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 cannot 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.

Hazards

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
Data hazards

Example:

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

Structural hazards

What does the CPU do in case of a structural hazard?

The pipeline is stalled (a bubble is inserted).

Data hazards

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
"Forwarding" hazards summary

"MEM stage" hazards:
- add $1, $2, $3
- add $4, $1, $5

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

condition:
- \( \text{EX/MEM,RegisterRd} = \text{ID/EX,RegisterRs} \) or
- \( \text{EX/MEM,RegisterRd} = \text{ID/EX,RegisterRt} \)

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.)

```
low  $2,20($1)
and $4,$2,$5
or  $6,$2,$6
add $9,$4,$2
slt $1,$6,$7
```

```
```

**Data hazards**

“WB stage” hazards:

- add $1,$2,$3
- add $7,$7,$7
- add $4,$1,$5
  
  or
  - add $1,$2,$3
  - add $7,$7,$7
  - add $4,$5,$1

condition:

MEM/WB,RegisterRd = ID/EX,RegisterRs or
MEM/WB,RegisterRd = ID/EX,RegisterRt

**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  $6,$2,$6
add $9,$4,$2
slt $1,$6,$7
```

Page 4
Control hazards

First improvement: Detect the branch earlier

beq $1, $3, 72
\text{\textit{<next instr>}}

\text{\textbf{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.

\text{\texttt{0x040 0010 BEQ $7, $8, XREF}}
\text{\texttt{0x040 0014 AND $12, $13, $14}}
\text{\texttt{XREF: 0x040 0020 OR $16, $17, $18}}

Data hazards

\text{\textbf{Hazard detection unit (or pipeline interlock)}}

During the ID stage, the hazard detection unit checks if the instruction following a \texttt{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.

\text{\textbf{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.

\text{\textbf{Simple solution: Stall the pipeline until the branch is resolved.}}

beq $1, $3, 72
\text{\textit{<next instr>}}
**Control hazards**

**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:  
```
add $3, $4, $5  
beq $1, $2, L1  
```

becomes this code:
```
add $3, $4, $5  
```

---

**FOR loops**

```
for(i = 0; i < 10; ++i)  
{
  <some code>  
}
```

**In MIPS:**
```
SUB $2, $2, $20  
BeginFor:  
  SLT $22, $2 10  
  BEQ $22, $0, EndFor  
  ...  
  <some code>  
  ...  
  ADDI $2, $2 1  
  J BeginFor  
EndFor:  
```

w many bubbles?

---

**WHILE loops**

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

**In MIPS:**
```
SUB $2, $2, $20  
BeginWhile:  
  ...  
  <some code>  
  ...  
  ADDI $2, $2 1  
  SLT $22, $2 11  
  BEQ $22, $0, BeginWhile  
EndWhile:  
```

w many bubbles?
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