Readings

- **Last time and today**
  - Introduction to microarchitecture and single-cycle microarchitecture
    - H&H, Chapter 7.1-7.3
    - P&P, Appendices A and C
  - Multi-cycle microarchitecture
    - H&H, Chapter 7.4
    - P&P, Appendices A and C

- **Tomorrow and next week**
  - Pipelining
    - H&H, Chapter 7.5
  - Pipelining Issues
    - H&H, Chapter 7.8.1-7.8.3
Agenda for Today & Next Few Lectures

- Instruction Set Architectures (ISA): LC-3 and MIPS
- Assembly programming: LC-3 and MIPS
- Microarchitecture (principles & single-cycle uarch)
- Multi-cycle microarchitecture
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
Recall: A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*

\[
\text{AS} = \text{Architectural (programmer visible) state at the beginning of a clock cycle}
\]

\[
\text{Process instruction in one clock cycle}
\]

\[
\text{AS'} = \text{Architectural (programmer visible) state at the end of a clock cycle}
\]
Recall: The Instruction Processing “Cycle”

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
Instruction Processing “Cycle” vs. Machine Clock Cycle

- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, *each phase can take multiple clock cycles to complete*
Recall: Single-Cycle Machine

- Single-cycle machine

AS: Architectural State
Recall: Datapath and Control Logic

- An instruction processing engine consists of two components
  - **Datapath**: Consists of hardware elements that deal with and transform data signals
    - **functional units** that operate on data
    - **hardware structures** (e.g., wires, muxes, decoders, tri-state bufs) that enable the flow of data into the functional units and registers
    - **storage units** that store data (e.g., registers)
  - **Control logic**: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data
Single-Cycle Datapath for 
Arithmetic and Logical Instructions
Datapath for R- and I-Type ALU Insts.

if MEM[PC] == ADDI rt rs immediate
GPR[rt] ← GPR[rs] + sign-extend (immediate)
PC ← PC + 4

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Single-Cycle Datapath for Data Movement Instructions
Datapath for Non-Control-Flow Insts.

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Single-Cycle Datapath for Control Flow Instructions
Jump Instruction

- Unconditional branch or jump

\[ j \text{ target} \]

<table>
<thead>
<tr>
<th>j (2)</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- 2 = opcode
- immediate (target) = target address

Semantics

if \( \text{MEM}[\text{PC}] == j \text{ immediate}_{26} \)

\[
\text{target} = \{ \text{PC}^{\ast}[31:28], \text{immediate}_{26}, 2'\text{b00} \}
\]

\[\text{PC} \leftarrow \text{target}\]

\(^\dagger\) This is the incremented PC
Unconditional Jump Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

if MEM[PC]==J immediate26
PC = { PC[31:28], immediate26, 2’b00 }

What about JR, JAL, JALR?
Other Jumps in MIPS

- **jal**: jump and link (function calls)
  - **Semantics**
    - if $\text{MEM}[\text{PC}] == \text{jal } \text{immediate}_{26}$
      - $\text{ra} \leftarrow \text{PC} + 4$
      - target = \{ $\text{PC}^\dagger[31:28], \text{immediate}_{26}, 2'b00$ \}
      - $\text{PC} \leftarrow \text{target}$

- **jr**: jump register
  - **Semantics**
    - if $\text{MEM}[\text{PC}] == \text{jr } \text{rs}$
      - $\text{PC} \leftarrow \text{GPR(rs)}$

- **jalr**: jump and link register
  - **Semantics**
    - if $\text{MEM}[\text{PC}] == \text{jalr } \text{rs}$
      - $\text{ra} \leftarrow \text{PC} + 4$
      - $\text{PC} \leftarrow \text{GPR(rs)}$

\(^\dagger\) This is the incremented PC
Aside: MIPS Cheat Sheet

- On the course website
Conditional Branch Instructions

- **beq (Branch if Equal)**

  ```
  beq $s0, $s1, offset #$s0=rs,$s1=rt
  ```

<table>
<thead>
<tr>
<th>beq (4)</th>
<th>rs</th>
<th>rt</th>
<th>immediate=offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **Semantics (assuming no branch delay slot)**

  ```
  if MEM[PC] == beq rs rt immediate_{16}
  target = PC\textsuperscript{+} + sign-extend(immediate) \times 4
  if GPR[rs] == GPR[rt] then PC \leftarrow target
  else PC \leftarrow PC + 4
  ```

- **Variations:** beq, bne, blez, bgtz

\textsuperscript{\dagger} This is the incremented PC
Conditional Branch Datapath (for you to finish)

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

How to uphold the delayed branch semantics?
Putting It All Together

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
Single-Cycle Control Logic
### Single-Cycle Hardwired Control

- As combinational function of $\text{Inst} = \text{MEM}[\text{PC}]$

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Consider
  - All R-type and I-type ALU instructions
  - `lw` and `sw`
  - `beq`, `bne`, `blez`, `bgtz`
  - `j`, `jr`, `jal`, `jalr`

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- R-Type

- I-Type

- J-Type
Generate Control Signals (in Orange Color)

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
## Single-Bit Control Signals (I)

<table>
<thead>
<tr>
<th></th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RegDest</strong></td>
<td>GPR write select according to ( rt ), i.e., ( \text{inst}[20:16] )</td>
<td>GPR write select according to ( rd ), i.e., ( \text{inst}[15:11] )</td>
<td>( \text{opcode}==0 )</td>
</tr>
<tr>
<td><strong>ALUSrc</strong></td>
<td>2(^{nd}) ALU input from 2(^{nd}) GPR read port</td>
<td>2(^{nd}) ALU input from sign-extended 16-bit immediate</td>
<td>( \text{opcode}!=0 ) &amp;&amp; ((\text{opcode}!=\text{BEQ}) ) &amp;&amp; ((\text{opcode}!=\text{BNE}) )</td>
</tr>
<tr>
<td><strong>MemtoReg</strong></td>
<td>Steer ALU result to GPR write port</td>
<td>Steer memory output to GPR write port</td>
<td>( \text{opcode}==\text{LW} )</td>
</tr>
<tr>
<td><strong>RegWrite</strong></td>
<td>GPR write disabled</td>
<td>GPR write enabled</td>
<td>( \text{opcode}!=\text{SW} ) &amp;&amp; ((\text{opcode}!=\text{Bxx}) ) &amp;&amp; ((\text{opcode}!=\text{J}) ) &amp;&amp; ((\text{opcode}!=\text{JR}) )</td>
</tr>
</tbody>
</table>

JAL and JALR require additional RegDest and MemtoReg options
## Single-Bit Control Signals (II)

<table>
<thead>
<tr>
<th></th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MemRead</strong></td>
<td>Memory read disabled</td>
<td>Memory read port returns load value</td>
<td>opcode==LW</td>
</tr>
<tr>
<td><strong>MemWrite</strong></td>
<td>Memory write disabled</td>
<td>Memory write enabled</td>
<td>opcode==SW</td>
</tr>
<tr>
<td><strong>PCSrc&lt;sub&gt;1&lt;/sub&gt;</strong></td>
<td>According to PCSrc&lt;sub&gt;2&lt;/sub&gt;</td>
<td>next PC is based on 26-bit immediate jump target</td>
<td>(opcode==J)</td>
</tr>
<tr>
<td><strong>PCSrc&lt;sub&gt;2&lt;/sub&gt;</strong></td>
<td>next PC = PC + 4</td>
<td>next PC is based on 16-bit immediate branch target</td>
<td>(opcode==Bxx) &amp;&amp; “bcond is satisfied”</td>
</tr>
</tbody>
</table>

JR and JALR require additional PCSrc options.
R-Type ALU

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]*
I-Type ALU

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Branch (Not Taken)

Some control signals are dependent on the processing of data

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Some control signals are dependent on the processing of data.
What is in That Control Box?

