Midterm Project

Hardware Design - ECE 311

October 25, 2022

Ahmad Malik

Professor Tim Hoerning

Abstract

The purpose of the lab is to discuss the MIPS and ARM processors architecture. Then using Vivado Design Suite, we built a basic single-cycle 32-bit MIPS processor, constructed the necessary functional blocks such as ALU, registers, controller, etc, and ran a simulation through the use of a test-bench. I then attempted to test the CPU on the. Finally, we considered the possibility of pipelining our CPU for faster performance.

Introduction

In the past few labs, we were successfully able to use code from verilog files and used the Vivado program to program FPGAs on the ZedBoard. We coded simple blocks of logic that represent typical computer components such as RAM, multiplexers, decoders, etc. In this project the goal is try to fully combine all of the blocks of code to build a functioning computer that can read machine instructions to read, write, and output data appropriately. We must decide on two possible computer architectures: MIPS or ARM. We will research the strengths and weaknesses of each architecture, and ultimately chose one to implement into the Zedboard. The first iteration of the process will be single-cycle, and later we will consider pipelining to increase the speed of our computer

Part 1: Overview

MIPS Architecture

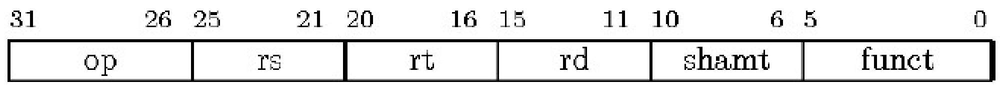
One of the most well-supported CPU architectures is the MIPS architecture, which has a large infrastructure of industry-standard tools, software, and services to facilitate quick, dependable, and affordable development. It's popularity is due to its simple and easy to follow implementation, but at the cost of efficiency. It supports 32 registers, which are separated by their specific uses. Register $0 holds binary 0 and register $1 is normally reserved for the assembler. Refer to the table below that displays the purposes of each register in MIPS architecture:

Name	Register #	Usage
$zero	0	The constant value 0
$at	1	Used by assembler
$vo-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return pointer

There are three kinds of instructions for the MIPS architecture: I, J, and R type.

R type: takes in three registers as arguments. “rs” is the source register, “rt” is the target register, and “rd” is the destination register. The “op” and “shamt” field is typically filled with 0’s for R-type and the “funct” specifies the function bits which are used to separate the specific instruction. See below for the structure of a typical R type instruction for 32 bit wide MIPS architecture:

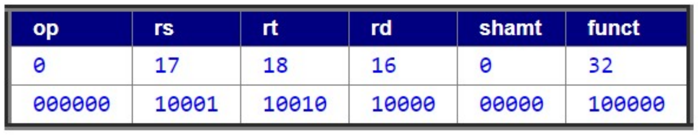
R-format (Register)



op	6-bit opcode
rs	5-bit source register specifier
rt	5-bit target register specifier
rd	5-bit destination register specifier
shamt	5-bit shift amount
funct	6-bit function field

***Example*:**

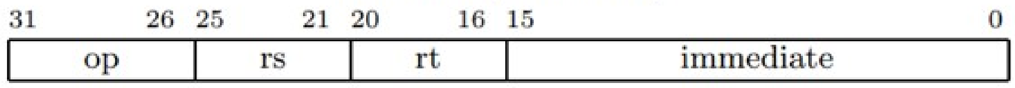
```
add $s0, $s1, $s2    (registers 16, 17, 18)
```



op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
000000	10001	10010	10000	00000	100000

I type: these instructions support operations such as Load, Store, Branch, and Immediate ALU operations. Typically one or two register file locations are specified as well as a 16-bit immediate value which may be used as an operand or an address.

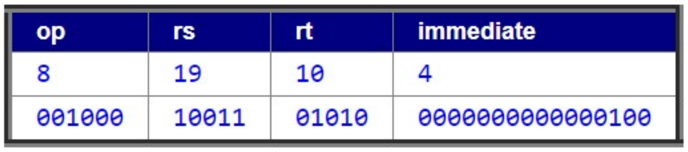
I-format (Immediate)



op	6-bit opcode
rs	5-bit source register specifier
rt	5-bit target register specifier
immediate	16-bit immediate

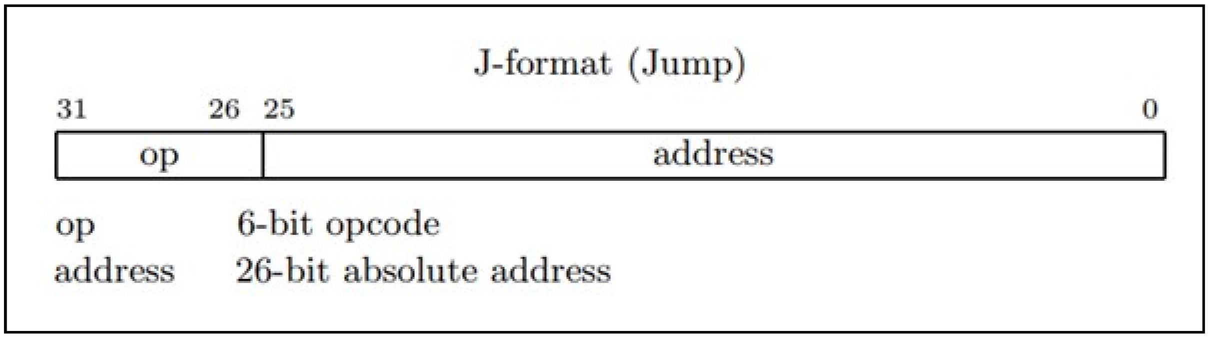
Example:

```
addi $t2, $s3, 4        (registers 10 and 19)
```



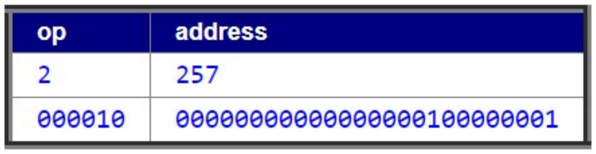
op	rs	rt	immediate
8	19	10	4
001000	10011	01010	0000000000000100

J type: these instructions require a 26-bit coded address field to specify the target of the jump. During execution, the high order four bits of the PC (the address of the instruction following the jump), the 26 bits of the target field, and two 0 bits are concatenated to create the 32-bit jump target address.



