ECE/CS 552: Data Path and Control
Instructor: Mikko H Lipasti
Fall 2010
University of Wisconsin-Madison
Lecture notes based on set created by Mark Hill.

Processor Implementation
- Forecast – heart of 552 – key to project
  - Sequential logic design review (brief)
  - Clock methodology (FSD)
  - Datapath – 1 CPI
    - Single instruction, 2’s complement, unsigned
    - Control
    - Multiple cycle implementation (information only)
    - Microprogramming
    - Exceptions

Review Sequential Logic
- Logic is combinational if output is solely function of inputs
  - E.g. ALU of previous lecture
- Logic is sequential or “has state” if output function of:
  - Past and current inputs
  - Past inputs remembered in “state”
  - Of course, no magic

- Clock high, Q = D, ~Q = ~D after prop. Delay
- Clock low Q, ~Q remain unchanged
  - Level-sensitive latch

Review Sequential Logic
- E.g. Master/Slave D flip-flop
  - While clock high, Q_M follows D, but Q_S holds
  - At falling edge Q_M propagates to Q_S

Review Sequential Logic
- Can build:
- Why can this fail for a latch?
Clocking Methodology

- **Motivation**
  - Design data and control without considering clock
  - Use Fully Synchronous Design (FSD)
    - Just a convention to simplify design process
    - Restricts design freedom
    - Eliminates complexity, can guarantee timing correctness
    - Not really feasible in real designs
    - Even in 554 you will violate FSD

Our Methodology

- Only flip-flops
- All on the same edge (e.g. falling)
- All with same clock
  - No need to draw clock signals
  - All logic finishes in one cycle

Our Methodology, cont’d

- No clock gating!
  - Book has bad examples
- Correct design:

Delayed Clocks (Gating)

- Problem:
  - Some flip-flops receive gated clock late
  - Data signal may violate setup & hold req’t

FSD Clocking Rules

- $T_{\text{clock}} = \text{cycle time}$
- $T_{\text{setup}} = \text{FF setup time requirement}$
- $T_{\text{hold}} = \text{FF hold time requirement}$
- $T_{\text{FF}} = \text{FF combinational delay}$
- $T_{\text{comb}} = \text{Combinational delay}$
- **FSD Rules:**
  - $T_{\text{clock}} > T_{\text{FF}} + T_{\text{comb max}} + T_{\text{setup}}$
  - $T_{\text{FF}} + T_{\text{comb min}} > T_{\text{hold}}$

Datapath – 1 CPI

- Assumption: get whole instruction done in one long cycle
- Instructions:
  - add, sub, and, or slt, lw, sw, & beq
- To do
  - For each instruction type
  - Putting it all together
Fetch Instructions
- Fetch instruction, then increment PC
  - Same for all types
- Assumes
  - PC updated every cycle
  - No branches or jumps
- After this instruction fetch next one

ALU Instructions
- and $1, $2, $3 #$1 <= $2 & $3
  - E.g. MIPS R-format
  - Opcode rs rt rd shamt function
    \[
    \begin{array}{cccccc}
    6 & 5 & 5 & 5 & 5 & 6
    \end{array}
    \]

Load/Store Instructions
- lw $1, immed($2) #$1 <= M[SE(immed)+$2]
  - E.g. MIPS I-format:
    \[
    \begin{array}{ccc}
    6 & 5 & 5 & 16
    \end{array}
    \]

Branch Instructions
- beq $1, $2, addr if ($1 -= $2) PC = PC + addr<<2
  - Actually
    \[
    \text{newPC} = PC + 4
    \]
    \[
    \text{target} = \text{newPC} + \text{addr} \ll 2 \text{ in MIPS offset from newPC}
    \]
    \[
    \text{if } ((\text{S1} - \text{S2}) = 0)
    \]
    \[
    \text{PC} = \text{target}
    \]
    \[
    \text{else}
    \]
    \[
    \text{PC} = \text{newPC}
    \]

Branch Instructions
- PC + 4 from instruction dispatch

All Together
Control Overview

- Single-cycle implementation
  - Datapath: combinational logic, I-mem, regs, D-mem, PC
  - Last three written at end of cycle
  - Need control – just combinational logic!
- Inputs:
  - Instruction (I-mem out)
  - Zero (for beq)
- Outputs:
  - Control lines for muxes
  - ALUop
  - Write-enables

**Control Overview**

- Fast control
  - Divide up work on “need to know” basis
  - Logic with fewer inputs is faster
- E.g.
  - Global control need not know which ALUop

**ALU Control**

- Assume ALU uses

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add</td>
<td>000000</td>
<td>100000</td>
</tr>
<tr>
<td>sub</td>
<td>sub</td>
<td>000000</td>
<td>100010</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>000000</td>
<td>100100</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td>000000</td>
<td>100101</td>
</tr>
<tr>
<td>slt</td>
<td>slt</td>
<td>000000</td>
<td>101010</td>
</tr>
</tbody>
</table>
- ALU-ctrl = f(opcode, function)

**But...don't forget**

- To simplify ALU-ctrl
  - ALUop = f(opcode)
  - 2 bits
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>add</td>
<td>10011</td>
<td>xx</td>
</tr>
<tr>
<td>sw</td>
<td>add</td>
<td>10101</td>
<td>xx</td>
</tr>
<tr>
<td>beq</td>
<td>sub</td>
<td>000010</td>
<td>10010</td>
</tr>
</tbody>
</table>