- **Combinational Logic** → **Hardwired Control**
  - Idea: Control signals generated combinatorially based on bits in instruction encoding

- **Sequential Logic** → **Sequential Control**
  - Idea: A memory structure contains the control signals associated with an instruction
    - Called **Control Store**

- **Both types of control structure can be used in single-cycle processors**
  - Choice depends on latency of each structure + how much on the critical path control signal generation is, etc.
Another Single-Cycle MIPS Processor (from H&H)

See backup slides to reinforce the concepts we have covered. They are to complement your reading:
H&H, Chapter 7.1-7.3, 7.6
Another Complete Single-Cycle Processor

Single-cycle processor. Harris and Harris, Chapter 7.3.
Example: Single-Cycle Datapath: `lw` fetch

**STEP 1:** Fetch instruction

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: `lw` register read

**STEP 2:** Read source operands from register file

\[ \text{lw }\$s3, \text{1}($\theta) \quad \# \text{read memory word 1 into } \$s3 \]

**I-Type**

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: \texttt{lw} immediate

- **STEP 3:** Sign-extend the immediate

\texttt{lw} $\texttt{\$s3}, 1(\texttt{\$0})$  # read memory word 1 into $\texttt{\$s3}$

\textbf{I-Type}

\begin{tabular}{|c|c|c|c|}
  \hline
  \textbf{op} & \textbf{rs} & \textbf{rt} & \textbf{imm} \\
  \hline
  6 bits & 5 bits & 5 bits & 16 bits \\
  \hline
\end{tabular}
Single-Cycle Datapath: lw address

