Generation of Target Code

Chapter 8, Section 8.1, 8.2, 8.3, 8.6, 8.8
Generating Target Code

• Input:
  – IR from the front end (e.g., three-address code)
  – Symbol table information
  – Results from control-flow and dataflow analyses of the IR (e.g., basic blocks, live variables, etc.)
  – The IR may have already been optimized with machine-independent optimizations (e.g., common subexpression elimination, code motion, etc.)

• Output:
  – Assembly code for the target machine (to be processed later by an assembler, to replace mnemonics such as ADD with actual binary opcodes)
  – Or, directly produce relocatable object code for the target machine
Outline

• Part 1: Example of a simple instruction set for the target machine
• Part 2: Mapping between compile-time names and run-time addresses
• Part 3: Code generation for a basic block
  – Issues of register allocation
• Part 4: Effective use of registers
Part 1: Target Machine Instruction Set

- **RISC (reduced instruction set computer)**
  - Many registers, three-address instructions, relatively simple/few instructions (uniform, fixed-length)
  - Alpha, ARM, MIPS, PA-RISC, PowerPC, SPARC

- **CISC (complex instruction set computer)**
  - Few registers, two-address instructions, different register classes, large number of instructions with variable length
  - System/360, PDP-11, VAX, 68000, x86

- **Stack-based machine**
  - Operands on top of a stack; pop them, perform the operation, push the result on the stack
  - JVM (a virtual machine, not a real machine)
Our Discussion: Artificial Instruction Set

- Byte-addressable machine with general-purpose registers R1, R2, ... and a limited instruction set
  - All operands have integer type
  - Each instruction can be labeled (essentially, by the address of this instruction in memory, which can be the target of jumps)
  - Loads, stores, computations, jumps

- Example: load operations
  - **LD r, addr**: load the contents of location `addr` into register `r`
    - Memory-to-register: **LD R1, x** where `x` is based on some addressing mode
    - Register-to-register: **LD R2, R1**
Addressing Modes

• **Direct**: just use a variable name `y` from the IR
  – In reality, it refers to the memory location reserved for `y`: at compile time, we must decide how to map `y` to a particular memory address (more later)
  – E.g., the actual instruction could be `LD R1, 0x1FB4`

• **Indexed**: `a(r)`, `a` is a variable, `r` is a register
  – Consider the address of `a`
  – The accessed location is at an offset from this address; the value of the offset is in `r`
  – E.g., `LD R1, a(R2)` is `R1 = contents(a + contents(R2))`
  – Useful for accessing arrays: `a` is the base address of the array (i.e., the address of the first element), and `r` contains the offset
Addressing Modes

• Indexed with address constant: \texttt{const(r)}
  – \texttt{LD R1, 100(R2)} is \( R1 = \text{contents}(100 + \text{contents}(R2)) \)

• Immediate: \texttt{#const}
  – There is no memory location to be accessed
  – E.g., \texttt{LD R1, #100} is \( R1 = 100 \)
  – E.g., \texttt{ADD R1, R1, #100} increments \( R1 \) by 100

• Addressing modes are very specific to the hardware architecture, so in this course we will stay away from specific details
Artificial Instruction Set

• Load operations **LD r, addr**: load the contents of location **addr** into register **r**

• Store operations **ST dest, r**: store the contents of register **r** into location **dest**

• Computation operations
  – **OP r, src1, src2**: binary operators
  – **OP r, src**: unary operators

• Jumps
  – **BR L**: unconditional; L is really the address of the target instruction in the code layout in memory
  – **Bcond r, L**: conditional, depending on the value in register **r**: e.g., **BLTZ R1, L23** jumps if contents(R1) < 0
Examples of Target Code

• Three-address code: \( x = y - z \)
  
  LD R1, y  
  LD R2, z  
  SUB R1, R1, R2  
  ST x, R1  
  
  If \( y \) or \( z \) are already in registers, we can avoid LD  
  If later \( x \) is used only for operations involving registers, but not for anything else, we can avoid ST

