Pipelining

• Improve performance by increasing instruction throughput

Ideal speedup is number of stages in the pipeline. Do we achieve this?
Pipelining

• What makes it easy
  – all instructions are the same length
  – just a few instruction formats
  – memory operands appear only in loads and stores

• What makes it hard?
  – structural hazards: suppose we had only one memory
  – control hazards: need to worry about branch instructions
  – data hazards: an instruction depends on a previous instruction

• We’ll build a simple pipeline and look at these issues

• We’ll talk about modern processors and what really makes it hard:
  – exception handling
  – trying to improve performance with out-of-order execution, etc.
Basic Idea

- **What do we need to add to actually split the datapath into stages?**

(See Figure 6.10)
Can you find a problem even if there are no dependencies?
What instructions can we execute to manifest the problem?
Corrected Datapath

(See Figure 6.18)

To preserve write register number for \texttt{lw} instruction
Example: Pipe Stages for \texttt{lw} instruction

(See Figure 6.13)
Example: Pipe Stages for \texttt{lw} instruction  

(See Figure 6.13)
Example: Pipe Stages for \texttt{lw} instruction

(See Figure 6.14)

```
Example:

\begin{itemize}
\item Instruction memory
\item Address
\item Instruction
\item PC
\end{itemize}

\begin{itemize}
\item IF/ID
\item ID/EX
\item EX/MEM
\item MEM/WB
\end{itemize}

\begin{itemize}
\item Address
\item Write data
\item Read register 1
\item Read register 2
\item Registers
\item Read data 1
\item Read data 2
\item ALU result
\item Add
\item Shift left 2
\item Zero
\item 16
\item 32
\item Sign extend
\item 1
\item 0
\end{itemize}

\begin{itemize}
\item Write register
\item Write data
\item Data memory
\item Address
\item Read data
\item Data memory
\item Address
\item Read data
\item Data memory
\item Write data
\item Data memory
\item Write data
\item Data memory
\item Write data
\end{itemize}

Chapter 6

© 1998 Morgan Kaufmann Publishers
Modifications by Dr. J © 2002 Cal State Univ, Chico
Example: Pipe Stages for `lw` instruction

(See Figure 6.15)
Example: Pipe Stages for `lw` instruction

(See Figure 6.15)

[Diagram showing pipeline stages for the `lw` instruction, including IF/ID, ID/EX, EX/MEM, and MEM/WB stages with corresponding operations like read register, write data, and ALU operations.]

- **Example:** Pipe Stages for `lw` instruction

  - Instruction memory
  - Address
  - PC
  - IF/ID
  - ID/EX
  - EX/MEM
  - MEM/WB
  - Write back

  - 0 Mux
  - 4
  - Add
  - ALU result
  - Shift left 2
  - Add
  - Zero ALU result
  - Mux
  - 0
  - 1
  - ALU
  - Data memory
  - Address
  - Read data
  - Write data

  (See Figure 6.15)
Datapath for all pipe stages for `lw` instruction
Example: Pipe Stages for sw instruction

(See Figure 6.13)
Example: Pipe Stages for *sw* instruction

(See Figure 6.13)
Example: Pipe Stages for *sw* instruction

(See Figure 6.16)
Example: Pipe Stages for \texttt{sw} instruction

(See Figure 6.17)
Example: Pipe Stages for \texttt{sw} instruction

(See Figure 6.17)
Datapath for all pipe stages for sw instruction
Graphically Representing Pipelines

Multiple-clock-cycle pipeline diagram

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths
Graphically Representing Pipelines

(See Figure 6.21)

- Traditional **multiple-clock-cycle pipeline diagram** of two instructions
- Alternatively, **single-clock-cycle pipeline diagrams**, such as those in slides 6 through 10 and slides 12 through 16, can be used.
  - Shows the state of the entire datapath during a single clock cycle

Program execution order (in instructions)

