Contents

`https://utah.instructure.com/courses/461447`
help-cs3710@lists.utah.edu

1 Table of Contents

2 Tuesday August 22

2.1 Course Overview

Working in teams, students employ the concepts of digital logic design and computer organiation to design, implement and test a computing system

2.2 Course Materials

All coursework is linked in on Canvas. There is no textbook.

2.3 Lectures

Meet in the classroom (WEB L 102) most Tuesdays and Thursdays for a short period

2.4 Labs

Last portion of each Tuesday lecture and the entire class period on most THursdays will be spent in the lab (MEB 3133)

2.5 Coursework and Grading

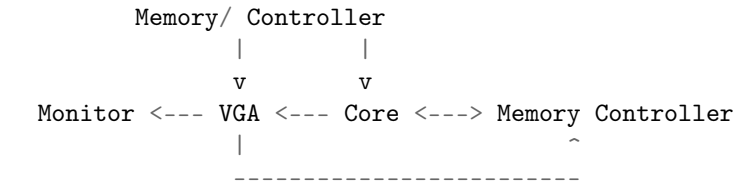
- Final Design Project (25%)
- Student Participation (20%)
- Midterm (15%)
- Checkpoints (40%)

2.6 Class Discussion

2.6.1 Verilog

No need ot execute a program Just Archetecture

2.6.2 Class Overview



3 Thursday August 24

3.1 Verilog

Verilog is an archetecture, where order of assignment doesnt matter

3.2 Nexys3

Diligent Nexys3 is the board Spartan 6 is the FPGA

3.3 ISE Project

- Create Verilog Module
  - Each in a heirarchy
  - Top level should be name of project
- Constraints File
  - In module we set Net's We bind the network to outside pinpads with what voltage and what clock to use
  - Gives time feedback of whether or not instruction will complete in time

Code Should be warning and error free

Go get Master UCF File from Canvas / Files, copy it into project, instead of shitty point & click. Change the name to match variables

3.4 1st Project

{7: 0} leds –> to set a bunch of wires, with 7 being the MSB

```
reg[28:0] counter;
initial counter = 0;
always @ (posedge clk)
    counter <= counter + 1'b1
assign leds = counter{28:0}
```

4 Tuesday September 4

4.1 Debouncing example

```
reg[17:0] down_counter    = 0;
reg[17:0] polling_counter = 0;
reg[15:0] button_counter  = 0;

always @ (posedge clk)
begin
   polling_counter <= polling_counter + 1'b1;
   down_counter    <= down_counter + 1'b1;

   if(polling_counter == 0) // When it rolls over
   begin
       button_counter <= button_counter + down_count[17];
       // down_counter <= 0;
       down_counter   <= (17'b0, switch); // All zero's or all 1's
   end
end
```

4.1.1 Midterm Question

if $\text{down}_{\text{counter}}$ = [011111111...] and $\text{polling}_{\text{counter}}$ = [111111111...]

Would $\text{down}_{\text{counter}}$[17] ever be 1 if $\text{down}_{\text{counter}}$ <= 0; was in the $\text{polling}_{\text{counter}}$ loop?

No: Non-blocking assignment will take the last value it is given to calculate itself at the clock edge.

4.2 VGA Example

He only used assign calls On the posedge clk, he would allow the values to be the next value, so every value of the $\text{next}_{\text{pixel}}$, column, row, color would be calculated

He set VGACOLOR all in one variable, He allowed each pixel to be set for 3 counts

5 Tuesday September 12

5.1 Ternary Branches

out = bool ? A: bool ? c: 0;

5.2 Reduction Operators

&$\text{signal}_{\text{group}}$ = take every bit and then 'and' together

5.3 Scalar Multiplication

'{num {$\text{signal}_{\text{group}}$}}' = num * $\text{signal}_{\text{group}}$

5.4 Linear Feeback Random # Generator

```
initial seed = 500;
seed <= {seed[29:0], seed[30] ^ seed[27]};
```

Can expand by taking {30,29} and {27, 26}

5.5 Glyph

Glyphs = graphic for a character.

