John Arena

Professor Gertner

CSC 342/343

Lab 4

Due 4/17/19

Spring 2019

NOT COMPLETED YET

Table of Contents

Section 1) Objective

For this **Final Lab**, we will be designing a single cycle CPU based on 32 MIPS instruction set. This will be be essentially combing all the previous labs done so far. R and I instruction types will be supported.

- Input machine instructions you want to execute into SRAM memory on the board using switches. Have to store the address of first instruction in a register.
- Input data to SRAM using switches.
- Load the first instruction from SRAM into Instruction Register
- Start execution
- Display the final result using Seven Segment Displays

Section 2) Description and Specifications

Single Cycle CPU

We will be designing a single cycle cpu, which is a CPU that does everything on one clock cycle. That is load things into registers, do computations, increment PC, etc. Below in Figure 1 is a diagram

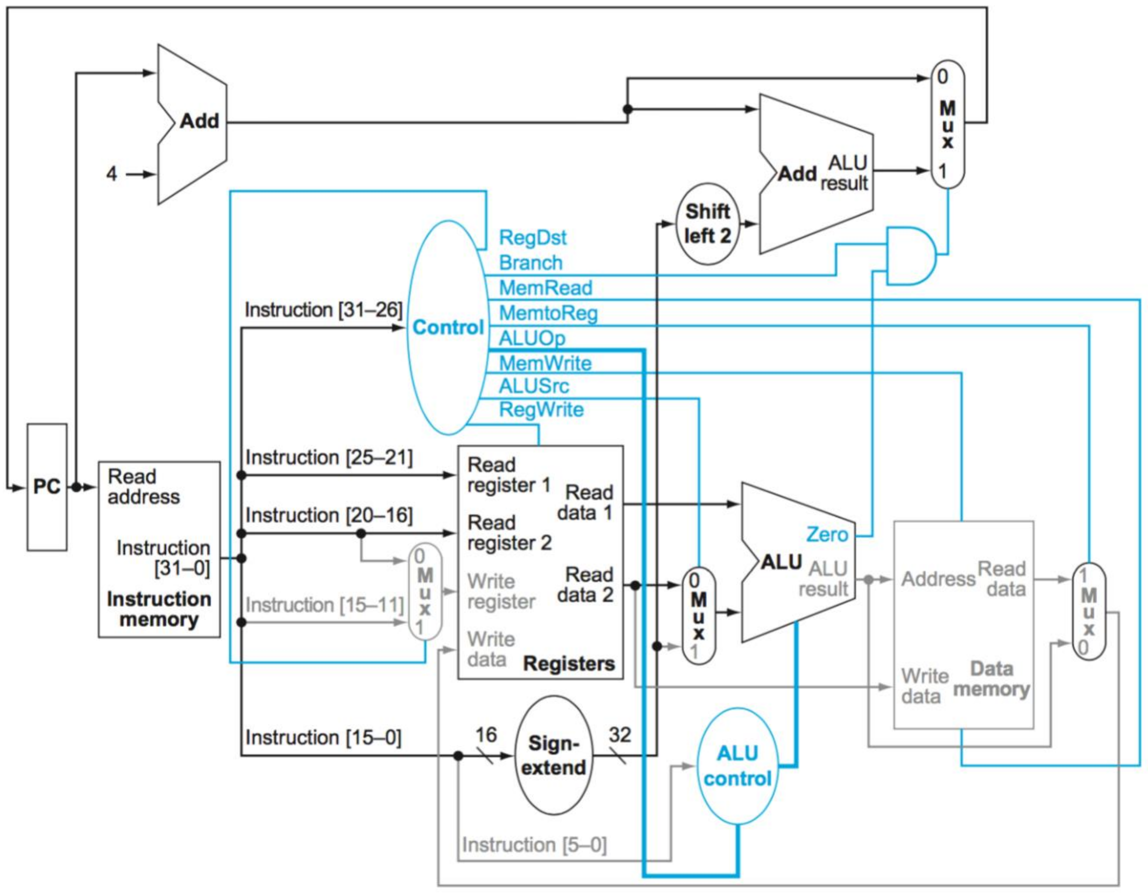


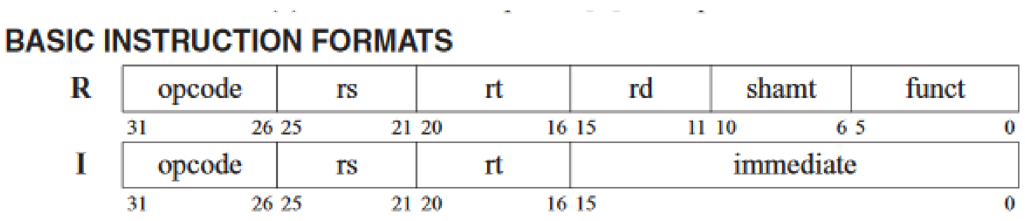
Figure 1: Single Cycle CPU

As can be seen, this has a lot of components from previous labs. For example, a 32 bit adder for incrementing the PC Counter, SRAM from Lab 2 for Instruction Memory, sign extender from

Lab3, and so on! So what are the new components? They are: **Register Memory, Control Unit, ALU Control, ALU, and Data Memory.**

