You are here

Source text

Compiler


Executable code

txt

exe
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

\[
\text{Array Base Reference: } (\%ecx, \%ebx, 4)
\]

\[
\begin{array}{cccccccc}
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
7 & 0 & 2 & 4 & 5 & 6 & 7 & 1 \\
\end{array}
\]

\[
\text{Mov (}\%ecx, \%ebx, 4\text{), } \%eax \\
\%ecx = \text{base} \\
\%ebx = 3
\]

offset = base + (index*scale) + displacement

offset = base + (3*4) + 0 = 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 $B_1$ to another block $B_2$ when the last statement of $B_1$ may jump to $B_2$
Example

B1
\[
\begin{align*}
t_1 &:= 4 \times i \\
t_2 &:= a[t_1] \\
\text{if } t_2 &\leq 20 \text{ goto } B_3
\end{align*}
\]

B2
\[
\begin{align*}
t_3 &:= 4 \times i \\
t_4 &:= b[t_3] \\
\text{goto } B_4
\end{align*}
\]

B3
\[
\begin{align*}
t_5 &:= t_2 \times t_4 \\
t_6 &:= \text{prod} + t_5 \\
\text{prod} &:= t_6 \\
\text{goto } B_4
\end{align*}
\]

B4
\[
\begin{align*}
t_7 &:= i + 1 \\
i &:= t_2 \\
\text{Goto } B_5
\end{align*}
\]
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 (first statement of a block)
    - 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 $B_2$

```plaintext
B_1
prod := 0
i := 1

B_2

\[
\begin{align*}
t_1 &:= 4 \times i \\
t_2 &:= a\[t_1]\ \\
t_3 &:= 4 \times i \\
t_4 &:= b\[t_3]\ \\
t_5 &:= t_2 \times t_4 \\
t_6 &:= prod \times t_5 \\
prod &:= t_6 \\
t_7 &:= i + 1 \\
i &:= t_7 \\
\text{if } i \leq 20 \text{ goto } B_2
\end{align*}
\]```
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;
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

<table>
<thead>
<tr>
<th>y = 42</th>
<th>x undefined, y live, z undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>z = 73</td>
<td>x undefined, y live, z live</td>
</tr>
<tr>
<td>$x = y + z$</td>
<td>x is live, y dead, z dead</td>
</tr>
<tr>
<td>print($x$);</td>
<td>x is dead, y dead, z dead</td>
</tr>
</tbody>
</table>

(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.
Computing Liveness Information

- 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

\[
\begin{align*}
x &= 1 \\
y &= x + 3 \\
z &= x \times 3 \\
x &= x \times z
\end{align*}
\]

can we change the order between 2 and 3?
common-subexpression elimination

- common-subexpression elimination

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

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= b
\end{align*}
\]
DAG Representation of Basic Blocks

a = b + c
b = a - d
c = b + c
d = a - d
DAG Representation of Basic Blocks

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

\[
\begin{align*}
  a &= x^2 \\
  b &= x^2 \\
  c &= x/2 \\
  d &= 1\times x
\end{align*}
\]
coming up next

- register allocation
The End