Topics in computer architecture

SPARC compilers

P.J. Drongowski
SandSoftwareSound.net

SPARC compilers

- "Optimizing compilers for SPARC," Steven S. Muchnick, SunTechnology, Summer 1988, pg. 64 - 77
- "A global optimizer for Sun FORTRAN, C, and Pascal," V. Ghodssi, et al., Summer 1986 USENIX Conference, June 1986, pg. 318 - 334
- Claim
  - Few choices for proper instruction, addressing mode
  - Can easily generate locally optimal code for expressions
  - Lets developers concentrate on
    - ◊ Runtime environment
    - ◊ Global code optimization

Registers

- Register allocation is key resource allocation issue
- Load/store architecture makes good allocation critical
- Three sets of registers are visible at any time
  - Global integer registers
    - Global integer register `g0` is special
    - Reads as value zero and ignores write operations
    - Subroutine caller must save global live variables
    - Global var's can be accessed by offset from base
  - Global floating-point registers
    - Function as FP registers (of course!)
    - Also used as caller-saved variables and temporaries
  - Windowed integer registers
    - `i0` to `i7` *ins*
    - `l0` to `l7` *locals*
    - `o0` to `o7` *outs*
    - `sp` (same as `o6`) stack pointer
    - `fp` (same as `i6`) frame pointer
    - `o7` return address

Addressing

- Computational instructions
  - Register Δ register ⇒ register
  - Register Δ immediate ⇒ register
- Load and store instructions
  - Register Δ register ⇒ effective address
  - Register Δ immediate ⇒ effective address
  - All immediate values are signed 13-bit integers
  - Use `g0` to form absolute address
- `sethi` instruction
  - Used to build 32-bit constants and addresses
  - Loads a 22-bit constant into high end of register
  - The low order 10 bits are set to zeroes
  - Example

```
sethi  %hi(loc), %i1
ld     [%i1+%lo(loc)], %i2
```

  - Loads word at address loc into `i2`
  - Most constants are short so `sethi` is rarily

Stack model

- Stack frame is addressed relative to `fp`
  - Parameters beyond the sixth (if any)
  - Parameters that must be memory addressable
  - Address of stack space for a C `struct` return value
  - Space for (window) overflow *in* and *local* registers
  - Automatic variables that must be memory addressable
  - Compiler generated temporaries
  - Saved floating-point registers
- Addressed relative to stack pointer `sp`
  - Temporaries
  - Outgoing procedure parameters

Procedure call and return

- Parameter passing
  - Move parameters to caller's *out* registers
  - Extra parameters are pushed on stack via sp
  - Six registers are available for parameter passing
- Procedure invocation instructions
  - `call` (one cycle plus delay slot)
    - 30-bit PC-relative word displacement
    - Stores return address in `o7`
  - `jmpl` (one cycle plus delay slot)
    - Jump and link instruction
    - Target is sum of 2 registers or register & immediate
    - Store return address in specified register
- Procedure prologue
  - `save` instruction changes register window
  - Caller's *outs* become procedure's *in* registers
  - New set of *locals* and *outs* are provided
  - Allocate new stack frame by setting new `sp` from old `sp`
- Execution
  - Unused *ins* and *locals* can be used
- Postlude (exit sequence)
  - Return value is written into one of the *in* registers
  - Value will be available as a caller *out* register
  - `restore` instruction deallocates register window
  - Resets the caller's stack pointer
  - `jmp` to return address
- Aggregate value return
  - C `struct` cannot be returned in a single register
  - Caller must allocate return area on stack
  - Address of memory area is passed as a parameter
  - Procedure checks if caller is expecting aggregate
  - Procedure looks for `unimp` instruction and block size
  - If found and size matches, then return normally
  - Else, execute unimp instruction and cause a trap

Multiply and divide

- No multiply, divide or remainder instructions in SPARC
- Must be synthesized from elementary operations
- Multiply step instruction `mulscc`
- Multiply by constant handled as special case
  - Uses sequence of shifts and adds
  - Will use subtraction if overflow detection is not needed
  - Example: Multiplication by 30