<u>Register Memory</u>

Starting with Register Memory, this is essentially the MIPS Registers. 32 registers in total. It has 3 register inputs, two sources and a destination. We know from MIPS green sheet

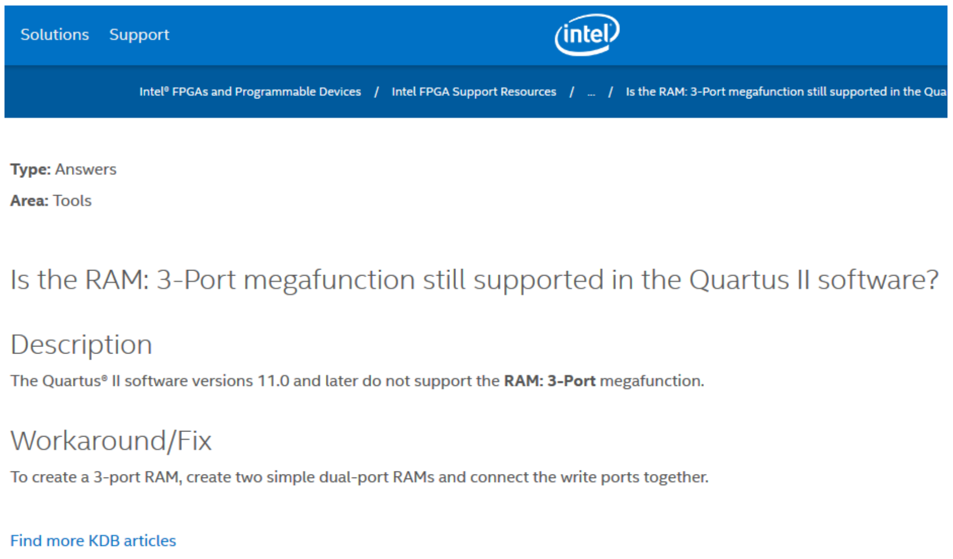


BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15 0		

Figure 2: Mips format instructions

that this is for R format. For I format, we will have a mux that let's us just use two of the inputs. Then, there is a data input for **load word**, a clock pin and a write enable pin. Outputs are qa and qb of read address a and b, respectively. Below is a picture of the circuit. It is a combination of two 2-port rams, with them sharing the same write registers, which is fine according to intel.



Solutions
Support
intel
Intel® FPGAs and Programmable Devices / Intel FPGA Support Resources / ... / Is the RAM: 3-Port megafunction still supported in the Qua

Type: Answers

Area: Tools

Is the RAM: 3-Port megafunction still supported in the Quartus II software?

Description

The Quartus® II software versions 11.0 and later do not support the **RAM: 3-Port** megafunction.

Workaround/Fix

To create a 3-port RAM, create two simple dual-port RAMs and connect the write ports together.

Find more KDB articles

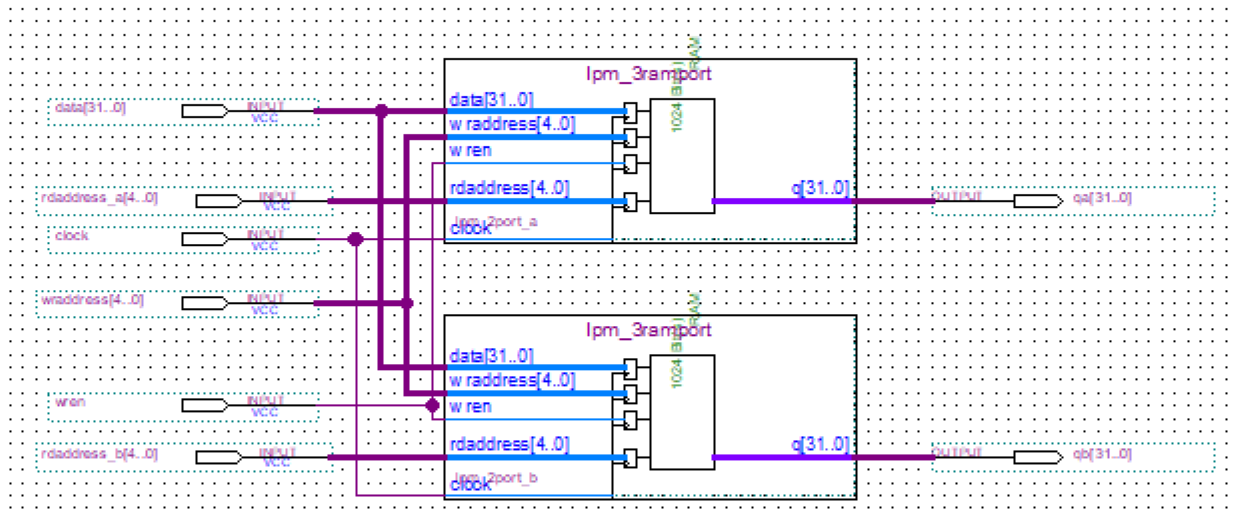


Figure 2a: 3ramport built using 2ramports.

