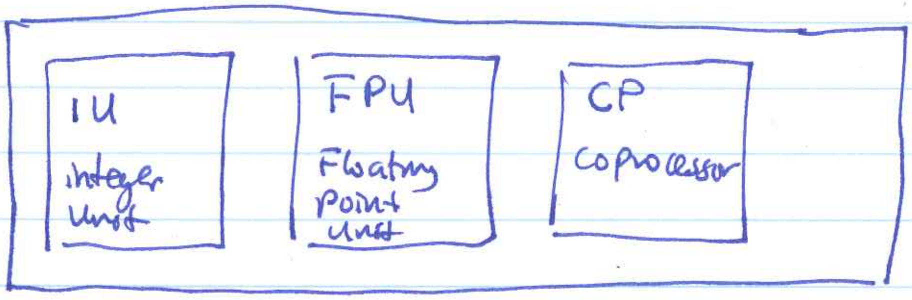
SPARC Architecture

SPARC = Scalable Processor ARChitecture.
Developed by Sun Microsystems

Based on the RISC II microprocessor
devolpped at Berkeley.

- Logically, the SPARC consists of 3 units:

The SPARC CPU:



- The integer unit (IU) is in control of the other processors.

~~The IU will execute integer instructions~~.
~~If~~
The IU will fetch instructions.
If instruction is an integer instruction, the IU will execute it.
If it is a floating point instruction, the IU will send it to the FPU.
Same with the coprocessor instruction

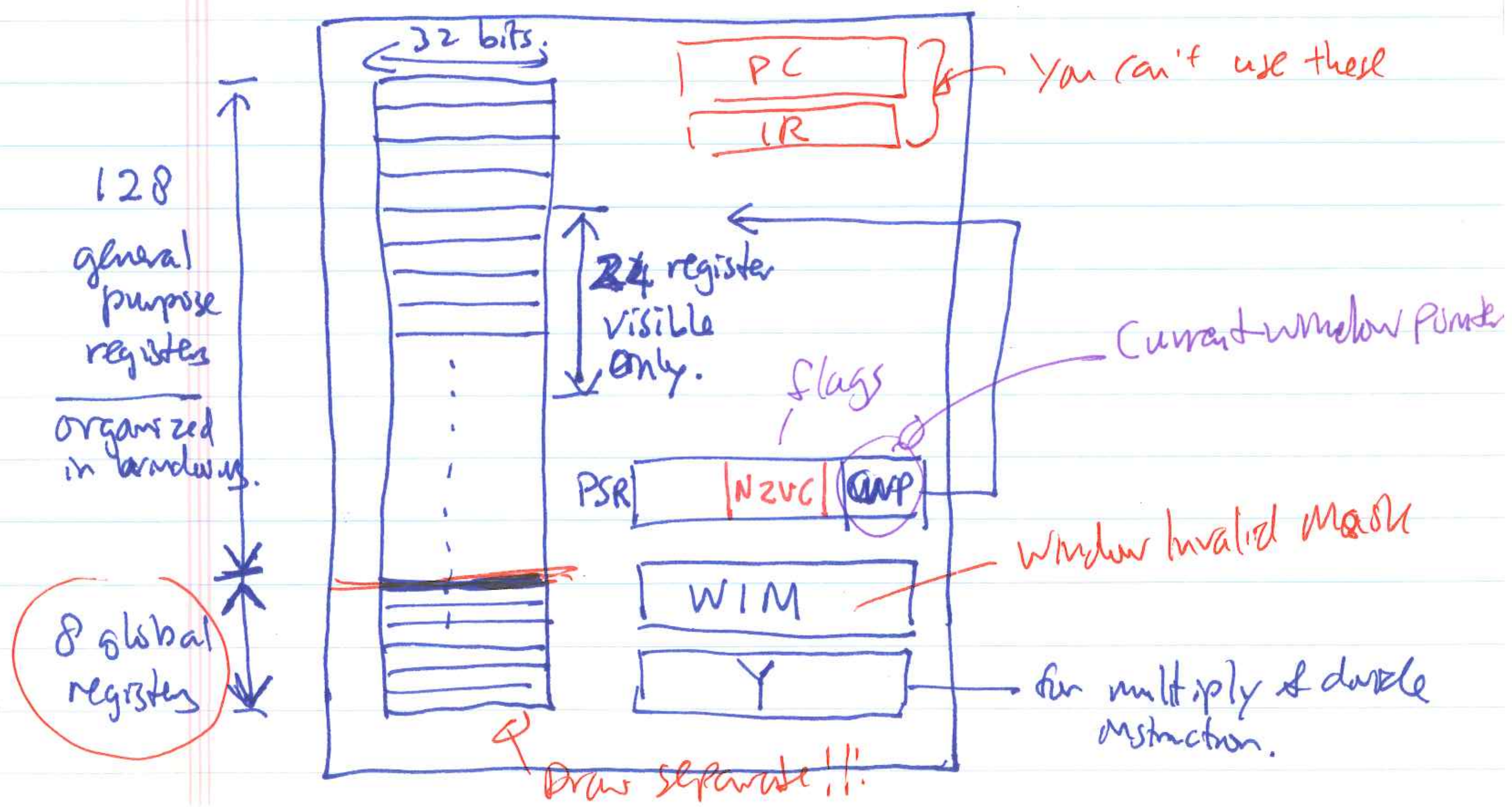
- Coprocessor :