op	6-bit opcode
address	26-bit absolute address

Example: Jumping to Address 1028

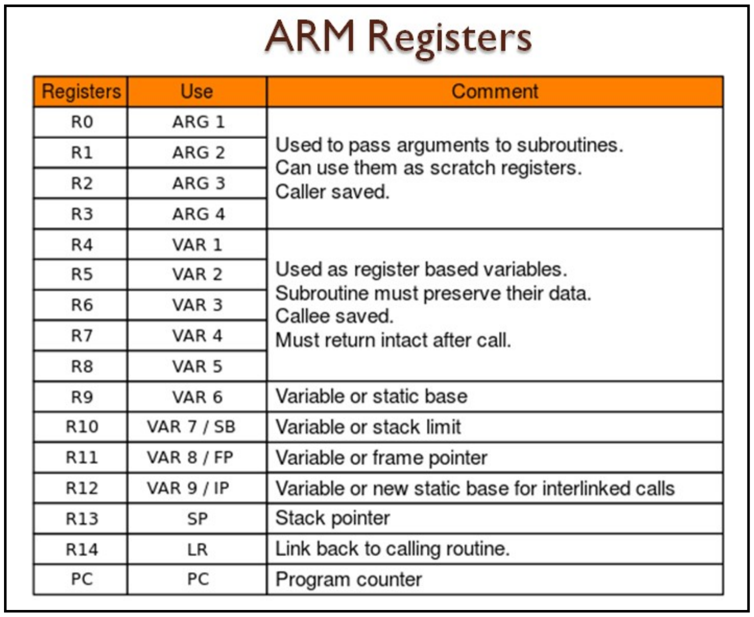


op	address
2	257
000010	00000000000000000100000001

ARM Architecture

ARM is a family of reduced instruction set computing (RISC) architectures for computer processors, with broad range of applications. It was formerly known as Advanced RISC Machine and is still commonly written as such today. When compared to complex instruction set computing (CISC) architecture processors like the x86 processors, RISC processors typically require fewer transistors, which benefits cost, power consumption, and heat dissipation. These characteristics are ideal for small, battery-operated devices, such as smart-phones, laptops, tablets, and other embedded systems, however they are also somewhat helpful for servers and desktops. ARM is a power-effective option for supercomputers, which use a lot of electricity.

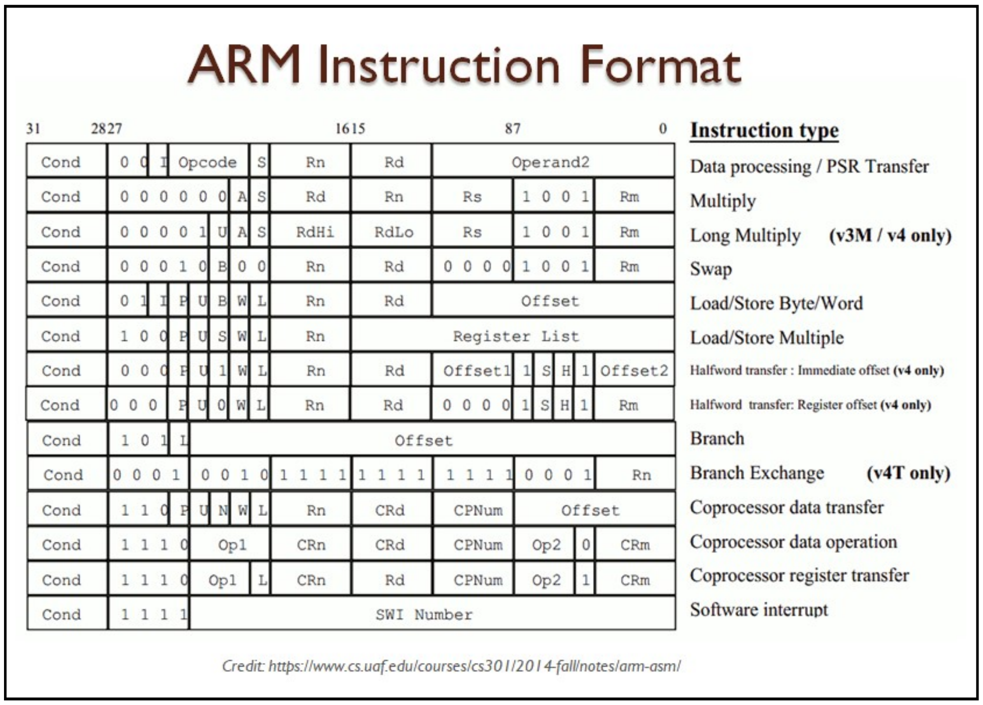
Unlike MIPS, ARM supports 16 general purpose registers. Despite this, ARM has a greater throughput and efficiency than MIPS because ARM processors support 64-bit data bus, so more data is processed per clock cycle. Below is a table describing the purpose of the 16 registers typically employed in ARM.



ARM Registers

Registers	Use	Comment
R0	ARG 1	Used to pass arguments to subroutines. Can use them as scratch registers. Caller saved.
R1	ARG 2	
R2	ARG 3	
R3	ARG 4	
R4	VAR 1	Used as register based variables. Subroutine must preserve their data. Callee saved. Must return intact after call.
R5	VAR 2	
R6	VAR 3	
R7	VAR 4	
R8	VAR 5	
R9	VAR 6	Variable or static base
R10	VAR 7 / SB	Variable or stack limit
R11	VAR 8 / FP	Variable or frame pointer
R12	VAR 9 / IP	Variable or new static base for interlinked calls
R13	SP	Stack pointer
R14	LR	Link back to calling routine.
PC	PC	Program counter

The ARM instruction set can be divided into six broad classes of instructions: Branch, Data-processing, Load and Store, Coprocessor, and Exception-generating instructions. Due to the complexity, ARM instructions can be identified using their opcode and the conditional flags. Below is chart depicting an ARM architecture with 32-bit instructions. Clearly it includes more instruction fields due to its ability to handle conditional flags.