```
lw $10, $20($1)
sub $11, $2, $3
```

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data access</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction fetch  Instruction decode  Execution  Data access  Write back
Instruction fetch  Instruction decode  Execution  Data access  Write back

Chapter 6
Single-cycle pipeline diagram for two instructions

lw $10, $20($1)
Single-cycle pipeline diagram for two instructions

sub $11, $2, $3  lw $10, $20($1)

(See Figure 6.22)
Single-cycle pipeline diagram for two instructions

\[ \text{sub } \$11, \$2, \$3 \quad \text{lw } \$10, \$20(\$1) \]
Single-cycle pipeline diagram for two instructions

sub $11, $2, $3  lw $10, $20($1)

(See Figure 6.23)
Single-cycle pipeline diagram for two instructions

sub $11, $2, $3   lw $10, $20($1)

(See Figure 6.24)
Single-cycle pipeline diagram for two instructions

sub $11, $2, $3

(See Figure 6.24)
Pipeline Control

(See Figure 6.25)
Pipeline control

• We have 5 stages. What needs to be controlled in each stage?
  – Instruction Fetch and PC Increment
  – Instruction Decode / Register Fetch
  – Execution
  – Memory Stage
  – Write Back

• How would control be handled in an automobile plant?
  – a fancy control center telling everyone what to do?
  – should we use a finite state machine?
Pipeline Control

- Pass control signals along just like the data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst</td>
<td>ALU Op1</td>
<td>ALU Op0</td>
</tr>
<tr>
<td>R-format</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>

(See Figure 6.29)
Datapath with Control

(See Figure 6.30)
Labeled Pipeline Execution with Control

**IF:** \( \text{lw} \ $10, 20 \ ($1) \)
**ID:** before<1>
**EX:** before<2>
**MEM:** before<3>
**WB:** before<4>

**Clock 1**

- **IF/ID:**
  - Instruction memory
  - Address
  - Instruction

- **ID/EX:**
  - Control
  - ID/EX
  - WB
  - M

- **EX/MEM:**
  - Add result
  - Shift left 2
  - ALUSrc
  - EX/MEM
  - M

- **MEM/WB:**
  - Address
  - Data memory
  - Address
  - Data
  - MemWrite

- **WB:**
  - Write data
  - MEM/WB
  - WB

- **Branch:**
  - MemRead

**Instructions:**

- \( \text{lw} \ $10, 20 \ ($1) \)
- \( \text{sub} \ $11, $2, $3 \)
- \( \text{and} \ $12, $4, $5 \)
- \( \text{or} \ $13, $6, $7 \)
- \( \text{add} \ $14, $8, $9 \)
Labeled Pipeline Execution with Control

IF: sub $11, $2, $3
ID: lw $10, 20($1)
EX: before<1>
MEM: before<2>
WB: before<3>

lw  $10, 20($1)
sub  $11, $2, $3
and  $12, $4, $5
or  $13, $6, $7
add  $14, $8, $9

Chapter 6

(See Figure 6.31)
Labeled Pipeline Execution with Control

IF: and $12, $4, $5
ID: sub $11, $2, $3
EX: lw $10, 20($1)
MEM: before<1>
WB: before<2>

Labeled Pipeline Execution Diagram:

- Instruction memory
- PC
- Instruction
- Address
- Read register 1
- Read register 2
- Write register
- Read data 1
- Read data 2
- Write data
- Instruction [15-0]
- Instruction [20-16]
- Instruction [15-11]
- Sign extend
- ALUOp
- RegDst
- ALUSrc
- Add result
- Shift left 2
- ALU
- Write data
- MemRead
- MemWrite
- Branch
- Write data
- Address
- Data memory
- MEM/WB
- IF/ID
- ID/EX
- EX/MEM
- MEM/WB
- Clock 3

Instructions:
- lw $10, 20($1)
- sub $11, $2, $3
- and $12, $4, $5
- or $13, $6, $7
- add $14, $8, $9

Chapter 6

(See Figure 6.32)
Labeled Pipeline Execution with Control

**IF**: or $13, $6, $7

**ID**: and $12, $4, $5

**EX**: sub $11, $2, $3

**MEM**: lw $10, 20($1)

**WB**: before<1>

(See Figure 6.32)

Clock 4

