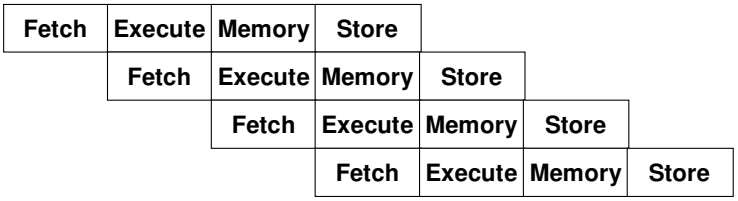
Pipelining

CS 217

Instruction Processing Steps

- Instruction fetch: Fetch and decode instruction, retrieve operands from registers
- Execute: Execute arithmetic instruction, compute branch target address, compute load/store memory address
- Memory access: Access memory for load or store, Fetch instruction at target of branch
- Store results: Write instruction results to registers

Pipelining



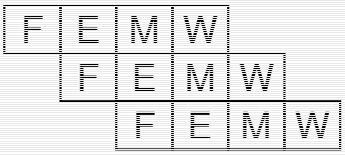

```
12    add  %i1, %i1, %o1
16    add  %i1, %o1, %o1
20    sub  %o1, 3, %o1
24    add  %o1, %i2, %o1
```

PC	nPC
12	16
16	20
20	24
24	28

Pipelined Load Instructions

- Problem: load followed by use

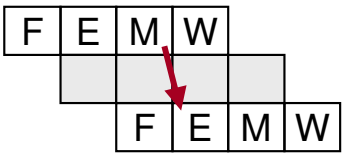
```
ld [%o0], %o1
add  %o1, %o2, %o2
```




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

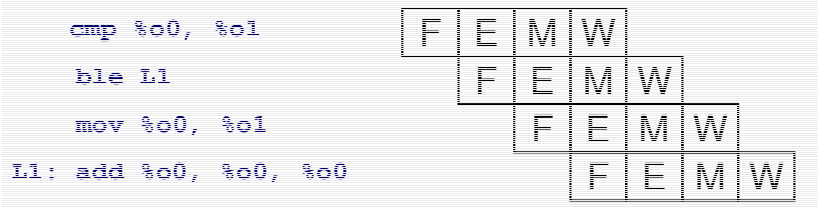
load delay slot →

```
add  %o1, %o2, %o2
```



Load delay slots are inserted automatically

Pipelined Branch Instructions

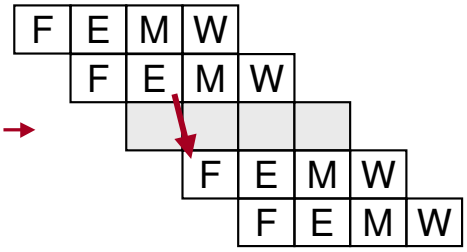


- Problem: instruction after branch

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



```
    cmp %o0, %o1
    ble L1
          branch delay slot →
    mov %o0, %o1
L1: add %o0, %o0, %o0
```



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

```
                                PC  nPC
12        cmp a,b               12  16
16        ble L1                16  20
20        nop                   20  36
24        mov a,c
28        ba L2
32        nop
36   L1:  mov b,c               36  40
40   L2:  ...                   40  44
```

Delay Slots

- One option: use **nop** in all delay slots

```
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

```
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
```

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

Delay Slots

- Optimizing compilers try to fill delay slots

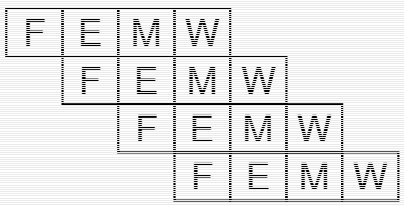
```
if (a>b) c=a; else c=b;

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

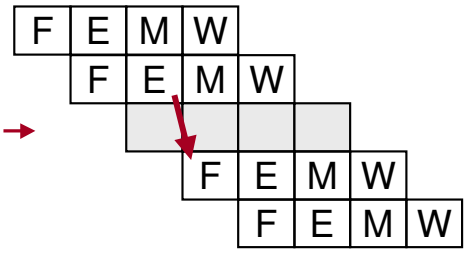
Pipelined Branch Instructions

- Problem: instruction after branch

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



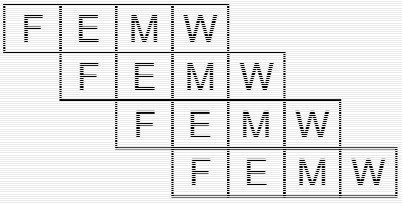

```
    cmp %o0, %o1
    ble L1
          branch delay slot →
    mov %o0, %o1
L1: add %o0, %o0, %o0
```



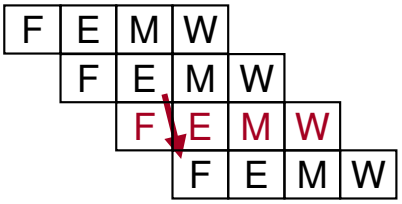
Pipelined Branch Instructions

- Problem: instruction after branch

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




```
    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

```
bg,a    L1
mov     a,c
```

the `,a` causes the `mov` instruction to be executed if the branch is taken, and not executed if the branch is not taken

- Exception

`ba,a  L` does not execute the delay-slot instruction

Annul Bit (cont)

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

```
     clr  i              clr   i
     ba   L2             ba,a  L2
L1:  1               L1: 2
     2                   . . .
     . . .               n
     n                   inc   i
     inc  i          L2: cmp   i,n
L2:  cmp  i,n            bl,a  L1
     bl   L1             1
     nop
```

While-Loop Example

```
while (...)
{
  stmt1
    :
  stmtn
}
```

```
test: cmp ...   ┐
      bx done   │ 3 instr
      nop       ┘
      stmt1
        :
      stmtn
      ba test   ┐
      nop       ┘ 2 instr
done: ...
```

While-Loop (cont)

- Move test to end of loop

```
test: cmp ...
      bx done
      nop
loop: stmt1
        :
      stmtn
      cmp ...
      bnx loop
      nop
done: ...
```

- Eliminate first test

```
      ba test
      nop
loop: stmt1
        :
      stmtn
test: cmp ...
      bnx loop
      nop
      ...
```

While-Loop (cont)

- Eliminate the **nop** in the loop

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

now 2 overhead instructions per loop

If-Then-Else Example

```
if (...) {
  t-stmt1
     :
  t-stmtn
}
else {
  e-stmt1
     :
  e-stmtm
}
```

How optimize?

```
        cmp ...
        bnx else
        nop
        t-stmt1
           :
        t-stmtn
        ba next
        nop
else:   e-stmt1
        e-stmt2
        :
        e-stmtm
next:   ...
```

If-Then-Else Example

```
if (...) {
  t-stmt1
     :
  t-stmtn
}
else {
  e-stmt1
     :
  e-stmtm
}
```

How optimize?

```
        cmp ...
        bnx, a else
        e-stmt1
        t-stmt1
           :
        t-stmtn
        ba next
        nop
else:   e-stmt2
        :
        e-stmtm
next:   ...
```

If-Then-Else Example

```
if (...) {
  t-stmt1
     :
  t-stmtn
}
else {
  e-stmt1
     :
  e-stmtm
}
```

How optimize?

```
        cmp ...
        bnx, a else
        e-stmt1
        t-stmt1
           :
        ba next
        t-stmtn
else:   e-stmt2
        :
        e-stmtm
next:   ...
```