12. Register Targeting

Compute values directly into the intended target register.

13. Interprocedural Code Motion

Move instructions across subprogram boundaries.

14. Call Inlining

At the site of a call, insert the body of a subprogram, with actual parameters initializing formal parameters.

15. Code Hoisting and Sinking

If the same code sequence appears in two or more alternative execution paths, the code may be *hoisted* to a common ancestor or *sunk* to a common successor. (This reduces code size, but does not reduce instruction count.)

16. Loop Unrolling

Replace a loop body executed N times with an expanded loop body consisting of M copies of the loop body. This expanded loop body is executed N/M times, reducing loop overhead and increasing optimization possibilities within the expanded loop body.

17. Software Pipelining

A value needed in iteration *i* of a loop is computed during iteration *i-1* (or *i-2*, ...). This allows long latency operations (floating point divides and square roots, low hit-ratio loads) to execute in parallel with other operations. Software pipelining is sometimes called *symbolic loop unrolling*.

18. Strength Reduction

Replace an expensive instruction with an equivalent but cheaper alternative. For example a division may be replaced by multiplication of a reciprocal, or a list append may be replaced by cons operations.

19. Data Cache Optimizations

- Locality Optimizations

  Cluster accesses of data values both spacially (within a cache line) and temporally (for repeated use). *Loop interchange* and *loop tiling* improve temporal locality.

- Conflict Optimizations

  Adjust data locations so that data used consecutively and repeatedly don't share the same cache location.

20. Instruction Cache Optimizations

Instructions that are repeatedly re-executed should be accessed from the instruction cache rather than the secondary cache or memory. Loops and "hot" instruction sequences should fit within the cache.

Temporally close instruction sequences should not map to conflicting cache locations.

Reading Assignment

- Read "Modern Microprocessors—A 90 Minute Guide!," by Jason Patterson.

SPARC Overview

- SPARC is an acronym for
  Scalable Processor ARChitecture
- SPARCs are load/store RISC processors
  Load/store means only loads and stores access memory directly.
  RISC (Reduced Instruction Set Computer) means the architecture is simplified with a limited number of instruction formats and addressing modes.

- Instruction format:

  `add %r1,%r2,%r3`

  Registers are prefixed with a `%`

  Result is stored into last operand.

  `ld [adr],%r1`

  Memory addresses (used only in loads and stores) are enclosed in brackets
- Distinctive features include *Register Windows* and *Delayed Branches*

Register Windows

The SPARC provides 32 general-purpose integer registers, denoted as **`%r0`** through **`%r31`**.

These 32 registers are subdivided into 4 groups:

Globals:	**`%g0`** to **`%g7`**
In registers:	**`%i0`** to **`%i7`**
Locals:	**`%l0`** to **`%l7`**
Out registers:	**`%o0`** to **`%o7`**

There are also 32 floating-point registers, **`%f0`** to **`%f31`**.

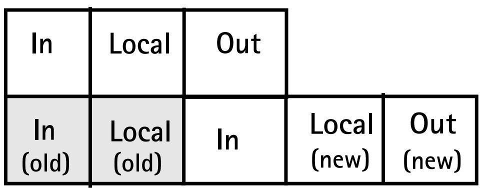
A SPARC processor has an implementation-dependent number of *register windows*, each consisting of 16 distinct registers.

The "in", "local" and "out" registers that are accessed in a procedure depend on the current register window. The "global" registers are independent of the register windows (as are the floating-point registers).

A register window may be pushed or popped using SPARC **`save`** and **`restore`** instructions.

After a register window push, the "out" registers become "in" registers and a fresh set of "local" and "out" registers is created:

Before **`save`**:



After **`save`**

Why the overlap between "in" and "out" registers? It's a convenient way to pass parameters—the caller puts parameter values in his "out" registers. After a call (and a **`save`**) these values are *automatically* available as "in" registers in the newly created register window.

SPARC procedure calls normally advance the register window. The "in" and "local" registers become hidden, and the "out" registers become the "in" registers of the called procedure, and new "local" and "out" registers become available.

