Section Notes 4 (Worksheet)

Assembly Instructions and Bitwise Operators

Overview:

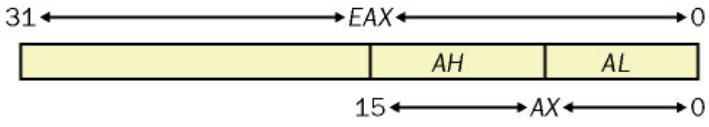
- Registers
- Assembly Operands (Immediates, Registers, Memory)
- Condition Codes
- Jumps
- Control Flow (loops)
- Procedure Calls
- Bitwise Operators

Meet the Registers

Register	Internal memory storage locations within the processor
Data Registers	
AX	primary accumulator, typically contains the return value
BX	base register (indexed addressing)
CX	count register (loop counts, etc.)
DX	data register (input/output operations, misc.)
Pointer Registers	
IP	instruction pointer -- address of next instruction to be executed
SP	stack pointer -- current offset value within the program stack
BP	base pointer -- base of the subroutine stack, used to reference parameter variables

Important Observations!

- Registers are commonly known as EAX, EBX, etc. -- the additional “E” means the register is 32-bits, rather than the 16-bit, two-character registers. (Fun fact: RAX is the 64-bit version of EAX)



Value	Bits
EAX	0–31
AX	0–15
AH	8–15
AL	0–7

- Stacks grow down in memory. This means that addresses decrease as the stack grows.
- The top of the stack's address is contained in the register %ESP, and is the lowest address of all stack elements.
- (%register) returns the value contained at the address the register refers to
- %register returns the address the register refers to
- <cmd> <v1> <v2> treats v1 as the (src / destination) and v2 as the (src / destination)

Exercises

pushl %ebp stores %ebp on the stack. What two instructions does it equal, and why are the others wrong?

a. **subl $4, %esp**
   movl %ebp, (%esp)

b. **addl $4, %esp**
   movl %ebp, %esp

c. **subl $4, (%esp)**
   movl (%ebp), %esp

popl %ebp pops the top value off the stack and stores it in %ebp. What two instructions does it equal, and why are the others wrong?

a. **movl (%esp), %ebp**
   addl $4, (%esp)

b. **movl %esp, %ebp**
   addl $4, %esp

c. **movl %esp, %ebp**
   addl $4, %esp

d. **movl (%esp), %ebp**

addl $4, %esp

Assembly Operand Specifiers

Assembly instructions can take three different types of operands: a constant, or immediate, value, a register value, or a memory value.

Type	**Form**	**Operand value**	**Example**
Immediate	*$Imm*	*Imm*	$42
Register	E_a	$R[E_a]$	%eax
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] * s]$	$42(%esp, %edx, 4)

Things to note:

1. *b* for “base”, i for “index”, s for “scale” or “size”
2. *s* has to be one of 1, 2, 4, or 8

The “Memory” operand form is one of many ways to access memory, and is considered the most general. This is the most useful form to remember, because we can derive all of the others from it. In essence, the other forms leave off some of the arguments. One way to think of those forms is as supplying “default” arguments to this specifier, where the defaults are 0 for *Imm*, $R[E_b]$, and $R[E_i]$, and 1 for *s*.

The full list of memory operand specifiers is given in Figure 3.3 of the text (pg. 169).

Exercise 1

Assume the following values are stored at the indicated memory addresses and registers.

Address	**Value**	**Register**	**Value**
0x100	0xFF	%eax	0x100
0x104	0xAB	%ecx	0x1
0x108	0x13	%edx	0x3
0x10C	0x11		

Fill in the missing value for each operand:

Operand	**Value**
%eax	0x100
0x104	
$0x108	
(%eax)	0xFF
4(%eax)	0xAB
9(%eax, %edx)	
260(%ecx, %edx)	
0xFC(, %ecx, 4)	
(%eax, %edx, 4)	

Exercise 2

A function with prototype int decode(int x, int y, int z); is compiled into assembly. The body of the code is as follows:

```
1 # x at %ebp+8, y at %ebp+12, z at %ebp+16
2 movl  12(%ebp), %edx
3 subl  16(%ebp), %edx
4 movl  %edx, %eax
5 sall  $31, %eax
6 sarl  $31, %eax
7 imull 8(%ebp), %edx
8 xorl  %edx, %eax
```

Parameters `x`, `y`, and `z` are stored at memory locations with offsets 8, 12, and 16 relative to the address in register `%ebp`. The code stores the return value in register `%eax`. Write the C code for decode that will have an effect equivalent to our assembly code.

```
int decode(int x, int y, int z) {

}
```

Condition Codes

EFLAGS is a 32 bit register that contains separate bits for each of the condition flags, which are set automatically by the CPU to represent the result of the previously executed instruction. Examples of condition flags include the following:

CF: Carry Flag	The most recent operation generated a carry out of the most significant bit. Used to detect overflow of 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 (negative or positive)

Typically these flags are set or cleared as the result of an instruction (e.g. `add, sub, cmp,` etc.) and can then be used to conditionally set a single byte (`set`), jump to a new part of the program (`jmp`) or transfer some data (`mov`).

