CS302 Spr'99 Lecture Notes
Lecture 10

Machine Code Generation

- Instruction Selection
- Register Allocation and Assignment
- Optimization

Issues:

- Complexity of Target Machine
- Level of Translation: expression, statement, basic block, routine, program?
- Management of Scarce Resources

Approaches to Instruction Selection

For **RISC** targets, translate one IR instruction to one or more target instructions.

For **CISC** targets, translate several IR instructions to one target instruction.

```
Example Source: a := b  (assuming a,b in frame)

3-addr IR: t1 = fp-12
           t2 = *t1
           t3 = fp+8
           *t3 = t2

Typical Tree IR:          mov
                          / \
                         /   \
                       mem   mem
                        |     |
                        +     +
                       / \   / \
                      fp  8 fp -12

extreme RISC: add %fp,-12,%r3
              ld [%r3], %r7
              add %fp,8,%r4
              st %r7,[%r4]

moderate RISC:  ld [%fp-12], %r7
                st %r7, [%fp + 8]

CISC:   move [%fp-12],[%fp+8]
```

Simplistic SPARC Instruction Selection for PCAT

• Generate instructions directly from AST, using 3-address style.

• Include explicit code for array and record calculations.

• (Alternatively, could do one-to-one translation of IR.)

• Take advantage of SPARC's M[reg+const] addressing mode to generate good code for frame references.

```
DO THIS: ld [%fp-12], %r7

NOT THIS: add %fp,-12,%r3
          ld [%r3], %r7
```

• Use (small) constants directly where possible.

```
DO THIS: add %r1,42,%r1

NOT NOT: mov 42,%r2
         add %r1,%r2,%r1
```

• Fill delay slots with `nop`'s, unless producing a "canned" sequence that can use them.

Register Allocation and Assignment

Task: Manage scarce resources (registers) in environment with imperfect information (static program text) about dynamic program behavior.

General aim is to keep frequently-used values in registers as much as possible, to lower memory traffic. Can have a **large** effect on program performance.

Variety of approaches are possible, differing in sophistication and in scope of analysis used.

Allocator may be unable to keep every "live" variable in registers; must then "spill" variables to memory. Spilling adds new instructions, which often affects the allocation analysis, requiring a new iteration.

If spilling is necessary, what should we spill? Some heuristics:

• Don't spill variables used in inner loops.

• Spill variables not used again for "longest" time.

• Spill variables which haven't been updated since last read from memory.

Simplistic Register Management for PCAT

• Assume variables "normally" live in memory.

• Fetch values into registers just before (each) use in an expression.

• Register use never spans statements.

• Use auxiliary table to track register use.

```
static int reg_used[32];
int getreg();   /* returns first free reg */
void freereg(int r);  /* frees specified reg */
```

• Certain SPARC registers are reserved.

• Remember that same register can be used as source and target:

```
/* assume source operands in t1,t2 */
freereg(t1);
freereg(t2);
int t3 = getreg();
printf("add \%r%d,\%r%d,\%r%d", t1, t2, t3);
```

might produce `add %r3,%r5,%r3`

• Ignore possibility of spills.

Register Allocation for Expressions

Choice of evaluation order can affect number of registers needed.

```
Example: (a+b) - ((c+d) - (e+f))      -
                                     / \
                                    /   \
                                   +     -
                                  / \   / \
                                 a   b /   \
                                      +     +
                                     / \   / \
                                    c   d e   f
```

If we compute left child first, need 4 regs, but doing right child first needs only 3.

```
load a,r1         load c,r1
load b,r2         load d,r2
add r1,r2,r1      add r1,r2,r1
load c,r2         load e,r2
load d,r3         load f,r3
add r2,r3,r2      add r2,r3,r2
load e,r3         sub r1,r2,r1
load f,r4         load a,r2
add r3,r4,r3      load b,r3
sub r2,r3,r2      add r2,r3,r2
sub r1,r2,r1      sub r2,r1,r1
```

Minimizing Registers Needed to Evaluate Expression Trees

Key idea (Sethi & Ullman): At each node, **first** evaluate subtree requiring largest number of registers to evaluate. Can then save result of this evaluation in a register while doing other subtree.

