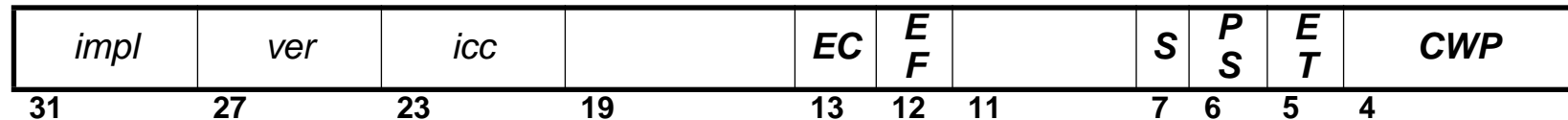
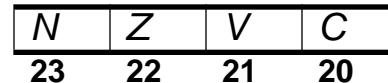


# Condition Codes

- processor state register (*psr*)



- integer condition codes — the *icc* field — holds 4 bits



**N** set if the last ALU result was negative

**Z** set if the last ALU result was zero

**V** set if the last ALU result overflowed

**C** set if the last ALU instruction that modified *icc* caused a carry out of, or a borrow into, bit 31

- cc* versions of the integer arithmetic instructions set all the codes
- cc* versions of the logical instructions set only **N** and **Z**
- tests on the condition codes implement conditionals and loops
- carry and overflow are used to implement multiple-precision arithmetic
- see page 28 in the SPARC Architecture Manual, §4.8 in Paul

# Compare and Test

---

- test and compare *synthetic* instructions set condition codes

- to test a single value

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

- compare two values

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

- using `%g0` as a destination discards the result

# Carry and Overflow

---

- if the carry bit (**c**) is set
  - the last addition resulted in a carry
  - or the last subtraction resulted in a borrow
- carry is needed to implement arithmetic using numbers represented in several words, e.g. multiple-precision addition

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

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

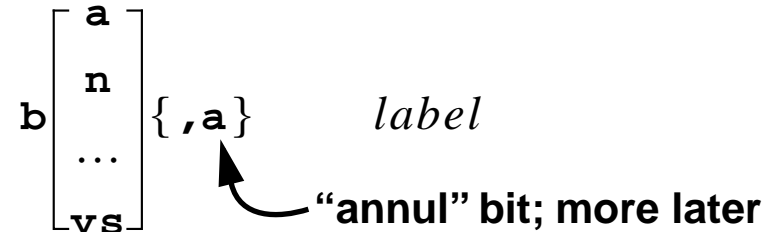
the ***most-significant word*** is in the ***even*** register;  
 the ***least-significant word*** is in the ***odd*** register

- overflow (**v**) indicates that the result of signed addition or subtraction doesn't fit

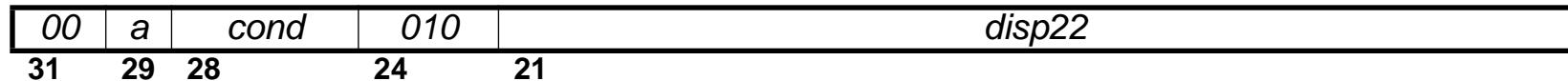
# Branches

---

- branch instructions transfer control based on *icc*



branches are format 2 instructions



- target is a ***PC-relative*** address and is  $PC + 4 \times disp22$ , where *PC* is the address of the branch instruction
- unconditional branches

<b><i>branch</i></b>	<b><i>condition</i></b>	<b><i>synthetic synonym</i></b>
<b>ba</b>	branch always	<b>jmp</b>
<b>bn</b>	branch never	<b>nop</b>

## Branches, cont'd

---

- raw condition-code branches

<i>branch</i>	<i>condition</i>	<i>synthetic synonym</i>
<b>bnz</b>	<b>!Z</b>	
<b>bz</b>	<b>Z</b>	
<b>bpos</b>	<b>!N</b>	
<b>bneg</b>	<b>N</b>	
<b>bcc</b>	<b>!C</b>	<b>bgeu</b>
<b>bcs</b>	<b>C</b>	<b>blu</b>
<b>bvc</b>	<b>!V</b>	
<b>bvs</b>	<b>V</b>	

- comparisons

<i>branches</i>	<i>signed</i>	<i>unsigned</i>	<i>synthetic synonym</i>
<b>be</b>	<b>Z</b>	<b>Z</b>	<b>bz</b>
<b>bne</b>	<b>!Z</b>	<b>!Z</b>	<b>bnz</b>
<b>bg bgu</b>	<b>!(Z   (N^V))</b>	<b>!(C   Z)</b>	
<b>ble bleu</b>	<b>Z   (N^V)</b>	<b>C   Z</b>	
<b>bge bgeu</b>	<b>!(N^V)</b>	<b>!C</b>	
<b>bl blu</b>	<b>N^V</b>	<b>C</b>	

# Control Transfer

---

- normally, instructions are fetched and executed from sequential memory locations
- program counter, *PC*, is address of the current instruction, and the program counter, *nPC*, is address of the next instruction:  $nPC = PC + 4$
- branches, control-transfer instructions change *nPC* to something else
- control-transfer instructions

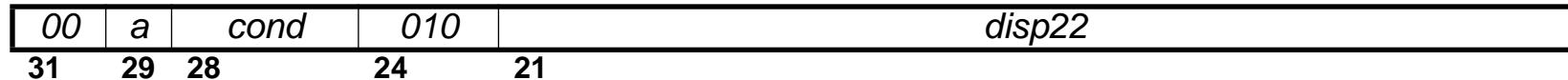
<i>instruction</i>	<i>type</i>	<i>addressing mode</i>
<i>bicc</i>	conditional branches	<i>PC</i> -relative
<i>fbfcc</i>	floating point	<i>PC</i> -relative
<i>cbccc</i>	coprocessor	<i>PC</i> -relative
<i>jmp1</i>	jump and link	register indirect
<i>rett</i>	return from trap	register indirect
<i>call</i>	procedure call	<i>PC</i> -relative
<i>ticc</i>	traps	register-indirect vectored

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

## Control Transfer, cont'd

---

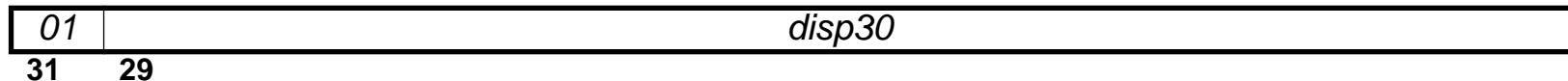
- branches



$$nPC = PC + 4 \times \text{signextend}(disp22)$$

jumping to an arbitrary location may require two branches, but branches are used to build conditionals and loops in “small” code blocks

- calls



$$nPC = PC + 4 \times \text{zeroextend}(disp30)$$

is multiplied by 4 because all instructions are word aligned

- ***position-independent*** code is code whose correct execution does not depend on where it is loaded, i.e., all instructions use *PC*-relative addressing

# Branching Examples

---

- if-then-else

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

becomes

```
#define a %10
#define b %11
#define c %13

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

- loops

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

becomes

```
#define i %10
#define n %11

    clr i
L1: cmp i,n
    bge L2; nop
    ...
    inc i
    ba L1; nop
L2:
```

- gcc generates

```
    clr i
    ba L5; nop
L2: ...
    inc i
L5: cmp i,n
    bl L2; nop
```