Code generation and local optimization
Generating assembly

• How do we convert from three-address code to assembly?
  • Seems easy! But easy solutions may not be the best option

• What we will cover:
  • Peephole optimizations
  • Address mode selection
  • “Local” common subexpression elimination
  • “Local” register allocation
  • More complex code generation
Naïve approach

- “Macro-expansion”
- Treat each 3AC instruction separately, generate code in isolation

ADD A, B, C

LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C

MUL A, 4, B

LD A, R1
MOV 4, R2
MUL R1, R2, R3
ST R3, B
Why is this bad? (II)

MUL A, 4, B →
LD A, R1
MOV 4, R2
MUL R1, R2, R3
ST R3, B
Why is this bad? (II)

MUL A, 4, B →

LD A, R1
MOV 4, R2
MUL R1, R2, R3
ST R3, B

Too many instructions
Should use a different instruction type
Why is this bad? (II)

MUL A, 4, B  

LD A, R1
MOV 4, R2
MUL R1, R2, R3
ST R3, B

MUL A, 4, B  

LD A, R1
MULI R1, 4, R3
ST R3, B

Too many instructions
Should use a different instruction type
Why is this bad? (II)

ADD A, B, C
ADD C, A, E

LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C

LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C
LD C, R4
LD A, R5
ADD R4, R5, R6
ST R6, E
Why is this bad? (II)

ADD A, B, C
LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C

ADD A, B, C
ADD C, A, E
LD A, R5
ADD R4, R5, R6
ST R6, E

Redundant load of C
Redundant load of A
Uses a lot of registers
Why is this bad? (II)

ADD A, B, C
ADD C, A, E

LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C
LD C, R4
LD A, R5
ADD R4, R5, R6
ST R6, E

Redundant load of C
Redundant load of A
Uses a lot of registers
Why is this bad? (III)
Why is this bad? (III)

ADD A, B, C
ADD A, B, D

LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C

Wasting instructions recomputing A + B

LD A, R4
LD B, R5
ADD R4, R5, R6
ST R6, D
How do we address this?

• Several techniques to improve performance of generated code
  • Peephole optimizations to remove redundant instructions
  • Address mode selection to choose better instructions
  • Common subexpression elimination to remove redundant computation
  • Register allocation to reduce number of registers used
Address mode selection

- Even a simple instruction may have a large set of possible address modes and combinations

+ A B C

- Can be indirect, register, memory address, indexed, etc.
- Can be literal, register, memory address, indexed, etc.
- Can be literal, register, memory address, indexed, etc.

- Dozens of potential combinations!
More choices for address mode

- Auto increment/decrement (especially common in embedded processors as in DSPs)
  - e.g., load from this address and increment it
- Why is this useful?
- Three-address instructions
- Specialized registers (condition registers, floating point registers, etc.)
- “Free” addition in indexed mode
  MOV (R1)offset R2
- Why is this useful?
Peephole optimizations

• Simple optimizations that can be performed by pattern matching

• Intuitively, look through a “peephole” at a small segment of code and replace it with something better

• Example: if code generator sees \texttt{ST \ R \ X; \ LD \ X \ R}, eliminate load

• Can recognize sequences of instructions that can be performed by single instructions

\texttt{LDI R1 R2; \ ADD R1 4 R1} replaced by

\texttt{LDINC R1 R2 4} \ /\!/load from address in R1 then inc by 4
Peephole optimizations

- **Constant folding**
  
  \[
  \begin{align*}
  \text{ADD } \text{lit1}, \text{lit2}, \text{Rx} & \rightarrow \text{MOV } \text{lit1} + \text{lit2}, \text{Rx} \\
  \text{MOV } \text{lit1}, \text{Rx} & \rightarrow \text{MOV } \text{lit1} + \text{lit2}, \text{Ry} \\
  \text{ADD } \text{li2}, \text{Rx}, \text{Ry} & \rightarrow \text{MOV } \text{lit1} + \text{lit2}, \text{Ry}
  \end{align*}
  \]

- **Strength reduction**
  
  \[
  \begin{align*}
  \text{MUL } \text{operand}, 2, \text{Rx} & \rightarrow \text{SHIFTL } \text{operand}, 1, \text{Rx} \\
  \text{DIV } \text{operand}, 4, \text{Rx} & \rightarrow \text{SHIFTR } \text{operand}, 2, \text{Rx}
  \end{align*}
  \]

- **Null sequences**
  
  \[
  \begin{align*}
  \text{MUL } \text{operand}, 1, \text{Rx} & \rightarrow \text{MOV } \text{operand}, \text{Rx} \\
  \text{ADD } \text{operand}, 0, \text{Rx} & \rightarrow \text{MOV } \text{operand}, \text{Rx}
  \end{align*}
  \]
Peephole optimizations

- Combine operations
  
  \[
  \text{JEQ L1} \\
  \text{JMP L2} \quad \rightarrow \quad \text{JNE L2} \\
  \text{L1: \ldots}
  \]

- Simplifying
  
  \[
  \text{SUB operand, 0, Rx} \quad \rightarrow \quad \text{NEG Rx}
  \]

