x86 Cheat Sheet

Fall 2012

1 x86 Registers

x86 assembly code uses eight 32-bit registers. Additionally, the lower bytes of some of these registers may be accessed independently as 16- or 8-bit registers. The register names are as follows:

4-byte register	**Bytes 0-1**	**Byte 1**	**Byte 0**
`%eax`	`%ax`	`%ah`	`%al`
`%ecx`	`%cx`	`%ch`	`%cl`
`%edx`	`%dx`	`%dh`	`%dl`
`%ebx`	`%bx`	`%bh`	`%bl`
`%esi`	`%si`	–	–
`%edi`	`%di`	–	–
`%esp`	`%sp`	–	–
`%ebp`	`%bp`	–	–

More information about registers in x86 can be found on pages 168-169 of the textbook. For more details of register usage, see Register Usage, below.

2 Operand Specifiers

The basic types of operand specifiers are below. In the following table,

- *Imm* refers to a constant value, e.g. `0x8048d8e` or `48`,
- E_x refers to a register, e.g. `%eax`,
- R[E_x] refers to the value stored in register E_x, and
- M[x] refers to the value stored at memory address x.

Type	**Form**	**Operand value**	**Name**
Immediate	$*Imm*	*Imm*	Immediate
Register	E_a	R[E_a]	Register
Memory	*Imm*	M[*Imm*]	Absolute
Memory	(E_a)	M[R[E_b]]	Absolute
Memory	*Imm*(E_b, E_i, s)	M[*Imm*+R[E_b]+(R[E_i]$\times s$)]	Scaled indexed

More information about operand specifiers can be found on pages 169-170 of the textbook.

3 x86 Instructions

In the following tables,

- "byte" refers to a one-byte integer (suffix `b`),
- "word" refers to a two-byte integer (suffix `w`), and
- "double word" refers to a four-byte integer (suffix `l`).

3.1 Data Movement

Instruction		Description	Page #
`movb`	S, D		
`movw`	S, D	Move source to destination	171
`movl`	S, D		
`movsbw`	S, D	Move byte to word (sign extended)	
`movsbl`	S, D	Move byte to double word (sign extended)	171
`movswl`	S, D	Move word to double word (sign extended)	
`movzbw`	S, D	Move byte to word (zero extended)	
`movzbl`	S, D	Move byte to double word (zero extended)	171
`movzwl`	S, D	Move word to double word (zero extended)	
`pushb`	S		
`pushw`	S	Push byte/word/double word onto stack	171
`pushl`	S		
`popb`	D		
`popw`	D	Pop byte/word/double word from stack	171
`popl`	D		

3.2 Arithmetic Operations

3.2.1 Unary Operations

Instruction		Description	Page #
`incb`	D		
`incw`	D	Increment by 1	178
`incl`	D		
`decb`	D		
`decw`	D	Decrement by 1	178
`decl`	D		
`negb`	D		
`negw`	D	Arithmetic negation	178
`negl`	D		
`notb`	D		
`notw`	D	Bitwise complement	178
`notl`	D		

3.2.2 Binary Operations

Instruction		Description	Page #
`leal`	S, D	Load effective address of source into destination	177
`addb`	S, D		
`addw`	S, D	Add source to destination	178
`addl`	S, D		
`subb`	S, D		
`subw`	S, D	Subtract source from destination	178
`subl`	S, D		
`imulb`	S, D		
`imulw`	S, D	Multiply destination by source	178
`imull`	S, D		
`xorb`	S, D		
`xorw`	S, D	Bitwise XOR destination with source	178
`xorl`	S, D		
`orb`	S, D		
`orw`	S, D	Bitwise OR destination with source	178
`orl`	S, D		
`andb`	S, D		
`andw`	S, D	Bitwise AND destination with source	178
`andl`	S, D		

3.2.3 Shift Operations

Instruction		Description	Page #
`salb` / `shlb`	k, D		
`salw` / `shlw`	k, D	Left shift by k	179
`sall` / `shll`	k, D		
`sarb`	k, D		
`sarw`	k, D	Arithmetic right shift by k	179
`sarl`	k, D		
`shrb`	k, D		
`shrw`	k, D	Logical right shift by k	179
`shrl`	k, D		

3.2.4 Special Arithmetic Operations

Instruction		Description	Page #
`imull`	S	Signed full multiply of `%eax` by S Result stored in `%edx:%eax`	182
`mull`	S	Unsigned full multiply of `%eax` by S Result stored in `%edx:%eax`	182
`cltd`		Sign extend `%eax` into `%edx`	182
`idivl`	S	Signed divide `%edx:%eax` by S Quotient stored in `%eax` Remainder stored in `%edx`	182
`divl`	S	Unsigned divide `%edx:%eax` by S Quotient stored in `%eax` Remainder stored in `%edx`	182

