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

\[
\begin{align*}
\text{AS} &= \text{Architectural (programmer visible) state at the beginning of a clock cycle} \\
\text{Process instruction in one clock cycle} \\
\text{AS'} &= \text{Architectural (programmer visible) state at the end of a clock cycle}
\end{align*}
\]
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

![Diagram showing the relationship between combinational logic and sequential logic in a single-cycle machine. The diagram includes labels for Combinational Logic, Sequential Logic (State), and AS (Architectural State).]
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
if MEM[PC] == ADDI rt rs immediate
GPR[rt] ← GPR[rs] + sign-extend (immediate)
PC ← PC + 4
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 \(\text{MEM}[\text{PC}] == j \text{ immediate}_{26}\)

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

\(\text{PC} \leftarrow \text{target}\)

\(^{†}\text{This is the incremented PC}\)
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 \( \text{MEM}[\text{PC}] == \text{jal} \) immediate\(_{26}\)
      
      - \( $ra \leftarrow \text{PC} + 4 \)
      
      - \( \text{target} = \{ \text{PC}^{\dagger}[31:28], \text{immediate}_{26}, 2^'b00 \} \)
      
      - \( \text{PC} \leftarrow \text{target} \)

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

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

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


- On the course website
Conditional Branch Instructions

- beq (Branch if Equal)

\[
\text{beq\ }$s0,$s1,\text{ offset } #$s0=rs,$s1=rt
\]

<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)

\[
\text{if } \text{MEM[PC]} == \text{beq rs rt \text{ immediate}}_{16}
\]
\[
\text{target } = \text{PC}^\dagger + \text{sign-extend(\text{immediate})} \times 4
\]
\[
\text{if } \text{GPR[rs]} == \text{GPR[rt]} \text{ then } \text{PC } \leftarrow \text{target}
\]
\[
\text{else } \text{PC } \leftarrow \text{PC } + \text{4}
\]

- Variations: beq, bne, blez, bgtz

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

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>

R-Type

<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>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></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>
<td></td>
</tr>
</tbody>
</table>

I-Type

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

J-Type

Consider

- All R-type and I-type ALU instructions
- $\text{lw}$ and $\text{sw}$
- $\text{beq}$, $\text{bne}$, $\text{blez}$, $\text{bgtz}$
- $\text{j}$, $\text{jr}$, $\text{jal}$, $\text{jalr}$
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) \land (\text{opcode} != \text{BEQ}) \land (\text{opcode} != \text{BNE}))</td>
</tr>
<tr>
<td><strong>MemtoReg</strong></td>
<td>Steer ALU result to GPR write port</td>
<td>steer memory load 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}) \land (\text{opcode} != \text{Bxx}) \land (\text{opcode} != \text{J}) \land (\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 return load value</td>
<td>( \text{opcode}==\text{LW} )</td>
</tr>
<tr>
<td><strong>MemWrite</strong></td>
<td>Memory write disabled</td>
<td>Memory write enabled</td>
<td>( \text{opcode}==\text{SW} )</td>
</tr>
<tr>
<td><strong>PCSrc(_1)</strong></td>
<td>According to ( \text{PCSrc}_2 )</td>
<td>next PC is based on 26-bit immediate jump target</td>
<td>( \text{(opcode}==\text{J}) \ | \ \ (\text{opcode}==\text{JAL}) )</td>
</tr>
<tr>
<td><strong>PCSrc(_2)</strong></td>
<td>next PC = PC + 4</td>
<td>next PC is based on 16-bit immediate branch target</td>
<td>( \text{(opcode}==\text{Bxx}) \ &amp; \ &amp; \ \text{“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.]**
Some control signals are dependent on the processing of data.