The SPARC chip set only include the IU & the FPU. But it allows the user to add one more coprocessor. ~~so far, there are no addit coprocessor~~

Coprocessor performs a dedicated task very fast. FPU is an example of a coprocessor ("Math co-processor"). So SPARC allows in additon to the FPU, another coprocessor.

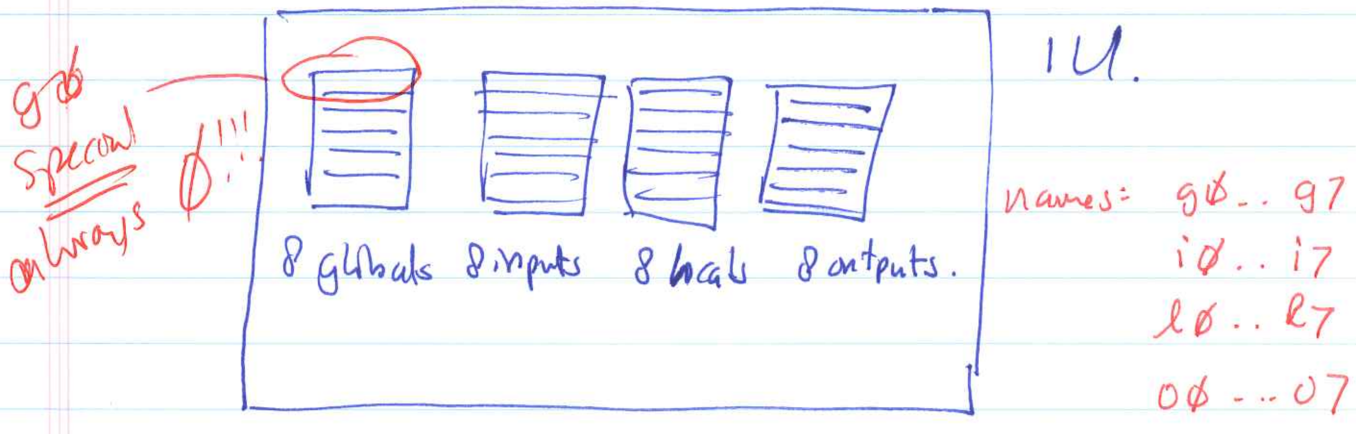
- The IU :

Schematically :



The SPARC IU

- Logically, the SPARC IU has 32 registers: organized into 4 sets of 8 registers.:



- Physically, the SPARC IU has 138 general purpose registers, divided into:

  8 global registers.