```
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
```
Labeled Pipeline Execution with Control

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

lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

(See Figure 6.33)
Labeled Pipeline Execution with Control (See Figure 6.33)

IF: after<1>

ID: add $14, $8, $9

EX: or $13, $6, $7

MEM: and $12, $4, $5

WB: sub $11...

lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Labeled Pipeline Execution with Control  

Instruction memo

lw  $10, 20($1)
sub  $11, $2, $3
and  $12, $4, $5
or  $13, $6, $7
add  $14, $8, $9
Labeled Pipeline Execution with Control

(See Figure 6.34)

IF: after<3>

ID: after<2>

EX: after<1>

MEM: add $14,$8,$9

WB: or $13 ...
Labeled Pipeline Execution with Control

(See Figure 6.35)

IF: after<4>  ID: after<3>  EX: after<2>  MEM: after<1>  WB: add $14 ...
Pipelined Dependencies

- Problem with starting next instruction before first is finished
  - dependencies that “go backward in time” are data hazards

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>V</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>
<td>–20</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)

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

V (See Figure 6.36)
Software Solution

- Have compiler guarantee no hazards
- Where do we insert the “nops”?

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

- Problem: this really slows us down!
• Use temporary results, don’t wait for them to be written
  – register file forwarding to handle read/write to same register
  – ALU forwarding

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of register $2$</td>
<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>
<tr>
<td>Value of EX/MEM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Value of MEM/WB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)
- sub $2$, $1$, $3$
- and $12$, $2$, $5$
- or $13$, $6$, $2$
- add $14$, $2$, $2$
- sw $15$, 100($2$)

**what if this $2$ was $13$?**
Forwarding

(See Figure 6.40)
Instruction Sequence with Forwarding

**IF:** sub $2, $1, $3  
**ID:** before<1>

Clock 1

### Clock 1

**IF/ID:**
- Instruction memory
- IF/ID: sub $2, $1, $3

**ID/EX:**
- Registers
- ID/EX: before<1>

**EX/MEM:**
- ALU
- EX/MEM: before<2>

**MEM/WB:**
- Data memory
- MEM/WB: before<3>

**WB:**
- Registers
- WB: before<4>

### Forwarding

- Forwarding unit
- Forwarding: sub $2, $1, $3
- and $4, $2, $5
- or $4, $4, $2
- add $9, $4, $2

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

**and $4, $2, $5**

**or $4, $4, $2**

**add $9, $4, $2**

Chapter 6

? 1998 Morgan Kaufmann Publishers

Modifications by Dr. J ? 2002 Cal State Univ, Chico
sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2
Instruction Sequence with Forwarding

(See Figure 6.41)

**IF:** or $4, $4, $2  
**ID:** and $4, $2, $5  
**EX:** sub $2, $1, $3  
**MEM:** before<1>  
**WB:** before<2>

Clock 3

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>Registers</th>
<th>Forwarding unit</th>
<th>Data memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $2, $1, $3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and $4, $2, $5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $4, $4, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $9, $4, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

? 1998 Morgan Kaufmann Publishers  
Modifications by Dr. J 2002 Cal State Univ, Chico
Instruction Sequence with Forwarding

(See Figure 6.41)

IF: add $9, $4, $2
ID: or $4, $4, $2
EX: and $4, $2, $5
MEM: sub $2 ... WB: before<1>

Clock 4

sub $2, $1, $3
and $4, $2, $5
or $3, $4, $2
add $9, $4, $2
Instruction Sequence with Forwarding

IF: after<1>

ID: add $9, $4, $2

EX: or $4, $4, $2

MEM: and $4...

WB: sub $2...

Clock 5

IF/ID: after<1>

ID: add $9, $4, $2

EX: or $4, $4, $2

MEM: and $4...

WB: sub $2...

(See Figure 6.42)

sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

Chapter 6

? 1998 Morgan Kaufmann Publishers
Modifications by Dr. J ? 2002 Cal State Univ, Chico
Instruction Sequence with Forwarding

**IF: after<2>**

**EX:** add $9, $4, $2

**MEM:** or $4 ...

**WB:** and $4 ...

(See Figure 6.42)

