Comp 303
Computer Architecture
A Pipelined Datapath Control
Lecture 13
Pipelined Datapath with Control Signals
### Recall: ALU Control Bits

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>ALUop</th>
<th>Instruction operation</th>
<th>Function field</th>
<th>Desired ALU action</th>
<th>ALU control</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>00</td>
<td>load word</td>
<td>XXXXXXX</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>SW</td>
<td>00</td>
<td>store word</td>
<td>XXXXXXX</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>branch equal</td>
<td>XXXXXXX</td>
<td>subtract</td>
<td>110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>AND</td>
<td>100100</td>
<td>and</td>
<td>000</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>OR</td>
<td>100101</td>
<td>or</td>
<td>001</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>set on less than</td>
<td>101010</td>
<td>set on less than</td>
<td>111</td>
</tr>
</tbody>
</table>

**Diagram:**
- **Opcode** to **Main Control**
- **func** to **ALU Control (Local)**
- **ALUop** to **ALU Control (Local)**
- **ALUctr** to **ALU**
The Values of Control Lines for The Last Three Pipeline Stages

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Execution / Address Calculation stage control lines</th>
<th>Memory Access stage control lines</th>
<th>Write Back stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst</td>
<td>ALU Op0</td>
<td>ALU Op1</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Beq</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Diagram:

- **Instruction**: Flow of instructions through pipeline stages.
- **Control**: Flow of control signals through pipeline stages.
- **IF/ID**: Instruction fetch and decode stage.
- **ID/EX**: Instruction fetch and execute stage.
- **EX/MEM**: Execute and memory access stage.
- **MEM/WB**: Memory write-back stage.

The diagram illustrates the flow of instructions and control signals through the pipeline stages, with specific control lines for each stage.
The Pipelined Datapath with Control Signals
An Example to Clarify Pipelined Control

- Let’s look at what’s happening in the pipeline for the following program.
  
lw $10, 20($1)
  sub $11, $2, $3
  and $12, $4, $5
  or $13, $6, $7
  add $14, $8, $9

- Code does not have any data, control, or structural hazard.
IF: lw $10,20($1)
IF: sub $11,$2,$3

ID: lw $10,20($1)

Clock 2
IF: and $12, $4, $5
ID: sub $11, $2, $3
EX: lw $10, 20($1)

Clock 3
IF: add $14, $8, $9
ID: or $13, $6, $7
EX: and $12, $4, $5
MEM: sub $11, $2, $3
WB: lw $10...

Clock 5
Data Hazards

- Previous example shows us how independent instructions that do not use the results calculated by prior instructions are executed.
- This is not the case with real programs.
- Let’s look at the following code sequence.

  ```
  sub $2, $1, $3
  and $12, $2, $5
  or $13, $6, $2
  add $14, $2, $2
  sw $15, 100($2)
  ```

- The last four instructions are all dependent on the register $2 of the first instruction.
- Assume that register $2 had the value of 10 before the subtract instruction and -20 afterwards.
Data Hazards (Cont’)

<table>
<thead>
<tr>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
<th>CC9</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
</tbody>
</table>

The value of $2$: 10

sub $2$, $1$, $3$

and $12$, $2$, $5$

or $13$, $6$, $2$

add $14$, $2$, $2$

sw $15$, 100($2$)
Data Hazards (Cont’)

- **Simple solution**: Compiler inserts `nop` instructions between the `sub`-and `instructions`.
  - `nop` (no operation) instruction neither modifies data nor writes a result.

    ```
    sub      $2, $1, $3
    nop
    nop
    and     $12, $2, $5
    or      $13, $6, $2
    add     $14, $2, $2
    sw      $15, 100($2)
    ```

- **Result**: It works but 2 clock cycles will be wasted.
  - Performance decrease
It is possible to detect data hazard and then forward the proper value to resolve the hazard.

When an instruction tries to read a register in its EX stage that an earlier instruction intends to write in its WB stage.

This is the case between `sub`-and instruction below:

```
sub $2, $1, $3
and $12, $2, $5
```

This hazard can be detected by simply checking:

```
EX/MEM.RegisterRd = ID/EX.RegisterRs = $2
```
Another hazard is between sub-or instructions:

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2

This hazard can be detected by simply checking:

MEM/WB.RegisterRd = ID/EX.RegisterRt = $2

There is no data hazard between sub-add and sub-sw instructions.
Summary of Data Hazard Conditions

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

This actually refers to destination field of an instruction. It is \textit{rd} field in \textbf{R-type} instructions and \textit{rt} field in \textbf{I-type} instructions. Mux in the \textbf{EX} stage chooses the correct one, therefore, \textbf{EX/MEM} and \textbf{MEM/WB} pipeline registers store this information as a \textit{rd} field (EX/MEM.Register\textit{Rd} and MEM/WB.Register\textit{Rd}).