**STEP 4:** Compute the memory address

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: lw memory read

**STEP 5:** Read from memory and write back to register file

```plaintext
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: `lw` PC increment

**STEP 6:** Determine address of next instruction

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Similarly, We Need to Design the Control Unit

- **Control signals** are generated by the decoder in control unit

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOP&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>001000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>000010</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>XX</td>
<td>1</td>
</tr>
</tbody>
</table>
Another Complete Single-Cycle Processor (H&H)
Your Reading Assignment

- Please read the Lecture Slides & the Backup Slides

- Please do your readings from the H&H Book
  - H&H, Chapter 7.1-7.3, 7.6
**Single-Cycle Uarch I (We Developed in Lectures)**

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
Single-Cycle Uarch II (In Your Readings)

Single-cycle processor. Harris and Harris, Chapter 7.3.
Evaluating the Single-Cycle Microarchitecture
A Single-Cycle Microarchitecture

- Is *this* a good idea/design?

- When is this a good design?

- When is this a bad design?

- How can we design a better microarchitecture?
Performance Analysis Basics
Recall: Performance Analysis Basics

- Execution time of a single instruction
  - \( \{\text{CPI}\} \times \{\text{clock cycle time}\} \)
    - CPI: Number of cycles it takes to execute an instruction

- Execution time of an entire program
  - Sum over all instructions \( \{\text{CPI}\} \times \{\text{clock cycle time}\} \)
  - \( \{\text{# of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\} \)
Processor Performance

How fast is my program?

- Every program consists of a series of instructions
- Each instruction needs to be executed
Processor Performance

■ How fast is my program?
  ▪ Every program consists of a series of instructions
  ▪ Each instruction needs to be executed

■ How fast are my instructions?
  ▪ Instructions are realized on the hardware
  ▪ Each instruction can take one or more clock cycles to complete
  ▪ \textit{Cycles per Instruction} = CPI
Processor Performance

How fast is my program?
- Every program consists of a series of instructions
- Each instruction needs to be executed

How fast are my instructions?
- Instructions are realized on the hardware
- Each instruction can take one or more clock cycles to complete
- \textit{Cycles per Instruction} = CPI

How long is one clock cycle?
- The critical path determines how much time one cycle requires = \textit{clock period}
- 1/clock period = \textit{clock frequency} = how many cycles can be done each second
Processor Performance

- As a general formula
  - Our program consists of executing $N$ instructions
  - Our processor needs $CPI$ cycles (on average) for each instruction
  - The clock frequency of the processor is $f$
    - the clock period is therefore $T=1/f$
Processor Performance

- **As a general formula**
  - Our program consists of executing $N$ instructions
  - Our processor needs CPI cycles (on average) for each instruction
  - The clock frequency of the processor is $f$
    - the clock period is therefore $T=1/f$

- **Our program executes in**

\[
N \times \text{CPI} \times \frac{1}{f} = \]

$N \times \text{CPI} \times T$ seconds
Performance Analysis of Our Single-Cycle Design
A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction
What is the Slowest Instruction to Process?

- Let’s go back to the basics

- All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result

- Do each of the above phases take the same time (latency) for all instructions?
Let’s Find the Critical Path
Example Single-Cycle Datapath Analysis

- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

<table>
<thead>
<tr>
<th>steps</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>resources</td>
<td>mem</td>
<td>RF</td>
<td>ALU</td>
<td>mem</td>
<td>RF</td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>I-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>600</td>
</tr>
<tr>
<td>SW</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>550</td>
</tr>
<tr>
<td>Branch</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>
Let's Find the Critical Path

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
R-Type and I-Type ALU

[Diagram showing the timing and operations of R-Type and I-Type ALU, with labels indicating delays such as 100ps, 200ps, 250ps, 350ps, and 400ps.]

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Branch Taken
Jump

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
What About Control Logic?

- How does that affect the critical path?

- Food for thought for you:
  - Can control logic be on the critical path?
  - Historical example:
    - CDC 5600: control store access too long...
What is the Slowest Instruction to Process?

- Real world: **Memory is slow (not magic)**

- What if memory *sometimes* takes 100ms to access?

- Does it make sense to have a simple register to register add or jump to take \{100ms+all else to do a memory operation\}?  

- And, what if you need to access memory more than once to process an instruction?  
  - Which instructions need this?  
  - Do you provide multiple ports to memory?
Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction

- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle

- Not necessarily the simplest way to implement an ISA
  - Single-cycle implementation of REP MOVS (x86) or INDEX (VAX)?

- Not easy to optimize/improve performance
  - Optimizing the common case (frequent instructions) does not work
  - Need to optimize the worst case all the time
(Micro)architecture Design Principles

- **Critical path design**
  - Find and **decrease the maximum combinational logic delay**
  - Break a path into multiple cycles if it takes too long

- **Bread and butter (common case) design**
  - Spend time and resources on where it matters most
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case

- **Balanced design**
  - Balance instruction/data flow through hardware components
  - **Design to eliminate bottlenecks**: balance the hardware for the work
Single-Cycle Design vs. Design Principles

- Critical path design
- Bread and butter (common case) design
- Balanced design

How does a single-cycle microarchitecture fare with respect to these principles?
Aside: System Design Principles

- When designing computer systems/architectures, it is important to follow good principles
  - Actually, this is true for *any* system design
    - Real architectures, buildings, bridges, ...
    - Good consumer products
    - Mechanisms for security/safety-critical systems
    - ...

- Remember: “principled design” from our second lecture
  - Frank Lloyd Wright: “architecture [...] based upon principle, and not upon precedent”
Aside: From Lecture 2

- “architecture [...] based upon principle, and not upon precedent”
This
That
Recall: Takeaways

- It all starts from the **basic building blocks and design principles**

- And, **knowledge of how to use, apply, enhance them**

- **Underlying technology might change** (e.g., steel vs. wood)
  - but **methods of taking advantage of technology bear resemblance**
  - **methods used for design depend on the principles employed**
Aside: System Design Principles

- We will continue to cover key principles in this course.
- Here are some references where you can learn more:
  - Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS Conference, April 1967. (Amdahl’s Law → Common-case design)
A Key System Design Principle

- Keep it simple

- “Everything should be made as simple as possible, but no simpler.”
  - Albert Einstein

- And, keep it low cost: “An engineer is a person who can do for a dime what any fool can do for a dollar.”

- For more, see:
Can We Do Better?
Multi-Cycle Microarchitectures
Multi-Cycle Microarchitectures

- **Goal**: Let each instruction take (close to) only as much time it really needs

- **Idea**
  - Determine clock cycle time independently of instruction processing time
  - Each instruction takes as many clock cycles as it needs to take
    - Multiple state transitions per instruction
    - The states followed by each instruction is different
Recall: The “Process Instruction” Step

- ISA specifies abstractly what AS’ should be, given an instruction and AS
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no “intermediate states” between AS and AS’ during instruction execution
    - One state transition per instruction

- Microarchitecture implements how AS is transformed to AS’
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: **multiple** state transitions per instruction
    - Choice 1: \( \text{AS} \rightarrow \text{AS'} \) (transform AS to AS’ in a single clock cycle)
    - Choice 2: \( \text{AS} \rightarrow \text{AS+MS1} \rightarrow \text{AS+MS2} \rightarrow \text{AS+MS3} \rightarrow \text{AS'} \) (take multiple clock cycles to transform AS to AS’)

83
Multi-Cycle Microarchitecture

\[ AS = \text{Architectural (programmer visible) state at the beginning of an instruction} \]

Step 1: Process part of instruction in one clock cycle

Step 2: Process part of instruction in the next clock cycle

\[ \ldots \]

\[ AS' = \text{Architectural (programmer visible) state at the end of a clock cycle} \]
Benefits of Multi-Cycle Design

- **Critical path design**
  - Can keep reducing the critical path independently of the worst-case processing time of any instruction

- **Bread and butter (common case) design**
  - Can optimize the number of states it takes to execute “important” instructions that make up much of the execution time

- **Balanced design**
  - No need to provide more capability or resources than really needed
    - An instruction that needs resource X multiple times does not require multiple X’s to be implemented
    - Leads to more efficient hardware: Can reuse hardware components needed multiple times for an instruction
Downsides of Multi-Cycle Design

- Need to store the intermediate results at the end of each clock cycle
  - Hardware overhead for registers
  - Register setup/hold overhead paid multiple times for an instruction
Remember: Performance Analysis

- Execution time of a single instruction
  - \( \{\text{CPI}\} \times \{\text{clock cycle time}\} \)
  - CPI: Cycles Per Instruction

- Execution time of an entire program
  - Sum over all instructions \([\{\text{CPI}\} \times \{\text{clock cycle time}\}]\)
  - \(\{\text{# of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\}\)

- Single-cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long

- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

In multi-cycle, we have two degrees of freedom to optimize independently
A Multi-Cycle Microarchitecture

A Closer Look
How Do We Implement This?


An elegant implementation:
- The concept of microcoded/microprogrammed machines
Multi-Cycle Microarchitectures

Key Idea for Realization

- One can implement the “process instruction” step as a finite state machine that sequences between states and eventually returns back to the “fetch instruction” state.
- A state is defined by the control signals asserted in it.
- Control signals for the next state are determined in current state.
Recall: The Instruction Processing “Cycle”

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into “states”
  - A stage in the instruction processing cycle can take multiple states

- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state

- The behavior of the entire processor is specified fully by a finite state machine

- In a state (clock cycle), control signals control two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle
One Example Multi-Cycle Microarchitecture
Remember: Single-Cycle MIPS Processor
Multi-Cycle MIPS Processor

- **Single-cycle microarchitecture:**
  - cycle time limited by longest instruction (lw) → low clock frequency
  - three adders/ALUs and two memories → high hardware cost

- **Multi-cycle microarchitecture:**
  + higher clock frequency
  + simpler instructions take few clock cycles
  + reuse expensive hardware across multiple cycles
  - sequencing overhead paid many times
  - hardware overhead for storing intermediate results

- **Multi-cycle requires the same design steps as single cycle:**
  - datapath
  - control logic
What Do We Want To Optimize?

- Single-cycle microarchitecture uses two memories
  - One memory stores instructions, the other data
  - *We want to use a single memory (lower cost)*
What Do We Want To Optimize?

- **Single-cycle microarchitecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (lower cost)

- **Single-cycle microarchitecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use only one ALU for all operations (lower cost)
What Do We Want To Optimize?

- **Single-cycle microarchitecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (lower cost)

