The SPARC Architecture

Jingke Li

Portland State University

The SPARC Architecture

SPARC = *S*calable *P*rocessor *ARC*hitecture

- One of the ealiest RISC processor architectures, developed from research at UC Berkeley (ca. 1980).
- Commercialized by SUN Microsystems, licensed freely to others.

Main Features:

Simple, uniform instruction set allowing fast cycle times.
(Goal — "One instruction per cycle.")

- Up to 128 general-purpose registers
- All arithmetic ops are register-to-register
- Only simple load/store to memory
- Register windows

Versions of the SPARC Architecture

- V7 (1986) — 32-bit, the first published version.
- V8 (1990) — 32-bit, added hardware multiply and divide.
- V9 (1993) — 64-bit, extended the floating-point register file.

The UltraSPARC Processors:

- *Implement the SPARC V9* — All arithmetic is performed to 64-bit precision; 64-bit virtual memory addresses.
- *Superscalar CPU* — More than one instruction can be issued at a time for execution.

Registers

Physical vs. Visible Registers:

- *For CISC architectures* — Typically all general-purpose physical registers are directly addressable in the user program.
- *For RISC architectures* — There can be hundreds of general-purpose registers on a processor. Directly addressing them in programs would make instructions more complex (e.g. more bits are need to represent each register) and register savings more expensive. So typically, only a portion of the physical registers are made visible to a program at any one time.

The SPARC has up to 128 general-purpose physical registers (actual number depends on implementation). However, only 32 registers are visible to a program at any time.

SPARC's Visible Registers

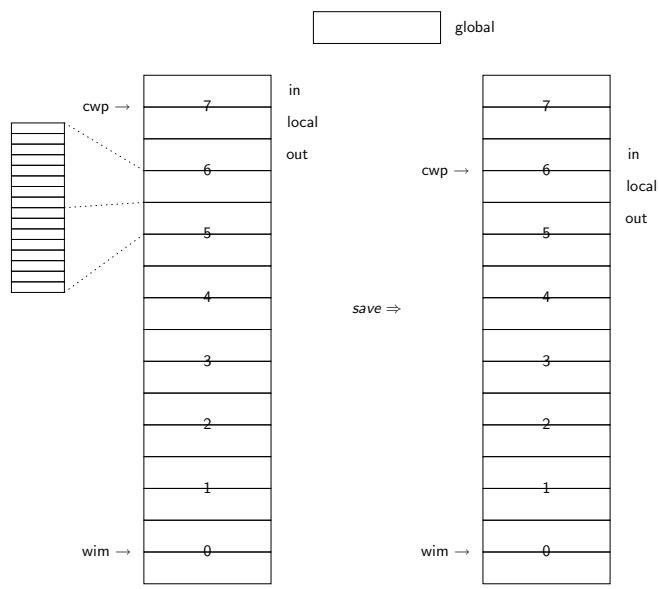
SPARC provides 32 registers for each (active) subroutine, which are organized into four sets:

- %g0–%g7 — for global data
- %l0–%l7 — for local data
- %i0–%i7 — for incoming arguments
- %o0–%o7 — for arguments to a subroutine

Register names look alike in each routine, but actual registers referred to are different: the 8 global registers are mapped directly to 8 physical registers; while the other 24 registers are mapped to one of many *windows* on a large register file.

Register Windows

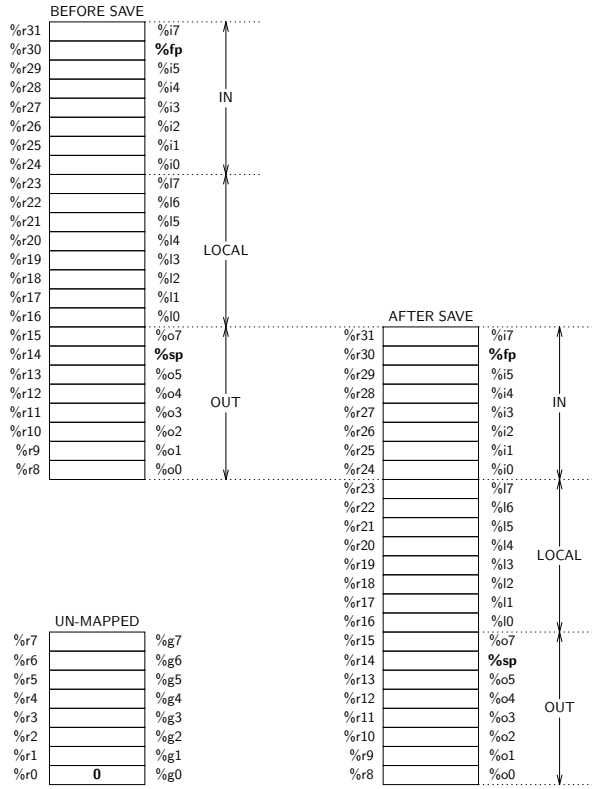
- A register file contains many (overlapping) windows; each consists of 3 sets of registers (24 registers in total).
- Only one window is active (visible) at any time.
- The active window shifts by 2 sets (16 registers) each time a procedure is called.