- 64 bits to describe each letter
- Stored in a (.coe) which is a file that contains comma seperated binary
- 16-bit words
- 126 bit aligned

Line 0 char	Padding	Line 1 char
64-bits	128	64-bits

5.6 Pixel Generation

```
Req delivered, col and row prepared
edge ________________________________________ latch col/row, reset statemachine

Provides Data
edge ________________________________________ latch address

Computes Memory Address, give to RAM
edge ________________________________________ latch ram address

Extract bits, compute color
edge ________________________________________ latch color (background or foreground)
```

5.7 RAM

```
Port 1                          Port 2

                ---------------
Address -----> |               | <--- Address
Data Out <---- | Block Ram     | ---> Data Out
Data In <----- |               | <--- Data In
WriteEnable -> |               | <--- Write Enable
Clk ---------> |               | <--- Clk
                -------------
```

6 Tuesday September 19

6.1 Glyph COE

#+begin$_{\text{src}}$ C COE FILE ONE CHARS GLYPH

Entry 0	–>	0-1 Word	——>	TOP ROW	Bottom Row
		2-3 Word		01010101	1001001011
		4-5 Word	rightmost leftmost		
		6-7 Word			

#+end$_{\text{Src}}$ C

1. Physical pixel

2. Logical Pixel # [0..640), [0...480)
3. Char row / column
4. mem address that stores that text

6.2 Combinatorial Logic

When building combinatorial logic, always assign something. Even at the beginning will have it avoid having an unintentional latch. Most always @ * can be assigned using a ternary operator.

6.3 Project Timeline

- Form Teams by September 28
- Design assembly language before fall break
- Core running by November 2
- Write code and interface w/ hardware by November
- Write a short document and poster
- Demo = last week of classes

6.4 Core

- Must execute instructions
  - Fetch an instruction from RAM
  - Do it, Advance the PC
- Register File (Assembly Register)
- Must Use 16-bit addressing
- No MIPS / Design Custom assembly
  - Basic arithmatic
  - Jump of some sort
  - Load and Store from mem
  - Conditionals / Branch
- Should use 24-bit addresses
- Should Consider multiword instructions

```
|----------|
|Glyphs    |
|----------|0x60000
|Text      |
|----------|0x40000
|Code      |
|----------|0x00000
```

6.5 Projects

- Meet requirements, be simple, be complete (YES)
- Meet requirements, be ambitiious, be undone (NO)

6.6 Assembler

takes instructions (add, A, B) and translates them into instructions for memory Also keeps track of where things are landing in memory, this is done with a 2-pass assembler.

7 Tuesday September 26

Thursday before fall break

- Document .pdf
  - Instruction set architecture
  - Memory Layout Diagram
  - Example Code
  - Sketch Block Diagram of your computer (Optional)

This should be an engineering document (be very specefic)

7.1 Instruction Set Archetecture

- How many registers ?
  - How many can be read at once ?
  - How many can be written at once ?

```
/*

          -------
  index--- |     |----data out
  index--- |     |---data out
           |REG  |
           |FILE |
```

```
          -------
How many bits of index do you need?
How wide are they? (Each address gives you 16 bits)
*/
```

- Memory - How is it organized?
  - 32k of 16 bit words as block ram

```
/*
          -------
0x000000 |BLOCK|
0x007fff | RAM |
         |     |
         |     |
0x7fffff -------

From 0x007fff to 0x7fffff is pseudo-static ram (off-chip)
*/
```

- How does data get into the processor ?
  - Since multiple devices should be accessing memory, we need an arbitrator that lists memory access priorities.
  - Interrupts, Polling (periodically checks io device), Direct Memory Access (designate certain memory addresses to io)
  - Core should not be directly aware of what the IO does.
- Instuction Set
  - The more instruction you have the more opcode bits you need
  - Have to have memory instructions (write to memory, read from memory) (16-bit memory)
  - List intruction, encodings, address modes
    - add (dest), (immediate) –> takes immediate and adds it to dest
      - 16 / 32 bit encoding in memory (first word should describe its length)
    - (AFTER FALL BREAK) - describe what is happening during each instruction
    - add
      - PC-> address (1) FETCH
      - PC <- PC + 1 (1)
      - Mem -> data (2) GET INSTRUCTION
      - Decide next state (2)
      - REG -> REG + IMMD (3) ADD

