CS302 Languages and Compiler Design II

Lecture 2

Introduction to SPARC Architecture

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

One of original RISC machines

- RISC = **R**educed **I**nstruction **S**et **C**omputer
- Developed from research at U.C.-Berkeley ca. 1980
- Commercialized by Sun Microsystems
- Licensed freely to others.
- Similar, parallel effort at Stanford led to MIPS machine
- Also IBM RS/6000, HP/PA, DEC ALPHA, etc.

Key idea: simple uniform instruction set allowing **fast** cycle times (and fast time to market)

Has eclipsed CISC machines (C = **C**omplex) like the DEC VAX, but not the Intel x86 line.

SPARC

~ 32 general-purpose registers

Only simple load/store to memory

```
ld [%r2], %r3
ld [%r2 + 12], %r3
ld [%r2 + %r7], %r3  ; 13 bit signed offset
```

All arithmetic/logical ops are register-to-register

```
add %r2, %r1, %r1
add %r2, 17, %r5     ; 13 bit signed immediate
```

Ops set condition codes, which are tested by `Bicc` ops, e.g.

```
bne label  ; branch if last op result <> 0
nle label  ; branch if last op result <= 0
```

Special instruction to load high-order bits of full-word constant

```
sethi %hi(value), %r7  ; net effect is to load
or %r7,%lo(value), %r7 ; all of 'value' into r7
```


SPARC - Delayed Branches

Implementation is **pipelined**.

It takes time to process a branch, because must fetch and decode target instruction

So why not execute a useful instruction in the meantime?

Specify this instruction by placing it in the **delay slot:**

```
tst %r6  ; executes first
bne L1   ; executes third
inc %r5  ; delay slot: executes second, whether
         ; or not branch is executed
```

Also applies to calls and returns:

```
call _fred
move 99, %o0   ; executes before call
move %o0, %l7  ; executes after call
```

Loads, stores, floating point ops may also cause delays, but programmer generally can't see these directly.

SPARC - Register Windows

Very important to keep data (parameters, locals, temporaries, etc.) in registers as much as possible.

Subroutines normally mess up register allocation, because registers must be saved at a call (at least if the callee is "unknown.")

This can be done before the call ("caller-save") or after ("callee-save"); in either case, register contents must be stored on stack, which is **slow**.

SPARC idea: provide **multiple** separate sets of registers, one for each active subroutine.

Register **names** look alike in each routine, but **actual** registers referred to are different.

The visible set of registers is called the **register window**.

SPARC - Registers

Actually want some of the visible registers to be shared between caller/callee, to pass arguments → "overlapping windows."

There are 32 registers visible in each routine

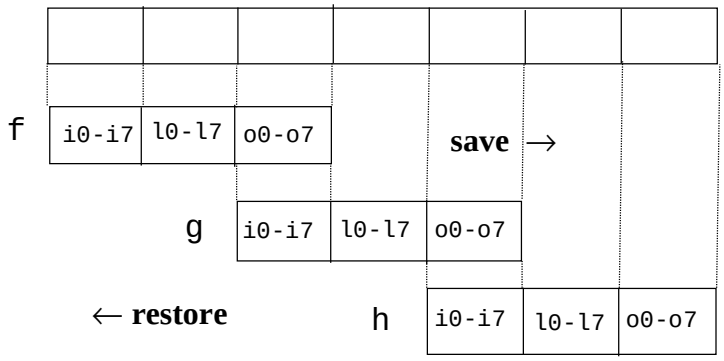
- 8 are **global** (shared among all routines)
  `%g0,…,%g7  (= %r0,…%r7)`
- 8 are **local** to routine
  `%l0,…,%l7   (= %r16,…,%r23)`
- 8 are shared with **caller**, so contain "**inputs**"; also contain return values
  `%i0 ,…,%i7 (= %r24,…,%r31)  %fp = %i6`
- 8 are shared with **callee**, so are "**outputs**"
  `%o0,…,%o7 (= %r8,…, %r15)   %sp = %o6`

SPARC - Register Window Shifts

At top of each routine, a `save` instruction is used to shift the window, so that the caller's "out" registers become the callee's "in" registers.

The `restore` instruction reverses the process.

Suppose `f` calls `g` and `g` calls `h`.



SPARC - More register windows

How many windows can exist at one time?

Answer: Implementation-dependent.

Example: Most current machines have 7 windows (I believe) for total of

- 8 global
- + 7 x 8 local
- + 7 x 8 shared input/output
- = 120 physical registers

What happens when we run out of windows while trying to do a `save`?

- Hardware traps to O/S, which dumps window contents onto the stack!
- This also happens during a context switch!
- User code **must** co-operate by leaving 16 words free above %sp at all times!

Is all this worth it? Dubious.

SPARC Activation Records (for C)

`%sp` - always points to true top of stack.

`%fp` - "frame pointer" - points to top of stack at entry to procedure.

The `save` instruction creates a stack frame, by subtracting a specified number of bytes from `%sp` and switching windows; **old** `%sp` becomes **new** `%fp`.

For assignment 1, **minimum** frame must be big enough to hold:

- 0 words for locals, temporaries (you may have more).
- 0 words for outgoing parameters beyond 6th (if any call from procedure needs them)
- 7 words every callee is allowed to assume are there.
- 16 words hardware is allowed to assume is there.
- Total: 23 words * 4 bytes/word = 92 bytes

Frame size must always be a multiple of 8 bytes, so round up to 96 bytes.