![Diagram showing the flow of instructions and data through the pipeline stages of processing, including forwarding paths for registers and data movement across stages.](Diagram)

- **IF/ID:** Instruction
- **ID:** Instruction memory
- **IF/ID:** Instruction memory
- **Registers:** $2, $1, $3
- **Registers:** $4, $2, $5
- **Registers:** $4, $4, $2
- ** Registers:** $9, $4, $2

**Clock 6**

### Example Instructions
- `sub $2, $1, $3`
- `and $4, $2, $5`
- `or $4, $4, $2`
- `add $9, $4, $2`

**Chapter 6**
Can't always forward

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.

Thus, we need a hazard detection unit to “stall” the load instruction.
Stalling

- We can stall the pipeline by keeping an instruction in the same stage

Program execution order (in instructions)

lw $2, 20($1)

and $4, $2, $5

or $8, $2, $6

add $9, $4, $2

slt $1, $6, $7

(See Figure 6.45)
Hazard Detection Unit

- Stall by letting an instruction that won’t write anything go forward

(See Figure 6.46)
Instruction Sequence with Hazard Detection

IF: \texttt{lw} $\$2, 20(\$1)$

ID: before\texttt{<1>}

EX: before\texttt{<2>}

MEM: before\texttt{<3>}

WB: before\texttt{<4>}

Clock 1

lw $\$2, 20(\$1)$
and $\$4, \$2, \$5$
or $\$4, \$4, \$2$
add $\$9, \$4, \$2$
Instruction Sequence with Hazard Detection

IF: and $4, $2, $5
ID: lw $2, 20($1)
EX: before<1>
MEM: before<2>
WB: before<3>

lw $2, 20($1)
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

(See Figure 6.47)
Instruction Sequence with Hazard Detection

IF: or $4, $4, $2
ID: and $4, $2, $5
EX: lw $2, 20($1)
MEM: before<1>
WB: before<2>

lw $2, 20($1)
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

Clock 3
Instruction Sequence with Hazard Detection

**IF:** or $4, $4, $2

**ID:** and $4, $2, $5

**EX:** bubble

**MEM:** `lw $2` ...

**WB:** before<1>

(See Figure 6.48)

lw $2, 20($1)

and $4, $2, $5

or $4, $4, $2

add $9, $4, $2
Instruction Sequence with Hazard Detection

IF: add $9, $4, $2
ID: or $4, $4, $2
EX: and $4, $2, $5
MEM: bubble
WB: lw $2 ...

Clock 5

IF/ID: add $9, $4, $2
ID/ID: or $4, $4, $2
EX/EX: and $4, $2, $5
MEM/MEM: bubble
WB/WB: lw $2 ...

Instruction memory

Registers

Hazard detection unit

Data memory

lw $2, 20($1)
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

(See Figure 6.48)

Chapter 6

? 1998 Morgan Kaufmann Publishers
Modifications by Dr. J ? 2002 Cal State Univ, Chico

56
Instruction Sequence with Hazard Detection

IF: after<1>
ID: add $9, $4, $2
EX: or $4, $4, $2
MEM: and $4 ... WB: bubble

lw $2, 20($1)
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

(See Figure 6.49)
Instruction Sequence with Hazard Detection

**IF:** after<2>

**ID:** after<1>

**EX:** add $9, $4, $2

**MEM:** or $4 ... 

**WB:** and $4 ...

(See Figure 6.49)

PC

IF/IDWrite

IF/ID

Instruction

Instruction memory

Registers

Hazard detection unit

Control

Mux

EX

MEM

WB

Clock 7

Data memory

IF/ID.RegisterRs

IF/ID.RegisterRt

IF/ID.RegisterRd

ID/EX.RegisterRt

ID/EX.RegisterRt

ID/EX.MemRead

ID/EX

IF/ID.Write

MEM/WB.RegisterRd

MEM/WB

IF/Write

MB/WB RegisterRd

P/CWrite

PC

lw $2, 20($1)

and $4, $2, $5

or $4, $4, $2

add $9, $4, $2

'? 1998 Morgan Kaufmann Publishers
Modifications by Dr. J ? 2002 Cal State Univ, Chico
Branch Hazards

- **When we decide to branch, other instructions are in the pipeline!**

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>Time (in clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CC 1</td>
</tr>
<tr>
<td>40 beq $1, $3, 7</td>
<td>IM</td>
</tr>
<tr>
<td>44 and $12, $2, $5</td>
<td>IM</td>
</tr>
<tr>
<td>48 or $13, $6, $2</td>
<td>IM</td>
</tr>
<tr>
<td>52 add $14, $2, $2</td>
<td>IM</td>
</tr>
<tr>
<td>72 lw $4, 50($7)</td>
<td>IM</td>
</tr>
</tbody>
</table>
Branch Hazards

• **Approach #1: Assume/Predict “branch not taken”**
  - Hence, load instructions sequentially following a branch.
  - If branch is taken (incorrect prediction), instructions instructions in the IF, ID, and EX stages must be discarded.
  - Need to add hardware for *flushing* if the branch is taken.

• **Approach #2: Reduce the delay of branches**
  - Move the branch decision hardware earlier in the pipeline – from the MEM stage to the ID stage.
  - Branch address calculation has to occur in the ID stage!

