Lecture 11 – Code Generation

THEORY OF COMPILATION

Eran Yahav

www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec11.pptx
Reference: Dragon8, MCDO 2-4

You are here

Compiler

Source text
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate (IR)
Code Gen.

Executable code

target languages

IR + Symbol Table

Code Gen.

Absolute machine code
Relative machine code
Assembly

From IR to ASM: Challenges

- mapping IR to ASM operations
  - what instruction(s) should be used to implement an IR operation?
  - how do we translate code sequences
- call/return of routines
  - managing activation records
- memory allocation
- register allocation
- optimizations
Intel IA-32 Assembly

- Going from Assembly to Binary...
  - Assembling
  - Linking

- AT&T syntax vs. Intel syntax
- We will use AT&T syntax
  - matches GNU assembler (GAS)

IA-32 Registers

- Eight 32-bit general-purpose registers
  - EAX – accumulator for operands and result data. Used to return value from function calls.
  - EBX – pointer to data. Often use as array-base address
  - ECX – counter for string and loop operations
  - EDX – I/O pointer (GP for us)
  - ESI – GP and source pointer for string operations
  - EDI – GP and destination pointer for string operations
  - EBP – stack frame (base) pointer
  - ESP – stack pointer

- EFLAGS register
- EIP (instruction pointer) register
- Six 16-bit segment registers
- ... (ignore the rest for our purposes)

Not all registers are born equal

- EAX
  - Required operand of MUL, IMUL, DIV and IDIV instructions
  - Contains the result of these operations
- EDX
  - Stores remainder of a DIV or IDIV instruction (EAX stores quotient)
- ESI, EDI
  - ESI – required source pointer for string instructions
  - EDI – required destination pointer for string instructions
- Destination Registers of Arithmetic operations
  - EAX, EBX, ECX, EDX
- EBP – stack frame (base) pointer
- ESP – stack pointer

IA-32 Addressing Modes

- Machine-instructions take zero or more operands
- Source operand
  - Immediate
  - Register
  - Memory location
    - (I/O port)
- Destination operand
  - Register
  - Memory location
    - (I/O port)
Immediate and Register Operands

- Immediate
  - Value specified in the instruction itself
  - GAS syntax – immediate values preceded by $
    - add $4, %esp
- Register
  - Register name is used
  - GAS syntax – register names preceded with %
    - mov %esp, %ebp

Memory and Base Displacement Operands

- Memory operands
  - Value at given address
  - GAS syntax - parentheses
    - mov (%eax), %eax
- Base displacement
  - Value at computed address
  - Address computed out of
    - base register, index register, scale factor, displacement
    - offset = base + (index*scale) + displacement
    - Syntax: disp(base,index, scale)
    - movl $42, $2(%eax)
    - movl $42, $1(%eax,%ecx,4)

Base Displacement Addressing

Array Base Reference

\[ \text{Mov} \%ecx, \%ebx, 4 \]
\[ \%ecx = \text{base} \]
\[ \%ebx = 4 \]
\[ \text{offset} = \text{base} + (\text{index} \times \text{scale}) + \text{displacement} \]
\[ \text{offset} = \text{base} + (3 \times 4) + 0 = \text{base} + 12 \]

How do we generate the code?

- break the IR into basic blocks
- basic block is a sequence of instructions with
  - single entry (to first instruction), no jumps to the middle of the block
  - single exit (last instruction)
  - code execute as a sequence from first instruction to last instruction without any jumps
- edge from one basic block B1 to another block B2 when the last statement of B1 may jump to B2
creating basic blocks

- Input: A sequence of three-address statements
- Output: A list of basic blocks with each three-address statement in exactly one block
- Method
  - Determine the set of leaders
    - The first statement is a leader
    - Any statement that is the target of a conditional or unconditional jump is a leader
    - Any statement that immediately follows a goto or conditional jump statement is a leader
  - For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

control flow graph

- A directed graph $G=(V,E)$
- nodes $V =$ basic blocks
- edges $E =$ control flow
  - $(B_1, B_2) \in E$ when control from $B_1$ flows to $B_2$

example

```plaintext
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i,j] = 0.0;
  for i from 1 to 10 do
    a[i,i] = 1.0;
end
```
Variable Liveness

- A statement $x = y + z$
  - defines $x$
  - uses $y$ and $z$
- A variable $x$ is live at a program point if its value is used at a later point

```
y = 42
z = 73
x = y + z
print(x);
```

(x showing state after the statement)

Computing Liveness Information

- between basic blocks – dataflow analysis (next lecture)
- within a single basic block?
  - idea
    - use symbol table to record next-use information
    - scan basic block backwards
    - update next-use for each variable

Computing Liveness Information

- INPUT: A basic block $B$ of three-address statements.
  symbol table initially shows all non-temporary variables in $B$ as being live on exit.

- OUTPUT: At each statement $i: x = y + z$ in $B$, liveness and next-use information of $x$, $y$, and $z$ at $i$.

- Start at the last statement in $B$ and scan backwards
  - At each statement $i: x = y + z$ in $B$, we do the following:
    1. Attach to $i$ the information currently found in the symbol table regarding the next use and liveness of $x$, $y$, and $z$.
    2. In the symbol table, set $x$ to "not live" and "no next use."
    3. In the symbol table, set $y$ and $z$ to "live" and the next uses of $y$ and $z to $i$

```
x = 1
y = x * 3
z = x + 3
x = x * z
```
can we change the order between $z$ and $3$?
common-subexpression elimination

- common-subexpression elimination

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

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

DAG Representation of Basic Blocks

algebraic identities

\[
\begin{align*}
a &= x^2 \\
b &= x^2 \\
c &= x/2 \\
d &= x
\end{align*}
\]

\[
\begin{align*}
a &= x^2 \\
b &= x+x \\
c &= x^{0.5} \\
d &= x
\end{align*}
\]
coming up next

- register allocation

The End