• Three-address code: \( b = a[i] \), where an array element is 8 bytes  
  
  LD R1, i  
  MUL R1, R1, #8  
  LD R2, a(R1)  
  ST b, R2
Examples of Target Code

• Three-address code: \( a[j] = c \)
  
  \[
  \begin{align*}
  &\text{LD } R1, \ c \\
  &\text{LD } R2, \ j \\
  &\text{MUL } R2, \ R2, \ #8 \\
  &\text{ST } a(R2), \ R1
  \end{align*}
  \]

• Three-address code: \( x = \ast p \) (this is with pointers, we have not discussed them earlier)
  
  \[
  \begin{align*}
  &\text{LD } R1, \ p \\
  &\text{LD } R2, \ 0(R1) \quad \text{// } R2 = \text{contents}(0 + \text{contents}(R1)) \\
  &\text{ST } x, \ R2
  \end{align*}
  \]

• Three-address code: \( \ast p = y \)
  
  \[
  \begin{align*}
  &\text{LD } R1, \ p \\
  &\text{LD } R2, \ y \\
  &\text{ST } 0(R1), \ R2
  \end{align*}
  \]
Examples of Target Code

• Three-address code: if (x<y) goto L
  
  LD R1, x
  LD R2, y
  SUB R1, R1, R2
  BLTZ R1, M
  
  – M is the address of the first machine instruction that was generated from the translation of the three-address instructions with label L

• In all of these examples, we would really like to have the operands already in registers, in order to avoid LD and ST instructions
  – More on this later
Part 2: Addresses for Names

• Eventually, at run time, the memory will be:
  – Code segment: the sequence of instructions
  – Static segment: memory locations for global variables
    • They exist for the lifetime of the entire program
  – Stack segment: local variables, in stack frames
  – Heap segment: dynamically-allocated memory

• **Static allocation**: the compiler chooses the address at compile time: e.g. three-address instruction \( y=7 \) leads to \( \text{ST } y, \#7 \) which really is \( \text{ST } 0x1FB4, \#7 \)

• **Stack allocation**: assume a specialized register \( \text{SP} \) whose contents is the address of the start of the current stack frame in the stack segment
Addresses for Names

- **Stack allocation** (cont’d): the relative offset of a local variable from the start of the frame is decided at compile time (i.e., frame layout for all locals)
  - When we write `LD R1, y` we mean `LD R1, offset_y(SP)`
    - Recall that indexed addressing `const(r)` here means `contents(const + contents(r))`
- Initial value of SP: defined by `main` (or by loader)
  - `LD SP, #600` – start the stack segment from address 600
- Key issue: in the generated code, we need to update SP immediately before/after calls
Calls

• Before a call:
  – Increment the stack pointer SP: ADD SP, SP, #68
    • Here 68 is the size of the stack frame of the caller, which is determined at compile time
  – In the callee’s frame, remember the return address (the address of the caller’s instruction that immediately follows the jump instruction; details omitted)
  – Jump: BR 300 (the callee’s 1st instr is at address 300)

• After a call:
  – Restore the stack pointer SP: SUB SP, SP, #68
  – Use the remembered return address as the target of a jump back into the caller
Part 3: Code Generation for a Basic Block

• For illustration, a simple code generator for a given basic block (Section 8.6)
  – Keeps track of which values are in which registers, so that we can avoid generating unnecessary LD and ST
  – No promises of optimality or quality

• Let’s simplify the instruction set even more ...
  – LD $r$, mem: load from memory to register
  – ST mem, $r$: store from register to memory
  – OP $r$, $r$, $r$: all operations are on registers

• Go through the sequence of three-address instructions (in order) and generate code
  – Use registers when possible
Bookkeeping Information

