CS201 W'00 Lecture Notes
Lecture 2

SPARC Architecture

SPARC = Scalable Processor ARChitecture

- One of original RISC machines.
- Based on research at UC Berkeley in mid-80's.
- Commercialized by SUN Microsystems.
- Licensed freely to other chip manufacturers.
- Other RISCs: MIPS, IBM RS/6000, DEC ALPHA, etc.

Key idea: Keep instructions simple, so that CPI = 1 and cycle time is fast.

Main features:

- Early models: 32-bit instructions, address space, registers
- V9 (ultra) models: 64-bit address space and registers; downwards compatible
- 32 (almost) general-purpose integer registers; register windows
- 3-address register-register instructions
- Condition codes control jumps
- Load/store architecture
- Separate FP registers
- Visible delay slots for pipelining

Registers

`%g0` – always 0

`%g1 - %g7` – globals, shared by all routines

`%o0 - %o5` – arguments (from point-of-view of caller or leaf callee)

`%sp (%o6)` – stack pointer

`%o7` – return address (from point-of-view of caller or leaf callee)

`%l0-%l7` – local values (not for leaf routine)

`%i0-%i5` – arguments (from point-of-view of non-leaf callee)

`%fp (%i6)` – frame pointer

`%i7` – return address (from point-of-view of non-leaf callee)

Condition Codes

Only set by explicit `cc` forms of operations

`Z` – set if result was zero

`N` – set if result was negative (2's complement)

`V` – set if result overflowed (2's complement)

`C` – set if result carried (unsigned)

Register-Register Instructions

Basics:

`add %l1,%l2,%l2`

– add contents of `%l1` and `%l2` and put result in `%l2`.

`addcc %l1,%l2,%l2`

– same, but set condition codes

Similarly: `sub`,`umul`,`smul`,`udiv`,`sdiv`,`and`,`or`,`xor`, etc.

Second operand can be small **immediate value** in the range [-4096,+4095].

Synthetics:

`mov %l1,%l2` == `or %g0,%l1,%l2`

`cmp %l1,%l2` == `subcc %l1,%l2,%g0`

`tst %l1` == `orcc %g0,%l1,%g0`

`inc %l1` == `add %l1,1,%l1`

etc.

More Register Instructions

Shifts:

`sll %l1,%l2,%l3`

– shift contents of `%l1` left by the number of bits specified by contents of `%l2`, and store result in `%l3`.

Similarly:

`srl` – shift right (logical)

`sra` – shift right (arithmetic – sign extending)

Large literals:

`set` *value*`, %l1`

– synthetic for

`sethi %hi(`*value*`),%l1;`
`or %l1,%lo(`*value*`),%l1`

`sethi` v`,%l1`

– zeroes low-order 10 bits of `%l1` and stores 22-bit value v into high-order 22 bits of `%l1`.

`%hi(`v`)` = rightshift(v,10)

`%lo(`v`)` = and(v,0x3ff)

Can sometimes replace `or` with more useful operation.

Note: `nop` == `sethi 0,%g0`

Memory Conventions

Memory is byte-addressable. Addresses are 4 bytes (V8) or 8 bytes (V9).

Multiple-byte units:

byte	1
half-word	2
word	4
double-word	8
quad-word	16
extended word (V9)	16

SPARC uses **big-endian** byte ordering within larger units (V9: can optionally select **little-endian**), i.e., most significant byte corresponds to lowest address:

So, after executing

```
set 0xAABBCCDD,%l1
mov 0x1000,%l0
st %l1, [%l0]
```

memory looks like this:

```
1000: AA
1001: BB
1002: CC
1003: DD
```


Memory Instructions

All data movement to/from memory use explicit store/load instructions.

`ld [%l1],%l2`

– load a word (4 bytes) from location v into `%l2`, where v is contents of `%l1`.

`ld [%l1+`*offset*`],%l2`

– load a word (4 bytes) from location v + *offset* into `%l2`, where v is contents of `%l1`. (Offset in range [-4096,+4095].)

`ld [%l1+%l3],%l2`

– load a word (4 bytes) from location $v + w$ into `%l2`, where v is contents of `%l1` and w is contents of `%l3`.

`ldsb` – load signed byte (sign extend)

`ldub` – load unsigned byte (no sign extend)

Similarly: `ldsh,lduh` for halfwords (2 bytes); `ldd` for double words (8 bytes).

Similarly: equivalent store operations `st,stb,sth,std`.