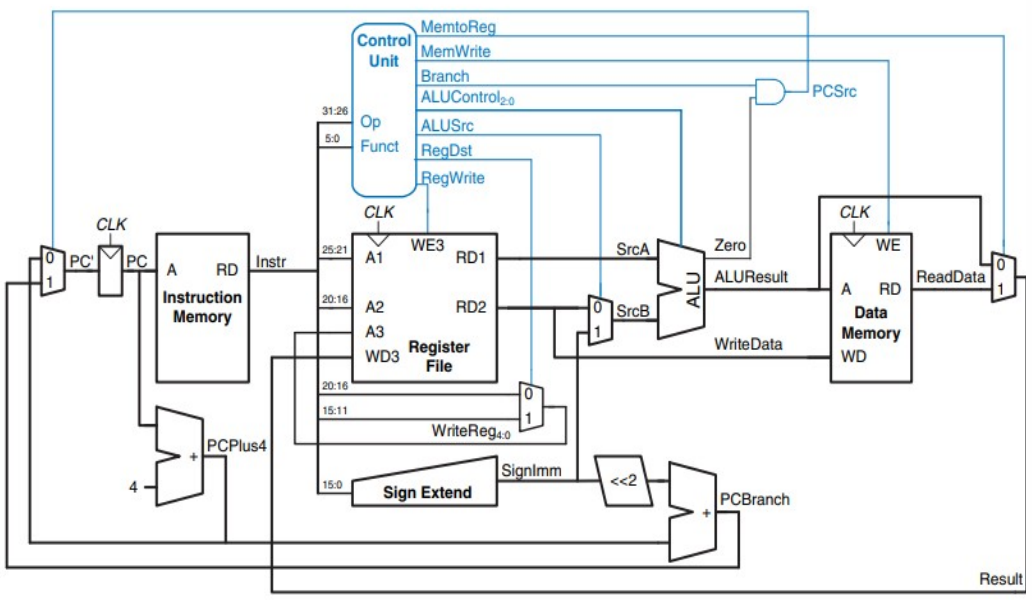
ARM Instruction Format

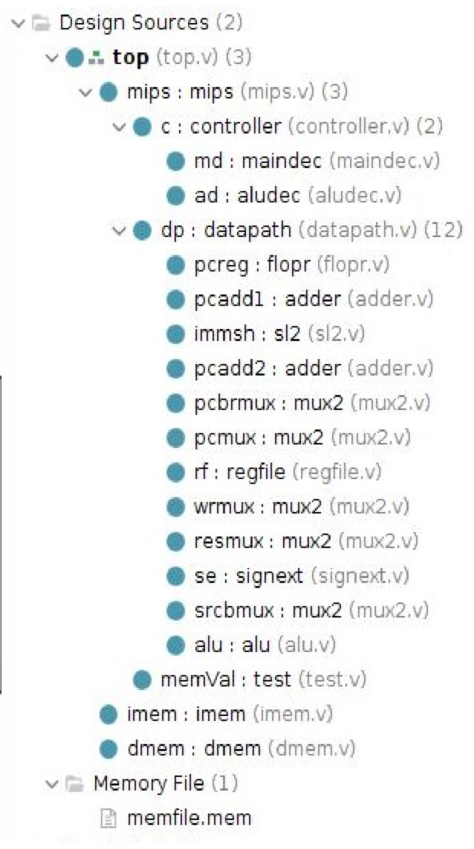
31 28	27	16	15	8	7	0	Instruction type
Cond	0 0 I Opcode S	Rn	Rd	Operand2			Data processing / PSR Transfer
Cond	0 0 0 0 0 0 A S	Rd	Rn	Rs	1 0 0 1	Rm	Multiply
Cond	0 0 0 0 1 U A S	RdHi	RdLo	Rs	1 0 0 1	Rm	Long Multiply **(v3M / v4 only)**
Cond	0 0 0 1 0 B 0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm	Swap
Cond	0 1 I P U B W L	Rn	Rd	Offset			Load/Store Byte/Word
Cond	1 0 0 P U S W L	Rn	Register List				Load/Store Multiple
Cond	0 0 0 P U 1 W L	Rn	Rd	Offset1	1 S H 1	Offset2	Halfword transfer : Immediate offset **(v4 only)**
Cond	0 0 0 P U 0 W L	Rn	Rd	0 0 0 0	1 S H 1	Rm	Halfword transfer: Register offset **(v4 only)**
Cond	1 0 1 L	Offset					Branch
Cond	0 0 0 1 0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn	Branch Exchange **(v4T only)**
Cond	1 1 0 P U N W L	Rn	CRd	CPNum	Offset		Coprocessor data transfer
Cond	1 1 1 0 Op1	CRn	CRd	CPNum	Op2 0	CRm	Coprocessor data operation
Cond	1 1 1 0 Op1 L	CRn	Rd	CPNum	Op2 1	CRm	Coprocessor register transfer
Cond	1 1 1 1 SWI Number						Software interrupt

Credit: https://www.cs.uaf.edu/courses/cs301/2014-fall/notes/arm-asm/

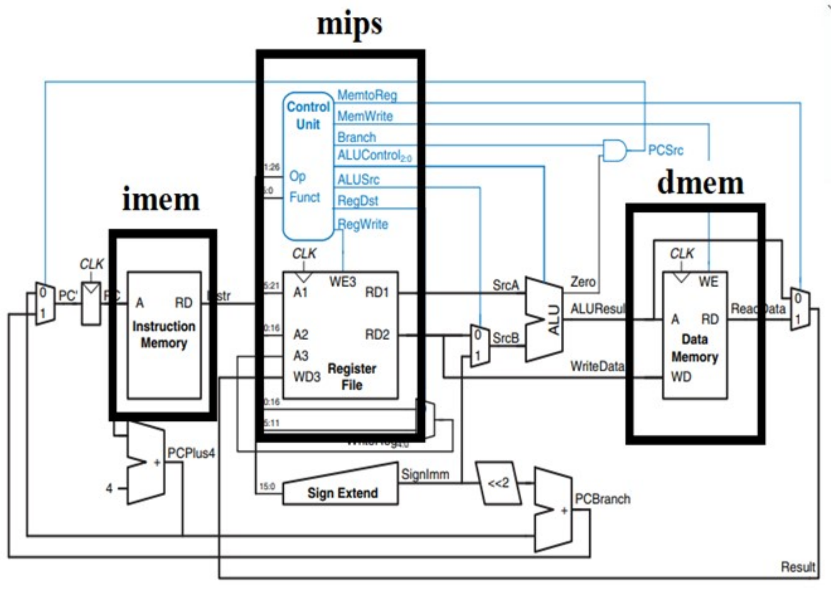
Part 2: Vivado Implementation

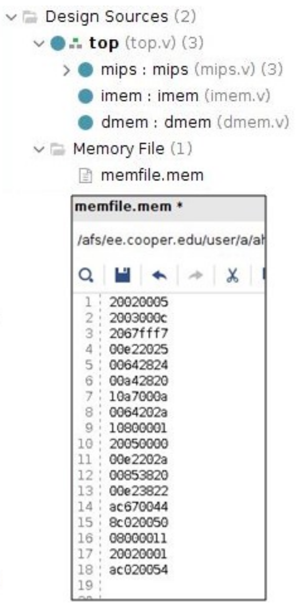
In my implementation, I decided to work off of the MIPS Implementation outlined in the course textbook. Below is the block diagram incorporating all the modules and sub modules that are used to build a multi-cycle processor. On the right is the hierarchy of verilog files that were used to build the CPU.




```
Design Sources (2)
  top (top.v) (3)
    mips : mips (mips.v) (3)
      c : controller (controller.v) (2)
        md : maindec (maindec.v)
        ad : aludec (aludec.v)
      dp : datapath (datapath.v) (12)
        pcreg : flopr (flopr.v)
        pcadd1 : adder (adder.v)
        immsh : sl2 (sl2.v)
        pcadd2 : adder (adder.v)
        pcbrmux : mux2 (mux2.v)
        pcmux : mux2 (mux2.v)
        rf : regfile (regfile.v)
        wrmux : mux2 (mux2.v)
        resmux : mux2 (mux2.v)
        se : signext (signext.v)
        srcbmux : mux2 (mux2.v)
        alu : alu (alu.v)
      memVal : test (test.v)
    imem : imem (imem.v)
    dmem : dmem (dmem.v)
  Memory File (1)
    memfile.mem
```

Below are the main blocks and their corresponding verilog files. Notice the .mem file code on the bottom left that holds the machine that will execute various instructions and result in a value of 7 being written in address 84 register 2 upon successful execution.

