Quick Tips on Converting Java Programs to Assembly Language

Dhananjai M. Rao

(raodm@muohio.edu) CSA Department Miami University

1. Java to Assembly Conversion

This chapter provides a quick mapping of different constructs from Java to Assembly. Note that the assembly language uses the AT&T syntax and is geared toward the use of the GNU assembler. Furthermore the assembly code snippets are shown in an incremental fashion rather the repeating the complete content.

3.1. Basic Environment

All assembly code developed for use with the GNU assembler must be placed with the code segment which is indicated using ".text" assembler directive. If data is to be explicitly placed in the data segment, then the ".data" assembler directive maybe used prior to definition of the data elements. Figure 1 illustrates the basic start up environment for an assembly program.

```
/* This is a comment */
/* Indicate beginning of code segment */
.text
/* All of your instructions and subprograms go here */
/* indicate beginning of data */
.data
/* All variables and data go here */
Figure 1: Basic setup of an assembly program.
```

3.2. Defining Variables

In assembly language programming there is no well defined concept of variables. Instead data is simply stored at a specific address. Furthermore, no explicit data types are associated with variables. The only factor that really matters is the number of bytes written or read and how the bytes are interpreted. Interpretation of the contents depends on the instruction being used. If an address to a set of bytes is used in an arithmetic instruction, then the bytes are viewed as integers or numbers. Alternatively, if the address to a set of bytes is used in a string instruction, the contents are viewed as a string.

In order to ease use of addresses, the assembler permits the use of labels or **symbols** that are associated with specific addresses. Furthermore, the assembler provides a few intrinsic data types to ease allocation of space and specification of initial values. Figure 2 presents a sample assembly program with definitions for three labels of different data types. Note that when the assembly program is compiled all the labels are ultimately resolved to addresses by the assembler. Furthermore, addresses of the data members are assigned automatically by the assembler and are <u>guaranteed to be contiguous</u>.

```
/* Indicate beginning of code segment */
.text
.data
/* Define a variable called varS of size 2 bytes with
initial value of -22 */
varS: .short -22
/* Define a variable called varA of size 4 bytes with
initial value of 512 in hex */
varA: .int 0x200
/* Define a string variable called varB */
varB: .string "Hello!"
/* Define a floating point variable */
varF: .float 3.142
/* Define two bytes with predefined values */
trailer: .byte 0x55, 0xaa
```

Figure 2: Assembly code to declare data members.

3.3. Arithmetic Expressions

Arithmetic expressions are typically used to represent calculations. The task of converting an arithmetic expression in Java to Assembly can be broken down into the following steps:

- a. Break each operation in the expression into individual operations assigning some temporary location to store result of each operation.
- b. Organize the operations based on the precedence of the operator in Java language and based on nesting within parentheses. This is critical to preserve the original value of the expression.
- c. Assign a general purpose register for the first temporary location and write assembly operator for each step using the temporary register.
- d. After all operators have been translated, store the result into the specified variable.

Here is an example of how the aforementioned steps are applied in practice. Consider translation of the java expression "A=B+C*D" (all of integer variables) into assembly. First the expression is broken down into the following operations:

Temp = D Temp = Temp * C Temp = Temp + B A = Temp

Note that the above operations already pay heed to the operator precedence in the Java language. Next, assume that "%eax" is the general purpose register that is going to be used as the temporary location. Then the above sequence gets translated into the following assembly code:

Step	Assembly Code
%eax = D	Mov D, %eax
%eax = $%$ eax * C	Mul C
%eax = $%$ eax + B	Add B
A = % eax	Mov %eax, A

Note that in the above sequence, the Mul instruction implicitly uses register <code>%eax</code> and therefore the translation was more straightforward. Therefore, choosing <code>%eax</code> as the temporary register is a good rule of thumb.

3.4. IF...ELSE

In Java language the IF...ELSE construct is used to conditionally execute different parts of a program. These Java statements are translated to assembly through a combination of Boolean expressions and conditional jumps. Boolean expressions are similar in philosophy as arithmetic expressions except that they result in *true* or *false* values. Boolean expressions are evaluated in the same manner as arithmetic expressions while paying heed to short-circuiting principles. Here is an example of the translation:

Java	Assembly
if $((a < 1) \parallel (a > 100))$	Movl a, %eax /* Load a into eax. */
	Cmpl %eax, \$1 /* Check it with 1 */
	JL ifBody /* If it is less then do if body */
	Cmpl %eax, \$100 /* If not check it with 100 */
	JG ifBody /* If it is greater do if body */
	Jmp elseBody /* If not do the else body */
{	ifBody:
b = 10;	movl \$10, %ebx
,	movl %ebx, b
}	Jmp ifEnd
else {	elseBody:
b = 20;	movl \$20, %ebx
,	movl %ebx, b
}	ifEnd:

The table below illustrates the translation except that the Boolean expression involves a logical and clause rather than a logical or.

Java	Assembly	
if ((a < 1) && (a > -100))	Movl a, %eax /* Load a into eax. */	
	Cmp %eax, \$1 /* Check it with 1 */	
	JG elseBody /* If it is greater,do else body */	
	Cmpl %eax, \$-100/* If not check it with -100 */	
	JG ifBody /* If it is greater do if body */	
	Jmp elseBody /* If not do the else body */	
{	ifBody:	
b = 10:	Movl \$10, %ebx	
7	movl %ebx, b	
}	jmp ifEnd	
else {	elseBody:	
b = 20:	mov \$20, %ebx	
7	mov %ebx, b	
}	ifEnd:	

3.5. LOOPS

Both for and while loops in Java language are translated to intermediate IF...ELSE constructs and then the IF...ELSE translation mechanism described earlier is applied to convert loops to assembly. The following table provides a translation of loop constructs to IF...ELSE constructs.

Java Original	Translated
for(i=0; (i < 100); i++) {	i = 0;
// Body of for loop	loop:
}	if (i < 100) {
	// Body of for loop
	i++;
	goto loop;
	}
while $((i < 10) \&\& (j < 20))$ {	loop:
// Body for while loop	if $((i < 10) \&\& (j < 20))$ {
}	// Body for while loop
	goto loop;
	}