- When windows run out — Dumps window's contents onto the stack (*expensive!*). (Also, 64-byte space on the stack must be reserved for each call at all times.)

The Shifting of Active Register Window

- Caller's and callee's register windows overlap — they share one set of registers.
- The shared registers can pass arguments seamlessly from caller to callee — no additional data copying needed.



Benefits and Issues of Register Windows

The use of register windows provides the following benefits:

- It enables (up to 8) arguments to be passed from a caller to a callee in registers *automatically*.
- It also enables the processor designer to independently decide how many physical registers to implement.

The design works great as long as the physically registers do not run out.

However, studies show that long chains of procedure calls are not that uncommon in modern (especially OOP) programs. Furthermore, many procedures have none or few parameters. These findings question the wisdom of dedicating 8 registers for *every* procedural call.

Improvements:

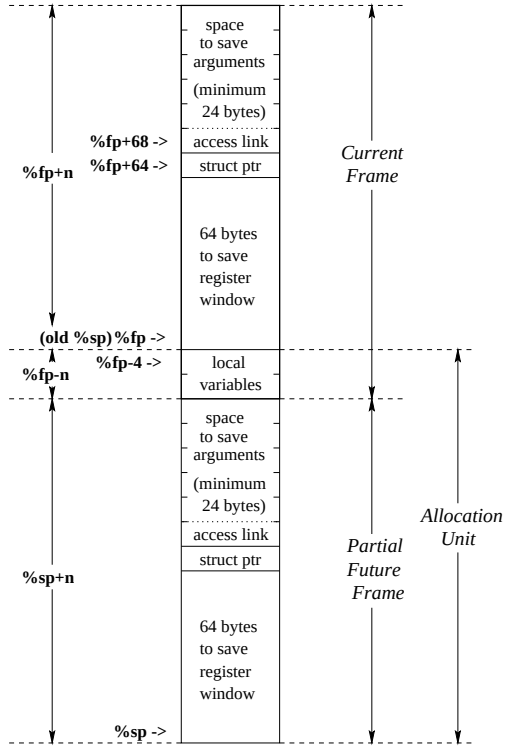
- Allowing the register windows to be of *variable size* (e.g. AMD 29k).
- Using the more powerful *register renaming* technique (e.g. IA-64).

SPARC Stack Organization

The runtime stack is for storing procedure-call related dynamic data.
There are two key pointers:
`%fp` — frame pointer
`%sp` — stack pointer

Stack Frame Layout:
(The stack grows downward.)

- The 16 words just above the frame pointer are reserved for potential register-window dumping.
- The slot at `[%fp+64]` is reserved for pointer to aggregate return value.



SPARC Stack Organization (cont.)

- Local variables are stored below the frame pointer and are accessed with negative offsets; the first one starts at `[%fp-4]`.
- The parameters start at `[%fp+68]`. The first six words are always reserved for the first six parameters (even if there is no parameters at all). If there are more than six parameters, the extra ones continue from `[%fp+92]`.
- There is no need to set up control links. The old stack pointer `%sp` (i.e. `%o6`) automatically becomes the new frame pointer `%fp` (i.e. `%i6`) when control is transferred from caller to callee.
- Minimum frame allocation size:

parameters:	≥ 24
saving register window:	64
struct pointer:	4
local variable:	≥ 0
total:	≥ 92 (rounds up to 96)

Pipelining

On modern processors, the execution of an instruction is decomposed into a sequence of steps, for example:

- *Instruction Fetch* — Fetch and decode the instruction, obtain any operands from the register file.
- *Execute* — Execute an arithmetic instruction; compute a memory address for a branch, load, or store instruction.
- *Memory Access* — Access memory for a load or store instruction; fetch the branch target instruction.
- *Store Results* — Write the results back to the register file.

While the steps from the same sequence must be executed in order, the sequences from different instructions can often be executed in a *pipelined* form.

Pipelining (cont.)

- *Non-Pipelined Execution:*

```
-------------------------------------------------------------
Fetch   Execute Memory  Store   Fetch  Execute Memory  Store
-------------------------------------------------------------
  1       2       3       4       5       6       7       8
```

  Each of the four hardware components sits idle 75% of the time.