Exercise

For each one of the following, determine which flags are set by the add instruction and why.

[a]

```
movl $0x40, %eax
movl $0xffffffc0, %ebx
addl %eax, %ebx
```

[b]

```
movl $0x2a, %eax
movl $0xffffffc0, %ebx
addl %eax, %ebx
```

[c]

```
movl $0x7FFFFFF0, %eax
movl $0x2c, %ebx
addl %eax, %ebx
```

Jumps

There are two methods of performing jumps: *direct* and *indirect*. For direct jumps, the destination is specified as a label (e.g. `jmp .L1` or, after compiling, `jmp 0x8049994`) and is encoded as part of the instruction. For indirect jumps, the jump target is read from a register or a memory location and is preceded by a '*'. For example:

```
jmp *%eax
```

uses the value in register `%eax` as the jump target.

Certain jumps are combined with certain condition flags to create conditional jumps:

Instruction		Synonym	Description
`je`	*Label*	`jz`	Equal / zero
`jne`	*Label*	`jnz`	Not equal / not zero
`js`	*Label*		Negative
`jns`	*Label*		Nonnegative
`jg`	*Label*	`jnle`	Greater
`jge`	*Label*	`jnl`	Greater or equal
`jl`	*Label*	`jnge`	Less
`jle`	*Label*	`jng`	Less or equal
`ja`	*Label*	`jnbe`	Above
`jae`	*Label*	`jnb`	Above or equal
`jb`	*Label*	`jnae`	Below
`jbe`	*Label*	`jna`	below or equal

Exercise

Which of the condition flags do each of the above jump instructions use in determining if it will execute the jump?

Conditional jump instruction	**Jump condition**
`je`	
`jne`	
`js`	
`jns`	

`jg`	~ZF & ~(OF ^ SF)
`jge`	~(OF ^ SF)
`jl`	(OF ^ SF)
`jle`	
`ja`	~CF & ~ZF
`jae`	~CF
`jb`	CF
`jbe`	CF | ZF

((unsigned)v1 > (unsigned)v2) ⇒ v1 - v2

CF: Carry Flag	The most recent operation generated a carry out of the most significant bit. Used to detect overflow of 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 (negative or positive)

Control Flow: Loops

Let us now see how loops are implemented using conditional jumps. The following is a simple function to compute a Fibonacci sequence:

```
int fibonacci(int n) {
   int i = 0;
   int val = 0;
   int nval = 1;
   do {
       int t = val + nval;
       val = nval;
       nval = t;
       i++;
   } while (i < n);
   return val;
}
```

Generate the assembly code in the cs61 machine:
`$ gcc -O2 -S -m32 fibonacci.c`

Let's look at the code of this function, and focus on the code inside the loop.

Register	Variable	Initially
%ecx	i	0
%ebx	val	0
%edx	nval	1
%esi	n	n
%eax	t	0

```
fibonacci:
      pushl   %ebp              # save old value of %ebp
      xorl    %ecx, %ecx              # i = 0
      movl    %esp, %ebp              # %ebp = base of current stack frame
      movl    $1, %edx                # nval = 1
      pushl   %esi              # save previous value of %esi
      movl    8(%ebp), %esi     # load n into %esi
      pushl   %ebx              # save previous value of %ebx
      xorl    %ebx, %ebx              # val = 0
      jmp     .L2                     # jump to .L2
.L7:
      movl    %eax, %edx              # nval = t
.L2:
      addl    $1, %ecx                # i++
      cmpl    %esi, %ecx              # compare i to n
      leal    (%edx,%ebx), %eax       # t = val + nval
```

```
movl   %edx, %ebx              # val = nval
jl     .L7                     # Jump if i < n
popl   %ebx                    # restore %ebx
movl   %edx, %eax              # Set nval (==val) as the ret. value
popl   %esi                    # restore %esi
popl   %ebp                    # restore %ebp
ret                            # pop return address and jump to it
```

Note that assembly code instructions do not always appear in the same order as the corresponding code in the C program. For example, `i` is incremented near the beginning of the loop in the assembly program, but is incremented at the end of the loop in the C source program. The compiler is free to re-arrange the order of the instructions as long as it does not change the meaning, or behavior, of the code.