- **Single-cycle microarchitecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use only one ALU for all operations (lower cost)

- **Single-cycle microarchitecture: each instruction takes one cycle**
  - The slowest instruction slows down every single instruction
  - We want to determine clock cycle time independently of instruction processing time
    - Divide each instruction into multiple clock cycles
    - Simpler instructions can be very fast (compared to the slowest)
Let’s Construct the Multi-Cycle Datapath
Consider the lw Instruction

- For an instruction such as: `lw $t0, 0x20($t1)`

- We need to:
  - Read the instruction from memory
  - Then read $t1 from register array
  - Add the immediate value (0x20) to calculate the memory address
  - Read the content of this address
  - Write to the register $t0 this content
Multi-Cycle Datapath: Instruction Fetch

- We will consider `lw`, but fetch is the same for all instructions
  - **STEP 1:** Fetch instruction

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

read from the memory location \([rs]+imm\) to location \([rt]\)

**I-Type**
Multi-Cycle Datapath: \texttt{lw} register read

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{op} & \textbf{rs} & \textbf{rt} & \textbf{imm} \\
\hline
6 bits & 5 bits & 5 bits & 16 bits \\
\hline
\end{tabular}
\end{table}
Multi-Cycle Datapath: lw immediate

I-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Multi-Cycle Datapath: lw address

I-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Multi-Cycle Datapath: lw memory read

I-Type

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Multi-Cycle Datapath: lw write register

I-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Multi-Cycle Datapath: increment PC
Multi-Cycle Datapath: \textit{sw}

- Write data in rt to memory
Multi-Cycle Datapath: R-type Instructions

- Read from rs and rt
  - Write ALUResult to register file
  - Write to rd (instead of rt)
Multi-Cycle Datapath: beq

- Determine whether values in rs and rt are equal
  - Calculate branch target address:
    \[ \text{Target Address} = (\text{sign-extended immediate} \ll 2) + (\text{PC} + 4) \]
Complete Multi-Cycle Processor
Let’s Construct
the Multi-Cycle Control Logic
Control Unit

Control Unit

Main Controller (FSM)

OpCode_{5:0}  MemtoReg  RegDst  IorD  PCSrc  ALUSrcB_{1:0}  ALUSrcA  IRWrite  MemWrite  PCWrite  Branch  RegWrite

RegWrite

ALUOp_{1:0}

Funct_{5:0}  ALUControl_{2:0}

ALU Decoder

Multiplexer Selects

Register Enables
Main Controller FSM: Fetch
Main Controller FSM: Fetch

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

IorD
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
Main Controller FSM: Decode

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

S1: Decode
Main Controller FSM: Address Calculation

S0: Fetch

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
Reset

S1: Decode

Op = LW
or
Op = SW

S2: MemAdr

Op = LW
or
Op = SW
Main Controller FSM: Address Calculation

S0: Fetch
- IorD = 0
- AluSrcA = 0
- ALUSrcB = 01
- ALUOp = 00
- PCSrc = 0

S1: Decode
- IRWrite
- PCWrite
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00
- Reset

S2: MemAdr
Op = LW
or
Op = SW

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S2: MemAdr
- Op = LW
- Op = SW

CLK
A
RD
Instr / Data
Memory
A1
A3
WD3
RD2
RD1
WE3
A2
CLK
Sign Extend
Register
File
0
1
0
1
00
01
10
11
4
CLK
EN
ALUSrcB1:0
IRWrite
IorD
PCWrite
PCEn
X
0
0
Main Controller FSM: \texttt{lw}

### States
- **S0: Fetch**
- **S1: Decode**
- **S2: MemAdr**
- **S3: MemRead**
- **S4: Mem Writeback**

### Registers
- \texttt{IorD}
- \texttt{AluSrcA}
- \texttt{ALUSrcB}
- \texttt{ALUOp}
- \texttt{PCSrc}
- \texttt{IRWrite}
- \texttt{PCWrite}

### Op Code
- \texttt{LW}
- \texttt{SW}

### Transition Rules
- From **S0: Fetch** to **S1: Decode**:
  - \texttt{Op = LW}
- From **S2: MemAdr** to **S3: MemRead**:
  - \texttt{Op = LW}
- From **S3: MemRead** to **S4: Mem Writeback**:
  - \texttt{IorD = 1}

### Example Transition
- \texttt{IorD = 0}
- \texttt{AluSrcA = 0}
- \texttt{ALUSrcB = 01}
- \texttt{ALUOp = 00}
- \texttt{PCSrc = 0}
- \texttt{IRWrite}
- \texttt{PCWrite}

- \texttt{ALUSrcA = 1}
- \texttt{ALUSrcB = 10}
- \texttt{ALUOp = 00}

- \texttt{IorD = 1}
- \texttt{RegWrite}
- \texttt{MemtoReg = 1}
- \texttt{MemtoReg = 1}
Main Controller FSM: sw

S0: Fetch
- IorD = 0
- AluSrcA = 0
- ALUSrcB = 01
- ALUOp = 00
- PCSrc = 0
- IRWrite
- PCWrite

S1: Decode
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00
- IorD ...

S2: MemAdr
- Op = LW
- or
- Op = SW

S3: MemRead
- IorD = 1

S4: MemWriteback
- RegDst = 0
- MemtoReg = 1
- RegWrite

S5: MemWrite
- IorD = 1
- MemWrite
Main Controller FSM: R-Type

S0: Fetch
- IorD = 0
- AluSrcA = 0
- ALUSrcB = 01
- ALUOp = 00
- PCSrc = 0
- IRWrite
- PCWrite

S1: Decode
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00
- IorD = 1
- RegDst = 1
- MemtoReg = 0
- RegWrite

S2: MemAdr
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00
- Op = LW
- or
- Op = SW

S3: MemRead
- IorD = 1

S4: Mem Writeback
- RegDst = 0
- MemtoReg = 1
- RegWrite

S5: MemWrite
- Op = R-type

S6: Execute
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 10
- Op = R-type

S7: ALU Writeback
- RegDst = 1
- MemtoReg = 0
- RegWrite
Main Controller FSM: beq

S0: Fetch
Reset

S1: Decode
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

S2: MemAdr
Op = LW
or
Op = SW

S3: MemRead
Op = LW

S4: MemWriteback
IorD = 1
MemWrite

S4: MemWriteback
RegDst = 0
MemtoReg = 1
RegWrite

S5: MemWrite
IorD = 1

S6: Execute
ALUSrcA = 0
ALUSrcB = 00
ALUOp = 00

S7: ALU Writeback
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch

S8: Branch
Op = BEQ
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch

S5: MemWrite
RegDst = 1
MemtoReg = 0
RegWrite

S6: Execute
ALUSrcA = 1
ALUSrcB = 11
ALUOp = 00

S2: MemAdr
Op = R-type

S3: MemAdr
Op = LW
or
Op = SW

S6: Execute
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 00

S7: ALU Writeback
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

S8: Branch
Op = BEQ
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch

S5: MemWrite
RegDst = 1
MemtoReg = 0
RegWrite
Complete Multi-Cycle Controller FSM

S0: Fetch
- Reset
- iorD = 0
- AluSrcA = 0
- ALUSrcB = 01
- ALUOp = 00
- PCSrc = 0
- IRWrite
- PCWrite

S1: Decode
- ALUSrcA = 0
- ALUSrcB = 11
- ALUOp = 00

S2: MemAdr
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00

S3: MemRead
- Op = LW
- or
- Op = SW

S4: MemWriteback
- iorD = 1
- MemWrite

S5: MemWrite
- RegDst = 0
- MemtoReg = 1
- RegWrite

S6: Execute
- Op = R-type

S7: ALU Writeback
- S5: MemWrite
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 10

S8: Branch
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 01
- PCSrc = 1
- Branch
Main Controller FSM: \texttt{addi}

- **S0: Fetch**
  - \( IorD = 0 \)
  - \( AluSrcA = 0 \)
  - \( ALUSrcB = 01 \)
  - \( ALUOp = 00 \)
  - \( PCSrc = 0 \)
  - \( IRWrite \)
  - \( PCWrite \)

- **S1: Decode**
  - \( ALUSrcA = 0 \)
  - \( ALUSrcB = 11 \)
  - \( ALUOp = 00 \)

- **S2: MemAdr**
  - \( ALUSrcA = 1 \)
  - \( ALUSrcB = 10 \)
  - \( ALUOp = 00 \)
  - \( Op = LW \)

- **S3: MemRead**
  - \( IorD = 1 \)
  - \( MemWrite \)

- **S4: MemWrite**
  - \( RegDst = 0 \)
  - \( MemtoReg = 1 \)
  - \( RegWrite \)

- **S5: MemWrite**
  - \( Op = SW \)

- **S6: Execute**
  - \( ALUSrcA = 1 \)
  - \( ALUSrcB = 00 \)
  - \( ALUOp = 01 \)

- **S7: ALU Writeback**
  - \( RegDst = 1 \)
  - \( MemtoReg = 0 \)
  - \( RegWrite \)

- **S8: Branch**
  - \( ALUSrcA = 1 \)
  - \( ALUSrcB = 00 \)
  - \( ALUOp = 01 \)
  - \( PCSrc = 1 \)
  - \( Branch \)

- **S9: ADDI Execute**
  - \( Op = ADDI \)

- **S10: ADDI Writeback**
  - \( Op = ADDI \)
Main Controller FSM: addi

- **S0: Fetch**: Reset
- **S1: Decode**: ALUSrcA = 0, ALUSrcB = 01, ALUOp = 00, PCSrc = 0, IRWrite, PCWrite
- **S2: MemAdr**: ALUSrcA = 1, ALUSrcB = 10, ALUOp = 00, Op = LW or SW
- **S3: MemRead**: ALUSrcA = 1, ALUSrcB = 10, ALUOp = 00, Op = LW
- **S4: Mem Writeback**: IorD = 1, MemWrite, RegDst = 0, MemtoReg = 1, RegWrite
- **S5: MemWrite**: IorD = 1, MemWrite, RegDst = 1, MemtoReg = 0, RegWrite
- **S6: Execute**: ALUSrcA = 1, ALUSrcB = 00, ALUOp = 10, PCSrc = 1, Branch, Op = ADDI
- **S7: ALU Writeback**: ALUSrcA = 1, ALUSrcB = 10, ALUOp = 00, Op = ADDI, Execute
- **S8: Branch**: ALUSrcA = 1, ALUSrcB = 00, ALUOp = 01, PCSrc = 1, Branch, Op = BEQ
- **S9: ADDI**: ALUSrcA = 1, ALUSrcB = 10, ALUOp = 00, RegDst = 0, MemtoReg = 0, RegWrite
- **S10: ADDI Writeback**: ALUSrcA = 1, ALUSrcB = 10, ALUOp = 00, RegDst = 0, MemtoReg = 0, RegWrite
Extended Functionality: j
Control FSM: j

- **S0: Fetch**
  - IorD = 0
  - AluSrcA = 0
  - ALUSrcB = 01
  - ALUOp = 00
  - PCSrc = 00
  - IRWrite
  - PCWrite

- **S1: Decode**
  - ALUSrcA = 0
  - ALUSrcB = 11
  - ALUOp = 00

- **S2: MemAdr**
  - ALUSrcA = 1
  - ALUSrcB = 10
  - ALUOp = 00
  - Op = LW or SW

- **S3: MemRead**
  - IorD = 1
  - MemWrite

- **S4: Mem Writeback**
  - RegDst = 0
  - MemtoReg = 1
  - RegWrite

- **S5: MemWrite**
  - IorD = 1
  - MemWrite

- **S6: Execute**
  - RegDst = 0
  - MemtoReg = 0
  - RegWrite
  - Op = ADDI

- **S7: ALU Writeback**
  - RegDst = 0
  - MemtoReg = 1
  - RegWrite

- **S8: Branch**
  - ALUSrcA = 1
  - ALUSrcB = 00
  - ALUOp = 01
  - PCSrc = 01
  - Branch

- **S9: ADDI Execute**
  - ALUSrcA = 1
  - ALUSrcB = 00
  - ALUOp = 10

- **S10: ADDI Writeback**
  - RegDst = 0
  - MemtoReg = 0
  - RegWrite

- **S11: Jump**
  - ALUSrcA = 0
  - ALUSrcB = 11
  - ALUOp = 00
  - Op = J
Control FSM: j

- **S0: Fetch**
  - Reset
  - \( \text{IorD} = 0 \), \( \text{AluSrcA} = 0 \), \( \text{ALUSrcB} = 01 \), \( \text{ALUOp} = 00 \)
  - \( \text{IRWrite} \), \( \text{PCWrite} \)

- **S1: Decode**
  - \( \text{ALUSrcA} = 0 \), \( \text{ALUSrcB} = 11 \), \( \text{ALUOp} = 00 \)

- **S2: MemAdr**
  - \( \text{Op} = \text{LW} \) or \( \text{Op} = \text{SW} \)
  - \( \text{ALUSrcA} = 1 \), \( \text{ALUSrcB} = 00 \), \( \text{ALUOp} = 10 \)

- **S3: MemRead**
  - \( \text{IorD} = 1 \)
  - \( \text{Op} = \text{SW} \)

- **S4: Mem Writeback**
  - \( \text{RegDst} = 0 \), \( \text{MemtoReg} = 1 \), \( \text{RegWrite} \)

- **S5: MemWrite**
  - \( \text{IorD} = 1 \)
  - \( \text{MemWrite} \)

- **S6: Execute**
  - \( \text{RegDst} = 1 \), \( \text{MemtoReg} = 0 \), \( \text{RegWrite} \)

- **S7: ALU Writeback**
  - \( \text{ALUSrcA} = 1 \), \( \text{ALUSrcB} = 00 \), \( \text{ALUOp} = 01 \)
  - \( \text{PCSrc} = 01 \)
  - Branch

- **S8: Branch**
  - \( \text{Op} = \text{BEQ} \)

- **S9: ADDI**
  - \( \text{ALUSrcA} = 1 \), \( \text{ALUSrcB} = 00 \), \( \text{ALUOp} = 01 \)
  - \( \text{RegDst} = 0 \), \( \text{MemtoReg} = 0 \), \( \text{RegWrite} \)

- **S10: ADDI**
  - \( \text{Op} = \text{ADDI} \)

- **S11: Jump**
  - \( \text{PCSrc} = 10 \)
  - \( \text{PCWrite} \)
  - \( \text{Op} = \text{J} \)
Review: Single-Cycle MIPS Processor
Review: Single-Cycle MIPS FSM

- Single-cycle machine

AS: Architectural State
Review: Multi-Cycle MIPS Processor
Review: Multi-Cycle MIPS FSM

What is the shortcoming of this design?

What does this design assume about memory?
What If Memory Takes > One Cycle?

- Stay in the same “memory access” state until memory returns the data
- “Memory Ready?” bit is an input to the control logic that determines the next state
Another Example: Microprogrammed Multi-Cycle Microarchitecture
Recall: How Do We Implement This?


An elegant implementation:
  - The concept of microcoded/microprogrammed machines
Example uProgrammed Control & Datapath

For your own study
P&P Revised Appendix C
On website
+ In Backup Slides

Microarchitecture of the LC-3b, major components

Control)

Data Path

Control

Datapath

Memory, I/O

Data

Data, Inst.

IR[15:11]

BEN

Addr

3

16

23

35

16

26

3

9

(J, COND, IRD)

16
Microprogrammed Control Terminology

- Control signals associated with the current state
  - Microinstruction

- Act of transitioning from one state to another
  - Determining the next state and the microinstruction for the next state
  - Microsequencing

- **Control store** stores control signals for every possible state
  - Store for microinstructions for the entire FSM

- **Microsequencer** determines which set of control signals will be used in the next clock cycle (i.e., next state)
Design of Digital Circuits, Spring 2018, Lecture 13
- Microprogramming (ETH Zürich, Spring 2018)
  - https://www.youtube.com/watch?v=u4GhShuBP3Y&list=PL5Q2soXY2Zi_QedyPWtRmFUJ2F8DdYP7l&index=13

Computer Architecture, Spring 2013, Lecture 7
- Microprogramming (CMU, Spring 2013)
  - https://www.youtube.com/watch?v=_igvSl5h8cs&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=7
We did not cover the following slides. They are for your benefit.
Please study these to reinforce the concepts we covered in lectures.

Please do the readings together with these slides:
H&H, Chapter 7.1-7.3, 7.6
Another Single-Cycle MIPS Processor (from H&H)

These are slides for your own study. They are to complement your reading H&H, Chapter 7.1-7.3, 7.6
What to do with the Program Counter?

- The PC needs to be incremented by 4 during each cycle (for the time being).
- Initial PC value (after reset) is 0x00400000

