Chapter 1 - Assembly and Disassembly

In this chapter we will be going over assembly and some basic disassembly. We will primarily be focusing on x86 in this chapter.

1.1 Intro to Assembly

1.1.1 General Purpose Registers

Depending on the mode of operation there are going to be between 8 to 16 different available general purpose registers. Each of these registers is then divided into subregisters:

Register Encoding	**Bits 32 - 63**	**Bits 16 - 31**	**Bits 8 - 15**	**Bits 0 - 7**
0	RAX			
	EAX			
		AX		
			AH	AL
3	RBX			
	EBX			
		BX		
			BH	BL
1	RCX			
	ECX			
		CX		
			CH	CL
2	RDX			
	EDX			
		DX		
			DH	DL
6	RSI			
	ESI			
		SI		
				SIL
7	RDI			

Register Encoding	Bits 32 - 63				Bits 16 - 31		Bits 8 - 15	Bits 0 - 7
					EDI			
							DI	
								DIL
5	RBP							
					EBP			
							BP	
								BPL
4	RSP							
					ESP			
							SP	
								SPL
8	R8							
					R8D			
							R8W	
								R8B
9	R9							
					R9D			
							R9W	
								R9B
10	R10							
					R10D			
							R10W	
								R10B
11	R11							
					R11D			
							R11W	
								R11B
12	R12							
					R12D			

Register Encoding	Bits 32 - 63				Bits 16 - 31		Bits 8 - 15	Bits 0 - 7
							R12W	
								R12B
13	R13							
					R13D			
							R13W	
								R13B
14	R14							
					R14D			
							R14W	
								R14B
15	R15							
					R15D			
							R15W	
								R15B

All R*, SIL, DIL, BPL, and SPL registers are only available in long mode. Registers AH, BH, CH, and DH cannot be used in instructions that are not valud outside the long mode.

1.1.2 Accumulators

The EAX register is known as an ***accumulator*** and is used with division and multiplication. It is used both as the implied and target operands. For multiplication this will look like:

Operand Size	Source 1	Source 2	Destination
8 bits	AL	8 bit register or 8 bit memory	AX
16 bits	AX	16 bit register or 16 bit memory	DX:AX
32 bits	EAX	32 bit register or 32 bit memory	EDX:EAX
64 bits	RAX	64 bit register or 64 bit memory	RDX:RAX

For division we store the quotient and remainder:

Operand Size	Dividend	Divisor	Quotient	Remainder

Operand Size	Dividend	Divisor	Quotient	Remainder
8/16 bits	AX	8 bit register or 8 bit memory	AL	AH
16/32 bits	DX:AX	16 bit register or 16 bit memory	AX	DX
32/64 bits	EDX:EAX	32 bit register or 32 bit memory	EAX	EDX
64/128 bits	RDX:RAX	64 bit register or 64 bit memory	RAX	RDX

1.1.3 Counter

The ECX register is also known as the counter register. This register is used in loops as a loop iteration counter. This register's least significnt part, CL, is also often used in bitwise shift operations where it contains the number of bits to be shifted.

1.1.4 Stack Pointer

The ESP register is the stack pointer, this register together with the SS register describe the stack area of a thread. SS contains the descriptor of the stack segment and ESP is the index that points to the current position within the stack.

1.1.5 Source and Destination Indices

The ESI and EDI registers are the source and destination index registers used in string operations. ESI contains the source address and EDI contains the destination address.

1.1.6 Base Pointer

The EBP register is called the base pointer and its most common use is to point to the base of the stack frame during function calls.

1.1.7 EFlags Register

The EFlags register lets us get information about the last operation performed. Everything from overflow in multiplication / division, to carry in subtraction.

Bit 0 - Carry Flag

The carry flag is mostly used for the detected of carrying or borrowing in arithmetic operations and is set if the bit width result of the last such operation exceeds the width of the ALU (arithmetic logic unit).

Bit 2 - Parity Flag

The parity flag is set to 1 in case the number of 1s in the least significant byte is even; otherwise the flag is set to 0.

Bit 4 - Adjust Flag

The adjust flag signals when a carry or borrow occurred in the four least significant bits and is primarily used with binary coded decimal arithmetics.

Bit 6 - Zero Flag

The zero flag is set when the result of an arithmetic or bitwise operation is 0. This includes operations that do not store the result (comparison or test).

Bit 7 - Sign Flag

The sign flag is set when the last mathematical operation resulted in a negative number.

Bit 8 - Trap Flag

The trap flag causes a single step interrupt after every executed instruction.

Bit 9 - Interrupt Enable Flag

The interrupt enable flag defines whether the processor will or will not react to incoming interrupts.

Bit 10 - Direction Flag

The direction flag controls the direction of string operations. An operation is performed from the lower address to the higher address if the flag is reset (0) or from the higher address to the lower address if the flag is set (1).

Bit 11 - Overflow Flag

The overflow flag is set when the result of the operation is either too small or too big a number to fit into the destination operand.

