Single-Cycle Examples, Multi-Cycle Introduction

Today's Menu
- Single cycle examples
- Single cycle machines vs. multi-cycle machines
  - Why multi-cycle?
  - Comparative performance
  - Physical and Logical Design of Datapath and Control
  - Microprogramming

Example of Derived Control: ALU Control Signals

- If we have 5 arithmetic operations
  - And
  - Or
  - Add
  - Subtract
  - Set-on-less-than
- Need 3 bits to distinguish among them

ALU Control for Each Type of Instruction

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW 00</td>
<td>0</td>
<td>XXXXX</td>
<td>add</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>SW 00</td>
<td>0</td>
<td>XXXXX</td>
<td>add</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>Branch equal 01</td>
<td>branch equal</td>
<td>XXXXX</td>
<td>subtract</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>R-type 10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>R-type 10</td>
<td>subtract</td>
<td>100100</td>
<td>subtract</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>R-type 10</td>
<td>AND</td>
<td>100100</td>
<td>and</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>R-type 10</td>
<td>OR</td>
<td>100101</td>
<td>or</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>R-type 10</td>
<td>set-on-less-than</td>
<td>101010</td>
<td>set-on-less-than</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

Example of Derived Control: ALU Control Signals

- ALUop is an encoding created to distinguish
  - LW/SW, Branch Equal, R-type
- It is derived from the instruction opcode through some decoding logic

Let's try a simple example: generate the ALU control signals for the three instructions above

```java
if (ALUop == 00) {
    ALUcontrol = 010;
} else if (ALUop == 01) {
    ALUcontrol = 110;
} else if (ALUop == 11) {
    ALUControl = 111;
}
```

Mapping Control Bits to Hardware
When is ALU Control Bit 2 == 1

To optimize the control logic, identify patterns that cannot occur (e.g., ALUOp = 11)
• Signals that are irrelevant for the function we want to compute
• The more don’t care signals, the better the chance for optional logic
  • Don’t care signals can be replaced by 0 or 1 as needed in the logic optimization process

Note that:
• There is a pattern that cannot occur for ALUOp
• Thus, 01 can be replaced by X, and 10 by 1s
• Function bits are ignored for logic instructions (LW, SW, branch)
• For all other combinations not included in the previous truth table, the outputs are don’t care cases

Truth Table for the Full Set of ALU Control Bits

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Operation</th>
<th>Function code</th>
<th>ALU Action</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>load word</td>
<td>XXXXXXX</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>store word</td>
<td>XXXXXXXX</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>branch equal</td>
<td>XXXXXXXX</td>
<td>subtract</td>
<td>110</td>
</tr>
</tbody>
</table>

Hardware Description Language

\( (\text{ALUOp1} = 0 \& \text{ALUOp0} = 0) \)
  \text{ALUControl[2]} = 0; \text{ALUControl[1]} = 1; \text{ALUControl[0]} = 0;

\( (\text{ALUOp1} = 0 \& \text{ALUOp0} = 1) \)
  \text{ALUControl[2]} = 1; \text{ALUControl[1]} = 1; \text{ALUControl[0]} = 0;

\( \text{Other combinations not included in the previous truth table, the outputs are don’t care cases} \)

Truth Table for the Full Set of ALU Control Bits

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Function code</th>
<th>ALU Action</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Function code</th>
<th>ALU Action</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Block of Logic

ALU Control

When is ALU Control Bit 2 == 1

Let’s pick one bit of the output and examine it, closely...

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Function code</th>
<th>ALU Op1</th>
<th>ALU Op0</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>110</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

When is ALU Control Bit 2 == 1 (cont.)

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Function code</th>
<th>ALU Op1</th>
<th>ALU Op0</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>110</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Function code</th>
<th>ALU Op1</th>
<th>ALU Op0</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>110</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Operation Bit 2 is “1” when
• ALUOp bit is “1” OR
• ALUOp bit is “1” AND Function Code Bits F2 F1 F0 = “111”

ALU Control Bit 2

Appendix

When is ALU Control Bit 2 == 1 (cont.)