```verilog
reg [31:0] PC_p, PC_n; // Present and next state of PC

// [...]

assign PC_n <= PC_p + 4; // Increment by 4;

always @ (posedge clk, negedge rst)
begin
    if (rst == '0') PC_p <= 32'h00400000; // default
    else PC_p <= PC_n; // when clk
end
```
We Need a Register File

- **Store 32 registers, each 32-bit**
  - $2^5 = 32$, we need 5 bits to address each

- **Every R-type instruction uses 3 register**
  - Two for reading (RS, RT)
  - One for writing (RD)

- **We need a special memory with:**
  - 2 read ports (address x2, data out x2)
  - 1 write port (address, data in)
Register File

```verilog
input [4:0] a_rs, a_rt, a_rd;
input [31:0] di_rd;
input we_rd;
output [31:0] do_rs, do_rt;

reg [31:0] R_arr [31:0]; // Array that stores regs

// Circuit description
assign do_rs = R_arr[a_rs]; // Read RS
assign do_rt = R_arr[a_rt]; // Read RT

always @(posedge clk)
  if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```
Register File

```verilog
input [4:0] a_rs, a_rt, a_rd;
input [31:0] di_rd;
input we_rd;
output [31:0] do_rs, do_rt;

reg [31:0] R_arr [31:0]; // Array that stores regs

// Circuit description; add the trick with $0
assign do_rs = (a_rs != 5'b00000)? // is address 0?
    R_arr[a_rs] : 0; // Read RS or 0

assign do_rt = (a_rt != 5'b00000)? // is address 0?
    R_arr[a_rt] : 0; // Read RT or 0

always @ (posedge clk)
    if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```
Data Memory Example

Will be used to store the bulk of data

```vhdl
input [15:0] addr; // Only 16 bits in this example
input [31:0] di;
input we;
output [31:0] do;

reg [31:0] M_arr [0:65535]; // Array for Memory

// Circuit description
assign do = M_arr[addr]; // Read memory

always @(posedge clk)
  if (we) M_arr[addr] <= di; // write memory
```
Single-Cycle Datapath: \texttt{lw} fetch

\[ \text{\textbf{STEP 1:} Fetch instruction} \]

\begin{verbatim}
\texttt{lw} $s3, 1($0)  \quad \# \text{read memory word 1 into } s3
\end{verbatim}

\hspace{1cm}

\begin{verbatim}
I-Type
\end{verbatim}

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{op} & \textbf{rs} & \textbf{rt} & \textbf{imm} \\
6 bits & 5 bits & 5 bits & 16 bits \\
\hline
\end{tabular}
\end{center}
Single-Cycle Datapath: lw register read

**STEP 2:** Read source operands from register file

\[
\text{lw}\; \$s3, 1(\$0) \quad \# \text{read memory word 1 into } \$s3
\]

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
**Single-Cycle Datapath: lw immediate**

- **STEP 3**: Sign-extend the immediate

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: lw address

**STEP 4:** Compute the memory address

\[ \text{lw } \$s3, 1(\$0) \]  # read memory word 1 into $s3

**I-Type**

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: \texttt{lw} memory read

**STEP 5:** Read from memory and write back to register file

 lw  \$s3, 1(\$0)  # read memory word 1 into \$s3

**I-Type**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>imm</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: `lw` PC increment

**STEP 6:** Determine address of next instruction

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

---

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: \texttt{sw}

- Write data in \texttt{rt} to memory

\texttt{sw} $\$t7, 44(\$0)$  \# write \$t7 into memory address 44

\textbf{I-Type}

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: R-type Instructions

- **R-type Instructions**: Read from rs and rt, write ALUResult to register file

R-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<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>
</tr>
</tbody>
</table>
Single-Cycle Datapath: beq

- Determine whether values in rs and rt are equal
- Calculate BTA = (sign-extended immediate << 2) + (PC+4)