  128 window registers.

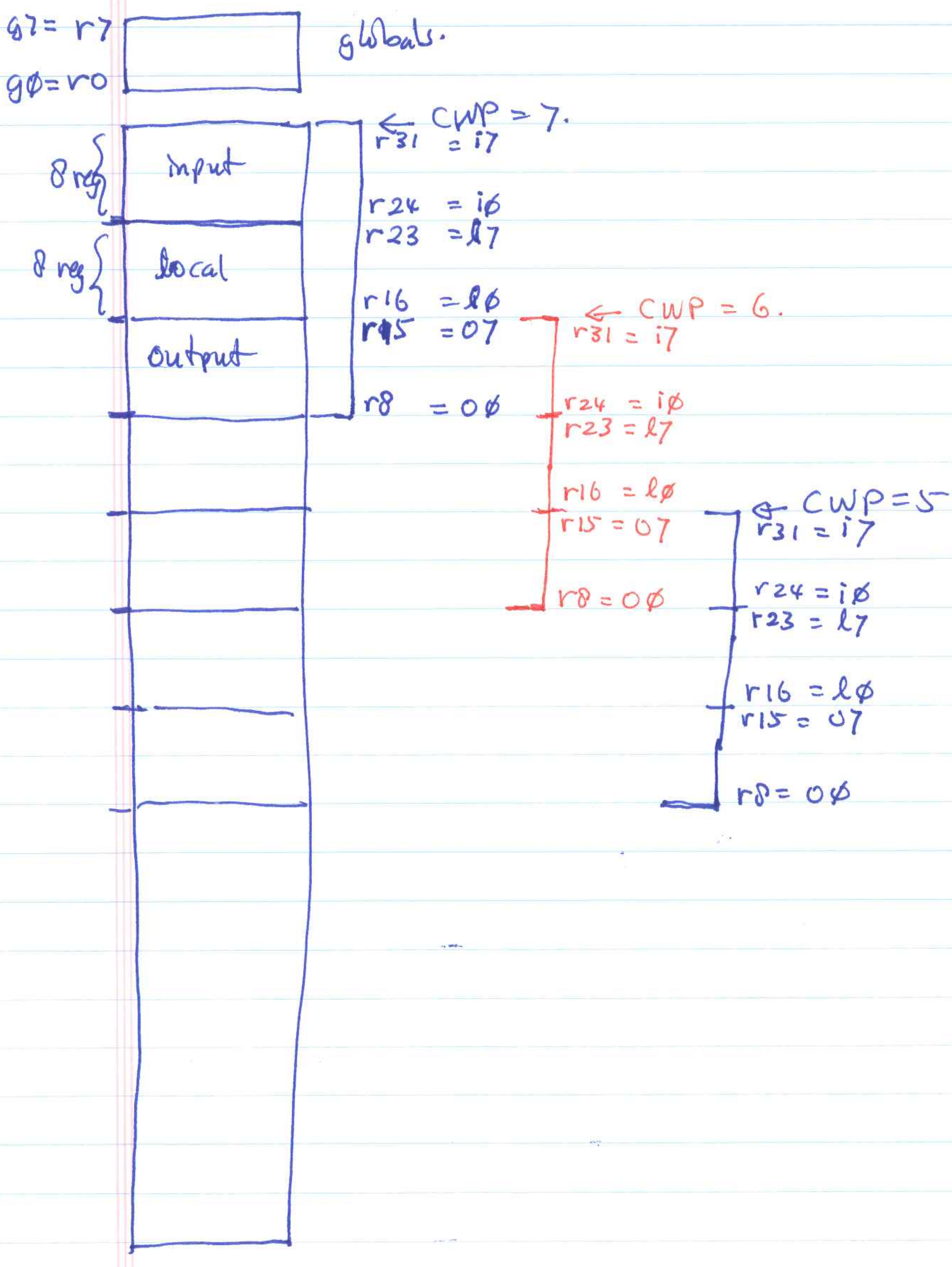
$\boxed{g0 \equiv 0}$ always

- The window registers are organized into 8 sets of overlapping register windows, numbered from 0 - 7.

