

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 run-time address of this instruction in memory, which can be the target of jumps)
 - Loads, stores, computations, jumps
- Example: load operations
 - **LD dest, addr**: load the contents of location **addr** into location **dest**
 - Memory-to-register: **LD R1, x**
 - **x** is based on some **addressing mode**
 - Register-to-register: **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: **const(r)**
 - **LD R1, 100(R2)** is $R1 = \text{contents}(100 + \text{contents}(R2))$
- Immediate: **#const**
 - There is no memory location to be accessed
 - E.g., **LD R1, #100** is $R1 = 100$
 - E.g., **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 $\text{contents}(R1) < 0$

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: $a[j] = c$

```
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 $\text{contents}(\text{const} + \text{contents}(r))$
- 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**

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
 - **LD reg, mem**: load from memory to register
 - **ST mem, reg**: store from register to memory
 - **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

R1	R2	R3	a	b	c	d	t	u	v
			a	b	c	d			

- First three-address instruction: **t = a - b**

- Select R1 for a, R2 for b, R2 for t (more on this later ...)
- 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)

R1	R2	R3	a	b	c	d	t	u	v
a	t		a,R1	b	c	d	R2		

Example

R1	R2	R3	a	b	c	d	t	u	v
a	t		a,R1	b	c	d	R2		

- Second three-address instruction: **u = a - c**
 - Select R1 for a, R3 for c, R1 for u
 - des(R1) contains a: no need to issue **LD R1, a**
 - des(R3) does **not** contain c: issue **LD R3, c** and then update **des(R3) = { c }** and **des(c) = { c, R3 }**
 - Issue **SUB R1, R1, R3** and then update **des(R1) = { u }** and **des(u) = { R1 }** and **des(a) = { a }**
 - Need to remove u from des(u) and R1 from des(a)

R1	R2	R3	a	b	c	d	t	u	v
u	t	c	a	b	c,R3	d	R2	R1	

Example

R1	R2	R3	a	b	c	d	t	u	v
u	t	c	a	b	c,R3	d	R2	R1	

- Third three-address instruction: **$v = t + u$**
 - 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)

R1	R2	R3	a	b	c	d	t	u	v
u	t	v	a	b	c	d	R2	R1	R3

Code Generation – Case 1

- Three-address operation $x1 = x2 \text{ op } x3$
 - Select registers R1, R2, R3 (more details later ...)
 - If $\text{des}(R2)$ does **not** contain $x2$: issue **LD R2, x2** and update the descriptors:
 - Set $\text{des}(R2)$ to contain only $x2$
 - Add R2 to $\text{des}(x2)$
 - Remove R2 from any other address descriptor
 - Same for R3 and $x3$
 - Issue **OP R1, R2, R3** and update the descriptors:
 - Set $\text{des}(R1)$ to contain only $x1$
 - Set $\text{des}(x1)$ to contain only R1
 - Note: will not contain the memory location for $x1$
 - Remove R1 from any other address descriptor

Code Generation – Case 2

- Three-address copy $x1 = x2$
 - Select the same R for both (more details later ...)
 - If $\text{des}(R)$ does **not** contain $x2$: issue **LD R, x2** and update the descriptors:
 - Set $\text{des}(R)$ to contain only $x2$, add R to $\text{des}(x2)$, remove R from any other address descriptor
 - More updates:
 - Add $x1$ to $\text{des}(R)$
 - Set $\text{des}(x1)$ to contain only R

$a = d$

R1	R2	R3	a	b	c	d	t	u	v
u	t	v	a	b	c	d	R2	R1	R3

LD R2, d

R1	R2	R3	a	b	c	d	t	u	v
u	a,d	v	R2	b	c	d,R2		R1	R3

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 **ST x , R** where R is some register in $\text{des}(x)$

	R1	R2	R3	a	b	c	d	t	u	v
exit	d	a	v	R2	b	c	R1			R3

ST a, R2
ST d, R1

- For this specific example, assume that t , u , and v are not used outside of the block (e.g. they are temps)

Selecting the Registers to Use

- Three-address operation $x1 = x2 \text{ op } x3$: choose R2
 - If $x2$ is currently in some R, as indicated by $\text{des}(x2)$, select that R
 - Else, if there is some R with an empty $\text{des}(R)$, select R
 - Else, consider each R and compute its “weight” as the number of **spill instructions** it requires; selects one R with 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 $x1$ and v is not $x3$: no spill
 - If v is not live after this instruction: no spill
 - Otherwise, spill v : issue **ST v , R** to write the value of v back to memory and then update $\text{des}(v)$ to add v

Selecting the Registers to Use (cont'd)

- Given a three-address operation $x1 = x2 \text{ op } x3$
 - 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 $\text{des}(x1)$, select R but only if $\text{des}(R)$ contains **only** x1
 - Else, if there is some R with an empty $\text{des}(R)$, select R
 - Else, if x2 is not live after this instruction, and $\text{des}(R2)$ has only x2 (after LD R2, x2 if necessary): select R2
 - Else, same check for x3 and R3
 - Else, selects R with minimum spills and issue them
- Given a three-address operation $x1 = x2$
 - Select R2 for x2, as shown earlier, and also use it for x1

t = a - b

LD R1, a

LD R2, b

SUB R2, R1, R1

R1	R2	R3	a	b	c	d	t	u	v
			a	b	c	d			

u = a - c

LD R3, c

SUB R1, R1, R3

R1	R2	R3	a	b	c	d	t	u	v
a	t		a,R1	b	c	d	R2		

v = t + u

ADD R3, R2, R1

R1	R2	R3	a	b	c	d	t	u	v
u	t	c	a	b	c,R3	d	R2	R1	

a = d

LD R2, d

R1	R2	R3	a	b	c	d	t	u	v
u	t	v	a	b	c	d	R2	R1	R3

d = v + u

ADD R1, R3, R1

R1	R2	R3	a	b	c	d	t	u	v
u	a,d	v	R2	b	c	d,R2		R1	R3

exit

ST a, R2

ST d, R1

R1	R2	R3	a	b	c	d	t	u	v
d	a	v	R2	b	c	R1			R3

R1	R2	R3	a	b	c	d	t	u	v
d	a	v	a,R2	b	c	d,R1			R3

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

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



```
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

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



```
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];  
}
```

Unroll the j loop by an
unroll factor of 4

Another Unrolled Version

Unroll the i loop by a factor of 4

```
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)

```
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

```
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

```
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