Which line in the assembly actually causes the code to loop? What lines are important in making sure that we don't loop forever?

Now we'll look at fibonacci defined slightly differently:

```
int fibonacci(int n) {
       // ignoring negative n
       if(n == 0 || n == 1)
         return n;
       else
         return fibonacci(n-2) + fibonacci(n-1);
}
```

What is the stack going to look like midway through a call to, say, fibonacci(100000)?

Let's try one more time:

```
int fibonacci(int n) {
   if(n < 3)
       return 1;
   else
       return fibonacci_helper(n-2,1,1);
}

int fibonacci_helper(int n, int n0, int n1) {
   if(n == 0)
       return n1;
   return fibonacci_helper(n-1, n1, n0+n1);
}
```

What's so different about this particular implementation of fibonacci? What happens to the stack / what does the stack look like midway through a call to fibonacci(100000)?

Exercise

Consider the following assembly code:

```
# x at %ebp+8, n at %ebp+12
1      movl    8(%ebp), %esi   (x)
```

```
2       movl    12(%ebp), %ebx  (n)
3       movl    $-1, %edi       (result)
4       movl    $1, %edx        (mask)
5  .L2:
6       movl    %edx, %eax      (eax = mask)
7       andl    %esi, %eax      (eax = eax & x); (eax = mask & x);
8       xorl    %eax, %edi      (result=result^eax); (result=result^(mask&x))
x))
9       movl    %ebx, %ecx      (ecx = ebx = n)
10      sall    %cl, %edx       (mask <<= n)
11      testl   %edx, %edx      (sets ZF if edx == 0)
12      jne     .L2             (jump if ~ZF);
13      movl    %edi, %eax
```

The preceding code was generated by compiling C code that had the following overall form. Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register `%eax`. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables.

a. Which registers hold program values `x`, `n`, `result`, and `mask`?
b. What are the initial values of `result` and `mask`?
c. What is the test condition for `mask`?
d. How does `mask` get updated?
e. How does `result` get updated?
f. Fill in all the missing parts of the C code.

```
1  int loop(int x, int n)
2  {
3      int result = _________;
4      int mask;
5      for (mask = __________; mask __________; mask = _________) {
6          result ^= __________;
7      }
8      return result;
9  }
```