• For each register: a register descriptor
  – A set of variable names from the three-address code
  – For each variable in the set, the register contains the current (“latest”) value of the variable
  – At the start of the basic block, the descriptor is empty

• For each variable from the three-address code: an address descriptor
  – A set of locations where the current value of the variable can be found
  – Contains zero or more registers, and possibly the actual memory location for this variable
Example

• At the beginning of the basic block

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
</tbody>
</table>

• First three-address instruction: \( t = a - b \)
  – Suppose we select R1 for a, R2 for b, R2 for t
  – des(R1) does not contain a: issue LD R1, a and then update des(R1) = \{ a \} and des(a) = \{ a, R1 \}
  – des(R2) does not contain b: issue LD R2, b and then update des(R2) = \{ b \} and des(b) = \{ b, R2 \}
  – Issue SUB R2, R1, R2 and then update des(R2) = \{ t \} and des(t) = \{ R2 \} and des(b) = \{ b \}

• Need to remove t from des(t) and R2 from des(b)
Example

- Second three-address instruction: \( u = a - c \)
  - Suppose we select R1 for a, R3 for c, R1 for u
  - \( \text{des}(R1) \) contains a: no need to issue \textbf{LD R1, a}
  - \( \text{des}(R3) \) does **not** contain c: issue \textbf{LD R3, c} and then update \( \text{des}(R3) = \{ c \} \) and \( \text{des}(c) = \{ c, R3 \} \)
  - Issue \textbf{SUB R1, R1, R3} and then update \( \text{des}(R1) = \{ u \} \) and \( \text{des}(u) = \{ R1 \} \) and \( \text{des}(a) = \{ a \} \)

- Need to remove u from \( \text{des}(u) \) and R1 from \( \text{des}(a) \)
Example

- Third three-address instruction: $v = t + u$
  - Suppose we select R2 for t, R1 for u, R3 for v
  - des(R2) contains t: no need to issue LD R2, t
  - des(R1) contains u: no need to issue LD R1, u
  - Issue ADD R3, R2, R1 and then update des(R3) = { v } and des(v) = { R3 } and des(c) = { c }
- Need to remove v from des(v) and R3 from des(c)
Code Generation – Case 1

• Three-address operation \( x_1 = x_2 \text{ op } x_3 \)
  – Select registers R1, R2, R3 for \( x_1, x_2, x_3 \) respectively
  – If des(R2) does not contain \( x_2 \): issue \( LD \ R2, x_2 \) and update the descriptors:
    • Set des(R2) to contain only \( x_2 \)
    • Add R2 to des(x2)
    • Remove R2 from any other address descriptor
  – Same for R3 and \( x_3 \)
  – Issue \( OP \ R1, R2, R3 \) and update the descriptors:
    • Set des(R1) to contain only \( x_1 \)
    • Set des(x1) to contain only R1
    – Note: will not contain the memory location for \( x_1 \)
    • Remove R1 from any other address descriptor
Code Generation – Case 2

• Three-address copy \( x_1 = x_2 \)
  – Select the same \( R \) for both
  – If \( \text{des}(R) \) does not contain \( x_2 \): issue \( \text{LD} \ R, \ x_2 \) and update the descriptors:
    • Set \( \text{des}(R) \) to contain only \( x_2 \), add \( R \) to \( \text{des}(x_2) \), remove \( R \) from any other address descriptor
  – More updates:
    • Add \( x_1 \) to \( \text{des}(R) \)
    • Set \( \text{des}(x_1) \) to contain only \( R \)

\[
\begin{array}{cccccccccccc}
R1 & R2 & R3 & a & b & c & d & t & u & v \\
\hline
u & t & v & a & b & c & d & R2 & R1 & R3 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
R1 & R2 & R3 & a & b & c & d & t & u & v \\
\hline
u & a,d & v & R2 & b & c & d,R2 & R1 & R3 \\
\end{array}
\]
Code Generation – Case 3