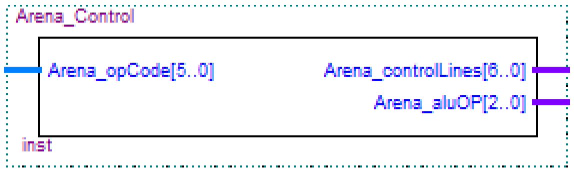
Control Unit

Next is the control unit. This takes in the upper [31..26] bits of the instruction, which we know from MIPS instructions that those bits is the opcode. The control unit will then basically output enable lines for other things on the circuit, which we will refer to as control lines, since they control things on the circuit. Below is the VHDL code with the appropriate comments.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 5 - FINAL LAB - Spring 2019
Due: 5/15/19
-- Arena_Control.vhd

library ieee;
use ieee.std_logic_1164.all;


entity Arena_Control is
      port(
            Arena_opCode: in std_logic_vector(5 downto 0); -- 6 bits from top
6 bits of Instruction, denoted OPCODE
            Arena_controlLines: out std_logic_vector(6 downto 0); -- 7
control lines
            -- CL6: RegWrite, CL5: ALUSrc, CL4: MemWrite, CL3: MemtoReg, CL2:
MemRead, CL1: Branch, CL0: RegDst  --
            Arena_aluOP: out std_logic_vector(2 downto 0) -- opcode to be
sent to ALU for correct operation
```

```
            );
end Arena_Control;

architecture Arena_Control_arch of Arena_Control is
begin
      process (Arena_opCode)
      begin
            case Arena_opCode is
                  when "000000" => -- R Type Instruction, ex add, sub
                  Arena_controlLines <= "1000001"; -- RegDst, RegWrite
                  Arena_aluOP <= "010";
                  when "100011" => -- I Type instruction, load word
                  Arena_controlLines <= "1101100"; -- RegWrite, ALUSrc,
MemToReg, MemRead
                  Arena_aluOP <= "000";
                  when "101011" => -- I type instruction, save word
                  Arena_controlLines <= "0110000"; -- ALUSrc, MemWrite
                  Arena_aluOP <= "000";
                  when "000100" => -- I type instruction, branch on equal
                  Arena_controlLines <= "0000010";
                  Arena_aluOP <= "001";
                  when "001101" => -- I type instruction ,ORI (used for
things such as load immediate)
                  Arena_controlLines <= "1100000"; -- RegWrite, ALUSrc
                  Arena_aluOP <= "111"; --"111"
                  when others =>
                  null;
            end case;
      end process;
end Arena_Control_arch;
```

Figure 3: Control Unit VHDL



As can be seen, based on a certain opcode, we get a certain output for control lines and aluOP.

What is ALUOp? This is an opcode that is sent to the next circuit, the ALU Control unit. It will serve as a purpose to help figure out which operation to perform for whichever instruction.

<u>ALU Control</u>

The ALU Control tells the ALU what operation to perform. First, what is ALU? It means Arithmetic Logical Unit, hence the name since it performs operations. The ALU Control's truth table was provided by the textbook, but I extended it to below.

ALUOP2	ALUOP1	ALUOP0	F5	F4	F3	F2	F1	F0	Operation
0	0	0	X	X	X	X	X	X	0010
0	0	1	X	X	X	X	X	X	0110
0	1	0	X	X	0	0	0	0	0010
0	1	X	X	X	0	0	1	0	0110
0	1	0	X	X	0	1	0	0	0000
0	1	0	X	X	0	1	0	1	0001
0	1	X	X	X	1	0	1	0	0111
1	1	1	0	0	1	1	0	1	0001

Figure 4: ALU Control Truth Table

Based on those inputs, we will get a certain output to be fed to the ALU. Below is the VHDL Code.

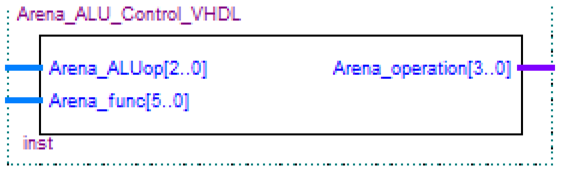
```
-- (First, Last) John Arena - CSC 342/343 - Lab 5 - FINAL LAB - Spring 2019
Due: 5/15/19
-- Arena_ALU_Control_VHDL.vhd

library ieee;
use ieee.std_logic_1164.all;


entity Arena_ALU_Control_VHDL is
```

```vhdl
	port(
		Arena_ALUop: in std_logic_vector(2 downto 0); -- op code
		Arena_func: in std_logic_vector(5 downto 0); -- func code
		Arena_operation: out std_logic_vector(3 downto 0) -- operation
out
		);
end Arena_ALU_Control_VHDL;

architecture Arena_ALU_Control_VHDL_arch of Arena_ALU_Control_VHDL is
begin
	process (Arena_func, Arena_ALUop)
	begin
		case Arena_func is
			when "000000" => -- Add
				if(Arena_ALUop = "010") then
					Arena_operation <= "0010";
				else
					null;
				end if;
			when "010000" => -- Add
				if(Arena_ALUop = "010") then
					Arena_operation <= "0010";
				else
					null;
				end if;
			when "100000" => -- Add
				if(Arena_ALUop = "010") then
					Arena_operation <= "0010";
				else
					null;
				end if;
			when "110000" => -- Add
				if(Arena_ALUop = "010") then
					Arena_operation <= "0010";
				else
					null;
				end if;


			when "000010" => -- Sub
				if(Arena_ALUop = "010" or Arena_ALUop = "011") then
					Arena_operation <= "0110";
				else
					null;
				end if;
			when "010010" => -- Sub
				if(Arena_ALUop = "010" or Arena_ALUop = "011") then
					Arena_operation <= "0110";
				else
					null;
				end if;
			when "100010" => -- Sub
				if(Arena_ALUop = "010" or Arena_ALUop = "011") then
					Arena_operation <= "0110";
				else
					null;
				end if;
```