- Special cases (taking advantage of ++/--)
  
  \[
  \text{ADD 1, Rx, Rx} \quad \rightarrow \quad \text{INC Rx} \\
  \text{SUB Rx, 1, Rx} \quad \rightarrow \quad \text{DEC Rx}
  \]

- Address mode operations
  
  \[
  \text{MOV A R1} \\
  \text{ADD 0(R1) R2 R3} \quad \rightarrow \quad \text{ADD @A R2 R3}
  \]
Common subexpression elimination

- **Goal:** remove redundant computation, don’t calculate the same expression multiple times

1: \( A = B + C \times D \)

2: \( E = B + C \times D \)

Keep the result of statement 1 in a temporary and reuse for statement 2

- **Difficulty:** how do we know when the same expression will produce the same result?

1: \( A = B + C \times D \)

2: \( B = \text{<new value>} \)

B is “killed.” Any expression using B is no longer “available,” so we cannot reuse the result of statement 1 for statement 3

3: \( E = B + C \times D \)

- **This becomes harder with pointers (i.e., how do we know when B is killed?)**
Common subexpression elimination

• Two varieties of common subexpression elimination (CSE)
  • Local: within a single basic block
    • Easier problem to solve (why?)
  • Global: within a single procedure or across the whole program
    • Intra- vs. inter-procedural
    • More powerful, but harder (why?)
    • Will come back to these sorts of “global” optimizations later
CSE in practice

• Idea: keep track of which expressions are “available” during the execution of a basic block
  • Which expressions have we already computed?
  • Issue: determining when an expression is no longer available
    • This happens when one of its components is assigned to, or “killed.”
  • Idea: when we see an expression that is already available, rather than generating code, copy the temporary
    • Issue: determining when two expressions are the same
Maintaining available expressions

- For each 3AC operation in a basic block
  - Create name for expression (based on lexical representation)
  - If name not in available expression set, generate code, add it to set
    - Track temporary that holds expression and any variables used to compute expression
  - If name in available expression set, generate move instruction
  - If operation assigns to a variable, kill all dependent expressions
Example

Three address code

+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D

Available expressions:
Example

Three address code

+ A  B  T1
+ T1  C  T2
+ A  B  T3
+ T1  T2  C
+ T1  C  T4
+ T3  T2  D

Available expressions: “A+B”

Generated code

ADD A  B  R1
Example

Three address code

\[
+ A \ B \ T1 \\
+ T1 \ C \ T2 \\
+ A \ B \ T3 \\
+ T1 \ T2 \ C \\
+ T1 \ C \ T4 \\
+ T3 \ T2 \ D
\]

Generated code

ADD A B R1
ADD R1 C R2

Available expressions: “A+B” “T1+C”
Example

Three address code

+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D

Generated code

ADD A B R1
ADD R1 C R2
MOV R1 R3

Available expressions: “A+B” “T1+C”
Example

Three address code

+ A  B  T1
+ T1  C  T2
+ A  B  T3
+ T1  T2  C
+ T1  C  T4
+ T3  T2  D

Generated code

ADD A  B  R1
ADD R1  C  R2
MOV R1  R3
ADD R1  R2  R5; ST R5  C

Available expressions: “A+B” “T1+C” “T1+T2”
Example

Three address code

+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D

Generated code

ADD A B R1
ADD R1 C R2
MOV R1 R3
ADD R1 R2 R5; ST R5 C
ADD R1 C R4

Available expressions: “A+B” “T1+T2” “T1+C”
Example

Three address code

+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D

Generated code

ADD A B R1
ADD R1 C R2
MOV R1 R3
ADD R1 R2 R5; ST R5 C
ADD R1 C R4
ADD R3 R2 R6; ST R6 D

Available expressions: “A+B” “T1+T2” “T1+C” “T3+T2”
What are some downsides to this approach? Consider the two highlighted operations

Three address code

+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D

Generated code

ADD A B R1
ADD R1 C R2
MOV R1 R3
ADD R1 R2 R5; ST R5 C
ADD R1 C R4
ADD R3 R2 R6; ST R6 D
What are some downsides to this approach? Consider the two highlighted operations.

Three address code
+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D

Generated code
ADD A B R1
ADD R1 C R2
MOV R1 R3
ADD R1 R2 R5; ST R5 C
ADD R1 C R4
ST R5 D

This can be handled by an optimization called *value numbering*, which we will not cover now (although we may get to it later).
Aliasing

• One of the biggest problems in compiler analysis is to recognize aliases – different names for the same location in memory

• Aliases can occur for many reasons
  • Pointers referring to same location, arrays referencing the same element, function calls passing the same reference in two arguments, explicit storage overlapping (unions)

• Upshot: when talking about “live” and “killed” values in optimizations like CSE, we’re talking about particular variable names

• In the presence of aliasing, we may not know which variables get killed when a location is written to
Memory disambiguation

• Most compiler analyses rely on memory disambiguation
• Otherwise, they need to be too conservative and are not useful
• Memory disambiguation is the problem of determining whether two references point to the same memory location
  • Points-to and alias analyses try to solve this
• Will cover basic pointer analyses in a later lecture
Register allocation

