Pipelining

CS 217
<table>
<thead>
<tr>
<th>Instruction Processing Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction fetch:</strong></td>
</tr>
<tr>
<td><strong>Execute:</strong></td>
</tr>
<tr>
<td><strong>Memory access:</strong></td>
</tr>
<tr>
<td><strong>Store results:</strong></td>
</tr>
</tbody>
</table>
Pipelining

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Execute</th>
<th>Memory</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Execute</td>
<td>Memory</td>
<td>Store</td>
</tr>
<tr>
<td>Fetch</td>
<td>Execute</td>
<td>Memory</td>
<td>Store</td>
</tr>
<tr>
<td>Fetch</td>
<td>Execute</td>
<td>Memory</td>
<td>Store</td>
</tr>
</tbody>
</table>

### Instruction Stream

<table>
<thead>
<tr>
<th>PC</th>
<th>nPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

- \( add \) %i1, %i1, %o1
- \( add \) %i1, %o1, %o1
- \( sub \) %o1, 3, %o1
- \( add \) %o1, %i2, %o1
Pipelined Load Instructions

- Problem: load followed by use

```
ld [%o0], %o1
add %o1, %o2, %o2
```

Load delay slots are inserted automatically
Pipelined Branch Instructions

• Problem: instruction after branch

cmp %o0, %o1
ble L1
mov %o0, %o1
L1: add %o0, %o0, %o0

branch delay slot
Updating the Program Counter

- Fetch instruction at address stored in nPC
  - Most instructions: nPC = PC + 4
  - Branch instructions: nPC is computed in execute stage

- Execute instruction at address stored in PC
  - After execute: PC = nPC

<table>
<thead>
<tr>
<th>PC</th>
<th>nPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>36</td>
</tr>
<tr>
<td>24</td>
<td>36</td>
</tr>
<tr>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>40</td>
</tr>
<tr>
<td>40</td>
<td>44</td>
</tr>
</tbody>
</table>

Example Code:
```
12  cmp  a,b
16  ble L1
20  nop
24  mov a,c
28  ba L2
32  nop
36  L1: mov b,c
40  L2: ...
```
Delay Slots

• One option: use `nop` in all delay slots

```c
for (i=0; i<n; i++)
    ...

#define i %l0
#define n %l1
clr i
L1: cmp i, n
    bge L2; nop
    ...
    inc i
ba L1; nop
```
Delay Slots

- Optimizing compilers try to avoid delay slots

```c
for (i=0; i<n; i++)
    ...

#define i %l0
#define n %l1
clr i
L1: cmp i,n
    bge L2; nop
    ...
    inc i
    ba L1; nop
L2: cmp i,n
    bl L1; nop
```
```
Delay Slots

- Optimizing compilers try to fill delay slots

```c
if (a>b) c=a; else c=b;

    cmp a,b
    ble L1;
    nop
    mov a,c
    ba L2;
    nop
L1:  mov b,c
L2:  ...
```

```c
    cmp a,b
    ble L1
    mov b,c
    mov a,c
L1:  ...
```
Pipelined Branch Instructions

- Problem: instruction after branch

```plaintext
cmp %o0, %o1
ble L1

mov %o0, %o1

L1: add %o0, %o0, %o0
```
Pipelined Branch Instructions

- Problem: instruction after branch

```
cmp %o0, %o1
ble L1
L1: add %o0, %o0, %o0
mov %o0, %o1
```

Programmer should try to insert independent instructions in branch delay slots
Annul Bit

- Controls the execution of the delay-slot instruction
  
  \[ \text{bg,}a\quad L1 \]
  \[ \text{mov}\quad a, c \]
  
  The \( a \) causes the \text{mov} instruction to be executed if the branch is taken, and not executed if the branch is not taken.

- Exception
  
  \[ \text{ba,}a\quad L \] does not execute the delay-slot instruction
Annul Bit (cont)

- Optimized for (i=0; i<n; i++) 1;2;...;n

```assembly
    clr i
    ba L2
L1: 1
    2
    . .
    n
    inc i
L2: cmp i,n
    bl,a L1
    nop
    clr i
    ba,a L2
L1: 2
    . .
    n
    inc i
L2: cmp i,n
    bl,a L1
    1
```
While-Loop Example

```c
while (...) {
    stmt_1
    ...
    stmt_n
}
```

```
test: cmp ...
    bx done
    nop
    stmt_1
    ...
    stmt_n
    ba test
    nop
done: ...
```

3 instr

2 instr
While-Loop (cont)

• Move test to end of loop

```assembly
test: cmp ...  
    bx done
    nop
loop: stmt_1  
    :     
    stmt_n
    cmp ...  
    bnx loop
    nop
done: ...
```

• Eliminate first test

```assembly
ba test
    nop
loop: stmt_1  
    :     
    stmt_n
    test: cmp ...
    bnx loop
    nop
...```
While-Loop (cont)

- Eliminate the `nop` in the loop

```plaintext
ba test
nop
loop: stmt₂
  :  
  stmtₙ
test: cmp ...
  bnₓ,a  loop
  stmt₁
  ...
```

now 2 overhead instructions per loop
If-Then-Else Example

```c
if (...) {
    t-stmt_1
    :
    t-stmt_n
} else {
    e-stmt_1
    :
    e-stmt_m
}
```

How optimize?

```c
cmp ... 
bnx else
nop
    t-stmt_1
    :
    t-stmt_n
    ba next
    nop
else: e-stmt_1
    e-stmt_2
    :
    e-stmt_m
next: ...
```
If-Then-Else Example

if (...) {
    t-stmt_1
    :
    t-stmt_n
}
else {
    e-stmt_1
    :
    e-stmt_m
}

How optimize?

cmp ...
bnx, a else
e-stmt_1
t-stmt_1
    :
t-stmt_n
ba next
nop
else: e-stmt_2
    :
e-stmt_m
next: ...
If-Then-Else Example

```plaintext
if (...) {
    t-stmt_1
    : t-stmt_n
}
else {
    e-stmt_1
    : e-stmt_m
}

How optimize?
```