Branch (Not Taken)

**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.
Jump
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.
Review: Complete Single-Cycle Processor

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

JAL, JR, JALR omitted
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

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

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 register read

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

```
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 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: \texttt{lw} address

**STEP 4:** Compute the memory 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>

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

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

**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>
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>
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 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**
  - \{CPI\} \times \{clock cycle time\}
  - CPI: Number of cycles it takes to execute an instruction

- **Execution time of an entire program**
  - Sum over all instructions \([\{CPI\} \times \{clock cycle time\}]\)
  - \{# of instructions\} \times \{Average CPI\} \times \{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
  ▪ *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
    - *Cycles per Instruction* = *CPI*

- How long is one clock cycle?
  - The critical path determines how much time one cycle requires = *clock period*.
  - 1/clock period = *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**

\[
N \times CPI \times \frac{1}{f} = N \times CPI \times T \text{ seconds}
\]
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
  1. Instruction fetch (IF)
  - Decode
  2. Instruction decode and register operand fetch (ID/RF)
  - Evaluate Address
  3. Execute/Evaluate memory address (EX/AG)
  - Fetch Operands
  4. Memory operand fetch (MEM)
  - Execute
  5. Store/writeback result (WB)
  - Store Result

- Do each of the above phases take the same time (latency) for all instructions?
Let's Find the Critical Path

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
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 resources</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td></td>
<td>50</td>
<td>400</td>
</tr>
<tr>
<td>I-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td></td>
<td>50</td>
<td>400</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></td>
<td>550</td>
</tr>
<tr>
<td>Branch</td>
<td>200</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td>350</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
R-Type and I-Type ALU
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 does not work (e.g. common instructions)
  - 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
    - ...

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

Multi-Cycle Microarchitecture

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

1. Step 1: Process part of instruction in one clock cycle

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

\[ \cdots \]

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


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
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 the 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 the 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

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

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 register read

**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: \texttt{lw} immediate

\underline{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></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 memory read

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: \texttt{l}w write register

\begin{center}
\textbf{I-Type}
\end{center}

\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}
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:
    \[ \text{Target Address} = (\text{sign-extended immediate } \ll 2) + (PC+4) \]
Complete Multi-Cycle Processor
Let’s Construct the Multi-Cycle Control Logic
Control Unit

Main Controller (FSM)

- MemtoReg
- RegDst
- IorD
- PCSrc
- ALUSrcB
- ALUSrcA
- IRWrite
- MemWrite
- PCWrite
- Branch
- RegWrite

ALUOp

ALU Decoder

ALUControl

OpCode

Register Enables

Multiplexer Selects
Main Controller FSM: Fetch

S0: Fetch
Reset
Main Controller FSM: Fetch

S0: Fetch

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

Reset
Main Controller FSM: Decode

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
Reset
S0: Fetch
S1: Decode
Main Controller FSM: Address Calculation

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

S1: Decode
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Reset

S2: MemAddr
Op = LW
or
Op = SW

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S2: MemAdr
Op = LW
or
Op = SW
Main Controller FSM: \texttt{lw}

- **S0: Fetch**
  - Reset
  - \texttt{IorD = 0}
  - \texttt{AluSrcA = 0}
  - \texttt{ALUSrcB = 01}
  - \texttt{ALUOp = 00}
  - \texttt{PCSrc = 0}
  - \texttt{IRWrite}
  - \texttt{PCWrite}

- **S1: Decode**
  - \texttt{Op = LW}
  - \texttt{RegDst = 0}
  - \texttt{MemtoReg = 1}
  - \texttt{RegWrite}

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

- **S3: MemRead**
  - \texttt{IorD = 1}

- **S4: Mem Writeback**
  - \texttt{RegDst = 0}
  - \texttt{MemtoReg = 1}
  - \texttt{RegWrite}
Main Controller FSM: sw

S0: Fetch

Reset

S1: Decode

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

S2: MemAddr

Op = LW
or
Op = SW

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S3: MemRead

Op = LW

IorD = 1

S4: MemWrite

RegDst = 0
MemtoReg = 1
RegWrite

S5: MemWrite

Op = LW
or
Op = SW

IorD = 1
MemWrite
Main Controller FSM: R-Type

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

S1: Decode
- Op = LW or SW
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 00
- IorD = 1
- RegDst = 0
- MemtoReg = 1
- RegWrite

S2: MemAdr
- Op = LW
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00
- IorD = 1
- MemWrite

S3: MemRead
- Op = SW
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 10
- MemWrite

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

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

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

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