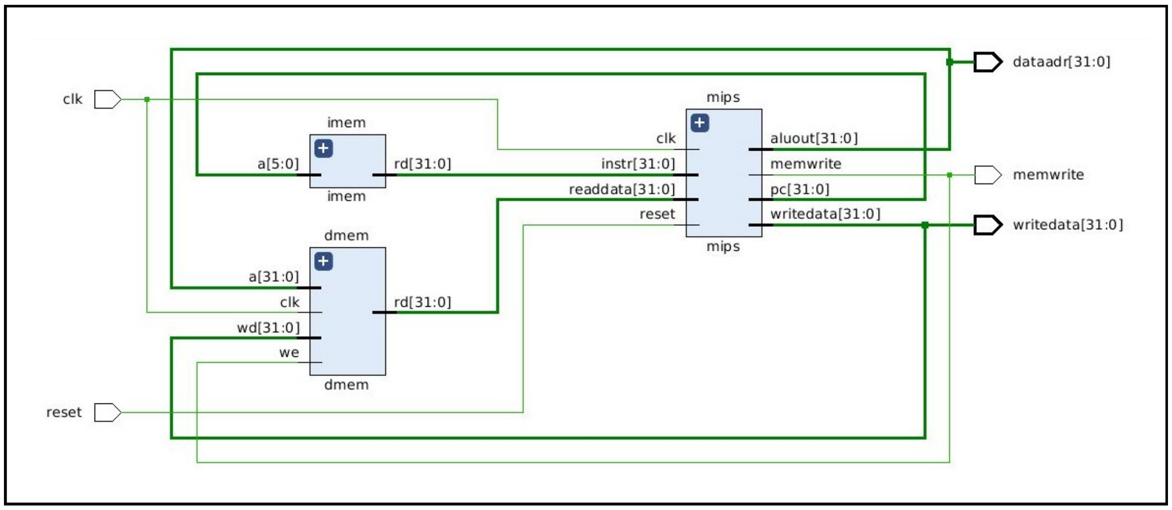



```
Design Sources (2)
  top (top.v) (3)
    mips : mips (mips.v) (3)
    imem : imem (imem.v)
    dmem : dmem (dmem.v)
  Memory File (1)
    memfile.mem
```

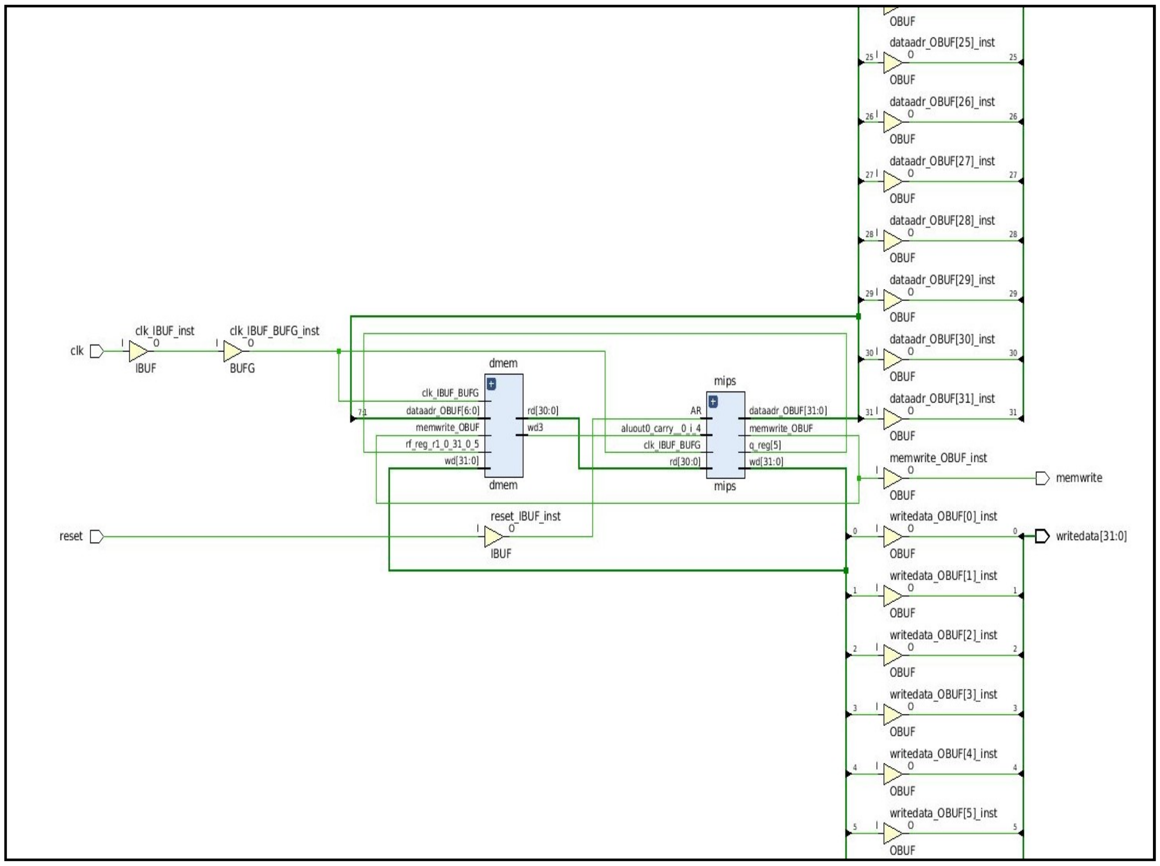
```
memfile.mem *
/afs/ee.cooper.edu/user/a/al

20020005
2003000c
2067fff7
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
ac020054
```

