14:332:493:02 – Fall 2017
Written By: Gregory Leonberg

Lab 5: It's all about the Processors

Introduction:

In this lab, we are going to use almost everything we have learned and created thus far in order to make the most recognizable design in all of ECE; a processor. Our design is a general purpose processor with some application specific instructions for video and communications. In this way, it is similar to a simplified Application-Specific Instruction-Set Processor (ASIP).

In order to implement it, a fully detailed Instruction Set Architecture (ISA) has been created that will be the core of the design. By modifying previous components to fit our needs and leveraging the memory IP available in the Vivado tool, an entire computer system will be created with relatively little effort on our part. Once the design has been created, it will be tested by running programs that were written in assembly and passed through a custom assembler as well as running it through a full system simulator.

Prelab – What Kind of Chip you got in there?

Background:

Before we can go about making our processor, we have to understand how it works. To do so, we must examine the ISA that has been created. This will tell us the available instructions, memory organization, register file structure, and instruction formats.

In general, a processor consists of a group of "dumb" components, like memories, general-purpose registers, multiplexers, and an ALU, and a "smart" controller. By using a Finite State Machine as the controller, we can put those components to work to accomplish meaningful tasks.

For the bulk of this lab, we will be creating and modifying these components in order to prepare them for our top level design. Then we will create the most complicated part; the controller.

Task(s):

- Read through the ISA specification attached at the end of the manual.
- Watch this video on the Vivado IP Integrator Tool: https://www.youtube.com/watch?v=ZXiygbhmZoE
- Modify your entity "myALU" from Lab 2 to have a width of 16 bits and the following opcodes given in the table at the top of the next page. Make it synchronous with a clock enable input.

- Modify your entity “pixel_pusher” from Lab 4 to output a 64x64 resolution image from hcount = [0, 63] and vcount = [0, 63] using a 12-bit “addr”. It will read in a 16-bit pixel instead of an 8-bit one with the following distribution: [R,G,B] = [5,6,5]

Opcode	Function
x“0”	A + B
x“1”	A – B
x“2”	A + 1
x“3”	A – 1
x“4”	0 – A
x“5”	A << 1 (shift left logical)
x“6”	A >> 1 (shift right logical)
x“7”	A >>> 1 (shift right arithmetic)
x“8”	A and B
x“9”	A or B
x“A”	A xor B
x“B”	A < B (signed) (as bit 0 of output)
x“C”	A > B (signed) (as bit 0 of output)
x“D”	A = B (as bit 0 of output)
x“E”	A < B (as bit 0 of output)
x“F”	A > B (as bit 0 of output)

Table 1: myALU opcode table, CPU version

Part 1: Wanna be Hackers? Code Crackers? Slackers?

<u>Background:</u>

In order to use our processor, we’re going to need some programs to run on it. In order to get those, we’re going to write them ourselves. However, writing binary instructions is really annoying and time-intensive. Instead, an assembler has been prepared that will convert raw text assembly programs into binary files to use with our design.

For our test programs, we will do simple tasks that exercise the IO interfaces available in the architecture, the UART and the VGA display. By using these interfaces with some simple programs, we can easily prove the basic functionality of the design.

In order to understand the format of the assembly language, please look at the provided “fancy.txt” file that has been provided. Also look at the comments in the assembler source for a description of the allowed formats.

<u>Task(s):</u>

- Write an assembly program that prints the string “Hello_World” over the UART, one byte at a time by reading characters from a string declared in the data segment and sending them until it encounters a null character. After it finishes, it should continuously loop through reading a character from the UART into a register and writing that register value to all memory locations in the video memory. Assemble it and store the text and data COE files for later use. Simulate it to verify code functionality using the provided simulator.

Part 2: Every File Inspected, no Viruses Detected

Background:

In order for our processor to have data to work with, we need memories. For simplicity, our system uses a pure Harvard Architecture with separate instruction and data memories. In this way, our executable code is stored in a single port read-only memory that is read from by the controller. Our data memory is stored in a single port ram memory that allows read/write access. Additionally, we implement our Register File as a dual port memory in order to allow double reads for R-type instructions. Finally, our VGA controller reads data from a dual port ram known as a framebuffer (which can be written and read to by the CPU) in order to output the contents of the framebuffer to the screen.

