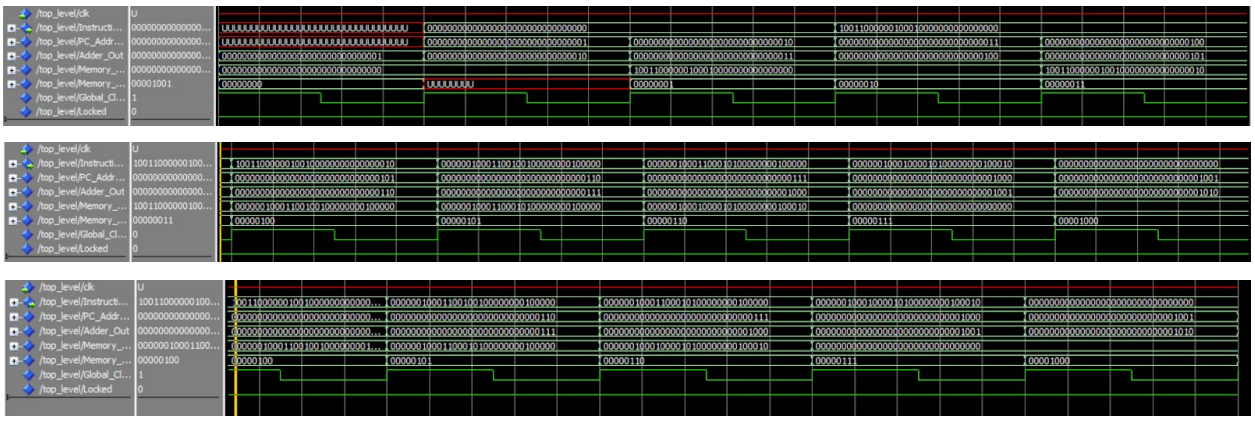

Figure 7

The top wave is the instruction out. No data is read until the third clock edge. This is due to the data delay from the time the PC address is given to the time the data is out. The required clock edges are from the IR, memory, then back to the IR respectively. This data is read from the memory which was preloaded by the .mif file. The data is as expected and appears in order of which it is placed in the .mif file. In class it was discussed that data is stored either as a byte or a word. In this case, I implemented a word storage. This means the addition of 1 to the PC every time will index every word, and therefore instruction, in order.

The PC address is next. This is the way the program keeps track of where the program is at. The address is fed to the IR and the adder. If the program ran forever, the PC would increase until it overflows then would restart. The data format is in binary, making it clear to see the addition of 1 every time.

The adder output is the same as the PC input since we have not yet implemented any other input to the program counter. In the future, it will need to be multiplexed to decide which input the PC should take. This is future work and is planned to be done. This wave trails the PC as it is shown by the delay of 1 clock cycle.

The wave labeled “memory address” is the output of the IR which takes the lowest 8 bits of PC and uses them to address the memory. The instruction register does the conversion in this case, but in an implementation, the wires would not need to be connected.

Lastly, the clock waveform is shown. This is not the same clock as is given by the chip. This is because the frequency at which the global clock operates is far too high and creates negative slack. This would also make the project unscalable, since this portion is likely not the slowest, further increasing the slack. Instead, a PLL is used to better control the frequency at which the design operates. This solves the issue of slack while maintaining the purpose of the design.

Using the timing analyzer portion of the flow summary, FMax can be produced as 350.39 MHz. In a similar fashion, setup time slack can be found at 97.146. This means the frequency at which I am currently operating the device is slower than possible. This was to avoid any future issues with timing. I largely predict the most delay in the arithmetic logic unit (ALU). By keeping this future design in mind, I wanted to allow as much time as needed. Though it cannot be predicted as a precise amount, this cannot be forgotten while designing other features of the project. Should this prove to be incorrect, the PLL can be adjusted to account for the correct frequency at which to operate the fetch unit.

In conclusion, this is a robust implementation of a MIPS fetch unit. The design utilizes a program counter, instruction register, adder and memory to retrieve data that a programmer sends to it. Keeping in mind that this is a smaller part of the larger project, I was vigilant in my design choices as to not negatively impact my future designs. As the project stands at the moment, another designer may use my current progress to implement the rest of the MIPS processor.