Announcements

Please come and talk to me about your progress

• Grades
• Project
Pipeline Scheduling

Instruction scheduled by compiler
- move instruction to reduce stall

\[
\begin{align*}
\text{ld} & \ [A], \ R1 \\
\text{ld} & \ [B], \ R2 \\
\text{add} & \ R1, \ R2, \ R3 \\
\text{st} & \ R3, \ [C] \\
\text{ld} & \ [E], \ R4 \\
\text{ld} & \ [F], \ R5 \\
\text{add} & \ R4, \ R5, \ R6 \\
\text{st} & \ R6, \ [D]
\end{align*}
\]

\[A = B + C\]

\[
\begin{align*}
\text{ld} & \ [E], \ R4 \\
\text{ld} & \ [F], \ R5 \\
\text{add} & \ R4, \ R5, \ R6 \\
\text{st} & \ R6, \ [D]
\end{align*}
\]

\[D = E - F\]

Before Scheduling

\[
\begin{align*}
\text{ld} & \ [A], \ R1 \\
\text{ld} & \ [B], \ R2 \quad \text{stall} \\
\text{add} & \ R1, \ R2, \ R3 \\
\text{st} & \ R3, \ [C] \\
\text{ld} & \ [E], \ R4 \\
\text{ld} & \ [F], \ R5 \quad \text{stall} \\
\text{sub} & \ R4, \ R5, \ R6 \\
\text{st} & \ R6, \ [D]
\end{align*}
\]
After Scheduling

Id [A], R1
Id [B], R2
Id [E], R4
add R1, R2, R3
Id [F], R5
st R3, [C]
sub R4, R5, R6
st R6, [D]

Delayed Load

No pipeline interlocks
Compiler responsible to remove RAW hazards
In case of RAW, must introduce a dummy “nop” instruction
  □ Load “delay” slot
No longer popular!
Other Data Hazards

WAR
- not in DLX: read early write late
- e.g., auto-increment: need multiple WB stages

WAW
- not in DLX: register writes are in order
- e.g., instructions with multi-cycle EX stages (DIVF)

Control Hazards

When an instruction affects which instructions execute next
or change the PC

```assembly
st R2, [R1]
bn R3, R4, loop
sub -, -, -
```

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>st R2, [R1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne R3, R4, loop</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>??</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Control Hazards (Cont.)

VAX e.g.,
- Emer and Clark report 39% change the PC
- Naïve solution adds approx 5 cycles
- => control hazards add 2 to CPU or ~20% increase!

DLX e.g.,
- H&P report 13% change the PC
- Naïve solution adds approx 3 cycles
- => control hazards add 0.39 to CPU or ~30% increase!

Handling Control Hazards (Cont.)

Move up control point
- find out whether branch is taken earlier
- compute target address fast

e.g., in ID stage
- target = PC + immediate
- if (rs1 op 0) PC = target

```
st  R2, [R1]      IF ID  EX  MEM  WB
bne R3, R4, loop IF ID  EX  MEM  WB
??       IF
```

One cycle stall but requires special PC adder
ISA and Control Hazards

Comparisons in ID stage
- must be fast
- can’t afford to subtract
- compares with 0 are simple
- gt, lt test sign-bit
- eq, ne must OR all bits
More general compares need ALU

Control Hazard Techniques

Branch prediction
- guess the direction of branch
- minimize penalty when right
- may increase penalty when wrong

Techniques
- static: compiler through ISA
- dynamic: hardware (later)
  - guess the target address early
  - guess which way the branch goes
Static Techniques

- predict always taken
- predict always not-taken
- predict backward taken
- predict specific opcodes taken
- delayed branches

Predict Always Not-Taken/Taken

Predict not-taken
- Fetch from PC + 4
- Once branch is resolved (EX stage)
  - if taken abort
  - ok if no state change until branch is resolved
  - problem with intermediate pipeline state: condition codes, auto-increment

Predict taken
- Must know target before branch is decoded!
  - Can use prediction
  - Special hardware for fast decode
  - Execute both paths! (expensive?)
Delayed Branch

Always execute the next n instructions
- the instructions are executed whether the branch is taken or not

Compiler must fill the slots (just like delayed load)

Hard to fill the slots
- where to get instructions from?
- Not-taken path?
- Taken path?

No longer popular!

Comparison of Branch Schemes

14% of instructions are PC-changing
65% of those are branches

\[
\text{CPI penalty} = \%\text{branches} \times \left((\%\text{taken} \times \text{taken-penalty}) + (\%\text{not-taken} \times \text{not-taken-penalty})\right)
\]

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Taken Penalty</th>
<th>Not-Taken Penalty</th>
<th>CPI penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>3</td>
<td>3</td>
<td>0.42</td>
</tr>
<tr>
<td>Fast branch</td>
<td>1</td>
<td>1</td>
<td>0.14</td>
</tr>
<tr>
<td>Not-taken</td>
<td>1</td>
<td>0</td>
<td>0.09</td>
</tr>
<tr>
<td>Taken0</td>
<td>0</td>
<td>1</td>
<td>0.05</td>
</tr>
<tr>
<td>Taken1</td>
<td>1</td>
<td>1</td>
<td>0.14</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>0.5</td>
<td>0.07</td>
</tr>
</tbody>
</table>
Interrupts

Some examples
- peripherals call: disk, network, graphics, etc.
- system call, page fault, divide by zero

Interrupts (a.k.a. faults, exceptions, traps) often require
- surprise jump to a vectored address
- linking return address
- saving of PSW (PC, CCs, etc.)
- state change (e.g., to kernel mode)

Classifying Interrupts

1a. Synchronous
   - function of program state (e.g., overflow, page fault)

1b. Asynchronous
   - external device

2a. User request
   - OS call

2b. Coerced
   - from OS or hardware (page fault, bus error, protection violation)
Handling Interrupts (¶)

Precise interrupts
- complete instructions before the offending instruction
- abort (effects of) instructions after
- save PC (delayed branches?)
- force trap instruction into IF

Must handle simultaneous interrupts
- IF, MEM: memory access (page fault, alignment, protection)
- ID: illegal/privileged instruction
- EX: arithmetic exception
- asynchronous: any stage?