Code Generation

Rupesh Nasre.

CS3300 Compiler Design
IIT Madras
Aug 2015
Role of Code Generator

• From IR to target program.
• Must preserve the semantics of the source program.
  – Meaning *intended* by the programmer in the original source program should carry forward in each compilation stage until code-generation.
• Target code should be of high quality
  – execution time or space or energy or …
• Code generator itself should run efficiently.
  1. instruction selection,
  2. register allocation and
  3. instruction ordering.
Code Generator in Reality

- The problem of generating an optimal target program is undecidable.
- Several subproblems are NP-Hard (such as register allocation).
- Need to depend upon
  - Approximation algorithms
  - Heuristics
  - Conservative estimates
Input + Output

Intermediate representation

Code Generator

Target machine code

- 3AC (Quadruples / Triples / Indirect triples)
- VM instructions (bytecodes / stack machine codes)
- Linear representations (postfix)
- Graphical representation (syntax trees / DAGs)

- RISC (many registers, 3AC, simple addressing modes, simple ISA)
- CISC (few registers, 2AC, variety of addressing modes, several register classes, variable length instructions, instructions with side-effects)
- Stack machine (push / pop, stack top uses registers, used in JVM, JIT compilation)

It helps to assume an assembler. Imagine if in A3 you had to generate machine code and manipulate bits rather than generating x86 assembly.
IR and Target Code

Intermediate representation

Code Generator

Target machine code

\{ 
R0 = y 
R0 = R0 + z 
x = R0 
\}

\{ 
LD R0, y 
ADD R0, R0, z 
ST x, R0 
\}

What is the issue with this kind of code generation?
IR and Target Code

Intermediate representation

\[
\begin{align*}
R0 &= y \\
R0 &= R0 + z \\
x &= R0
\end{align*}
\]

Target machine code

\[
\begin{align*}
LD R0, y \\
ADD R0, R0, z \\
ST x, R0
\end{align*}
\]

Code Generator

Generate code for

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

\[
\begin{align*}
LD R0, b \\
ADD R0, R0, c \\
ST a, R0 \\
LD R0, a \\
ADD R0, R0, e \\
ST d, R0
\end{align*}
\]
Instruction Selection

- Complexity of instruction selection depends upon
  - Level of the IR
    Low-level IR can help generate more efficient code. e.g., *intsize* versus 4.
  - Nature of the ISA
    Uniformity and completeness of ISA affects the code. e.g., floats required to be loaded in special registers.
  - Desired quality of the generated code
    Context and amount of information to process affects the code quality. e.g., INC a versus *LD R0, a*; *ADD R0, R0, #1*; *ST a, R0*
Register Allocation

- Register allocation involves
  - *Allocation*: which variables to be put into registers
  - *Assignment*: which register to use for a variable
- Finding an optimal assignment of registers to variables is NP-Complete.
- Architectural conventions complicate matters.
  - Combination of registers used for double-precision arithmetic.
  - Result is stored in accumulator.
  - Registers are reserved for special instructions.
  - ...

Thought exercise: How to use graph coloring for register allocation?
Instruction Ordering

- Instruction order affects execution efficiency.
- Picking the best order is NP-complete.
- Optimizer / Code generator needs to look at multiple instructions at a time.

**Classwork:** Create an example IR whose generated code results in the same meaning but different efficiency for different orders.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R0 = a</td>
<td>1</td>
<td>R0 = a</td>
<td>1</td>
<td>R0 = a</td>
</tr>
<tr>
<td>2</td>
<td>R1 = b</td>
<td>2</td>
<td>R1 = b</td>
<td>2</td>
<td>R1 = b</td>
</tr>
<tr>
<td>3</td>
<td>R2 = c</td>
<td>3</td>
<td>R2 = R0 + R1</td>
<td>3</td>
<td>R0 = c</td>
</tr>
<tr>
<td>4</td>
<td>R3 = R0 + R1</td>
<td>4</td>
<td>R2 = R0 + R1</td>
<td>5</td>
<td>R3 = R2 + R0</td>
</tr>
<tr>
<td>5</td>
<td>R4 = R2 + R3</td>
<td>6</td>
<td>d = R4</td>
<td>6</td>
<td>d = R3</td>
</tr>
</tbody>
</table>
## A Typical Target Machine Model

