=============================================================

**Lab \_\_7\_\_ Report**

**Title \_Enhanced Single-cycl MIPS Processor\_\_\_**

**Semester \_\_\_\_Fall 2016\_\_\_\_\_ Date \_\_\_\_\_November 11, 2016\_\_\_**

**by**

**Name \_\_\_\_\_\_\_\_Kevin Manan\_\_\_\_\_ SID \_\_\_\_008355464\_\_\_**

**Name \_\_\_\_\_\_\_\_\_\_\_Nadim Sarras**\_\_\_\_ **SID \_\_0083826087\_\_**

**Lab Record**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Performed by (print name)** | **Checked by (print name)** | **Successfully Completed** | **Partially Completed\*** | **Failed or Not Performed\*** |
| Kevin Manan | Nadim Sarras | x |  |  |
| Nadim Sarras | Kevin Manan | x |  |  |

**\*Detailed descriptions must be given in the report.**

**San Jose State University**

**Department of Computer Engineering**

**CMPE 140Lab Report**

**Enhanced Single-cycle MIPS Processor**

**Purpose**

Lab 7 is an important lab that allows us to create our own MIPs processor on top of the pre-existing one. The instructions that the team added were MULTU, MFHI, MFLO, as well as JR, and JAL. These instructions expands the previous MIPS architecture significantly as the previous architecture does not have a form of multiplication and only had basic arithmetic. This is significant because now the team is able to augment designs that were previously built and generate our own version. This is an important skill for a computer architect, because industry requires a lot of teamwork as well as working with previous architectures.

**Materials & Software**

1. Vivado 2015.2
2. Verilog HDL
3. Nexys DDR4 FPGA Board
4. MIPs Greencard

**Task**

There were 4 main task that the team had to complete. The first task is the creation of the schematic that has the MULTU, MFHI, MFLO, JR, and JAL instruction as well as the previous instructions that were prebuilt. Once that task is complete, the team is able to move to the next object which is the creation of the data path and control unit. This objective requires the creation of individual test benches to unit test the operation of each instruction for both the control unit as well as the data path. Once these two objects are completed, the team is able work on the top level. This does not test the overall top level, but rather tests the combination of both the control unit and data path working synchronously. Finally, the team needs to be install the program to the Nexys FPA board and test outputs using the switches for inputs and the 8-segment LEDs for the outputs.

**Discussion & Results**

*Schematic:*

The schematic that the team created, seen below under the Appendix. The design features a new multiplier ALU that outputs a 64 bit wire that splits into two 32 bit wires that writes into our Register file that will hold both of our special registers. If the write enable is not selected, then our new register file will output either the High register or the Low register based off our Select Out input. The Multiplication implementation added to the schematic has been marked in maroon. The MFHI and MFLO is colored in green in our schematic and is controlled by a multiplexer, the output of the multiplexer is then feeded back to our register file and is placed based off Register Destination.

JR or Jump Register is colored orange in our schematic. The main components of Jump Register is the multiplexer that feeds the output of the register selected to the program counter flip-flop.

Finally the JAL instruction was designed with two components. The first would be the linking, this is done by feeding 31 to register address. This is a vital step because the return address is stored in register 31 as stated in our MIPS greencard. The other component is Jump instruction that receives 4 of the most significant bits of the Program Counter as well as the address of the JAL instruction shifted by 2.

The team designed the schematic with previous instructions in mind. By instantiating a new multiplier ALU instead of using the existing one, the team was able to generate accomplish the MULTU, MFHI, and MFLO instruction without conflicting the original or subtract instructions.

*Control Unit:*

The control unit was designed as a combinatorial circuit. The control unit is split into two sections: a main decoder and a alu decoder. The purpose of this split is to split the control signals.   
The main decoder mainly handles I-type and J-type instructions having specific control signals in the main decoder. The main decoder for these two instructions have specific outputs, while R-type instructions are generalized and will all have the same output for R-type instructions.

The ALU decoder in the original design only controlled the ALU. The team designed the ALU controller to include more R-type operations and is able to generate more specific control signals that allows R-type instructions to be more unique. In the ALU decoder, I-type and J-type are more generalized and is only affected by the ALU’s add or subtract functionality.