RTL Schematic of Single-Cycle CPU

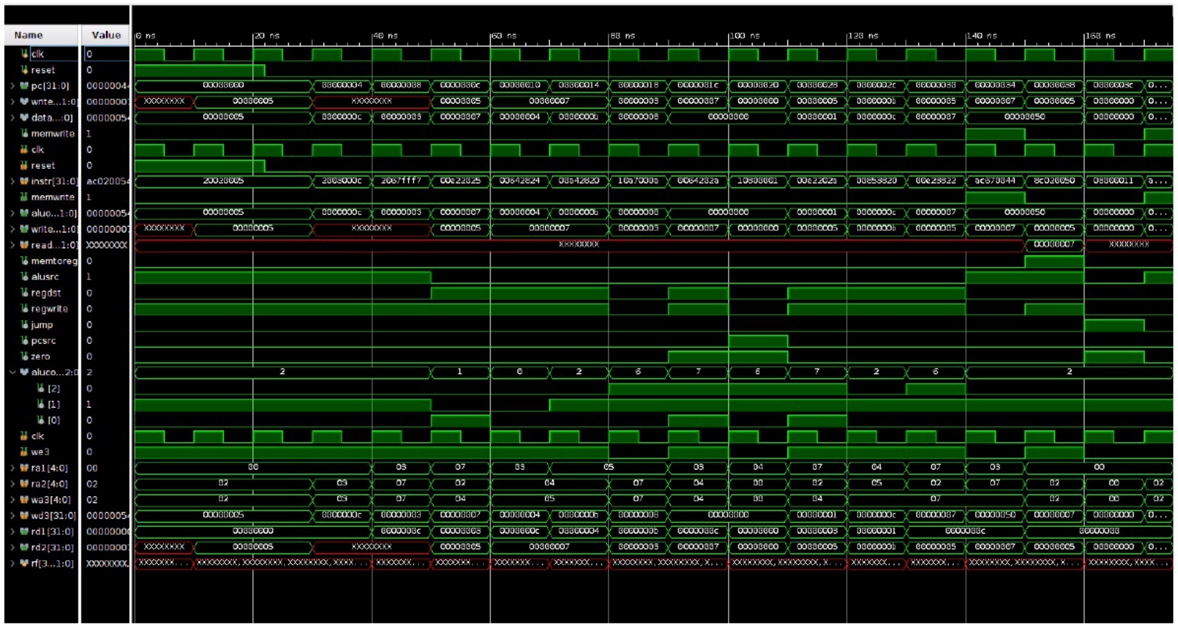


Synthesized/Implemented Schematic of Single-Cycle CPU

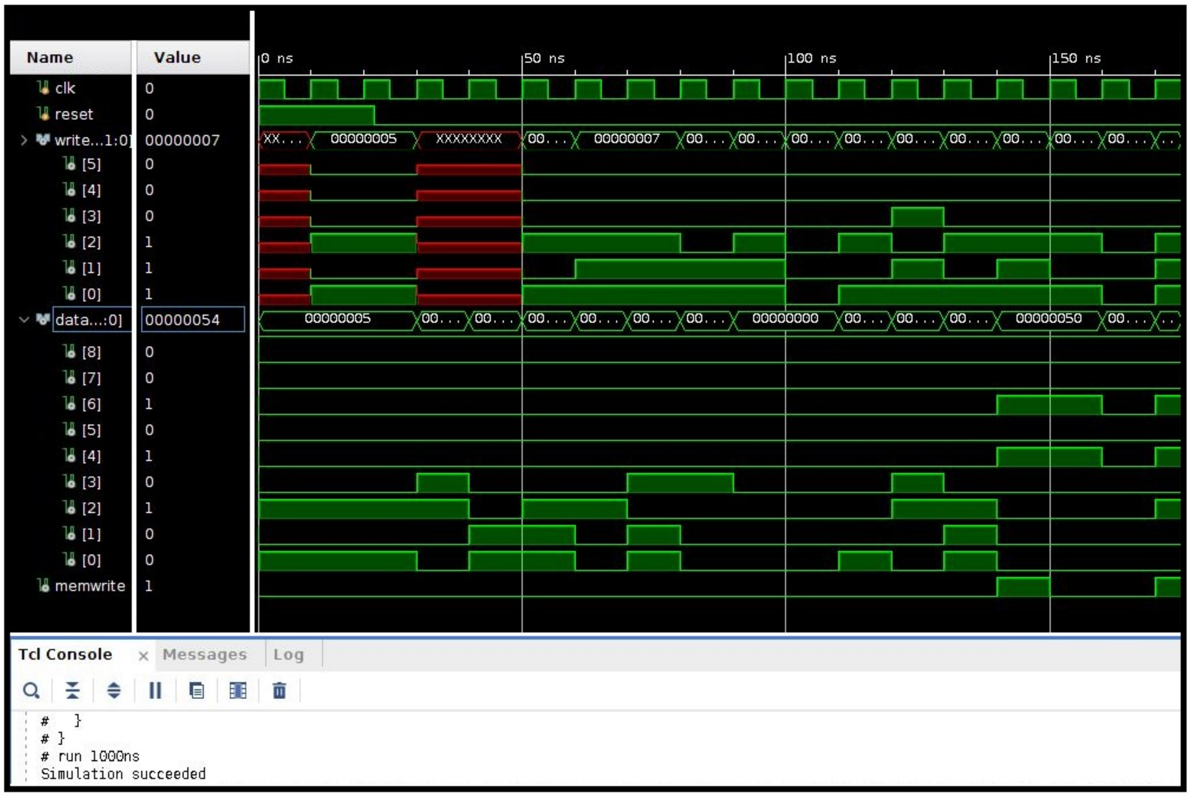


Note: In the constraint file, I set the inputs from the Zedboard to be the push buttons N15 and R18 as the Clock and Reset signals, respectively. The outputs of this I did not assign for this version of the CPU, However I do so later.

Behavioral Simulation of Single-Cycle CPU (all data path variables)



Behavioral Simulation of Single-Cycle CPU (relevant variables only)

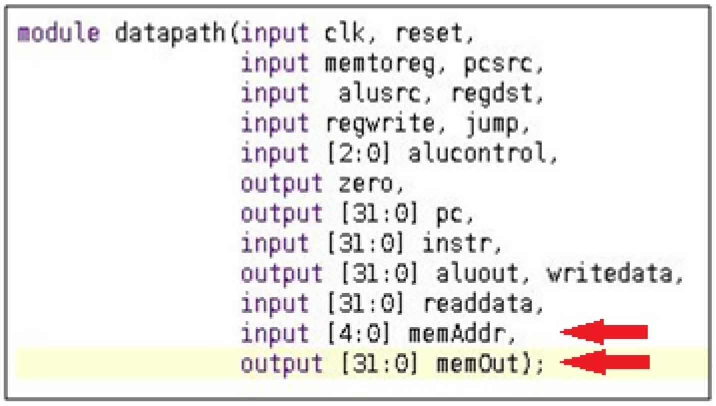


```
# }
# }
# run 1000ns
Simulation succeeded
```

Note: The Console Prints “Simulation Succeeded” indicating the processor and testbench ran successfully. Although “writedata” is correct (write with value 7), for some reason “dataaddr” holds value 54. It should be 84. Not sure why this is.

