CS 61C: Great Ideas in Computer Architecture (Machine Structures)  
Lecture 30: Single-Cycle CPU  
Datapath Control Part 2  

Instructor: Dan Garcia  
http://inst.eecs.berkeley.edu/~cs61c
In a refreshing NY Times Op-Ed, Gary Marcus and Ernest Davis tell us that Big Data is useful, but often overhyped as the panacea. Great as a tool, but keep in perspective!
Review: Processor Design 5 steps

Step 1: Analyze instruction set to determine datapath requirements
– Meaning of each instruction is given by register transfers
– Datapath must include storage element for ISA registers
– Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic
Processor Design: 5 steps

Step 1: Analyze instruction set to determine datapath requirements
– Meaning of each instruction is given by register transfers
– Datapath must include storage element for ISA registers
– Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic
Register-Register Timing:
One Complete Cycle (Add/Sub)

Clk

PC

Rs, Rt, Rd,

Op, Func

ALUctr

RegWr

busA, B

busW

RegWr

Rd

Rs

Rt

RegFile

clk

busW

ALUctr

busA

32

busB

32

ALU

32

Register Write Occurs Here
Register-Register Timing:
One Complete Cycle

- **Clk**: One Complete Cycle
- **PC**: Instruction Memory Access Time
- **Rs, Rt, Rd, Op, Func**: Delay through Control Logic
- **ALUctr**: Register File Access Time
- **RegWr**: ALU Delay
- **busA, B**: Register Write Occurs Here

The diagram illustrates the timing of operations in a register-register architecture, including access to memory, control logic, register file, ALU, and register write operations. The diagram shows the flow of data and the timing delays associated with each operation.
3c: Logical Op (or) with Immediate

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

Writing to Rt register (not Rd)!!

What about Rt Read?
3d: Load Operations

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

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

```plaintext
<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>
```

```
RegDst
Rd  Rt
1   0

RegWr
Rs  Rt
5   5

RegFile
Rw  Ra  Rb

ZeroExt

imm16

ALUSrc
```

```
ALU
32

busA
32

busB

ZeroExt

imm16

16

RegFile

clk

32
```
3d: Load Operations

- \( R[rt] = Mem[R[rs] + SignExt[imm16]] \)

Example: \( lw \ rt, rs, imm16 \)
3e: Store Operations


Ex.: `sw rt, rs, imm16`

<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 showing the flow of data with labels for `RegDst`, `ALUctrl`, and `MemtoReg`.
3e: Store Operations

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

Example: \( \text{sw } rt, rs, \text{imm16} \)

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

Diagram showing the flow of data and control signals in a pipeline stage for store operations.
3f: The Branch Instruction

```

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>
```

`beq rs, rt, imm16`
- `mem[PC]` Fetch the instruction from memory
- `Equal = R[rs] == R[rt]` Calculate branch condition
- if `(Equal)` Calculate the next instruction’s address
  - `PC = PC + 4 + (SignExt(imm16) x 4)`
else
  - `PC = PC + 4`
Datapath for Branch Operations

\[
\text{beq } rs, rt, \text{imm}16
\]

Datapath generates condition (Equal)

Already have mux, adder, need special sign extender for PC, need equal compare (sub?)
Instructions Fetch Unit including Branch

- if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4

How to encode nPC_sel?
- Direct MUX select?
- Branch inst. / not branch inst.

Let’s pick 2nd option

Q: What logic gate?

<table>
<thead>
<tr>
<th>nPC_sel</th>
<th>zero?</th>
<th>MUX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Putting it All Together: A Single Cycle Datapath

Inst Memory

Adr

\[ \begin{align*}
\text{Rs} & \rightarrow \text{Rt} \\
\text{Rd} & \rightarrow \text{Imm16}
\end{align*} \]

RegDst

\[ \begin{align*}
\text{Rd} & \rightarrow \text{Rt} \\
\text{Equal}
\end{align*} \]

RegWr

\[ \begin{align*}
\text{Rs} & \rightarrow \text{Rt} \\
\text{Rw} & \rightarrow \text{Ra} \\
\text{Rb}
\end{align*} \]

ALUctr

MemtoReg

\[ \begin{align*}
\text{MemWr}
\end{align*} \]

Mem

\[ \begin{align*}
\text{Data In}
\end{align*} \]

ExtOp

ALUSrc

\[ \begin{align*}
\text{Extender}
\end{align*} \]

32

\[ \begin{align*}
\text{ALU}
\end{align*} \]

32

\[ \begin{align*}
\text{Data Memory}
\end{align*} \]

32

\[ \begin{align*}
\text{WrEn} \\
\text{Addr}
\end{align*} \]

clk

imm16

nPC_sel

4

Adder

Adder

Mux

PC Ext

4

RegFile

busW

Data In

eq

Equal

\[ \begin{align*}
\text{busA} & \rightarrow 32 \\
\text{busB} & \rightarrow 32
\end{align*} \]

32

\[ \begin{align*}
\text{MemMemory}
\end{align*} \]

0

1

1

1
Datapath Control Signals

- **ExtOp**: “zero”, “sign”
- **ALUsrc**: 0 ⇒ regB; 1 ⇒ immed
- **ALUctr**: “ADD”, “SUB”, “OR”
- **MemWr**: 1 ⇒ write memory
- **MemtoReg**: 0 ⇒ ALU; 1 ⇒ Mem
- **RegDst**: 0 ⇒ “rt”; 1 ⇒ “rd”
- **RegWr**: 1 ⇒ write register
Given Datapath: RTL $\rightarrow$ Control
RTL: The Add 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>
</tbody>
</table>