The splitting of the two decoders accomplishes a cleaner program that is able to generalize control signals based off their Type.

Testing the control signal is done by inputting test instructions for the function and alu operation. Once these signals have propagated in the circuit, we then test the output of control unit by comparing the output signals with an expected output.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Key: |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| R Type |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| I Type |  | **Inputs** |  | **Outputs** |  |  |  |  |  |  |  |  |  |  |  |  |
| J Type |  |  |  | **maindec** |  |  |  |  |  |  |  | **aludec** |  |  |  |  |
|  | **Instuctions** | **OP** | **Func** | **Jump** | **JAL** | **MemtoReg** | **MemWrite** | **Branch** | **ALU\_src** | **RegDest** | **RegWrite** | **ALU\_Ctrl [2:0]** | **JR** | **HiLoSel** | **PWE (product WE)** | **MFHILO** |
|  | MULTU | 000000 | 011001 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | xxx | 0 | x | 1 | 0 |
|  | MFHI | 000000 | 010000 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | xxx | 0 | 1 | 0 | 1 |
|  | MFLO | 000000 | 010010 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | xxx | 0 | 0 | 0 | 1 |
|  | JR | 000000 | 001000 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | xxx | 1 | x | 0 | 0 |
|  | JAL | 000011 | xxxxxx | x | 1 | 1 | 0 | 0 | x | x | 1 | xxx | 0 | x | 0 | 0 |
|  | add | 000000 | 100000 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 010 | 0 | x | 0 | 0 |
|  | sub | 000000 | 100010 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 110 | 0 | x | 0 | 0 |
|  | and | 000000 | 100100 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 000 | 0 | x | 0 | 0 |
|  | or | 000000 | 100101 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 001 | 0 | x | 0 | 0 |
|  | slt | 000000 | 101010 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 111 | 0 | x | 0 | 0 |
|  | lw | 100011 | xxxxxx | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 010 | 0 | x | 0 | 0 |
|  | sw | 101011 | xxxxxx | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 010 | 0 | x | 0 | 0 |
|  | beq | 000100 | xxxxxx | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 110 | 0 | x | 0 | 0 |
|  | j | 000010 | xxxxxx | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | xxx | 0 | x | 0 | 0 |
|  | addi | 001000 | xxxxxx | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 010 | 0 | x | 0 | 0 |

Table 1: Control Signal Output Table per instruction

*Data Path:*

The data path utilizes multiple modules, all of which serve a specific purpose in executing certain instructions. The main modules of the data path consist of; program counter, register file, and the product data register. The program counter is responsible for providing the next address of the next instruction to be executed. The register file is a key element of the datapath as it is where the inputs as well as some outputs of the instructions will be stored. The product data register is a custom module we designed, it consists of two 32 bit registers. One of these registers will contain information from the 32 MFHI bits after the product of the mult instruction is fed into the register. While the other register will intake the 32 MFLO bits representing the 32 least significant bits of the product wire after the MULT instructions executes. Since the data path does not utilize the instruction memory, data memory, or control unit, the data path had to account for the missing connections. Instead of having an instruction memory, the data path takes a 32 bit instruction input instead. In addition, since the data memory is excluded the data path gives an aluout and result output. Lastly, the modules within the datapath rely on the control signals given from the control unit. When testing the datapath both these controls signals and the instruction must be manipulated manually as an input into the data path, in order to effectively test the data flow of the data path. Table 1, below shows the methodology behind testing the datapath. The corresponding instruction and control signals were set to enable an instruction to activate. After of which, the test bench would look at the respective function of the instruction to verify the instruction was successful.

|  |  |
| --- | --- |
| Instruction | Component to Verify |
| add | Wd3 register |
| sub | Wd3 register |
| and | Wd3 register |
| or | Wd3 register |
| slt | Wd3 register |
| lw | Specified register |
| sw | aluout wire |
| beq | program counter |
| j | program counter |
| addi | Specified register |
| MULTU | Wire output from Mult module |
| MFHI | MFHI register in preg module |
| MFLO | MFLO register in preg module |
| JR | program counter |
| JAL | Register 31, program counter |