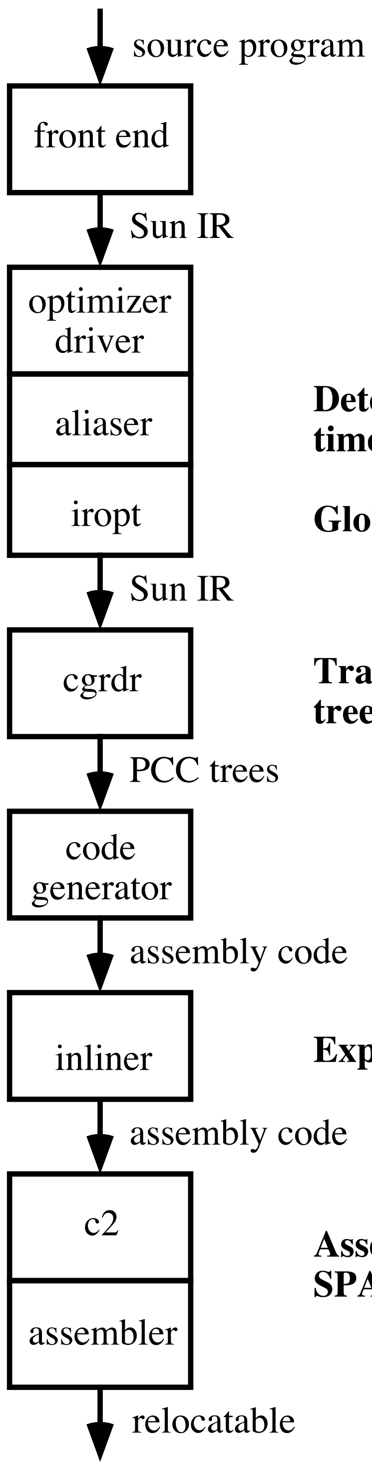
```
sll  %o2, 1, %o2    ! 2 * X -> X
sll  %o2, 4, %o3    ! 16 * X -> Y
sub  %o3, %o2, %o2  ! y - X -> X
```

- Runtime leaf routines
  - Used for multiplication of two variables and all divisions
  - Statistics were gathered on multiply and divide
  - Biased to terminate quickly for common cases
  - Example multiplication statistics
    - 90% have at least one nonnegative operand
    - 90% have one operand of 7 bits or less
    - 99% have one operand at most 9 bits long
    - Thus, choose shorter operand in multiply
    - Average multiply takes less than 6 cycles
    - Average `var x var` takes 24 cycles

Tagged data support

- Special add and subtract instructions
- Treat low order two bits as a type tag
- Can (optionally) cause a trap
- `taddcc` (`taddcctv`)
  - Add two operands together and store result
  - Set overflow if either tag is non-zero or add overflows
- Example: Common LISP *fixnum* arithmetic
  - Sum and detection of *fixnum* performed in one step
  - Cuts add time from six to three

SPARC compiler structure



Determines which variables may at some time point to same location

Global optimizer

Translate Sun IR (intermediate code) to the PCC trees used by the code generator

Expand procedures inline

Assembly level optimizer integrated with SPARC assembler

Global optimization

- Loop-invariant code motion
- Induction-variable strength reduction
- Common subexpression elimination
- Copy propagation
- Register allocation (modified graph coloring)
- Dead code elimination
- Loop unrolling
- Tail-recursion elimination
- No interprocedural optimization

Peephole optimization

- Eliminates unnecessary jumps
- Eliminates redundant loads and stores
- Deletes unreachable code
- Does loop inversion
- Utilizes machine idioms
- Performs register coalescing
- Handles instruction scheduling
- Does leaf-routine optimization
- Performs cross jumping
- Handles constant propagation

Tail recursion elimination

- Self-recursive procedure
- Only action after it returns to self is to itself return
- Recursion can be transformed to iteration
- Savings due to optimization
    - Reduces window overflows and underflows
    - Saves stack allocation, manipulation and deallocation
- Detect call by reference and suppress elimination
- Number of parameters to C routine can vary (ouch!)

Loop unrolling

- Replace loop body with several copies of the body
- Adjust control code as necessary
- Advantages
  - Especially good for loops with constant bounds
  - Reduces overhead of looping
  - Increases effectiveness of instruction scheduling
- Conditions for unrolling
  - Contain only a single basic block (straight-line code)
  - Generate at most 40 triples of Sun IR code
  - Contain floating-point operations
  - Have simple loop control
- Loop body is unrolled once; more are switch controlled

Instruction scheduling

- Special scheduling cases
  - Delay branch filling (or conditional annulment)
  - Overlap load and instruction without dependency
  - Parallel execution of integer and FP instructions
  - Parallel execution of floating add and multiply
- All cases may interact with one another!
- Effectiveness (Stanford benchmark)
  - Branch filling - utilizes all but 5% of slots
  - Overlap load - 74% scheduled without dependency

In-line expansion

- Replace calls with assembly language code sequence
- Advantages
  - Save execution time
  - Allow further improvement by peephole optimizer

Leaf-routine optimization

- A routine that does not call any other routine
- Must use just a few registers and no local stack frame
- Eliminate `save` and `restore` (maybe 15%?)
- Reduces the number of window overflows and underflows