X86-NASM

STANDARD COMMANDS

Comment your code with a semicolon (;)! The assembler won't read anything after it.

Move

`mov ax,bx ;` `ax = bx`

- ➔ Use this command when you want to move a value around. You can also use this to move from a **variable** to a **register**
- ➔ You can also use it to move a **constant** (ie. number) into a register
- ➔ Mov works both ways. You can also move the contents of a **register** into **memory**
  - ◆ Usually, you use a **variable** as the base, then add to it. For example, `[output + ecx]` is the memory location ecx **bytes** from output
- ➔ Some examples:
  - ◆ `mov bl, 5 ;move the value 5 into the register bl`
  - ◆ `mov rsi, output ;move the variable output into register rsi`
  - ◆ `mov eax, ebx ;move the value in ebx to eax`
  - ◆ `mov [output + ecx], bh # move the value in bh to the memory location [output + ecx]`

Load Effective Address

`lea eax, [output+esi+0x10]` `;eax = output + esi + 0x10`

- ➔ An **effective address** is just a location in memory. You can think of it as an address on your street.
  - ◆ output would be the location of your house
  - ◆ esi would be a number, let's say 0x5. Added to 0x10, that's 0x15. So the address from this computation would be 0x15 houses down from yours
- ➔ Some examples (you won't use lea in the programs you write, but it will be useful for our final project)
  - ◆ `lea eax, [ebx, esi * 4] ;load ebx + esi*4 into eax`

Add

`add eax,ebx ;``eax = eax + ebx`

- ➔ Add two values. You can add the values in two registers, a register and a constant (ex. 0x5), or dereference an address to access a location in memory
- ➔ Some examples:

- ◆ `add eax, ebx ;if eax is 2 and ebx is 3, eax will now be 5`
- ◆ `add eax, 5 ;if eax is 2 then eax will now be 7`
- ◆ `add eax, [0x400ee4] ;if eax is 2 and 3 is at the address 0x400ee4, then eax will now be 5`

➔ You can subtract by adding a negative number

Shift right

`shr eax, 7` `# eax = eax >> 7`

- ➔ Before: eax = 0110 0001 (binary 97)
- ➔ Instruction: `shr eax,3`
- ➔ After: eax = 0000 1100 (now binary 12 (97 / 2^3))
- ➔ **Shifting** is the act of shifting the bits in a binary* number back and forth. It can be used to **multiply** and **divide** numbers
- ➔ Right shifting is used to **divide by powers of 2**
- ➔ `shr eax, 7` divides the value in eax by 2^7, or 128
- ➔ Examples
  - ◆ `shr eax, 2 ( 1000 → 0010, binary 8 → binary 2 )`

Shift left

`shl eax, 7` `# eax = eax << 7`

- ➔ Before: eax = 0100 0011 (binary 67)
- ➔ Instruction: `shl eax,3`
- ➔ After: eax = 0010 0001 1000 (now binary 536 (67 * 2^3))
- ➔ Left shifting is used to **multiply by powers of 2**

BITWISE OPERATIONS

You probably won't use these much in the class, but it's important to know and you can use it to do some cool math!

And

`and eax,ebx` `# ax = eax & ebx`

- ➔ A bitwise instruction i.e. each bit is handled individually
  - ◆ 1 & 1 = 1
  - ◆ 1 & 0 = 0
  - ◆ 0 & 0 = 0
- ➔ Eg: 0101 & 1100 = 0100

Or

or eax,ebx

ax = ax | bx

- ➔ A bitwise instruction i.e. each bit is handled individually
  - ◆ 1 | 1 = 1
  - ◆ 1 | 0 = 1
  - ◆ 0 | 0 = 0
- ➔ Eg: 0101 & 1100 = 1101

Not

not eax

eax = !eax

- ➔ A bitwise instruction i.e. each bit is handled individually
  - ◆ !1 = 0
  - ◆ !0 = 1
- ➔ Eg: !0101 = 1010

CONDITIONALS

Compare

cmp eax, ebx

- ➔ **Comparing** is used to jump around the program. Say you only want to do the next few lines if the value of eax is 5 or greater. You must use the compare instruction first to set the necessary **flags** that tell the computer what to do next