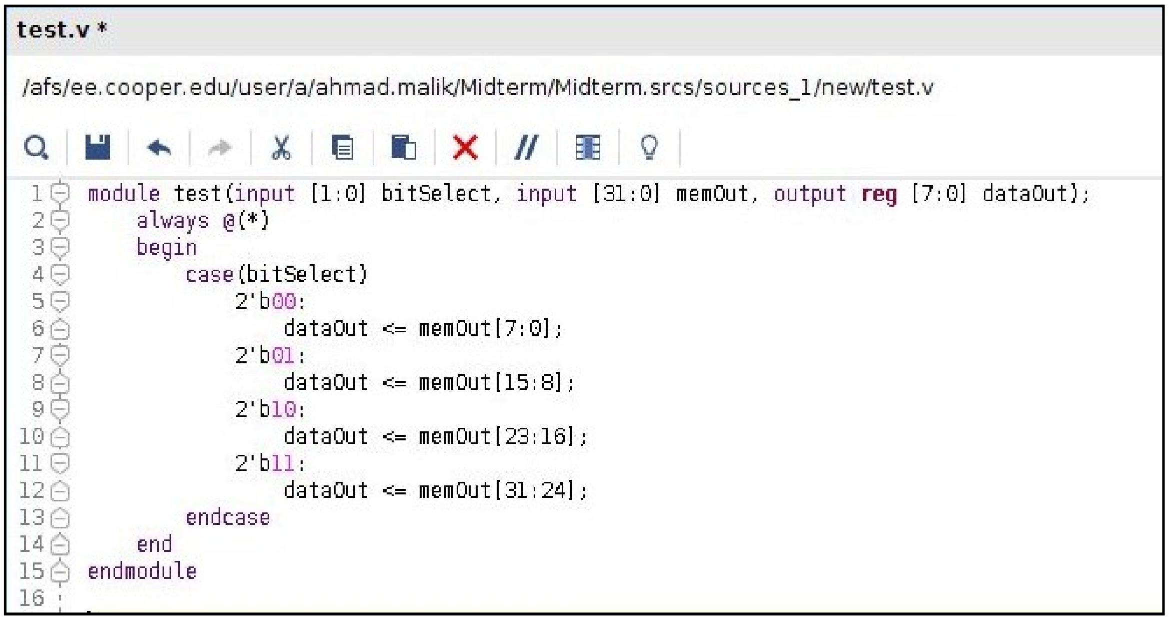
Part 3: Vivado Implementation with Outputs

I modified the data path to support an extra two arguments that serve as a way to output the contents of a 32bit register given a 5bit address of a register. That way, we can see when the register has value 7 in register $2. Since Zedboard has 8 LEDS, we need to split 32bits into 4 sections so we can view the value in the register before it is written to memory.



```
module datapath(input clk, reset,
                input memtoreg, pcsrc,
                input  alusrc, regdst,
                input regwrite, jump,
                input [2:0] alucontrol,
                output zero,
                output [31:0] pc,
                input [31:0] instr,
                output [31:0] aluout, writedata,
                input [31:0] readdata,
                input [4:0] memAddr,
                output [31:0] memOut);
```

This file under the mips.v file hierarchy serves to output the data of any register that is specified in the inputs of the Zedboard. It takes in a 2bit input which determines which section of the 32 bit wide Register value to read and outputs each portion as 8-bits on the Zedboard (for the 8 LEDS)