• End of the basic block
  – Consider every variable x that is live at the exit of the basic block; if we don’t have liveness analysis information, assume all variables are live
  – If des(x) does **not** contain x, the “latest” value of x is not yet in memory but still in some register: need to issue `ST x, R` where R is some register in des(x)

<table>
<thead>
<tr>
<th>exit</th>
<th>ST a, R2</th>
<th>ST d, R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>v</td>
</tr>
<tr>
<td>R2</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>v</td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• For this specific example, assume that t, u, and v are not used outside of the block (e.g., they are temps)
\begin{align*}
t &= a - b \\
\text{LD R1, } a & \\
\text{LD R2, } b & \\
\text{SUB R2, R1, R1} & \\
\text{u} &= a - c \\
\text{LD R3, c} & \\
\text{SUB R1, R1, R3} & \\
\text{v} &= t + u \\
\text{ADD R3, R2, R1} & \\
\text{a} &= d \\
\text{LD R2, d} & \\
\text{d} &= v + u \\
\text{ADD R1, R3, R1} & \\
\text{exit} & \\
\text{ST a, R2} & \\
\text{ST d, R1} & \\
\end{align*}
Heuristics for Selecting Registers

• For example, how do we select a register for x2 in \( x_1 = x_2 \text{ op } x_3 \)?
  – If x2 is currently in some R, as shown by \( \text{des}(x_2) \), select that R for x2 (no need to load)
  – Otherwise: consider each R and compute its “weight” as the number of spill instructions it requires; selects one R that requires a small number of spills

• For any v in \( \text{des}(R) \), spill it back to memory?
  – If \( \text{des}(v) \) has elements other than R: no spill
  – If v is not live after this instruction: no spill
  – Otherwise, spill v: issue \( \text{ST } v, R \) to write the value of v back to memory; then add v to \( \text{des}(v) \)
Global Register Allocation

• This was a simplified local approach: considers only the code inside a basic block
  – All live variables are stored back to memory ("spilled") at the end of the basic block

• **Global** register allocation: across the boundaries of basic blocks (inside the same procedure)

• Heavily investigated topic
  – Very important for performance: much more efficient to do operations on registers, and to avoid going to memory as much as possible
  – Register allocation via **graph coloring** [Chaitin et al. 1981]
  – A large number of other approaches
Main Ideas [Chaitin et al. 1981]

• Define **live ranges** in the three-address code
  – Live range for a variable $v$: the set of CFG nodes $n$ such that upon entry into $n$, $v$ is initialized and $v$ is live
  – Two ranges **interfere** if they have common CFG nodes

• Interference graph
  – Nodes are live ranges; edges represent interference
  – **Graph coloring**: each node has a color; if two nodes share an edge, they must have different colors

• Given R registers: try to color with R colors; if not successful, delete a node (do not assign it to a register at all) and try coloring again
  – Graph coloring with min number of colors is NP-complete

• Alternative: Poletto & Sarkar 1999, linear time (web page)
Part 4: Register Pressure and Optimizations

- **Register pressure**: need more registers than available, and thus need to spill a lot
  - Problem: other optimizations may increase it
  - Example: loop unrolling

    ```c
    for ( i = 0 ; i < 4096 ; i++ )
    c[i] = a[i] + b[i];
    ```

    ```c
    for ( i = 0 ; i < 4095 ; i +=2 ) {
        c[i] = a[i] + b[i];
       c[i+1] = a[i+1] + b[i+1];
    } // unroll factor of 2
    ```

  - Reduces the “control overhead” of the loop: makes the loop exit test (i < 4096) less frequently
  - Hardware advantages: instruction-level parallelism; fewer pipeline stalls
  - Problem: high unroll factors may degrade performance due to register pressure and spills
Register Pressure Study

• Thanks to Albert Hartono for these examples
  – trac.mcs.anl.gov/projects/performance/wiki/Orio