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Function code</th>
<th>ALU Op1</th>
<th>ALU Op0</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>110</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Function code</th>
<th>ALU Op1</th>
<th>ALU Op0</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>110</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Function code</th>
<th>ALU Op1</th>
<th>ALU Op0</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>110</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Function code</th>
<th>ALU Op1</th>
<th>ALU Op0</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>110</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Mapping to Logic Gates

- Operation Bit 2 is "1" when
  - ALUOp bit is "1"
  - ALUOp1 bit is "1" and Function Code F1 = "1"

Using Don't Care Information

- If you use "don't cares" on the inputs:
  - make sure you propagate backward!
  - Do not use in a conflicting way for some other output!
- Sometimes, this is not worth the trouble

Truth Table for Full Set of ALU Control Bits

- Operation Bit 1 is "1" when
  - ALUOp bit is "1"
  - Function Code Bit F2 = "0"
  - Using the don't care information, Bit 1 is "1" when
    - ALUOp1 bit is "0"
    - Function Code Bit F2 = "0"

Repeat: When is ALU Control Bit 0 == 1

- ALU Control Bit 0

Repeat: When is ALU Control Bit 0 == 1

- ALU Control Bit 1

Mapping to Logic Gates

- Operation Bit 1 is "1" when
  - ALUOp bit is "0"
  - Function Code Bit 2 (F2) is "0"
Truth Table for Full Set of ALU Control Bits

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>F5</th>
<th>F4</th>
<th>F3</th>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0  1 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1  0 1 1 0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0  1 0 1 0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0  1 1 1 0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1  1 0 0 1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1  0 1 0 1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1  1 0 0 1</td>
</tr>
</tbody>
</table>

- Operation Bit 0 is "1" when
  - ALUOp1 is "1" AND
  - Function Code Bit 3 (F3) is "1" OR Function Code Bit 0 (F0) is "1"
  - We use the don't care information for Bit 0

Mapping to Logic Gates

- Operation Bit 0 is "1" when
  - ALUOp1 is "1" AND
  - Function Code Bit 3 (F3) is "1" OR Function Code Bit 0 (F0) is "1"

So What Have We Done?

- We've built logic to decode the ALU function control bits
  - We use the OpCode input and the function input
  - We generate control signals to control the ALU operation
- This is called Instruction Decode
  - Using the instruction to control the operation we do on data
- In general, a processor has two parts:
  - Control Path
    - Decodes the instruction to determine what to do
    - Handle what we should do (e.g., should we branch, add, subtract, load, store, etc.)
  - Data Path
    - Moves and processes data
    - Handle doing it (e.g., get the data and add it, calculate the branch address, etc.)

Basic Datapath with Control for Our ALU
Another Control Example: How to do PC Select?

Basic Datapath with Control for Our ALU

Other Control Logic to be Decoded

Note we did not show logic for this one.

We did this one. Function Field from Instruction

Another Control Example: How to do PC Select?
Mechanics: Implementing Jumps

Jump instruction contains 26 bit immediate field (address)
- When jump instruction executes, the new PC is constructed as follows:
  - Top 4 bits of PC stay the same
  - Jump instruction’s address is appended to the top 4 bits of the PC
  - Bottom two bits are set to zero

How Datapaths Appear in Silicon: Pentium

Integer ALUs. Superscalar means here "more than one" ALU.
Floating point datapath. Pipelined means just what you would think here: it has pipeline registers to make it faster

How Datapaths Appear in Silicon: Power PC

Floating point unit
2 Integer units. Again, more than 1 due to superscalar execution model

How Datapaths Appear in Silicon: ALPHA

Instruction Cache
Data Cache
Data and Control Busses
Memory Controller
Memory Interface
Bus Interface
Complex Instruction Unit
Complex Instruction Execution Unit
Pipelined Floating Point
2 Integer units
Common “Layout Style” for These Things

- Roughly speaking: tall and skinny

Why “Tall & Skinny”: Bit-Slice Style ALUs

- Logic dominated by wide busses (32, 64) of the operand bits

Why “Tall & Skinny”: Bit-Slice Style ALUs

- Logic dominated by wide busses (32, 64) of the operand bits

Why “Tall & Skinny”: Bit-Slice Style ALUs

- When computations get complex, deep in logic, you pipeline

Pipelining

- When computations get complex, deep in logic, you pipeline
Pipelining

- When computations get complex, deep in logic, you pipeline.