A register window is advanced using the **`save`** instruction, and rolled back using the **`restore`** instruction. These instructions are separate from the procedure **`call`** and **`return`** instructions, and can sometimes be optimized away.

For example, a *leaf procedure*—one that contains no calls—can be compiled without use of **`save`** and **`restore`** if it doesn't need too many registers. The leaf procedure must then make do with the caller's registers, modifying only those the caller treats as volatile.

Register Conventions

Global Registers

`%g0` is unique: It *always* contains 0 and can *never* be changed.

`%g1` to **`%g7`** have global scope (they are unaffected by **`save`** and **`restore`** instructions)

`%g1` to **`%g4`** are volatile across calls; they may be used between calls.

`%g5` to **`%g7`** are reserved for special use by the SPARC ABI (application binary interface)

Local Registers

`%l0` to **`%l7`**

May be freely used; they are unaffected by deeper calls.

In Registers

These are also the caller's out registers; they are unaffected by deeper calls.

`%i0`

Contains incoming parameter 1.

Also used to return function value to caller.

`%i1` to **`%i5`**

Contain incoming parameters 2 to 6 (if needed); freely usable otherwise.

`%i6` (also denoted as **`%fp`**)

Contains frame pointer (stack pointer of caller); it must be preserved.

`%i7`

Contains return address -8 (offset due to delay slot); it must be preserved.

Out Registers

Become the in registers for procedures called from the current procedure.

`%o0`

Contains outgoing parameter 1.

It also contains the value returned by the called procedure.

It is volatile across calls; otherwise it is freely usable.

`%o1` to **`%o5`**

Contain outgoing parameters 2 to 6 as needed.

These are volatile across calls; otherwise they are freely usable.

`%o6` (also denoted as **`%sp`**)

Holds the stack pointer (and becomes frame pointer of called routines)

It is reserved; it must *always* be valid (since TRAPs may modify the stack at any time).

`%o7`

Is volatile across calls.

It is loaded with address of caller on a procedure call.

Special SPARC Instructions

```
save   %r1,%r2,%r3
save   %r1,const,%r3
```

This instruction pushes a register window *and* does an add instruction (**`%r3`** = **`%r1+%r2`**). Moreover, the operands (**`%r1`** and **`%r2`**) are from the *old* register window, while the result (**`%r3`**) is in the *new* window.

Why such an odd definition?

It's ideal to allocate a new register window *and* push a new frame.

In particular,

```
save   %sp,-frameSize,%sp
```

pushes a new register window. It also adds **`-frameSize`** (the stack grows downward) to the old stack pointer, initializing the new stack pointer. (The old stack pointer becomes the current frame pointer)

`restore %r1,%r2,%r3`

`restore %r1,const,%r3`

This instruction pops a register window *and* does an add instruction (**`%r3`** = **`%r1+%r2`**). Moreover, the operands (**`%r1`** and **`%r2`**) are from the *current* register window, while the result (**`%r3`**) is in the *old* window.

Again, why such an odd definition?

It's ideal to release a register window *and* place a result in the return register (**`%o0`**).

In particular,

```
restore   %r1,0,%o0
```

pops a register window. It also moves the contents of **`%r1`** to **`%o0`** (in the caller's register window).

`call label`

This instruction branches to **`label`** *and* puts the address of the call into register **`%o7`** (which will become **`%i7`** after a **`save`** is done).

`ret`

This instruction returns from a subprogram by branching to **`%i7+8`**. Why *8 bytes* after the address of the call? SPARC processors have *delayed branch* instructions, so the instruction immediately after a branch (or a **`call`**) is executed *before* the branch occurs! Thus two instructions after the call is the normal return point.

`mov const,%r1`

You can load a small constant (13 bits or less) into a register using a **`mov`**. (**`mov`** is actually implemented as an **`or`** of **`const`** with **`%g0`**).

But how do you load a 32 bit constant? One instruction (32 bits long) isn't enough. Instead you use:

```
sethi %hi(const),%r1
or %r1,%lo(const),%r1
```

That is, you extract the high order 22 bits of **`const`** (using **`%hi`**, an assembler operation). **`sethi`** fills in these 22 bits into **`%r1`**, clearing the lowest 10 bits. Then **`%lo`** extracts the 10 low order bits of const, which are or-ed into **`%r1`**.

