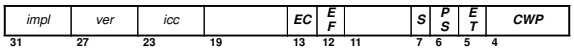
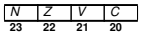
Condition Codes

- processor state register (`psr`)



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



  - ***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 o***v***erflowed
  - ***C*** set if the last ALU instruction that modified ***icc*** caused a ***c***arry 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` src_1`,`src_2 — `subcc` src_1`,`src_2`,%g0`

  `cmp` *src*`,`*value* — `subcc` *src*`,`*value*`,%g0`
- using `%g0` as a destination discards the result; the instruction is executed only for the side-effect of setting the condition codes.

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

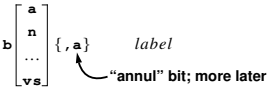
```
addcc %g3,%g5,%g7
addxcc %g2,%g4,%g6
```

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

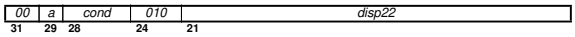
  the ***most-significant word*** is in the ***even*** register;
  the ***least-significant word*** is in the ***odd*** register
- overflow (***V***) indicates that the result of signed addition or subtraction does not fit

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

branch	***condition***	***synthetic synonym***
`ba`	branch always	`jmp`
`bn`	branch never	`nop`

Branches, cont'd

- raw condition-code branches

branch	*condition*	*synthetic synonym*
`bnz`	`!`***Z***	
`bz`	***Z***	
`bpos`	`!`***N***	
`bneg`	***N***	
`bcc`	`!`***C***	`bgeu`
`bcs`	***C***	`blu`
`bvc`	`!`***V***	
`bvs`	***V***	

- comparisons

branches	*signed*	*unsigned*	*synthetic synonym*
`be`	***Z***	***Z***	`bz`
`bne`	`!`***Z***	`!`***Z***	`bnz`
`bg bgu`	`!(`***Z*** `|` `(`***N*** `^` ***V***`))`	`!(`***C*** `|` ***Z***`)`	
`ble bleu`	***Z*** `|` `(`***N*** `^` ***V***`)`	***C*** `|` ***Z***	
`bge bgeu`	`!(`***N*** `^` ***V***`)`	`!`***C***	
`bl blu`	***N*** `^` ***V***	***C***	

Copyright ©1995 D. Hanson, K. Li & J.P. Singh

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

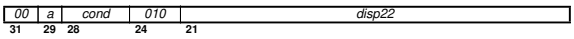
instruction	*type*	*addressing mode*
`b`***icc***	conditional branches	***PC***-relative
`fb`***fcc***	floating point	***PC***-relative
`cb`***ccc***	coprocessor	***PC***-relative
`jmpl`	jump and link	register indirect
`rett`	return from trap	register indirect
`call`	procedure call	***PC***-relative
`t`***icc***	traps	register-indirect vectored

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

Copyright ©1995 D. Hanson, K. Li & J.P. Singh

Control Transfer, cont'd

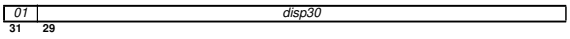
- branches



nPC = ***PC*** + 4 × 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 × 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

Copyright ©1995 D. Hanson, K. Li & J.P. Singh

Branching Examples

- if-then-else

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

becomes

```
#define a %l0
#define b %l1
#define c %l3

    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 %l0
#define n %l1

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

- `lcc` generates

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

Copyright ©1995 D. Hanson, K. Li & J.P. Singh