Lab 4: Register File and Memory
10 points

Instructor: Yifeng Zhu[*]

Due: One week

Objectives:

- Build a register file.
- Build instruction memory and data memory.
- Interface with the VGA debug environment.

1. Overview

A ***combinational circuit*** neither contains a periodic clock signal nor has any provisions for storage. No feedback is involved, and the output always depends on the input provided. The name combinational is derived from the combinations of logic gates used for such circuits.

A ***sequential circuit*** involves feedback and has memory (such as registers and RAM). It also has a periodic clock signal; hence, the output is a function of time in addition to being a function of inputs and previous outputs. The name *sequential* is derived as the output is produced in sequences as the clock circuit enables and disables the functioning. (A latch is also a sequential circuit but has no clock signal and hence is a particular case. It is also the basic building block of any sequential circuit.)

Registers have the following key features.

- Their current "state" depends on the past sequence of inputs they have seen.
- They respond to a "clock" signal, *i.e.*, they only change behavior or state at the instant in time when the clock signal changes from 0 to 1. This instant is called the "positive edge" of the clock signal.

In the pipelined processor design, you must use onboard RAM for instruction and data memory. Note that the instruction memory and data memory are separated. We are following Harvard Architecture

In this lab, you will instantiate a register and memory components, store values in them, retrieve those values, and display them on VGA.

2. Register

A register is used to remember a multi-bit value for later use. Each register can remember one multi-bit value. We call the value currently remembered in the register the register's "value" or the value "stored" by the register.

[*] The author (Yifeng Zhu) gratefully acknowledges borrowing parts of this homework assignment from "ENGINEERING 100 (Section 700) Introduction to Computing Systems" ©2006 by Dr. Peter Chen.

Four input signals control the operation of a register: ***clock***, ***reset***, W***riteEnable***, and ***data_in***. The data_in has 4 bits in this lab. A register supports two operations (based on the values of these signals):

- **Reset**: sets the value of the register to all 0s. A reset operation will occur at the positive edge of the clock if the value of the reset signal at that time is 1.
- **WriteEnable** changes the value currently stored in the register to equal the value of the input parameter data_in. For shorthand, we say we are "writing data_in to the register". A write will occur at the positive edge of the clock if the value of the write signal at that time is 1.

The register outputs a multi-bit value, data_out, the 4-bit value currently stored in the register. This value changes immediately after the value stored in the register changes.

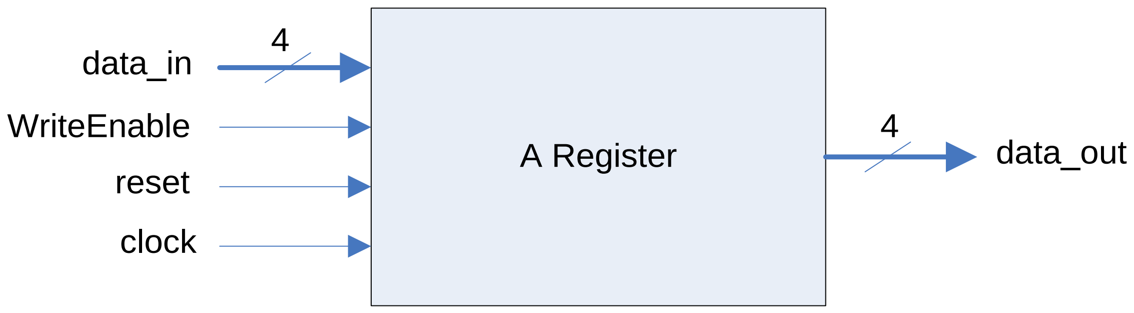


Figure 1. A simple register that has only 4-bit

```
// file register.v

module register(
    input wire clock,
    input wire reset,
    input wire write,
    input wire [3:0] data_in,
    output reg [3:0] data_out);

    always @(posedge clock) begin
        if (reset == 1'b1) begin
            data_out <= 4'h0;
        end else if (write == 1'b1) begin
            data_out <= data_in;
        end else begin
            data_out <= data_out;
        end
    end

endmodule
```

Read *register.v* and identify the operations supported by this component (which is a 4-bit register). This module introduces two new Verilog constructs:

- **always @(posedge clock)** specifies that the actions in this always block should occur only at the positive edge of the clock. These are called "edge triggered" always blocks.
- <= is used as the assignment operator (instead of the normal = operator) within an edge-triggered always block.

3. Register File

A register file is an array of registers. Like a register, the signals control a register file, including the ***clock***, ***reset***, ***WriteEnable*** and ***data_in***. Like a register, a register file outputs a signal ***data_out***. However, because memory is an array of registers, it needs additional information to control which element of the array is being operated. This additional information is called the memory's **address**, similar to an array index.

