Announcements

• Lab 2, Part B was due today
  – Go over answers today
• Lab 3, Part A due in 1 week
  – Create datapath of exec unit
    • Based on your ISA… anyone not have one?
  – Work with your partner on this
Tracking “progress”

• How to track “time” in CPU
  – Q: How can I tell when X cycles have passed since signal Y was asserted?
  – A: X flip-flops in a row

```
load_store_valid  load_store_valid_r  load_data_ready
```

“Test” testbench

• Is a fetch unit affected by load or store operations?
  – How do we tell?
“Branch” Testbench

- Note how ‘instruction_valid’ changes
- jump vs branch?

```
addr instruction
#0 nonbranch_op0
#1 nonbranch_op1
#2 conditional branch to #7 (predicted taken)
#3 nonbranch_op3
#4 nonbranch_op4
#5 nonbranch_op5
#6 nonbranch_op6
#7 branch to #0
```

“Custom” Testbench

- Simultaneously assert
  - ‘restart’ from 0x10
  - store ‘0xDEAD’ at 0x10
Implementation Result

- My results (from Xilinx 9.1):
  - 5.517ns Cycle Time
  - 181.26MHz
- Quick Poll:
  - Who got a better result?

The Critical Path
Before we move on to Lab 3…

• Any questions?

Back to the Future: 141L Edition
Creating a Core

- Core ties together fetch-unit and exec-unit
  - Instantiate one of each of these in the core module

```verilog
define core#(parameter D_WIDTH = 34, PA_WIDTH = 4)
  (input clk,
   input reset,
   // I/O interface
   output in_req,
   output out_req,
   output [PA_WIDTH-1 : 0] in_addr,
   output [PA_WIDTH-1 : 0] out_addr,
   input  [D_WIDTH-1 : 0] in_data,
   output [D_WIDTH-1 : 0] out_data,
   input  in_ack,
   input  out_ack )
```

Backend Components

- Register File
- Data Memory
  - Separate from instruction memory
- Misc. logic components
  - ALU
  - Sign extender
  - MUX
- Will be “leaf” modules to be instantiated in backend
Register File

- Read: combinational logic
- Write: sequential logic
- Easily implemented by using flip-flops
- Sample interface:

```vhdl
module regfile#(parameter SEL_WIDTH = 4, D_WIDTH = 34) {
    input clk,
    input we,
    input [SEL_WIDTH-1 : 0] read_sel,
    input [SEL_WIDTH-1 : 0] write_sel,
    input [D_WIDTH-1 : 0]   din,
    output [D_WIDTH-1 : 0]   dout
}
```

Data Memory

- Use ‘CORE Generator’
  - Optimized implementation
  - Initialize with your ‘*.coe’ file
  - 8K addresses, 34-bit words
  - Tutorial is available on class website

- Wrapper
  - Provides general interface
  - Read XOR Write
  - ‘refused’ signal?
    - Must try req again
The Backend Datapath

- we
- write_sel
- din
- read_sel
- regfile
- dout
- read_write_req
- write_en
- addr
- din
- Generated Ram Module
- Wrapper
- dout
- refused

I/O Interface

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>in_ack == 0</th>
<th>in_ack == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>in_req &lt;= 1</td>
<td></td>
<td>reg[rt] &lt;= din</td>
</tr>
<tr>
<td>in_addr &lt;= channel</td>
<td>...</td>
<td>‘in’ completes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>out</th>
<th>out_ack == 0</th>
<th>out_ack == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>out_req &lt;= 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>out_addr &lt;= channel</td>
<td>...</td>
<td>‘out’ completes</td>
</tr>
<tr>
<td>dout &lt;= data</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Single-, Multi-, or Pipelined?

• It’s up to you to decide
• Trade-offs (a 141 review):
  – Single:
  – Multi:
  – Pipelined:

Part A Deliverables

• “Leaf” Modules
  – Reg File, Data Mem, etc.
  – Implemented with RTL Verilog

• Execution Unit Datapath
  – Implemented with structural Verilog

• Datapath Schematic
  – Do NOT use Xilinx schematic generator
  – Explain how instructions go through the datapath
Questions?