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>Read Register File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 1</td>
<td></td>
</tr>
<tr>
<td>Inst 2</td>
<td></td>
</tr>
<tr>
<td>Inst 3</td>
<td></td>
</tr>
<tr>
<td>Inst 4</td>
<td></td>
</tr>
</tbody>
</table>

Why Pipeline?

- **Speed**: Combinational Logic
  - The longer it is, the slower the clock.

- **Area**: Use all parts at once
  - Each instruction reads the register file first then does an ALU operation. If we pipeline, we can have different instructions using them at the same time. (Better throughput!)

Single Cycle Design Summary

**Datapath design**
- First, make sure you have all the "right stuff" in the datapath
- Make sure you know all your ISA instructions, formats, opcodes, etc.
- Lay out a basic, simple datapath, try your instructions
- You want to avoid being surprised later that you missed an essential unit

**Controlpath design**
- Make sure you know all the control signals you need, and precisely what they do
- Make sure you know when you have a "direct" control signal (e.g., it’s in the instruction format itself) vs. "derived" control signal (e.g., you need to create from a block of logic)
- Make sure you can actually build the derived signals
- Try your instructions, make sure your control works OK
- Do you need more units? Different units?
- The extra logic you will need is the "stage management" stuff for stalls, hazards, etc.

Typical Instruction Execution

1) Fetch instruction from memory
2) Decode instruction
3) If necessary, perform an ALU operation
4) If memory access, perform load/store
5) Write results back to register file and increment the PC

Complete Single-cycle Datapath

Single Cycle “Design Process”

1. Go through each instruction type
2. Figure out the resources and data paths required
   - How many adders for PC, operation, etc.
   - How many ports to memory
3. Overlay requirements of all instructions
   - Where two values need to get to one place, insert a multiplexor
   - Keep track of required control
4. Design the control logic
5. You’re done!

Result:
- Easy to understand machine.
- "Visual model" of the ISA
- Only storage objects are architectural objects:
  - Memory, Register File, PC
What's Wrong with a Single-Cycle Implementation

It was assumed that data flows through all parts of the datapath in ONE clock cycle:
- From Register file to memory
- From Memory to register file
- From PC to PC
- From Register File to Register File

How long is a cycle:
- ALU: 10 ns
- Register File: 5 ns
- Memory: 10 ns
- Assume everything else takes zero time

Difficult to implement variable cycle clock

Usually run the clock at the SLOWEST speed:
- This is called the critical path
- The critical path is the path through the system which limits performance

What if we add a multiplier or divider?
- FP (floating point) math can take a very long time
  - 100's of ns for multiply and divide
  - Lots of techniques to reduce time - will cover later on

What if we break the machine into parts?

Instruction Timings

<table>
<thead>
<tr>
<th>Instr Type</th>
<th>InstrMem</th>
<th>RegRead ALU</th>
<th>DataMem</th>
<th>Reg Write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>35 ns</td>
</tr>
<tr>
<td>Load</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>-</td>
<td>40 ns</td>
</tr>
<tr>
<td>Store</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>-</td>
<td>35 ns</td>
</tr>
<tr>
<td>Branch</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>-</td>
<td>25 ns</td>
</tr>
<tr>
<td>Jump</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10 ns</td>
</tr>
</tbody>
</table>

Multi-cycle Solution

Idea: Let the FASTEST instruction determine clock period

Instr Class 1
Instr Class 2
Instr Class 3

Takes 4 cycles
Takes 2 cycles

Less Wasted Time

Multi-cycle Reality

We are going to go further than allowing the fastest instruction to determine rate
We are going to break EVERY instruction up into phases
MIPS Architecture Instruction Classes