Unconditional Jump

jmp label # jump to label

Conditional jump

j(cc) label # jump to label if condition (see condition codes below) true (cc = condition code. ex jle = jump if less than)

- ➔ **Important!!** You must use the cmp command **first**. Then the condition will be applied to whatever you compared

Condition code - Meaning

E - Equal

Z - Zero

NE - Not equal

NZ - Not zero

B/ NAE- Below/ Not above or equal
NB/ AE - Not below / Above or equal
BE/ NA - Below or equal / Not above
NBE/ A - Not below or equal / Above
L/ NGE - Less than / Not greater than or equal
NL/ GE - Not less than / Greater than or equal
LE/ NG - Less than or equal / Not greater than
NLE/ G - Not less than or equal/ Greater than

Goto C → X86 NASM

Declaration (initialized)

- `string word = “hangman”;`
- `int a = 5;`
- `char c = ‘a’;`

Initialized variable declarations go in the **.data section**. Initialized means that they are immediately set equal to a value, like “hangman” or the number 5. Since you already know the value of all the variables you want to initialize, you can put their values in **memory** before the program even starts.

```
section .data
      word: db 'hangman'
      a:    db 0x05
      c:    db 'a'
```

db stands for ‘declare byte’. You’re simply telling the assembler that you want to declare some bytes with the following data inside of them. The **labels** (word, a, and c) are how you can access this data anywhere in the program later. If you want a little more space, you can use **dw** instead. This stands for ‘declare word’. A **word** is made up of **four bytes or 32 bits**.

Declaration (uninitialized)

- `char guess;`
- `int i;`

Uninitialized variables go in the **.bss section**. Uninitialized means that you don’t know what value you want to put in that variable yet (like if the user wants to input a guess) but you know you’re gonna need a name and space for it later. You can use the bss section to tell the assembler that you want to reserve space in memory for some **future data**. Similar to initialized data, you use a **label** so you can refer back to it later.

```
section .bss
      guess: resb 1
      i:     resb 1
```

resb stands for ‘reserve byte’. The 1 after is simply telling the assembler to reserve 1 byte. You can use any number here. If you wrote `resb 4`, you would be reserving 4 bytes.

Conditionals and jumps (converting the goto part)

- `if (i == 5) goto endwhile;`
- `if (x >= 2) goto done;`

Writing a conditional in assembly is actually done in 2-3 steps. The first thing you need to do, if you haven't already, is put the value you want to compare (i, x) into a register. Let's say you already declared i and x way back in your .data section. i and x were the labels you used when you declared them.

```
mov eax, i          ;put the value of i into the register eax
mov ebx, x          ;put the value of x into the register ebx

cmp eax, 5          ;compare eax and 5
je  endwhile        ;if they were equal jump to (goto) endwhile

cmp ebx, 2          ;compare ebx and 2
jge done            ;if ebx was >= 2, jump to (goto) done
```

The jump command is always in the format **jxx** where **xx** is the **condition code**. A list of condition codes is in the x86 cheat sheet. To execute a condition goto statement (a goto statement with an if before it), you **MUST** use the cmp command **before** you jump.

If you just have a statement that has goto without a conditional, you can use **`jmp`** to jump no matter what.

Printing and getting input in X86

- `print(output)`
- `inputchar(guess)`

To output and print in nasm assembly, you must use system calls. Here is a chart of useful system calls and how to use them:

Name	**eax**	**edi**	**rsi**	**edx**
read	0	0 (stdin)	label you want to read into	length of input
write	1	1 (stdout)	label you want to print out	length of output
exit	60	0 (no error)	N/A	N/A

