LABS

Week 5: Compiling; Assembly Language

1. Data Formats

Due to its origins as a 16-bit architecture that expanded into a 32-bit one, Intel uses the term "word" to refer to a 16-bit data type. Based on this, they refer to 32- bit quantities as "double words." They refer to 64-bit quantities as "quad words." Most instructions we will encounter operate on bytes or double words.

Figure 1.1 shows the IA32 representations used for the primitive data types of C. Most of the common data types are stored as double words. This includes both regular and long int's, whether or not they are signed. In addition, all pointers (shown here as char *) are stored as 4-byte double words. Bytes are commonly used when manipulating string data

C declaration	Intel data type	Assembly code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long int`	Double word	`l`	4
`long long int`	—	—	4
`char *`	Double word	`l`	4
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	10/12

Figure 1.1 Sizes of C data in IA32. IA32 does not provide hardware support for 64-bit integer arithmetic. Compiling code with long long data requires generating sequences of operations to perform the arithmetic in 32-bit chunks.

Floating-point numbers come in three different forms: single-precision (4-byte) values, corresponding to C data type float; double-precision (8-byte) values, corresponding to C data type double; and extended-precision (10-byte) values. gcc uses the data type long double to refer to extended-precision floating point values. It also stores them as 12-byte quantities to improve memory system performance.

As the table indicates, most assembly-code instructions generated by **gcc** have a single-character suffix denoting the size of the operand. For example, the data movement instruction has three variants: **movb** (move byte), **movw** (move word), and **movl** (move double word). The suffix 'l' is used for double words, since 32-bit quantities are considered to be "long words," a holdover from an era when 16-bit word sizes were standard. Note that the assembly code uses the suffix 'l' to denote both a 4-byte integer as well as

an 8-byte double-precision floating-point number. This causes no ambiguity, since floating point involves an entirely different set of instructions and registers.

2. Accessing Information

An IA32 central processing unit (CPU) contains a set of eight registers storing 32-bit values. These registers are used to store integer data as well as pointers. Figure 2.1 diagrams the eight registers. Their names all begin with **%e**, but otherwise, they have peculiar names. With the original 8086, the registers were 16 bits and each had a specific purpose. The names were chosen to reflect these different purposes. With flat addressing, the need for specialized registers is greatly reduced. For the most part, the first six registers can be considered general-purpose registers. with no restrictions placed on their use. We said "for the most part," because some instructions use fixed registers as sources and/or destinations. In addition, within procedures there are different conventions for saving and restoring the first three registers (%eax, %ecx, and %edx) than for the next three (%ebx, %edi, and %esi). The final two registers (%ebp and %esp) contain pointers to important places in the program stack. They should only be altered according to the set of standard conventions for stack management.

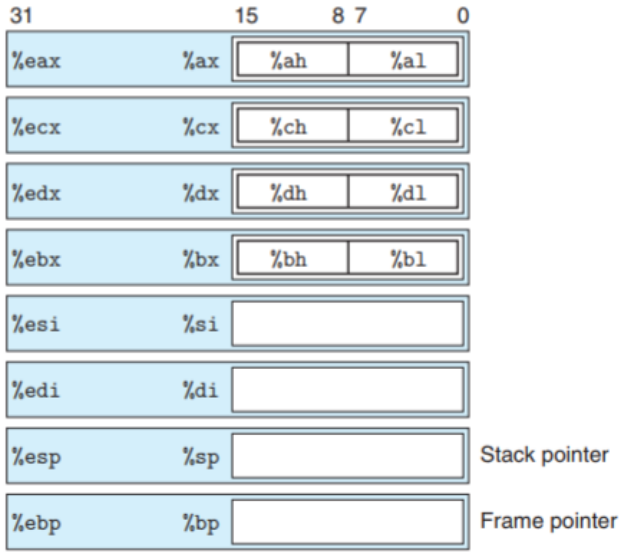

Figure 2.1 IA32 integer registers. All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The 2 low-order bytes of the first four registers can be accessed independently.

As indicated in Figure 2.1, the low-order 2 bytes of the first four registers can be independently read or written by the byte operation instructions. This feature was provided in the 8086 to allow backward compatibility to the 8008 and 8080—two 8-bit microprocessors that date back to 1974. When a byte instruction updates one of these single-byte "register elements," the remaining 3 bytes of the register do not change. Similarly, the low-order 16 bits of each register can be read or written by word operation instructions. This feature stems from IA32's evolutionary heritage as a 16-bit microprocessor and is also used when operating on integers with size designator short.

3. Data Movement Instructions

Instruction		Effect	Description
MOV	*S*, *D*	$D \leftarrow S$	Move
`movb`		Move byte	
`movw`		Move word	
`movl`		Move double word	
MOVS	*S*, *D*	$D \leftarrow \text{SignExtend}(S)$	Move with sign extension
`movsbw`		Move sign-extended byte to word	
`movsbl`		Move sign-extended byte to double word	
`movswl`		Move sign-extended word to double word	
MOVZ	*S*, *D*	$D \leftarrow \text{ZeroExtend}(S)$	Move with zero extension
`movzbw`		Move zero-extended byte to word	
`movzbl`		Move zero-extended byte to double word	
`movzwl`		Move zero-extended word to double word	
`pushl`	*S*	R[%esp] ← R[%esp] − 4; M[R[%esp]] ← *S*	Push double word
`popl`	*D*	*D* ← M[R[%esp]]; R[%esp] ← R[%esp] + 4	Pop double word

Figure 3.1 Data movement instructions

Among the most heavily used instructions are those that copy data from one location to another. The generality of the operand notation allows a simple data movement instruction to perform what in many machines would require a number of instructions. Figure 3.1 lists the important data movement instructions. As can be seen, we group the many different instructions into instruction classes, where the instructions in a class perform the same operation, but with different operand sizes. For example, the **mov** class consists of three instructions: **movb**, **movw**, and **movl**. All three of these instructions perform the same operation; they differ only in that they operate on data of size 1, 2, and 4 bytes, respectively.