- **add rd, rs, rt**
  - **MEM[PC]** Fetch the instruction from memory
  - **R[rd] = R[rs] + R[rt]** The actual operation
  - **PC = PC + 4** Calculate the next instruction’s address
Instruction Fetch Unit at the Beginning of Add

- Fetch the instruction from Instruction memory: Instruction = MEM[PC]
  - same for all instructions
Single Cycle Datapath during Add

\[ R[rd] = R[rs] + R[rt] \]
Instruction Fetch Unit at End of Add

- **PC = PC + 4**
  - Same for all instructions except: Branch and Jump

![Diagram of Instruction Fetch Unit]

- **Inst Memory**
- **Inst Address**
  - Mux
  - Adder
  - **clk**
  - **nPC sel=+4**
  - **PC Ext**
  - **imm16**
P&H Figure 4.17
### Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfer</th>
</tr>
</thead>
</table>
| **add** | R[rd] ← R[rs] + R[rt]; PC ← PC + 4  
ALUsrc=RegB, ALUctr="ADD", RegDst=rd, RegWr, nPC_sel="+4" |
| **sub** | R[rd] ← R[rs] − R[rt]; PC ← PC + 4  
ALUsrc=RegB, ALUctr="SUB", RegDst=rd, RegWr, nPC_sel="+4" |
| **ori** | R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4  
ALUsrc=Im, Extop="Z", ALUctr="OR", RegDst=rt, RegWr, nPC_sel="+4" |
| **lw** | R[rt] ← MEM[ R[rs] + sign_ext(Imm16)]; PC ← PC + 4  
ALUsrc=Im, Extop="sn", ALUctr="ADD", MemtoReg, RegDst=rt, RegWr, nPC_sel = “+4” |
| **sw** | MEM[ R[rs] + sign_ext(Imm16)] ← R[rs]; PC ← PC + 4  
ALUsrc=Im, Extop="sn", ALUctr = “ADD”, MemWr, nPC_sel = “+4” |
| **beq** | if (R[rs] == R[rt]) then PC ← PC + sign_ext(Imm16) || 00  
else PC ← PC + 4  
nPC_sel = “br”, ALUctr = “SUB” |
## Summary of the Control Signals (2/2)

<table>
<thead>
<tr>
<th>See Appendix A</th>
<th>func</th>
<th>op</th>
<th>10 0000</th>
<th>10 0010</th>
<th>We Don’t Care :-)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>00 0000</td>
<td>00 0000</td>
<td>00 1101</td>
</tr>
<tr>
<td>RegDst</td>
<td>add</td>
<td>sub</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nPCsel</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>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</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>
</tr>
</tbody>
</table>

### Instruction Formats

<table>
<thead>
<tr>
<th>Field</th>
<th>R-type</th>
<th>I-type</th>
<th>J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>31 26 21 16</td>
<td>11 6 0</td>
<td>op</td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shamt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>target address</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **R-type**: add, sub
- **I-type**: ori, lw, sw, beq
- **J-type**: jump
Boolean Expressions for Controller

RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPcSel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq  (assume ALUctr is 00 ADD, 01 SUB, 10 OR)
ALUctr[1] = or

Where:

rtype = ~op5 • ~op4 • ~op3 • ~op2 • ~op1 • ~op0,
or = ~op5 • ~op4 • op3 • op2 • ~op1 • op0
lw = op5 • ~op4 • ~op3 • ~op2 • op1 • op0
sw = op5 • ~op4 • ~op3 • ~op2 • op1 • ~op0
beq = ~op5 • ~op4 • ~op3 • op2 • ~op1 • ~op0
jump = ~op5 • ~op4 • ~op3 • ~op2 • op1 • ~op0

add = rtype • func5 • ~func4 • ~func3 • ~func2 • ~func1 • ~func0
sub = rtype • func5 • ~func4 • ~func3 • ~func2 • func1 • ~func0

How do we implement this in gates?
Controller Implementation

```
Controller
  | Implementa
  | Pon
  | add
  | sub
  | ori
  | lw
  | sw
  | beq
  | jump

RegDst
ALUSrc
MemtoReg
RegWrite
MemWrite
nPcsel
Jump
ExtOp
ALUctr[0]
ALUctr[1]
```

```
“AND” logic
  | opcode
  | func

“OR” logic
```
1) We should use the main ALU to compute PC=PC+4 in order to save some gates.

2) The ALU is inactive for memory reads (loads) or writes (stores).
Clicker Survey for CS Retreat

If we add more faculty, what should we do for upper-division courses?

a) We should have more sections of the same courses, so lecture is smaller

b) We should have more semester-long courses

c) We should have more half-semester-long courses
Summary: Single-cycle Processor

• Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
     • Formulate Logic Equations
     • Design Circuits
Bonus Slides

• How to implement Jump
Single Cycle Datapath during Jump

- New PC = \{ PC[31..28], target address, 00 \}
Single Cycle Datapath during Jump

- New PC = { PC[31..28], target address, 00 }
**Instruction Fetch Unit at the End of Jump**

- New PC = \{ PC[31..28], target address, 00 \}

How do we modify this to account for jumps?
**Instruction Fetch Unit at the End of Jump**

- New PC = \{ PC[31..28], target address, 00 \}

**Query**
- Can Zero still get asserted?
- Does nPC_sel need to be 0?
  - If not, what?