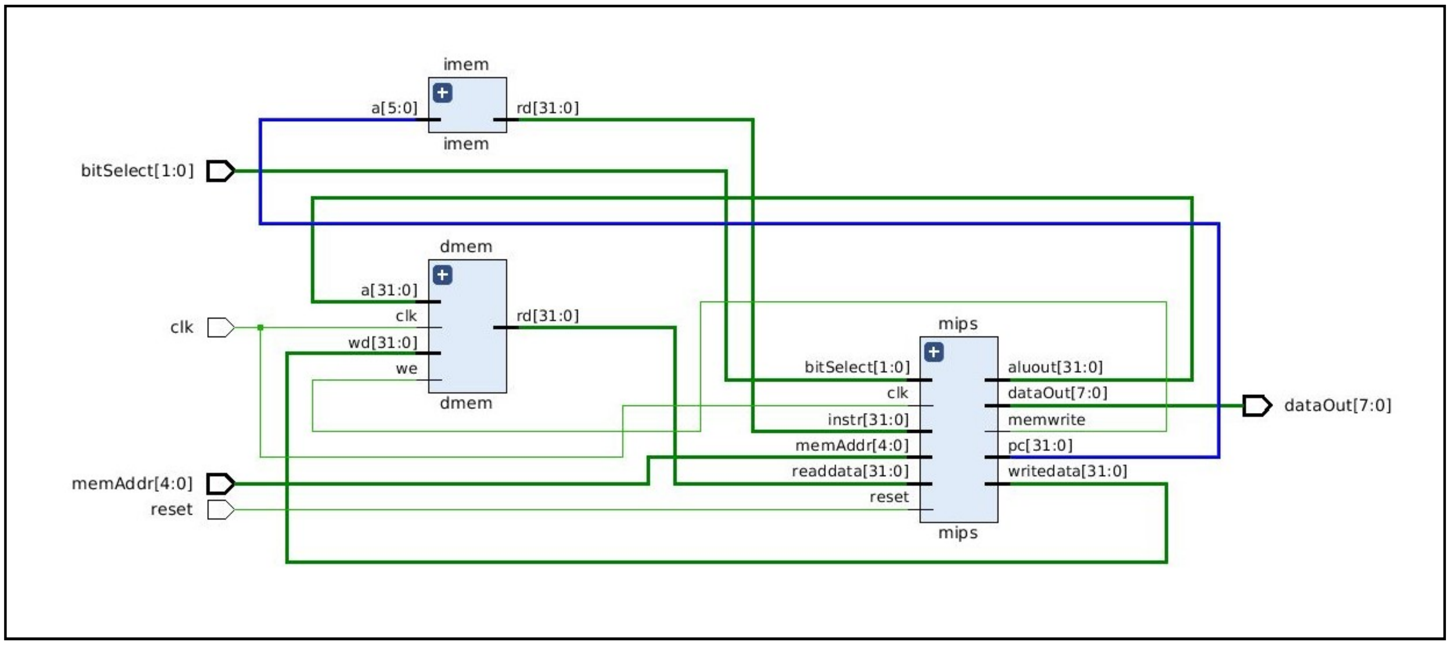


test.v *

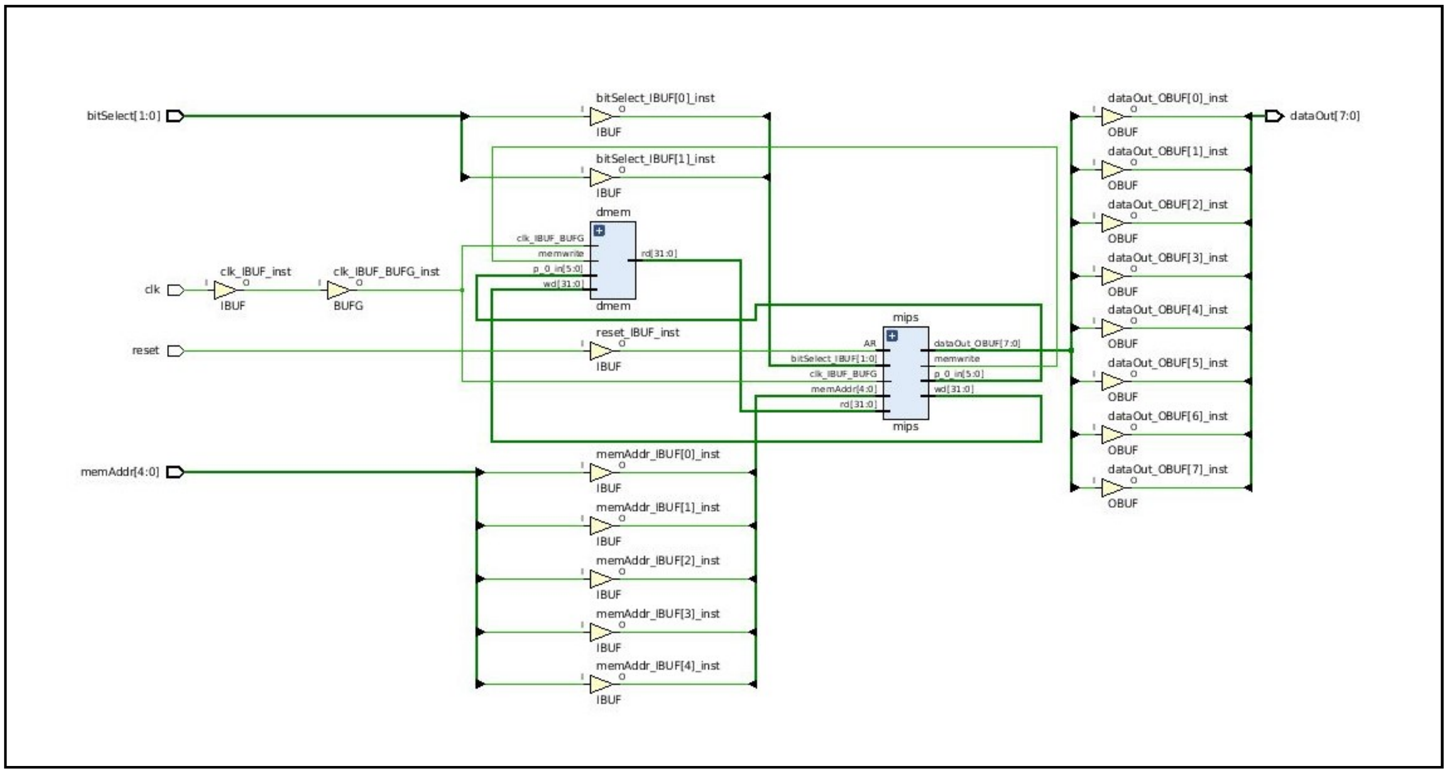
/afs/ee.cooper.edu/user/a/ahmad.malik/Midterm/Midterm.srcs/sources_1/new/test.v

```
module test(input [1:0] bitSelect, input [31:0] memOut, output reg [7:0] dataOut);
    always @(*)
    begin
        case(bitSelect)
            2'b00:
                dataOut <= memOut[7:0];
            2'b01:
                dataOut <= memOut[15:8];
            2'b10:
                dataOut <= memOut[23:16];
            2'b11:
                dataOut <= memOut[31:24];
        endcase
    end
endmodule
```

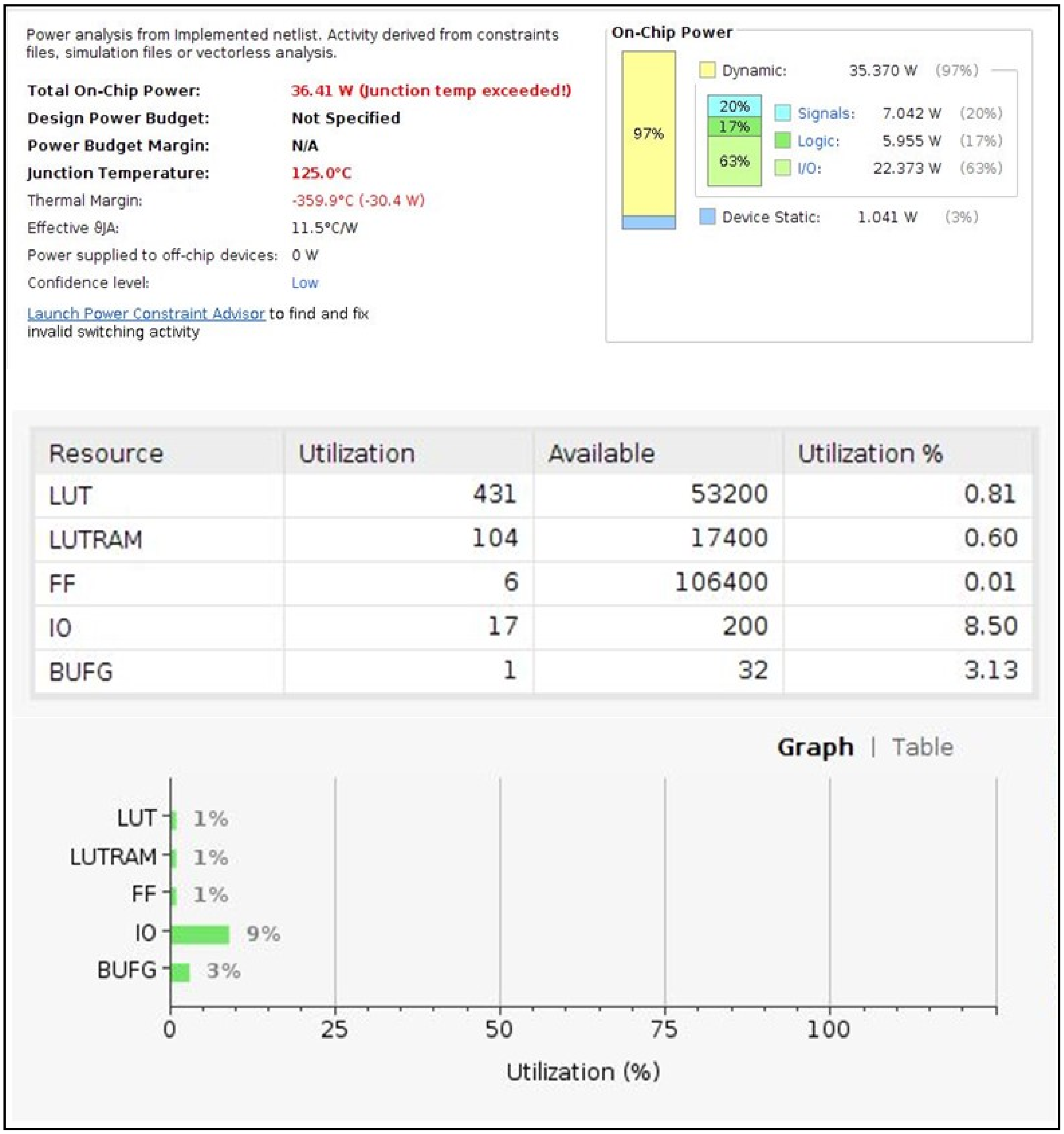
RTL Schematic of Single-Cycle CPU with I/O