- ~~Which~~ Each window contains 8 input, 8 local & 8 output registers.

- Which set of registers are visible depends on the value in CWP (current window pointer).

Window Register organization:



- "Output" registers in window 7 overlap with "input" registers in window 6.

That means:

the same register is referred to when

CWP = 7 and we refer to register ox
and CWP = 6 and we refer to register ix

x = ∅, 1, 2 ... 7.

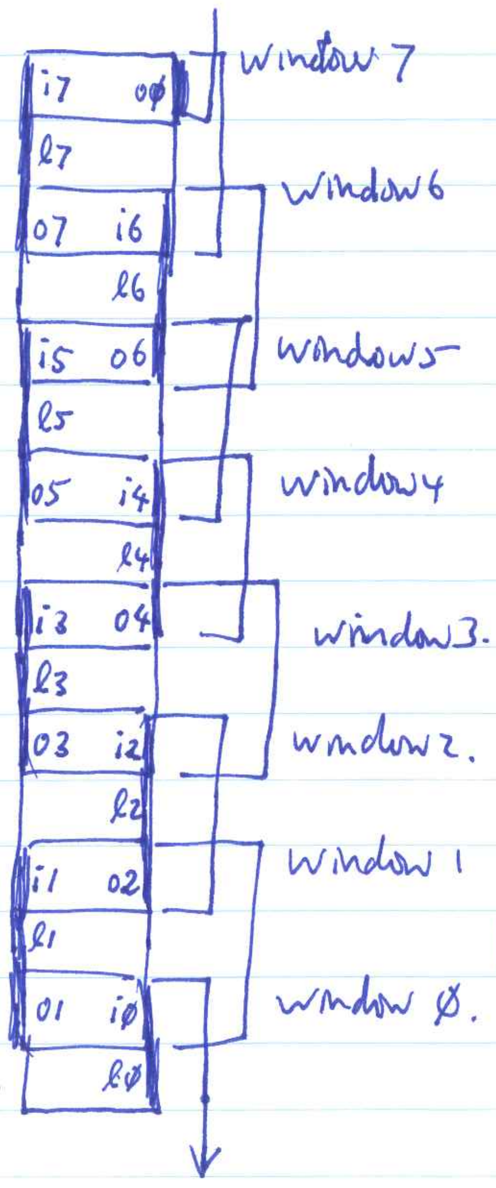
- Same is true in window 6 & window 5: output reg's in window 6 are the same registers as the input reg's in window 5.

So: the same register is referred to when:

CWP = 6 & we refere to reg. Ox
and CWP = 5 & we refer to reg. ix
x = 0, 1, 2, ... 7.

- Local registers do not overlap.
  - So local registers are private, ~~can not be~~ they can only be accessed if CWP is set correctly.

• ~~Finally~~, The window registers are organized in a circular manner:



Demo: sparc-WINDOWS

• We will discuss the usage later when we discuss subroutines.

- The Global registers are always accessible, regardless the value of CWP.

- Register %g∅ is special: it is ~~a~~ hardwired to the constant ∅.

  When g∅ is read, it returns the value ∅.
  When g∅ is written to, the effect is nil.

SPARC instruction set

- All SPARC instructions are 4 bytes (32 bits, 1 word) long and must start on a word boundary.
  - word boundary = address that is divisible by 4.
- SPARC can address 2^{32} bytes.
- Q: how many bits do you really need to specify the location of an instruction?
  - 30 (!!!) because the last 2 digits must be 00.

- Arithmetic & logic instructions has 3 operands:

  source 1 operand
  source 2 operand
  destination operand.

