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

Hubertus Franke
frankeh@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: $R0, \ R1, \ ... \ Rn-1$
Simple Target-Machine

- **Addressing modes**
  - variable name
  - \(a(r)\) means \(\text{contents}(a + \text{contents}(r))\)
  - \(*a(r)\) means:
    - \(\text{contents}(\text{contents}(a + \text{contents}(r)))\)
  - immediate: \#constant (e.g. LD R1, #100)
Simple Target-Machine

Cost

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

\[ X = Y - Z \]

\[
\begin{align*}
\text{LD} & \quad R1, y & \quad \text{// } R1 = y \\
\text{LD} & \quad R2, z & \quad \text{// } R2 = z \\
\text{SUB} & \quad R1, R1, R2 & \quad \text{// } R1 = R1 - R2 \\
\text{ST} & \quad x, R1 & \quad \text{// } x = R1 \\
\end{align*}
\]

\[ b = a[i] \]

(8-byte elements)

\[
\begin{align*}
\text{LD} & \quad R1, i & \quad \text{// } R1 = i \\
\text{MUL} & \quad R1, R1, 8 & \quad \text{// } R1 = R1 \times 8 \\
\text{LD} & \quad R2, a(R1) & \quad \text{// } R2 = \text{contents}(a + \text{contents}(R1)) \\
\text{ST} & \quad b, R2 & \quad \text{// } b = R2 \\
\end{align*}
\]

\[ x = \ast p \]

\[
\begin{align*}
\text{LD} & \quad R1, p & \quad \text{// } R1 = p \\
\text{LD} & \quad R2, 0(R1) & \quad \text{// } R2 = \text{contents}(0 + \text{contents}(R1)) \\
\text{ST} & \quad x, R2 & \quad \text{// } x = R2 \\
\end{align*}
\]
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
RETURN
```
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
code for the first procedure
HALT

ADD  SP, SP, #caller.recordSize
ST   *SP, #here + 16
BR   callee.codeArea
SUB  SP, SP, #caller.recordSize
BR   *O(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
to
action5
call q
action6
call q
return
```

```
100: LD SF, #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: ACTION3     // code for p
180: HALT

... // contains a conditional jump to 456
300: ACTION4     // code for q
320: ADD SP, SP, #qsize  // push return address
328: ST *SP, #344  // call p
336: BR 200
344: SUB SP, SP, #qsize  // call q
352: ACTION5
372: ADD SP, SP, #qsize  // push return address
380: BR *SP, #306  // call q
388: BR 300
396: SUB SP, SP, #qsize  // return
404: ACTION6
424: ADD SP, SP, #qsize  // stack starts here
432: ST *SP, #440
440: BR 300
448: SUB SP, SP, #qsize
456: BR *0(SP)
```
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.

\[
\begin{align*}
1) & \quad i = 1 \\
2) & \quad j = 1 \\
3) & \quad t1 = 10 \times i \\
4) & \quad t2 = t1 + j \\
5) & \quad t3 = 8 \times t2 \\
6) & \quad t4 = t3 - 88 \\
7) & \quad a[t4] = 0.0 \\
8) & \quad j = j + 1 \\
9) & \quad \text{if } j \leq 10 \text{ goto (3)} \\
10) & \quad i = i + 1 \\
11) & \quad \text{if } i \leq 10 \text{ goto (2)} \\
12) & \quad i = 1 \\
13) & \quad t5 = i - 1 \\
14) & \quad t6 = 88 \times t5 \\
15) & \quad a[t6] = 1.0 \\
16) & \quad i = i + 1 \\
17) & \quad \text{if } i \leq 10 \text{ goto (13)}
\end{align*}
\]
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) \( t_1 = 10 \times i \)
   4) \( t_2 = t_1 + j \)
   5) \( t_3 = 8 \times t_2 \)
   6) \( t_4 = t_3 - 88 \)
   7) \( a[t_4] = 0.0 \)
   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) \( t_5 = i - 1 \)
   14) \( t_6 = 88 \times t_5 \)
   15) \( a[t_6] = 1.0 \)
   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.