- *Pipelined Execution:*

```
-------------------------------------------------------------
Fetch   Execute Memory  Store
        Fetch   Execute Memory  Store
                Fetch   Execute Memory  Store
                        Fetch   Execute Memory  Store
                                Fetch   Execute Memory  Store
-------------------------------------------------------------
  1       2       3       4       5       6       7       8
```

  Except for the leading in and leading out sections, the hardware components are fully utilized.

Issues with Load Instructions

The use of the loaded data must wait for the memory access of the load instruction:

```
load [%o0], %o1
add  %o1, %o2, %o2

-----------------------------------------------
Fetch   Execute  Memory  Store
        --       --      --      --
                 Fetch   Execute Memory  Store
-----------------------------------------------
```


Issues with Control-Transfer Instructions

The fetch of the target instruction must wait for the execution of the branch instruction:

```
    bl    loop
or
    call  foo

-----------------------------------------------
Fetch   Execute                                 <-- fetch target addr
        --      --      --      --              <-- delay slot
                Fetch   Execute Memory  Store
-----------------------------------------------
```

The skipped cycle is called the *branch delay slot*, and it corresponds to an explict instruction in the SPARC architecture — the *next* instruction after the branch instruction.

```
call  foo
nop         <-- delay slot instruction
```

Delay Slot Instructions

SPARC has two program counters, PC and nPC, which hold *next-inst* and *next-next-inst* pointers, respectively. The general instruction execution model is:

> Fetch an instruction with the address in PC, execute it;
> copy nPC into PC; and update nPC to the next address.

For a control-transfer instruction, the instruction in the delay slot (which is pointed to by nPC) will be executed *before* the actual control transfer:

```
                                        order
----------------------------------------
    call foo                            2
    mov 3, %0                           1

foo:
    save    %sp, -112, %sp              3
    ...
----------------------------------------
```


Optimization Using Delay Slots

Source Code: `if (a > b) c = a; else c = b;`

- *Unoptimized Target Code* — 5 cycles

```
    cmp a, b
    mov b, c
    ble L1
    nop
    mov a, c
L1:
```

- *Optimized Target Code* — 4 cycles

```
    cmp a, b
    ble L1
    mov b, c
    mov a, c
L1:
```

The Annul Bit

What if the branch with an instruction in its delay slot does not take?

Use the *annul bit* to control the execution of the delay-slot instruction:

```
bg,a L1
mov a,c
```

the annul bit ",a" causes the `mov` instruction *not* to be executed if the branch is *not* taken, but still executed if the branch is taken.

Exception — `ba,a L` does not execute its delay-slot instruction.

Common Wisdom:

- Don't use annul bit unless absolutely necessary
- Place `nop` after control-transfer instructions if not sure what to put

SPARC Instructions

- *Arithmetic:*
  `add, sub, smul, sdiv, ...`
- *Logical:*
  `and, andn, or, orn, xor, xorn, ...`
- *Shifts:*
  `sll, srl, sra, ...`
- *Loads and Stores:*
  `ldsb, ldsh, ldub, lduh, ld, ldd, stb, sth, st, std, ...`
- *Branch:*
  `ba, bn, bne, be, bg, ble, bge, bl, bgu, bleu, ...`
- *Controls:*
  `save, restore, call, jmpl, ...`
- *Miscellaneous:*
  `sethi, ...`

SPARC Instruction Examples

- *Simple arithmetics:*

```
add %i1, 1, %o0
sll %o0, 2, %o0
srl %i0, 31, %g2
```

- *Procedure call interface:*

Caller:

```
    jmpl %o0, %o7
or
    call %o0
```

Write %pc to %o7 and perform a delayed jump to the address in %o0.

Callee:

```
    save %sp, -112, %sp
    ...
    jmpl %i7+8, %g0
or
    save %sp, -112, %sp
    ...
    ret
```

- *An idiom for loading a 32-bit value to a register:*

```
sethi %hi(0x123cf456), %o0
or    %o0, %lo(0x123cf456), %o0
```


Assembly Code Example

```
------------------
int foo(int a) {
  return a+1;
}

int main() {
  int b = foo(3);
  printf("%d\n",b);
  return 0;
}
------------------
```

```
.section    ".text"
foo:
    retl
    add     %o0, 1, %o0
main:
    save    %sp, -112, %sp
    call    foo, 0
    mov     3, %o0
    mov     %o0, %o1
    sethi   %hi(.LLC0), %o0
    call    printf, 0
    or      %o0, %lo(.LLC0), %o0
    ret
    restore %g0, 0, %o0

.section    ".rodata"
.LLC0:
    .asciz  "%d\n"
```