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>LD R1, x</td>
</tr>
<tr>
<td>Store</td>
<td>ST R1, x</td>
</tr>
<tr>
<td>Computation</td>
<td>SUB R1, R2, R3</td>
</tr>
<tr>
<td>Unconditional Jump</td>
<td>BR main</td>
</tr>
<tr>
<td>Conditional Jump</td>
<td>BLTZ R1, main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>LD R1, 100000</td>
</tr>
<tr>
<td>Named / Variable</td>
<td>LD R1, x</td>
</tr>
<tr>
<td>Variable Indexed</td>
<td>LD R1, a(R2)</td>
</tr>
<tr>
<td>Immediate Indexed</td>
<td>LD R1, 100(R2)</td>
</tr>
<tr>
<td>Indirect</td>
<td>LD R1, *100(R2)</td>
</tr>
<tr>
<td>Immediate</td>
<td>LD R1, #100</td>
</tr>
</tbody>
</table>
Example Code Generation using our Target Machine Model

Source

... 
  x = y - z

IR

... 
  R1 = y 
  R2 = z 
  R1 = R1 – R2 
  x = R1

Target code

... 
  LD R1, y 
  LD R2, z 
  SUB R1, R1, R2 
  ST x, R1

Optimized target code

... 
  ; y already in R1 
  LD R2, z 
  SUB R1, R1, R2 
  ST x, R1

... 
  ; x is not used 
  LD R1, y 
  LD R2, z
Homework

- Exercises 8.2.3 from ALSU book.
Basic Blocks and CFG

• A basic block is a maximal sequence of consecutive 3AC instructions such that
  – Single-entry: Control-flow enters the basic-block through only the first instructions in the block.
  – Single-exit: Control leaves the block only after the last instruction.

• Thus, if control reaches a basic block, all instructions in it are executed in sequence.
  – No branching from in-between or no jumps to in-between instructions.

• Basic-blocks together form a control-flow graph.
for (ii = 0; ii < 10; ++ii) {
    for (jj = 0; jj < 10; ++jj) {
        a[ii][jj] = 0;
    }
}
for (ii = 0; ii < 10; ++ii)
a[ii][ii] = 1;
for (ii = 0; ii < 10; ++ii) {
    for (jj = 0; jj < 10; ++jj) {
        a[ii][jj] = 0;
    }
}
for (ii = 0; ii < 10; ++ii)
a[ii][ii] = 1;
i = 0
L2:
j = 0
L1:
t1 = 10 * i
t2 = t1 + j
t3 = 4 * t2
t4 = t3 - 44
a[t4] = 1
j = j + 1
if j < 10 goto B3
i = i + 1
if i < 10 goto L1
i = i + 1
if i < 10 goto L2
i = 1
L3:
t5 = i - 1
t6 = 44 * t5
a[t6] = 1
i = i + 1
if i < 10 goto L3
Optimizations using CFG

• **Local**: within a basic-block
  - Local common sub-expressions
  - Deal-code elimination
  - Use of algebraic identities

• **Global**: across blocks
  - Common sub-expression elimination
  - Strength reduction
  - Data-flow analysis
Local Common Sub-expressions Elimination

\[ a = b + c \]
\[ b = a - d \]
\[ c = b + c \]
\[ d = a - d \]

\[ a + a \ast (b - c) + (b - c) \ast d \]

- Does not distinguish properly between different variable instances.
- It is unclear why certain variable should be used or a new one should be formed.
- We need use-def information.
Local Common Sub-expressions Elimination

- Variables have initial DEFs.
- Each DEF creates a new instance of the variable (recall SSA).
- Each USE refers to the latest DEF.

\[
\begin{align*}
a & = b + c \\
b & = a - d \\
c & = b + c \\
d & = a - d \\
a_1 & = b_0 + c_0 \\
b_1 & = a_1 - d_0 \\
c_1 & = b_1 + c_0 \\
d_1 & = a_1 - d_0
\end{align*}
\]
Local Common Sub-expressions Elimination

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

Classwork: Find the Basic Block DAG (expression DAG) for the above Basic Block.

No common expressions
Dead-code Elimination

- Remove root from the DAG that have no live variables attached.
  - There could be multiple roots in the DAG.
  - We may be able to apply this repeatedly.