1. Label each node with minimum number of registers needed to evaluate subtree.

```
risc_label(t)
  if isLeaf(t) then
    t->label = 1  (depends on instruction set)
  else
    label(t->left)
    label(t->right)
    if (t->left->label == t->right->label)
      t->label = t->left->label + 1
    else
      t->label = max(t->left->label,
                     t->right->label)
```

2. Use labels to guide order of code emission; emit code for higher-numbered subtree **first**.

3. If we run out of registers (when does this happen?), spill to temporary memory locations.

Sethi-Ullman Numbering Example

```
          -^3
         / \
        /   \
      +^2    -^3
     / \    / \
   a^1  b^1/   \
        +^2    +^2
       / \    / \
     c^1  d^1e^1  f^1
```


Other Issues in Tree Evaluation Order

Some machines (e.g., X86) allow one operand (e.g., left one) to be a complex expression, while the other must be a register, which also holds the result ("accumulator" style):

```
add 37,r0     ;  r0 <- r0 + 37
add b, r1     ;  r1 <- r1 + b
sub r0,r1     ;  r1 <- r1 - r0
```

- These machines have different Sethi-Ullman numbering, e.g., right **leaves** may require no registers at all.
- **If** we must spill registers, it is better to evaluate left child of non-commutative operators (like -,/) **last**, because they will require their left operand in a register anyhow.

Can use associativity to make trees "less bushy, " e.g.

```
        +^3                      +^2
       / \                      / \
      /   \        becomes    +^2  d^1
    +^2    +^2               / \
   / \    / \              +^2  c^1
 a^1  b^1c^1  d^1          / \
                         a^1  b^1
```


Basic Blocks

- Extend analysis of register use to program units larger than expressions but still completely analyzable at compile time.
- **Basic Block** = sequence of instructions with single entry & exit.
- If first instruction of BB is executed, so is remainder of block (in order).
- To calculate basic blocks:

(1) Determine BB **leaders** (→) :

(a) First statement in routine
(b) Target of any jump (conditional or unconditional).
(c) Statement following any jump.

(What about subroutine calls?)

(2) Basic block extends from leader to (but not including) next leader (or end of routine).

Basic Block Example

```
prod := 0;
i := 1;
while i <= 20 do
  prod := prod + a[i] * b[i];
  i := i + 1
end
```

```
→  1. prod := 0
   2. i := 1
→  3. if i > 20 goto 14
→  4. t1 := i * 4
   5. t2 := addr a
   6. t3 := *(t2+t1)
   7. t4 := i * 4
   8. t5 := addr b
   9. t6 := *(t5+t4)
  10. t7 := t3 * t6
  11. prod := prod + t7
  12. i := i + 1
  13. goto 3
→ 14. ---
```


Register Assignment Within Basic Blocks

Idea: Let program variables and temporaries stay in registers as long as possible.

```
Example:        Naive code:     Better code:

a := b + c      ld b,r0         ld b,r0
d := a + b      ld c,r1         ld c,r1
a := b + 1      add r0,r1,r0    add r0,r1,r2
                st r0,a         add r2,r0,r1
                ld a,r0         st r1,d
                ld b,r1         add r0,1,r0
                add r0,r1,r1    st r0,a
                st r1,d
                ld b,r0
                add r0,1,r0
                st r0,a
```

• Simplest to operate on one basic block at a time; can extend to multiple blocks with some effort.

• Assume an infinite supply of registers; later "spill" some to memory if required.

• Registers behave like a **cache** for memory locations.

• Nasty problems for source-level debuggers and dump utilities – where **is** that variable?!?

Liveness

To determine how long to keep a given variable in a register, need to know the range of instructions for which the variable is **live**.

A variable is **live** immediately following an instruction if its current value will be needed in the future.

It's easy to calculate live ranges within a basic block, just by working backwards through the block.

Can assume that all user variables are live at the end of a basic block, i.e., that their values may be used in a subsequent block. (If doing BB-level register allocation, must save the values back in memory at the end of the block anyhow.)

Treat temporaries like user variables, except that they are assumed dead at end of BB.

