# Code generation: continued

- previously we looked at code generation using a simple tree-walking routine, with an emphasis on binary operations
- still need to address:
- handling of function calls,
- handling of mixed data types in operations,
- special issues with the assignment operation(s),
- how to handle conditional expressions and branching (with huge impact on loops, if/else, etc)

# Function calls

- discussed earlier the need to emit code in the caller/callee segments to handle the transfer of control between them, setting up and cleaning up activation record, etc
- at assembly language level, this should fit seamlessly in the operation sequence(with return value winding up in a register somewhere)
- Possibility of side effects limits ability to perform optimizations based on order of ops: (a + b – f(a) + a + b), suppose f has side effect on a then we can't reuse the (a+b) ... f could even have side effect on b if its global

# Mixed-type operations

- given (a + b), if types of a and b are not identical then need to pick which kind of “+” we're performing, and insert code to implicitly convert one of the operand values
- tree walk or output routines might then be performing type checking, and emitting extra code for implicit conversion

# Assignment operations

- Generally accepted idea for x = expr is to evaluate RHS and store value in variable on LHS
- this allows right-to-left chaining, e.g. x = y = z
- means assign has very low precedence, so performed last
- again need to typecheck RHS vs LHS and insert appropriate conversion code if needed
- More complex assignments, e.g. x += y, may involve multiple operations at assembly language level

# Boolean operations

- not universal, but a common precedence scheme is
  - OR (lowest)
  - AND
  - < <= = != > >=
  - + -
  - * /
  - negation
  - ( expr )
- allows expressions like if ( a < b AND b < c )

# Short circuiting

- Need to be aware when source and target languages have different expectations w.r.t. short circuiting expressions
- Short circuit based on idea that
  - “true OR x” is true,
  - “false AND x” is false:
- don't need to evaluate x in either case
- (note some similar ideas hold in other areas, e.g. 0*x is 0)

# Relationship to hardware

- Translation of HLL statements to assembly-level statements often heavily affected by nature of test-and-branch operations at the hardware level
- Four common schemes we'll look at:
  - condition codes (cc)
  - condition codes + conditional move
  - boolean compare
  - predicated execution

# Condition codes approach

- Compare operation compares two ops, say R1 R2, sets variety of flags in a condition register to show if R1<R2, R1<=R2, R1=R2,R1!=R2, etc
- Suite of branching operations take a condition register and two labels, jump to one label if condition is true, else other
- e.g. if r1 < r2 jump to label1, else jump to label2:
  - compare r1,r2,cc1
  - branchLT cc1, label1, label2

# CC + conditional move

- Adds one more set of operations, each takes a condition register, two data registers, and a destination register
- if condition is true then stores first data value in destination, otherwise stores second
- e.g. if r1 < r2 then r5 = r1, else r5 = r2
  - compare r1,r2,cc1
  - moveLT cc1, r1, r2, r5

# Boolean compare

- drops condition code registers entirely, uses a suite of compare operations that each check a specific relationship and set a true/false value in destination register
- branch instruction takes register and two labels, jumps to one label if register contains true, otherwise to other label
- e.g. If r1 < r2 then r3 = true, else r3 = false
  - compareLT r1, r2, r3
  - branch r3, label1, label2

# Predicated execution

- requires support at hardware level
- allows instructions that take a register as first argument and another instruction as the second
- if first argument is true then executes second argument
- e.g. If r1 is true then r4 = r2 + r3
  - (r1)? add r2,r3,r4

# Ex: if (a <= b) then x = y + z else x = i - j

```
// cc version
   compare ra, rb, cc1
   branchLE cc1,L1,L2
L1:add ry,rz,rx
   jump L3
L2:sub ri,rj,rx
L3:
```

```
// boolean compare
   compareLE ra, rb, r1
   branch r1,L1,L2
L1:add ry,rz,rx
   jump L3
L2:sub ri,rj,rx
L3:
```

```
// with conditional move,
// here computes both answers
// and picks one
   compare ra, rb, cc1
   add ry,rz,r1
   sub ri,rj,r2
   moveLE r1,r2,rx
```

```
// predicated execution
// adds true/false test for
// each operation
   compareLE ra, rb, r1
   not r1, r2
   (r1)? add ry,rz,rx
   (r2)? sub ri,rj,rx
```