```
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
ALUSrcA = 0
ALUSrcB = 11
ALUOp = ...
IorD = 1
MemWrite
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
RegDst = 0
MemtoReg = 1
RegWrite
S4: MemWriteback
S5: MemWrite
S6: Execute
S7: ALU Writeback
S8: Branch
Op = BEQ
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch
S0: Fetch
S1: Decode
S2: MemAddr
S3: MemRead
S4: MemWriteback
S5: MemWrite
S6: Execute
S7: ALU Writeback
S8: Branch
```

Op = LW
or
Op = SW
Op = R-type
Op = BEQ
Op = LW
Op = SW

Complete Multi-Cycle Controller FSM

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

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

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

S3: MemRead
- Op = LW

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

S5: MemWrite
- Op = R-type

S6: Execute
- Op = BEQ

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

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

Op = LW
or
Op = SW
Main Controller FSM: addi
Main Controller FSM: `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`

- **S3: MemRead**
  - `Op = LW`
  - `Op = SW`

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

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

- **S6: Execute**
  - `Op = R-type`
  - `Op = ADDI`

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

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

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

- **S10: ADDI Writeback**
  - `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**
  - $ALUSrcA = 1$
  - $ALUSrcB = 10$
  - $ALUOp = 00$

- **S3: MemRead**
  - $Op = LW$

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

- **S5: Mem Writeback**
  - $IorD = 1$
  - $MemWrite$
  - $RegWrite$

- **S6: Execute**
  - $Op = R-type$
  - $Op = BEQ$
  - $PCSrc = 01$
  - $Branch$

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

- **S8: Branch**
  - $Op = ADDI$

- **S9: ADDI Execute**
  - $ALUSrcA = 1$
  - $ALUSrcB = 00$
  - $ALUOp = 01$
  - $PCSrc = 01$
  - $Branch$

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

- **S11: Jump**
  - $Op = J$
Control FSM: j

S0: Fetch
Reset

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

S2: MemAdr
Op = LW
or
Op = SW

S3: MemRead
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S4: MemWriteback
RegDst = 0
MemtoReg = 1
RegWrite

S5: MemWrite
IorD = 1
MemWrite

S6: Execute
Op = R-type

S7: ALU Writeback
RegDst = 1
MemtoReg = 0
RegWrite

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

S9: ADDI
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

S10: ADDI
RegDst = 0
MemtoReg = 0
RegWrite

S11: Jump
Pcsrc = 10
PCWrite

Op = J
Review: Single-Cycle MIPS Processor
Review: Single-Cycle MIPS FSM

- Single-cycle machine

- **AS**: Architectural State

- **AS'**: State Logic

- Combinational Logic

- Sequential Logic (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
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)
Detailed Lectures on Microprogramming

- **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](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=PL5PHm2jkkXmidJ0d59REog9jDnPDTG6IJ&index=7](https://www.youtube.com/watch?v=_igvSl5h8cs&list=PL5PHm2jkkXmidJ0d59REog9jDnPDTG6IJ&index=7)
Backup Slides on Single-Cycle Uarch for Your Own Study

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

**STEP 1:** Fetch instruction

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

\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: \textit{lw} register read

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

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

\begin{center}
\textbf{I-Type}
\end{center}

\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` 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

```latex
\text{lw} \ \$s3, \ 1\$(\theta) \ \# \ 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{lw} memory read

\textbf{STEP 5:} Read from memory and write back to register file

\begin{align*}
lw & \quad \texttt{$s3, 1($0)} & \# \text{read memory word } 1 \text{ into } \texttt{$s3}$
\end{align*}
Single-Cycle Datapath: lw PC increment

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

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

### 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: sw

- Write data in rt to memory

sw $t7, 44($0)  # write t7 into memory address 44

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: R-type Instructions

- Read from rs and rt, write ALUResult to register file

\[ \text{add } t, b, c \quad \# t = b + c \]

**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

- **beq** $s0, s1, target  # branch is taken

- Determine whether values in rs and rt are equal
- Calculate $BTA = (\text{sign-extended immediate } \ll 2) + (PC+4)$
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

<table>
<thead>
<tr>
<th>ALUOp</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 field</td>
</tr>
<tr>
<td>11</td>
<td>n/a</td>
</tr>
</tbody>
</table>
**ALU Does the Real Work in a Processor**

![ALU Diagram](image)

<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</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>
Control Unit: ALU Decoder

<table>
<thead>
<tr>
<th>ALUOp_{1:0}</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_{1:0}</th>
<th>Funct</th>
<th>ALUControl_{2:0}</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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>$\text{Op}_{5:0}$</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>0000000</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&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>
<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>funct</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>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>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>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>1</td>
</tr>
</tbody>
</table>

![Control Unit Diagram](image)
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>0000000</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&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>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>X</td>
<td>XX</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?

\[ N \times CPI \times (1/f) \]
How can I Make the Program Run Faster?