• Unrolling for matrix-vector multiplication

```c
for ( i=0; i<=N-1; i++ ) {
    for ( j=0; j<=N-1; j++ ) {
        y[i] = y[i] + A[i][j]*x[j];
    }
}

Unroll the j loop by an unroll factor of 4

```c
for ( i=0; i<=N-1; i++ ) {
    for ( j=0; j<=N-4; j=j+4 ) {
        y[i]=y[i]+A[i][j]*x[j];
        y[i]=y[i]+A[i][j+1]*x[j+1];
        y[i]=y[i]+A[i][j+2]*x[j+2];
        y[i]=y[i]+A[i][j+3]*x[j+3];
    }
    for ( ; j<=N-1; j=j+1 ) // if N%4 != 0
        y[i]=y[i]+A[i][j]*x[j];
}
```
Another Unrolled Version

Unroll the i loop by a factor of 4

```cpp
for ( i=0; i<=N-1; i=i+4 ) {
    for ( j=0; j<=N-1; j=j++ ) {
        y[i]=y[i]+A[i][j]*x[j];
    }
    for ( j=0; j<=N-1; j=j++ ) {
        y[i+1]=y[i+1]+A[i+1][j]*x[j];
    }
    for ( j=0; j<=N-1; j=j++ ) {
        y[i+2]=y[i+2]+A[i+2][j]*x[j];
    }
    for ( j=0; j<=N-1; j=j++ ) {
        y[i+3]=y[i+3]+A[i+3][j]*x[j];
    }
} // assume N%4 == 0
```

Fuse the j loops (unroll-and-jam)

```cpp
for ( i=0; i<=N-4; i=i+4 ) {
    for ( j=0; j<=N-1; j++ ) {
        y[i]=y[i]+A[i][j]*x[j];
        y[i+1]=y[i+1]+A[i+1][j]*x[j];
        y[i+2]=y[i+2]+A[i+2][j]*x[j];
        y[i+3]=y[i+3]+A[i+3][j]*x[j];
    }
}
```
Unroll the i loop by a factor of 2 and the j loop by a factor of 2 and then fuse the j loops

```c
for ( i=0; i<=N-2; i=i+2 ) {
    for ( j=0; j<=N-2; j=j+2 ) {
        y[i]=y[i]+A[i][j]*x[j];
        y[i]=y[i]+A[i][j+1]*x[j+1];
        y[i+1]=y[i+1]+A[i+1][j]*x[j];
        y[i+1]=y[i+1]+A[i+1][j+1]*x[j+1];
    }
}
```
Scalar Replacement
Replace array references with scalars

```c
for ( i=0; i<=N-2; i=i+2 ) {
    double scv_3, scv_4;
    scv_3 = y[i]; scv_4 = y[i+1];
    for ( j=0; j<=N-2; j=j+2 ) {
        double scv_1, scv_2;
        scv_1 = x[j]; scv_2 = x[j+1];
        scv_3 = scv_3 + A[i][j] * scv_1;
        scv_3 = scv_3 + A[i][j+1] * scv_2;
        scv_4 = scv_4 + A[i+1][j] * scv_1;
        scv_4 = scv_4 + A[i+1][j+1] * scv_2;
    }
    y[i] = scv_3; y[i+1] = scv_4;
}
```
Experimental Setup

• N=10000
• Scalar replacement used in all experiments
• gcc 4.2.4 with -O3 optimization flag
• Multi-core Intel Xeon workstation
  – dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB), 32 KB L1 cache, 12 MB of L2 cache, 16 GB of DDR2 RAM
• All combination of unroll factors for i and j: values of 1, 2, 3, ..., 32 (total: 1024 versions)
• The Orio tool determined that unroll factor 10 for i and unroll factor 1 for j is the best
Unroll j only

Execution time (secs)

Unroll factors (of loop j)
Unroll i only

best in all experiments
Unroll both i and j; j is factor 2