Alignment

All reads and writes of multiple-byte units must be **aligned** on addresses divisible by the size of the unit, so, e.g.,:

`st %l1,[%l2]`

is only valid if the contents of %l2 is divisible by 4.

Invalidly aligned accesses cause a **trap**, which is handled by the operating system and kills your program.

Also, instructions (always 4 byte words) must be word-aligned.

Register numbers for `ldd,std` must be even; two registers are read/written, with most-significant word in lower register.

E.g., if memory looks like this

```
1000:  AABBCCDD
1004:  EEFF0011
```

then after executing

```
mov 0x1000,%l0
ldd [%l0],%l2
```

we have:

```
%l2:  AABBCCDD
%l3:  EEFF0011
```

Branches

Branches can be **unconditional** or **conditional** on the values of the condition codes, e.g.,

`ba` – branch always

`bne` – branch not-equal (i.e, `Z` bit clear)

`bneg` – branch if negative (i.e., `N` bit set)

Branch targets are 22-bit signed **relative** displacements from the current pc.

They are then multiplied by 4 to form the full displacement (makes sense because instructions are always word aligned).

So range of possible branches is $[-2^{23}, +2^{23}-1]$, i.e., +/- 8MB.

Comparisons and conditionals

Conditional branches typically follow a `cmp` (synthetic) instruction

`cmp %l1,%l2` == `subcc %l1,%l2,%g0`

and get their names accordingly, e.g.,:

`bl` - branch if less (i.e, `N` bit is set or `V` bit is set, but not both)

Example:

```
cmp $r_1$,$r_2$
bl label   ! branch if $r_1 < r_2$
```

Why does this work?

$r_1 < r_2$ iff $r_1 - r_2 < 0$.

There are two cases:

(i) The subtraction did not overflow. Then V is clear and the sign bit is correct, i.e., set, so N is set.

(ii) The subtraction overflowed. Then V is set but the sign bit is incorrect, i.e., unset, so N is clear.

Jumps and Calls

`call` *label*

– store %pc into %o7, and jump to *label*.

Target address is specified as 30-bit signed displacement from the current pc; this is multiplied by 4 to form full displacement, so **any** address can be used.

`jmpl` *addr*, *reg*
– store %pc into *reg* and jump to *addr*, which is specified just as for loads/stores.

Supports computed jumps or calls, and subroutine return:

`retl == jmpl %o7+8, %g0` (return from leaf routines)
`ret == jmpl %i7+8, %g0` (return from non-leaf routines)

Delay slots

All branches, calls, and jumps are actually **delayed** by one instruction; i.e., the instruction immediately **following** the branch is executed **before** the branch occurs!

```
       clr %o3
       tst %o4
       be b                    ! branch iff %o4 == 0
       mov 1,%o3               ! ``delay slot''
  a:   ...                     ! here %o3 = 1
       ...
  b:   ...                     ! here %o3 = 1  (!!)
```

Why?? All SPARC implementations are **pipelined**: while the instruction at address *pc* is being executed, the one at $pc + 4$ is being fetched and decoded.

Recall that branches cause problems for pipelines: if the instruction at *pc* is a branch that gets taken, the instruction at $pc + 4$ should not be executed, so the work of decoding it is wasted.

By insisting that the instruction at $pc + 4$ is **always** executed anyway, the work won't be wasted after all.

What to put in a delay slot? `nop` is always safe, but the whole reason for exposing the pipelining is to allow programmer (or compiler) to put something useful in the slot.

Programming with the delay slot

Option 1: Move instruction from before the branch into slot:

```
      mov 1,%g2         ==>      ba c
      ba c                       mov 1,%g2
      nop
```

Option 2: Copy instruction from target into slot and adjust target:

```
      ba c                ==>      ba d
      nop                          mov 1,%g2
      ...                          ...
  c:  mov 1,%g2                c:  mov 1,%g2
      mov %g2,%g3              d:  mov %g2,%g3
```

Warning: testing of condition codes by branch occurs **before** delay slot instruction is executed.

```
      tst %o3
      bne b                  ! branch iff %o3 <> 0
      subcc %o3,%o3,%o3      ! ''delay slot''
  a:  ...                    ! here %o3 = 0
      ...
  b:  ...                    ! here %o3 = 0  (!!)
```

Don't try to put another branch in the delay slot! (Behavior is very weird.)