Table 2: Data Path Test Connections

*Top Level:*

The top level is done by compiling the main decoder, ALU decoder, and controller top level from the control unit and combining the data path under the mips module. Once the mips module is completed, the team is then able to work on a test bench for the mips module. The test bench for the mips module is done by inserting 32 bit instructions and then checking vital points of the top level outputs. Vital points are the aluout and the writedata. The aluout is vital because it’s able to display the results of the add and addi instruction. Another vital point of testing would be the writedata output. This output is used when MFLO instruction is fed into the mips module. The test bench is designed by the following algorithm:

addi $t0, $0, 1

addi $t1, $0, 2

add $t1, $t0, $t2

multu $t1, $t0, $t3

mflo $t4

Of course the top level is not tested by MIPs instructions, however, the team was able to generate these instructions by cycling through the clock and in each cycle input the 32 bit instruction that is represented as the above MIPs code. Once these instructions execute, the vital points mentioned earlier are tested. The team was able to generate an output of the top level with majority of it functioning. The current test bench tests for add, addi, multu, and mflo is operational as well as all of the other extended instructions. The team chose to test these signals primarily because the following lab requires the team to work on factorials and the multiply as well as the mflo/mfhi are the main instructions that will be utilized. Since timing was an issue, the team decided to create a test bench that tests the new instructions that were most vital for completing the next labs. All instructions added to the single cycle MIPS Processor were successful.

*FPGA Board:*

The FPGA board was an effective tool in order to debug and implement the top level design of the enhanced single-cycle MIPS processor. The switches were used to feed in instructions and inputs into the FPGA board. While the LED displays were configured to show the corresponding output of the instruction. The Program Counter would increment on the left side of the LED’s indicating what instruction was currently executing. While the right side of the LEDS showed the contents of what register was selected using the slide switches. All instructions were successful.

**Conclusion**

Ultimately, the team was able to accomplish the entire lab with no issues. The control unit and test benches were successful and the top level represented all of our instructions being operational. This lab encompasses many fundamental skills for creating a MIPs processor and educated the team about augmenting instructions on an already existing single cycle MIPs processor. Although the team ran into many hindrances and issues, the team had learned how to extend instructions, create test benches for different simulations, and ultimately understand how the MIPs processor operates in a lower level.