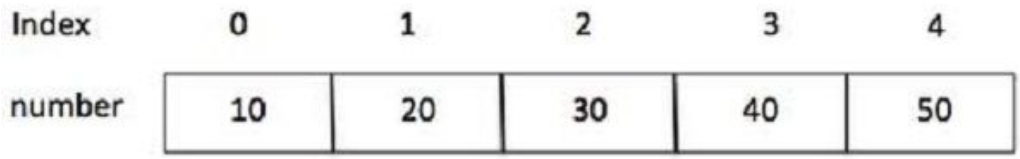
Below is a full program to help you understand how to use syscalls:
It's worth noting that the syscall assumes that output and guess (basically any value in rsi) will hold the address of where you want to put the data. It won't change the actual value inside guess since it's changing the data at the provided address.

```
section .data
      output: '-------', 0xa, 0xd   ; declare output with endline (0xa,
0xd)
      outputLen: equ $-output       ; declare the length of the output

section .bss
      guess: resb 1     ; declare the uninitialized variable guess (1 byte)

section .text
      global _start

_start:
      mov eax, 1              ; eax = 1 = sys_write
      mov edi, 1              ; edi = 1 = stdout
      mov rsi, output         ; rsi = output = label you want to print
      mov edx, outputLen      ; edx = outputLen = length of output
      syscall

      mov eax, 0              ; eax = 0 = sys_read
      mov edi, 0              ; edi = 1 = stdin
      mov rsi, guess          ; rsi = guess = label you want to read into
      mov edx, 1              ; edx = 1 = size of input (guess = 1 byte)
      syscall

      mov eax, 60       ; eax = 60 = sys_exit (exit the program correctly)
      mov edi, 0        ; edi = 0 = no error (error code)
      syscall
```

Array Access

- `word[i] = 'b'`
- `third = word[3]`

An **array** is a series of characters or numbers that are all related and right next to each other in memory. To access different pieces of an array, most programming languages use the square bracket operator (`[]`). The number in the brackets is called the index. Let's say you have an array called 'number':



`number[0]` would be `10`, the first element. `number[2]` would be `30`.

So how does this related to our word? In C and assembly language, strings are actually just arrays of characters, like this:

Index	0	1	2	3	4	5	6
Value	h	a	n	g	m	a	n

If you have the statement `word[i] = 'b'`, and `i` was equal to `2`, then the array would change to this:

Index	0	1	**2**	3	4	5	6
Value	h	a	**b**	g	m	a	n

In assembly, only the address of the first value is stored in memory (in this case, the h). So, if you declared word like this:

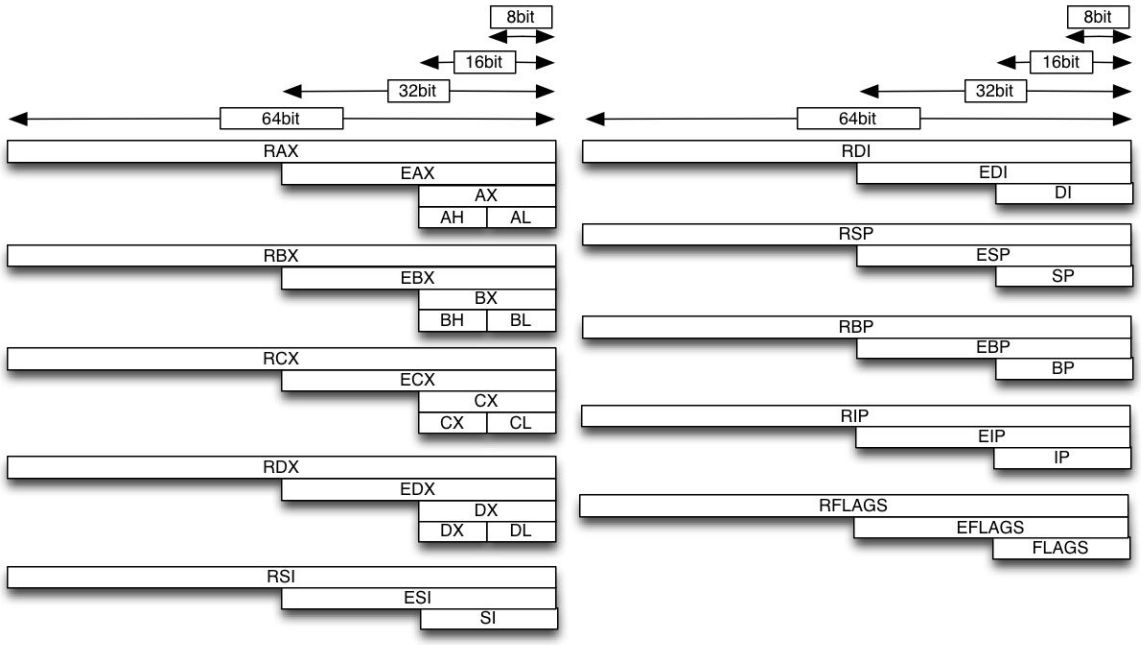
```
section .data
      word: db 'hangman'
      wordLen: equ $-word
```

`word` would technically hold only the address of the h. That's why you need the `wordLen` variable to remember how long the word is.