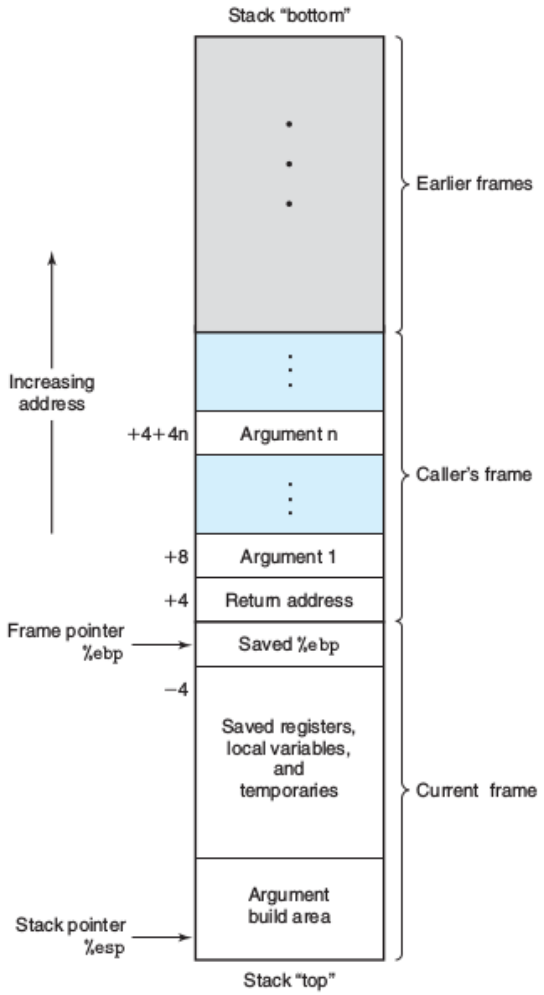
Procedure Calls

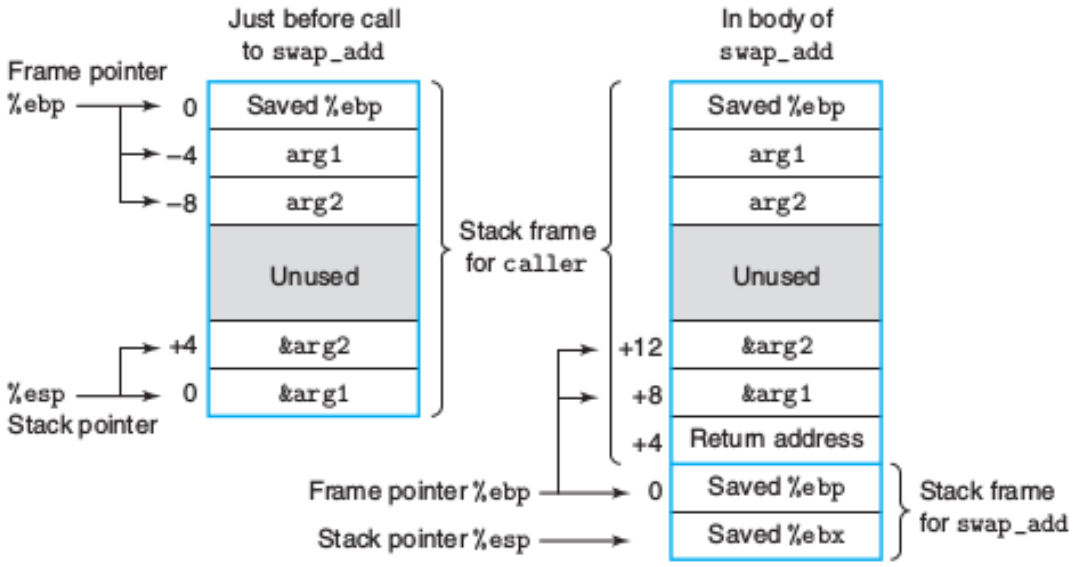
Vocabulary		
Caller	Function that calls a function	`void add_one(int x);`

Callee	Function that gets called	`int main() {`
		`  int x = 1;`
		`  add_one(x);`
		
		`printf("yay!");`
		`  return 0;`
		`}`

Callee-saved	Type of register -- the callee function, on return, must ensure the registers' value is the same as when the function was called	

Caller-saved	Type of register -- the caller function must save the register's value even should the callee function modify the value	`void add_one(int x)`
		`{`
		`  x += 1;`
		`  return;`
		`}`





Some Questions:

1. Which registers do you think are caller-saved, looking at this function? Callee-saved?
2. What do you think the "ret" command, typically called at the end of the function, does? (Hint: it pertains to the return address)
3. How do we know how much space to allocate to each function (i.e. determine the size of the function "stack")?
4. Why are arguments pushed with the first argument pushed on last to the stack? (Hint: think *printf*)

Let's say we are given the following assembly code for a function:

```
1 pushl %edi
2 pushl %esi
3 pushl %ebx
4 sub $0x24, %esp
5 movl 24(%ebp), %eax
6 imull 16(%ebp), %eax
7 movl 24(%ebp), %ebx
8 leal 0(, %eax, 4), %ecx
9 addl 8(%ebp), %ecx
10 movl %ebx, %edx
11 subl 12(%ebp), %edx
.....
20 popl %ebx
21 popl %esi
22 popl %edi
```

1. Why are `%edi, %esi,and %ebx` pushed onto the stack at the beginning of this function and popped off at the end?
2. What about `%eax, %edx, and %ecx`? Why aren't they put on the stack?
3. What do `24(%ebp)` and `16(%ebp)` refer to?
4. Why do we subtract 0x24 from %esp? What might be put in that area?

Let's say we are given the following C code for a function:

```
int proc(void) {
   int x, y;
   scanf("%x %x", &y, &x);
   return x - y;
}
```

And the compiler generates this assembly code for it:

```
1 proc:
2   pushl %ebp
3   movl %esp, %ebp
4   subl $24, $esp
5   addl $-4, %esp
6   leal -4(%ebp), %eax
7   pushl %eax
8   leal -8(%ebp), %eax
9   pushl %eax
10  pushl $.LC0         # Pointer to string "%x %x"
11  call scanf
12  movl -8(%ebp), %eax
13  movl -4(%ebp), %edx
14  subl %eax, %edx
15  movl %edx, %eax
16  movl %ebp, %esp
17  popl %ebp
18  ret
```

Let's assume procedure `proc` starts executing with the following register values:

`%esp = 0x800040`

`%ebp` = `0x800060`

Suppose proc calls scanf (line 11) and scanf reads values 0x46 and 0x53 from the standard input. Assume the string "%x %x" is stored at memory location `0x300070` (i.e., the label `.LC0` is translated to the address 0x300070).

a. What value does `%ebp` get on line 3?
b. At what addresses are local variables `x` and `y` stored?
c. What is the value of `%esp` after line 10?
d. What does the stack frame look like before line 11? If the line numbers all the way on the left were the addresses of the instructions, what value would the call instruction push onto the stack?