```
when "110010" => -- Sub
      if(Arena_ALUop = "010" or Arena_ALUop = "011") then
            Arena_operation <= "0110";
      else
            null;
      end if;


when "000100" => -- AND
      if(Arena_ALUop = "010") then
            Arena_operation <= "0000";
      else
            null;
      end if;
when "010100" => -- AND
      if(Arena_ALUop = "010") then
            Arena_operation <= "0000";
      else
            null;
      end if;
when "100100" => -- AND
      if(Arena_ALUop = "010") then
            Arena_operation <= "0000";
      else
            null;
      end if;
when "110100" => -- AND
      if(Arena_ALUop = "010") then
            Arena_operation <= "0000";
      else
            null;
      end if;

when "000101" => -- OR
      if(Arena_ALUop = "010") then
            Arena_operation <= "0001";
      else
            null;
      end if;
when "010101" => -- OR
      if(Arena_ALUop = "10") then
            Arena_operation <= "0001";
      else
            null;
      end if;
when "100101" => -- OR
      if(Arena_ALUop = "010") then
            Arena_operation <= "0001";
      else
            null;
      end if;
when "110101" => -- OR
      if(Arena_ALUop = "010") then
            Arena_operation <= "0001";
      else
            null;
      end if;
```

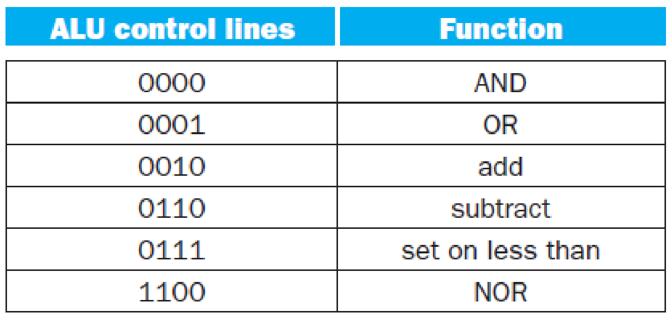
```
                when "001010" => -- Set Less Than
                        if(Arena_ALUop = "010" or Arena_ALUop = "011") then
                                Arena_operation <= "0111";
                        else
                                null;
                        end if;

                when "011010" => -- Set Less Than
                        if(Arena_ALUop = "010" or Arena_ALUop = "011") then
                                Arena_operation <= "0111";
                        else
                                null;
                        end if;

                when "101010" => -- Set Less Than
                        if(Arena_ALUop = "010" or Arena_ALUop = "011") then
                                Arena_operation <= "0111";
                        else
                                null;
                        end if;
                when "111010" => -- Set Less Than
                        if(Arena_ALUop = "010" or Arena_ALUop = "011") then
                                Arena_operation <= "0111";
                        else
                                null;
                        end if;
                when others =>
                null;
        end case;
        case Arena_ALUop is
                when "000" => -- Add for SW
                        Arena_operation <= "0010";
                when "001" => -- Sub for LW
                        Arena_operation <= "0110";
                when "111" => -- OR for ORI
                        Arena_operation <= "0001";
                when others =>
                        null;
        end case;
    end process;
end Arena_ALU_Control_VHDL_arch;
```

Figure 5: ALU Control VHDL



ALU

This is the Arithmetic Logical Unit. This performs the operations between the register data coming from the Register Memory outputs QA and QB, respectively. It uses the output from the last circuit, ALU Control, to determine which operation to take. Below is a table from the book Im basing my operations on.



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

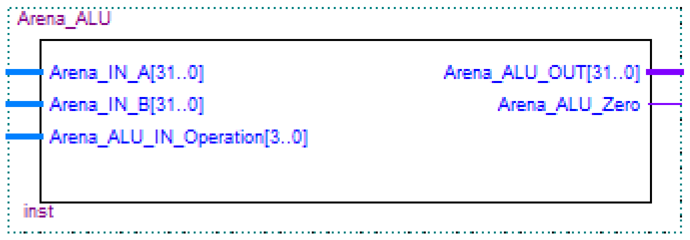
Figure 6: Operations corresponding to Alu Control Output

This circuit will take two inputs and perform the corresponding operation above according to the table. Below is the VHDL code.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 5 - FINAL LAB - Spring 2019
Due: 5/15/19
-- Arena_ALU.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity Arena_ALU is
      port(
            Arena_IN_A: in std_logic_vector(31 downto 0); -- 32 bit input A
            Arena_IN_B: in std_logic_vector(31 downto 0); -- 32 bit input B
            Arena_ALU_IN_Operation: in std_logic_vector(3 downto 0); --
Operation input
            Arena_ALU_OUT: out std_logic_vector(31 downto 0); -- 32 bit
output of ALU
            Arena_ALU_Zero: out std_logic -- zero flag branch, set less than,
etc.
            );
end Arena_ALU;
```

```vhdl
architecture Arena_ALU_arch of Arena_ALU is
signal Arena_sign_conv_A : signed(31 downto 0);
signal Arena_sign_conv_B : signed(31 downto 0);
signal Arena_arithmetic_conv : signed (31 downto 0);
begin
      process(Arena_ALU_IN_Operation)
      begin
            case Arena_ALU_IN_Operation is
                  when "0000" => -- Operation bitwise AND
                  Arena_ALU_OUT <= Arena_IN_A and ARENA_IN_B;
                  when "0001" => -- Operation bitwise OR
                  Arena_ALU_OUT <= Arena_IN_A or ARENA_IN_B;
                  when "0010" => -- Operation ADDITION
                        Arena_sign_conv_A <= signed(Arena_IN_A);
                        Arena_sign_conv_B <= signed(Arena_IN_B);
                        Arena_arithmetic_conv <= Arena_sign_conv_A +
Arena_sign_conv_B;
                        Arena_ALU_OUT <=
std_logic_vector(Arena_arithmetic_conv);
                  when "0110" => -- Operation SUBTRACT
                        Arena_sign_conv_A <= signed(Arena_IN_A);
                        Arena_sign_conv_B <= signed(Arena_IN_B);
                        Arena_arithmetic_conv <= Arena_sign_conv_A -
Arena_sign_conv_B;
                        Arena_ALU_OUT <=
std_logic_vector(Arena_arithmetic_conv);
                        if(Arena_arithmetic_conv =
"00000000000000000000000000000000") then
                              Arena_ALU_Zero <= '1';
                        else
                              Arena_ALU_Zero <= '0';
                        end if;
                  when "0111" => -- Set on less than
                        if(Arena_IN_A < Arena_IN_B) then
                              Arena_ALU_Zero <= '1';
                        else
                              Arena_ALU_Zero <= '0';
                        end if;
                  when others =>
                  null;
            end case;
      end process;
end Arena_ALU_arch;
```