Basic block starts with a leader instruction and stops before the following leader instruction.
\begin{align*}
B_1 & \quad i = 1 \\
B_2 & \quad j = 1 \\
B_3 & \quad t_1 = 10 \cdot i \\
& \quad t_2 = t_1 + j \\
& \quad t_3 = 8 \cdot t_2 \\
& \quad t_4 = t_3 - 88 \\
& \quad j = j + 1 \\
& \quad \text{if } j \leq 10 \text{ goto } B_3 \\
B_4 & \quad i = i + 1 \\
& \quad \text{if } i \leq 10 \text{ goto } B_2 \\
B_5 & \quad i = 1 \\
B_6 & \quad t_5 = i - 1 \\
& \quad t_6 = 88 \cdot t_5 \\
& \quad a[t_6] = 1.0 \\
& \quad i = i + 1 \\
& \quad \text{if } i \leq 10 \text{ goto } B_6
\end{align*}
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

B1
i = 1

B2
j = 1

B3
\begin{align*}
  t_1 &= 10 \times i \\
  t_2 &= t_1 + j \\
  t_3 &= 8 \times t_2 \\
  t_4 &= t_3 - 88 \\
  j &= j + 1 \\
  \text{if } j \leq 10 \text{ goto } B_3
\end{align*}

B4
\begin{align*}
  i &= i + 1 \\
  \text{if } i \leq 10 \text{ goto } B_2
\end{align*}

B5
i = 1

B6
\begin{align*}
  t_5 &= i - 1 \\
  t_6 &= 88 \times t_5 \\
  a[t_6] &= 1.0 \\
  i &= i + 1 \\
  \text{if } i \leq 10 \text{ goto } B_6
\end{align*}

EXIT
DAG Representation of Basic Blocks

• Leaves for initial values of variables (we may not know the values so we use a0, b0, ...)
• 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
Optimizations

• We can often achieve optimizations by first focusing on basic blocks
  – Local common expressions
  – Dead code elimination
  – Statement reordering
  – Algebraic rewriting
Live Variables

• The use of a name in a three-address statement is defined as follows. Suppose three-address statement \( i \) assigns a value to \( x \).

• If statement \( j \) has \( x \) as an operand, and control can flow from statement \( i \) to \( j \) along a path that has no intervening assignments to \( x \), then we say statement \( j \) uses the value of \( x \) computed at statement \( i \).

• We further say that \( x \) is live at statement \( i \).
Finding Local Common Subexpressions

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

\[ a = b + c; \]
\[ b = b - d \]
\[ c = c + d \]
\[ e = b + c \]
Algebraic Rewrite

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

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

• Constant folding: evaluate constant expressions at compile time and replace the constant expressions by their values.
Representation of Array References

• Array references need special consideration:

\[
\begin{align*}
x &= a[i] \\
a[j] &= y \\
z &= a[i]
\end{align*}
\]

• Can we use common sub expression to optimize?

\[
\begin{align*}
x &= a[i] \\
a[j] &= y \\
z &= a[i] \quad \rightarrow \quad x &= a[i] \\
a[j] &= v \\
z &= x
\end{align*}
\]
Representation of Array References

- Introduce special ops ( =[] and []= ) to DAG

\[
x = a[i] \\
a[j] = y \\
z = a[i]
\]
Representation of Array References

- Introduce special ops ( =[] and [[]]= ) to DAG
- Sometimes have go beyond children:

\[
b = 12 + a \\
x = b[i] \\
b[j] = y
\]
Pointer Assignments

- Pointer aliasing is a problem:

  \[
  x = *p \\
  *q = y
  \]

- Treat similar to array with new operators:
  ( *= and =* )

- Kills all node constructed in DAG till then

- Some pointer analysis is possible:

  \[
  p = &x \\
  *p = y
  \]
So

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