For debug purposes, the register file has two extra inputs (read_address_debug, and clock_debug) and one extra output (data_out_debug). The read_address_debug can be set up by the switches on the board, and the data_out_debug can be displayed on the 7-segment LEDs or the LCD. A separate clock_debug allows you to check the value of a specific register even when the processor is not running.

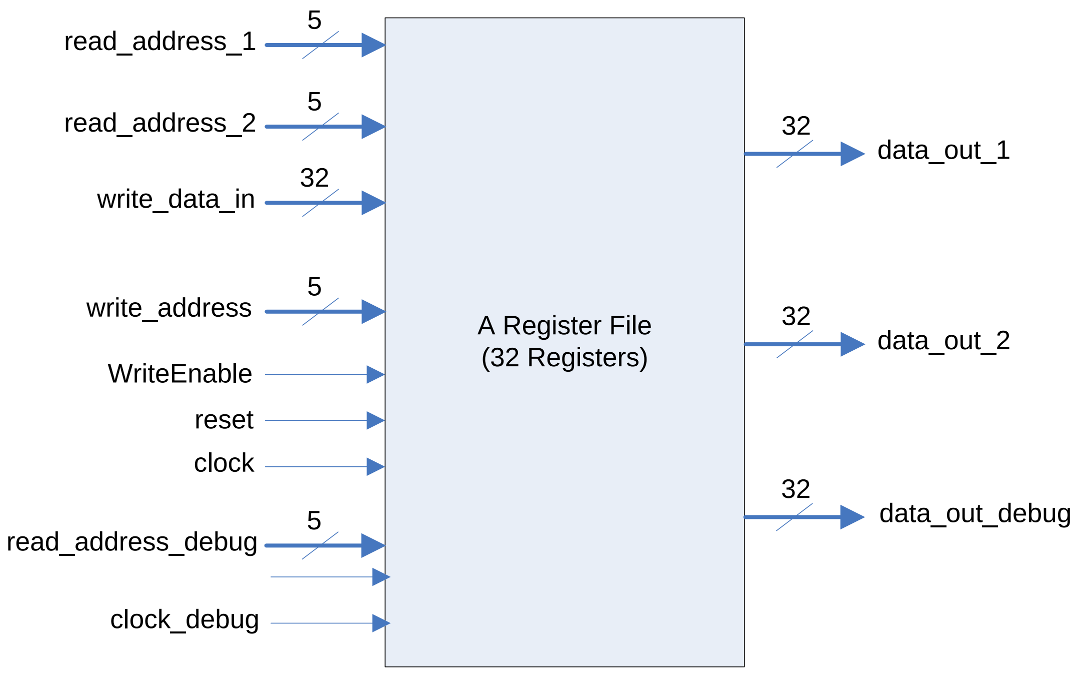


Figure 2. A register file with 32 registers, and each register has 32 bits

The address is stored in an internal register of the memory. One must first write an address to the address port to operate on the register file. The current value of the address determines which register is written to or read from on subsequent access operations.

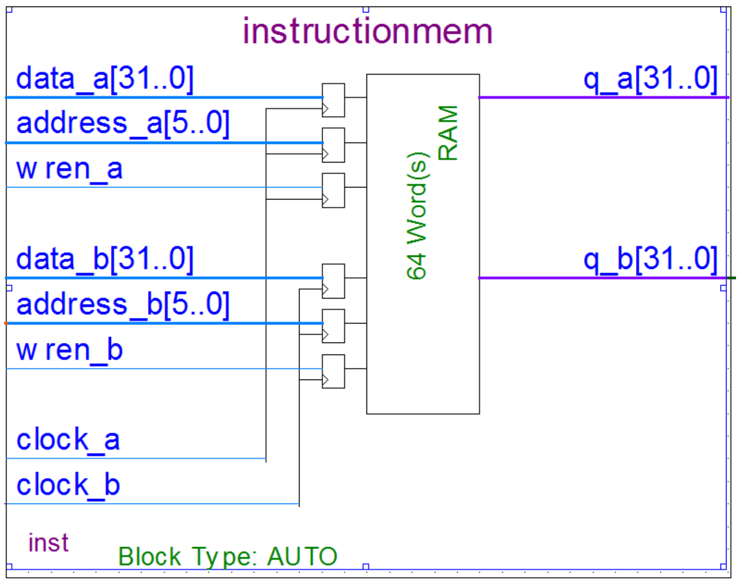
Note that the register file uses two clocks.

