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
Processor Implementation
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 a 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

Implementation:

- 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)**

![Diagram of D-FF WriteEnable](image)

1. MUX
2. D-FF
3. Clock
4. Write enable
5. Output
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
You will design this!!

Inst Memory

4

Adder

Adder

Mux

PC Ext

imm16

16

32

ExtOp

ALUSrc

nPC_sel

RegDst

Rd
Rt

Equal

ALUctr
MemWr
MemtoReg

ALUctr

MemWr

MemtoReg
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

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

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

```plaintext
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>instruction word</td>
</tr>
</tbody>
</table>

Next Address Logic

Next Address Logic

Adder

PC

Instruction Word

Clk

4
ALU Instructions

- \( R[rd] \leftarrow R[rs] \text{ op } R[rt] \)  
  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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

32 32-bit Registers

ALUctr

RegWr

busW

32

Clk

Rw Ra Rb

32

busA

busB

32

Result

32
Logical operation with Immediate

- \( R[rt] \leftarrow R[rs] \text{ op } \text{ZeroExt}[imm16] \)

### ALU Control

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
<th>rd?</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>

### ALU Source

<table>
<thead>
<tr>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000000</td>
</tr>
</tbody>
</table>

### Registers

- 32 32-bit Registers
- \( Rw, Ra, Rb \)
- \( Rd, Rt \)
- \( Rs \)
- \( RegWr, RegDst \)
- \( Clk, busW, busA, busB \)
Load Instruction

- \( R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]] \)

Example: \( \text{lw} \ rt, \text{imm16}(rs) \)
Store Instruction

- $\text{Mem}[R[rs] + \text{SignExt}[\text{imm16}] \leftarrow R[rt]]$

Example: $\text{sw } rt, \text{imm16}(rs)$
Conditional Branch Instruction

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

What is this?
Datapath for 'beq'

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

\[\text{beq} \quad \text{rs, rt, imm16}\]

- Datapath generates condition (equal)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

**Diagram:**

- **PC Ext:**
  - Imm16
  - PC
  - clk

- **Adder:**
  - nPC_sel
  - 4

- **Mux:**
  - 32

- **Equal?**
  - Cond
  - Eq
  - Rw, Ra, Rb
  - busA, busB
  - 32 32-bit Registers
  - 5, 5, 5

- **RegWr:**
  - 5

- **busW:**
  - Clk
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**: Clock signal
- **PC**: Program Counter
  - Instruction Memory Access Time
- **Rs, Rt, Rd, Op, Func**: Instruction fields
  - Instruction Memory Access Time
  - Delay through Control Logic
- **ALUctr**: ALU Control
  - Instruction Memory Access Time
- **RegWr**: Register Write
  - Register File Access Time
  - ALU Delay
- **busA, B**: Bus signals
  - Register File Access Time
- **busW**: Bus signal
  - Register File Access Time

**32 32-bit Registers**

- **Rw, Ra, Rb**: Register signals
- **busA, busB**: Bus signals
- **ALU**: Arithmetic Logic Unit
  - Result: 32-bit output

Register Write Occurs Here
“lw” Instruction

- Longer critical path
- lower bound on cycle time

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
Worst case timing

Clk

PC

Rs, Rt, Rd, Op, Func

ALUctr

ExtOp

ALUSrc

MemtoReg

RegWr

busA

busB

Address

busW

Instruction Memory Access Time

Delay through Control Logic

Register Write Occurs

Register File Access Time

Delay through Extender & Mux

ALU Delay

Data Memory Access Time
Single-cycle Datapath

- Performance Implications
  - Minimize all three
  - Insts/prog fixed -- f(interface, compiler)
  - \( \text{CPI} = 1 \) : As good as it gets (*)
  - Clock cycle time: high, “lw” critical path
Processor Implementation

- Next: Control for Single-Cycle Datapath
Control for Datapath

Instruction<31:0>

Inst Memory Adr

Op Fun Rt Rs Rd Imm16

Control

nPC_sel RegWr RegDst ExtOp ALUSrc ALUctr MemWr MemtoReg Equal

DATA PATH
Controls for Add Operation