Loading a 64 bit constant (in SPARC V9, which is a 64 bit processor) is far nastier:

```
sethi   %uhi(const),%r_tmp
or      %r_tmp,%ulo(const),%r_tmp
sllx    %r_tmp,32,%r_tmp
sethi   %hi(const),%r
or      %r,%lo(const),%r
or      %r_tmp,%r,%r
```

Delayed Branches

In the SPARC, transfers of control (branches, calls and returns) are *delayed*. This means the instruction *after* the branch (or call or return) is executed *before* the transfer of control.

For example, in SPARC code you often see

```
ret
restore
```

The register window restore occurs first, then a return to the caller occurs.

Another example is

```
call subr
mov 3,%o0
```

The load of `subr`'s parameter is placed after the call to `subr`. But the `mov` is done before `subr` is actually called.

Why are Delayed Branches Part of the SPARC Architecture?

Because of pipelining, several instructions are partially completed before a branch instruction can take effect. Rather than lose the computations already done, one (or more!) partially completed instructions can be allowed to complete before a branch takes effect.

How does a Compiler Exploit Delayed Branches?

A peephole optimizer or code scheduler looks for an instruction logically before the branch that can be placed in the branch's *delay slot*. The instruction should not affect a conditional branch's branch decision.

```
mov  3,%o0      call subr
call subr       mov  3,%o0
nop
```

(before) (after)

Another possibility is to "hoist" the target instruction of a branch into the branch's delay slot.

```
   call subr        call subr+4
   nop              mov 100,%l1
   ...              ...
subr:            subr:
   mov 100,%l1      mov 100,%l1
```

(before) (after)

Hoisting branch targets doesn't work for conditional branches—we don't want to move an instruction that is executed *sometimes* (when the branch is taken) to a position where it is *always* executed (the delay slot).

Annulled Branches

An *annulled branch* (denoted by a "**,a**" suffix) executes the instruction in the delay slot *if* the branch is taken, but *ignores* the instruction in the delay slot if the branch isn't taken.

With an annulled branch, a target of a conditional branch can be hoisted into the branch's delay slot.

```
   bz   else         bz,a  else+4
   nop               mov 100,%l1
 ! then code       ! then code
   ...               ...
else:             else:
   mov 100,%l1       mov 100,%l1
```

(before) (after)

SPARC Frame Layout (on Run-Time Stack)

The Stack Grows Downward

Caller's Frame	← %fp (old %sp)
Local Variables	
alloca() space	
Memory Temps & Saved FP Registers	
Parms past 6	
Parms 1 to 6 (memory home)	
Address of return value	
Register Window Overflow (16 words)	← %sp

Minimum frame Size (in gcc) is 112 bytes! (16+1+6+4 words, double aligned)

Examples of SPARC Code

```
int incr(int i){
   return i+1; }
```

Unoptimized:

```
incr:
    save    %sp, -112, %sp
    st      %i0, [%fp+68]
    ld      [%fp+68], %o1
    add     %o1, 1, %o0
    mov     %o0, %i0
    b       .LL2
         nop
.LL2:
    ret
    restore
```



```
int   main(){
   int a;
   return incr(a);}
```

Unoptimized:

```
main:
   save    %sp, -120, %sp
   ld      [%fp-20], %o0
   call    incr, 0
   nop
   mov     %o0, %i0
   b       .LL3
   nop
.LL3:
   ret
   restore
```

```
int incr(int i){
   return i+1; }
```

Optimized:

```
incr:
    retl
    add   %o0, 1, %o0
```

```
int   main(){
   int a;
   return incr(a);}
```

Optimized:

```
main:
   save    %sp, -112, %sp
! Where is a ????
   call    incr, 0
   nop
   ret
   restore %g0, %o0, %o0
```


With More Extensive Optimization (including inlining) we get:

```
incr:
   retl
   add     %o0, 1, %o0

main:
   retl
   add     %o0, 1, %o0
```