Figure 7: ALU VHDL Code



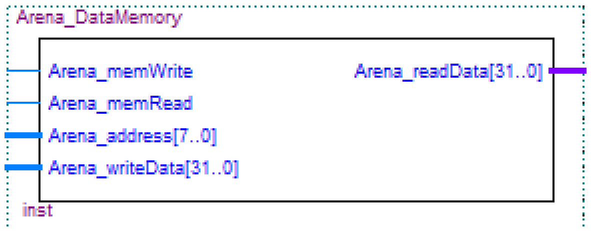
Data Memory

Finally there is the Data Memory. This is actually not as hard as it sounds in VHDL. I essentially made an array of 256 locations, each with 32 bit logic vectors. The reason I only made it 256 locations and not any bigger was I have a compiling time issue that makes it almost 20 minutes to compile if any larger! So I made it 256 for the sake of compiling, but it can be extended if I can resolve the issue. It takes in an address for writing to data, and data in for data to write, which usually comes from the ALU (or sometimes save word taking information from the register to save into memory). Below is the VHDL code

```
-- (First, Last) John Arena - CSC 342/343 - Lab 5 - FINAL LAB - Spring 2019
Due: 5/15/19
-- Arena_DataMemory.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity Arena_DataMemory is
	port(
		Arena_memWrite, Arena_memRead: in std_logic; -- Control Line
inputs
		Arena_address: in std_logic_vector(7 downto 0); -- 10 bit address
		Arena_writeData: in std_logic_vector(31 downto 0); -- 32 bit
address, 32 bit input data
		Arena_readData: out std_logic_vector(31 downto 0) -- Read data
(out) 32 bits
	);
end Arena_DataMemory;

architecture Arena_DataMemory_arch of Arena_DataMemory is
type dataMem_loc_array is array(0 to 255) of std_logic_vector(31 downto 0); -
- 256 locations of 32 bit data
signal dataMem_loc : dataMem_loc_array;
begin
process(Arena_memWrite, Arena_memRead)
begin
	if(Arena_memWrite = '1') then -- If memWrite enabled
		dataMem_loc(to_integer(unsigned(Arena_address))) <=
Arena_writeData; -- Write data
	else
		null;
	end if;
	if(Arena_memRead = '1') then -- if memRead enabled
```

```
            Arena_readData <=
dataMem_loc(to_integer(unsigned(Arena_address)));
      else
            null;
      end if;
end process;
end Arena_DataMemory_arch;
```

Figure 8: Data Memory VHDL Code



<u>Section 3) Simulations</u>

Below are simulations for components

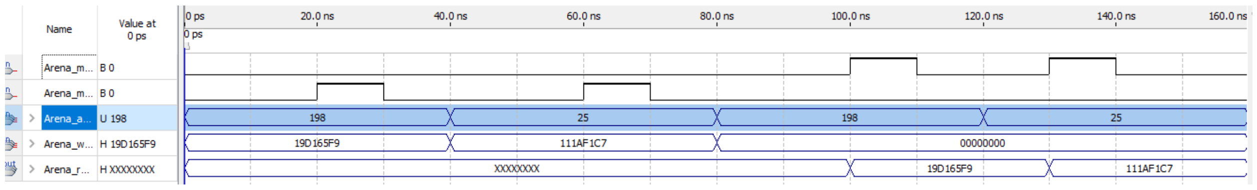


Figure 9: Data Memory Simulation

As can be seen, once the clock goes high, the data on the 4th line is read from the address that it was stored at the beginning of the simulation.