N x CPI x (1/f)

- Reduce the number of instructions
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
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)
  - Use better compilers

- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel
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)
  - Use better compilers

- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - 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>$t_{\text{pcq_PC}}$</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>

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

\[ T_c = t_{pcq\_PC} + 2t_{mem} + t_{RF\_read} + t_{mux} + t_{ALU} + t_{RF\_setup} \]

\[ = [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps} \]

\[ = 925 \text{ ps} \]

<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>
Single-Cycle Performance Example

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: `beq, j`
  - 4 cycles: `R-Type, sw, addi`
  - **5 cycles: `lw`** Realistic?

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

- **Average CPI** = \((0.11 + 0.02) \times 3 + (0.52 + 0.10) \times 4 + (0.25) \times 5\) 
  = 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_{\text{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{RF_read}} )</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>( t_{\text{RF_setup}} )</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>$t_{RFread}$</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>$t_{RFsetup}$</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
  - $T_c = 325$ 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
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
Backup Slides on Microprogrammed Multi-Cycle Microarchitectures
These Slides Are Covered in A Past Lecture

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](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=PL5PHm2jkkXmidJQOd59REog9jDnPDTG6IJ&index=7](https://www.youtube.com/watch?v=_igvSl5h8cs&list=PL5PHm2jkkXmidJQOd59REog9jDnPDTG6IJ&index=7)
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
Example uProgrammed Control & Datapath

Read P&P Revised Appendix C
On website

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

1. Processing in Datapath 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

Supplemental Figures
A Bad Clock Cycle!

Alternative - A BAD ONE!

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
A Simple LC-3b Control and Datapath

Read P&P Revised Appendix C
On website

Microarchitecture of the LC-3b, major components
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[15:11], COND[1:0], IRD—9 bits of control signals produced by the current instruction.
2. IR[15:12]—16 bits of instruction.
3. BEN to indicate whether or not a BR should be taken.
4. Memory, I/O—16 bits of data, Addr 16 bits of address.

Microarchitecture of the LC-3b, major components
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 → state 33 → state 35
  - Decode phase: state 32
Figure C.2: A state machine for the LC-3b
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>
<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>LSHFI/1</td>
<td>NO, YES</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

Simple Design
of the Control Structure
APPENDIX C. THE MICROARCHITECTURE OF THE LC-3B, BASIC MACHINE

IRD
Address of Next State

0,0,IR[15:12]


COND1 COND0

BEN R IR[11]

