Datapath Control Signals

- **ExtOp**: “zero”, “sign”
- **ALUsrc**: 0 → regB; 1 → immed
- **ALUctr**: “ADD”, “SUB”, “OR”
- **MemWr**: 1 ⇒ write memory
- **MemtoReg**: 0 ⇒ ALU; 1 ⇒ Mem
- **nPC_sel**: 0 ⇒ “+4”; 1 ⇒ “br”
- **RegDst**: 0 ⇒ “rt”; 1 ⇒ “rd”
- **RegWr**: 1 ⇒ write register
Where Do Control Signals Come From?

![Diagram](image-url)

**DATA PATH**

- nPC_sel
- RegWr
- RegDst
- ExtOp
- ALUSrc
- ALUctr
- MemWr
- MemtoReg

**Instruction<31:0>**

- Op
- Fun

**Inst Memory**

- Adr

**Control**
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** | $R[rd] \leftarrow R[rs] + R[rt]$; $PC \leftarrow PC + 4$
| | \(ALU_{src}=\text{RegB}, \ ALU_{ctr}=\text{"ADD"}, \ RegDst=rd, \ RegWr, \ nPC\_sel=\text{"+4"} \) |
| **sub** | $R[rd] \leftarrow R[rs] - R[rt]$; $PC \leftarrow PC + 4$
| | \(ALU_{src}=\text{RegB}, \ ALU_{ctr}=\text{"SUB"}, \ RegDst=rd, \ RegWr, \ nPC\_sel=\text{"+4"} \) |
| **ori** | $R[rt] \leftarrow R[rs] + \text{zero\_ext}(\text{Imm16})$; $PC \leftarrow PC + 4$
| | \(ALU_{src}=\text{Im}, \ Extop=\text{"Z"}, \ ALU_{ctr}=\text{"OR"}, \ RegDst=rt, \ RegWr, \ nPC\_sel=\text{"+4"} \) |
| **lw** | $R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign\_ext}(\text{Imm16})]$; $PC \leftarrow PC + 4$
| | \(ALU_{src}=\text{Im}, \ Extop=\text{"sn"}, \ ALU_{ctr}=\text{"ADD"}, \ MemtoReg, \ RegDst=rt, \ RegWr, \ nPC\_sel = \text{"+4"} \) |
| **sw** | $\text{MEM}[R[rs] + \text{sign\_ext}(\text{Imm16})] \leftarrow R[rs]$; $PC \leftarrow PC + 4$
| | \(ALU_{src}=\text{Im}, \ Extop=\text{"sn"}, \ ALU_{ctr} = \text{"ADD"}, \ MemWr, \ nPC\_sel = \text{"+4"} \) |
| **beq** | if $(R[rs] == R[rt])$ then $PC \leftarrow PC + \text{sign\_ext}(\text{Imm16})$ \mid \mid 00$
| | else $PC \leftarrow PC + 4$
| | $nPC\_sel = \text{"br"}, \ ALU_{ctr} = \text{"SUB"}$ |
# Summary of the Control Signals (2/2)

See Appendix A

<table>
<thead>
<tr>
<th>op</th>
<th>10 0000</th>
<th>10 0010</th>
<th>We Don’t Care :-)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 0000</td>
<td>00 0000</td>
<td>00 1101</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ori</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| ExtOp | x | x | 0 | 1 | 1 | x | x |
| ALUctr<2:0> | Add | Subtract | Or | Add | Add | Subtract | x |

### 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 |
- jump
Boolean Exprs for Controller

Op 0-5 are really Instruction bits 26-31
Func 0-5 are really Instruction bits 0-5