Some of these memories (the instruction and data memories) will be auto generated, in a similar method to the one used in Lab 4, in order to "program" them in advance with the outputs of our assembler. For the others, we are going to manually create them ourselves using simple VHDL that infers memory blocks.

In order to infer memory, we start by modeling our memory as an array of std_logic_vectors, similar to how a ROM was inferred in Lab 3 for your NetID. By describing the behavior of operations on indexes in that signal via some clever indexing and type conversions (think Lab 3), we can infer memories that the tool will then replace with the appropriate hardware primitives.

Task(s):

- Create an entity called "regs" where you infer a true dual port (both ports can independently either read or write to any location) memory consisting of 32 16-bit words (64 Bytes). It should have the following black box interface and behavior:

```vhdl
entity regs is port (
            clk, en, rst : in std_logic;
            id1, id2 : in std_logic_vector(4 downto 0); -- Addresses
            wr_en1, wr_en2 : in std_logic;
            din1, din2 : in std_logic_vector(15 downto 0);
            dout1, dout2 : out std_logic_vector(15 downto 0)
);
end regs;
```

```
dout1 = registers(id1)
dout2 = registers(id2)
if reset is 1:
        registers(all) = 0
else on every clock edge when enable is 1:
        registers(0) = 0
        if wr_en1:
                registers(id1) = din1
        if wr_en2:
                registers(id2) = din2
```

- Create an entity called "framebuffer" where you infer a dual port memory consisting of 4096 16-bit words (8 KiB). It should have the following black box interface and behavior. *[Notice how we have two different enables going into this device. The CPU side and the video display side are on two different clock domains, operating independently of each other. Because one side is only reading we can safely do this on a dual port memory. Otherwise we would have to navigate access with a controller and some FIFOs]*

```vhdl
entity framebuffer is port (
            clk1, en1, en2, ld : in std_logic;
            addr1, addr2 : in std_logic_vector(11 downto 0);
            wr_en1: in std_logic;
            din1: in std_logic_vector(15 downto 0);
            dout1, dout2 : out std_logic_vector(15 downto 0)
);
end framebuffer;
```

See the following VHDL design for reference:

```vhdl
-- A parameterized, inferable, true dual-port, dual-clock block RAM in VHDL.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bram_tdp is
generic (
    DATA    : integer := 72;
    ADDR    : integer := 10
);
port (
    -- Port A
    a_clk   : in  std_logic;
    a_wr    : in  std_logic;
    a_addr  : in  std_logic_vector(ADDR-1 downto 0);
    a_din   : in  std_logic_vector(DATA-1 downto 0);
    a_dout  : out std_logic_vector(DATA-1 downto 0);

    -- Port B
    b_clk   : in  std_logic;
    b_wr    : in  std_logic;
    b_addr  : in  std_logic_vector(ADDR-1 downto 0);
    b_din   : in  std_logic_vector(DATA-1 downto 0);
    b_dout  : out std_logic_vector(DATA-1 downto 0)
);
end bram_tdp;

architecture rtl of bram_tdp is
    -- Shared memory
    type mem_type is array ( (2**ADDR)-1 downto 0 ) of std_logic_vector(DATA-1 downto 0);
    shared variable mem : mem_type;
begin

-- Port A
process(a_clk)
begin
    if(a_clk'event and a_clk='1') then
        if(a_wr='1') then
            mem(conv_integer(a_addr)) := a_din;
        end if;
        a_dout <= mem(conv_integer(a_addr));
    end if;
end process;

-- Port B
process(b_clk)
begin
    if(b_clk'event and b_clk='1') then
        if(b_wr='1') then
            mem(conv_integer(b_addr)) := b_din;
        end if;
        b_dout <= mem(conv_integer(b_addr));
    end if;
end process;

end rtl;
```

Figure 1: Dual Port RAM Infer Sample

Part 3: Where'd you get your CPU?

Background:

We have finally reached it, the big kahuna. In this section we are going to make the brains of the operation, the controller. Everything ends up becoming a state in our machine and adding custom instructions is as simple as adding more states to the machine. This makes the controller mostly mindless grunt work, since the FSM states themselves are relatively simple. Our FSM models a multi-cycle CPU with an average CPI of roughly 5 clock cycles.

Task(s):

- Create the FSM for the control unit. It should have the following entity declaration. In order to facilitate the development of the controller, a simplified state diagram and state behavior table have been given below. They will need more states in some areas due to latency with memories and the ALU (see page below).

```
entity controls is port (
      -- Timing Signals
      clk, en, rst : in std_logic;
      -- Register File IO
      rID1, rID2 : out std_logic_vector(4 downto 0);
      wr_enR1, wr_enR2 : out std_logic;
      regrD1, regrD2 : in std_logic_vector(15 downto 0);
      regwD1, regwD2 : out std_logic_vector(15 downto 0);
      -- Framebuffer IO
      fbRST : out std_logic;
      fbAddr1 : out std_logic_vector(11 downto 0);
      fbDin1 : in std_logic_vector(15 downto 0);
      fbDout1 : out std_logic_vector(15 downto 0);
      -- Instruction Memory IO
      irAddr : out std_logic_vector(13 downto 0);
      irWord : in std_logic_vector(31 downto 0);
      -- Data Memory IO
      dAddr : out std_logic_vector(14 downto 0);
      d_wr_en : out std_logic;
      dOut : out std_logic_vector(15 downto 0);
      dIn : in std_logic_vector(15 downto 0);
      -- ALU IO
      aluA, aluB : out std_logic_vector(15 downto 0);
      aluOp : out std_logic_vector(3 downto 0);
      aluResult : in std_logic_vector(15 downto 0);
      -- UART IO
      ready, newChar : in std_logic;
      send : out std_logic;
      charRec : in std_logic_vector(7 downto 0);
      charSend : out std_logic_vector(7 downto 0)
);
end controls
```

- Simulate your controller for a simple arithmetic instruction by asserting the instruction "add $r3 $r4 $r5" (x"00C85000"). Verify that it goes through the proper states and asserts the correct control signals.

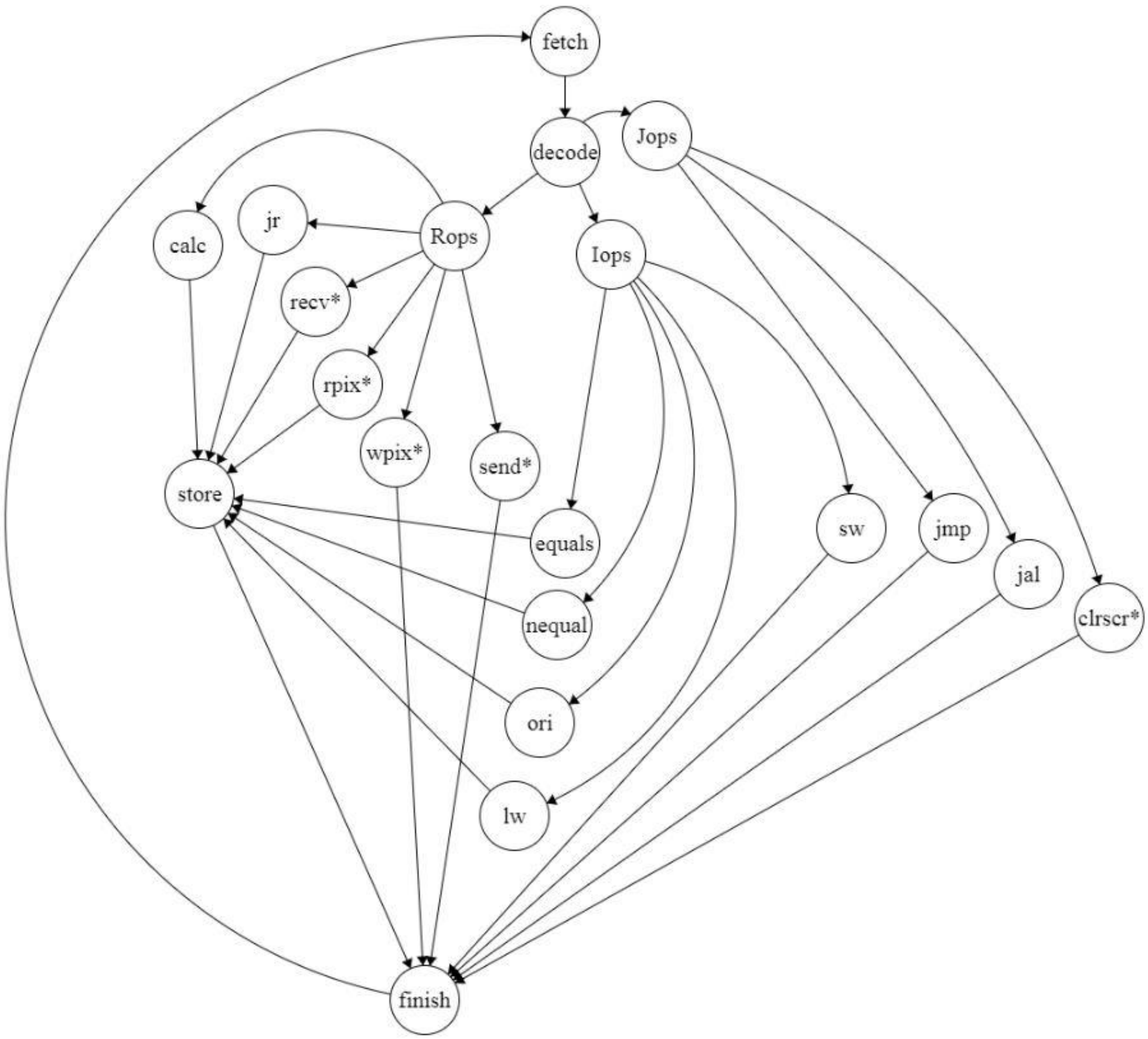

Figure 2: "controls" Simplified FSM Diagram (asterisk denotes ASIP instruction)

NOTE: Reading from irMem and dMem has a 1 clock cycle latency. Your results will not be available until the next rising edge after you assert your control signals. This 1 cycle latency also exists for reading the result from your ALU

NOTE: Any control signals asserted should be immediately de-asserted on the next state in your state machine

Current State	Actions	Next State
fetch	get current pc from reg into signal	decode
decode	store irMem[pc_signal] into a signal store pc_signal+1 into register 1	R_ops if opcode top bits are 00 or 01 else I_ops if 10 else J_ops
Rops	Break up instruction into arguments Use arguments to fetch register contents for reg2 and reg3 into signals	jr if opcode is 01101 else recv if 01100 else rpix if 01111 else wpix if 01110 else send if 01011 else calc
Iops	Break up instruction into arguments Use arguments to fetch register contents for reg2 into signal	equals if opcode bottom 3 bits are 000 else nequal if 001 else ori if 010 else lw if 011 else sw
Jops	Break up instruction into arguments	jmp if opcode is 11000 else jal if 11001 else clrscr
calc	Apply the register operands and the correct opcode to the ALU and store the result into an alu result signal	store
store	Store the alu result signal into the appropriate register given by argument reg1	finish
jr	Read the register value specified and store it in alu result signal	store
recv (asip)	Store charRec into alu result signal	recv if newChar is 0 else store
rpix (asip)	Read the framebuffer memory at the address of the value in reg2 and store it in alu result signal	store
wpix (asip)	Store the value read from reg2 into framebuffer[reg1]	finish
send (asip)	Make send 1 and assign the value read from reg1 to charSend	If ready is 1 finish, else send
equals	If values equal set alu signal to immediate and set reg1 signal to pc id	store
nequal	If values not equal set alu signal to immediate and set reg1 signal to pc id	store
ori	Store the result of the immediate bitwise ORed with the value from reg2 into the alu signal	store
lw	Set the value of the alu signal to the value in dmem[reg2+imm]	store
sw	Store the value of reg1 into dmem[reg2+imm]	finish
jmp	Set the value of the pc register to the immediate	finish
jal	Set the value of the ra register to the value of the pc register and set the value of the pc register to the immediate	finish
clrscr (asip)	Set fbRST to 1	finish
finish	De-assert any required control signals	fetch

Table 2: "controls" State Transition and Behavior Table

Part 4: This isn't even my Final Form

<u>Background:</u>

In order to implement this design on the board, which is clearly gargantuan, we are going to take advantage of a graphical entry tool called the IP Integrator. In order to start working in the IP Integrator, click "Create Block Design" under the IP Integrator flow tab. Specify some arbitrary name for you design and click OK. This will bring you into the integrator window. From here, drop in your designs by right clicking inside the main window and selecting "Add Module" double click the module you wish to add and it will be dropped into the block diagram view. Ignore all connection automation prompts.

Once all your modules have been dropped in, you can connect them by clicking on a port and dragging the connection to another port. In order to make ports go to your top level IO, right click and select "Make External". Then rename the external contact to match the name in your XDC file by right clicking the port, selecting "External Port Properties", and renaming the port name.

After connecting all of your modules, you will need to add ports that are tied to high impedance (Our CTS and RTS signals for the UART PMOD). In order to do so, go to the ports list in the left hand helper window. Right click the "ports" section and select "Create port". Set the name of your top level port and the direction. Then click OK. That's it, it stays unconnected to be tied to high impedance via a tri-state buffer.

Now click the checkmark icon in order to validate your block design and check for any errors. Once that is done the design is finished.

Finally, go back to the "Project Manager" flow tab. Right click your block diagram file and select "Create HDL Wrapper". The tool will generate warnings about width mismatches for our memories and our controller. This is fine. It will only connect the lower bits together, which is what we want. Then set the VHDL file it creates as your top level design. You are now ready to proceed to synthesis.

Task(s):

- Create an appropriately modified XDC file.

- Implement your design in a top level according to the simplified block diagram given below in order to test it on the board using the IP Integrator Tool described in the background section above. Please include the block diagram image you generate in your final lab report.

- Create a single port rom, called "irMem" (Our instruction memory), using the Xilinx Block Memory IP in the same manner as described in Lab 4 Part 2. Set the width to 32 bits and the depth to 16384 (a total of 64 KiB). *[Notice how this memory is not large enough to have a 16 bit address space since we are addressing by double word rather than byte. Our processor has a "virtual" address space of 16 bits for the instruction memory but our physical system only has a 14 bit instruction address space due to memory limitations. In our top level design, we will ignore the upper 2 bits of our address when addressing this memory]*. Disable any output registering and set the enables to always enabled. In the "Other Options" tab, set it to initialize using the text coe file generated in Part 1.

- Create a single port ram, called "dMem" (our data memory), using the Xilinx Block Memory IP in the same manner as described in Lab 4 Part 2. Set the width to 16 bits and the depth to 32768 (a total of 64 KiB). *[Notice how this memory is not large enough to have a 16 bit address space since we are addressing by word rather than byte. Our processor has a "virtual" address space of 16 bits for the data memory but our physical system only has a 15 bit data address space due to memory limitations. In our top level design, we will ignore the top bit of our address when addressing this memory]*. Disable any output registering and set the enables to always enabled. In the "Other Options" tab, set it to initialize using the data coe file generated in Part 1.

Debug Hints:

- You can simulate your entire HDL wrapper and watch your entire system execute cycle by cycle. Very useful for testing your design. It will behave exactly the same way in the physical system. Be careful of sensitivity lists for processes to avoid mismatch vs. hardware.

- If Vivado decides to simulate post synthesis and dissolve your busses into wires, copy the top level HDL wrapper and the other HDL files into a new project. Generate the dMem and irMem using the IP Library like in lab 4. Modify the component names in the HDL wrapper. Now you can simulate the RTL version and add signals such as your state to get better waveforms that are easier to read as well as a properly enumerated state instead of one-hot encoded.

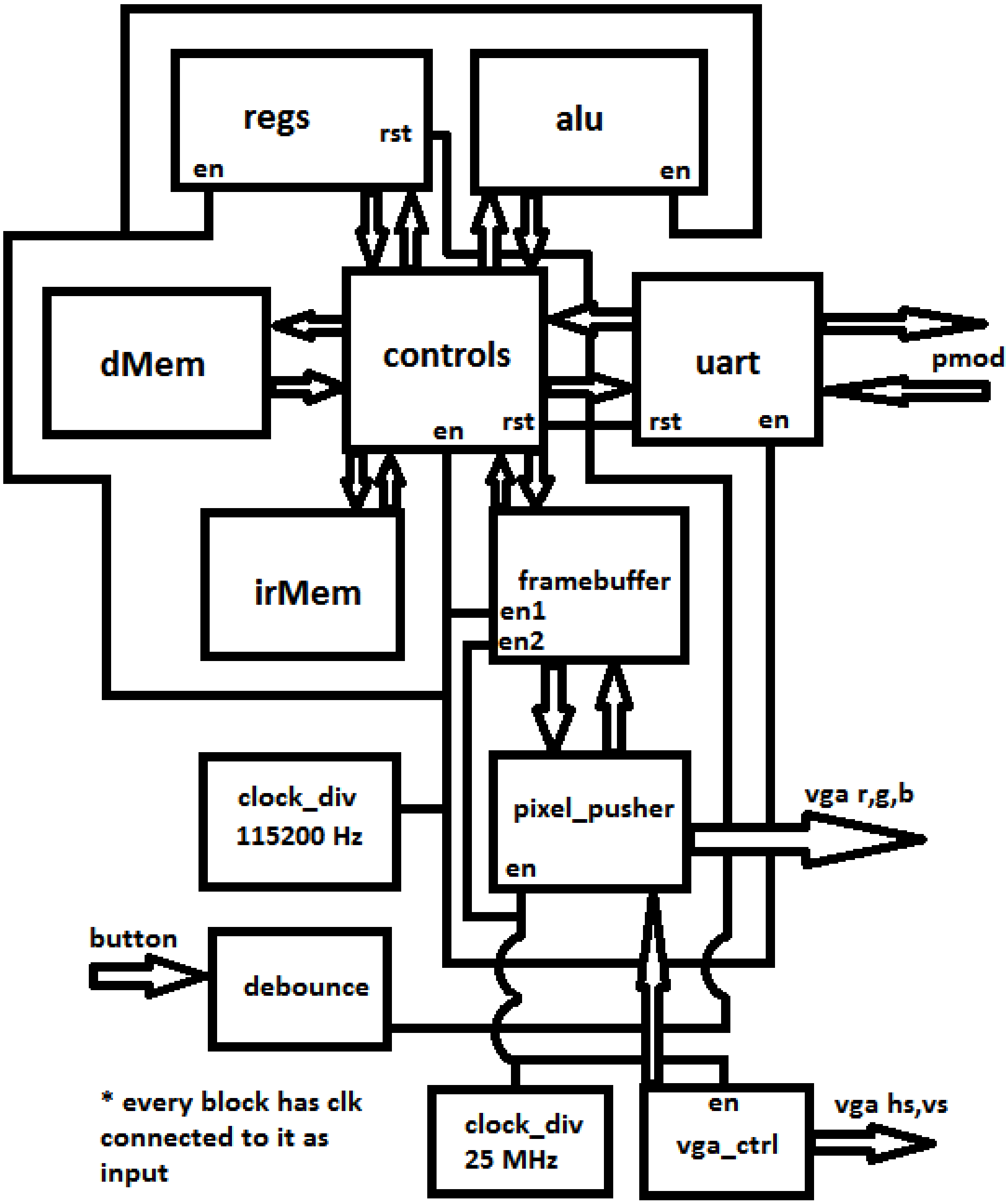

Figure 3: Lab 5 Simplified Top Level Block Diagram

Extra Credit:

Task(s):