- Simple code generation: use a register for each temporary variable, load from a variable on each read, store to a variable at each write

- Problems
  - Real machines have a limited number of registers – one register per temporary may be too many
  - Loading from and storing to variables on each use may produce a lot of redundant loads and stores

- Goal: allocate registers to variables and temporaries to do two things
  - Eliminate loads and stores
  - Minimize register spills
Register allocation basics

- One approach: assume all variables are in memory, load into registers as needed

- Alternate approach: start with unlimited pool of *virtual registers*
  - Whenever a new register is needed (e.g., a new temporary is created, a variable is loaded, etc.) create a new virtual register
    - No re-use of registers
  - Need to worry about aliasing

//a and b are aliased

LD a R1
LD b R2
Dealing with aliasing

• Immediately before loading a variable $x$
  • For each variable aliased to $x$ that is already in a register, save it to memory (i.e., perform a store)
  • This ensures that we load the right value
• Immediately before storing a variable $x$
  • For each register associated with a variable aliased to $x$, mark it as invalid
  • So next time we use the variable, we will reload it
• Conservative approach: assume all variables are aliased (in other words, reload from memory on each read, store to memory on each write)
  • Better alias analysis can improve this
  • At subroutine boundaries, still often use conservative analysis
Global vs. local

- Same distinction as global vs. local CSE
  - Local register allocation is for a single basic block
  - Global register allocation is for an entire function (but not interprocedural – why?)

- Will cover some local allocation strategies now, global allocation later
Top-down register allocation

- For each basic block
  - Find the number of references of each variable
  - Assign registers to variables with the most references
- Details
  - Keep some registers free for operations on unassigned variables and spilling
  - Store *dirty* registers at the end of BB (i.e., registers which have variables assigned to them)
  - Do not need to do this for temporaries (why?)
Bottom-up register allocation

For each tuple op A B C in a BB, do
    \( R_x = \text{ensure}(A) \)
    \( R_y = \text{ensure}(B) \)
    if A dead after this tuple, free(\( R_x \))
    if B dead after this tuple, free(\( R_y \))
    \( R_z = \text{allocate}(C) \) //could use \( R_x \) or \( R_y \)
    mark \( R_z \) dirty

At end of BB, for each dirty register
    generate code to store register into appropriate variable

- We will present this as if A, B, C are variables in memory. Can be modified to assume that A, B and C are in virtual registers, instead
Bottom-up register allocation

**ensure**(opr)
  if opr is already in register r
    return r
  else
    r = allocate(opr)
    generate load from opr into r
    return r

**free**(r)
  if r is marked dirty
    generate store
    mark r as free

**allocate**(opr)
  if there is a free r
    choose r
  else
    choose r with most distant use
    free(r)
    mark r associated with opr
    return r

- Requires calculating *def-use chains*
- Easy to calculate within a BB:
  - Start at end of block, all variables marked dead
  - When a variable is used, mark as live, record use
  - When a variable is defined, record def, variable dead above this
  - Creates chains linking uses of variables to where they were defined
- We will discuss how to calculate this across BBs later
Allocation considerations

- Use *register coloring* to perform global register allocation
  - Will see this next
- Find right order of optimizations and register allocation
  - Peephole optimizations can reduce register pressure, can make allocation better
  - CSE can actually *increase* register pressure
- Different orders of optimization produce different results
- Register allocation still an open research area
  - For example, how to do allocation for JIT compilers
Generating code from IR trees.

Idea:

if evaluating R takes more registers than L, it is better to

- evaluate R
- save result in a register
- evaluate L
- do the (binary) operation

This is because result of R takes a register.
Determining Register Needs

Assuming both register-to-register and storage-to-register instructions

For ID nodes (these are leaf nodes):
- left: 1 register
- right: 0 registers (*use op from memory*)

Register need of the combined tree:
\[ X = \begin{cases} \text{L+1,} & \text{if } R = L \\ \max(R, L), & \text{if } R \neq L \end{cases} \]
Algorithm for Code Generation Using Register-Need Annotations

Recursive tree algorithm. Each step leaves result in R1
(R1 is the first register in the list of available registers)

Case 1: right branch is an ID:
• generate code for left branch
• generate OP ID,R1   (op,R1,ID,R1)

Case 2: min(L,R) >= max available registers:
• generate code for right branch
• spill R1 into a temporary T
• generate code for left branch
• generate OP T,R1
Remaining cases: at least one branch needs fewer registers than available

Case 3: \( R < \text{max available registers} \):
- generate code for left branch
- remove first register (R1) from available register list
- generate code for right branch (result in R2)
- generate OP R2,R1

Case 4: \( L < \text{max available registers} \):
- temporarily swap R1 and R2
- generate code for right branch
- remove first register (R2) from available register list
- generate code for left branch (result in R1)
- generate OP R2,R1
Example Tree Code Generation

\[(A-B)+((C+D)+(E*F))\]

Note: life gets more interesting if some of the leaves are reused/across trees