U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

Prof. Karu Sankaralingam

Single-Cycle Processor (5.1-5.4)

www.cs.wisc.edu/~karu/courses/cs552

Slides combined and enhanced by Karu Sankaralingam from work by Falsafi, Hill, Marculescu, Nagle, Patterson, Roth, Rutenbar, Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Outline

• Sequential logic & Clocking methodology
• Single-Cycle Datapath - 1 CPI
• Single-Cycle Control
• Defer to Later
  • Multiple cycle implementation
  • Microprogramming
  • Exceptions

Review Sequential Logic

• Logic is combinational if output is solely function of inputs
  - e.g., ALU
• Logic is sequential or "has state" if output is a function of
  - past and current inputs
  - past inputs "remembered" in "state"
  - but no magic!
Review: D Latch

- Clock high: Q = D, after some min to max propagate delay
- Clock low: Q, Q remain unchanged
- Sensitive to clock level

Review: D Flip-flop

- D flip-flop - built from 2 D-latches
  - while clock high, D flows into 1st latch, but not 2nd
  - in 2nd Q retains old value
- Remember D at falling edge & propagate thru 2nd latch

Review Latch vs. Flip-Flop

- Can build with flip-flops.
- Why does this fail for latch?
  - Q may rush via feedback path back to D before clock falls

D-FF WriteEnable (forbidden design)

- What if D-FF state unchanged on some clock cycles?
- Logically add "WriteEnable"
- Implementation could AND Clock & WriteEnable
  - Called "clock gating" or "qualifying the clock"
  - Forbidden (in this class)
- Real designs do clock gating but tricky due to glitches
D-FF WriteEnable (preferred design)

552 Clocking Methodology Goals

- Simplicity $\rightarrow$ Correctness
- Restricts freedom but eliminates errors
- Allows design datapath/control without thinking about clocks

552 Clocking Methodology Rules

- We provide D-FF design
- Use this D-FF for all processor state
- Same unqualified clock for all D-FFs
- Combinational logic must finish in one cycle

552 Clocking Methodology Implications

- Clock Becomes Implicit
  - same clock; same edge; no glitches
- You concentrate on getting logic correct
- Clock cycle time determined by worst-case sequential logic delay (plus a little)
- When you’re paid the big bucks, …
Processor Implementation

- Next: Single-Cycle Datapath

Outline

- Sequential logic design review
- Clocking methodology
- Datapath - 1 CPI
  - single instruction, 2's complement, unsigned
- Control
- Multiple cycle implementation
- Microprogramming
- Exceptions

Recap: Comb. Logic

- Adder
- ALU
- Mux
Recap: Storage

- Register
  - for PC
- Register file
  - 32 registers
  - 2 read ports/buses
  - 1 write port/bus
- Memory
  - 1 input bus
  - 1 output bus
  - Not bidirectional

Computer as State Machine

- Storage elements
  - Memory, Register file, PC
- Combinational elements
  - ALUs, Adders, Muxes

Processor Implementation

- Implementation determines
  - CPI
  - Cycle time
- Inst. Count = f(interface, compiler)
  - Interface = ISA, endianness etc.

Datapath - 1 CPI

- Assumption: Get one whole instruction done in one long cycle
  - fetch, decode/read operands, execute, memory, writeback
  - useful way to represent steps and identify required datapath elements: RTL
- For single instruction
- Put it together
Register Transfer Language

- RTL gives the meaning of the instructions
- All start by fetching the instruction

\[
\begin{align*}
\text{op} \, | \, \text{rs} \, | \, \text{rt} \, | \, \text{rd} \, | \, \text{shamt} \, | \, \text{funct} &= \text{MEM}[\text{PC}] \\
\text{op} \, | \, \text{rs} \, | \, \text{rt} \, | \, \text{Imm16} &= \text{MEM}[\text{PC}] \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU</td>
<td>R[rd] ← R[rs] + R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>SUBU</td>
<td>R[rd] ← R[rs] − R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>ORi</td>
<td>R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4</td>
</tr>
<tr>
<td>LOAD</td>
<td>R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4</td>
</tr>
<tr>
<td>STORE</td>
<td>MEM[R[rs] + sign_ext(Imm16)] ← R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>BEQ</td>
<td>if (R[rs] == R[rt]) then PC ← PC + sign_ext(Imm16)</td>
</tr>
</tbody>
</table>

