4 Multi-Cycle Datapath

- To improve performance, real computer systems divide instruction execution into multiple clock cycles.

- Advantages
  - The work required by the typical instructions can be divided over approximately equal, smaller *elementary operations*. The clock cycle can be set to the longest elementary operation.
  - Shorter instructions can use fewer clock cycles; total execution time is reduced.
  - Datapath resources can be reused on each cycle, saving resources (in single cycle datapath, all operations must occur in parallel).

- Cost
  - Results of ALU or other units may have to be saved in registers, if the inputs to those units change on the next clock cycle, but we still need the result for later so requires more registers
  - Slight extra time to load and read the new registers.
Basic Multicycle Datapath Resources

![Diagram of basic multicycle datapath resources]

- **Memory**: Read address, MemData, Write address, Write Data
- **Instruction Register**: Instruction [25-21], Instruction [20-16], Instruction [15-0]
- **Register**: Read register 1, Read register 2, Write register, Write data
- **ALU**: ALUOp, zero, ALU Result
- **Control**: ALUSelA, ALUSelB
- **Shift**: shift left 2, shift 4
- **Sign Extend**: sign extend 16, sign extend 32
- **MUX**: 0, 1, 2, 3

Copyright 1995 by Coherence LTD., all rights reserved. (Revised: Oct 97 by Rafi Lohev, Oct 99 by Yair Wiseman)
Basic 5-step datapath operations

- Instruction Fetch; Increment program counter (PC)
- Decode instruction and read register operands
- Execute R-Type; Compute address; Branch compare
- Memory Access; Write register
- Write register on Load
First two steps same for all instructions

- **Instruction Fetch**
  - Contents of PC to memory as an address to read from
  - output of memory = instruction ==> to Instruction Register

  \[
  \text{IR} = \text{Memory} [\text{PC}] \\
  \text{PC} = \text{PC} + 4
  \]

- **Decode**
  - Combinational logic will be analyzing op-code during this cycle, so we don't know yet which instruction to execute. Which operations can we do that will be useful for all instructions, or at least not harmful?
    - Read source operand registers' contents. This will never change during the instruction, so we don't need to save them in a register. Source operands are called here \(A\) and \(B\).
    - Compute branch target address (by ALU). Save in special register called \(Target\). Will be used only if instruction is a branch and the branch is taken.

  \[
  A = \text{Registers} [\text{IR}[25 - 21]] \\
  B = \text{Registers} [\text{IR}[20 - 16]] \\
  \text{Target} = \text{PC} + (\text{sign-extend} (\text{IR}[15-0]) \ll 2)
  \]
**Execution step**

There are three possibilities: R-type, Ld/Str, Branch

- **R-type**
  - ALU performs operation on source operands A and B.
    \[ \text{ALUResult} = A \text{ op } B \]

- **Ld/Str**
  - ALU performs an Add to calculate address for Memory.
  - Operands are source register A and base from IR[15-0].
    \[ \text{ALUResult} = A + (\text{sign-extend (IR[15-0])}) \]

- **Branch**
  - ALU is used to compare the source register operands A and B.
  - Zero signal from ALU output will be used to decide if Branch is taken.
    \[ \text{If} (A == B) \text{ PC } = \text{Target} \]
Memory Access

There are two possibilities: Ld/Str, R-type

- Ld/Str
  - Result computed in previous step (ALUResult) used to address memory
  - For a Load, data is output by memory in this step; called memory-data.
  - For a Store, register operand B is written to memory.
  - ALU control from previous step must be maintained stable because we need the result for all of this step.

\[
\text{memory-data} = \text{Memory}[\text{ALUResult}]
\]

or

\[
\text{Memory}[\text{ALUResult}] = B
\]

- R-type
  - Result computed in previous step (ALUResult) is written to register file.
  - RegWrite must be active to write the register file.
  - NOTE: The output of the register file must remain stable from 2 previous steps, through the completion of this step, as well as the ALU control from previous step. Even if this step writes to the same register as either A or B, no changes are actually written until after the clock edge. Therefore, register file output can remain stable until the end of this step.

\[
\text{Registers}[\text{IR}[15-11]] = \text{ALUResult}
\]
**Write Back**

- Last step required only for Load instructions
  - result of access to memory from previous step is written to register number at IR[20-16].

\[
\text{Registers[ IR[20-16] ] } = \text{ memory-data}
\]
Jump instruction

- Jump instruction format:

<table>
<thead>
<tr>
<th>op</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- **op**: operation code.
- **Address**: Address in a 256MB (=2^{28}).