3.3 Comparison and Test Instructions

Instruction		Description	Page #
`cmpb`	S_2, S_1	Set condition codes according to $S_1 - S_2$	185
`cmpw`	S_2, S_1		
`cmpl`	S_2, S_1		
`testb`	S_2, S_1	Set condition codes according to $S_1 \& S_2$	185
`testw`	S_2, S_1		
`testl`	S_2, S_1		

3.4 Accessing Condition Codes

3.4.1 Conditional Set Instructions

Instruction		Description	Condition Code	Page #
`sete` / `setz`	D	Set if equal/zero	`ZF`	187
`setne` / `setnz`	D	Set if not equal/nonzero	`~ZF`	187
`sets`	D	Set if negative	`SF`	187
`setns`	D	Set if nonnegative	`~SF`	187
`setg` / `setnle`	D	Set if greater (signed)	`~(SF^OF)&~ZF`	187
`setge` / `setnl`	D	Set if greater or equal(signed)	`~(SF^OF)`	187
`setl` / `setnge`	D	Set if less (signed)	`SF^OF`	187
`setle` / `setng`	D	Set if less or equal	`(SF^OF)|ZF`	187
`seta` / `setnbe`	D	Set if above (unsigned)	`~CF&~ZF`	187
`setae` / `setnb`	D	Set if above or equal (unsigned)	`~CF`	187
`setb` / `setnae`	D	Set if below (unsigned)	`CF`	187
`setbe` / `setna`	D	Set if below or equal (unsigned)	`CF|ZF`	187

3.4.2 Jump Instructions

Instruction		Description	Condition Code	Page #
`jmp`	*Label*	Jump to label		189
`jmp`	**Operand*	Jump to specified location		189
`je` / `jz`	*Label*	Jump if equal/zero	`ZF`	189
`jne` / `jnz`	*Label*	Jump if not equal/nonzero	`~ZF`	189
`js`	*Label*	Jump if negative	`SF`	189
`jns`	*Label*	Jump if nonnegative	`~SF`	189
`jg` / `jnle`	*Label*	Jump if greater (signed)	`~(SF^OF)&~ZF`	189
`jge` / `jnl`	*Label*	Jump if greater or equal(signed)	`~(SF^OF)`	189
`jl` / `jnge`	*Label*	Jump if less (signed)	`SF^OF`	189
`jle` / `jng`	*Label*	Jump if less or equal	`(SF^OF)|ZF`	189
`ja` / `jnbe`	*Label*	Jump if above (unsigned)	`~CF&~ZF`	189
`jae` / `jnb`	*Label*	Jump if above or equal (unsigned)	`~CF`	189
`jb` / `jnae`	*Label*	Jump if below (unsigned)	`CF`	189
`jbe` / `jna`	*Label*	Jump if below or equal (unsigned)	`CF|ZF`	189

3.4.3 Conditional Move Instructions

Instruction		Description	Condition Code	Page #
`cmove` / `cmovz`	S, D	Move if equal/zero	`ZF`	206
`cmovne` / `cmovnz`	S, D	Move if not equal/nonzero	`~ZF`	206
`cmovs`	S, D	Move if negative	`SF`	206
`cmovns`	S, D	Move if nonnegative	`~SF`	206
`cmovg` / `cmovnle`	S, D	Move if greater (signed)	`~(SF^OF)&~ZF`	206
`cmovge` / `cmovnl`	S, D	Move if greater or equal(signed)	`~(SF^OF)`	206
`cmovl` / `cmovnge`	S, D	Move if less (signed)	`SF^OF`	206
`cmovle` / `cmovng`	S, D	Move if less or equal	`(SF^OF)|ZF`	206
`cmova` / `cmovnbe`	S, D	Move if above (unsigned)	`~CF&~ZF`	206
`cmovae` / `cmovnb`	S, D	Move if above or equal (unsigned)	`~CF`	206
`cmovb` / `cmovnae`	S, D	Move if below (unsigned)	`CF`	206
`cmovbe` / `cmovna`	S, D	Move if below or equal (unsigned)	`CF|ZF`	206

3.5 Procedure Call Instructions

Instruction		Description	Page #
`call`	*Label*	Push return address and jump to label	221
`call`	**Operand*	Push return address and jump to specified location	221
`leave`		Set `%esp` to `%ebp`, then pop top of stack into `%ebp`	221
`ret`		Pop return address from stack and jump there	221

4 Coding Practices

4.1 Commenting

Each function you write should have a comment at the beginning describing what the function does and any arguments it accepts. In addition, we strongly recommend putting comments alongside your assembly code stating what each set of instructions does in pseudocode or some higher-level language. Line breaks are also helpful to group statements into logical blocks for improved readability.

4.2 Arrays

Arrays are stored in memory as contiguous blocks of data. Typically an array variable acts as a pointer to the first element of the array in memory. To access a given array element, the index value is multiplied by the element size and added to the array pointer. For instance, if `arr` is an array of `int`s, the statement:

```
arr[i] = 3;
```

can be expressed in x86 as follows (assuming the address of arr is stored in `%eax` and the index `i` is stored in `%ecx`):

```
movl $3, (%eax, %ecx, 4)
```

More information about arrays can be found on pages 232-241 of the textbook.