Assuming a and b are live (used later) while c and e are not, then
- We can remove e1.
- Once e1 is removed, c1 can also be removed.
Algebraic Identities

• Algebraic properties
  - $x + 0 = 0 + x = x$
  - $x - 0 = x$
  - $x \times 1 = 1 \times x = x$
  - $x / 1 = x$

• Strength reduction
  - $x^2 = x \times x$
  - $2 \times x = x + x$
  - $x / 2 = x \times 0.5$

• Constant folding
  - $2 \times 3.14 = 6.28$
Algebraic Identities

• Commutativity and Associativity
  – DAG construction can help us here.
  – Apart from checking \textit{left op right}, we could also check \textit{right op left} for commutativity.
    \begin{itemize}
    \item e.g., \((a + b) + (b + a)\).
    \item e.g., \(a = b + c; e = c + d + b;\)
  \end{itemize}

• Some algebraic laws are not obvious.
  – e.g., Can you optimize \textit{if} \((x > y) a = b + x + c − y\)?
    However, we need to worry about underflows.
Array References

- Array references cannot be treated like usual variables.

We represent $a[\text{ii}]$ as a node with two or three children depending upon whether it is rvalue or lvalue.

How do you decide the order in which assignments are executed?

Wrong

Correct
Array References

- Array references cannot be treated like usual variables.

\[
\begin{align*}
x &= a1 \\
a2 &= y \\
z &= a1 \\
x &= a[\text{ii}] \\
a[\text{jj}] &= y \\
z &= a[\text{ii}] \\
x &= a[\text{ii}] \\
b[\text{jj}] &= y \\
z &= a[\text{ii}]
\end{align*}
\]

Depending upon how much time a compiler can afford,
- it would either analyze if \(a[\text{ii}]\) and \(b[\text{jj}]\) are referring to the same memory location \textbf{OR}
- conservatively assume that they \textbf{MAY} be referring to the same location.
Aliasing

- The issue with array references is called aliasing.
- Two expressions may refer to the same memory location at the execution time.
  - `a[iii]` and `a[jj]`
  - `*p` and `*q`
  - Pass by reference variables
- Local processing may fail to identify aliasing
  - Precise alias analysis is computationally difficult.
Peephole Optimization

• Consider a sliding window of instructions and optimize it.

• Repeated passes are often helpful.
  - Redundant-instruction elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - ...
Peephole Optimization

• Consider a sliding window of instructions and optimize it.

• Repeated passes are often helpful.
  – Redundant load/store elimination
  – Dead-code elimination
  – Control-flow optimization
  – Algebraic simplifications
  – Use of machine idioms
  – ...
Peephole Optimization

- Consider a sliding window of instructions and optimize it.
- Repeated passes are often helpful.
  - Redundant load/store elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - ...

```
if debug == 1 goto L1
goto L2
L1:
  print debug info
L2:
  ...
```

```
debug = 0
constant propagation
L2:
  ...
```
Peephole Optimization

• Consider a sliding window of instructions and optimize it.

• Repeated passes are often helpful.
  - Redundant load/store elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - ...

Remove “L1: goto L2” if no jumps to it.
Can be generalized to conditional jump to L1.
Peephole Optimization

• Consider a sliding window of instructions and optimize it.
• Repeated passes are often helpful.
  – Redundant load/store elimination
  – Dead-code elimination
  – Control-flow optimization
  – Algebraic simplifications
  – Use of machine idioms
  – ...

\[
\begin{align*}
x &= x + 0 \\
z &= y \times 32 \\
\end{align*}
\]
Peephole Optimization

- Consider a sliding window of instructions and optimize it.
- Repeated passes are often helpful.
  - Redundant load/store elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - ...
Register Allocation

• Memory hierarchy: Network, File system, Main memory, L3 cache, L2, L1, Registers.
  – Capacity reduces, access time reduces.

• Critical to allocate and assign registers for efficiency.
  – Register versus Memory could be ~10x performance difference.