A Simple Implementation

- ADD and SUB
  - addU rd, rs, rt
  - subU rd, rs, rt

- OR Immediate:
  - ori rt, rs, imm16

- LOAD and STORE Word
  - lw rt, rs, imm16
  - sw rt, rs, imm16

- BRANCH:
  - beq rs, rt, imm16

Fetch Instructions

- Fetch instruction, then update PC
- PC updated (at the end of) every cycle
  - What if no branches or jumps?

ALU Instructions

- R[rd] ← R[rs] op R[rt];  
  Example: addU rd, rs, rt
  - Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
  - ALUctr and RegWr: control logic after decoding the instruction

[Diagram of ALU operations]
Logical operation with Immediate

- \( R[rt] \leftarrow R[rs] \text{ op } \text{ZeroExt}[\text{imm}16] \)

Load Instruction

- \( R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[\text{imm}16]] \)
  - Example: \( \text{lw} \ rt, \text{imm}16(rs) \)

Store Instruction

- \( \text{Mem}[R[rs] + \text{SignExt}[\text{imm}16]] \leftarrow R[rt] \)
  - Example: \( \text{sw} \ rt, \text{imm}16(rs) \)

Conditional Branch Instruction

- \( \text{beq rs, rt, imm16} \)
  - \( \text{IR} = \text{Mem}[\text{PC}] \) // Fetch the instruction from memory
  - \( \text{Equal} \leftarrow R[rs] = R[rt] \) // Calculate the branch condition
  - if (COND eq 0) // Calculate the next instruction's address
    - \( PC \leftarrow PC + 4 + (\text{SignExt}(\text{imm}16) \times 4) \)
  - else
    - \( PC \leftarrow PC + 4 \)
Datapath for 'beq'

- beq rs, rt, imm16
  - Datapath generates condition (equal)

### Summary

- For a given instruction
  - Describe operation in RTL
  - Use ALUs, Registers, Memory, adders to achieve reqd. functionality
- To add instructions
  - Rinse and repeat
  - Reuse components by using muxes
- Controls : later
  - Selection controls for muxes
  - ALU controls for ALU ops
  - Register address controls
  - Write enables for registers/memory

### Cycletime

- What should the clock period be?
  - Enough to compute the next state values
    - Propagation clk-to-Q (new state)
    - Comb. Logic delay
    - Setup requirements
Timing: R-type inst

Clk

PC

Rs, Rt, Rd,

Op, Func

ALUsrc

RegWr

busA, B

busW

Old Value

New Value

Instruction Memory Access Time

Delay through Control Logic

Register File Access Time

ALU Delay

Register Write Occurs Here

Worst case timing

Clk

PC

Rs, Rt, Rd,

Op, Func

ALUsrc

ExtOp

MemReg

MemWrite

RegWrite

busA

busB

busW

Old Value

New Value

Instruction Memory Access Time

Delay through Control Logic

Register File Access Time

ALU Delay

Register Write Occurs Here

Single-cycle Datapath

Inst. Count

Cycle Time

CPI

• Performance Implications
  - Minimize all three
  - Insts/prog fixed -- f(interface, compiler)
  - CPI = 1 : As good as it gets (*)
  - Clock cycle time : high, "lw" critical path

“lw” Instruction

- Longer critical path
  - lower bound on cycletime

Critical Path (Load Operation) =
PC’s Clk-to-Q +
Instruction Memory’s Access Time +
Register File’s Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

lw Instruction

• Longer critical path
  - lower bound on cycletime

Ideal Instruction Memory

Instruction

Address

Data In

Data Memory

Ideal Instruction Memory

Next Address

Data Memory Access Time

Register Write Occurs Here

Data Memory Access Time
Processor Implementation

- Next: Control for Single-Cycle Datapath

Controls for Add Operation

- $R[rd] = R[rs] + R[rt]$

Meaning of Control Signals

- $rs, rt, rd$ and $imm16$ hardwired in datapath
- $nPC_{sel}$: $0 \Rightarrow PC + 4; 1 \Rightarrow PC + 4 + \text{SignExt}(Im16) \mid \mid 00$
Meaning of Control Signals