- Since some of the instructions (i.e. \texttt{sw, beq}) do not write to register file, the above policy is inaccurate. Consider the following code sequence:

  \begin{tabular}{ll}
  \texttt{sw} & $1, 100($5) \\
  \texttt{add} & $3, $1, $2
  \end{tabular}

  \texttt{sw $1, 100($5)} \quad \texttt{add $3, $1, $2} \quad \texttt{EX/MEM.RegisterRd = ID/EX.RegisterRs} \quad \texttt{$1 = 100 +$5}

- This problem can be solved simply by checking RegWr signal.
Another problem: What happens if $0$ is used as a destination register?
- A non-zero value would not be forwarded.

Therefore, hazard detection should be the following:

**EX hazard:**
if (EX/MEM.RegWr
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA=10

if (EX/MEM.RegWr
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB=10

**MEM hazard:**
if (MEM/WB.RegWr
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA=01

if (MEM/WB.RegWr
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB=01
Consider the following sequence:

- add $1, $1, $2
- add $1, $1, $3
- add $1, $1, $4
- ...

In this case, the result is forwarded from MEM stage because the result in the MEM stage is the more recent result than the result in WB stage. Thus, the control for the MEM hazard:

**MEM hazard:**

```plaintext
if (MEM/WB.RegWr
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA=01

if (MEM/WB.RegWr
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB=01
```
<table>
<thead>
<tr>
<th>Mux control</th>
<th>Source</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForwardA = 00</td>
<td>ID/EX</td>
<td>The first ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardA = 10</td>
<td>EX/MEM</td>
<td>The first ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardA = 01</td>
<td>MEM/WB</td>
<td>The first ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
<tr>
<td>ForwardB = 00</td>
<td>ID/EX</td>
<td>The second ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardB = 10</td>
<td>EX/MEM</td>
<td>The second ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardB = 01</td>
<td>MEM/WB</td>
<td>The second ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
</tbody>
</table>
The Pipelined Datapath with Forwarding

The diagram illustrates the pipelined datapath with forwarding, which is a key component in modern processors. The pipeline stages are divided into five main phases: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each stage has specific registers and forwarding units to facilitate data flow and ensure correct execution of instructions. The control signals and data paths are clearly shown, highlighting the data flows from one stage to another, including register reads, ALU operations, data memory access, and write-back operations. The forwarding unit is crucial for avoiding hazards by allowing values to be sent directly from one stage to another, bypassing the memory stage when possible.
Signed Immediate Enhancement
Data Hazards and Stalls

Forwarding does not solve the problem. We need a hazard detection unit. and instruction needs to be stalled one cycle.
The control for the hazard detection unit is:

\[
\text{if (ID/EX.MemRd and (ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt))}
\]

stall the pipeline

Checks if the instruction is a load
Data Hazards and Stalls (Cont’)

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CC1 (Instruction)</th>
<th>CC2 (Instruction)</th>
<th>CC3 (Instruction)</th>
<th>CC4 (Instruction)</th>
<th>CC5 (Instruction)</th>
<th>CC6 (Instruction)</th>
<th>CC7 (Instruction)</th>
<th>CC8 (Instruction)</th>
<th>CC9 (Instruction)</th>
<th>CC10 (Instruction)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw $2, 20($1)</code></td>
<td>Im</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>and $4, $2, $5</code></td>
<td>Im</td>
<td>Reg</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>or $8, $2, $6</code></td>
<td>Im</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add $9, $4, $2</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>slt $1, $6, $7</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `and` and `or` instructions repeat in CC4 what they did in CC3
## Data Hazards and Stalls (Cont’)

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted (0)</th>
<th>Effect when asserted (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>The register destination number for the Write register comes from the rt field (bits 20:16).</td>
<td>The register destination number for the Write register comes from the rd field (bits 15:11).</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None.</td>
<td>The register on the Write register input is written with the value on the Write data input.</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>The second ALU operand comes from the second register file output (Read data 2).</td>
<td>The second ALU operand is the sign-extended, lower 16 bits of the instruction.</td>
</tr>
<tr>
<td>PCSrc</td>
<td>The PC is replaced by the output of the adder that computes the value of PC + 4.</td>
<td>The PC is replaced by the output of the adder that computes the branch target.</td>
</tr>
<tr>
<td>MemRead</td>
<td>None.</td>
<td>Data memory contents designated by the address input are put on the Read data output.</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None.</td>
<td>Data memory contents designated by the address input are replaced by the value on 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 fed to the register Write data input comes from the data memory.</td>
</tr>
</tbody>
</table>
The Pipelined Datapath with Forwarding and Hazard Detection Unit
Actually, the number of instructions needs to be flushed can be reduced from 3 to 1 instruction (shown in the following slide) when the direction of branch is mispredicted.
Datapath for Branch (including HW to flush the pipeline)
Reading Assignment

- Read Chapter 4
- Find the error in one of the Hazard Detection Functions in the chapter.