• C allows register variables.
  – register int a; // not always a good idea.
  – register int a asm(“r12”); // tries a specific register.
  – gcc -ffixed-r12 … // reserve r12.
Register Allocation

**Classwork**: Allocate registers for the following code.

```plaintext
while (b) {
    a = b + c
    d = d - b
    e = a + f
    if (e) {
        b = d + f
        e = a - c
        if (b) goto fun
    } else {
        f = a - d
    }
    b = d + c
}
print b, c, d, e, f
```

- First-Come-First-Served way is often **not** the best policy for register allocation.
- We need to perform some analysis to find out the benefit of allocating registers to variables.
- We may have to assign cost / benefit to various operations within a loop.
- What if we say that K registers would be allocated to the top K variables that have the maximum number of uses?
- By paying a small spilling cost, we may be able to increase the benefit of K registers to more than K variables.

\[
\text{benefit}(x, B) = F(\text{use}(x, B), \text{live}(x, B))
\]

Variable x, Basic block B

**use** returns the number of uses.

**live** returns 0 or 1 based on if x is live after leaving B and defined in B.
benefit(x, B) = F(use(x, B), live(x, B))

use returns number of uses. live returns 0 or 1 based on if x is live after leaving B and also defined in B.

use(a, B1) = 1, live(a, B1) = 1
use(a, B2) = 1, live(a, B2) = 0
use(b, B3) = 0, live(b, B3) = 1
...

Overall benefit S(x) = sum(benefit(x, B)) for all B

Say, S(a) = 4, S(b) = 5, S(c) = 3, S(d) = 6, S(e) = 4, S(f) = 4.

- Assign R0, R1, R2 to a, b and d globally (global allocation).
- Use remaining register R3 inside blocks (local allocation).
Allocation

- R1 and R2 remain assigned to b and d throughout.
- R3 is loaded repeatedly inside the loop as an auxiliary register.
- a is not live at the start, hence it is not loaded initially.
- At the end of the loop, the register values are stored back.

*Overall benefit* $S(x) = \text{sum}(\text{benefit}(x, B))$ for all B

Say, $S(a) = 4$, $S(b) = 5$, $S(c) = 3$, $S(d) = 6$, $S(e) = 4$, $S(f) = 4$.

- Assign R0, R1, R2 to a, b and d globally (global allocation).
- Use remaining register R3 inside blocks (local allocation).
Register Allocation as Graph Coloring

- Vertices? Edges?
- **Vertices**: Variables (or their instances)
- **Edges**: Co-Live information
  - If x and y are live at the same program point, add an (undirected) edge between x and y.
- **Vertex coloring** colors neighbors differently.
  - Thus, vertex coloring colors x and y differently, if they are live at the same program point.
  - This means, x and y should not use the same register.
  - **Corollary**: if x and z have the same color, they can reuse the register (at different program points).
Live Ranges

\[
\begin{align*}
  a &= b + c \\
  d &= d - b \\
  e &= a + f \\
  \text{if } (e) \{ \\
\end{align*}
\]

\[
\begin{align*}
  f &= a - d \\
  \text{acde} \\
  \text{cdef} \\
\end{align*}
\]

\[
\begin{align*}
  b &= d + f \\
  e &= a - c \\
  \text{if } (b) \text{ goto } \text{fun} \\
\end{align*}
\]

\[
\begin{align*}
  b &= d + c \\
  \text{while } (b) \\
\end{align*}
\]

\[
\begin{align*}
  \text{bcdef} \\
  \text{bcdef} \\
  \text{bcdef} \\
  \text{bcdef} \\
  \text{bcdef} \\
\end{align*}
\]

\[
\begin{align*}
  \text{print } b, c, d, e, f \\
\end{align*}
\]

( obtained from analyzing fun)
This means, in basic block B1, \( b \) and \( e \) could use the same register.

**Classwork:** Try it for

```plaintext
a = b + c
d = d - b
e = a + f
if (e) {
```

**What is the issue with what we did?**
Coloring gave us the maximum number of registers required for the program.
However, in practice, the number of registers is fixed.
Therefore, we need to generate spill code for storing a variable into memory (ST $x$, $R$) and then reload the register with the next variable (LD $R$, $y$).
Data Flow Analysis

• Flow-sensitive: Considers the control-flow in a function

• Operates on a flow-graph with nodes as basic-blocks and edges as the control-flow

• Examples
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination

What is the value of b?
Reaching Definitions

• Every assignment is a definition

• A definition $d$ reaches a program point $p$ if there exists a path from the point immediately following $d$ to $p$ such that $d$ is not killed along the path.