1. The clock_debug must be vga_clk because the data inputs to the VGA module must be synchronized with the vga_clock.
2. The clock signal is your processor's main clock. For example, if we want to run the processor at a clock frequency of 1Hz, we can use the 1 Hz clock output from the clock divider to drive the register file. In addition, this clock signal can also be connected to a switch so that we can manually generate the clock signal. This allows us to debug the processor step by step manually.

4. Building Instruction and Data Memory

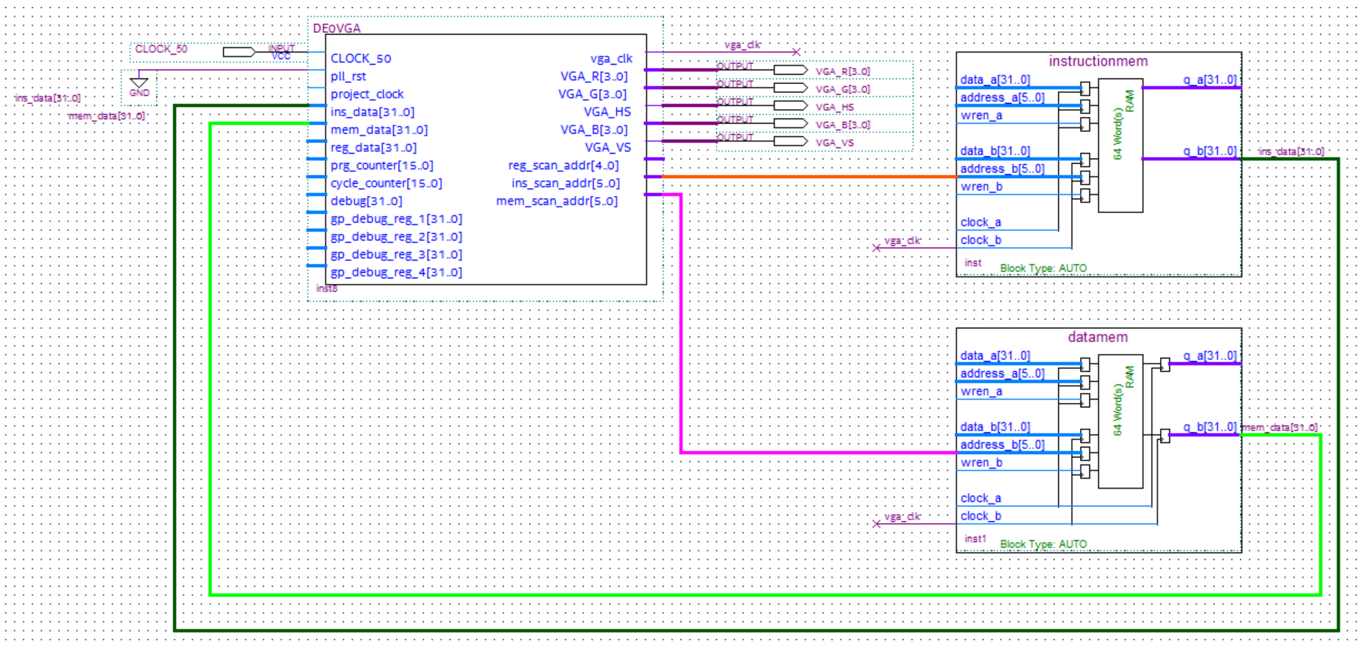
We will use the on-chip memory in this lab to build instruction and data memory. Because we are implementing a processor with Harvard architecture, we need two separate memory modules. We will use the IP provided by Altera. We use ***RAM 2-PORT IP*** to build the instruction and data memory. We use dual ports because one port will be used for the processor to fetch instructions or data, and the other port will be used for debugging module.

In Quartus, navigate to **Tools** → **IP catalog** → **Library** → **Basic Functions** → **On Chip Memory**

