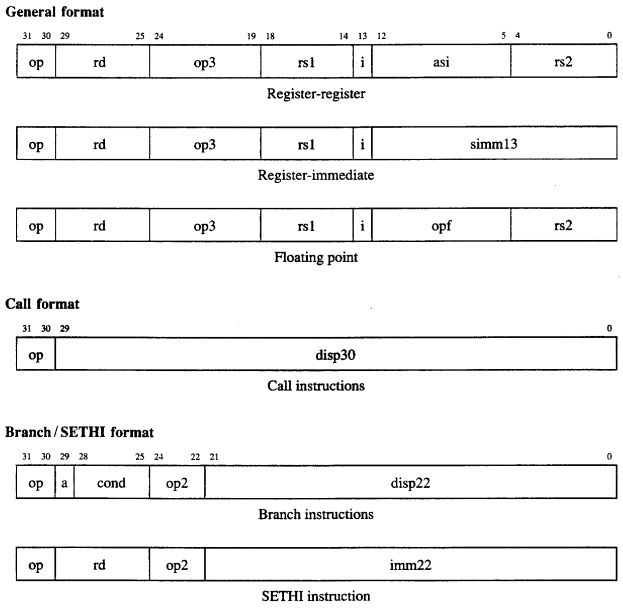
SPARC Overview

- The SPARC CPU, used in Sun/4 workstations, has a RISC LOAD / STORE architecture
  - All arithmetic and logical operations use either operands in registers, or immediate values
  - “ld” (load) and “st” (store) instructions are used to access memory
  - Memory is byte-addressable, but each ld & st operates on 32 bits
- The SPARC has 32 32-bit registers for use by the programmer
- The SPARC has a small instruction set, and a limited number of addressing modes
  - Each instruction is 32 bits wide

SPARC Registers

- The SPARC has 32 registers for use by the programmer
  - “%” indicates “register” in the “as” assembler (the assembler that we will be using)
  - Each register is 32 bits wide
- The registers are divided into 4 sets:
  - Global registers (%g0 – %g7) are used like global variables
    - %g0 is permanently set to zero
  - Local registers (%l0 – %l7) are used like local variables within a subroutine
  - In registers (%i0 – %i7) and out registers (%o0 – %o7) are used for subroutine parameter passing
    - %o6 and %o7 are reserved for special uses, and should not be used by the programmer

SPARC Instruction Formats



SPARC Addressing Modes

- Arithmetic instructions operate on either three registers, or two registers and an immediate value

```
add   %l1,%l2,%l3   ! %l3 = %l1 + %l2
add   %l1,100,%l3   ! %l3 = %l1 + 100
```

  - Note order — first two operands are source, last operand is destination
  - Immediate values (constants) are written as simple numbers, must be 2nd operand
- “ld” (load) and “st” (store) instructions operate on a pointer and a register

```
ld [%l1+4],%l5        ! %l5 = M[%l1+4]
ld [%l1+%l2],%l5      ! %l5 = M[%l1+%l2]
```

  - Pointer can be either value in register plus a constant, or sum of two register values
  - Second operand can be 0 or %g0

SPARC Arithmetic and Logical Instructions (Partial List)

- Basic arithmetic instructions

  add integer addition

  addcc same, but set condition codes

  sub, subcc integer subtraction…

  - Plus instructions for extended precision
  - There is ***no*** multiply or divide instruction!

- Basic logical instructions

  and, andcc bitwise and

  andn, andncc same, but with op2'

  or, orcc, orn, orncc bitwise or

  xor, xorcc, xorn, xorncc bitwise xor

- Plus shift instructions, and instructions for floating-point arithmetic

SPARC Synthetic Instructions (Partial List)

- Synthetic instructions are instructions recognized by the assembler, that are actually implemented in machine language by other instructions
- clr (clear)

```
clr     %l2             or  %g0,%g0,%l2
```

- inc (increment)

```
inc     %l2             add  %l2,1,%l2
```

- mov (move)

```
mov     %l2,%l3         or  %g0,%l2,%l3
mov     300,%l3         or  %g0,300,%l3
```

- set (load register with address)

```
set     addr,%l3        varies…
```

- Plus others not shown here…

The "set" Synthetic Instruction

- We have been using the "set" instruction to load an address into a register:

```
set    msg, %o0              ! load address of "msg"
```

- But — why not use a "ld" instruction??

```
load  msg, %o0               ! load address of "msg"
```

  - This doesn't work, because a "ld" instruction can only load immediate values up to 13 bits wide, and all addresses are 32 bits wide

- So we use a "set" synthetic instruction, which the assembler implements as:

```
sethi #hi(msg), %o0          ! load address of "msg"
or     %o0, #lo(msg), %o0
```

  - "sethi" loads a 22 bit immediate value into the 22 most significant bits of a register, and clears the 10 least significant bits
  - #hi( ) = 22 msbits #lo( ) = 10 lsbits

SPARC Assembler Programs

- We will use the assembler "as" on the Sun computer nimitz.mcs.kent.edu
  - This is ***not*** the same assembler as the one used in Maccabe
- General structure of programs:
  - One instruction per line
  - Labels are specified at the beginning of the line, followed by a colon (":")
  - Comments are prefaced with an exclamation point ("!"), and last until the end of the line
  - C-style comments ("/* … */") may also be used
- Assembler directives (pseudo-ops) begin with a period (".")

A Simple SPARC Assembler Program

```
    .data
msg:
    .ascii "Value is '%c'\n\0"      ! string for printf

    .text
    .global    _main                ! main must be global
    .global    _printf              ! linker will find printf
_main:
    save %sp, -64, %sp              ! space to save registers

    mov  "a", %o1                   ! load an 'a' to be printed
    set  msg, %o0                   ! load address of "msg"
    call _printf                    ! call printf to print out
    nop                             !   the character in %o1

    add  %o1, 1, %o1                ! convert 'a' into 'b'
    set  msg, %o0                   ! load address of "msg"
    call _printf                    ! call printf to print out
    nop                             !   the character in %o1

    sub  %o1, 040, %o1              ! convert 'b' into 'B'
    set  msg, %o0                   ! load address of "msg"
    call _printf                    ! call printf to print out
    nop                             !   the character in %o1

    mov  1, %g1                     ! exit request
    ta   0                          ! trap (return) to Unix
```


Notes on Simple Program

- Uses .data, .bss, and .text segment like the ones discussed in Maccabe
  - Initialized data: .byte, .word, .ascii
  - Uninitialized data: .skip
- ".global" marks a symbol as global
  - Every program must include a global symbol "_main", which is where loader will begin executing the program
- Parameters are passed to printf in %o0 (pointer to format) and %o1 (character)
- Things I can't explain quickly…
  - Always include "save…" at beginning of programs, and "mov…" and "ta…" at end of programs
  - For now, always include "nop" (no operation) after a subroutine call