```
beq $s0, $s1, target  # branch is taken
```
Complete Single-Cycle Processor
Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate *(MUX)*

- **Write Address of Register File**
  - Either RD or RT *(MUX)*

- **Write Data In of Register File**
  - Either ALU out or Data Memory Out *(MUX)*

- **Write enable of Register File**
  - Not always a register write *(MUX)*

- **Write enable of Memory**
  - Only when writing to memory (sw) *(MUX)*

*All these options are our control signals*
Control Unit

- **Control Unit**
  - **Opcode**
  - **Main Decoder**
    - MemtoReg
    - MemWrite
    - Branch
    - ALUSrc
    - RegDst
    - RegWrite

- **ALU Decoder**
  - **ALUOp**
  - **Funct**

- **ALUOp** table:
  - 00: add
  - 01: subtract
  - 10: look at funct field
  - 11: n/a
ALU Does the Real Work in a Processor

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A &amp; ~B</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
ALU Internals

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A \lor B</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A &amp; \neg B</td>
</tr>
<tr>
<td>101</td>
<td>A \lor \neg B</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
Control Unit: ALU Decoder

<table>
<thead>
<tr>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Look at Funct</td>
</tr>
<tr>
<td>11</td>
<td>Not Used</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Funct</th>
<th>ALUControl&lt;sub&gt;2:0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td>010 (Add)</td>
</tr>
<tr>
<td>X1</td>
<td>X</td>
<td>110 (Subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100000 (add)</td>
<td>010 (Add)</td>
</tr>
<tr>
<td>1X</td>
<td>100010 (sub)</td>
<td>110 (Subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100100 (and)</td>
<td>000 (And)</td>
</tr>
<tr>
<td>1X</td>
<td>100101 (or)</td>
<td>001 (Or)</td>
</tr>
<tr>
<td>1X</td>
<td>101010 (slt)</td>
<td>111 (SLT)</td>
</tr>
</tbody>
</table>
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$Op_{5:0}$</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op_{5:0}</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>add</td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op_{5:0}</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>add</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>add</td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do
# More Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>0000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>add</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>add</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>sub</td>
</tr>
</tbody>
</table>

- **New Control Signal**
  - **Branch**: Are we jumping or not?
### Control Unit: Main Decoder

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath Example: or
Extended Functionality: addi

- No change to datapath
## Control Unit: `addi`

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
</tr>
<tr>
<td>addi</td>
<td>001000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>
Extended Functionality: j
## Control Unit: Main Decoder

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op_{5:0}</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>0000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>XX</td>
<td>1</td>
</tr>
</tbody>
</table>
Review: Complete Single-Cycle Processor (H&H)
A Bit More on Performance Analysis
How can I Make the Program Run Faster?

\[ N \times \text{CPI} \times \frac{1}{f} \]
How can I Make the Program Run Faster?

\[ N \times CPI \times (1/f) \]

- Reduce the number of instructions
  - Make instructions that ‘do’ more (CISC – complex instruction sets)
  - Use better compilers
How can I Make the Program Run Faster?

\[ N \times CPI \times \left(\frac{1}{f}\right) \]

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC – complex instruction sets)
  - Use better compilers

- **Use fewer cycles to perform each instruction**
  - Simpler instructions (RISC – reduced instruction sets)
  - Use multiple units/ALUs/cores in parallel
How can I Make the Program Run Faster?

\[ N \times \text{CPI} \times (1/f) \]

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC – complex instruction sets)
  - Use better compilers

- **Use fewer cycles to perform each instruction**
  - Simpler instructions (RISC – reduced instruction sets)
  - Use multiple units/ALUs/cores in parallel

- **Increase the clock frequency**
  - Find a ‘newer’ technology to manufacture
  - Redesign time critical components
  - Adopt pipelining
Performance Analysis of Single-Cycle vs. Multi-Cycle Designs
Single-Cycle Performance

- $T_C$ is limited by the critical path ($1w$)
Single-Cycle Performance

- Single-cycle critical path:
  - $T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$

- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$
## Single-Cycle Performance Example

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clock-to-Q</td>
<td>( t_{pcq_{PC}} )</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>( t_{setup} )</td>
<td>20</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>( t_{mux} )</td>
<td>25</td>
</tr>
<tr>
<td>ALU</td>
<td>( t_{ALU} )</td>
<td>200</td>
</tr>
<tr>
<td>Memory read</td>
<td>( t_{mem} )</td>
<td>250</td>
</tr>
<tr>
<td>Register file read</td>
<td>( t_{RFread} )</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>( t_{RFsetup} )</td>
<td>20</td>
</tr>
</tbody>
</table>

\[ T_c = \]
### Single-Cycle Performance Example

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clock-to-Q</td>
<td>$t_{pcq_PC}$</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>$t_{\text{setup}}$</td>
<td>20</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>$t_{\text{mux}}$</td>
<td>25</td>
</tr>
<tr>
<td>ALU</td>
<td>$t_{\text{ALU}}$</td>
<td>200</td>
</tr>
<tr>
<td>Memory read</td>
<td>$t_{\text{mem}}$</td>
<td>250</td>
</tr>
<tr>
<td>Register file read</td>
<td>$t_{\text{RFread}}$</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>$t_{\text{RFsetup}}$</td>
<td>20</td>
</tr>
</tbody>
</table>

\[
T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{\text{RFread}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{RFsetup}} \\
= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps} \\
= 925 \text{ ps}
\]
Example:

For a program with 100 billion instructions executing on a single-cycle MIPS processor:
Example:

For a program with 100 billion instructions executing on a single-cycle MIPS processor:

\[
\text{Execution Time} = \# \text{ instructions} \times \text{CPI} \times T_c \\
= (100 \times 10^9)(1)(925 \times 10^{-12} \text{ s}) \\
= 92.5 \text{ seconds}
\]
Multi-Cycle Performance: CPI

- Instructions take different number of cycles:
  - 3 cycles: \texttt{beq, j}
  - 4 cycles: \texttt{R-Type, sw, addi}
  - 5 cycles: \texttt{lw} \textbf{Realistic?}

- CPI is weighted average, e.g. SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

\textit{Average CPI} = (0.11 + 0.02) 3 + (0.52 + 0.10) 4 + (0.25) 5 
= 4.12
Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

\[ T_c = \]
Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

\[ T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \]
## Multi-Cycle Performance Example

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clock-to-Q</td>
<td>t_{pcq_{PC}}</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>t_{setup}</td>
<td>20</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>t_{mux}</td>
<td>25</td>
</tr>
<tr>
<td>ALU</td>
<td>t_{ALU}</td>
<td>200</td>
</tr>
<tr>
<td>Memory read</td>
<td>t_{mem}</td>
<td>250</td>
</tr>
<tr>
<td>Register file read</td>
<td>t_{RFread}</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>t_{RFsetup}</td>
<td>20</td>
</tr>
</tbody>
</table>

\[ T_c = \]
## Multi-Cycle Performance Example

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clock-to-Q</td>
<td>$t_{pcq_PC}$</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>$t_{setup}$</td>
<td>20</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>$t_{mux}$</td>
<td>25</td>
</tr>
<tr>
<td>ALU</td>
<td>$t_{ALU}$</td>
<td>200</td>
</tr>
<tr>
<td>Memory read</td>
<td>$t_{mem}$</td>
<td>250</td>
</tr>
<tr>
<td>Register file read</td>
<td>$t_{RF_read}$</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>$t_{RF_setup}$</td>
<td>20</td>
</tr>
</tbody>
</table>

\[
T_c = t_{pcq\_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \\
= [30 + 25 + 250 + 20] \text{ ps} \\
= 325 \text{ ps}
\]
Multi-Cycle Performance Example

- For a program with 100 billion instructions executing on a multi-cycle MIPS processor
  - CPI = 4.12
  - $T_c = 325 \text{ ps}$
- **Execution Time**
  \[
  \text{Execution Time} = (\# \text{ instructions}) \times \text{CPI} \times T_c \\
  = (100 \times 10^9)(4.12)(325 \times 10^{-12}) \\
  = 133.9 \text{ seconds}
  \]
- This is slower than the single-cycle processor (92.5 seconds). *Why?*
  - Did we break the stages in a balanced manner?
  - Overhead of register setup/hold paid many times
  - How would the results change with different assumptions on memory latency and instruction mix?
Review: Single-Cycle MIPS Processor
Review: Single-Cycle MIPS FSM

- Single-cycle machine

![Diagram showing single-cycle machine with Combinational Logic, AS', Sequential Logic (State), and AS: Architectural State.]

AS: Architectural State
Review: Multi-Cycle MIPS Processor

[Diagram of a MIPS processor with various components labeled.]
Review: Multi-Cycle MIPS FSM

What is the shortcoming of this design?

What does this design assume about memory?
What If Memory Takes > One Cycle?

- Stay in the same “memory access” state until memory returns the data
- “Memory Ready?” bit is an input to the control logic that determines the next state
Backup Slides on Microprogrammed Multi-Cycle Microarchitectures
Microprogrammed Control Terminology

- Control signals associated with the current state
  - Microinstruction

- Act of transitioning from one state to another
  - Determining the next state and the microinstruction for the next state
  - Microsequencing

- Control store stores control signals for every possible state
  - Store for microinstructions for the entire FSM

- Microsequencer determines which set of control signals will be used in the next clock cycle (i.e., next state)
Lectures on Microprogrammed Designs

- Design of Digital Circuits, Spring 2018, Lecture 13
  - Microprogramming (ETH Zürich, Spring 2018)
  - https://www.youtube.com/watch?v=u4GhShuBP3Y&list=PL5Q2soXY2Zi_QedyPWtRmFUJ2F8DdYP7l&index=13