Branch Ready Addr. Mode

IRD

Address of Next State

6

6
### Appendix C. The Microarchitecture of the LC-3B, Basic Machine

<table>
<thead>
<tr>
<th>Bit Configuration</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000 (State 0)</td>
<td></td>
</tr>
<tr>
<td>000001 (State 1)</td>
<td></td>
</tr>
<tr>
<td>000010 (State 2)</td>
<td></td>
</tr>
<tr>
<td>000011 (State 3)</td>
<td></td>
</tr>
<tr>
<td>000100 (State 4)</td>
<td></td>
</tr>
<tr>
<td>000101 (State 5)</td>
<td></td>
</tr>
<tr>
<td>000110 (State 6)</td>
<td></td>
</tr>
<tr>
<td>000111 (State 7)</td>
<td></td>
</tr>
<tr>
<td>001000 (State 8)</td>
<td></td>
</tr>
<tr>
<td>001001 (State 9)</td>
<td></td>
</tr>
<tr>
<td>001010 (State 10)</td>
<td></td>
</tr>
<tr>
<td>001011 (State 11)</td>
<td></td>
</tr>
<tr>
<td>001100 (State 12)</td>
<td></td>
</tr>
<tr>
<td>001101 (State 13)</td>
<td></td>
</tr>
<tr>
<td>001110 (State 14)</td>
<td></td>
</tr>
<tr>
<td>001111 (State 15)</td>
<td></td>
</tr>
<tr>
<td>010000 (State 16)</td>
<td></td>
</tr>
<tr>
<td>010001 (State 17)</td>
<td></td>
</tr>
<tr>
<td>010010 (State 18)</td>
<td></td>
</tr>
<tr>
<td>010011 (State 19)</td>
<td></td>
</tr>
<tr>
<td>010100 (State 20)</td>
<td></td>
</tr>
<tr>
<td>010101 (State 21)</td>
<td></td>
</tr>
<tr>
<td>010110 (State 22)</td>
<td></td>
</tr>
<tr>
<td>010111 (State 23)</td>
<td></td>
</tr>
<tr>
<td>011000 (State 24)</td>
<td></td>
</tr>
<tr>
<td>011001 (State 25)</td>
<td></td>
</tr>
<tr>
<td>011010 (State 26)</td>
<td></td>
</tr>
<tr>
<td>011011 (State 27)</td>
<td></td>
</tr>
<tr>
<td>011100 (State 28)</td>
<td></td>
</tr>
<tr>
<td>011101 (State 29)</td>
<td></td>
</tr>
<tr>
<td>011110 (State 30)</td>
<td></td>
</tr>
<tr>
<td>011111 (State 31)</td>
<td></td>
</tr>
<tr>
<td>100000 (State 32)</td>
<td></td>
</tr>
<tr>
<td>100001 (State 33)</td>
<td></td>
</tr>
<tr>
<td>100010 (State 34)</td>
<td></td>
</tr>
<tr>
<td>100011 (State 35)</td>
<td></td>
</tr>
<tr>
<td>100100 (State 36)</td>
<td></td>
</tr>
<tr>
<td>100101 (State 37)</td>
<td></td>
</tr>
<tr>
<td>100110 (State 38)</td>
<td></td>
</tr>
<tr>
<td>100111 (State 39)</td>
<td></td>
</tr>
<tr>
<td>101000 (State 40)</td>
<td></td>
</tr>
<tr>
<td>101001 (State 41)</td>
<td></td>
</tr>
<tr>
<td>101010 (State 42)</td>
<td></td>
</tr>
<tr>
<td>101011 (State 43)</td>
<td></td>
</tr>
<tr>
<td>101100 (State 44)</td>
<td></td>
</tr>
<tr>
<td>101101 (State 45)</td>
<td></td>
</tr>
<tr>
<td>101110 (State 46)</td>
<td></td>
</tr>
<tr>
<td>101111 (State 47)</td>
<td></td>
</tr>
<tr>
<td>110000 (State 48)</td>
<td></td>
</tr>
<tr>
<td>110001 (State 49)</td>
<td></td>
</tr>
<tr>
<td>110010 (State 50)</td>
<td></td>
</tr>
<tr>
<td>110011 (State 51)</td>
<td></td>
</tr>
<tr>
<td>110100 (State 52)</td>
<td></td>
</tr>
<tr>
<td>110101 (State 53)</td>
<td></td>
</tr>
<tr>
<td>110110 (State 54)</td>
<td></td>
</tr>
<tr>
<td>110111 (State 55)</td>
<td></td>
</tr>
<tr>
<td>111000 (State 56)</td>
<td></td>
</tr>
<tr>
<td>111001 (State 57)</td>
<td></td>
</tr>
<tr>
<td>111010 (State 58)</td>
<td></td>
</tr>
<tr>
<td>111011 (State 59)</td>
<td></td>
</tr>
<tr>
<td>111100 (State 60)</td>
<td></td>
</tr>
<tr>
<td>111101 (State 61)</td>
<td></td>
</tr>
<tr>
<td>111110 (State 62)</td>
<td></td>
</tr>
<tr>
<td>111111 (State 63)</td>
<td></td>
</tr>
</tbody>
</table>
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>COND&lt;sub&gt;0&lt;/sub&gt;; Unconditional</td>
</tr>
<tr>
<td>COND&lt;sub&gt;1&lt;/sub&gt;:</td>
<td>Memory Ready</td>
</tr>
<tr>
<td>COND&lt;sub&gt;2&lt;/sub&gt;:</td>
<td>Branch</td>
</tr>
<tr>
<td>COND&lt;sub&gt;3&lt;/sub&gt;:</td>
<td>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

The microarchitecture of the LC-3B, basic machine, involves several key components and signals. The Address of Next State is represented by 0,0,IR[15:12]. This signal, along with J[5], J[4], J[3], J[2], J[1], and J[0], feeds into a series of logic gates, including those for Branch, Ready, and Address Mode.