• **Note: These two approaches are not mutually exclusive …**
Flushing Instructions

(See Figure 6.51)
Instruction Sequence with Pipelined Branch

IF: \text{sub} $10, $4, $8$

ID: before<1>

EX: before<2>

MEM: before<3>

WB: before<4>

Clock 1

36 \text{ sub} $10, $4, $8$

40 \text{ beq} $1, $3, 7$

44 \text{ and} $12, $2, $5$

... 

72 \text{ lw} $4, 50($7$)
Instruction Sequence with Pipelined Branch

IF: beq $1, $3, 7
ID: sub $10, $4, $8
EX: before<1>
MEM: before<2>
WB: before<3>

Clock 2

36 sub $10, $4, $8
40 beq $1, $3, 7
44 and $12, $2, $5
... 72 lw $4, 50($7)
Chapter 6

Instruction Sequence with Pipelined Branch

(See Figure 6.52)

Instruction memory

Registers

Data memory

Forwarding unit

Hazard detection unit

IF: and $12, $2, $5
ID: beq $1, $3, 7

EX: sub $10, $4, $8
MEM: before<1>
WB: before<2>

36 sub $10, $4, $8
40 beq $1, $3, 7
44 and $12, $2, $5
...
72 lw $4, 50($7)

IF/Flush
IF/ID
ID/EX
EX/MEM
MEM/WB
WB

Instruction Sequence with Pipelined Branch

(See Figure 6.52)

IF: \textbf{lw} $4, 50 (7)$

ID: \textbf{bubble (nop)}

EX: \textbf{beq} $1, $3, 7$

MEM: \textbf{sub} $10 \ldots$

WB: before<1>

Clock 4

36 \textbf{sub} $10, 4, 8$

40 \textbf{beq} $1, 3, 7$

44 \textbf{and} $12, 2, 5$

... 

72 \textbf{lw} $4, 50 (7)$

? 1998 Morgan Kaufmann Publishers
Modifications by Dr. J ? 2002 Cal State Univ, Chico
Dynamic Branch Prediction

- **Branch prediction** optimizes the two techniques that deal with branch hazards …

- **Branch prediction buffer** or **branch history table**
  - Use a small memory indexed by the lower portion of the address of the branch instruction.
  - This memory contains a bit to indicate whether the branch was recently taken (1) or not (0).
  - Example:
    - Consider a loop branch that branches nine times in a row, then is not taken once.
    - The prediction accuracy of this branch that is taken 90% of the time is only 80% - we will likely predict incorrectly twice, rather than once, when a branch is not taken.
  - Question: At what stage of the pipeline should the branch prediction hardware reside?
Dynamic Branch Prediction

- **Branch prediction buffer**, continued …
  - 2-bit prediction schemes provide better accuracy
  - 2 bits used to encode state information for the finite state machine
  - **Question**: What is the prediction accuracy for a loop branch that branches nine times in a row, then is not taken once?
A \textit{branch delay slot} is the position immediately following a branch, typically containing an instruction that must be flushed if the branch is taken.
Exceptions

• An exception or interrupt is essentially an unscheduled procedure call.
• The address of the instruction causing the exception is saved in a register (e.g. in MIPS, this is the exception program counter or the EPC), and the computer jumps to a predefined address to invoke the appropriate routine for that exception.
• Causes of exceptions include:
  – Arithmetic overflow or underflow
  – I/O device requests
  – Invoking an operating system service from a user program
  – Using an undefined instruction
  – Hardware malfunction
• In MIPS, in the case of an arithmetic overflow, we need to
  – flush instructions in the IF, ID, and EX stages
  – transfer control to the exception routine at location 0x40000040
Exceptions

(See Figure 6.55)
Exceptions (See Figure 6.56)

Overflow exception

lw $16, 50($7)
slt $15, $6, $7
add $1, $2, $1
or $13, ...
and $12, ...

Chapter 6
Exceptions

(See Figure 6.56)
Superscalar Pipelining

- **Superpipelining** – pipelined processors that have longer pipelines
- **Superscalar** – replicate internal components of a computer so it can launch/issue several multiple instructions in every pipelining stage
Improving Performance

• Try and avoid stalls! For example, *reorder* these instructions:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

• Can reordering be implemented on software?
• Can reordering be implemented on hardware?
Dynamic Pipeline Scheduling

- The hardware performs the “scheduling”
  - hardware tries to find instructions to execute
  - out of order execution is possible
  - speculative execution and dynamic branch prediction
Dynamic Pipeline Scheduling