What definitions reach B3?
DFA Equations

- $\text{in}(B) = \text{set of data flow facts entering block B}$
- $\text{out}(B) = \ldots$
- $\text{gen}(B) = \text{set of data flow facts generated in B}$
- $\text{kill}(B) = \text{set of data flow facts from the other blocks killed in B}$
DFA for Reaching Definitions

- in(B) = U out(P) where P is a predecessor of B
- out(B) = gen(B) U (in(B) – kill(B))
- Initially, out(B) = { }

\[
\begin{align*}
gen(B0) &= \{D1, D2\} & \text{kill}(B0) &= \{D3, D4, D6\} \\
gen(B1) &= \{D3, D4\} & \text{kill}(B1) &= \{D0, D1, D2, D6\} \\
gen(B2) &= \{D5, D6\} & \text{kill}(B2) &= \{D1, D3\} \\
gen(B3) &= \{\} & \text{kill}(B3) &= \{\}
\end{align*}
\]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>{D1, D2}</td>
<td>{}</td>
<td>{D1, D2}</td>
<td>{}</td>
</tr>
<tr>
<td>{}</td>
<td>{D3, D4}</td>
<td>{D1, D2}</td>
<td>{D3, D4}</td>
<td>{D1, D2}</td>
</tr>
<tr>
<td>{}</td>
<td>{D5, D6}</td>
<td>{D1, D2}</td>
<td>{D2, D5, D6}</td>
<td>{D1, D2}</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
<td>{D3, D4, D5, D6}</td>
<td>{D3, D4, D5, D6}</td>
<td>{D2, D3, D4, D5, D6}</td>
</tr>
</tbody>
</table>
Algorithm for Reaching Definitions

for each basic block $B$

compute $\text{gen}(B)$ and $\text{kill}(B)$

$out(B) = \{\}$

do {

for each basic block $B$

$\text{in}(B) = U \text{out}(P)$ where $P \in \text{pred}(B)$

$out(B) = \text{gen}(B) U (\text{in}(B) - \text{kill}(B))$

} while $\text{in}(B)$ changes for any basic block $B$
Classwork

- \( \text{in}(B) = \cup \text{out}(P) \) where \( P \) is a predecessor of \( B \)
- \( \text{out}(B) = \text{gen}(B) \cup (\text{in}(B) - \text{kill}(B)) \)

- Initially, \( \text{out}(B) = \{ \} \)

\[
\begin{align*}
\text{gen}(B0) &= \{D1, D2\} & \text{kill}(B0) &= \{D3, D4, D6, D8\} \\
\text{gen}(B1) &= \{D3, D4\} & \text{kill}(B1) &= \{D1, D2, D6, D8\} \\
\text{gen}(B2) &= \{D5, D6\} & \text{kill}(B2) &= \{D2, D3, D7, D8\} \\
\text{gen}(B3) &= \{D7, D8\} & \text{kill}(B3) &= \{D2, D3, D5, D6\}
\end{align*}
\]
## DFA for Reaching Definitions

<table>
<thead>
<tr>
<th>Domain</th>
<th>Sets of definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transfer function</strong></td>
<td>in(B) = U out(P)</td>
</tr>
<tr>
<td></td>
<td>out(B) = gen(B) U (in(B) - kill(B))</td>
</tr>
<tr>
<td><strong>Direction</strong></td>
<td>Forward</td>
</tr>
<tr>
<td><strong>Meet / confluence operator</strong></td>
<td>U</td>
</tr>
<tr>
<td><strong>Initialization</strong></td>
<td>out(B) = { }</td>
</tr>
</tbody>
</table>
DFA for Live Variables

<table>
<thead>
<tr>
<th>Domain</th>
<th>Sets of variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer function</td>
<td>( \text{in}(B) = \text{use}(B) \cup (\text{out}(B) - \text{def}(B)) )</td>
</tr>
<tr>
<td></td>
<td>( \text{out}(B) = \bigcup \text{in}(S) ) where ( S ) is a successor of ( B )</td>
</tr>
<tr>
<td>Direction</td>
<td>Backward</td>
</tr>
<tr>
<td>Meet / confluence operator</td>
<td>( \cup )</td>
</tr>
<tr>
<td>Initialization</td>
<td>( \text{in}(B) = { } )</td>
</tr>
</tbody>
</table>

A variable \( v \) is live at a program point \( p \) if \( v \) is used along some path in the flow graph starting at \( p \). Otherwise, the variable \( v \) is dead.