Branching

CS 217

Condition Codes

- Processor State Register (PSR)

- Integer condition codes (icc)
  - **N** set if the last ALU result was **n**egative
  - **Z** set if the last ALU result was **z**ero
  - **V** set if the last ALU result was o**v**erflowed
  - **C** set if the last ALU instruction that modified the icc caused a **c**arry out of, or a borrow into, bit 31

Condition Codes (cont)

- **cc** versions of the integer arithmetic instructions set all the codes
- **cc** versions of the logical instructions set only **N** and **Z** bits
- Tests on the condition codes implement conditional branches and loops
- Carry and overflow are used to implement multiple-precision arithmetic

Compare and Test

- Synthetic instructions set condition codes

```
tst reg          orcc reg,%g0,%g0

cmp src1,src2    subcc src1,src2,%g0
cmp src,value    subcc src,value,%g0
```

- Using `%g0` as the destination discards the result

Carry and Overflow

- If the carry bit is set
  - the last addition resulted in a carry, or
  - the last subtraction resulted in a borrow
- Used for multi-word addition

```
addcc %g3,%g5,%g7
addxcc %g2,%g4,%g6

(%g6,%g7) = (%g2,%g3) + (%g4,%g5)
```

  the most significant word is in the even register
- Overflow indicates result of subtraction (or signed-addition) doesn't fit

Branch Instructions

- Transfer control based on icc

```
b [a|n|..|z] {,a}   label
```

target is a PC-relative address: PC + 4 x disp22
where PC is the address of the branch instruction

Branch Instructions (cont)

- Unconditional branches (and synonyms)

`ba`	`jmp`	branch always
`bn`	`nop`	branch never

- Raw condition-code branches

`bnz`	!Z
`bz`	Z
`bpos`	!N
`bneg`	N
`bcc`	!C
`bcs`	C
`bvc`	!V
`bvs`	V

Branching Instructions (cont)

- Comparisons

instruction	signed	unsigned
`be`	Z	Z
`bne`	!Z	!Z
`bg bgu`	!(Z | (N^V))	!(C | Z)
`ble bleu`	Z | (N^V)	C | Z
`bge bgeu`	!(N^V)	!C
`bl blu`	N^V	C

Control Transfer

- Instructions normally fetched and executed from sequential memory locations
- PC is the address of the current instruction, and nPC is the address of the next instruction (nPC = PC + 4)
- Branches and control transfer instructions change nPC to something else

Control Transfer (cont)

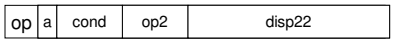
- Control transfer instructions

instruction	type	addressing mode
bicc	conditional branch	PC-relataive
jmpl	jump and link	register indirect
rett	return from trap	register indirect
call	procedure call	PC-relative
ticc	traps	register indirect (vectored)

PC-relative addressing is like register displacement addressing that uses the PC as the base register

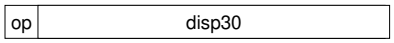
Control Transfer (cont)

- Branch instructions



nPC = PC + 4 x signextend(disp22)

- Calls



nPC = PC + zeroextend(disp30)

position-independent code does not depend on where it's loaded; uses PC-relative addressing

Branching Examples

- if-then-else

```
if (a > b)          #define a %l0
   c = a;           #define b %l1
else                #define c %l2
   c = b;           cmp a,b
                    ble L1; nop
                    mov a,c
                    ba L2; nop
                L1: mov b,c
                L2: ...
```

Branching Examples (cont)

- Loops

```
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:
```


Branching Examples (cont)

- Alternative implementation

```
for (i=0; i<n; i++)     #define i %l0
   . . .                #define n %l1
                        clr i
                        ba L2; nop
                   L1:  . . .
                        inc i
                   L2:  cmp i,n
                        bl L1; nop
```


Instruction Pipelining

- Pipeline

Fetch	Decode	Oprnd	Execute	Store		
	Fetch	Decode	Oprnd	Execute	Store	
		Fetch	Decode	Oprnd	Execute	Store

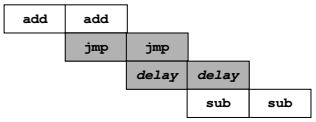
. . .

- PC is incremented by 4 at the Fetch stage to retrieve the next instruction

Pipelining (cont)

- A delay slot is caused by a **jmp** instruction

```
PC  nPC  instruction
 8  12   add
12  16   jmp 40
16  40   delay
  . . .
40  44   sub
```



Delay Slots

- One option: use **nop** in all 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: ...
```


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
   L2:  cmp  i,n       L2:  cmp   i,n
        bl   L1             bl,a  L1
        nop                 1
```


While-Loop Example

```
while (...)          test: cmp ...
{                          bx done      } 3 instr
   stmt1                   nop
     :                     stmt1
   stmtn                     :
}                          stmtn
                           ba test      } 2 instr
                           nop
                     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-stmtm
next:   ...
```