**Appendix**

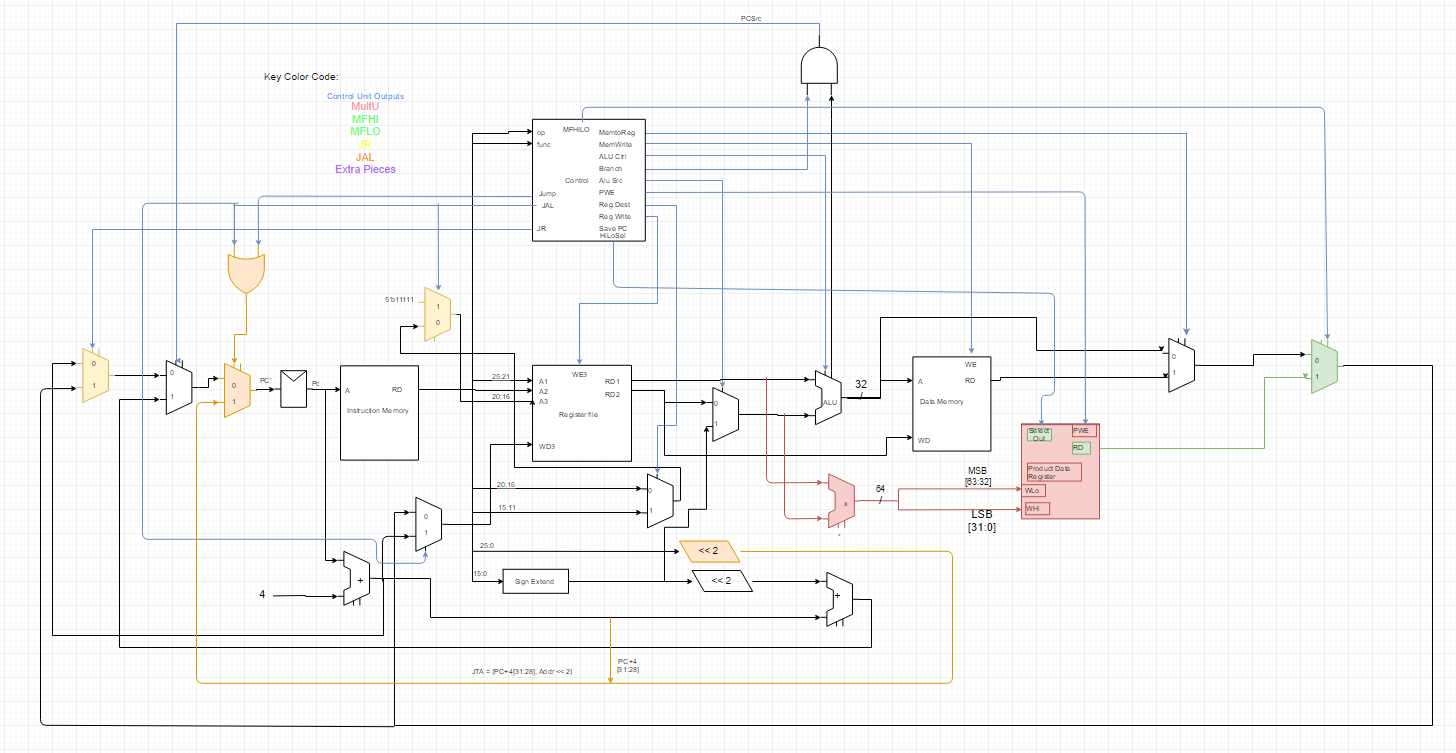


Figure 1: Data Path Schematic with new instructions

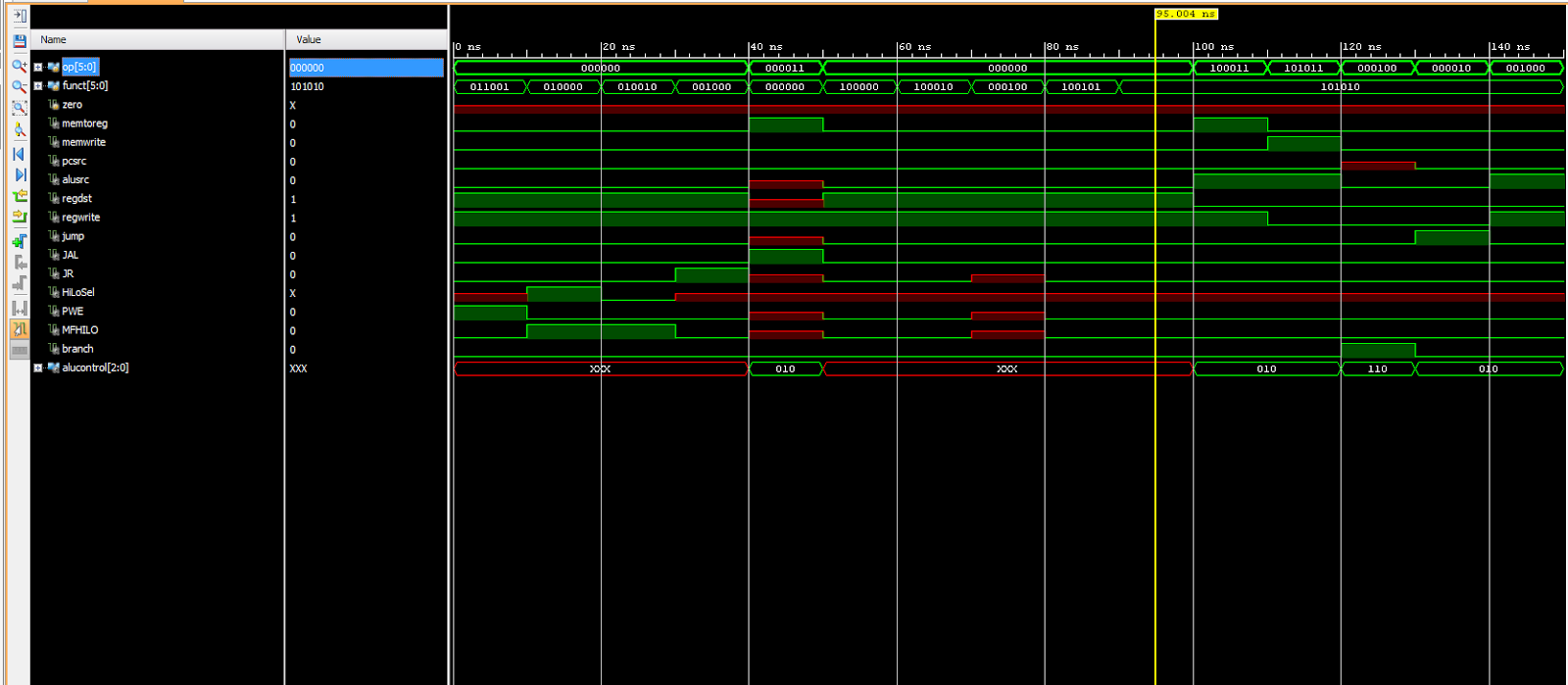