The output signals, BEN and R, are generated by the logic gates following the inputs, feeding into IR[11]. These signals are essential for determining the next state of the machine and for branch prediction and mode switching.

In both cases, the use of J[5] to J[0] helps to generate the necessary signals for the architecture, ensuring that the machine can operate efficiently and accurately.
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. \( J[5:0] \), \( \text{COND}[1:0] \), and \( \text{IRD} \)—9 bits of control logic provided by the control unit.

2. \( \text{ins}[15:2] \), where \( \text{id} \) identifies the opcode, and \( \text{ins}[11:1] \), where \( \text{dif} \) identifies the address of the subroutine.

3. \( \text{BEN} \) to indicate whether or not a BR should be taken.
Figure C.2: A state machine for the LC-3b

\[ \text{MAR} \leftarrow \text{PC} \]
\[ \text{PC} \leftarrow \text{PC} + 2 \]
\[ \text{MDR} \leftarrow \text{M} \]
\[ \text{IR} \leftarrow \text{MDR} \]

\[ \text{BDN} \leftarrow \text{IR}[11] \land N + \text{IR}[10] \land Z + \text{IR}[9] \land P \]
\[ \text{[IR][15:12]} \]

\[ \text{DR} \leftarrow \text{SR1} + \text{OP2}^* \]
\[ \text{set CC} \]

\[ \text{DR} \leftarrow \text{SR1} \land \text{OP2}^* \]
\[ \text{set CC} \]

\[ \text{DR} \leftarrow \text{SR1} \land \text{XOR} \land \text{OP2}^* \]
\[ \text{set CC} \]

\[ \text{MAR} \leftarrow \text{LSFH}([\text{IR}[7:0]], 1) \]

\[ \text{MDR} \leftarrow \text{M}([\text{MAR}[15:1]'0]) \]
\[ \text{R7} \leftarrow \text{PC} \]

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

\[ \text{DR} \leftarrow \text{SHF}([\text{SR}, A, D, \text{amt4}]) \]
\[ \text{set CC} \]

\[ \text{DR} \leftarrow \text{PC} + \text{LSFH}(\text{off9}, 1) \]
\[ \text{set CC} \]

\[ \text{MAR} \leftarrow \text{B} + \text{off6} \]

\[ \text{NOTES} \]
\[ \text{B + off6} : \text{Base} + \text{SEXT}[\text{offset6}] \]
\[ \text{PC + off9} : \text{PC} + \text{SEXT}[\text{offset9}] \]
\[ ^* \text{OP2 may be SR2 or SEXT}[\text{imm5}] \]
\[ ^* [15:8] \text{or [7:0]} \text{depending on MAR[0]} \]
A Simple Datapath Can Become Very Powerful
State Machine for LDW

Microsequencer

<table>
<thead>
<tr>
<th>IRD</th>
<th>Cond</th>
<th>J0</th>
<th>J1</th>
<th>J2</th>
<th>J3</th>
<th>J4</th>
<th>J5</th>
</tr>
</thead>
<tbody>
<tr>
<td>010010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fill in the microinstructions for the 7 states for LDW
The control structure is in the microsequencer (Figure C.5). As we have seen, it is the job of the microsequencer to produce the next state address.
<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>
<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>LSHFI/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
The microarchitecture of the LC-3B, basic machine

IRD

Address of Next State

0,0,IR[15:12]


COND1  COND0

BEN  R  IR[11]

Branch  Ready  Addr. Mode

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

IRD

Address of Next State

0,0,IR[15:12]
### Appendix C. The Microarchitecture of the LC-3B, Basic Machine

<table>
<thead>
<tr>
<th>BMD</th>
<th>Code</th>
<th>Instruction Name</th>
<th>Micro-instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Micro-instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Figure C.7: Specification of the control store
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 MOVS 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
REP MOVMS (DEST SRC)

IF AddressSize = 16
   THEN
      Use CX for CountReg;
   ENDIF
ELSE IF AddressSize = 64 and REX.W used
   THEN Use RCX for CountReg; Fl;
ELSE
   Use ECX for CountReg;
Fl;
ENDIF

WHILE CountReg ≠ 0
   DO
      Service pending interrupts (if any);
      Execute associated string instruction;
      CountReg ← (CountReg - 1);
      IF CountReg = 0
         THEN exit WHILE loop; Fl;
      ENDIF
      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;
      ENDIF
   OD;

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