- **ExtOp**: "zero", "sign"
- **ALUsrc**: 0 => regB; 1 => immed
- **ALUctr**: "add", "sub", "or"
- **RegW**: write dest register
- **MemW**: write memory
- **MemtoReg**: 1 => Mem
- **RegDst**: 0 => "rt"; 1 => "rd"

OAI Controls: Worksheet

- **R[rt] <= R[rs] or ZeroExt[Imm16]**

OAI Controls: Solution

- **R[rt] <= R[rs] or ZeroExt[Imm16]**

LW Controls

- **R[rt] <= Data Memory (R[rs] + SignExt[Imm16])**
SW Controls: Worksheet

- \( R[r] \rightarrow \text{Data Memory } (R[rs] + \text{SignExt[imm16]}) \)

BEQ Controls

- if \( (R[rs] - R[rt] = 0) \) then \( \text{Zero} \ll 1 \); else \( \text{Zero} \ll 0 \)

Summary of Control Signals

<table>
<thead>
<tr>
<th>Inst</th>
<th>Register Transfer</th>
<th>ALU src</th>
<th>ALU ctr</th>
<th>RegDst</th>
<th>RegWr</th>
<th>MemtoReg</th>
<th>nPC sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD ( r[d] \leftarrow R[r] + R[r] )</td>
<td>( PC \leftarrow PC + 4 )</td>
<td>RegB, ALUctr = &quot;add&quot;, RegDst = rd, RegWr, nPCSel = &quot;4&quot;</td>
<td>( - )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>SUB ( r[d] \leftarrow R[r] - R[r] )</td>
<td>( PC \leftarrow PC + 4 )</td>
<td>RegB, ALUctr = &quot;sub&quot;, RegDst = rd, RegWr, nPCSel = &quot;4&quot;</td>
<td>( - )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ORI ( r[t] \leftarrow R[rs] + \text{zero_ext[imm16]} )</td>
<td>( PC \leftarrow PC + 4 )</td>
<td>ALUsrc = Im, ExtOp = &quot;Z&quot;, ALUctr = &quot;or&quot;, RegDst = rt, RegWr, nPCSel = &quot;4&quot;</td>
<td>( - )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>LOAD ( R[t] \leftarrow \text{MEM}[R[rs] + \text{sign Ext[imm16]}; ] )</td>
<td>( PC \leftarrow PC + 4 )</td>
<td>ALUsrc = Im, ExtOp = &quot;St&quot;, ALUctr = &quot;add&quot;, MemtoReg, RegDst = rt, RegWr, nPCSel = &quot;4&quot;</td>
<td>( - )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>STORE ( \text{MEM}[R[rs] + \text{sign Ext[imm16]}] \leftarrow R[r] )</td>
<td>( PC \leftarrow PC + 4 )</td>
<td>ALUsrc = Im, ExtOp = &quot;St&quot;, ALUctr = &quot;add&quot;, MemtoReg, nPCSel = &quot;4&quot;</td>
<td>( - )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>BEQ, if ( (R[rs] = R[rt]) ) then ( PC \leftarrow PC + \text{sign Ext[imm16]} )</td>
<td>( 00 ) else ( PC \leftarrow PC + 4 );</td>
<td>ALUsrc = RegB, nPCSel = &quot;Beq AND equal&quot;, ALUctr = &quot;sub&quot;</td>
<td>( - )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Control Logic

- Logic must generate appropriate signals for all instructions
- Summary slide (previous)
  - A way of representing the truth table
- First:
  - Equations in terms of opcodes
- Next
  - Equations in terms of instruction bits