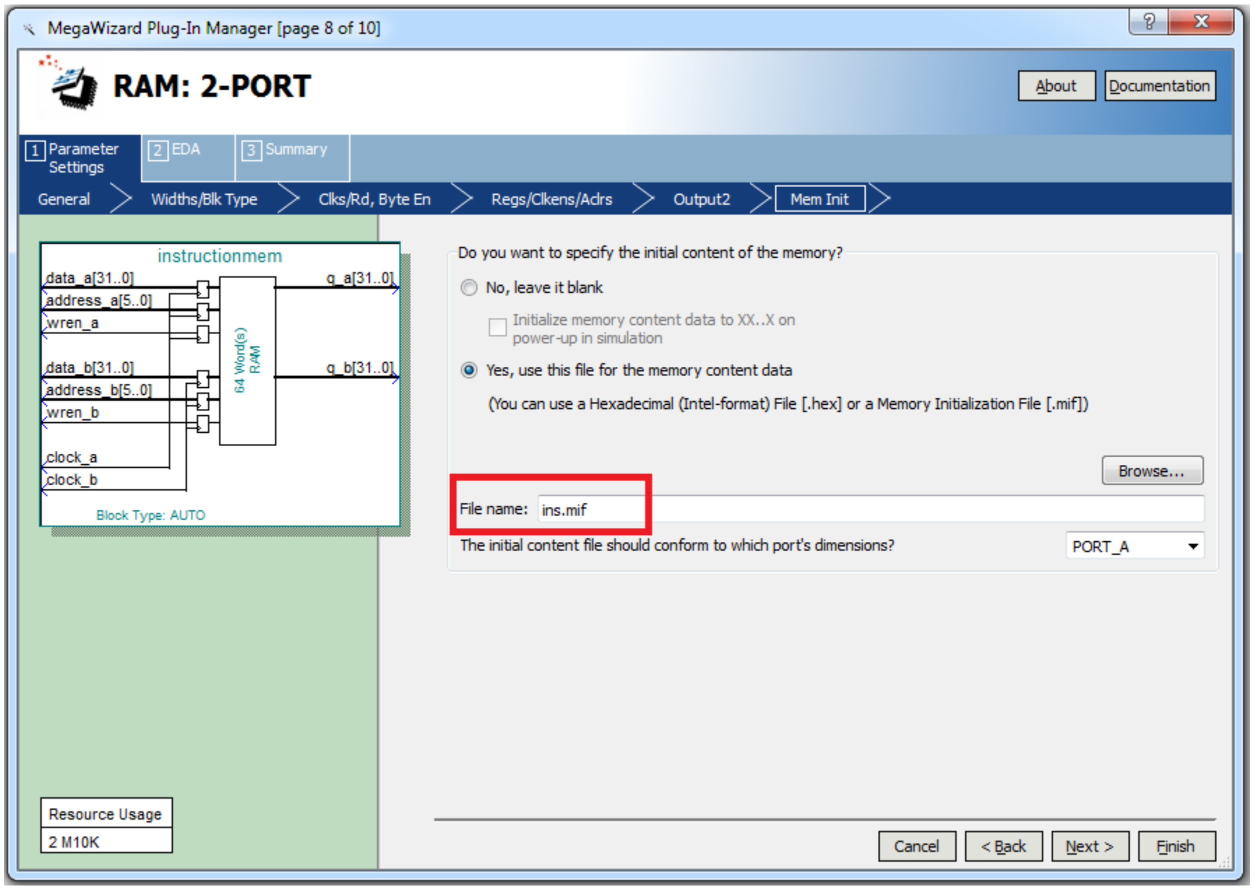


The following highlights the key features of each dual-port RAM.

- The memory size is in words. Conventionally, the memory size is in bytes. Therefore, after fetching one instruction, the program counter is added by 4, instead of 1.
- The memory module has separate dual clocks, with one clock for each port. The clock input of the port used for debugging should be the clock output of the VGA module (vga_clk).
- Each memory consists of 64 words. Our debug module can only show 64 words.
- The data output of each output port should NOT be registered.



5. Memory Initialization Format (MIF)


MegaWizard Plug-In Manager [page 8 of 10]

RAM: 2-PORT

Do you want to specify the initial content of the memory?

No, leave it blank [✓ Yes, use this file for the memory content data]

Initialize memory content data to XX..X on power-up in simulation

(You can use a Hexadecimal (Intel-format) File [.hex] or a Memory Initialization File [.mif])

Browse...

File name: ins.mif

The initial content file should conform to which port's dimensions? PORT_A

Resource Usage
2 M10K

The data of the RAM and Rom can be initialized according to an ASCII text file named Memory Initialization Format (with the extension *.mif*). The following lists basic 32-bit integer instructions and their corresponding MIF files for the instruction and data memory.

```
        .data   # Data Section
aArray: .word 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048,
              4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288,
              1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864,
              134217728, 268435456, 536870912, 1073741824, 2147483648,
aSize:   .word 32

bArray:  .word 0xDEADBEEF, 0xBAADF00D, 0xBADDCAFE, 0xFACEFEED, 0xFEEDBABE,
               0xF00DBABE, 0xBAAAAAAD, 0xFEEDC0DE
bSize:   .word 8

cArray:  .word 13, 6, 5, 10, 15, 19, 21, 17, 14, 3, 12, 18, 7, 4, 11, 20, 22, 23, 24, 2, 1, 8, 16, 9,
cSize:   .word 24

        .text   # Code Section
main:
        # RV32I Base Integer Instructions
        add t1, t2, t3
        sub t1, t2, t3
        xor t1, t2, t3
        or  t1, t2, t3
        and t1, t2, t3
        sll t1, t2, t3
        srl t1, t2, t3
        sra t1, t2, t3
        slt t1, t2, t3
        sltu t1, t2, t3

        addi t1, t2, -100
        xori t1, t2, -100
        ori  t1, t2, -100
        andi t1, t2, -100
        slli t1, t2, 10
        srli t1, t2, 10
        srli t1, t2, 10
        srai t1, t2, 10
        slti t1, t2, -100
        sltiu t1, t2, 10

        la t2, cArray
        lb t1, -32(t2)
        lh t1, -32(t2)
        lw t1, -32(t2)
        lbu t1, -32(t2)
        lhu t1, -32(t2)

        sb t1, -64(t2)
        sh t1, -64(t2)
        sw t1, -64(t2)

        beq t1, t2, exit
        bne t1, t2, exit
        blt t1, t2, exit
        bge t1, t2, exit
        bltu t1, t2, exit
        bgeu t1, t2, exit

        jal t1, exit
        jalr t1, t2, -100

        lui t1, 100000
        auipc t1, 100000

exit:   beq zero, zero, exit     # program stops here
```

The following is the MIF file to initialize the instruction memory.

```
DEPTH = 64;              % Memory depth and width are required     %
WIDTH = 32;              % Enter a decimal number                  %
ADDRESS_RADIX = HEX;     % Address and value radixes are optional  %
DATA_RADIX = HEX;        % Enter BIN, DEC, HEX, or OCT; unless     %
                          % otherwise specified, radixes = HEX      %
-- Specify initial data values for memory, format is address : data
CONTENT BEGIN
[00..3F]: 00000000;      % Range - Every address from 00 to 3F = 00000000 %

-- Initialize data
00 : 01c38333;
01 : 41c38333;
02 : 01c3c333;
03 : 01c3e333;
04 : 01c3f333;
05 : 01c39333;
06 : 01c3d333;
07 : 41c3d333;
08 : 01c3a333;
09 : 01c3b333;
0A : f9c38313;
0B : f9c3c313;
0C : f9c3e313;
0D : f9c3f313;
0E : 00a39313;
0F : 00a3d313;
10 : 00a3d313;
11 : 40a3d313;
12 : f9c3a313;
13 : 00a3b313;
14 : 0fc10397;
15 : 05838393;
16 : fe038303;
17 : fe039303;
18 : fe03a303;
19 : fe03c303;
1A : fe03d303;
1B : fc638023;
1C : fc639023;
1D : fc63a023;
1E : 02730463;
1F : 02731263;
20 : 02734063;
21 : 00735e63;
22 : 00736c63;
23 : 00737a63;
24 : 0100036f;
25 : f9c38367;
26 : 186a0337;
27 : 186a0317;
28 : 00000063;
[29..3F] : 00000000; % nop %
END;
```

Note: If multiple values are specified for the same address, only the last value is used.

The following is the MIF file to initialize the data memory.

```
DEPTH = 64;                % Memory depth and width are required     %
WIDTH = 32;                % Enter a decimal number                  %
ADDRESS_RADIX = HEX;       % Address and value radixes are optional  %
DATA_RADIX = HEX;           % Enter BIN, DEC, HEX, or OCT; unless     %
                            % otherwise specified, radixes = HEX      %
-- Specify initial data values for memory, format is address : data
CONTENT BEGIN
[00..3F] : 00000000;       % Range - Every address from 00 to 3F = 00000000 %
-- Initialize data
00 : 00000001;
01 : 00000002;
02 : 00000004;
03 : 00000008;
04 : 00000010;
05 : 00000020;
06 : 00000040;
07 : 00000080;
08 : 00000100;
09 : 00000200;
0A : 00000400;
0B : 00000800;
0C : 00001000;
0D : 00002000;
0E : 00004000;
0F : 00008000;
10 : 00010000;
11 : 00020000;
12 : 00040000;
13 : 00080000;
14 : 00100000;
15 : 00200000;
16 : 00400000;
17 : 00800000;
18 : 01000000;
19 : 02000000;
1A : 04000000;
1B : 08000000;
1C : 10000000;
1D : 20000000;
1E : 40000000;
1F : 80000000;
20 : 00000020;
21 : deadbeef;
22 : baadf00d;
23 : baddcafe;
24 : facefeed;
25 : feedbabe;
26 : f00dbabe;
27 : baaaaaad;
28 : feedc0de;
29 : 00000008;
2A : 0000000d;
2B : 00000006;
2C : 00000005;
2D : 0000000a;
2E : 0000000f;
2F : 00000013;
30 : 00000015;
31 : 00000011;
32 : 0000000e;
33 : 00000003;
34 : 0000000c;
35 : 00000012;
36 : 00000007;
37 : 00000004;
38 : 0000000b;
39 : 00000014;
3A : 00000016;
3B : 00000017;
```

6. Lab Instructions and Requirements

Please note that the following should be implemented in the same project.

6.1 Build the Register File.

a. Implement the register file shown in Figure 2. Initialize the register *i* with a value of *i*, where *i* = 0,1,…,31. For example, register 11 is initialized as `0x0000000A`. Verify the correctness of your design via the VGA display.

6.2 Build the Data Memory.

a. Initialize the data memory as the following. Please note that the data memory address in MIPS code is in terms of bytes, not words. Test your design on the Altera DE2 board using the display and control specification given at the end of this handout.
b. Please note that in the MIPS programs, the data segment starts at **`0x10010000`**. Since the data size of our test programs is small, thus you can ignore the most significant 16 bits in the data address.

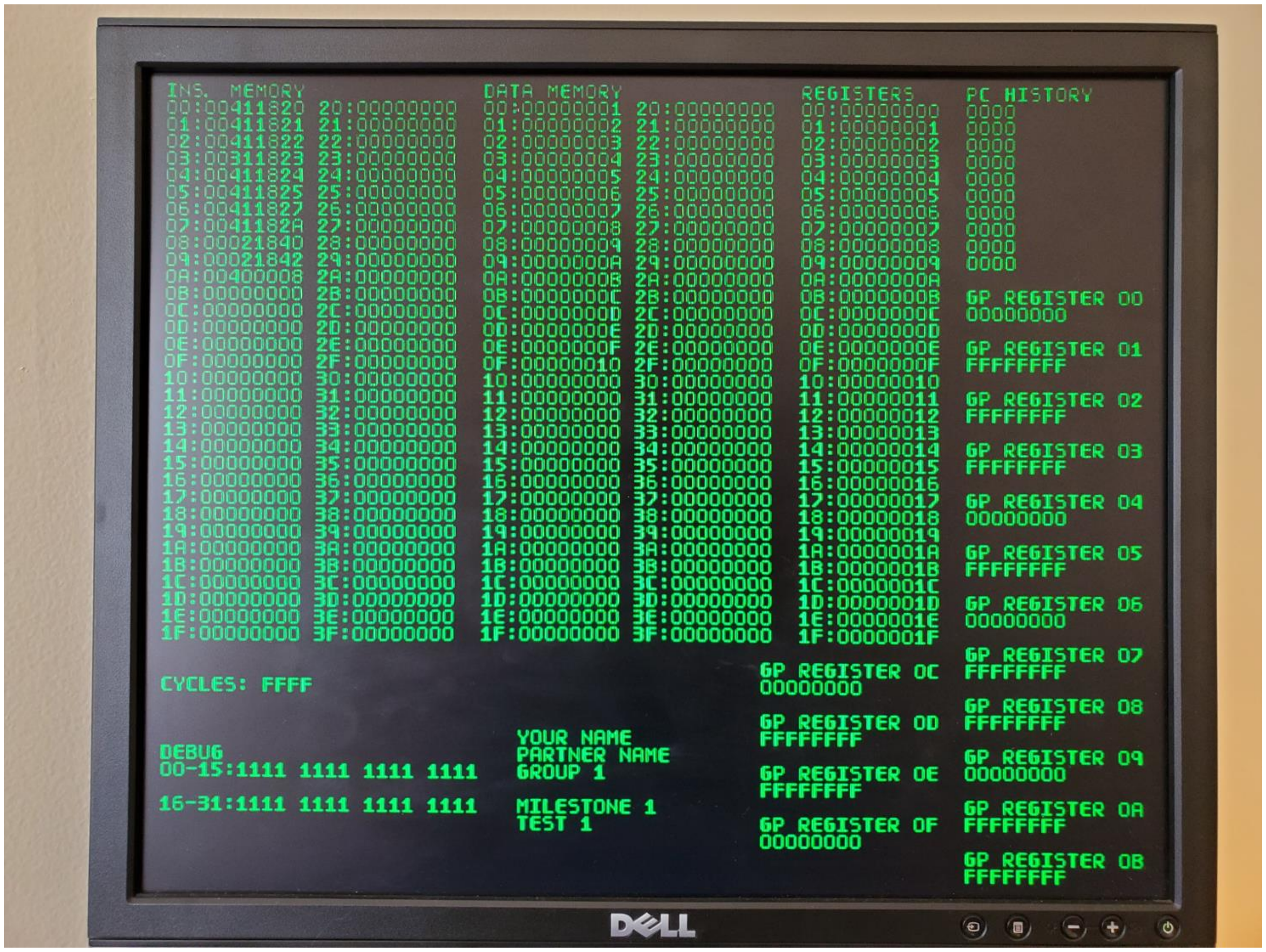
6.3 Build the Instruction Memory.

`

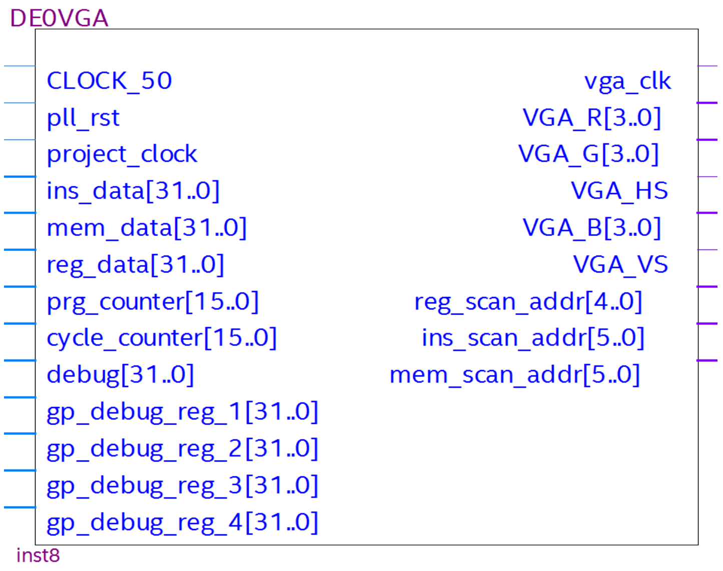
a. Initialize the instruction memory as the following. Please note that the instruction memory address in MIPS code is in terms of bytes, not words. Test your design on the Altera DE2 board using the display and control specification given at the end of this handout.
b. Please note that in the RISC-V programs, the instruction segment starts at **`0x00400000`**. Since the data size of our test programs is small, thus you can ignore the most significant 12 bits in the data address.

VGA Display Module

Acknowledgment: This module was designed and implemented by Tristan Woodrich.



```
INS. MEMORY                  DATA MEMORY                  REGISTERS    PC HISTORY
00:00411820 20:00000000      00:00000001 20:00000000      00:00000000  0000
01:00411821 21:00000000      01:00000002 21:00000000      01:00000001  0000
02:00411822 22:00000000      02:00000003 22:00000000      02:00000002  0000
03:00311823 23:00000000      03:00000004 23:00000000      03:00000003  0000
04:00411824 24:00000000      04:00000005 24:00000000      04:00000004  0000
05:00411825 25:00000000      05:00000006 25:00000000      05:00000005  0000
06:00411827 26:00000000      06:00000007 26:00000000      06:00000006  0000
07:0041182A 27:00000000      07:00000008 27:00000000      07:00000007  0000
08:00021840 28:00000000      08:00000009 28:00000000      08:00000008  0000
09:00021842 29:00000000      09:0000000A 29:00000000      09:00000009  0000
0A:00400008 2A:00000000      0A:0000000B 2A:00000000      0A:0000000A
0B:00000000 2B:00000000      0B:0000000C 2B:00000000      0B:0000000B  GP REGISTER 00
0C:00000000 2C:00000000      0C:0000000D 2C:00000000      0C:0000000C  00000000
0D:00000000 2D:00000000      0D:0000000E 2D:00000000      0D:0000000D
0E:00000000 2E:00000000      0E:0000000F 2E:00000000      0E:0000000E  GP REGISTER 01
0F:00000000 2F:00000000      0F:00000010 2F:00000000      0F:0000000F  FFFFFFFF
10:00000000 30:00000000      10:00000000 30:00000000      10:00000010
11:00000000 31:00000000      11:00000000 31:00000000      11:00000011  GP REGISTER 02
12:00000000 32:00000000      12:00000000 32:00000000      12:00000012  FFFFFFFF
13:00000000 33:00000000      13:00000000 33:00000000      13:00000013
14:00000000 34:00000000      14:00000000 34:00000000      14:00000014  GP REGISTER 03
15:00000000 35:00000000      15:00000000 35:00000000      15:00000015  FFFFFFFF
16:00000000 36:00000000      16:00000000 36:00000000      16:00000016
17:00000000 37:00000000      17:00000000 37:00000000      17:00000017  GP REGISTER 04
18:00000000 38:00000000      18:00000000 38:00000000      18:00000018  00000000
19:00000000 39:00000000      19:00000000 39:00000000      19:00000019
1A:00000000 3A:00000000      1A:00000000 3A:00000000      1A:0000001A  GP REGISTER 05
1B:00000000 3B:00000000      1B:00000000 3B:00000000      1B:0000001B  FFFFFFFF
1C:00000000 3C:00000000      1C:00000000 3C:00000000      1C:0000001C
1D:00000000 3D:00000000      1D:00000000 3D:00000000      1D:0000001D  GP REGISTER 06
1E:00000000 3E:00000000      1E:00000000 3E:00000000      1E:0000001E  00000000
1F:00000000 3F:00000000      1F:00000000 3F:00000000      1F:0000001F
                                                                       GP REGISTER 07
                                                       GP REGISTER 0C  FFFFFFFF
CYCLES: FFFF                                           00000000
                                                                       GP REGISTER 08
                                                       GP REGISTER 0D  FFFFFFFF
                                   YOUR NAME           FFFFFFFF
DEBUG                              PARTNER NAME                        GP REGISTER 09
00-15:1111 1111 1111 1111          GROUP 1             GP REGISTER 0E  00000000
                                                       FFFFFFFF
16-31:1111 1111 1111 1111          MILESTONE 1                         GP REGISTER 0A
                                   TEST 1              GP REGISTER 0F  FFFFFFFF
                                                       00000000
                                                                       GP REGISTER 0B
                                                                       FFFFFFFF
```



This package contains all the files need to get VGA running on DE0-CV boards. NOTE: The displays in the Kepware lab tend to display the output differently from each other; this is a function of the screens, not the code.

Files:

- *char_engine.v*: The module in charge of taking data and rendering it to video memory.
- *color_encoder.v*: The module in charge of determining the color of values from video memory.
- *DEOVGA.bdf*: block diagram of the VGA system
- *DEOVGA.bsf*: the symbol file of the above block diagram
- *vga_control.v*: The module that controls the timings necessary for screen output.
- *vgaclk.v*: Changes a 50mhz clock to a 25.125mhz clock needed for the other modules.
- *vram.v*: The video memory.

Once the files are added, you must connect the following I/O on the symbol to pins on the board.

- *CLOCK_50*: the input of the DE0-CV board's clock signal.
- *VGA_R*[3..0]: the red signal to the VGA network on the DE0-CV
- *VGA_G*[3..0]: the green signal to the VGA network on the DE0-CV
- *VGA_B*[3..0]: the blue signal to the VGA network on the DE0-CV
- *VGA_HS*: Horizontal sync on the VGA port
- *VGA_VS*: Vertical sync on the VGA port

Once connected, you should see the output on a monitor. To display data on the screen, you must put data onto the various inputs on the symbol. Key Inputs and Outputs:

- *vga_clk*: this is the same clock the screen uses to render information; it is necessary to run the debug ports of the register file, instruction and data memory at this clock speed to keep the data in sync.
- *pll_rst*: Resets the phase lock loop driving the VGA clock. It should not be necessary, but it is here in case.
- *project_clock*, and *prg_counter*: These are meant to display your project's program counter. It will show the 10 most recent values the input has received. The project clock needs to be the same clock your processor is using.
- *Instruction memory: ins_data*, and the output *ins_scan_addr*: *ins_scan_addr* is meant to send an address to your instruction memory and receive the returned value in ins_data. For this to occur, your instruction memory must use the vga_clock output.
- *Data memory*: *mem_data*, and *mem_scan_addr*: These behave the same as the instruction memory but are meant to connect to the system memory of your project.
- *Register File*: *reg_dat*, and *reg_scan_addr*: These are meant to connect to your projects register file; work the same as the above, except that it only sends out a 5-bit address.
- *debug*[31..0]: Making any of the bits on the debug output high results in a 1 appearing on the screen in the debug section corresponding to the bit made high.
- *gp_debug_reg_X*: 4 general purpose registers take any data received and will render it as a hex value on the screen.

ECE 473 Lab 5
TA Checkoff Sheet
Total: 10 points

Name: ___________________________

Date: ____________________________

Final Grade: ________________________

TA Signature: _______________________

You need to demo the following three requirements on the DE0 board.

1) Build the instruction memory and show the contents correctly. The instructions values are given in Section 6.3 (2 points)

2) Build the data memory and show the contents correctly. The data values are given in Section 6.2 (2 points)

3) Build the register file and show the contents correctly. The register values are given Section 6.1 (4 points)

4) Test the register file by using a manual clock. Use the following settings to update a register value (2 points).
   - SW[3..0] as the register address
   - SW[9..4] as the register value
   - Register clock signal will be generated by the push button KEY[0]