- SPARC is a Reduced Instruction Set Computer (RISC)
- A characteristic of RISC processors is very simple instruction encoding, very simple addressing modes.

- 4 instruction formats are used in SPARC :

type 1 :

	2	5	6	5	1	8	5	
1a		Dest	opcode	source1	0		Source 2	2 reg.
1b		Dest	opcode	source1	1	immediate constant		1 reg + 1 imm. Constant

10 or 11

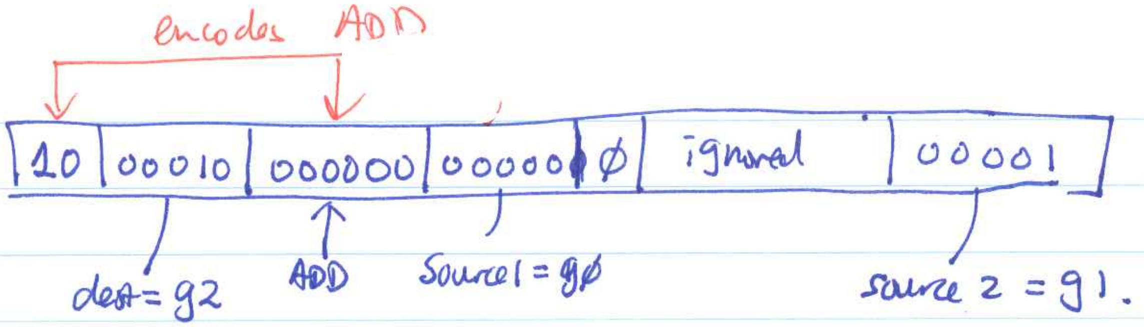
- Type 1 ~~format~~ format are used to encode arithm & logic operations , as well as load & store instructions.

- Load & store instructions move data between the SPARC CPU & the memory.
  ALL OTHER instructions operate on data in registers and DO NOT access memory.

- Format 1 a ~~[illegible]~~ is used to encode instructions with 2 source operands in registers and a destination operand in register.
  (5 bits encodes $2^5 = 32$ registers – 8 globals
  8 input
  8 local
  8 output
  32 registers.).

- Format 1 b is used to encode instructions with
  1 source operand in a register,
  1 source operand as immediate constant
  and dest. operand in register.

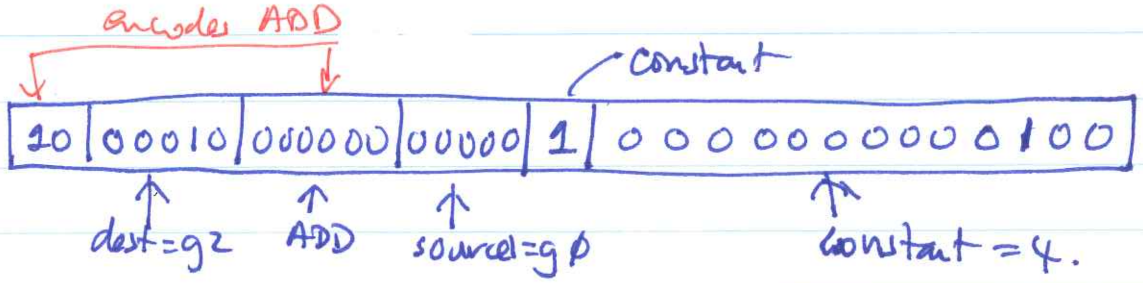
eg: 1a:



encodes: ADD %g0, %g1, %g2

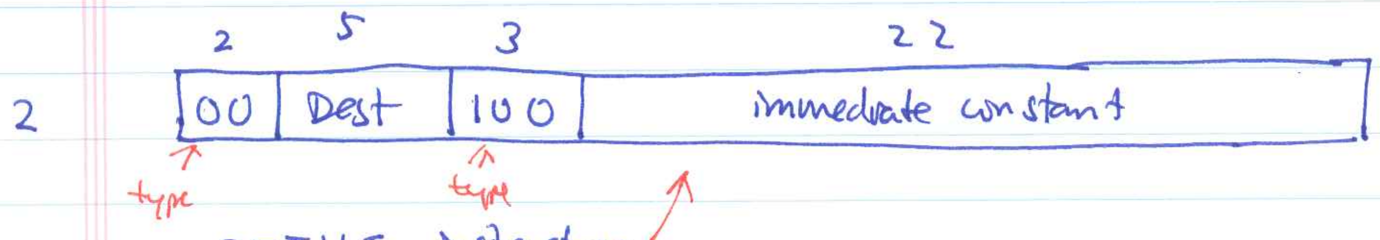
$\uparrow$ Source1 $\uparrow$ Source2 $\uparrow$ destination.

eg: 1b:

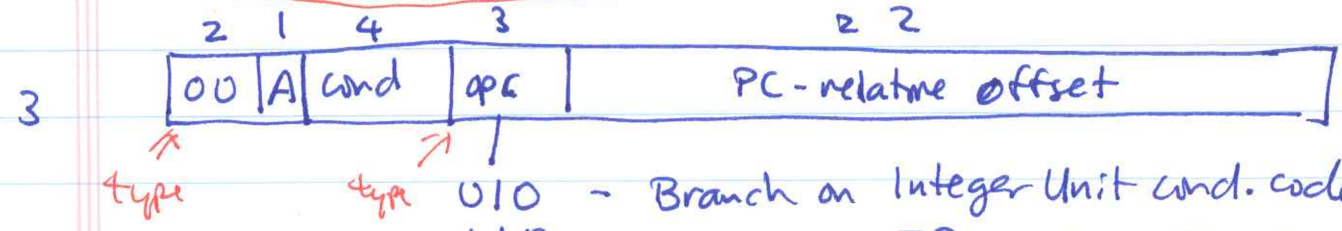


encodes: ADD %g0, 4, %g2

Format 2, 3, & 4 are used to encode special instructions:

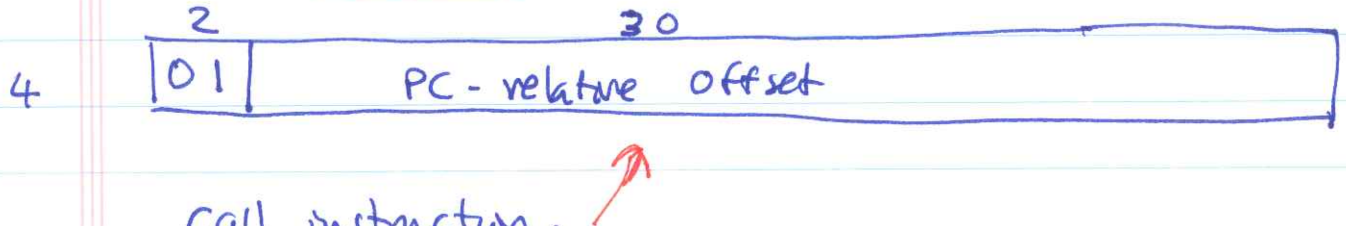


SETHI -instruction.



010 - Branch on Integer Unit cond. code.
110 - " " FP " " "
111 - " " Coproc. " " ".

Branch instructions.



Call instruction.

Alignment :

- SPARC can operate on data of size :

  byte
  half words (2 bytes)
  word (4 bytes) - word is 4 bytes in SPARC
  double words (8 bytes).

- LD & ST operation that reads & write to memory access the data and the data must be "aligned".

Rules: Byte size data can be stored at any addr.
half word size data must start at an addr. divisible by 2.
word size data must start at an addr. divisible by 4.
double words size data start at addr. divisible by. 8

NB: instruction starts at addr. div. by 4.

Assembler directives

.section ".text"

. ~~seg~~ "text" ~~start program instructions.~~

.section ".data"

. ~~seg "data"~~ ~~start~~ variable declarations.