- Wise Idea to dedicate one port of dual port ram to VGA.
- Want to decide in the core if you have status (FLAGS FROM 4400)
  - Most processors have status bits that show you the status of last inst
    - Zero (ZF) - set if last op produced 0
      - Branch if 0
    - Carry (CF) - set if last op produced a carry out
    - Overflow (OF) - set if last op produced overflow
    - Negative (NF) - set if last op produced a negative num
- Program Counter is out of register file, latched value
- Example Code
  - loop
  - accesses memory (adding array values, or printing to the screen)
  - calling / returning from code (subroutine)

7.2 Midterm

- One single side 8.5' * 11' handwritten notes allowed (must turn in after exam)
- Write an exam that makes certain that people understand VGA, timing and verilog code
  - cycle diagrams, small state machine verilog codes
  - short answer questions (what does this do)
  - how would this mux happen in verilog ?

  Data signal will be available in cycle 0, output value within a spec. Write verilog STUDY TIMING

revscare@gmail.com keotantritor@gmail.com logan.ropelato@utah.edu

8 Tuesday October 3

9 Tuesday October 17

9.1 Upcoming work

- Thursday (.pdf)
  - instruction set arch
  - memory layout

  - example code
- Tuesday (October 16th)
  - Project specification
  - 1 page description
  - block diagram of verilog
  - Simple, Complete
- Thursdy (October 26th)
  - functioning assembler
- Thursday (November 2)
  - functioning core
  - functioning memory

9.2 Assembler

- Pass 1, collect labels and addresses
  - Start from 0x000000 and count up
  - then when you hit a label store the count value Map<string, Integer>
- Pass 2, Generate Binary
  - when you get to a jump, you can fill in an address

9.3 Core

- Major peices
  - core state machine (sequential)
  - combinatorial logic (produces outputs)
  - Status regesters / counters
  - Temp registers
- Register file
  - Supply a read index and you get read data out the other side
  - Or write index some data, and a WE signal

*

10 Tuesday October 17th

10.1 Core Demonstration

10.1.1 VGA controller

No longer holds the dual port, now output's a memory address and inputs data memory.

10.1.2 Memory Controller

Has two sets of input and outputs for each port

10.1.3 Top Level

Initiatte VGA and Memory Controller and debouncer. We also want a way a way to use the seven segment display for debugging purposes