• \( R[rd] = R[rs] + R(rt) \)
Meaning of Control Signals

- \( rs, rt, rd \) and \( \text{imm16} \) hardwired in datapath
- \( n\text{PC}_\text{sel} \):
  - \( 0 \Rightarrow \text{PC} \leftarrow \text{PC} + 4 \)
  - \( 1 \Rightarrow \text{PC} \leftarrow \text{PC} + 4 + \text{SignExt(Im16)} || 00 \)
Meaning of Control Signals

- **ExtOp**: “zero”, “sign”
- **ALUsrc**: 0 => regB; 1 => imm
- **ALUctr**: “add”, “sub”, “or”

- **MemWr**: write memory
- **MemtoReg**: 1 => Mem
- **RegDst**: 0 => “rt”; 1 => “rd”
- **RegWr**: write dest register

![Control Signal Diagram]
ORI Controls: Worksheet

- \( R[rt] \leftarrow R[rs] \) or ZeroExt[Imm16]
**ORI Controls: Solution**

- R[rt] ← R[rs] or ZeroExt[Imm16]

![Diagram of ORI Control System]

- **RegDst = 0**
- **RegWr = 1**
- **ALUctr = Or**
- **ALUSrc = 1**
- **ExtOp = 0**
- **MemtoReg = 0**
- **nPC_sel = +4**

---

**Instructions:**

- **Instruction<31:0>**
- **Rt**
- **Rs**
- **Rd**
- **Imm16**
- **MemWr = 0**
- **Zero**
- **WrEn**
- **Data In**
- **Data Memory**
- **Addr**
LW Controls

- $R[rt] \leftarrow \text{Data Memory \{R[rs] + \text{SignExt[imm16]}\}}$

**Instruction Fetch Unit**

- $nPC_{\text{sel}} = +4$
- ALUctr = Add
- ALUSrc = 1

**Data Memory**

- MemWr = 0
- MemtoReg = 1

**Data Adr**

- Zero
- Instruction<31:0>

**ALU**

- WrEn
- Data In

**Register File**

- 32 32-bit Registers
- RegWr = 1
- RegDst = 0

**Mux**

- 32 32-bit Registers
- busW
- busA
- busB

**Extender**

- imm16
- ExtOp = 1
SW Controls: Worksheet

- \( R[rt] \rightarrow \text{Data Memory} \{ R[rs] + \text{SignExt}[\text{imm16}] \} \)
SW Controls: Worksheet

- \( R[rt] \leftarrow \text{Data Memory} \{R[rs] + \text{SignExt}[\text{imm16}]\} \)
BEQ Controls

- if (R[rs] - R[rt] == 0) then Zero ← 1; else Zero ← 0
Summary of Control Signals

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfer</th>
</tr>
</thead>
</table>
| ADD  | \[R_{rd} \gets R_{rs} + R_{rt};\] \[PC \gets PC + 4\] \\
|      | \[ALU_{src} = \text{RegB}, \ ALU_{ctr} = \text{"add"}, \ RegDst = rd, \ RegWr, \ nPC_{sel} = \text{"+4"}\] |
| SUB  | \[R_{rd} \gets R_{rs} - R_{rt};\] \[PC \gets PC + 4\] \\
|      | \[ALU_{src} = \text{RegB}, \ ALU_{ctr} = \text{"sub"}, \ RegDst = rd, \ RegWr, \ nPC_{sel} = \text{"+4"}\] |
| ORi  | \[R_{rt} \gets R_{rs} + \text{zero\_ext(Imm16)};\] \[PC \gets PC + 4\] \\
|      | \[ALU_{src} = \text{Im}, \ Extop = \text{"Z"}, \ ALU_{ctr} = \text{"or"}, \ RegDst = rt, \ RegWr, \ nPC_{sel} = \text{"+4"}\] |
| LOAD | \[R_{rt} \gets \text{MEM}[ R_{rs} + \text{sign\_ext(Imm16)}];\] \[PC \gets PC + 4\] \\
|      | \[ALU_{src} = \text{Im}, \ Extop = \text{"Sn"}, \ ALU_{ctr} = \text{"add"}, \ MemtoReg, \ RegDst = rt, \ RegWr, \ nPC_{sel} = \text{"+4"}\] |
| STORE| \[\text{MEM}[ R_{rs} + \text{sign\_ext(Imm16)\} \gets R_{rs};\] \[PC \gets PC + 4\] \\
|      | \[ALU_{src} = \text{Im}, \ Extop = \text{"Sn"}, \ ALU_{ctr} = \text{"add"}, \ MemWr, \ nPC_{sel} = \text{"+4"}\] |
| BEQ  | if \( R_{rs} == R_{rt} \) then \( PC \gets PC + \text{sign\_ext(Imm16)\} \) else \( PC \gets PC + 4; \) \\
|      | \[ALU_{src} = \text{RegB}, \ nPC_{sel} = \text{"Beq AND equal"}, \ ALU_{ctr} = \text{"sub"}\] |
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

- \( \text{nPC\_sel} \) <= if \((\text{OP} == \text{BEQ})\) then EQUAL else 0
- \( \text{ALU\_src} \) <= if \((\text{OP} == \text{"R-type"})\) then \(\text{regB}\) 
  elseif \((\text{OP} == \text{BEQ})\) then \(\text{regB}\), else \(\text{imm}\)
- \( \text{ALU\_ctr} \) <= if \((\text{OP} == \text{"R-type"})\) then \(\text{funct}\) 
  elseif \((\text{OP} == \text{ORi})\) then \(\text{OR}\) 
  elseif \((\text{OP} == \text{BEQ})\) then \(\text{sub}\) 
  else \(\text{add}\)
- \( \text{ExtOp} \) <= ____________
- \( \text{Mem\_Wr} \) <= ____________
- \( \text{Mem\_to\_Reg} \) <= ____________
- \( \text{Reg\_Wr:} \) <= ____________
- \( \text{Reg\_Dst:} \) <= ____________
Controls: Logic equations

- \( nPC\_sel \) <= if (OP == BEQ) then EQUAL else 0
- \( ALU\_src \) <= if (OP == “R-type”) then “regB”
  elseif (OP == BEQ) then regB, else “imm”
