Multi-Cycle CPU: 
Datapath and Control
Why a Multiple Clock Cycle CPU?

- the problem => single-cycle cpu has a cycle time long enough to complete the longest instruction in the machine
- the solution => break up execution into smaller tasks, each task taking a cycle, different instructions requiring different numbers of cycles or tasks
- other advantages => reuse of functional units (e.g., alu, memory)

\[
ET = IC \times CPI \times CT
\]
High-level View
Breaking Execution Into Clock Cycles

- We will have five execution steps (not all instructions use all five)
  - fetch
  - decode & register fetch
  - execute
  - memory access
  - write-back
- We will use Register-Transfer-Language (RTL) to describe these steps
Breaking Execution Into Clock Cycles

• Introduces extra registers when:
  – signal is *computed* in one clock cycle and *used* in another, **AND**
  – the inputs to the functional block that outputs this signal can *change* before the signal is written into a state element.

• Significantly complicates control. *Why?*

• The goal is to *balance* the amount of work done each cycle.
Multicycle datapath
1. Fetch

IR = Mem[PC]
PC = PC + 4

*(may not be final value of PC)*
2. Instruction Decode and Register Fetch

A = Reg[IR[25-21]]
B = Reg[IR[20-16]]
ALUOut = PC + (sign-extend (IR[15-0]) << 2)

• compute target before we know if it will be used (may not be branch, branch may not be taken)
• target is a new state element (temp register)
• everything up to this point must be Instruction-independent, because we still haven’t decoded the instruction.
• everything instruction (opcode)-dependent from here on.
3. Execution, memory address computation, or branch completion

- Memory reference (load or store)
  \[\text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0])\]

- R-type
  \[\text{ALUout} = A \text{ op } B\]

- Branch
  \[\text{if } (A == B) \quad \text{PC} = \text{ALUOut}\]

*At this point, Branch is complete, and we start over; others require more cycles.*
4. Memory access or R-type completion

- Memory reference
  - load
    \[ \text{MDR} = \text{Mem[ALUout]} \]
  - store
    \[ \text{Mem[ALUout]} = B \]

- R-type
  \[ \text{Reg[IR[15-11]]} = \text{ALUout} \]

*R-type is complete*
5. Memory Write-Back

\[ \text{Reg[IR[20-16]] = MDR} \]

*memory instruction is complete*
## Summary of execution steps

<table>
<thead>
<tr>
<th>Step</th>
<th>R-type</th>
<th>Memory</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td></td>
<td>IR = Mem[PC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Instruction Decode/</td>
<td></td>
<td>A = Reg[IR[25-21]]</td>
<td>if (A==B) then PC=ALUout</td>
</tr>
<tr>
<td>register fetch</td>
<td></td>
<td>B = Reg[IR[20-16]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUout = PC + (sign-extend(IR[15-0]) &lt;&lt; 2)</td>
<td></td>
</tr>
<tr>
<td>Execution, address</td>
<td>ALUout = A op B</td>
<td>ALUout = A + sign-extend(IR[15-0])</td>
<td></td>
</tr>
<tr>
<td>computation, branch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type</td>
<td>Reg[IR[15-11]] = ALUout</td>
<td>memory-data = Mem[ALUout] or Mem[ALUout]= B</td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td></td>
<td>Reg[IR[20-16]] = memory-data</td>
<td></td>
</tr>
</tbody>
</table>
Complete Multicycle Datapath

(support for what instruction just got added?)
1. Instruction Fetch

IR = Memory[PC]
PC = PC + 4
A = Register[IR[25-21]]
B = Register[IR[20-16]]
ALUOut = PC + (sign-extend (IR[15-0]) << 2)
3. Execution (R-type)

\[ \text{ALUout} = A \text{ op } B \]
4. R-type Completion

\[\text{Reg}[\text{IR}[15-11]] = \text{ALUout}\]
if (A == B)  PC = ALUOut
3. Memory Address Computation

\[
\text{ALUout} = A + \text{sign-extend}(\text{IR}[15-0])
\]
4. Memory Access

memory-data = Memory[ALUout], or
Memory[ALUout] = B
Reg[IR[20-16]] = memory-data
3. JMP Completion

\[ PC = PC[31-28] \mid (IR[25-0] \ll 2) \]
Multicycle Control

• Single-cycle control used combinational logic
• Multi-cycle control uses ??
• FSM defines a succession of states, transitions between states (based on inputs), and outputs (based on state)
• First two states same for every instruction, next state depends on opcode
Multicycle Control FSM

- Instruction fetch
- Decode and Register Fetch
  - Memory instructions
  - R-type instructions
  - Branch instructions
  - Jump instruction
First two states of the FSM

Instruction Fetch, \textit{state 0}:
- MemRead
- ALUSrcA = 0
- IorD = 0
- IRWrite
- ALUSrcB = 01
- ALUOp = 00
- PCWrite
- PCSource = 00

Instruction Decode/ Register Fetch, \textit{state 1}:

- Opcode = LW or SW
- Opcode = R-type
- Opcode = BEQ
- Opcode = JMP

Memory Inst FSM
R-type Inst FSM
Branch Inst FSM
Jump Inst FSM

Start
A = Register[IR[25-21]]
B = Register[IR[20-16]]
Target = PC + (sign-extend (IR[15-0]) << 2)
R-type Instructions

from state 1

Execution

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

Completion

?

To state 0
4. R-type Completion

Reg[IR[15-11]] = ALUout
BEQ Instruction

from state 1

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSSource = 01

To state 0
Memory Instructions

from state 1

MemRead
IorD = 1

MemWrite
IorD = 1

MemRead
MemtoReg = 1
RegDst = 0

write-back

Address
Computation

To state 0
3. Memory Address Computation

\[
\text{ALUout} = A + \text{sign-extend}(IR[15-0])
\]
JMP Instruction

from state 1

PCWrite
PCSource = 10

To state 0
The Whole FSM
Some Questions

• How many cycles will it take to execute this code?

    lw $t2, 0($t3)
    lw $t3, 4($t3)
    beq $t2, $t3, Label  #assume not taken
    add $t5, $t2, $t3
    sw $t5, 8($t3)

Label: ... 

• What is going on during the 8th cycle of execution?
• In what cycle does the actual addition of $t2 and $t3 take place?

• Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?
Finite State Machine for Control

- Implementation:
ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address $2^m$ entries in the ROM.
  - our outputs are the bits of data that the address points to.

\[\begin{array}{cccccc}
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}\]

$m$ is the "height", and $n$ is the "width"
ROM Implementation

- How many inputs are there?  
  6 bits for opcode, 4 bits for state = 10 address lines  
  (i.e., $2^{10} = 1024$ different addresses)
- How many outputs are there?  
  16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is $2^{10} \times 20 = 20K$ bits  
  (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same  
  — i.e., opcode is often ignored
Multicycle CPU Key Points

- Performance gain achieved from **variable-length** instructions
- \( ET = IC \times CPI \times \text{cycle time} \)
- Required very few new state elements
- More, and **more complex**, control signals
- Control requires **FSM**