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
Part 1: Target Machine Instruction Set

• RISC (reduced instruction set computer)
  – Many registers, three-address instructions, simple addressing modes, relatively simple and relatively few instructions (uniform, fixed-length)
  – Alpha, ARM, MIPS, PA-RISC, PowerPC, SPARC

• CISC (complex instruction set computer)
  – Few registers, two-address instructions, a variety of addressing modes, 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 runtime address of this instruction in memory, which can be the target of jumps)
  – Loads, stores, computations, jumps

• Example: load operations
  – \textbf{LD dest, addr}: load the contents of location \textit{addr} into location \textit{dest}
  – Memory-to-register: \textbf{LD R1, x}
    • \textit{x} is based on some addressing mode
  – Register-to-register: \textbf{LD R2, R1}
Addressing Modes (1/2)

• **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 is `LD R1, 0x1FB4`

• **Indexed**: \( a(r) \), \( a \) is a variable, \( r \) is a register
  – Consider the address of \( a \) (i.e., the l-value 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 \( R1 = \text{contents}(a + \text{contents}(R2)) \)
  – 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 (2/2)

• 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 dest, addr**: load the contents of location `addr` into location `dest`
• Store operations **ST dest, r**: store the contents of register `r` into location `dest`
• Computation operations
  – **OP dest, src1, src2**: binary operators
  – **OP dest, src**: unary operators
• Jumps
  – **BR L**: unconditional; L is really the address of the target instruction in the code layout
  – **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: \( x = y - z \)
  
  \[
  \text{LD } R1, y \\
  \text{LD } R2, z \\
  \text{SUB } R1, R1, R2 \\
  \text{ST } x, R1
  \]

  – 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
  
  \[
  \text{LD } R1, i \\
  \text{MUL } R1, R1, \#8 \\
  \text{LD } R2, a(R1) \\
  \text{ST } b, R2
  \]
Examples of Target Code

- **Three-address code: \( a[j] = c \)**
  
  ```
  LD R1, c
  LD R2, j
  MUL R2, R2, #8
  ST a(R2), R1
  ```

- **Three-address code: \( x = *p \)** (this is with pointers, we have not discussed them earlier)
  
  ```
  LD R1, p
  LD R2, 0(R1)  // R2 = contents(0 + contents(R1))
  ST x, R2
  ```

- **Three-address code: \( *p = y \)**
  
  ```
  LD R1, p
  LD R2, y
  ST 0(R1), R2
  ```
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 (1/2)

• 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 \( ST y, #7 \) which really is \( ST 0x1FB4, #7 \)
• Stack allocation: assume a specialized register \( SP \) whose contents is the address of the start of the current stack frame in the stack segment
Addresses for Names (2/2)

- 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: \texttt{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: \texttt{SUB SP, SP, #68}
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

• Very simplified instruction set
  – \texttt{LD reg, mem}: load from memory to register
  – \texttt{ST mem, reg}: store from register to memory
  – \texttt{OP reg, reg, reg}: 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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• First three-address instruction: \( t = a - b \)
  – Select R1 for \( a \), R2 for \( b \), R2 for \( t \) (more on this later …)
  – \( \text{des}(R1) \) does not contain \( a \): issue \( \text{LD R1, a} \) and then update \( \text{des}(R1) = \{ a \} \) and \( \text{des}(a) = \{ a, R1 \} \)
  – \( \text{des}(R2) \) does not contain \( b \): issue \( \text{LD R2, b} \) and then update \( \text{des}(R2) = \{ b \} \) and \( \text{des}(b) = \{ b, R2 \} \)
  – Issue \( \text{SUB R2, R1, R2} \) and then update \( \text{des}(R2) = \{ t \} \) and \( \text{des}(t) = \{ R2 \} \) and \( \text{des}(b) = \{ b \} \)

• Need to remove \( t \) from \( \text{des}(t) \) and R2 from \( \text{des}(b) \)
Example

<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>a</td>
<td>t</td>
<td></td>
<td>a,R1</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>R2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

• Three-address operation \( x_1 = x_2 \text{ op } x_3 \)
  – Select registers R1, R2, R3 (more details later ...)
  – If des(R2) does **not** contain \( x_2 \): issue LD R2, x2 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 (more details later ...)
  − If des($R$) does **not** contain $x_2$: issue $\text{LD } R, x_2$ and update the descriptors:
    • Set des($R$) to contain only $x_2$, add $R$ to des($x_2$), remove $R$ from any other address descriptor
  − More updates:
    • Add $x_1$ to des($R$)
    • Set des($x_1$) to contain only $R$

$$a = d$$
$$\text{LD } R2, d$$
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 \( \text{des}(x) \) does not contain \( x \), the “latest” value of \( x \) is not yet in memory but still in some register: need to issue \( \text{ST} \ x, \ R \) where \( R \) is some register in \( \text{des}(x) \)

<table>
<thead>
<tr>
<th>exit</th>
<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>d</td>
<td>a</td>
<td>v</td>
<td>R2</td>
<td>b</td>
<td>c</td>
<td>R1</td>
<td></td>
<td></td>
<td>R3</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)
Heuristics for Selecting Registers

• We will not discuss a precise algorithm, just some ideas

• Three-address operation $\mathbf{x_1 = x_2 \text{ op } x_3}$: choose $R_2$
  – If $x_2$ is currently in some $R$, as indicated by $\text{des}(x_2)$, select that $R$
  – If there is some $R$ with an empty $\text{des}(R)$, select $R$
  – Consider each $R$ and compute its “weight” as the number of spill instructions it requires; selects one $R$ whose $\text{des}(R)$ 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 $x_1$ and $v$ is not $x_3$: 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 and then update $\text{des}(v)$ to add $v$
Heuristics for Selecting Registers (cont’d)

• Given a three-address operation \( x_1 = x_2 \text{ op } x_3 \)
  – Choose a register R2 for x2 (spill if necessary)
  – Choose a register R3 for x3 (spill if necessary)
  – Choose a register R1 for x1
    • If x1 is currently in a register R, as indicated by des(x1), select R but only if des(R) contains only x1
    • If there is some R with an empty des(R), select R
    • If x2 is not live after this instruction, and des(R2) has only x2 (after LD R2, x2 if necessary): select R2
      – Else, same check for x3 and R3
    • Selects R with minimum spills and issue them

• Given a three-address operation \( x_1 = x_2 \)
  – Select R2 for x2, as shown earlier, and also use it for x1
<table>
<thead>
<tr>
<th>Expression</th>
<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>t = a − b</td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R1, a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R2, b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R2, R1, R1</td>
<td>a</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u = a − c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R3, c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R1, R1, R3</td>
<td>u</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v = t + u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>u</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R2, d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d = v + u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R3, R1</td>
<td>d</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST a, R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST d, R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 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
(detour) 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: make 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

```c
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)

```c
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 Both Loops

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
Unroll i only

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