Annul Bits

Remember that the delay slot instruction is executed whether or not the branch occurs. E.g, using Option 2, might write following incorrect code:

```
 loop: inc %l4
       ...
       cmp %l4,10
       bne loop
       nop   ! delay slot
       ...   | here %l4 == 10

 ==>

 loop:  inc %l4
 loop1: ...
        cmp %l4,10
        bne loop1
        inc %l4 ! delay slot
        ...      ! here %l4 == 11
```

Can fix this problem by setting the **annul** bit in the conditional branch, which says **not** to execute the delay slot instruction if the conditional branch is not taken.

Specify this by adding ",a" to branch instruction, e.g.,

```
bne,a loop1
```


Additional Machine Features

Floating point registers & instructions

Conditional trap instructions to enter supervisor mode

Access to `%y` register (used for `mul,div`)

Privileged instructions to read/write processor registers

Multiprocessor synchronization

V9 instructions:

- 64-bit load, store, arithmetic
- Branch prediction bits
- Predicated moves
- Prefetch data
- Population count

Instruction formats

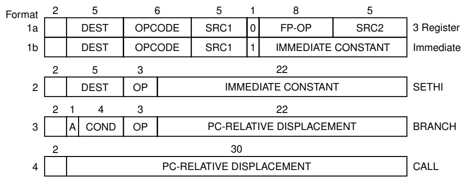

Figure 5-14. The original SPARC instruction formats.

Register and Calling Conventions

Want to keep useful values in registers, not memory. So **register assignment** is an important task of compilers (or assembly code writers).

But routines of large programs are often written and compiled **separately**. So assignment usually done on per-routine basis.

When one routine calls another, the caller and callee may both want to use the same registers. There must be a **register usage convention** to say who will **save** and **restore** the caller's values for any registers also used by the callee.

Also, need **calling convention** so one routine can pass values to/from another. Puts arguments in a fixed set of registers, so that every routine can find them. (These must also be saved when the callee itself becomes a caller.)

The **operating system** may also dictate conventions. E.g., if the O/S insists that it be allowed to push new data onto the stack at any time, then application code must keep the stack pointer register valid at all times.

Conventions may be described in an official **Applications Binary Interface**. Or, since most current processors have been designed to run C as their primary high-level language, whatever the vendor's C compiler and standard O/S does becomes the default convention.

Register Windows

On most machines, saving and restoring registers is done by explicit memory stores and loads. To allow **recursion**, values are normally saved on the **stack**, along with **control information** like the caller's return address, and any **spilled** values that don't fit in the available registers.

But SPARCs use a different idea called **register windows**. A SPARC processor has lots of **physical** registers, but each routine can see only a 32-register **window** onto the whole set.

When a call occurs, the callee gets to access a (mostly) different set of physical registers, even though it uses the same register **names** as the caller. Some registers are shared with the caller, to permit efficient argument passing.

More precisely, the register window for a routine contains:

8 `%i` registers shared with the caller

8 `%l` registers local to the routine

8 `%o` registers shared with any callee of the routine

In addition, every routine can see the

8 `%g` registers global to all routines

Because of the overlap, each window actually requires just 16 physical registers.

How many register windows? Depends: 2 to 32.

Register Windows

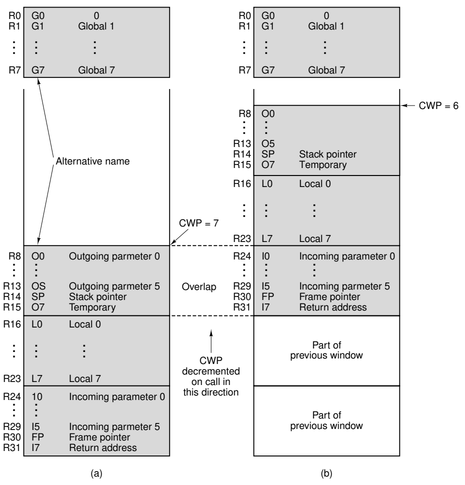

Figure 5-5. Operation of the UltraSPARC II register windows.

Programming with Register Windows

Routines using register windows must include a `save` instruction at the top, which gives access to a new window, and a `store` instruction at the bottom, to reset to the previous window.

The effect of `save` is that:

`%i` registers now point to same physical registers that `%o` registers did.

`%l` and `%o` now point to fresh physical registers.

Executing `restore` simply reverses the effect of last `save`.

Both `save` and `restore` also act like ordinary `add` insturctions, except that their arguments are read from the **old** window, and their result is written to the **new** window.

The renaming done by `save` sets `%fp (= %i6)` to the contents of `%sp (= %o6)`. So `save` is normally used to adjust the stack pointer to a new value, e.g.,

`save %sp,-`*framesize*`,%sp`

Then `restore == restore %g0,%g0,%g0`

thus resets the stack pointer to the frame pointer.