IF
- ALU: op code, operands
- Load: mem rd, mem rd, mem rd, mem rd
- Store: mem rd, mem rd, mem rd, mem rd
- Branch: mem rd, mem rd, mem rd, mem rd
- Jump: mem rd, mem rd, mem rd, mem rd

ID
- ALU: op code, operands
- Load: mem rd, mem rd, mem rd, mem rd
- Store: mem rd, mem rd, mem rd, mem rd
- Branch: mem rd, mem rd, mem rd, mem rd
- Jump: mem rd, mem rd, mem rd, mem rd

OF
- ALU: op code, operands
- Load: mem rd, mem rd, mem rd, mem rd
- Store: mem rd, mem rd, mem rd, mem rd
- Branch: mem rd, mem rd, mem rd, mem rd
- Jump: mem rd, mem rd, mem rd, mem rd

EX
- ALU: op code, operands
- Load: mem rd, mem rd, mem rd, mem rd
- Store: mem rd, mem rd, mem rd, mem rd
- Branch: mem rd, mem rd, mem rd, mem rd
- Jump: mem rd, mem rd, mem rd, mem rd

RS
- ALU: op code, operands
- Load: mem rd, mem rd, mem rd, mem rd
- Store: mem rd, mem rd, mem rd, mem rd
- Branch: mem rd, mem rd, mem rd, mem rd
- Jump: mem rd, mem rd, mem rd, mem rd

NI
- ALU: op code, operands
- Load: mem rd, mem rd, mem rd, mem rd
- Store: mem rd, mem rd, mem rd, mem rd
- Branch: mem rd, mem rd, mem rd, mem rd
- Jump: mem rd, mem rd, mem rd, mem rd

ALU
- Load: mem rd, mem rd, mem rd, mem rd
- Store: mem rd, mem rd, mem rd, mem rd
- Branch: mem rd, mem rd, mem rd, mem rd
- Jump: mem rd, mem rd, mem rd, mem rd

Note That...
- Instruction decode and register read must be done for all classes of instructions
- PC+4 can be done right after instruction fetch
- Address generation for Jump can be performed during the decode step
- The same adder can be shared among:
  - Instruction fetch logic (PC = PC + 4)
  - Address generation logic (Address = A + IR[15:0])
  - Branch target calculation (Next PC = (PC + 4) + IR[15:0])
  - ALU operations (ALUOutput = A + B)
- Same memory port can be used to access instructions and data

Also Note
- We need more storage
  - There are intermediate values between each phase
  - Examples:
    - Place to store instruction (Instruction Register)
    - Place to store ALU output
    - Place to store branch target
    - Place to store operands from RegFile
- This is storage NOT accessible from ISA

Multiple Cycle Implementation Datapath

Full Diagram of Multi-cycle Machine
Recall MIPS Instruction Formats...

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

Load or Store

<table>
<thead>
<tr>
<th>31 - 26</th>
<th>25-21</th>
<th>20-16</th>
<th>15-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 or 43</td>
<td>rs</td>
<td>rt</td>
<td>immediate (address)</td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>31 - 26</th>
<th>25-21</th>
<th>20-16</th>
<th>15-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>rs</td>
<td>rt</td>
<td>immediate (address)</td>
</tr>
</tbody>
</table>

Jump

<table>
<thead>
<tr>
<th>31 - 26</th>
<th>25-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>immediate (address)</td>
</tr>
</tbody>
</table>

Summary

- Single cycle implementations have to consider the worst case delay through the datapath to come-up with the cycle time.
- Multicycle implementations have the advantage of using a different number of cycles for executing each instruction.
- In general, the multicycle machine is better than the single cycle machine, but the actual execution time strongly depends on the workload.
- The most widely used machine implementation is neither single cycle, nor multicycle – it’s the pipelined implementation.
- We will walk through multicycle during the next lecture.