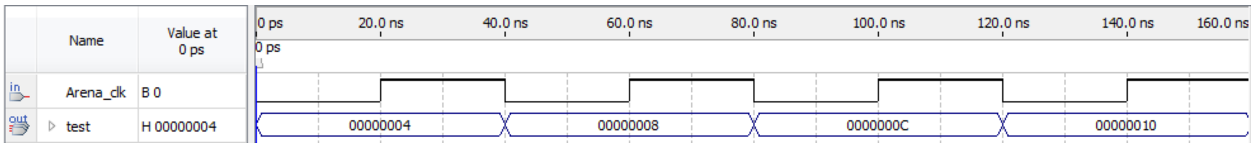


Figure 10: PC Incrementor

Here is the PC Incrementor. It increments the PC by 4. (Remember, each location is 4 bytes long!)

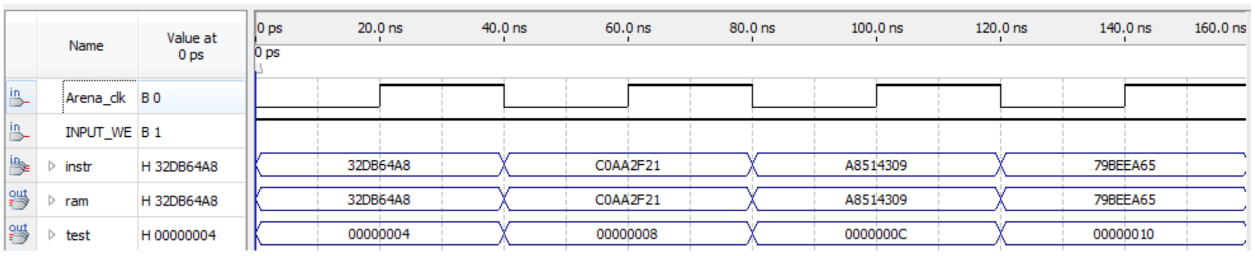


Figure 11: SRAM

Here is the SRAM simulation. Write Enable is high, and instruction is loaded in. RAM shows what is stored, and test refers to the PC Counter. As we can see, instructions are loaded at the appropriate times as the PC Counter increments.

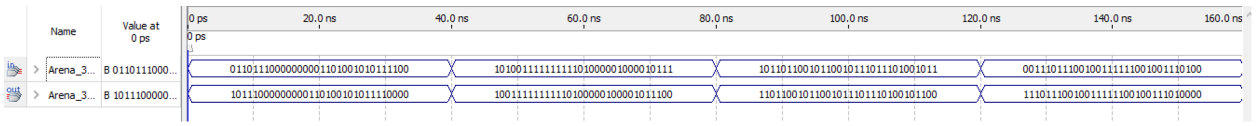


Figure 12: Shift left

This is the shift left unit. It is used for branching.

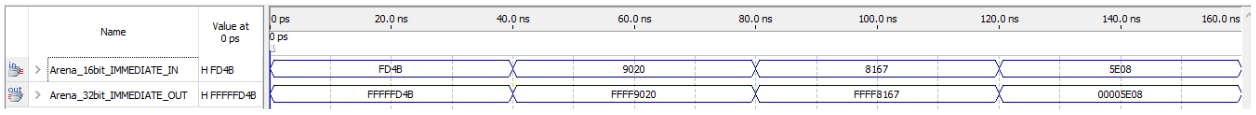


Figure 13: Sign Extension

This is the sign extension unit. It is for extending I instruction immediate values from 16 bits to 32 bits. Notice it also copies the sign, for example, FD48 has a sign bit for a negative value, so its extension is FFFFFD4B.

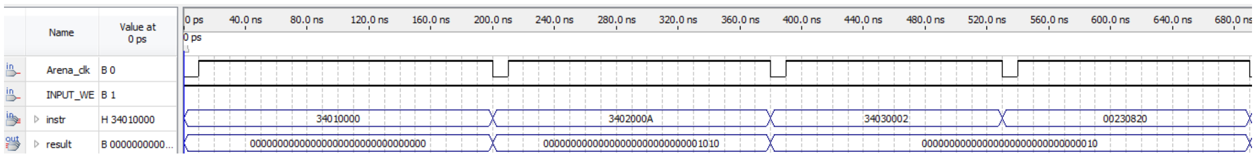


Figure 14: CPU TEST

These are for loading immediates into registers, 0 10 and 2 respectively. We know by going into **MARS Simulator** that LI instruction is really ORI with the result you want to store OR'd with the zero register. The last instruction adds 2 and 0. These are the instructions below. **Note I used registers that are not generally used for the purposes of storing numbers for additions, I just used them for simplicity of entering on the board.**

ORI $at $zero 0x0000

Binary: 00110100000000010000000000000000

Hex: 0x34010000

ORI $v0 $zero 0x000A

Binary: 00110100000000100000000000001010

Hex:0x3402000A

ORI $v1 $zero 0x0002

Binary: 00110100000000110000000000000010

Hex: 0x34030002

ADD $at $at $v1 – adds 0 and 2 and stores two into at.

