CSCI-GA.2130-001
Compiler Construction
Lecture 12: Code Generation I

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
**Requirements**

- Preserve semantic meaning of source program
- Make effective use of available resources of target machine
- Code generator itself must run efficiently

**Challenges**

- Problem of generating optimal target program is undecidable
- Many subproblems encountered in code generation are computationally intractable
Main Tasks of Code Generator

• **Instruction selection**: choosing appropriate target-machine instructions to implement the IR statements

• **Registers allocation and assignment**: deciding what values to keep in which registers

• **Instruction ordering**: deciding in what order to schedule the execution of instructions
Design Issues of a Code Generator

Input

- three-address presentations (quadruples, triples, ...)
- Virtual machine presentations (bytecode, stack-machine, ...)
- Linear presentation (postfix, ...)
- Graphical presentation (syntax trees, DAGs, ...)

Design Issues of a Code Generator

Target program

- Instruction set architecture (RISC, CISC)
- Producing absolute machine-language program
- Producing relocatable machine-language program
- Producing assembly language programs
Design Issues of a Code Generator

Instruction Selection

The complexity of mapping IR program into code-sequence for target machine depends on:

- Level of IR (high-level or low-level)
- Nature of instruction set (data type support)
- Desired quality of generated code (speed and size)
Design Issues of a Code Generator

Register Allocation

• Selecting the set of variables that will reside in registers at each point in the program

Register Assignment

• Picking the specific register that a variable will reside in
Design Issues of a Code Generator

Evaluation Order

– Selecting the order in which computations are performed
– Affects the efficiency of the target code
– Picking a best order is NP-complete
– Some orders require fewer registers than others
Simple Target-Machine

• Load/store operations
  – $LD \ dst, \ addr$
  – $ST \ x, \ r$

• Computation operations
  – $OP \ dst, \ src1, \ src2$

• Jump operations
  – $BR \ L$

• Conditional jumps
  – $Bcond \ r, \ L$

• Byte addressable

• $n$ registers: R0, R1, ... Rn-1
• Addressing modes
  – variable name
  – a(r) means contents(a + contents(r))
  – *a(r) means:
    contents(contents(a + contents(r)))
  – immediate: #constant (e.g. LD R1, #100)
Simple Target-Machine

Cost

- cost of an instruction = $1 + \text{cost of operands}$
- cost of register operand = 0
- cost involving memory and constants = 1
- cost of a program = sum of instruction costs
Examples

\[ X = Y - Z \]
- **LD** R1, y  // R1 = y
- **LD** R2, z  // R2 = z
- **SUB** R1, R1, R2  // R1 = R1 - R2
- **ST** x, R1  // x = R1

\[ b = a[i] \]  
(8-byte elements)
- **LD** R1, i  // R1 = i
- **MUL** R1, R1, 8  // R1 = R1 \times 8
- **LD** R2, a(R1)  // R2 = \text{contents}(a + \text{contents}(R1))
- **ST** b, R2  // b = R2

\[ x = *p \]
- **LD** R1, p  // R1 = p
- **LD** R2, 0(R1)  // R2 = \text{contents}(0 + \text{contents}(R1))
- **ST** x, R2  // x = R2
More Examples

• $a[j] = c$
• $*p = y$
• if $X < Y$ goto $L$
Generating Code for Handling the Stack

Size and layout of activation records are determined by the code generator using information from symbol table.

- Saves return address at beginning of activation record of callee.
- Constants giving address of beginning of activation record of callee.
- Transfers control to target code of procedure callee.

```
ST callee.staticArea, #here + 20
BR callee.codeArea
CALL callee
```

```
BR *callee.staticArea
```
Assumptions:
• c and p start at 100 and 200
• activation records for c and p: 300 and 364

The above assumptions mean **static allocation** ... What if it is not the case?
Stack Allocation

• The position of the activation record is not known until runtime
• Must use relative address to access elements of the activation record
• We need a register to keep track of the top of the stack

Remember: The book assumes, for simplicity, that stack grows toward the high memory. The reality is the opposite. The code we see here is based on the book convention.
LD    SP, #stackStart
ADD   SP, SP, #caller.recordSize
ST    *SP, #here + 16
BR    callee.codeArea
SUB   SP, SP, #caller.recordSize
HALT
BR    *0(SP)
Assumptions:
• First word in each activation is the return address
• start address of p, q, and m: 100, 200, and 300
• stack starts at 600

```
   // code for m
action1
   call q
action2
halt

   // code for p
action3
return

   // code for q
action4
call p
action5
call q
action6
call q
return
```

```
100:  LD SP, #600           // code for m
108:  ACTION1              // initialize the stack
128:  ADD SP, SP, #msize   // call sequence begins
136:  ST *SP, #152         // push return address
144:  BR 300               // call q
152:  SUB SP, SP, #msize   // restore SP
160:  ACTION1              // code for p
180:  HALT

200:  ACTION3              // return
220:  BR *0(SP)            // code for q

300:  ACTION4              // contains a conditional jump to 456
320:  ADD SP, SP, #qsize   // push return address
328:  ST *SP, #344         // call p
336:  BR 200               // code for q
344:  SUB SP, SP, #qsize
352:  ACTION5
372:  ADD SP, SP, #qsize
380:  BR *SP, #396         // push return address
388:  BR 300               // call q
396:  SUB SP, SP, #qsize
404:  ACTION6
424:  ADD SP, SP, #qsize
432:  ST *SP, #440         // push return address
440:  BR 300               // call q
448:  SUB SP, SP, #qsize
456:  BR *0(SP)            // return

600:  // stack starts here
```
Basic Blocks and Flow Graphs

- Graph presentation of intermediate code
- Nodes of the graph are called basic blocks
- Edges indicate which block follows which other block.