1.1.8 Different Types of Jumps

Assembly uses many different types of jumps, these normally reference some sort of flag, here is a table of all the jumps and what they check:

Instruction	Description	Signed-ness	Flags
JO	Jump if overflow		OF == 1
JNO	Jump if not overflow		OF == 0
JS	Jump if sign		SF == 1
JNS	Jump if not sign		SF == 0
JE	Jump if equal		ZF == 1
JZ	Jump if zero		ZF == 1

Instruction	Description	Signed-ness	Flags
JNE	Jump if not equal		ZF == 0
JNZ	Jump if not zero		ZF == 0
JB	Jump if below	Unsigned	CF == 1
JNAE	Jump if not above or equal	Unsigned	CF == 1
JC	Jump if carry	Unsigned	CF == 1
JNB	Jump if not below	Unsigned	CF == 0
JAE	Jump if above or equal	Unsigned	CF == 0
JNC	Jump if not carry	Unsigned	CF == 0
JBE	Jump if below or equal	Unsigned	CF == 1 Or ZF == 1
JNA	Jump if not above	Unsigned	CF == 1 Or ZF == 1
JA	Jump if above	Unsigned	CF == 0 Or ZF == 0
JNBE	Jump if not below or equal	Unsigned	CF == 0 Or ZF == 0
JL	Jump if less	Signed	SF != OF
JNGE	Jump if not greater or equal	Signed	SF != OF
JGE	Jump if greater or equal	Signed	SF == OF
JNL	Jump if not less	Signed	SF == OF
JLE	Jump if less or equal	Signed	ZF == 1 OR SF != OF
JNG	Jump if not greater	Signed	ZF == 1 OR SF != OF
JG	Jump if greater	Signed	ZF == 0 OR SF == OF
JNLE	Jump if not less or equal	Signed	ZF == 0 OR SF == OF
JP	Jump if parity		PF == 1
JPE	Jump if parity even		PF == 1
JNP	Jump if not parity		PF == 0
JPO	Jump if parity odd		PF == 0
JCXZ	Jump if CX register is 0		CX == 0
JECXZ	Jump if ECX register is 0		ECX == 0

1.1.9 Push and Pop

Instead of using registers we can also use the stack to save and access variables. The two most common variables used for this are:

Push: Instructs the processor to store the value of the oprand onto a stack and decrements the stack pointer.

and

POP: Retrives values stored on the stack. Operand for this instruction is the destination where the value should be stored. The instruction also increments the stack pointer.

These two commands are most often seen being used to save registers for function calls:

```
push ebx ; save EBX register on stack

; Make changes to EBX register

pop ebx ; restore EBX register from stack
```

1.2 Opcodes

1.2.1 What are opcodes?

Opcodes are our programs operation codes and are all 1 byte in size, these instructions then have between 1 to 4 bytes for operands available.

1.2.2 Opcode Table

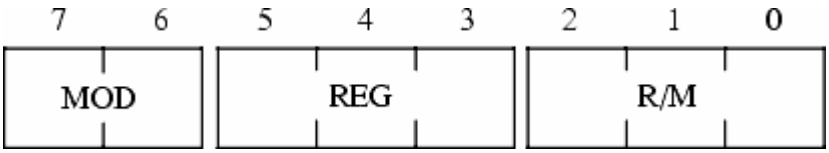
In most cases an instruction can be split into several different opcodes. This allows us to do instructions for different sized data as well as instructions based on certain registers.

Other opcodes can however represent several different instructions based on it's operands, such as the D1 opcode which can represent ROL, ROR, RCL, RCR, SHL, SAL, SHR, SAL, SHL, and SAR.

To look up all of the codes for x86 you can use the following site: http://ref.x86asm.net/coder32.html.

1.2.3 MOD-REG-R/M Byte

x86 uses the MOD-REG-R/M Byte in order to encode multiple registers into one single byte. This byte is segmented into three sections: the mod section (high 2 bits), the reg section (next 3 bits), and then the r/m section (final 3 bits).



Using these values as well as the size of the data we can get all register combinations within one byte:

16-bit ModR/M Byte

r8(/r)		AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)		AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)		EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)		MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)		XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg		ES	CS	SS	DS	FS	GS	res.	res.
eee		CR0	invd	CR2	CR3	CR4	invd	invd	invd
eee		DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7
(In decimal) /digit (Opcode)		0	1	2	3	4	5	6	7
(In binary) REG =		000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/ST0/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/ST1/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/ST2/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/ST3/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/ST4/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/ST5/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/ST6/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/ST7/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

32-bit ModR/M Byte

r8(/r)		AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)		AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)		EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)		MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)		XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg		ES	CS	SS	DS	FS	GS	res.	res.
eee		CR0	invd	CR2	CR3	CR4	invd	invd	invd
eee		DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7
(In decimal) /digit (Opcode)		0	1	2	3	4	5	6	7
(In binary) REG =		000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[*sib*]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[*sib*]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[*sib*]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/ST0/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/ST1/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/ST2/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/ST3/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/ST4/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/ST5/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/ST6/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/ST7/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF