CMPE011 Winter 2004

Pipeline hazards

- Structural hazards
- Data hazards
- Control (branch) hazards
- Branch prediction

Textbook: 6.4, 6.5, 6.6
What are “hazards”? 

Hazards are situations that prevent the next instruction(s) in the instruction stream from executing during its (their) designated clock cycle.

There are three kinds of hazard:

• structural hazards — caused by conflicts in resource usage
• data hazards — caused by instructions that depend on previous instructions’ outcome
• control (branch) hazards — caused by pipelining of instructions that change the PC
Structural hazards

Overlapping instructions execution requires duplication of resources to allow all possible combinations of instructions in the pipeline.

The machine is said to have a structural hazard if some combinations of instructions can not be accommodated.
Example: if our CPU had a single memory for instructions and data, then we could not read the PC while a load instruction is accessing the memory.
What does the CPU do in case of a structural hazard?

The pipeline is *stalled* (a *bubble* is inserted).
Data hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution.

_Data hazards_ occur when the pipeline changes the order of read/write accesses to operands with respect to the order in the unpipelined machine.

Two kinds:

- those that can be resolved with _forwarding_
- those that require pipeline stalls
Data hazards

Example:

\[
\begin{align*}
\text{sub} & \quad \$2, \ $1, \ $3 \\
\text{and} & \quad \$12, \ $2, \ $5 \\
\text{or} & \quad \$13, \ $6, \ $2 \\
\text{add} & \quad \$14, \ $2, \ $2 \\
\text{sw} & \quad \$15, \ 100($2) \\
\end{align*}
\]
Example (cont.)

\[
\begin{align*}
\text{sub } $2, $1, $3 & \quad \text{cc 1} \quad \text{MEM} \quad \text{REG} \quad \text{ALU} \quad \text{MEM} \quad \text{REG} \\
\text{and } $12, $2, $5 & \quad \text{cc 2} \quad \text{MEM} \quad \text{REG} \quad \text{ALU} \quad \text{MEM} \quad \text{REG} \\
\text{or } $13, $6, $2 & \quad \text{cc 3} \quad \text{MEM} \quad \text{REG} \quad \text{ALU} \quad \text{MEM} \quad \text{REG} \\
\text{add } $14, $2, $2 & \quad \text{cc 4} \quad \text{MEM} \quad \text{REG} \quad \text{ALU} \quad \text{MEM} \quad \text{REG} \\
\text{sw } $15, 100($2) & \quad \text{cc 5} \quad \text{MEM} \quad \text{REG} \quad \text{ALU} \quad \text{MEM} \quad \text{REG} \\
$1=70, $3=50 & \quad \text{cc 6} \quad \text{MEM} \quad \text{REG} \quad \text{ALU} \quad \text{MEM} \quad \text{REG} \\
$2= & \quad \text{cc 7} \quad \text{MEM} \quad \text{REG} \quad \text{ALU} \quad \text{MEM} \quad \text{REG} \\
\end{align*}
\]
Data hazards

Forwarding or bypassing

The result of an ALU operation is ready at the end of the EX stage, in the EX/MEM pipeline register $\rightarrow$ no need to wait until it is written into a register to use it as an operand.
Data hazards

Example (cont.)

\[
\begin{align*}
\text{sub} & \quad \$2, \$1, \$3 \\
\text{and} & \quad \$12, \$2, \$5 \\
\text{or} & \quad \$13, \$6, \$2 \\
\text{add} & \quad \$14, \$2, \$2 \\
\text{sw} & \quad \$15, 100(\$2) \\
\text{\$1=70, \$3=50} & \\
\text{\$2=} & \end{align*}
\]
Data hazards

Forwarding unit
“Forwarding hazards summary

“MEM stage” hazards:

add $1, $2, $3
add $4, $1, $5
or
add $1, $2, $3
add $4, $5, $1

condition:
EX/MEM.RegisterRd = ID/EX.RegisterRs or
EX/MEM.RegisterRd = ID/EX.RegisterRt
Data hazards

“WB stage” hazards:

\[
\begin{align*}
\text{add} & \quad $1, \; $2, \; $3 \\
\text{add} & \quad $7, \; $7, \; $7 \\
\text{add} & \quad $4, \; $1, \; $5 \\
\end{align*}
\]

or

\[
\begin{align*}
\text{add} & \quad $1, \; $2, \; $3 \\
\text{add} & \quad $7, \; $7, \; $7 \\
\text{add} & \quad $4, \; $5, \; $1 \\
\end{align*}
\]

condition:

\[
\begin{align*}
\text{MEM/WB.RegisterRd} & = \text{ID/EX.RegisterRs} \quad \text{or} \\
\text{MEM/WB.RegisterRd} & = \text{ID/EX.RegisterRt}
\end{align*}
\]
Pipeline stalls

In one case, forwarding cannot prevent a pipeline stall - when an instruction reads a register that is the destination of the preceding LOAD instruction.

Example:

```
lw   $2, 20($1)
and  $4, $2, $5
or   $8, $2, $6
add  $9, $4, $2
slt  $1, $6, $7
```
Data hazards

Example (cont.)

lw  $2,20($1)  
and $4,$2,$5 
or $8,$2,$6 
add $9,$4,$2  
slt $1,$6,$7
Example (cont.)

lw  $2,20($1)

and  $4,$2,$5

or  $8,$2,$6

add  $9,$4,$2

slt  $1,$6,$7

Data hazards
**Data hazards**

**Hazard detection unit (or pipeline interlock)**

During the ID stage, the hazard detection unit checks if the instruction following a LOAD uses the LOAD’s destination register as an operand. If this is the case, the HDU stalls the instruction following the load for one cycle.
Control hazards

Control (branch) hazards

Branch instructions change the execution flow (the PC) based on a condition. Normally, the condition is evaluated in the EX stage, “known” in the MEM stage, and used in the next stage.

Simple solution: Stall the pipeline until the branch is resolved.
First improvement: Detect the branch earlier

\[ \text{beq } $1, $3, 72 \]

\(<\text{next instr}>\)
Second improvement: **Speculative execution (static branch prediction)**

Always start executing the next instruction without stalling ("assume branch not taken"). If the branch is taken, flush the pipeline.
Control hazards

**FOR loops**

```c
for(i = 0; i < Q; ++i)
{
    <some code>
}
```

In MIPS:

```assembly
SUB    $2, $20, $20

BeginFor:  SLTI  $22, $20, 10
           BEQ   $22, $0,  EndFor
           ...
           <some code>
           ...
          
ADDI  $2, $20, 1
J     BeginFor

EndFor:    ...
```

How many bubbles?
Control hazards

WHILE loops

i = 0;
for { <some code>; ++i; }
while (i < 11);

In MIPS:

```
SUB $2 $20, $20

BeginWhile:     ...
    <some code>
    ...
ADDI $2 $20, 1
SLTI $22, $20, 11
BEQ $22, $0, BeginWhile

EndWhile:    ...
```

How many bubbles?
Third improvement: **Dynamic branch prediction**

A *branch prediction buffer* (or *branch history table*) keeps track of previous decisions.

We can keep just 1 bit or 2 bits:

- with 1 bit, the bit is set every time
- with 2 bits, the prediction must be wrong twice before it is changed
Different approach: Delayed branches

One or more instruction(s) immediately following a branch instruction is (are) always executed. The compiler takes care of filling the delay slot(s) with a proper instruction(s) if possible.

Example (1 delay slot):

This code:

\[
\begin{align*}
\text{add} & \quad $3, \quad $4, \quad $5 \\
\text{beq} & \quad $1, \quad $2, \quad \text{L1} \\
\text{nop} & \quad \text{(delay slot)} \\
\text{or} & \quad $6, \quad $7, \quad $8 \\
\text{sub} & \quad $9, \quad $10, \quad $11 \\
\text{L1: lw} & \quad $12, \ 0($13)
\end{align*}
\]

can become this code:

\[
\begin{align*}
\text{beq} & \quad $1, \quad $2, \quad \text{L1} \\
\text{add} & \quad $3, \quad $4, \quad $5 \\
\text{or} & \quad $6, \quad $7, \quad $8 \\
\text{sub} & \quad $9, \quad $10, \quad $11 \\
\text{L1: lw} & \quad $12, \ 0($13)
\end{align*}
\]
Unconditional jumps

Unconditional jumps are resolved in IF (not a hazard).
Recommended exercises

Ex 6.1, 6.2, 6.4, 6.11, 6.12, 6.13, 6.14, 6.15, 6.20