****

Figure 2: Control Unit Waveform Results

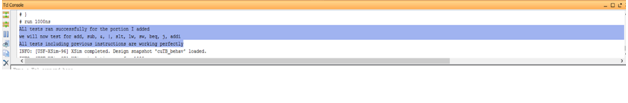
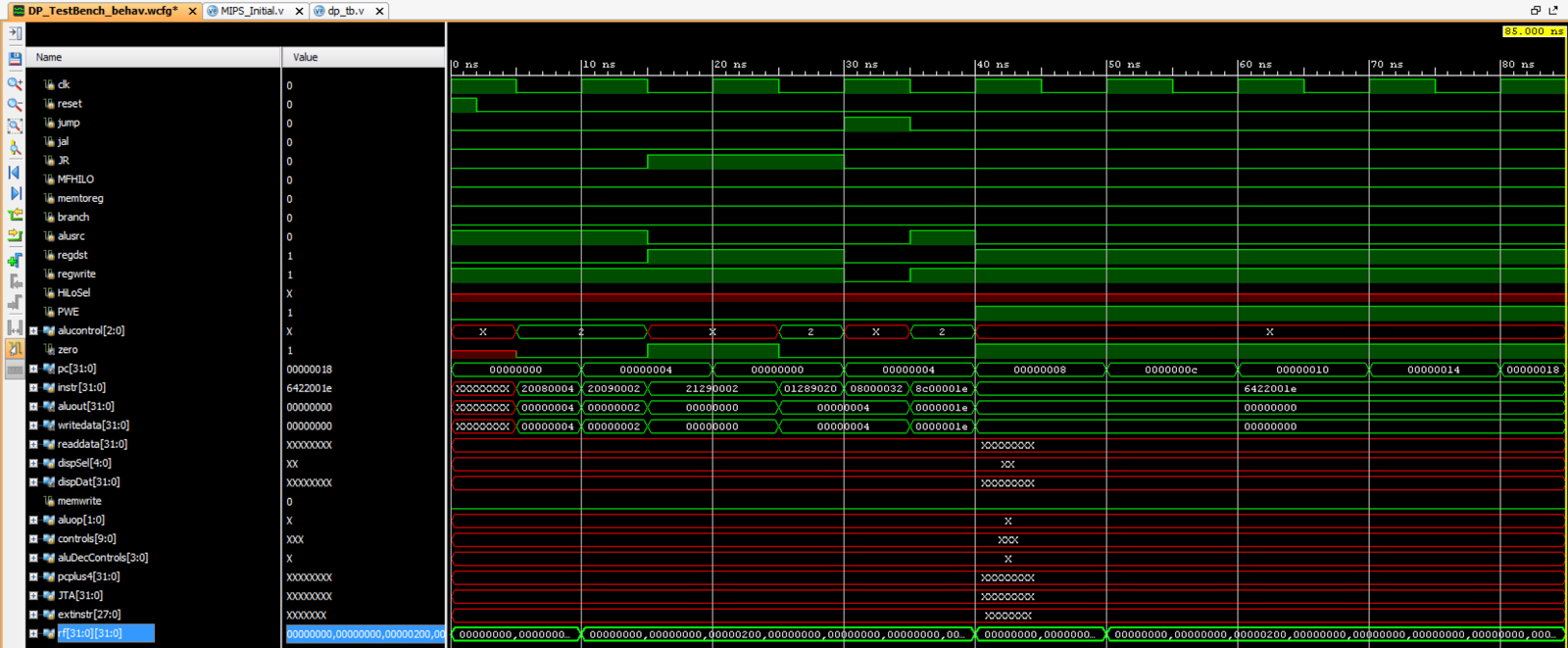


