CSCE 611:
Execution and Write Back Stages:
I-Type Branch, R/J-Type JUMP Instructions

Instructor: Jason D. Bakos
Schedule

- Final exam:
  - Final exam slots:
    - Monday, Dec. 5 at 2:00-4:00
    - Monday, Dec. 5 at 4:00-6:00 (regular time)
  - An example exam and grading rubric are posted on Dropbox

- Lab 5: Schedule your demos ASAP!
- Lab 6: No demos, require standard testbench
  - Due Fri. 12/2 (end of next week)
  - I will be out of town next week except Monday

- Lab 7:
  - Must be submitted and demonstrated to Dr. Bakos by Monday, Dec. 12
  - Both group members must be present at demo
Branch Instructions

- Examples:
  - `beq $1, $2, loop`
  - `bgez $1, loop`

- Target address is encoded as a signed value indicating number of instructions to skip relative to next instruction

- New PC =
  - Program counter in FETCH + sign extended immediate of instruction in EX

- Resolve branch in EX, depends on ALU
Jump Instructions

• **R-type JUMP**
  - Example:
    - jr $31
  - Unconditional
  - New PC = Regs[RS]

• **J-type JUMP**
  - Example:
    - j exit
  - Unconditional
  - New PC = instruction_EX bits 9:0
## Three Stage Pipeline

<table>
<thead>
<tr>
<th>cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>F</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>F</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch (t)</td>
<td>F</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fall through</td>
<td>F</td>
<td>E(S)</td>
<td>W(S)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>target</td>
<td>F</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch(nt)</td>
<td>F</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
EX Stage: Branch/Jump

- New datapaths for EX stage:
  - **[NEW PC TARGET]**
    Need to compute `branch_addr`, `jtype_addr`, `reg_addr` and feed back to FETCH
  - **[BRANCH RESOLUTION]**
    - bgez: sign bit of R[rs]
    - beq, bne: perform subtract, use ALU's 'zero' output
  - **[LINK ADDRESSES]**
    For and-link branches and jumps:
    - Add registered version of PC_FETCH as possible register write value
      - Expand regsel_EX to allow this value to be written to the register file
    - Expand rdrt_EX to allow a hardcoded 31 to be chosen as the destination register
Fetch Stage

- Need to accept a branch/jump target from next stage for taken branches or jumps
Fetch Stage

- Need to add a stall for instruction following a taken branch or jump
Fetch Stage

- Add a separate bus for branch vs. j-type jumps vs. r-type jumps
Fetch Stage

- Need to provide PC to next stage to compute branch targets
EX Stage: New Signals for Branch/Jump

- In execute stage:
  - `branch_addr_EX[9:0]`: set to `PC_FETCH + instr[9:0]`
  - `jtype_addr_EX[9:0]`: set to `inst[9:0]`
  - `reg_addr_EX[9:0]`: set to `readdata1[9:0]`
  - `pcsrc_EX[1:0]`: set according to results from ALU

![Diagram](image)
EX Stage: New Signals for Branch/Jump

- **Instruction Execution (EX)**
  - `instruction_EX [31:26]` to Control Unit
  - `rd1[31]` from RegFile
  - `rdrt_EX` from Control Unit

- **Register File (RegFile)**
  - `instruction_EX [15:11]`
  - `PC_FETCH`
  - `instruction_EX [20:16]`

- **ALU**
  - `zero`

- **Write Back**
  - `PC_EX`
  - `regdest_WB`
WB Stage: New Signals for Branch/Jump
EX Stage: Branch/Jump

- Example control unit code for branch:

```plaintext
if (opcode == 6'b000001) begin // bgez
    alu_op = 4'b1100; // slt
    alu_shamt = 5'bX;
    enhilo = 1'b0;
    regsel = 3'bX;
    regwrite = 1'b0;
    alusrc = ??; // varies per design
    rdrt = 2'bX; // now 2 bits!
    if (readdata1_EX[31] == 0) begin
        pcsr = 2'b01; // take branch and invalidate instruction being fetched
        stall = 1;
    end
```
Lab 6: Branch/Jump Instructions

- Objectives:
  1. Implement MIPS processor that implements instructions: `beq, bne, bgez, j, jal, jr`
  2. Test with test program
  3. Write a MIPS test program:
     - Computes square root of a value on switches
     - Displays value on 7-segment LEDs
     - Implement on DE2
     - Value on switches is (18,0) value
     - Solution stored as (32,24) internally
     - Value displayed on HEXs is (8,5)
     - You can’t light up decimal point on HEXs (not connected on DE2 board)
Test Program

```
li $4,1
li $5,2
beq $4,$5,target1 # not taken

b:
  bne $4,$5,target2 # taken

b:
  jal target4
  j exit

target1:
  j b
target2:
  j c
target3:
  j d
target4:
  jr $31
exit:
```
Square Root

- Basic idea:
  - For a input value $S$
  - Write a problem that solves equation $x^2 - S = 0$
  - Simplest solution: use binary search
    1. Choose initial guess for $X = 0.0$
    2. Choose initial step size $step$ (solution must be within $step^2$)
    3. Repeat until step == 0:
      1. val = $x^2 - S$
      2. if val < 0 then $x = x+step$ else $x = x-step$
      3. step = step / 2

- Example: find $\sqrt{2}$

<table>
<thead>
<tr>
<th>X</th>
<th>val</th>
<th>step</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-2</td>
<td>2</td>
<td>add</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>.5</td>
<td>add</td>
</tr>
<tr>
<td>1.5</td>
<td>0.25</td>
<td>.25</td>
<td>sub</td>
</tr>
<tr>
<td>1.25</td>
<td>-0.4375</td>
<td>.125</td>
<td>add</td>
</tr>
<tr>
<td>1.375</td>
<td>-0.1093</td>
<td>.0625</td>
<td>add</td>
</tr>
<tr>
<td>1.4375</td>
<td>0.0664</td>
<td>.03125</td>
<td>sub</td>
</tr>
<tr>
<td>1.40625</td>
<td>-0.0224</td>
<td>0.01563</td>
<td>sub</td>
</tr>
</tbody>
</table>
...
Fractional Conversion

- Switches allow a value from 0 to \(2^{18}-1\)
  - \(0 \leq x \leq 262143\)
  - \(0 \leq x^{.5} < 512\)
- Assume our BCD display is (8,5)
  - Need 3 digits to left of decimal point
  - Gives precision of 1/100000 (least significant BCD value is \(10^{-5}\))
  - Roughly equivalent to \(2^{-14}\) \((2^{-14} = 6.1 \times 10^{-5})\)
- Binary representation: \((32,14)\) fixed-point
  - Shift initial step 14 bits to left
  - Initial guess = 0, initial step = 256.0
- Use same algorithm as in lab 4, but first:
  - Multiply value by \(10^5\) \((100000)\)
  - Shift 13 bits to the right
  - If bit 0 is set then add 2
  - Shift one bit to the right
Input and Output

- CPUs need a mechanism by which to communicate with external devices (peripherals)

- There are three mechanisms for this:
  - **Programmed I/O:** CPU and peripheral communicate through load and store instructions to special memory addresses
    - Generally used only for short control messages and status updates
  
  - **Direct Memory Access:** an external device is used to allow peripherals to directly access system memory, without CPU involvement
    - Used for bulk data -- disk drives, network interfaces, video cards

  - **Interrupts:** special signals that peripherals can send to the CPU in order to notify it that an I/O, error, or timer event has occurred
Programmed I/O

• Loads and stores to specially-mapped address ranges can be used to:
  – Read a word from a **status register**
    • Used to poll the state of a peripheral
  – Write a word to a **control register**
    • Used to send an “instruction” to a peripheral

• In this course, your CPU must interface with:
  – VGA video memory
CSELib VGA Interface

- We have designed a VGA interface for you

- Our VGA interface has the following characteristics:
  - Output VGA signal for 640x480 frame
  - Each pixel is 24 bit color
    - 8 bits each for RED, GREEN, and BLUE

- Nominally, this would require \( 640 \times 480 \times 3 = 921600 \) bytes

- We don’t have this much on-chip memory, so we will *downsample* the frame data:
  - The frame size will be 80 x 60
  - 8x8 on 640x480 field
Writing Control Registers

- Although modern computers use DMA for sending data to video memory, we will treat each location in video memory as a control register.

- TODO:
  - Expand data address bus from 10 bits to 16 bits
  - Connect CPU’s address output to both memories
  - Memory addresses 0xC000 to 0xF000 are mapped to video memory
  - Instantiate display interface into “My Computer” design
  - Video memory is 8K x 24 bits
  - External hardware used to mask off `mem_we` signal and enable video memory if store instruction falls into video memory range
    - Example:
      ```
      assign mem_we = mem_addr[15:14] != 2'b11 ? mem_we_cpu : 1'b0;
      assign video_we = mem_addr[15:14] == 2'b11 ? mem_we_cpu : 1'b0;
      ```
CSELib VGA Interface: display_IF

- The VGA interface outputs the current contents of the frame memory to the monitor

```verilog
module display_if (
    input clk,
    input rst,
    input [12:0] mem_waddr,
    input [23:0] mem_wdata,
    input mem_web,

    output [7:0] VGA_R,
    output [7:0] VGA_G,
    output [7:0] VGA_B,
    output VGA_SYNC,
    output VGA_BLANK,
    output VGA_CLK,
    output VGA_HS,
    output VGA_VS);
```

- The 8 outputs must be wired to output pins on the top level design, named according to their corresponding pin names
display_IF

- Add files to your project from /usr/local/3rdparty/csce611/vga_interface:
  - display_if.v: top-level interface, instance this in your design
  - Display.v: downsamples 640x480x30 to 80x60x15
  - VGA_PLL.v: creates 25 MHz clock (for 640x480) from 50 MHz reference clock
  - VGA_Sync.v: generates synchronization signals
  - vmem8192x24.v: video (pixel) memory

- There is also a sample top-level file to test your VGA connection:
  - video_test.v

- To use:
  - Wire up clk, rst
  - To draw a pixel:
    - Assert mem_web
    - mem_waddr(12 downto 7) is the **ROW NUMBER**
    - mem_waddr(6 downto 0) is the **COLUMN NUMBER**
    - mem_wdata(23 downto 16) is the **RED** intensity
    - mem_wdata(15 downto 8) is the **GREEN** intensity
    - mem_wdata(7 downto 0) is the **BLUE** intensity
Project Objective

• Write an assembly program that implements an “orbiting cursor”
  – One pixel on screen
  – Foreground and background color at your discretion
  – Pixel should constantly follow a path in a clockwise circle around the center of screen
  – User can adjust the radius of the orbit, not exceeding the maximum or minimum sizes allowed by the screen

• Controls:
  • Add KEY inputs to REG30 input:
    – KEY3: while pressed, decrease radius
    – KEY2: while pressed, increase radius
Outline of Assembly Program

• Basic idea:
  - Keep track of cursor x-position: pos_x
  - Sweep it between +/- radius
  - Calculate y-position for each x position on circle
  - (Would be better to calculate both positions as a function of angle, but we don’t want to implement trig functions)
  - Paint cursor on each iteration

• To slow the animation down:
  - Assign each pixel position a number of distance units, e.g. $2^{\text{GRIDSIZE}}$ units, per pixel
    - In other words (32,GRIDSIZE) fixed-point
Pseudocode

- Initialize registers:
  - pos_x = 0
  - direction_x = 1 (to the right)
  - radius = 30 << GRIDSIZE
  - center_x = 40 << GRIDSIZE
  - center_y = 30 << GRIDSIZE
Pseudocode

- Loop:
  - if pos_x<(-radius) or pos_x>radius then
    - direction_x = not direction_x
  - if direction_x then
    - new_pos_x = pos_x + 1
  - else
    - new_pos_x = pos_x - 1
  - new_pos_y = sqrt(radius^2-new_pos_x^2)
  - if direction_x==0 then new_pos_y = -new_pos_y
  - store background pixel to
    - (pos_x+center_x)>>GRIDSIZE, (pos_y+center_y)>>GRIDSIZE
  - store foreground pixel to
    - (new_pos_x+center_x)>>GRIDSIZE, (new_pos_y+center_y)>>GRIDSIZE
  - pos_x = new_pos_x
  - pos_y = new_pos_y
  - if KEY3 then increase radius
  - if KEY2 then decrease radius
Practical Considerations

- Square root input and output are (32,0) values

- $x^2$ must fit inside of 32 bits
  - limits $2^{GRIDSIZE}$ to $\sqrt{2^{32}}/80 = 819.2 \Rightarrow 512$

- Ball will move one x-position pixel after 512 iterations of loop

- Assume loop requires 100 cycles

- This means the ball will make a cycle (160 positions) in $160 \times 100 \times 512 / 50e6 \text{ s} = .16 \text{ s}$ at its max radius
Lab 7

- Design computer design with video interface
- Implement “orbit” code
- Due Dec. 12 (submission and demo)