10.1.4 Core

Core will build the register file

```
module Core (input clk,
             output [14:0] core_to_memory_address,
             output [15:0] core_to_mem_data,
             output core_to_memory_write_enable,
             input [15:0] memory_to_core_data);
// Constants
parameter I_SETLO = 1'b0000;
parameter I_SETHI = 1'b0100;
parameter I_ADD = 1'b0010;
parameter I_LOADMEM = 1'b1000;
parameter I_STOREMEM = 1'b1001;
parameter I_BIZ = 1'b1010;
parameter I_BIM = 1'b1011;

parameter FETCH = 3'd0;
parameter DECODE = 3'd1;
parameter EXECUTE = 3'd2;
parameter LOAD1 = 3'd3;
parameter LOAD2 = 3'd4;
parameter STORE1 = 3'd5;
parameter BRANCH = 3'd6;

// Core state that persists between instructions
reg [14:0] pc = 15'h5400;  // Program Counter starts at instruction
reg        status_r;       // Zero register
reg        status_m;       // Negative register
```

```
reg [2:0]  core_state = FETCH;

// Core state that persists between cycles in 1 instructions
reg [15:0] data_1;
reg [15:0] data_2;
reg [3:0]  opcode;
reg [1:0]  dest_index;
reg [7:0]  immediate;
// Wires
wire [15:0] read_data_1;
wire [15:0] read_data_2;
reg  [1:0]  read_index_1; // Unlatched
reg  [1:0]  read_index_2; // Unlatched
reg  [1:0]  write_index;  // Unlatched
reg         write enable; // Unlatched
reg  [15:0] write_data;   // Unlatched

// Put register file into core
RegisterFile _RegisterFile (.clk (clk),
                            .read_index_1    (read_index_1),
                            .read_index_2    (read_index_2),
                            .read_data_1     (read_data_1),
                            .read_data_2     (read_data_2),
                            .write_index     (write_index),
                            .write_data      (write_data),
                            .write_enable    (write_enable));


// Combinatorial
always @ *
begin
    read_index_1 = 0;
    read_index_2 = 0;
    write_index = 0;
    write_data = 0;
    write_enable = 0;
    core_to_memory_address = 0;
    core_to_memory_data = 0;
    core_to_memory_write_enable = 0;

    case (core_state)
       FETCH :  begin core_to_memory_address = pc; end

       DECODE:  begin read_index_1 = memory_to_core_data[11:10];
                      read_index_2 = memory_to_core_data[9:8]; end
       EXECUTE: begin
```

```
                write_index = dest;
                write_enable = 1;
                if(opcode == I_ADD)
                    write_data = data_1 + data_2;
                else if(opcode == I_SETLO)
                    write_data = {data[15:8], immediate};
                else if(opcode == I_SETHI)
                    write_data = {immediate, data[7:0]};
                end

        LOAD1 :  begin core_to_memory_address = data_2;end

        LOAD2 :  begin
                 write_index = dest_index;
                 write_enable = 1;
                 wire_data =memory_to_core_data; end

        STORE1:  begin
                 core_to_memor_address = data_2;
                 core_to_memory_data = data_1;
                 core_to_memory_write_enable = 1;
                 end
    endcase

    case
end
// Sequential Logic
always @ posedge clk
begin
case (core_state)
       FETCH :  begin core_state <= DECODE;
                      pc <= pc + 1'b1; end

       DECODE:  begin
                      opcode =     memory_to_core_data[15:12]; // We want opcode now
                      immediate <=  memory_to_core_data[7:0];
                      dest_index <= memory_to_core_data[11:10];
                      data_1 <= read_data_1;
                      data_2 <= read_data_2;
                      if (opcode == I_SETLO ||  opcode == I_SETHI ||  opcode == I_SETLO ||
                           core_state <= EXECUTE;
                      else if (opcode == I_LOADMEM)
                           core_state <= LOAD1;
                      else if (opcode == I_STOREMEM)
                           core_state <= STORE1;
                      else if(opcode == I_BIM ||  opcode == I_BIZ)
```

```
                        core_state <= BRANCH;
                    end

        EXECUTE: begin
                 if(opcode == I_ADD)
                 begin
                    status_z <= write_data == 0;
                    status_m <= write_data[15];
                 end
                 core_state <= FETCH;  end

        LOAD1 :  begin  core_state <= LOAD2; end

        LOAD2 :  begin
                 status_z <= write_data == 0;
                 status_m <= write_data[15];
                 core_state <= FETCH; end

        STORE1:  begin core_state <= FETCH; end

        BRANCH:  begin
                 if (opcode == I_BIM && status_m)
                     pc <= pc + {7{immediate{7}}, immediate};
                 if (opcode == I_BIZ && status_z)
                     pc <= pc + {7{immediate{7}}, immediate};
                 core_state <= FETCH; end
end
```

10.1.5 Register File

```
module RegisterFile(input clk
                    input [1:0] read_index_1,
                    input [1:0] read_index_2,
                    output reg [15:0] read_data_1,
                    output reg [15:0] read_data_2,
                    input [1:0] write_index,
                    input [15:0] write_data,
                    input write_enable);
 reg [15:0] A = 0;
 reg [15:0] B = 0;
 reg [15:0] C = 0;
 reg [15:0] D = 0;
 // Register reads are combinatorial
 always @ *
 case (read_index_1)
    0: read_data_1 = A;
```

```
   1: read_data_1 = B;
   2: read_data_1 = C;
   3: read_data_1 = D;
endcase

always @ *
case (read_index_2)
   0: read_data_2 = A;
   1: read_data_2 = B;
   2: read_data_2 = C;
   3: read_data_2 = D;
endcase

// Register writes are sequential
always @ posedge clk
if(write_enable)
   case (write_index)
   0: A <= write_data;
   1: B <= write_data;
   2: C <= write_data;
   3: D <= write_data;
```