- **Example**: Suppose Label is 11111111111111111111111111111111 and PC is 10101010101010101010101010101000

```
j Label
```

```
000010
```

The address of the jump will be:

```
10101111111111111111111111111100
```

Always added

4 highest bits from PC
# Summary of 5-step Datapath Action

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches/jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td></td>
<td>IR = Memory[PC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td></td>
<td>A = Register[IR[25-21]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B = Register[IR[20-16]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
</tr>
<tr>
<td>Execution, address computation,</td>
<td>ALUResult = A op B</td>
<td>ALUResult = A + sign-extend(IR[15-0])</td>
<td>On branch: if (A == B) then PC = Target</td>
</tr>
<tr>
<td>or branch completion</td>
<td></td>
<td></td>
<td>On Jump: Write PC</td>
</tr>
<tr>
<td>Memory access or R-type</td>
<td>Register[IR[15-11]] = ALUResult</td>
<td>memory-data = Memory[ALUResult] or Memory[ALUResult] = B</td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td>Register[IR[20-16]] = memory-data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Multicycle Datapath (with Control)
### Control Lines Required for Multicycle Datapath

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted</th>
<th>Effect when asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Contents of memory at the write address are replaced by value in the write data input.</td>
</tr>
<tr>
<td>ALUSelA</td>
<td>The first ALU operand is the PC.</td>
<td>The first ALU operand comes from the register given by the Rs field.</td>
</tr>
<tr>
<td>RegDst</td>
<td>The register destination number for the register write comes from the Rt field.</td>
<td>The register destination number for the register write comes from the Rd field.</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>The register given by Write register number is written with the value in the write data input.</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>The value fed to the register write data input comes from the ALU.</td>
<td>The value from the Data memory is fed into the register write data input.</td>
</tr>
<tr>
<td>IorD</td>
<td>The PC is used to supply the address to the memory unit.</td>
<td>The output of the ALU is used to supply the address to the memory unit.</td>
</tr>
<tr>
<td>IRWrite</td>
<td>None</td>
<td>The value from the memory unit is written into the instruction register (IR).</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>The PC is written; the source is controlled by PCSource.</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>The PC is written if the Zero output from the ALU is also active.</td>
</tr>
<tr>
<td>TargetWrite</td>
<td>None</td>
<td>The output of the ALU is written into the register Target</td>
</tr>
</tbody>
</table>
### Additional Control Lines to Complete the Datapath

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCSource</strong></td>
<td>00</td>
<td>The ALU output is sent to the PC for writing.</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>The contents of the register Target are sent to the PC for writing.</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>The jump target address (PC + 4)[29-26] concatenated with IR[25-0] and shifted left two bits) is sent to the PC for writing.</td>
</tr>
<tr>
<td><strong>ALUSelB</strong></td>
<td>00</td>
<td>The second input to the ALU comes from the register given by the rt field.</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>The second input to the ALU is the constant 4.</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>The second input to the ALU is the sign-extended lower 16 bits of the IR</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>The second input to the ALU is the sign-extended and shifted lower 16 bits of the IR.</td>
</tr>
<tr>
<td><strong>ALUOp</strong></td>
<td>00</td>
<td>The ALU performs an add operation</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>The ALU performs a subtract operation</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>The function code field of the instruction determines the ALU operation.</td>
</tr>
</tbody>
</table>
R-type Instructions Flow

![Diagram of R-type Instructions Flow]
Load Instruction Flow

- **Instruction [31-26]**
  - Read address
  - Read data

- **Instruction [25-0]**
  - Write address
  - Write data

- **Instruction Register**
  - Instruction [25-21]
  - Instruction [20-16]

- **Registers**
  - Read register 1
  - Read register 2
  - Write register
  - Write data

- **ALU**
  - Result
  - Output 1
  - Output 2
  - Output 3
  - Output 4

- **Memory**
  - Read address
  - Write address
  - Write data

- **ALU control**
  - Target address

- **Jump address**
  - Shift left 2

- **Control**
  - Control inputs
  - Control outputs

- **Outputs**
  - PCWrite
  - PCWriteCond
  - lorD
  - MemWrite
  - IRWrite
  - MemtoReg
  - RegWrite
  - RegData

- **Instruction**
  - Instruction [15-0]
  - Instruction [15-11]
  - Instruction [31-28]
Store Instruction Flow
Branch Instruction Flow
Jump Instruction Flow