Figure 3: Control Unit Test Bench Console Output

****

Data Path Simulation Waveform Results

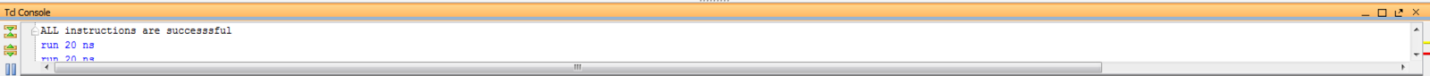
****

Figure 4: Data Path Test Bench Results

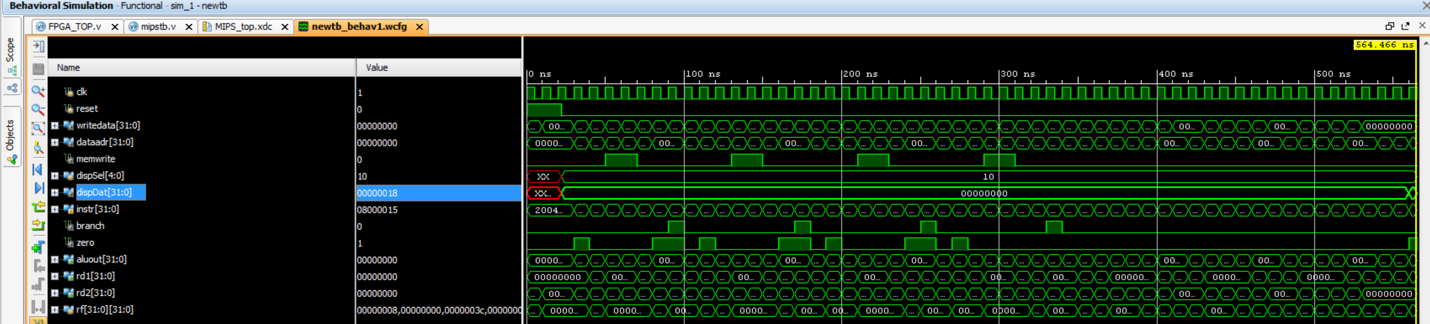