- Computer Architecture, Spring 2013, Lecture 7
  - Microprogramming (CMU, Spring 2013)
  - https://www.youtube.com/watch?v=_igvSl5h8cs&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=7

SAFARI

https://www.youtube.com/onurmutlulectures
Another Example: Microprogrammed Multi-Cycle Microarchitecture
How Do We Implement This?


---

THE BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

By M. V. Wilkes, M.A., Ph.D., F.R.A.S.

- An elegant implementation:
  - The concept of microcoded/microprogrammed machines
Recall: A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into “states”
  - A stage in the instruction processing cycle can take multiple states

- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state

- The behavior of the entire processor is specified fully by a finite state machine

- In a state (clock cycle), control signals control two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle
Microprogrammed Control Terminology

- Control signals associated with the current state
  - Microinstruction

- Act of transitioning from one state to another
  - Determining the next state and the microinstruction for the next state
  - Microsequencing

- Control store stores control signals for every possible state
  - Store for microinstructions for the entire FSM

- Microsequencer determines which set of control signals will be used in the next clock cycle (i.e., next state)
If, somehow, the instruction inadvertently contained IR[15:12] = 1010 or 1011, the
What Happens In A Clock Cycle?

- The control signals (microinstruction) for the current state control two things:
  - Processing in the data path
  - Generation of control signals (microinstruction) for the next cycle
  - *See Supplemental Figure 1 (next-next slide)*

- Datapath and microsequencer operate concurrently

- Question: why not generate control signals for the current cycle in the current cycle?
  - This could lengthen the clock cycle
  - Why could it lengthen the clock cycle?
  - *See Supplemental Figure 2*
Read P&P Revised Appendix C
On website

Microarchitecture of the LC-3b, major components
A Clock Cycle

1. Processing in Data Path for Cycle N
2. Generation of Control Signals for Cycle N+1

Latch
1) Results of current cycle N
2) Control signals needed for the next cycle N+1

Fig. 1
A Bad Clock Cycle!

Alternative - A BAD ONE!

0. Generation of Control Signals for Cycle N
1. Processing for Datapath for Cycle N

Step 1 is dependent on Step 0
If Step 0 takes non-zero time (it does!), clock cycle increases unnecessarily

→ Violates the "Critical Path Design" principle

Fig 2
Microarchitecture of the LC-3b, major components

Control Signals

Control

Data Path

Memory, I/O

Data, Inst.

Data

Addr

R

IR[15:11]

BEN

Microarchitecture of the LC-3b Control and Datapath

Read P&P Revised Appendix C
On website

A Simple LC-3b Control and Datapath
What Determines Next-State Control Signals?

- What is happening in the current clock cycle
  - See the 9 control signals coming from “Control” block
    - What are these for?

- The instruction that is being executed
  - IR[15:11] coming from the Data Path

- Whether the condition of a branch is met, if the instruction being processed is a branch
  - BEN bit coming from the datapath

- Whether the memory operation is completing in the current cycle, if one is in progress
  - R bit coming from memory
A Simple LC-3b Control and Datapath

Microarchitecture of the LC-3b, major components

1. J[5:0], COND[1:0], and IRD—9 bits of control signals provided by the current instruction.
2. Instr[15:12], which identifies the opcode, and Instr[11:11], which differentiates between JSR from JSR (i.e., the address is in the register file for the target of the subroutine call).
3. BEN to indicate whether or not a BR should be taken.
The State Machine for Multi-Cycle Processing

- The behavior of the LC-3b uarch is completely determined by
  - the 35 control signals and
  - additional 7 bits that go into the control logic from the datapath

- 35 control signals completely describe the state of the control structure

- We can completely describe the behavior of the LC-3b as a state machine, i.e. a directed graph of
  - Nodes (one corresponding to each state)
  - Arcs (showing flow from each state to the next state(s))
An LC-3b State Machine

- Patt and Patel, Revised Appendix C, Figure C.2

- Each state must be uniquely specified
  - Done by means of *state variables*

- 31 distinct states in this LC-3b state machine
  - Encoded with 6 state variables

- Examples
  - State 18,19 correspond to the beginning of the instruction processing cycle
  - Fetch phase: state 18, 19 \(\rightarrow\) state 33 \(\rightarrow\) state 35
  - Decode phase: state 32
Figure C.2: A state machine for the LC-3b

NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on MAR[0]
The FSM Implements the LC-3b ISA

- P&P Appendix A (revised):
LC-3b State Machine: Some Questions

- How many cycles does the fastest instruction take?
- How many cycles does the slowest instruction take?
- Why does the BR take as long as it takes in the FSM?
- What determines the clock cycle time?
LC-3b Datapath

- Patt and Patel, Revised Appendix C, Figure C.3

- Single-bus datapath design
  - At any point only one value can be “gated” on the bus (i.e., can be driving the bus)
  - **Advantage**: Low hardware cost: one bus
  - **Disadvantage**: Reduced concurrency – if instruction needs the bus twice for two different things, these need to happen in different states

- Control signals (26 of them) determine what happens in the datapath in one clock cycle
  - Patt and Patel, Revised Appendix C, Table C.1
Remember the MIPS datapath
<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Signal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD.MAR/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.MDR/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.IR/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.BEN/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.REG/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.CC/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.PC/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>GatePC/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateMDR/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateALU/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateMARMUX/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateSHF/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>PCMUX/2:</td>
<td>PC+2 ; select pc+2</td>
</tr>
<tr>
<td></td>
<td>BUS ; select value from bus</td>
</tr>
<tr>
<td></td>
<td>ADDER ; select output of address adder</td>
</tr>
<tr>
<td>DRMUX/1:</td>
<td>11.9 ; destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>R7 ; destination R7</td>
</tr>
<tr>
<td>SR1MUX/1:</td>
<td>11.9 ; source IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>8.6 ; source IR[8:6]</td>
</tr>
<tr>
<td>ADDR1MUX/1:</td>
<td>PC, BaseR</td>
</tr>
<tr>
<td>ADDR2MUX/2:</td>
<td>ZERO ; select the value zero</td>
</tr>
<tr>
<td></td>
<td>offset6 ; select SEXT[IR[5:0]]</td>
</tr>
<tr>
<td></td>
<td>PCOffset9 ; select SEXT[IR[8:0]]</td>
</tr>
<tr>
<td></td>
<td>PCOffset11 ; select SEXT[IR[10:0]]</td>
</tr>
<tr>
<td>MARMUX/1:</td>
<td>7.0 ; select LSHF(ZEXT[IR[7:0]],1)</td>
</tr>
<tr>
<td></td>
<td>ADDER ; select output of address adder</td>
</tr>
<tr>
<td>ALUK/2:</td>
<td>ADD, AND, XOR, PASSA</td>
</tr>
</tbody>
</table>

Table C.1: Data path control signals
LC-3b Datapath: Some Questions

- How does instruction fetch happen in this datapath according to the state machine?

- What is the difference between gating and loading?
  - Gating: Enable/disable an input to be connected to the bus
    - Combinational: during a clock cycle
  - Loading: Enable/disable an input to be written to a register
    - Sequential: e.g., at a clock edge (assume at the end of cycle)

- Is this the smallest hardware you can design?
LC-3b Microprogrammed Control Structure

- Patt and Patel, Appendix C, Figure C.4

- Three components:
  - Microinstruction, control store, microsequencer

- **Microinstruction**: control signals that control the datapath (26 of them) and help determine the next state (9 of them)

- Each microinstruction is stored in a *unique location* in the control store (a special memory structure)

- **Unique location**: address of the state corresponding to the microinstruction
  - Remember each state corresponds to one microinstruction

- Microsequencer determines the address of the next microinstruction (i.e., next state)
If, somehow, the instruction inadvertently contained IR[15:12] = 1010 or 1011, the
In both cases, the use of microcode is determined by the instruction word. In Tables C.1 and C.2, the
DR, SR1, and BEN. The remaining signal, R, is a signal generated by the memory in order to allow the

0,0,IR[15:12] 6

Address of Next State

IRD


Branch  Ready  Addr. Mode

BEN  R  IR[11]

COND1  COND0

IR: Address of Next State

IRD:
### APPENDIX C. THE MICROARCHITECTURE OF THE LC-3B, BASIC MACHINE

![Figure C.7: Specification of the control store](image-url)
LC-3b Microsequencer

- Patt and Patel, Appendix C, Figure C.5

- **The purpose of the microsequencer** is to determine the address of the next microinstruction (i.e., next state)
  - Next state could be conditional or unconditional

- Next state address depends on 9 control signals (plus 7 data signals)

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Signal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>J/6:</td>
<td></td>
</tr>
<tr>
<td>COND/2:</td>
<td>COND₀ ;Unconditional</td>
</tr>
<tr>
<td></td>
<td>COND₁ ;Memory Ready</td>
</tr>
<tr>
<td></td>
<td>COND₂ ;Branch</td>
</tr>
<tr>
<td></td>
<td>COND₃ ;Addressing Mode</td>
</tr>
<tr>
<td>IRD/1:</td>
<td>NO, YES</td>
</tr>
</tbody>
</table>

Table C.2: Microsequencer control signals
APPENDIX C. THE MICROARCHITECTURE OF THE LC-3B, BASIC MACHINE

IRD
Address of Next State


0,0,IR[15:12]

6

IRD

Branch
Ready
Addr. Mode

BEN
R
IR[11]

COND1 COND0

The microcode sequence of the LC-3B baseline machine.

The address of the next state is determined based on the branch conditions and the mode.

In both cases, the micro-coded sequence of the machine is as follows:

1. The address of the next state is calculated based on the branch conditions and mode.
2. The micro-coded sequence is generated for the next state.
3. The micro-coded sequence is used to generate the necessary signals for the next state.