Old

use .section

Note : . seg "text" & . seg "data" are used to tell the assembler what comes ~~of~~ next .

. align n = n can be 1, 2, 4 or 8.

tells the assembler to skip to next address that is divisible by n.

eg. since all instruction must start at word addr. location, you should

```
.seg "text"
.align 4
   ...  instruction.
```

eg. Word data like integer variable must start at word addr. location. You should also use :

```
.seg "data"
.align 4
   ... int variable
```

Reserving memory space :

label: .skip n (1) reserve n bytes
(2) assign memory addr. of first by to symb. const. "label".
(like label: ds.b n in M68000).

label: .word a,b,c,.... (1) reserve memory & init. the content to a, b, c ..., each location is 4 bytes.
(2) assign mem. addr. of first location to "label".

(like : label: ds.l a,b,c,...)

label: .half a,b, c, ... same, now half words (16 bits)

like label: ds.w a,b,c,...

label: .byte a,b,c,... same, now bytes.
like: label: ds.b a,b,c...

label: .ascii "text string" same as .byte, now converts string in ascii code.

Arithmetic & logic operations

- Each arithmetic & logic operations comes in 2 varieties:

  (1) one that will set the condition codes (flags NZVC)
  (2) one that will not.

- Arithmetic operations:

$-2^{+12} .. (2^{12}-1)$

ADD %r1, $\left\{ \begin{matrix} \boxed{\text{constant}} \\ \%r2 \end{matrix} \right\}$, %r3 $\qquad r3 = r_1 + \left\{ \begin{matrix} \text{const} \\ r2 \end{matrix} \right\}$

ADDCC %r1, $\left\{ \begin{matrix} \text{constant} \\ \%r2 \end{matrix} \right\}$, %r3 $\qquad r3 = r_1 + \left\{ \begin{matrix} \text{const} \\ r2 \end{matrix} \right\}$

set conditionflag (NZVC) $\qquad$ & set flags.

SUB %r1, $\left\{ \begin{matrix} \text{constant} \\ \%r2 \end{matrix} \right\}$, %r3 $\qquad r3 = r_1 - \left\{ \begin{matrix} \text{const} \\ r2 \end{matrix} \right\}$

SUBCC %r1, $\left\{ \begin{matrix} \text{constant} \\ \%r2 \end{matrix} \right\}$, %r3

- multiply

umul %r1, $\left\{ \begin{matrix} \text{constant} \\ \%r2 \end{matrix} \right\}$, %r3 $\qquad [Y, r3] = r1 * \left\{ \begin{matrix} \text{const} \\ r2 \end{matrix} \right\}$
unsigned

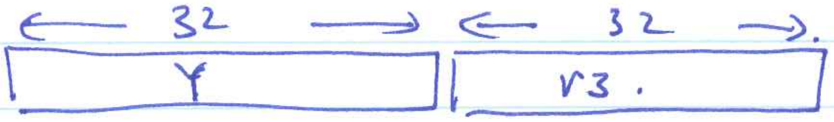
umulcc %r1, $\left\{ \begin{matrix} \text{constant} \\ \%r2 \end{matrix} \right\}$, %r3

smul %r1, $\left\{ \begin{matrix} \text{constant} \\ \%r2 \end{matrix} \right\}$, %r3 $\qquad [Y, r3] = r1 * \left\{ \begin{matrix} \text{const} \\ r2 \end{matrix} \right\}$
signed

smulcc %r1, $\left\{ \begin{matrix} \text{constant} \\ \%r2 \end{matrix} \right\}$, %r3