- The graph is useful for doing better job in:
  - Register allocation
  - Instruction selection
Basic Blocks

• Definition: maximal sequence of consecutive instructions such that
  – Flow of control can only enter the basic block from the first instruction
  – Control leaves the block only at the last instruction

• Each instruction is assigned to exactly one basic block
1) \( i = 1 \)
2) \( j = 1 \)
3) \( t1 = 10 \times i \)
4) \( t2 = t1 + j \)
5) \( t3 = 8 \times t2 \)
6) \( t4 = t3 - 88 \)
7) \( a[t4] = 0.0 \)
8) \( j = j + 1 \)
9) if \( j \leq 10 \) goto (3)
10) \( i = i + 1 \)
11) if \( i \leq 10 \) goto (2)
12) \( i = 1 \)
13) \( t5 = i - 1 \)
14) \( t6 = 88 \times t5 \)
15) \( a[t6] = 1.0 \)
16) \( i = i + 1 \)
17) if \( i \leq 10 \) goto (13)
Fist we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader.

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

```plaintext
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```
First we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader.
   1) $i = 1$
   2) $j = 1$

2. Any instruction that is the target of a conditional or unconditional jump is a leader.
   3) $t1 = 10 \times i$
   4) $t2 = t1 + j$
   5) $t3 = 8 \times t2$
   6) $t4 = t3 - 88$
   7) $a[t4] = 0.0$
   8) $j = j + 1$
   9) if $j <= 10$ goto (3)
   10) $i = i + 1$
   11) if $i <= 10$ goto (2)
   12) $i = 1$

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
   13) $t5 = i - 1$
   14) $t6 = 88 \times t5$
   15) $a[t6] = 1.0$
   16) $i = i + 1$
   17) if $i <= 10$ goto (13)
First we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader.
   
   1) $i = 1$
   2) $j = 1$

2. Any instruction that is the target of a conditional or unconditional jump is a leader.
   
   3) $t1 = 10 \times i$
   4) $t2 = t1 + j$
   5) $t3 = 8 \times t2$
   6) $t4 = t3 - 88$
   7) $a[t4] = 0.0$
   8) $j = j + 1$
   9) if $j \leq 10$ goto (3)
   10) $i = i + 1$
   11) if $i \leq 10$ goto (2)
   12) $i = 1$

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

   13) $t5 = i - 1$
   14) $t6 = 88 \times t5$
   15) $a[t6] = 1.0$
   16) $i = i + 1$
   17) if $i \leq 10$ goto (13)

Basic block starts with a leader instruction and stops before the following leader instruction.
ENTRY

B₁

i = 1

B₂

j = 1

B₃

t₁ = 10 * i
t₂ = t₁ + j
t₃ = 8 * t₂
t₄ = t₃ - 88
j = j + 1
if j <= 10 goto B₃

B₄

i = i + 1
if i <= 10 goto B₂

B₅

i = 1

B₆

t₅ = i - 1
t₆ = 88 * t₅
a[t₆] = 1.0
i = i + 1
if i <= 10 goto B₆

EXIT
Loops

• Most programs spend most of their execution time executing loops
• It is thus important to generate good code for loops.
• A set of nodes L in a flow graph is a loop if L contains a node e such that
  – e is not ENTRY
  – Only node e has predecessor outside L
  – Every node in L has a nonempty path, completely within L, to e
ENTRY

\( B_1 \) i = 1

\( B_2 \) j = 1

\( B_3 \)
- \( t_1 = 10 \times i \)
- \( t_2 = t_1 + j \)
- \( t_3 = 8 \times t_2 \)
- \( t_4 = t_3 - 88 \)
- \( j = j + 1 \)
- if \( j \leq 10 \) goto \( B_3 \)

\( B_4 \)
- \( i = i + 1 \)
- if \( i \leq 10 \) goto \( B_2 \)

\( B_5 \) i = 1

\( B_6 \)
- \( t_5 = i - 1 \)
- \( t_6 = 88 \times t_5 \)
- \( a[t_6] = 1.0 \)
- \( i = i + 1 \)
- if \( i \leq 10 \) goto \( B_6 \)

EXIT
DAG Representation of Basic Blocks

• Leaves for initial values of variables (we may not know the values so we use $a_0$, $b_0$, ...)

• Node for each expression

• Node label is the expression operation

• Next to the node we put the variable(s) for which the node produced last definition

• Children of a node consist of nodes producing last definition of operands
Finding Local Common Subexpressions

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= a - d
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
d &= a - d \\
c &= d + c
\end{align*}
\]
Construct the DAG for the basic block

\[
\begin{align*}
    d &= b \times c \\
    e &= a + b \\
    b &= b \times c \\
    a &= e - d
\end{align*}
\]
Dead Code Elimination

From the basic block DAG:
• Remove any root node that has no live variables
• Repeat until no nodes can be removed
Assumptions: a and b are live but c and e are not.

\[
a = b + c; \\
b = b - d \\
c = c + d \\
e = b + c
\]
More Basic-Block Optimizations

• Eliminate unnecessary computations such as algebraic identities:
  – $x+0 = 0+x = x$
  – $x*1 = 1*x = x$
  – $x-0=x$
  – $x/1=x$

• Reduction in strength: replace a more expensive operator by a cheaper one:
  – $x^2 = x*x$
  – $2*x = x+x$
  – $x/2 = x*0.5$

• Constant folding: evaluate constant expressions at compile time and replace the constant expressions by their values.
So

- Skim: 8.3.3, 8.5.4, 8.5.5, 8.5.6, and 8.5.7
- Read: 8.1 -> 8.5