5. Cache Improvements

We want to access data & instructions from the L1 cache whenever possible; misses into the L2 cache (or memory) are *expensive*!

We will layout data and program code with consideration of cache sizes and access properties.

6. Local & Global Optimizations

Identify unneeded or redundant code.

Decide where to place code.

Worry about debugging issues (how reliable are current values and source line numbers after optimization?)

7. Program representations

- Control Flow Graphs
- Program Dependency Graphs
- Static Single Assignment Form (SSA)

  Each program variable is assigned to in only *one* place.

  After an assignment $x_i = y_j$, the relation $x_i = y_j$ *always* holds.

  Example:

```
if (a)                 if (a)
     x = 1                  x1 =1
else x = 2;            else x2 =2;
print(x)               x3 = φ(x1,x2)
                       print(x3)
```

8. Data Flow Analysis

Determine invariant properties of subprograms; analysis can be extended to entire programs.

Model abstract execution.

Prove correctness and efficiency properties of analysis algorithms.

Review of Compiler Optimizations

1. Redundant Expression Elimination (Common Subexpression Elimination)

   Use an address or value that has been previously computed. Consider control and data dependencies.

2. Partially Redundant Expression (PRE) Elimination

   A variant of Redundant Expression Elimination. If a value or address is redundant along *some* execution paths, add computations to other paths to create a fully redundant expression (which is then removed).

   Example:

```
if (i > j)
   a[i] = a[j];
a[i] = a[i] * 2;
```

3. Constant Propagation

If a variable is known to contain a particular constant value at a particular point in the program, replace references to the variable at that point with that constant value.

4. Copy Propagation

After the assignment of one variable to another, a reference to one variable may be replaced with the value of the other variable (until one or the other of the variables is reassigned).

(This may also “set up” dead code elimination. Why?)

5. Constant Folding

An expression involving constant (literal) values may be evaluated and simplified to a constant result value. Particularly useful when constant propagation is performed.

6. Dead Code Elimination

Expressions or statements whose values or effects are unused may be eliminated.

7. Loop Invariant Code Motion

An expression that is *invariant* in a loop may be moved to the loop's header, evaluated once, and reused within the loop.

Safety and *profitability* issues may be involved.

8. Scalarization (Scalar Replacement)

A field of a structure or an element of an array that is repeatedly read or written may be copied to a local variable, accessed using the local, and later (if necessary) copied back.

This optimization allows the local variable (and in effect the field or array component) to be allocated to a register.

9. Local Register Allocation

Within a *basic block* (a straight line sequence of code) track register contents and reuse variables and constants from registers.

10. Global Register Allocation

Within a subprogram, frequently accessed variables and constants are allocated to registers. Usually there are *many more* register candidates than available registers.

11. Interprocedural Register Allocation

Variables and constants accessed by more than one subprogram are allocated to registers. This can *greatly* reduce call/return overhead.

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.

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.

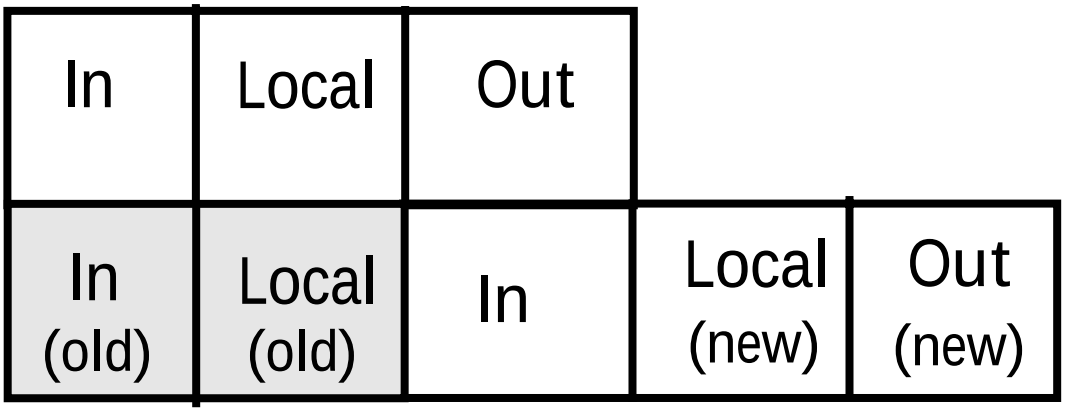
The "in", "local" and "out" registers that are accessed in a procedure depend on the current register window. The "global"

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.