4.3 Register Usage

There are eight 32-bit registers in x86: `%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%ebp`, and `%esp`. Of these, `%eax`, `%ecx`, and `%edx` are considered caller-save registers, meaning that they are not necessarily saved across function calls. By convention, `%eax` is used to store a function's return value, if it exists and is no more than 32 bits long. (Larger return types like structs are returned using the stack.) Registers `%ebx`, `%edi`, and `%esi` are callee-save registers, meaning that they are saved across function calls. Register `%ebp` is used as the *frame pointer*, a fixed pointer to the beginning of the current function's stack frame. Register `%esp` is used as the *stack pointer*, a pointer to the topmost element in the stack.

More information about register usage can be found on pages 223-224 of the textbook.

4.4 Stack Organization and Function Calls

4.4.1 Calling a Function

To call a function, the program should first place any necessary arguments on top of the stack, with the first argument topmost. The program should then execute the call instruction, which will push the return address onto the stack and jump to the start of the specified function.

Example:

```
# Call foo(1, 15)
pushl   $15         # Push 15 onto the stack as a 32-bit integer
pushl   $1          # Push 1 onto the stack as a 32-bit integer
call    foo         # Push return address and jump to label foo
addl    %esp, $8    # Pop arguments off of the stack
```

If the function has a return value, it will be stored in `%eax` after the function call.

4.4.2 Writing a function

An x86 program uses a region of memory called the *stack* to support function calls. As the name suggests, this region is organized as a stack data structure with the "top" of the stack growing towards lower memory addresses. For each function call, new space is created on the stack to store local variables and other data. This is known as a *stack frame*. To accomplish this, you will need to write some code at the beginning and end of each function to create and destroy the stack frame.

Setting Up: When a `call` instruction is executed, the address of the following instruction is pushed onto the stack as the return address and control passes to the specified function. Register `%ebp` is used as a *frame pointer* which provides a pointer to the start of the current stack frame. Unlike the stack pointer, which can fluctuate over the course of a function body, the frame pointer remains fixed during the lifetime of the function.

When setting up the stack frame for a function, the program should first push the old frame pointer onto the stack and set the new frame pointer to the current top of the stack:

```
pushl     %ebp
movl      %esp, %ebp
```

If the function is going to use any of the caller-save registers (`%ebx`, `%edi`, or `%esi`), each one's current value should be pushed onto the stack to be restored at the end:

```
pushl     %ebx
pushl     %edi
pushl     %esi
```

Finally, additional space may be allocated on the stack for local variables. While it is possible to make space on the stack as needed in a function body, it is generally more efficient to allocate this space all at once at the beginning of the function.

```
subl      $0x8, %esp          # Allocate 8 bytes of space on the stack
```

Using the Stack Frame: Once you have set up the stack frame, you can use it to store and access local variables:

- If the function has any arguments, these will be stored before the return address on the stack. Since the stack grows downwards, these will be at positive offsets from the frame pointer, with the first argument at `0x8(%ebp)`, the second at `0xC(%ebp)`, and so on.
- Local variables are stored at negative offsets from the frame pointer, after any saved registers. It is up to the programmer to decide how to organize these. In our example, local variables would be stored at `-0x10(%ebp)` and `-0x14(%ebp)`.
- When calling a function, that function's arguments must be placed at the top of the stack before the `call` instruction (see Calling a Function, above).

Cleaning Up: At the end of a function call, the program state should be restored. First, any saved registers should be restored:

```
movl     -0x4(%ebp), %ebx
movl     -0x8(%ebp), %edi
movl     -0xC(%ebp), %esi
```

Then the stack and frame pointers should be reset:

```
movl     %ebp, %esp          # Set the stack pointer to the current frame pointer
popl     %ebp                # Set the frame pointer to the callee's frame pointer
```

Alternatively, the above two commands can be replaced with the `leave` command:

```
leave
```

Finally, the program should return to the call site, using the `ret` command:

```
ret
```

Summary: Putting it together, the code for a function should look like this:

```
foo:
     pushl     %ebp                 # Update stack and frame pointers
     movl      %esp, %ebp
     pushl     %ebx                 # Save registers, if needed
     pushl     %edi
     pushl     %esi
     subl      $0x8, %esp           # Create additional space for local variables, if any

     # Function body

     movl      -0x4(%ebp), %ebx     # Restore any saved registers
     movl      -0x8(%ebp), %edi
     movl      -0xC(%ebp), %esi
     leave                          # Restore stack and frame pointers
     ret                            # Return
```

Note: if a function has a return value, it should be stored in `%eax` before beginning the cleanup process.

More information about the stack frame and function calls can be found on pages 219-232 of the textbook.