While there is a square bracket operator in assembly, it is used a bit differently. The square brackets tell the assembler that the value inside of it is an **address** and it needs to **dereference** it (that is, go to the address) to access the value.

```
mov eax, word                ; put the address of word into eax
mov ebx, 3                   ; put 3 into ebx. this is your index
mov [word + ebx], 0x62       ; the ascii value for 'b' is 0x62. put that
in word + ebx (3). These three lines are equivalent to word[3] = 'b'
```

Registers in x86



Each register is 64 bits, with a 32 bit part and a 16 bit part. Some registers also have two 8 bit parts. The idea is that since there are only 6 general purpose registers, you can use the smaller pieces to store more information, when the data you want to store is smaller.

Here is a chart with the sizes of data you'll probably be working with in this class. It will help you decide which size register to use.

Type	Examples	Size	Register Examples
Small number	-128 to 127	8 bits (1 byte)	`ah, al, bh, bl`
Medium number	-32768 to 32767	16 bits (2 bytes)	`ax, bx, cx, dx`
Large number	-2147483648 to 2147483647	32 bits (4 bytes)	`eax, ebx, ecx`
Character	'a', 'b', 'c'	8 bits (1 byte)	`ah, al, bh, bl`

WARNING

The pieces of each register are still technically the **same register**. That's why in the diagram the larger ones overlap with the smaller ones. This means, for example, if you are using register `ah` and decide to put something in `rax`, `eax` or `ax`, the value in ah will be overwritten since the larger pieces use the same space.

That means you can't use `ah` and `ax`, `eax` or `rax` at the same time (or any two+ registers that overlap in the diagram).

Hangman Goto C Answer Key

C Code

```
int main(){
	string word = "hangman";
	string output = "-------";
	int guessed = 0;
	while(guessed != len(word)){
		char guess = 'a';
		print(output);
		inputchar(guess);

		for(int i = 0; i < len(word); i++){
			if(guess == word[i]){

				output[i] = guess;
				print(output);
				guessed++;
			}
		}
	}
	print("You won!");
}
```

Goto C

```
		string word = "hangman";
		string output = "-------";
		int guessed = 0;
while:		if(guessed == len(word)) goto endwhile;
		char guess = 'a';
		print(output);
		inputchar(guess);
		int i = 0;
for:		if(i >= len(word)) goto endfor;
		if(guess != word[i]) goto endif;
		output[i] = guess;
		print(output);
		guessed++;
endif:		goto for
endfor:		goto while
endwhile:	print("You won!");
```

Arrays and Strings in x86

Let's say you write the following assembly code:

```
section .data
      word: db 'hangman'
      wordLen: equ $-word

section .text
      global _start

_start:
      mov eax, word                 ; put the address of word into eax
```

Then, you print the value of eax. The value that prints is

`0x400ee4`

Remember the '0x' means that this value is in **hexadecimal**.

Fill in the bold cells of the table. If you don't think we can know what's in that location, write a ?.

Address in memory	Content
0x300fe2 (location of word)	0x400ee4
...	
0x308e42 (location of wordLen)	7
...	
0x400ee3	
0x400ee4	
0x400ee5	
0x400ee6	
0x400ee7	
0x400ee8	
0x400ee9	
0x400eea	
0x400eeb	

You now add the following lines to your assembly code. Write in the blank what the value of eax will be after each line. It will be very helpful to look back at the filled out table.

Hint: Remember that the square brackets (`[]`) in assembly mean to go to the address inside of them. Find out the value inside the square brackets first and then try to figure out what is at that location.

Example:

```
mov   ebx, 0
mov   eax, [word + ebx]          eax = 'h'
```

```
mov   ebx, 3
mov   eax, [word + ebx]          eax = _____

mov   ebx, 6
mov   eax, [word + ebx]          eax = _____

mov   ebx, 4
mov   eax, [word + ebx]          eax = _____
```

Then, you add these lines:

```
mov   ebx, 2
mov   [word + ebx], 'y'
```

If you print **`word`** to the screen, what will be outputted? _____