The instructions in the mov class copy their source values to their destinations. The source operand designates a value that is immediate, stored in a register, or stored in memory. The destination operand designates a location that is either a register or a memory address. IA32 imposes the restriction that a move instruction cannot have both operands refer to memory locations. Copying a value from one memory location to another requires two instructions—the first to load the source value into a register, and the second to write this register value to the destination.

4. Arithmetic and Logical Operations

Figure 4.1 lists some of the integer and logic operations. Most of the operations are given as instruction classes, as they can have different variants with different operand sizes. (Only leal has no other size variants.) For example, the instruction class add consists of three addition instructions: addb, addw, and addl, adding bytes, words, and double words, respectively. Indeed, each of the instruction classes shown has instructions for operating on byte, word, and double-word data. The operations are divided into four groups: load effective address, unary, binary, and shifts. Binary operations have two operands, while unary operations have one operand.

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Figure 4.1 Integer arithmetic operations. The load effective address (leal) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use >>A and >>L to denote arithmetic and logic shift.

Example:

(a) C code

```
1   int arith(int x,
2             int y,
3             int z)
4   {
5       int t1 = x+y;
6       int t2 = z*48;
7       int t3 = t1 & 0xFFFF;
8       int t4 = t2 * t3;
9       return t4;
10  }
```

(b) Assembly code

```
    x at %ebp+8, y at %ebp+12, z at %ebp+16
1     movl    16(%ebp), %eax        z
2     leal    (%eax,%eax,2), %eax   z*3
3     sall    $4, %eax              t2 = z*48
4     movl    12(%ebp), %edx        y
5     addl    8(%ebp), %edx         t1 = x+y
6     andl    $65535, %edx          t3 = t1&0xFFFF
7     imull   %edx, %eax            Return t4 = t2*t3
```

5. Condition Codes

In addition to the integer registers, the CPU maintains a set of single-bit condition code registers describing attributes of the most recent arithmetic or logical operation. These registers can then be tested to perform conditional branches. The most useful condition codes are:

CF: Carry Flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

ZF: Zero Flag. The most recent operation yielded zero.

SF: Sign Flag. The most recent operation yielded a negative value.

OF: Overflow Flag. The most recent operation caused a two's-complement overflow—either negative or positive.

Instruction		Based on	Description
CMP	S_2, S_1	$S_1 - S_2$	Compare
cmpb		Compare byte	
cmpw		Compare word	
cmpl		Compare double word	
TEST	S_2, S_1	$S_1 \& S_2$	Test
testb		Test byte	
testw		Test word	
testl		Test double word	

Figure 5.1 Comparison and test instructions. These instructions set the condition codes without updating any other registers.

6. Jump Instructions and Their Encodings

Under normal execution, instructions follow each other in the order they are listed. A jump instruction can cause the execution to switch to a completely new position in the program. These jump destinations are generally indicated in assembly code by a label. Consider the following (very contrived) assembly-code sequence:

```
1      movl $0,%eax            Set %eax to 0
2      jmp .L1                 Goto .L1
3      movl (%eax),%edx        Null pointer dereference
4    .L1:
5      popl %edx
```

The instruction **jmp** .L1 will cause the program to skip over the **movl** instruction and instead resume execution with the **popl** instruction. In generating the object-code file, the assembler determines the

addresses of all labeled instructions and encodes the jump targets (the addresses of the destination instructions) as part of the jump instructions.

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	ZF	Equal / zero
`jne`	*Label*	`jnz`	~ZF	Not equal / not zero
`js`	*Label*		SF	Negative
`jns`	*Label*		~SF	Nonnegative
`jg`	*Label*	`jnle`	~(SF ^ OF) & ~ZF	Greater (signed >)
`jge`	*Label*	`jnl`	~(SF ^ OF)	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	SF ^ OF	Less (signed <)
`jle`	*Label*	`jng`	(SF ^ OF) | ZF	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	~CF & ~ZF	Above (unsigned >)
`jae`	*Label*	`jnb`	~CF	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	CF	Below (unsigned <)
`jbe`	*Label*	`jna`	CF | ZF	Below or equal (unsigned <=)

Figure 6.1 The jump instructions. These instructions jump to a labeled destination when the jump condition holds.

Assignment

Practice Problem 1

Compile and assemble the below C code. Try to annotate the assembly code.

```
int absdiff(int x,int y){
  if(x<y)
    return y-x;
  else
    return x-y;

}
```

For the above code, try to write the equivalent goto version that mimics the control flow of the assembly code.

Practice Problem 2

For the below Assembly Code (AT&T syntax) try to write the equivalent C code.

```
movl    8(%ebp),%edx
movl    12(%ebp),       %eax
testl   %eax    ,       %eax
je      .L3
testl   %edx,           %edx
jle     .L3
addl    %edx    ,       (%eax)
.L3:
```

Practice Problem 3

Compile and assemble the below C code. Try to annotate the assembly code.

```
int fact_do(int n)
{
  int result=1;
  do{
    result*=n;
    n=n-1;
  }while(n>1);
  return result;
}
```

Create a table of register usage.

Register	Variable	Initially

Practice Problem 4

For the below Assembly Code (AT&T syntax) try to write the equivalent C code and create the register usage table.

```
movl    8(%ebp) ,       %edx
movl    $1      ,       %eax
cmpl    $1      ,       %edx
jle     .L1
.L2:
imull   %edx    ,       %eax
subl    $1      ,       %edx
cmpl    $1      ,       %edx
jg      .L2
.L2:
```