- (0.5 points) Implement a paging system for your instruction and data memories. Use the top unused bits for their addressing to mux IO to multiple chips to allow for in-place memory expansion of the system (think adding RAM to a computer).
- (0.5 points) Unlink the UART from all the other components' enable domain and make it have its own dedicated clock divider.
- (0.5 points) Make the max value for the UART clock divider come from $r31 such that you can write a value to $r31 using the processor to change the UART baud rate.
- (1 point) Design and specify a video DMA block that takes as input a source base address in data memory, a destination base address in video memory, and a number of bytes. It then copies the memory from the source to the destination. You will need to convert your Data Memory to a dual port ram and add an instruction to the processor ISA by adding some states to the control FSM and modifying the assembler.
- (1 point) Implement the video DMA block that you have designed and test it with a sample program of your design.

Report:

See the associated lab report format guideline in Resources

Sources:

Lab 1 Manual

Lab 2 Manual

Lab 3 Manual

Lab 4 Manual

MIPS I ISA

https://danstrother.com/2010/09/11/inferring-rams-in-fpgas/

Instruction	**Opcode**	**Format**	**Meaning**
add	“00000”	[op][reg1][reg2][reg3]	reg1 = reg2 + reg3
sub	“00001”	[op][reg1][reg2][reg3]	reg1 = reg2 - reg3
sll	“00010”	[op][reg1][reg2][*reg3]	reg1 = reg2 << 1
srl	“00011”	[op][reg1][reg2][*reg3]	reg1 = reg2 >> 1
sra	“00100”	[op][reg1][reg2][*reg3]	reg1 = reg2 >>> 1
and	“00101”	[op][reg1][reg2][reg3]	reg1 = reg2 & reg3
or	“00110”	[op][reg1][reg2][reg3]	reg1 = reg2 | reg3
xor	“00111”	[op][reg1][reg2][reg3]	reg1 = reg2 ^ reg3
slt	“01000”	[op][reg1][reg2][reg3]	reg1 = (reg2 < reg3) ? 1 : 0
sgt	“01001”	[op][reg1][reg2][reg3]	reg1 = (reg2 > reg3) ? 1 : 0
seq	“01010”	[op][reg1][reg2][reg3]	reg1 = (reg2 == reg3) ? 1 : 0
send (asip)	“01011”	[op][reg1][*reg2][*reg3]	sendUART(reg1[7:0])
recv (asip)	“01100”	[op][reg1][*reg2][*reg3]	reg1 = x”00” & recvUART()
jr	“01101”	[op][reg1][*reg2][*reg3]	pc = reg1
wpix (asip)	“01110”	[op][reg1][reg2][*reg3]	framebuffer[reg1[11:0]] = reg2[15:0]
rpix (asip)	“01111”	[op][reg1][reg2][*reg3]	reg1 = framebuffer[reg2[11:0]]
beq	“10000”	[op][reg1][reg2][imm]	if(reg1 == reg2) pc = imm
bne	“10001”	[op][reg1][reg2][imm]	if(reg1 != reg2) pc = imm
ori	“10010”	[op][reg1][reg2][imm]	reg1 = reg2 | imm
lw	“10011”	[op][reg1][reg2][imm]	reg1 = dmem[reg2 + imm]
sw	“10100”	[op][reg1][reg2][imm]	dmem[reg2 + imm] = reg1
j	“11000”	[op][imm]	pc = imm
jal	“11001”	[op][imm]	ra = pc, pc = imm
clrscr (asip)	“11010”	[op][*imm]	framebuffer[all] = 0

Note: * means operand ignored

Table 3: GRISC Instruction Set

Register Index	**Nickname**	**Size (bits)**	**Special Behavior**
0	$zero	16	Resets to 0 every clock tick
1	$pc	16	Program Counter
2	$ra	16	Return Address for JAL
3-31	$r3-$r31	16	

Table 4: GRISC Register File Organization

Instruction Type	**Opcode (4 downto 3)**	**Format**
R	“00” or “01”	[op][reg1][reg2][reg3]
I	“10”	[op][reg1][reg2][imm]
J	“11”	[op][imm]

Table 5: GRISC Instruction Decoding Table