Can improve accuracy by calculating liveness over entire routines, including control flow statements, not just BBs.

To do this requires **iterative flow analysis** and the result is only **conservative approximation** to true liveness.

BB Code Generation using Liveness

Can combine code generation with "greedy" register allocation: bring each variable into a register when first needed, and leave it there as long as it's needed (if possible).

Maintain **register descriptors** saying which variable is in each register, and **address descriptors** saying where (in memory and/or a register) each variable is.

For each IR instruction `x := y op z` (other instructions similar):

1. If `y` isn't in a register, load it into a free one, updating descriptors.

2. Similarly for `z`.

3. If `y` and/or `z` are no longer live following this instruction, mark their registers as free.

4. Choose a free register for `x`, updating descriptors.

5. Generate instruction `op` r_y, r_z, r_x.

For the special case `x := y`, load `y` into a register, if necessary, and then mark that register as holding `x` too.

Must now be careful not to free a register unless `none` of its associated variables is live.

Example

```
Source code: d := (a-b) + (a-c) + (a-c)

                    Live after inst:
                  a  b  c
IR: t := a - b    a     c  t
    u := a - c             t  u
    v := t + u                u  v
    d := v + u                      d

                                Descriptors after
                                  Instructions
IR Statement        Code        Regs      Addrs

                                 -        a,b,c:mem

t := a - b      ld a,r0         r0:a      a:r0,mem
                ld b,r1         r1:t      b,c:mem
                sub r0,r1,r1              t:r1

u := a - c      ld c,r2         r0:u      a,b,c:mem
                sub r0,r2,r0    r1:t      u:r0
                                          t:r1

v := t + u      add r1,r0,r1    r0:u      a,b,c:mem
                                r1:v      u:r0
                                          v:r1

d := v + u      add r0,r1,r0    r0:d      a,b,c:mem
                st r0,d         r1:v      d:r0,mem
```


Register Interference Graphs

Mixing instruction selection and register allocation gets confusing; need a more systematic way to look at the problem.

• Initially generate code assuming an infinite number of "logical" registers; calculate live ranges

Previous Example:

```
                                  Live after instr.
ld a,t0           ; a:t0   t0
ld b,t1           ; b:t1   t0 t1
sub t0,t1,t2      ; t:t2   t0    t2
ld c,t3           ; c:t3   t0    t2 t3
sub t0,t3,t4      ; u:t4         t2    t4
add t2,t4,t5      ; v:t5               t4 t5
add t5,t4,t6      ; d:t6                     t6
st t6,d
```

• Build a **register interference graph**, which has

- a node for each logical register.

- an edge between two nodes if the corresponding registers are simultaneously live.

Coloring Interference Graphs

Interference Graph Example:

```
   ----------------
  /                \
 t0 ----- t1       t2
 |  ______________//
 | /        ______/
 |/        /
 t3       t4------ t5       t6
```

A **coloring** of a graph is an assignment of colors to nodes such that no two connected nodes have the same color. (Like coloring a map, where nodes=countries and edges connect countries with common border.)

Suppose we have k physical registers available. Then aim is to color interference graph with k or fewer colors. This implies we can allocate logical registers to physical registers without spilling.

In general case, determining whether a graph can be k-colored is hard (N.P. Complete, and hence probably exponential).

But a simple heuristic will **usually** find a k-coloring if there is one.

Graph Coloring Heuristic

1. Choose a node with fewer than k neighbors.

2. Remove that node. Note that if we can color the resulting graph with k colors, we can also color the original graph, by giving the deleted node a color different from all its neighbors.

3. Repeat until **either**

• there are no nodes with fewer than k neighbors, in which case we must spill; **or**

• the graph is gone, in which case we can color the original graph by adding the deleted nodes back in one at a time and coloring them.

Example:

```
   ----------------
  /                \
 t0 ----- t1       t2
 |  ______________//
 | /        ______/
 |/        /
 t3       t4------ t5       t6
```

Finds a 3-coloring. There cannot be a 2-coloring (why not?).

Each "color" corresponds to a physical register, so 3 registers will do for this example.