Controls: Logic equations

- \( nPC_{sel} \) \( = \) if \((OP == BEQ)\) then \(EQUAL\) else 0
- \( ALU_{src} \) \( = \) if \((OP == \text{\textit{R-type}})\) then \(regB\) else if \((OP == BEQ)\) then \( \text{\textit{imm}}\)
- \( ALU_{ctr} \) \( = \) if \((OP == \text{\textit{R-type}})\) then \( \text{\textit{func}}\) else if \((OP == \text{\textit{ORi}})\) then \("\text{\textit{OR}}"\) else \("\text{\textit{add}}"\)
- \( \text{ExtOp} \) \( = \) if \((OP == \text{\textit{ORi}})\) then \("\text{\textit{zero}}"\) else \("\text{\textit{sign}}"\)
- \( \text{MemWrt} \) \( = \) \((\text{OP} == \text{\textit{Store}})\)
- \( \text{MemtoReg} \) \( = \) \((\text{OP} == \text{\textit{Load}})\)
- \( \text{RegWrt} \) \( = \) if \((\text{OP} == \text{\textit{Store}})\) or \((\text{OP} == \text{BEQ})\) then 0 else 1
- \( \text{RegDst} \) \( = \) if \((\text{OP} == \text{\textit{Load}})\) or \((\text{OP} == \text{\textit{ORi}})\) then 0 else 1

Truth Table summary

<table>
<thead>
<tr>
<th></th>
<th>Add</th>
<th>Sub</th>
<th>Ori</th>
<th>Bv</th>
<th>Sw</th>
<th>Beq</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>0011</td>
<td>10011</td>
<td>10101</td>
<td>100000</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ori</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bv</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Appendix A

Op Code

- \text{\textit{add}}, \text{\textit{sub}}, \text{\textit{ori}}, \text{\textit{lw}}, \text{\textit{sw}}, \text{\textit{beq}}, \text{\textit{jump}}
- We Don't Care:
  - Funct: 00 0000
  - ALU src: 00 0000
  - ALU dest: 00 0010
  - ALU op: 10 0000
  - ALU src: 10 0000
  - ALU dest: 10 0010
  - ALU op: 10 0000
  - Operands: 00 0000
  - Immediate: 10 0000
  - Target address: 00 0000
  - Jump: 00 0000

Appendix B

Truth Table summary

<table>
<thead>
<tr>
<th></th>
<th>Add</th>
<th>Sub</th>
<th>Ori</th>
<th>Bv</th>
<th>Sw</th>
<th>Beq</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>0011</td>
<td>10011</td>
<td>10101</td>
<td>100000</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ori</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bv</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Local vs Global Control

- One more layer of abstraction
  - ALUctr
    - if (OP == "R-type") then funct    - else if (OP == OR) then "OR"    - else if (OP == BEQ) then "sub"    - else "add"

### Global Control: Truth Table

<table>
<thead>
<tr>
<th>R-type</th>
<th>op</th>
<th>Rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>xxx</td>
</tr>
</tbody>
</table>

**Encoding**

- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)
Truth Table for RegWrite

<table>
<thead>
<tr>
<th>op</th>
<th>000000</th>
<th>000001</th>
<th>000100</th>
<th>000101</th>
<th>001000</th>
<th>001001</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- RegWrite = R-type + ori + lw
  = \text{l}	ext{op}5\text{op}4\text{op}3\text{op}2\text{op}1\text{op}0 (R-type)
  + \text{l}	ext{op}5\text{op}4\text{op}3\text{op}2\text{op}1\text{op}0 (ori)
  + \text{l}	ext{op}5\text{op}4\text{op}3\text{op}2\text{op}1\text{op}0 (lw)

![Truth Table for RegWrite](image)

PLA implementation

![PLA implementation](image)

Putting it all together

![Putting it all together](image)

Outline

- Sequential logic & Clocking methodology
- Single-Cycle Datapath - 1 CPI
- Single-Cycle Control
- Defer to Later
  - Multiple cycle implementation
  - Microprogramming
  - Exceptions