- 6 bits
**ALU Control**

<table>
<thead>
<tr>
<th>Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>add, sub, and, ...</td>
</tr>
<tr>
<td>00</td>
<td>lw, sw</td>
</tr>
<tr>
<td>01</td>
<td>beq</td>
</tr>
</tbody>
</table>

- ALU-ctrl = f(ALUop, function)
- 3 bits 2 bits 6 bits
- Requires only five gates plus inverters

**Control Signals Needed**

- ALU-ctrl = f(ALUop, function)
- 3 bits 2 bits 6 bits
- Requires only five gates plus inverters

**Global Control**

- **R-format**: opcode rs rt rd shamt function
- **I-format**: opcode rs rt address/immmediate
- **J-format**: opcode address

**Global Control**

- Route instruction[25:21] as read reg1 spec
- Route instruction[20:16] are read reg2 spec
- Route instruction[20:16] (load) and and instruction[15:11] (others) to
  - Write reg mux
- Call instruction[31:26] op[5:0]

**Global Control**

- Global control outputs
  - ALU-ctrl - see above
  - ALU src - R-format, beq vs. ld/st
  - MemRead - lw
  - MemWrite - sw
  - MemtoReg - lw
  - RegDst - lw dst in bits 20:16, not 15:11
  - RegWrite - all but beq and sw
  - PCSrc - beq taken

**Global Control**

- Global control outputs
  - Replace PCSrc with
    - Branch beq
    - PCSrc = Branch * Zero
  - What are the inputs needed to determine above global control signals?
    - Just Op[5:0]
### Global Control

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>RegDst</th>
<th>ALUSrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrr</td>
<td>000000</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>others</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

- `RegDst = ~Op[0]`
- `ALUSrc = Op[0]`

### PLA
- In AND-plane, & selected inputs to get minterms
- In OR-plane, | selected minterms to get outputs
- E.g.

### Control Signals; Add Jumps

### What’s wrong with single cycle?
- Critical path probably lw:
  - I-mem, reg-read, alu, d-mem, reg-write
- Other instructions faster
  - E.g. `rr`: skip d-mem
- Instruction variation much worse for full ISA and real implementation:
  - FP divide
  - Cache misses (what the heck is this? – later)
Single Cycle Implementation

- **Solution**
  - Variable clock?
    - Too hard to control, design
  - Fixed short clock
    - Variable cycles per instruction

Multi-cycle Implementation

- Clock cycle = max(i-mem,reg-read+reg-write, ALU, d-mem)
- Reuse combinational logic on different cycles
  - One memory
  - One ALU without other adders
- But
  - Control is more complex
  - Need new registers to save values (e.g. IR)
    - Used again on later cycles
    - Logic that computes signals is reused

High-level Multi-cycle Data-path

- **Note:**
  - Instruction register, memory data register
  - One memory with address bus
  - One ALU with ALUOut register

Comment on busses

- Share wires to reduce #signals
  - Distributed multiplexor
- Multiple sources driving one bus
  - Ensure only one is active!

Multi-cycle Control Signals

Multi-cycle Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Sample Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Fetch</td>
<td>IR=MEM[PC] PC=PC+4</td>
</tr>
<tr>
<td>ID</td>
<td>Decode</td>
<td>A=RF(IR[25:21]) B=RF(IR[20:16]) Target=PC+SE(IR[15:0])&lt;&lt;2</td>
</tr>
<tr>
<td>EX</td>
<td>Execute</td>
<td>ALUout = A + SE(IR[15:0]) if beq ALUout = A op B if mr if(A==B)PC = target if beq</td>
</tr>
<tr>
<td>Mem</td>
<td>Memory</td>
<td>MEM[ALUout] = B if sw MDR = MEM[ALUout] iflw RF(IR[15:11])= ALUout if mr</td>
</tr>
<tr>
<td>WB</td>
<td>Writeback</td>
<td>Reg(IR[20:16])= MDR if lw</td>
</tr>
</tbody>
</table>
Multi-cycle Control

- Function of Op[5:0] and current step
- Defined as Finite State Machine (FSM) or
  - Micro-program or microcode

Finite State Machine (FSM)

- For each state, define:
  - Control signals for datapath for this cycle
  - Control signals to determine next state
- All instructions start in same IF state
- Instructions terminate by making IF next
  - After proper PC update, of course

Multi-cycle Example

- Datapath (Fig. 5.33 from book)
  - Will walk and $1, $2, $3 through datapath
- Look at control FSM (Fig. 5.42 from book)
  - Will walk and $1, $2, $3 through FSM
- Will skip
  - Repeat for lw, sw, beq taken, beq n-t, j

Multi-cycle Example (and)

Nuts and Bolts—More on FSMs

- You will be using FSM control for parts of your processor implementation
- There are multiple methods for specifying a state machine
  - Moore machine (output is function of state only)
  - Mealy machine (output is function of state/input)
- There are different methods of assigning states
FSMs--State Assignment

- State assignment is converting logical states to binary representation
  - Use ECE 352 methods to build real state machine
- Is state assignment interesting/important?
  - Judicious choice of state representation can make next state fcn/output fcn have fewer gates
  - Optimal solution is hard, but having intuition is helpful (CAD tools can also help in practice)

State Assignment--Example

- 10 states in multicycle control FSM
  - Each state can have 1 of 16 \(2^{4}\) encodings with “dense” state representation
  - Any choice of encoding is fine functionally as long as all states are unique
- Appendix C-26 example: RegWrite signal

Summary

- Processor implementation
  - Datapath
  - Control
- Single cycle implementation
- Next: microprogramming
- Remaining slides for reference only