Binary: 00000000001000110000100000100000

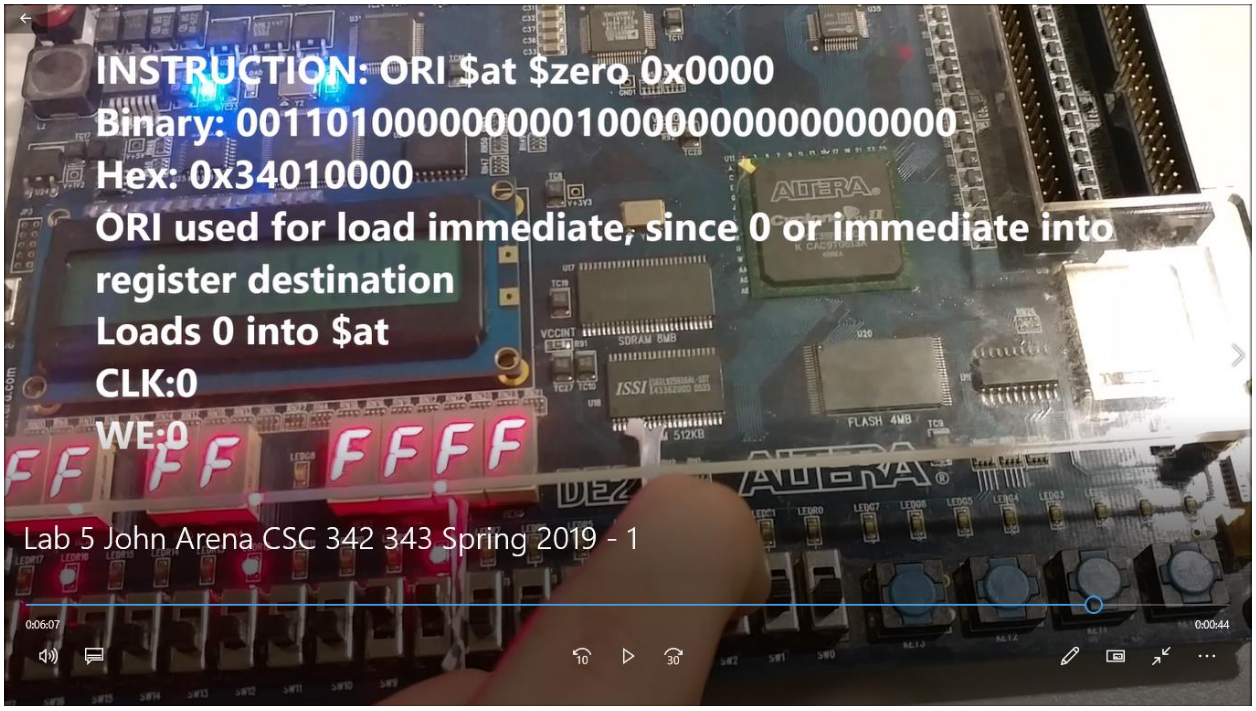
Hex: 0x00230820

These conversions were done by using the tool found at

<https://www.eg.bucknell.edu/~csci320/mips_web

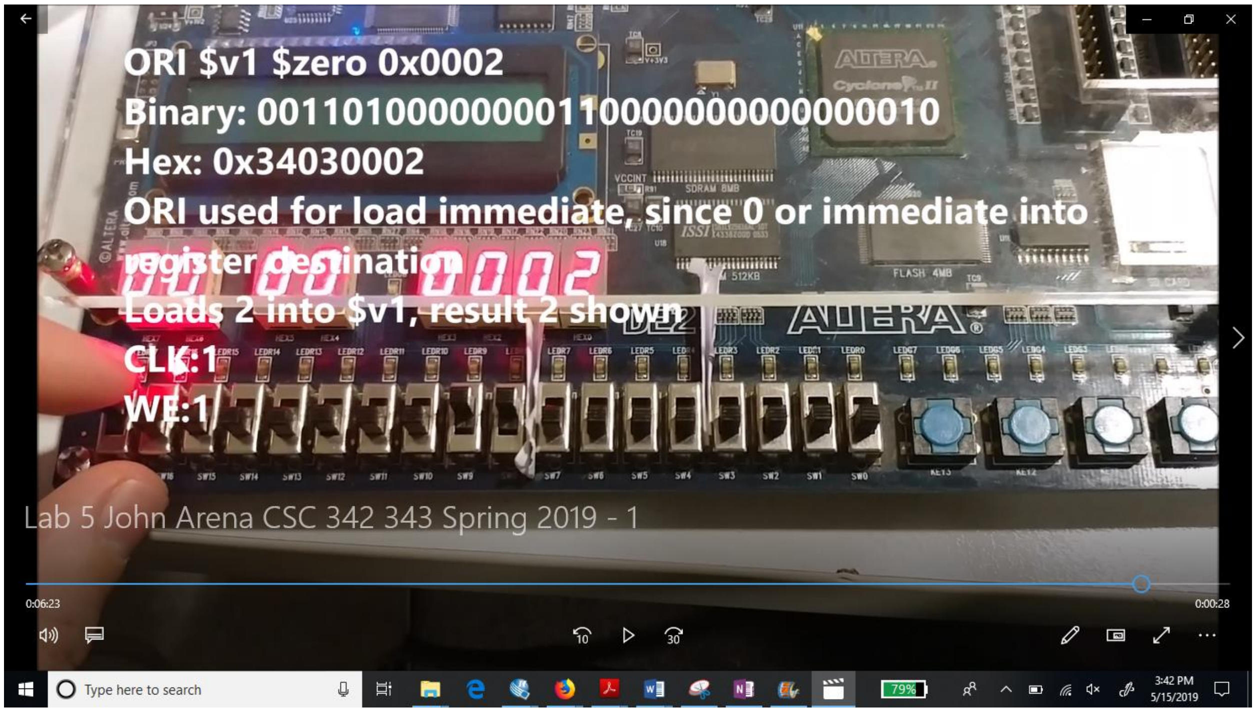
Section 4) Demonstration Pictures

(Not every case is shown in the pictures, all cases shown in video)



INSTRUCTION: ORI $at $zero 0x0000
Binary: 00110100000000010000000000000000
Hex: 0x34010000
ORI used for load immediate, since 0 or immediate into register destination
Loads 0 into $at
CLK:0
WE:0

Lab 5 John Arena CSC 342 343 Spring 2019 - 1



ORI $v1 $zero 0x0002

Binary: 00110100000000110000000000000010

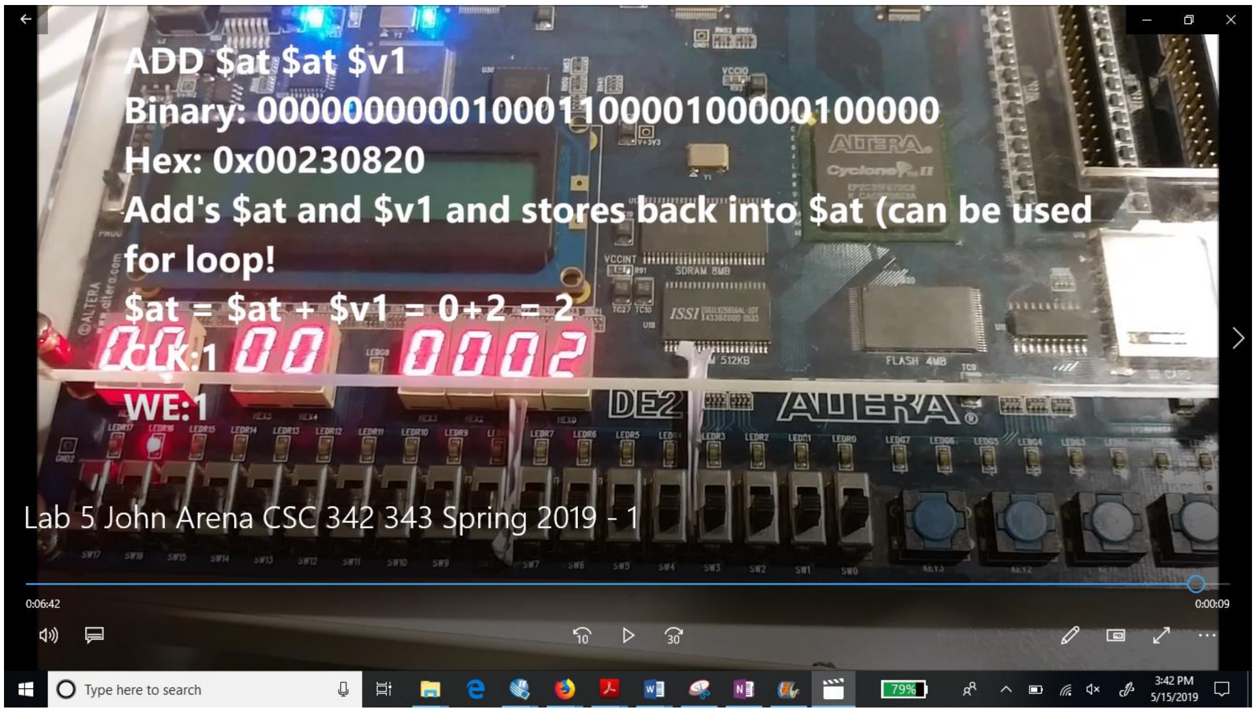
Hex: 0x34030002

ORI used for load immediate, since 0 or immediate into register destination

Loads 2 into $v1, result 2 shown

CLK:1

WE:1



ADD $at $at $v1

Binary: 00000000001000110000100000100000

Hex: 0x00230820

Add's $at and $v1 and stores back into $at (can be used for loop!

$at = $at + $v1 = 0+2 = 2

CLK:1

WE:1

Section 5) Conclusion

In this lab, I designed various control units to control the datapath for the single cycle cpu. It wasn’t too difficult as long as you read the textbook and the appendixs online for the design of things such as the ALU Control. I found out about the issues of getting things in a single cycle as described in class, as some instructions take longer to execute then others. Overall at the end I ran out of time, due to being behind due to a personal emergency I talked to Professor Gertner about through slack, I believe if I had more time I could of tested the rest on the board.