

CS322 W'05 Lecture Notes  
Lecture 10

**Approaches to Instruction Selection**

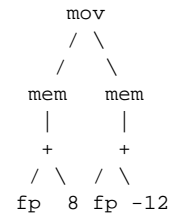
For **RISC** targets, translate one IR instruction to one or more target instructions.

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:



```
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]
```

**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

**Simplistic SPARC Instruction Selection for PCAT IR**

Generate instructions from 3-address-style IR.

- Already includes explicit code for array and record calculations.
- Still needs to resolve variable addresses.

(Alternatively, could generate SPARC code directly from AST.)

Approach:

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

```
FOR THIS IR: ld [a], %t20
```

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

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

- But don't try to improve IR code that is already expanded:

```
THIS PCAT: x.a (record dereference)
```

```
GAVE THIS IR: ld [x], %t10
              adda %t10,3,%t11
              ld [%t11],%t12
```

```
SETTLE FOR THIS: ld [%fp+8], %r3
                 add %r3,12,%r3
                 ld [%r3],%r3
```

- 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.

## Simplistic Register Management for PCAT

- Assume variables "normally" live in memory.
- Use existing (often redundant) fetches and stores present in IR.
- So: only need to allocate registers to IR temporaries (`%t`).
- Certain SPARC registers are reserved (see `Sparc.regUsable`).
- Ignore possibility of spills.
- Liveness information for all temporaries is already calculated for you (see Liveness).
- Use simple **linear scan** register allocator based on liveness intervals.

## 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.

## Liveness

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

A variable or temporary is **live** immediately following an instruction if its current value will be needed in the future (i.e., it will be used again, and it won't be changed before that use).

Example:

```

                                ! live after instruction:
mov 3, %t2                      ! %t2
mov %t2, %t3                    ! %t2 %t3
add %t3, 4, %t4                 | %t2      %t4
add %t2, %t4, %t4               |          %t4
st %t4, [a]                     | (nothing)
```

It's easy to calculate liveness for a consecutive series of instructions without branches, just by working backwards.

**Liveness (continued)**

But if a value can stay in a register over a jump, things get harder. Example:

```

                                ! live after instruction:
1   mov 0, %t1                  ! %t1   %t3
2 L1: add %t1, 1, %t2           !      %t2 %t3
3   add %t3, %t2, %t3          !      %t2 %t3
4   mul %t2, 2, %t1            ! %t1   %t3
5   cmp %t1, 1000              ! %t1   %t3
6   bl L1                       ! %t1   %t3
7   return %t3                  ! (nothing)

```

To calculate liveness in this case requires **iterative flow analysis** and the result is only **conservative approximation** to true liveness (more later).

The **live range** of a variable is the set of instructions which leave it live. E.g. in 2nd example, live range of %t1 is {1, 4, 5, 6}; live range of %t3 is {1, ..., 6}.

Basic idea: If two variables have disjoint live ranges, they can occupy the same physical register.

So in both examples, 2 physical registers suffice to allocate all temporaries without spilling.

**Linear Scan Allocation Algorithm Details**

1. Compute  $startpoint[i]$  and  $endpoint[i]$  of live interval  $i$  for each temporary. (For HW4, this has been done for you.). Store the intervals in a list in order of increasing start point.
2. Initialize set  $active := \emptyset$  and pool of free registers = all usable registers.
3. For each live interval  $i$  in order of increasing start point:
  - (a) For each interval  $j$  in  $active$ , in order of increasing end point
    - If  $endpoint[j] \geq startpoint[i]$  break to step 3b.
    - Remove  $j$  from  $active$ .
    - Add  $register[j]$  to pool of free registers.
  - (b) Set  $register[i] :=$  next register from pool of free registers, and remove it from pool. (If pool is already empty, need to spill.)
  - (c) Add  $i$  to  $active$ , sorted by increasing end point.

**Linear Scan Allocation**

Using live ranges turns out to be computationally expensive (more later).

A simple alternative is to approximate each live range by a **live interval**. This is the consecutive interval of instructions between the first and last use of each temporary. Example:

```

                                ! live after instruction:
1   mov 0, %t1                  ! %t1   %t3
2 L1: add %t1, 1, %t2           !      %t2 %t3
3   add %t3, %t2, %t3          !      %t2 %t3
4   mul %t2, 2, %t1            ! %t1   %t3
5   cmp %t1, 1000              ! %t1   %t3
6   bl L1                       ! %t1   %t3
7   return %t3                  ! (nothing)

```

Live ranges: %t1: 1,4,5,6 %t2:2,3 %t3:1,2,3,4,5,6

Live intervals: %t1: [1,6] %t2: [2,3] %t3: [1,6]

(Revised) Basic idea: if two temporaries have non-overlapping live intervals, they can occupy the same physical register.