Also, the renaming makes the return address stored by `call` into `%o7` accessible in `%i7` and allocates a new physical register to `%o7`, ready for another call.

Standard Calling Convention

1. Caller puts arguments into `%o0,%o1,...`.

2. Caller executes `call` instruction, which stores return address into `%o7`.

3. Callee executes `save` instruction, which renames registers as above and allocates a new stack frame.

4. Callee accesses arguments via `%i0,%i1,...`.

5. Callee makes free use of `%i` and `%l` registers.

6. Callee stores result value in `%i0`.

7. Callee executes `ret; restore` sequence of instructions, which does indirect jump to return address stored in `%i7`, and – in the delay slot – reverses the effect of `save`.

8. Caller retrieves result value from `%o0`.

Leaf routines that make no calls, need no local storage, and need few registers don't need a new window:

3. Nop (note: no new frame either).

4. Callee accesses arguments via `%o0,%o1,...`.

5. Callee makes free use of `%o` registers.

6. Callee stores result value in `%o0`.

7. Callee executes `retl` instruction, which does indirect jump to return address stored in `%o7`.

More Conventions

SPARC conventions about the layout of stack frames are quite complicated.

Here are simple rules to follow:

- Don't alter `%sp` except in a `save %sp,-`*framesize*`,%sp` instruction at the top of your routine.
- `%sp` (and hence *framesize*) must is always be a multiple of 8.
- The value of *framesize* must always be at least 96 bytes, of which the first 92 bytes (above %sp) must **not** be used by your code.
- If your routine needs local storage on the stack, start it at `%sp+92`, and adjust the overall framesize accordingly (remembering also the multiple of 8 rule).
- If you need to pass more than 6 words worth of arguments to a C routine, ask an expert.

Registers:

- Don't use `%g` registers except for `%g0` and `%g1`.
- If you want any floating point registers to remain valid over a call, you must save them yourself (remembering to make room in the current frame).

Examples

Caller (C code):

```
int sum3(int,int,int);
int i = sum3(1,2,3);
```

Caller (corresponding assembly code):

```
    mov 1, %o0
    mov 2, %o1
    call sum3
    mov 3, %o2   ! delay slot
    mov %o0, %l7 ! assuming %l7 holds variable ``i''
```

Assume `sum3` is intended to return the sum of its three args.

Callee (assembly code) as leaf routine:

```
.section ".text"
    .align 4
    .global sum3
sum3:
    add %o0,%o1,%o0  !
    retl             ! use ``retl'' not ``ret''
    add %o0,%o2,%o0  ! use delay slot
```


Examples

Suppose `sumsq3` returns the sum of squares of its three args.

Callee (assembly code) as non-leaf routine:

```
.section ".text"
    .align 4
    .global sumsq3
sumsq3:
    save %sp,-96,%sp ! minimum stack frame
    mul %i0,%i0,%o0  ! square each arg
    mul %i1,%i1,%o1
    call sum3
    mul %i2,%i2,%o2  ! delay slot
    mov %o0,%i0      ! copy result back
    ret
    restore          ! normal use of delay slot
```

Mechanics

To assemble `foo.s`, just feed it to `cc`, which will automatically pass it to the assembler (`as`) because of it's `.s` extension.

Examples:

To assemble `foo.s`, compile `bar.c`, and link them together into an executable `bar`:

```
cc -o bar bar.c foo.s
```

To assemble `foo.s` separately into a binary file `foo.o`:

```
cc -c foo.s
```

To compile `bar.c` separately into a binary file `bar.o`:

```
cc -c bar.c
```

To link the `.o` files together into an executable `bar`:

```
cc -o bar foo.o bar.o
```


C Interfacing

Arguments are passed in order, starting with `%o0`.

• Ordinary **integers** occupy one argument word.

• **Addresses** also occupy one word.

• Double-precision **floating point** numbers occupy two words.

• **Structures** occupy multiple words.

Return values:

• **Integer** return values come back in `%o0`.

• **Floating point** return values come back in `%f0`.)

• **Structure** return values are messy.

In C:

• Passing a variable `x` (other than an array) means passing the **value** of `x`.

• To pass the **address** of a variable `x`, pass `&x`. This way, the callee can "return a result" by updating the variable.

• Passing an array `x` means passing the address of `x[0]`; it's exactly the same as passing `&x[0]`.

• Strings are arrays!