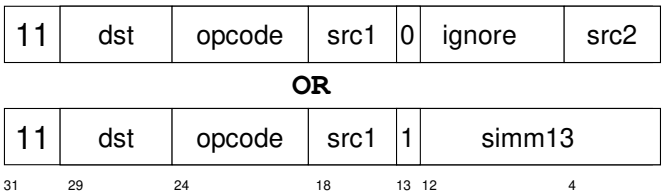
SPARC Instruction Set

CS 217

Load Instructions

- Move data from memory to a register
  - ld [u / s] [b / h / d] {a} [*address*], *reg*
- Examples:
  - `ld [%i1],%g2`
  - `ldud [%i1+%i2],%g3`

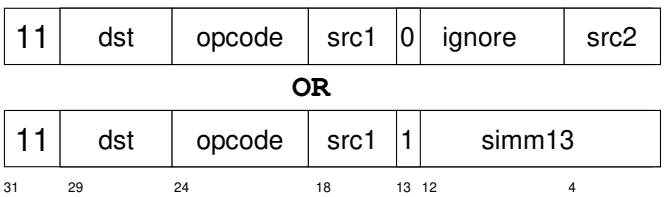


Load Instructions

- Move data from memory to a register
  - ld [u, s] [b, h, d] {a} [*address*], *reg*
- Details
  - fetched byte/halfword is right-justified
  - leftmost bits are zero-filled or sign-extended
  - double-word loaded into register pair*;* most significant word in *reg* (must be even); least significant in *reg+1*
  - address must be appropriately aligned

Store Instructions

- Move data from a register to memory
  - st [b, h, d] {a} *reg*, [*address*]
- Examples:
  - `st %g1,[%o2]`
  - `stb %g1,[%o2+o3]`

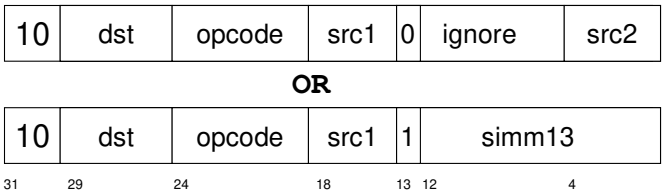


Store Instructions

- Move data from a register to memory
  - st [**b** | **h** | **d**] {a} *reg*, [*address*]
- Details
  - rightmost bits of byte/halfword are stored
  - leftmost bits of byte/halfword are ignored
  - *reg* must be even when storing double words

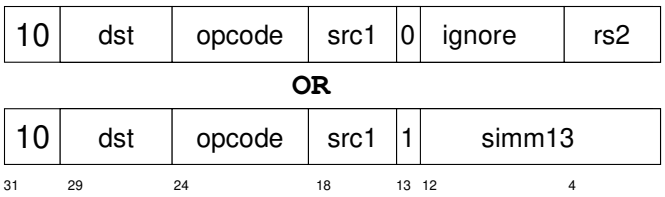
Arithmetic Instructions

- Arithmetic operations on data in registers
  - add{x}{cc} ***src1, src2, dst*** dst = src1 + src2
  - sub{x}{cc} ***src1, src2, dst*** dst = src1 - src2
- Examples:
  - **`add %o1,%o2,%g3`**
  - **`sub %i1,2,%g3`**
- Details
  - *src*1 and *dst* must be registers
  - *src2* may be a register or a signed 13-bit immediate



Bitwise Logical Instructions

- Logical operations on data in registers
  - and{cc} *src1, src2, dst* — *dst = src1 & src2*
  - andn{cc} *src1, src2, dst* — *dst = src1 & ~src2*
  - or{cc} *src1, src2, dst* — *dst = src1 | src2*
  - orn{cc} *src1, src2, dst* — *dst = src1 | ~src2*
  - xor{cc} *src1, src2, dst* — *dst = src1 ^ src2*
  - xnor{cc} *src1, src2, dst* — *dst = src1 ^ ~src2*