- \( ALU\_ctr \) <= if (OP == “R-type”) then \( funct \)
  elseif (OP == ORi) then “OR”
  elseif (OP == BEQ) then “sub”
  else “add”
- \( Ext\_Op \) <= if (OP == ORi) then “zero” else “sign”
- \( Mem\_Wr \) <= (OP == Store)
- \( Mem\_to\_Reg \) <= (OP == Load)
- \( Reg\_Wr: \) <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- \( Reg\_Dst: \) <= if ((OP == Load) || (OP == ORi)) then 0 else 1
# Truth Table Summary

## Table

<table>
<thead>
<tr>
<th>func</th>
<th>10 0000</th>
<th>10 0010</th>
<th>We Don’t Care :-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>00 0000</td>
<td>00 0000</td>
<td>00 1110 10 0011</td>
</tr>
<tr>
<td></td>
<td>10 1011</td>
<td>00 0100</td>
<td>00 0010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RegDst</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nPCsel</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUctr&lt;2:0&gt;</td>
<td>Add</td>
<td>Subtract</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
</tbody>
</table>

## Instruction Types

- **R-type**
  - op
  - rs
  - rt
  - rd
  - shamt
  - funct
  - add, sub

- **I-type**
  - op
  - rs
  - rt
  - immediate
  - ori, lw, sw, beq

- **J-type**
  - op
  - target address
  - (52)
  - jump
  - Sankaralingam
Local vs Global Control

• One more layer of abstraction
  
  • ALUctr <= if (OP == “R-type”) then funct
    elseif (OP == ORi) then “OR”
    elseif (OP == BEQ) then “sub”
    else “add”

<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></td>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td>6</td>
</tr>
</tbody>
</table>
Global Control: Truth Table

<table>
<thead>
<tr>
<th></th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ori</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>sw</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jump</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RegDsr</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUop&lt;N:0&gt;</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
</tbody>
</table>

CS/ECE 552

(54)
In this exercise, ALUop has to be 2 bits wide to represent:
- (1) “R-type” instructions
- “I-type” instructions that require the ALU to perform:
  - (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
- “I-type” instructions that require the ALU to perform:
  - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

<table>
<thead>
<tr>
<th>ALUop (Symbolic)</th>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
<tr>
<td></td>
<td>1 0 0</td>
<td>0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 1</td>
<td>xxx</td>
</tr>
</tbody>
</table>
### Global Control: Truth Table

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R-type</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Control</td>
<td>ALU Control (Local)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUctr</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Sankaralingam</td>
</tr>
</tbody>
</table>

CS/ECE 552 (56)
Truth Table for RegWrite

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
<td></td>
</tr>
<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**
- $= \neg op<5> \& \neg op<4> \& \neg op<3> \& \neg op<2> \& \neg op<1> \& \neg op<0> \ (R\text{-type})$
  + $\neg op<5> \& \neg op<4> \& op<3> \& op<2> \& \neg op<1> \& op<0> \ (ori)$
  + $op<5> \& \neg op<4> \& \neg op<3> \& \neg op<2> \& op<1> \& op<0> \ (lw)$
PLA implementation

R-type  ori  lw  sw  beq  jump

op<5>  op<5>  op<5>  op<5>  op<5>  op<5>

<0>  <0>  <0>  <0>  <0>  <0>

RegWrite  ALUSrc  RegDst  MemtoReg  MemWrite  Branch  Jump  ExtOp  ALUop<2>  ALUop<1>  ALUop<0>

CS/ECE 552

(58)
Putting it all together

Main Control

ALUct

Instruction Fetch Unit

MemtoReg

ALU

Data Memory

ExtOp

Extender

32 32-bit Registers

aluop

RegDst

ALUSrc

Instr<31:26>

Instr<5:0>

Instruction<31:0>

nPC_sel

Rt

Rs

Rd

Imm16

MemWr

Zero

WrEn

Addr

Data In

MemWr

32

16

5

5

5

32

32

1

32

32
Outline

• Sequential logic & Clocking methodology

• Single-Cycle Datapath - 1 CPI

• Single-Cycle Control

• Defer to Later
  • Multiple cycle implementation
  • Microprogramming
  • Exceptions