The remaining signal, R, is a signal generated by the memory in order to allow the

To generate DR, SR1, and BEN, the micro-coded sequence is as follows:

1. The address of the next state is calculated based on the branch conditions and mode.
2. The micro-coded sequence is generated for the next state.
3. The micro-coded sequence is used to generate the necessary signals for the next state.
4. The remaining signal, R, is a signal generated by the memory in order to allow the

The remaining signal, R, is a signal generated by the memory in order to allow the
The Microsequencer: Some Questions

- When is the IRD signal asserted?
- What happens if an illegal instruction is decoded?
- What are condition (COND) bits for?
- How is variable latency memory handled?
- How do you do the state encoding?
  - Minimize number of state variables (~ control store size)
  - Start with the 16-way branch
  - Then determine constraint tables and states dependent on COND
An Exercise in Microprogramming
Handouts

- 7 pages of Microprogrammed LC-3b design

A Simple LC-3b Control and Datapath

Microarchitecture of the LC-3b, major components

1. J[5:0], COND[1:0], and IRD—9 bits of control signals provided by the current instruction.
2. inst[15:12], which indicates the opcode, and inst[11:1] which defines the address for the target of the subroutine call.
3. BEN to indicate whether or not a BR should be taken.

If the LC-3b instruction is a BR, whether the conditional instruction for the branch happens.

The addressing mode for the target of the subroutine call is specified.

26 R Memory, I/O
23 Addr Data Path
16 Data, Inst. Addr
16 Data
3
Figure C.2: A state machine for the LC-3b

NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on MAR[0]
A Simple Datapath Can Become Very Powerful
Fill in the microinstructions for the 7 states for LDW
111

IR[11:9] → DR

DRMUX

IR[11:9] → DR

111

IR[11:9] → DR

DRMUX

IR[11:9] → DR

IR[8:6] → SR1

SR1MUX

IR[11:9] → DR

Logic

IR[11:9] → Logic

N

Z

P

BEN

(c)
<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Signal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD.MAR/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.MDR/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.IR/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.BEN/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.REG/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.CC/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.PC/1:</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>GatePC/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateMDR/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateALU/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateMARMUX/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateSHF/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>PCMUX/2:</td>
<td>PC+2</td>
</tr>
<tr>
<td></td>
<td>BUS</td>
</tr>
<tr>
<td></td>
<td>ADDER</td>
</tr>
<tr>
<td>DRMUX/1:</td>
<td>11.9</td>
</tr>
<tr>
<td></td>
<td>R7</td>
</tr>
<tr>
<td>SR1MUX/1:</td>
<td>11.9</td>
</tr>
<tr>
<td></td>
<td>8.6</td>
</tr>
<tr>
<td>ADDR1MUX/1:</td>
<td>PC, BaseR</td>
</tr>
<tr>
<td>ADDR2MUX/2:</td>
<td>ZERO</td>
</tr>
<tr>
<td></td>
<td>offset6</td>
</tr>
<tr>
<td></td>
<td>PCoffset9</td>
</tr>
<tr>
<td></td>
<td>PCoffset11</td>
</tr>
<tr>
<td>MARMUX/1:</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>ADDER</td>
</tr>
<tr>
<td>ALUK/2:</td>
<td>ADD, AND, XOR, PASSA</td>
</tr>
<tr>
<td>MIO.EN/1:</td>
<td>NO, YES</td>
</tr>
<tr>
<td>R.W/1:</td>
<td>RD, WR</td>
</tr>
<tr>
<td>DATA.SIZE/1:</td>
<td>BYTE, WORD</td>
</tr>
<tr>
<td>LSHF1/1:</td>
<td>NO, YES</td>
</tr>
</tbody>
</table>

Table C.1: Data path control signals
If, somehow, the instruction inadvertently contained IR[15:12] = 1010 or 1011, the
APPENDIX C. THE MICROARCHITECTURE OF THE LC-3B, BASIC MACHINE

The microcode of the LC-3B basic machine uses signals to control the operation of the machine. The signals include:

- **COND1**: Condition 1 signal.
- **COND0**: Condition 0 signal.
- **BEN**: Branch Enable signal.
- **R**: Ready signal.
- **IR[11]**: Instruction Register 11 signal.
- **Branch**: Branch signal.
- **Ready**: Ready signal.
- **Addr. Mode**: Address Mode signal.

These signals are generated based on the state of the machine and the status of the conditions. For example, the **BEN** signal is generated when the condition is met, and the **R** signal is generated when the machine is ready to proceed. The **Addr. Mode** signal determines the addressing mode used for the next instruction.

The **IRD** signal is used to indicate the address of the next state, and the **IRD** signal is used to indicate the instruction to be executed.

In the context of the LC-3B basic machine, these signals are used to control the operation of the machine, ensuring that the correct instructions are executed in the correct order.

Address of Next State
<table>
<thead>
<tr>
<th>Code</th>
<th>RD</th>
<th>RD(\uparrow)</th>
<th>LD</th>
<th>LD(\uparrow)</th>
<th>R</th>
<th>R(\uparrow)</th>
<th>ADD</th>
<th>ADD(\uparrow)</th>
<th>SUB</th>
<th>SUB(\uparrow)</th>
<th>ADDX</th>
<th>ADDX(\uparrow)</th>
</tr>
</thead>
</table>

![Figure C.7: Specification of the control store](image_url)

- State 0
- State 1
- State 2
- State 3
- State 4
- State 5
- State 6
- State 7
- State 8
- State 9
- State 10
- State 11
- State 12
- State 13
- State 14
- State 15
- State 16
- State 17
- State 18
- State 19
- State 20
- State 21
- State 22
- State 23
- State 24
- State 25
- State 26
- State 27
- State 28
- State 29
- State 30
- State 31
- State 32
- State 33
- State 34
- State 35
- State 36
- State 37
- State 38
- State 39
- State 40
- State 41
- State 42
- State 43
- State 44
- State 45
- State 46
- State 47
- State 48
- State 49
- State 50
- State 51
- State 52
- State 53
- State 54
- State 55
- State 56
- State 57
- State 58
- State 59
- State 60
- State 61
- State 62
- State 63

240
End of the Exercise in Microprogramming
Variable-Latency Memory

- The ready signal (R) enables memory read/write to execute correctly
  - Example: transition from state 33 to state 35 is controlled by the R bit asserted by memory when memory data is available.

- Could we have done this in a single-cycle microarchitecture?

- What did we assume about memory and registers in a single-cycle microarchitecture?
The Microsequencer: Advanced Questions

- What happens if the machine is interrupted?

- What if an instruction generates an exception?

- How can you implement a complex instruction using this control structure?
  - Think REP MOVS instruction in x86
    - string copy of N elements starting from address A to address B
The Power of Abstraction

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction: microprogramming.

- The designer can translate any desired operation to a sequence of microinstructions.

- All the designer needs to provide is:
  - The sequence of microinstructions needed to implement the desired operation.
  - The ability for the control logic to correctly sequence through the microinstructions.
  - Any additional datapath elements and control signals needed (no need if the operation can be “translated” into existing control signals).
Let’s Do Some More Microprogramming

- Implement REP MOV$ in the LC-3b microarchitecture

- What changes, if any, do you make to the
  - state machine?
  - datapath?
  - control store?
  - microsequencer?

- Show all changes and microinstructions

- Optional HW Assignment
x86 REP MOVVS (String Copy) Instruction

REP MOVVS (DEST SRC)

IF AddressSize = 16
    THEN
        Use CX for CountReg;
    ELSE IF AddressSize = 64 and REX.W used
        THEN Use RCX for CountReg; Fl;
    ELSE
        Use ECX for CountReg;
    Fl;
    WHILE CountReg ≠ 0
        DO
            Service pending interrupts (if any);
            Execute associated string instruction;
            CountReg ← (CountReg - 1);
            IF CountReg = 0
                THEN exit WHILE loop; Fl;
            IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
                or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
                THEN exit WHILE loop; Fl;
        OD;
    Fl;
    Fl;

How many instructions does this take in MIPS ISA?
How many microinstructions does this take to add to the LC-3b microarchitecture?
Aside: Alignment Correction in Memory

- **Unaligned accesses**

- LC-3b has byte load and byte store instructions that move data not aligned at the word-address boundary
  - Convenience to the programmer/compiler

- How does the hardware ensure this works correctly?
  - Take a look at state 29 for LDB
  - States 24 and 17 for STB
  - Additional logic to handle unaligned accesses

- P&P, Revised Appendix C.5
Aside: Memory Mapped I/O

- Address control logic determines whether the specified address of LDW and STW are to memory or I/O devices

- Correspondingly enables memory or I/O devices and sets up muxes

- An instance where the final control signals of some datapath elements (e.g., MEM.EN or INMUX/2) **cannot** be stored in the control store
  - These signals are dependent on memory address

- P&P, Revised Appendix C.6
Advantages of Microprogrammed Control

- Allows a very simple design to do powerful computation by controlling the datapath (using a sequencer)
  - High-level ISA translated into microcode (sequence of u-instructions)
  - Microcode (u-code) enables a minimal datapath to emulate an ISA
  - Microinstructions can be thought of as a user-invisible ISA (u-ISA)

- Enables easy extensibility of the ISA
  - Can support a new instruction by changing the microcode
  - Can support complex instructions as a sequence of simple microinstructions (e.g., REP MOVS, INC [MEM])

- Enables update of machine behavior
  - A buggy implementation of an instruction can be fixed by changing the microcode in the field
    - Easier if datapath provides ability to do the same thing in different ways
Update of Machine Behavior

- **The ability to update/patch microcode in the field** (after a processor is shipped) enables
  - Ability to add new instructions without changing the processor!
  - Ability to “fix” buggy hardware implementations

- **Examples**
  - IBM 370 Model 145: microcode stored in main memory, can be updated after a reboot
  - IBM System z: Similar to 370/145.
  - B1700 microcode can be updated while the processor is running
    - User-microprogrammable machine!
Multi-Cycle vs. Single-Cycle uArch

- Advantages
- Disadvantages
- For you to fill in