- All modern processors are very complicated
  - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
  - PowerPC and Pentium: branch history table
  - Compiler technology important

- This class has given you the background you need to learn more
Example: Single-cycle vs. multicycle vs. pipelined control

• **Problem:** Compare performance for single-cycle, multicycle, and pipelined control using the instruction mix for `gcc` from Figure 4.54 (frequency of the MIPS instructions for two programs, `gcc` and `spice`) on page 311.

  – Assume the operation times for the major functional units are 2 ns for memory access, 2 ns for ALU operations, and 1 ns for register file read or write.

  – For pipelined execution, assume that half of the load instructions are immediately followed by an instruction that uses the result, that the branch delay on misprediction is 1 clock cycle, and that one-quarter of the branches are mispredicted.

  – Assume that jumps always pay 1 full clock cycle of delay, so their average time is 2 clock cycles.
Example: Single-cycle vs. multicycle vs. pipelined control

• **Answer:** From Figure 4.54 (frequency of the MIPS instructions for two programs, gcc and spice) on page 311, assume that the instruction mix for gcc is

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads</td>
<td>23%</td>
<td>branches</td>
<td>19%</td>
</tr>
<tr>
<td>stores</td>
<td>13%</td>
<td>jumps</td>
<td>2%</td>
</tr>
<tr>
<td>R-format</td>
<td>43%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

– Recall that

\[
\text{CPU exec time} = \frac{\text{Instruction count}}{\text{CPI} \times \text{Clock cycle time}}
\]
Example: Single-cycle vs. multicycle vs. pipelined control

- **Answer (single-cycle control):**

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>6 ns</td>
</tr>
<tr>
<td>load word</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8 ns</td>
</tr>
<tr>
<td>store word</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7 ns</td>
</tr>
<tr>
<td>branch</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>5 ns</td>
</tr>
<tr>
<td>jump</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 ns</td>
</tr>
</tbody>
</table>

- **Single clock for all instructions**
  - cycle time = max (6,8,7,5,2) = 8.00 ns

- **Variable clock**
  - cycle time = 6\%43% + 8\%23% + 7\%13% + 5\%19% + 2\%2% = 6.32 ns
Example: Single-cycle vs. multicycle vs. pipelined control

• Answer (multicycle control):

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>No. of states</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads</td>
<td>5</td>
</tr>
<tr>
<td>R-format</td>
<td>4</td>
</tr>
<tr>
<td>stores</td>
<td>4</td>
</tr>
<tr>
<td>branches</td>
<td>3</td>
</tr>
<tr>
<td>jumps</td>
<td>3</td>
</tr>
</tbody>
</table>

− Then, assuming each state requires 1 clock cycle,
  • cycle time = 5\times23\% + 4\times43\% + 4\times13\% + 3\times19\% + 3\times2\% = \textbf{4.02 ns}
Example: Single-cycle vs. multicycle vs. pipelined control

• **Answer (pipelined control):**
  - Loads take 1 clock cycle w/o dependency and 2 w/ dependency
    • Average clock cycles per load = 1.5
  - **Correctly predicted branches take 1 clock cycle and 2 otherwise**
    • Average clock cycles per branch = 1.75% + 2.25% = 1.25
  - Stores and R-format instructions
    • CPI = 1
  - **Jump instructions**
    • CPI = 2
  - **Therefore,**
    • CPI = 1.5(.23% + 1.25/.19% + 1/.13% + 1.43% + 2/.2%) = 1.1825
    • Cycle time = 1.1825/.2 ns = **2.365 ns**
Example: Single-cycle vs. multicycle vs. pipelined control

- Answer:

<table>
<thead>
<tr>
<th>Control type</th>
<th>Cycle time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-cycle, single clock</td>
<td>8.00</td>
</tr>
<tr>
<td>single-cycle, variable clock</td>
<td>6.32</td>
</tr>
<tr>
<td>multicycle</td>
<td>4.02</td>
</tr>
<tr>
<td>pipelined</td>
<td>2.37</td>
</tr>
</tbody>
</table>

- So, for pipelined vs multicycle control,
  - pipelined is $4.02 \div 2.37 = 1.69$ times faster than multicycle control

- So, for pipelined vs single-cycle control,
  - pipelined is $6.32 \div 2.37 = 2.67$ times faster than variable clock
  - pipelined is $8.00 \div 2.37 = 3.38$ times faster than single clock