\[
\begin{align*}
\text{rtype} & = \sim\text{op}_5 \cdot \sim\text{op}_4 \cdot \sim\text{op}_3 \cdot \sim\text{op}_2 \cdot \sim\text{op}_1 \cdot \sim\text{op}_0, \\
\text{ori} & = \sim\text{op}_5 \cdot \sim\text{op}_4 \cdot \text{op}_3 \cdot \text{op}_2 \cdot \sim\text{op}_1 \cdot \text{op}_0 \\
\text{lw} & = \text{op}_5 \cdot \sim\text{op}_4 \cdot \sim\text{op}_3 \cdot \sim\text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\
\text{sw} & = \text{op}_5 \cdot \sim\text{op}_4 \cdot \text{op}_3 \cdot \sim\text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\
\text{beq} & = \sim\text{op}_5 \cdot \sim\text{op}_4 \cdot \sim\text{op}_3 \cdot \text{op}_2 \cdot \sim\text{op}_1 \cdot \sim\text{op}_0 \\
\text{jump} & = \sim\text{op}_5 \cdot \sim\text{op}_4 \cdot \sim\text{op}_3 \cdot \sim\text{op}_2 \cdot \text{op}_1 \cdot \sim\text{op}_0 \\
\text{add} & = \textbf{rtype} \cdot \text{func}_5 \cdot \sim\text{func}_4 \cdot \sim\text{func}_3 \cdot \sim\text{func}_2 \cdot \sim\text{func}_1 \cdot \sim\text{func}_0 \\
\text{sub} & = \textbf{rtype} \cdot \text{func}_5 \cdot \sim\text{func}_4 \cdot \sim\text{func}_3 \cdot \sim\text{func}_2 \cdot \text{func}_1 \cdot \sim\text{func}_0
\end{align*}
\]

How do we implement this in gates?
Controller Implementation

opcode  func

"AND" logic

add  sub  ori  lw  sw  beq  jump

RegDst  ALUSrc  MemtoReg  RegWrite  nPCsel  Jump  ExtOp  ALUctr[0]  ALUctr[1]
Boolean Exprs 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
ALUctr[1]   = ori

(assume ALUctr is 00 ADD, 01 SUB, 10 OR)

How do we implement this in gates?
Controller Implementation

```
opcode func
```

```
add sub ori lw sw beq jump
```

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

"AND" logic

"OR" logic
Call home, we’ve made HW/SW contact!

High Level Language Program (e.g., C) → Compiler → Assembly Language Program (e.g., MIPS) → Assembler → Machine Language Program (MIPS) → Machine Interpretation → Hardware Architecture Description (e.g., block diagrams) → Architecture Implementation → Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000 1010 1111 0101 1000 0000 1001 1100 0110 0101 1000 0000 1001 1000 0000 1001 1100 0110 1010 1111
```
Review: 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
Single Cycle Performance

• Assume time for actions are
  – 100ps for register read or write; 200ps for other events

• Clock rate is?

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

• What can we do to improve clock rate?
• Will this improve performance as well?
  Want increased clock rate to mean faster programs
Single Cycle Performance

• Assume time for actions are
  – 100ps for register read or write; 200ps for other events

• Clock rate is?

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

• What can we do to improve clock rate?
• Will this improve performance as well?
  Want increased clock rate to mean faster programs
Gotta Do Laundry

• Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  – Washer takes 30 minutes
  – Dryer takes 30 minutes
  – “Folder” takes 30 minutes
  – “Stasher” takes 30 minutes to put clothes into drawers
Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads
Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!
Pipelining Lessons (1/2)

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3X v. 4X in this example
• Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

• Pipeline rate limited by slowest pipeline stage

• Unbalanced lengths of pipe stages reduces speedup
Steps in Executing MIPS

1) **IFtch**: Instruction Fetch, Increment PC
2) **Dcd**: Instruction Decode, Read Registers
3) **Exec**:
   - Mem-ref: Calculate Address
   - Arith-log: Perform Operation
4) **Mem**:
   - Load: Read Data from Memory
   - Store: Write Data to Memory
5) **WB**: Write Data Back to Register
Single Cycle Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle
More Detailed Pipeline
IF for Load, Store, ...
ID for Load, Store, ...
EX for Load
MEM for Load
WB for Load – Oops!

Wrong register number
Corrected Datapath for Load
1) Thanks to pipelining, I have **reduced the time** it took me to wash my shirt.

2) Longer pipelines are **always a win** (since less work per stage & a faster clock).

3) We can **rely on compilers** to help us avoid data hazards by reordering instrs.
So, in conclusion

• You now know how to implement the control logic for the single-cycle CPU.
  – (actually, you already knew it!)

• Pipelining improves performance by increasing instruction throughput: exploits ILP
  – Executes multiple instructions in parallel
  – Each instruction has the same latency

• Next: hazards in pipelining:
  – Structure, data, control