Bitwise Operators

Description	c notation	Computer Algebra
Bitwise not (ones' complement)	~	¬
Bitwise or	|	|
Bitwise and	&	&
Bitwise xor	^	$\oplus$
Left shift	<<	<<
Right shift (logical)	>>	$\overset{u}{\gg}$
Right shift (arithmetic)	>>	$\overset{s}{\gg}$

The above table shows the bitwise operators available in c along with their computer algebra counterpart. **One operator to note in particular is right shift.** Logical right shift will shift the bits to the right and always fill with 0s while arithmetic shift will retain the sign bit while shifting. The C standard doesn't specify which should be used so either could be. Unsigned right shifts are always logical, most platforms use arithmetic right shift for signed data.

Given the value -13 how can we force a logical right shift of 1 bit (-13 $\overset{u}{\gg}$ 1)?

De Morgan's Laws

De Morgan's laws (also known as De Morgan's theorems) are two rules that allow certain boolean logic statements to be converted from using *not,and* logic to *not,or*. The theorems are as follows:

```
~(A & B) = ~A | ~B
~(A | B) = ~A & ~B
```

Using these laws it is sometimes possible to reduce the number of gates needed to represent certain logical statements. For an example see: http://www.allaboutcircuits.com/vol_4/chpt_7/8.html

When am I ever going to use this stuff anyway?

There are a number of uses for these types of operators, some of which you may use everyday without knowing it: masks and hash functions. We'll look at hash functions in next section. A mask or bitmask

is a value that when combined with the & operator can be used to isolate certain bits. It is used in IP routing among others. Your IP address is part of a subnet. All IP addresses on the same subnets can reach each other directly. That is any two addresses can communicate without passing through a router. The way you find the range of addresses on the subnet is with a mask. For example, you may have seen an address like:

ip 192.168.0.125 subnet mask 255.255.252.192

This means we are using 26 bits for the mask. We can write 255.255.255.192 in binary as:

```
11111111.11111111.11111111.11000000
```

Similarly, we can write the IP address 192.168.0.125 in binary as:

```
11000111.10101000.00000000.01111101
```

By applying & to these two addresses we can find the subnet:

```
   11000111.10101000.00000000.01111101
 & 11111111.11111111.11111111.11000000
   11000111.10101000.00000000.01000000
```

This means that any machine between:
11000111.10101000.00000000.01000000
and
11000111.10101000.00000000.01111111
are on the same network. This can be written in the more familiar 192.168.0.64 through 192.168.0.127.

Another good use of bitwise operators is to generate simple hashes.

A string hash pjw - Aho, Sethi, and Ullman pp. 434-438 (Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986):

```
unsigned pjw(char* ki, int length) {
    unsigned g,h = 0;
    for (int i = 0; i < length; ki++, i++) {
        // The top 4 bits of h are all zero
        h = (h << 4) + *ki;      // shift h 4 bits left, add in ki
        g = h & 0xf0000000;      // get the top 4 bits of h
        if (g != 0)              // if the top 4 bits aren't zero,
            h = h ^ (g >> 24);   //   move them to the low end of h
        h = h ^ g;
       // The top 4 bits of h are again all zero
    }
    return h;
}
```

Exercises

What does this do:

```
unsigned g( unsigned x ) {
```

```
        unsigned s, r, o;
        s = x & -x;
        r = x + s;
        o = x ^ r;
        o = (o >> 2)/s;
        return r | o;
}
```

Overflow checking (now without overflowing!)(Assuming 32 bit architecture)

One problem that you had to protect against in pset1 was what would happen if some code called your library in a way that would cause integer overflow. We can take advantage of bitwise operators to check if two numbers will overflow without causing an overflow:

```
// Check if adding x to y will cause an overflow.
int will_adding_overflow(int x, int y){
    uint32_t z = ~(x^y) & 0x80000000;
    return (z & ~(((x ^ z) + y) ^ y))>>31;
}
```

Can you come up with a similar function that will test if subtraction will overflow?

```
// Check if subtracting y from x will cause an overflow.
int will_subtracting_overflow(int x, int y){



}
```