Synthesized/Implemented Schematic of Single-Cycle CPU with I/O



FPGA Utilization



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**36.41 W (Junction temp exceeded!)**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**125.0°C**
Thermal Margin:	-359.9°C (-30.4 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

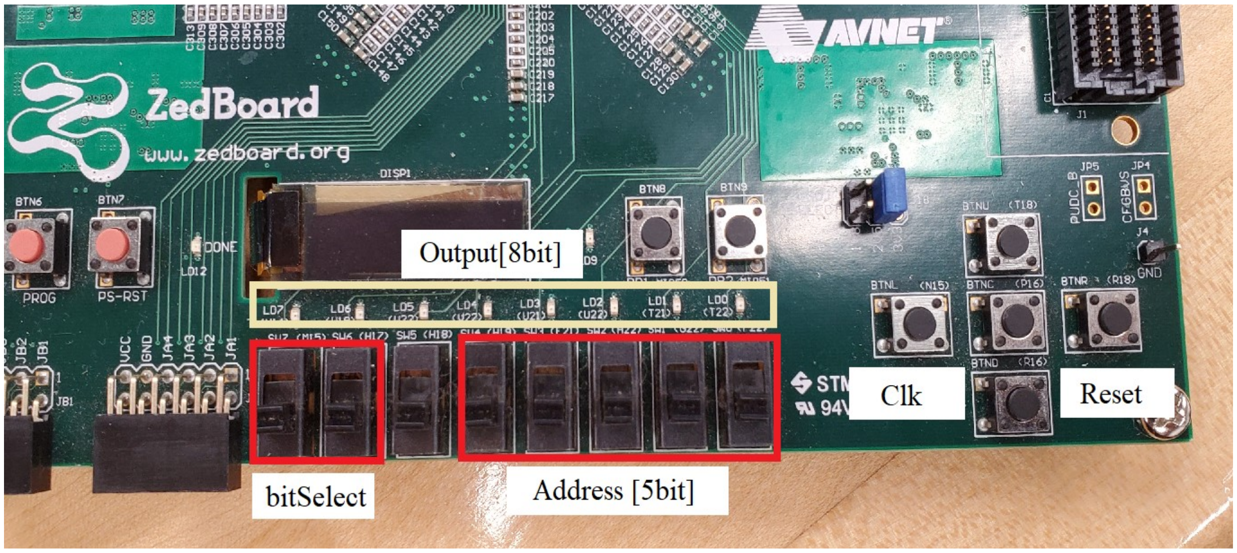
On-Chip Power

Dynamic:		35.370 W	(97%)
	Signals:	7.042 W	(20%)
	Logic:	5.955 W	(17%)
	I/O:	22.373 W	(63%)
Device Static:		1.041 W	(3%)

Resource	Utilization	Available	Utilization %
LUT	431	53200	0.81
LUTRAM	104	17400	0.60
FF	6	106400	0.01
IO	17	200	8.50
BUFG	1	32	3.13

It looks like this implementation puts a lot of strain on the Inputs and Outputs of the Zedboard. It leads to the most utilization and highest power consumption, resulting in the junction temperature to be succeeded. Despite this, one can generate the bit stream and program the Zedboard without hassle.

ZedBoard Inputs and Ouputs



Summary and Discussion

Experiment	Simulation	Theoretical
Using Verilog and the Vivado Design Suite, I successfully created a single-cycle MIPS processor. I was able to program the FPGA board to function as a MIPS processor, and was able to run the set of instructions from the textbook while seeing the outputs being written and loaded onto the registers by manually manipulating the CPU's clock using push buttons.	The simulation worked really well for the Single-Cycle CPU that used only the clock and reset as inputs, and the "writeaddr" ,"dataaddr", "memwrite" as outputs. Because several of our modules, like the ALU and regfile, were not written properly, it took us a few tries before we finally succeeded. I was able to quickly debug our code by looking at the states in the simulation. In the end the console displayed "simulation succeeded" indicating that the whole memory file holding instructions was read and executed correctly.	This project taught me about the MIPS processor's design as well as how to create one using Verilog and an FPGA board. We gained knowledge of the fundamental elements that any MIPS processor ought to contain. Given my current knowledge and experience in designing a MIPS processor, I hope to impove upon the design by implementing a multi-cycle processor so that instructions will execute faster.

Overall Conclusions

I used everything I learned in the labs and frequently used the course material while working on this project to research about and construct a single-cycle MIPS processor. I came to know of the MIPS architecture's finer details, like how data is represented, how instructions are allocateed and processed bitwise, Although the textbook gave the majority of the verilog code for the modules for the Single-Cycle MIPS processor, I added code to the mips module and altered the datapath so that we could read register values as we flipped switches and pushbuttons on the FPGA board to verify functionality. Overall the project was a success.

References

[1] Harris, D., & Harris, S. (2012). Digital Design and Computer Architecture (2nd ed.). Morgan Kaufmann.

[2] COE1502 - MIPS R2000 Architecture Instruction Formats. (n.d.). Pitt.edu. Retrieved November 7, 2022, from https://people.cs.pitt.edu/~don/coe1502/Reference/InstructionFormat.html

[3] MIPS Instruction Formats. (n.d.). Kzoo.edu. Retrieved November 7, 2022, from http://www.cs.kzoo.edu/cs230/Resources/MIPS/MachineXL/InstructionFormats.html

[4] Difference between MIPS and ARM. (2015, January 10). Compare the Difference Between Similar Terms. https://www.differencebetween.com/difference-between-mips-and-vs-arm/

[5] X86 instructions and ARM architecture: ARM architecture. (n.d.). Saylor Academy. Retrieved November 7, 2022, from https://learn.saylor.org/mod/book/view.php?id=27055&chapterid=3249