Figure 5: Top Level Simulation Waveform Results

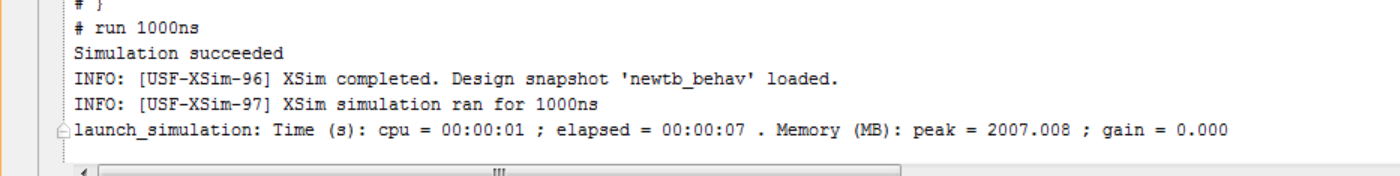


Figure 6: Top Level Test Bench Results

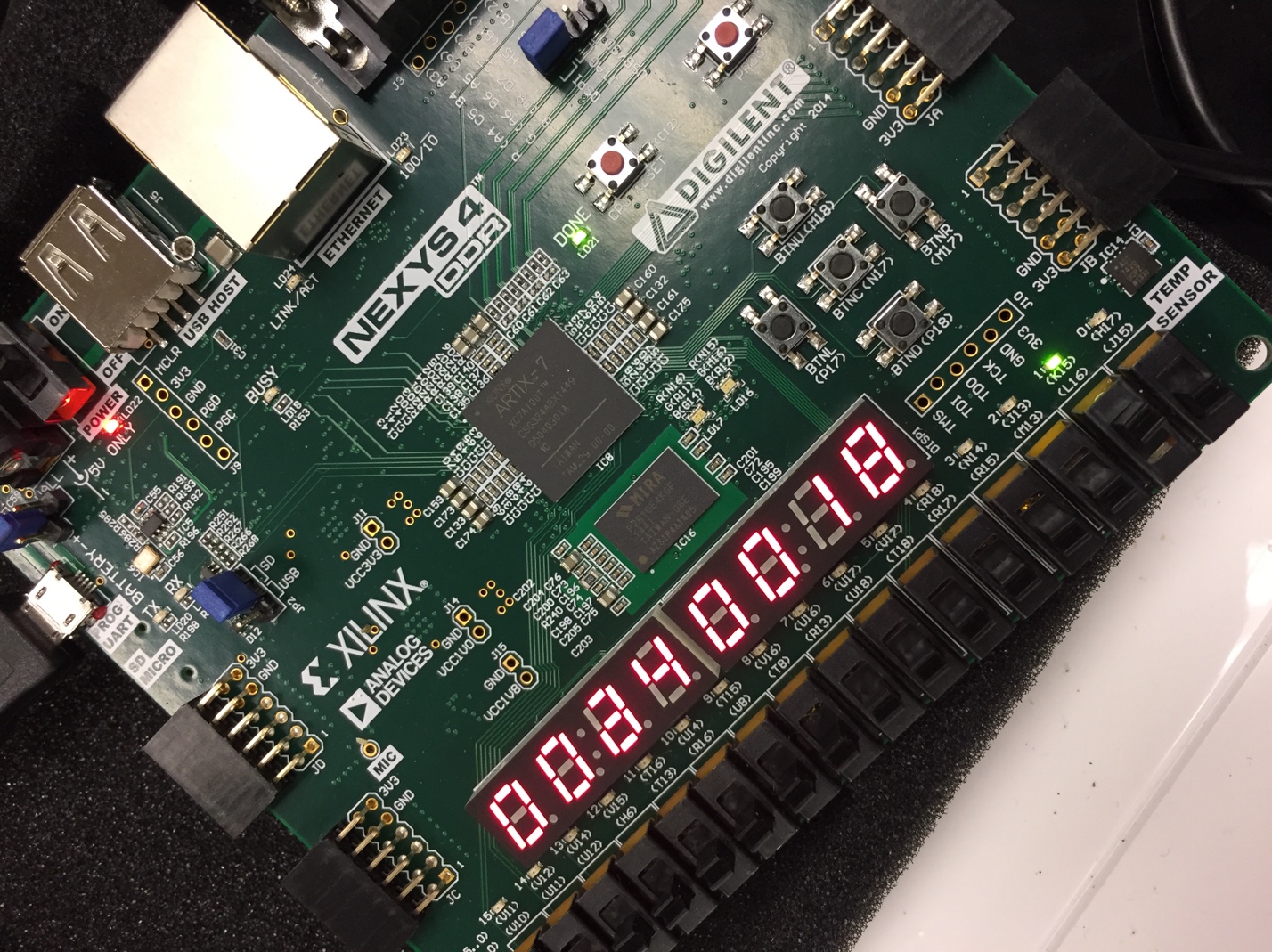


Figure 7: FPGA Implementation

\*All source code uploaded in separate zip file to canvas.