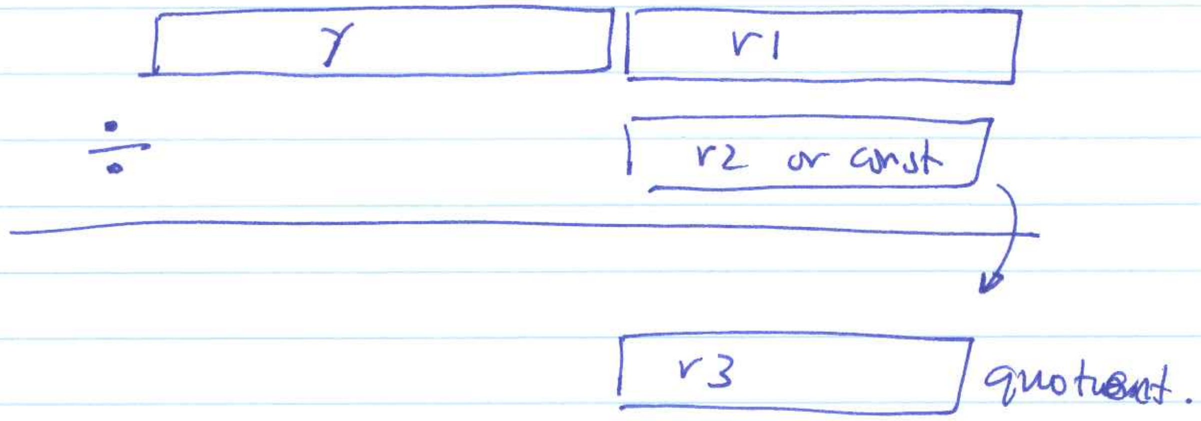
- The result $r_1 * \left\{ \begin{matrix} \text{const} \\ r_2 \end{matrix} \right\}$ is at most 64 bits.

  It is stored away in 2 registers :



- Divide :

  udiv %r1, $\left\{ \begin{matrix} \text{constant} \\ \text{\%r2} \end{matrix} \right\}$, %r3 — unsigned division



  r3 := [Y, r1] / r2 (quotient only).
  remainder is discarded.

  udivcc %r1, $\left\{ \begin{matrix} \text{constant} \\ \text{\%r2} \end{matrix} \right\}$, %r3

  sdiv %r1, $\left\{ \begin{matrix} \text{constant} \\ \text{\%r2} \end{matrix} \right\}$, %r3 — signed division.

  sdivcc %r1 , $\left\{ \begin{matrix} \text{constant} \\ \text{\%r2} \end{matrix} \right\}$, %r3.

- There is no negate instruction. Use:

  SUB %g0, %r1, %r1

  to negate ~~that~~ register r1.

- But assembler does recognize the mnemonic:

  neg %r1

Setting up & reading Y register:

(1) wr %r1, $\left\{ \begin{array}{l} \%r2 \\ \text{const} \end{array} \right\}$, %y

effect: y := %r1 XOR $\left\{ \begin{array}{l} \%r2 \\ \text{constant} \end{array} \right\}$.

sign extended to 32 bits.

(2) rd %y, %r1

effect: r1 := y.

Note: most of the time you don't need to use Y-reg. in multiplication because the result is 32 bits long.

Logic operators :

AND %r1 , $\left\{\begin{matrix}\text{const}\\ \%r2\end{matrix}\right\}$, %r3   r3 = r1 AND $\left\{\begin{matrix}\text{const}\\ r2\end{matrix}\right\}$

$\uparrow$ const is sign extended to 32 bits before AND is done.

ANDCC %r1 , $\left\{\begin{matrix}\text{const}\\ \%r2\end{matrix}\right\}$, %r3 .

OR %r1 , $\left\{\begin{matrix}\text{const}\\ \%r2\end{matrix}\right\}$, %r3

ORCC %r1 , $\left\{\begin{matrix}\text{const}\\ \%r2\end{matrix}\right\}$, %r3

Other logic operations :

XOR   exclusive or.

$\rightarrow$ 0 XNOR 0 = not(0) = 1
1 XNOR 0 = not(1) = 0

- SPARC has no NOT instruction :

use :   xnor %r1, %g0, %r1.

Assembler recognizes : not %r1.