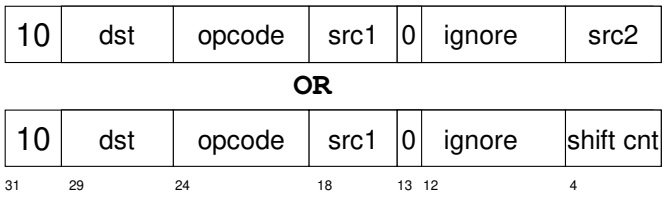


Shift Instructions

- Shift bits of data in registers
  - s [l / r] [l / a] *src1,* [*src2* / *0..31*], *dst*

sll: *dst = src1 << src2;*
slr: *dst = src1 >> src2;*

- Details
  - do not modify condition codes
  - sll and srl fill with 0, sra fills with sign bit
  - no sla



Floating Point Instructions

- Performed by floating point unit (FPU)
- Use 32 floating point registers: `%f0…%f31`
- Load and store instructions
  - ld [*address*],*freg*
  - ldd [*address*],*freg*
  - st *freg*,[*address*]
  - std *freg*,[*address*]
- Other instructions are FPU-specific
  - fmovs,fsqrt,fadd,fsub,fmul,fdiv,…

C Programs

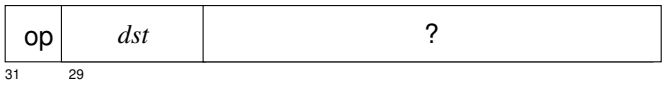
C Code	Assembly code

`x = a + 5000;`	`set a,%i1 (?)`
	`ld [%i1],%g1`
	
	`set 5000,%i1 (?)`
	
	`add %g1,%g2,%g1`
	
	`set x,%i1 (?)`
	`st %g1,[%i1]`

Data Movement

- How do we load a constant (e.g., address) into a register
  - set *value, dst ?*

Instruction format?



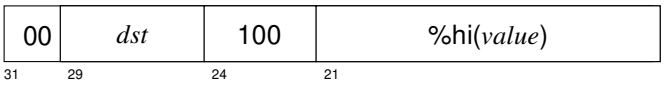
Data Movement

- Loading a constant (e.g., address) into a register

```
sethi %hi(value),dst
or dst,%lo(value),dst
```

- Details
  - if `%hi(`*value*`) == 0,` omit `sethi`
  - if `%lo(`*value*`) == 0,` omit `or`

sethi instruction format



Data Movement (cont)

- Example: direct addressing

```
set a,%g1          sethi %hi(a),%g1
ld [%g1],%g2       or %lo(a),%g1
                   ld [%g1],%g2
```

- Faster alternative

```
sethi%hi(a),%g1
ld [%g1+%lo(a)],%g2
```

Example

C Code	Assembly code

`x = a + 5000;`	`sethi%hi(a),%i1`
	`ld [%i1+%lo(a)],%g1`
	
	`sethi%hi(5000),%i1`
	`add %i1,%lo(5000),%g2`
	
	`add %g1,%g2,%g1`
	
	`sethi%hi(x),%i1`
	`st %g1,[%i1+%lo(x)]`

Synthetic Instructions

- Implemented by assembler with one or more "real" instructions; also called pseudo-instructions

Synthetic	Real
`mov src,dst`	`or %g0,src,dst`
`clr reg`	`add %g0,%g0,reg`
`clr [addr]`	`st %g0,[addr]`
`neg dst`	`sub %g0,dst,dst`
`neg src,dst`	`sub %g0,src,dst`
`inc dst`	`add dst,1,dst`
`dec dst`	`sub dst,1,dst`

Example Synthetic Instructions

- Complement
  - `neg reg` — `sub %g0,reg, reg`
  - `not reg` — `xnor reg,%g0,reg`
- Bit operations
  - `btst bits, reg` — `andcc reg, bits, %g0`
  - `bset bits, reg` — `or reg, bits, reg`
  - `bclr bits, reg` — `andn reg, bits, reg`
  - `btog bits, reg` — `xor reg, bits, reg`

Summary

- Assembly language
  - Provides convenient symbolic representation
  - Translated into machine language by assembler
- Instruction set
  - Use scarce resources (instruction bits) as effectively as possible
  - Key to good architecture design