CSCE 312: Computer Organization

David Kebo Houngninou

Machine Language

What is Machine Language?

A machine language is an agreed-upon formalism, designed to code low-level programs as series of machine instructions.

Machine instructions can:

1. Perform arithmetic and logic operations in the processor
2. Load values from memory
3. Store values to memory
4. Move values from one register to another
5. Test Boolean conditions etc.

A compiler generates machine code based on:

1. Rules of the programming language
2. The instruction set of the target machine.

Why knowing machine language?

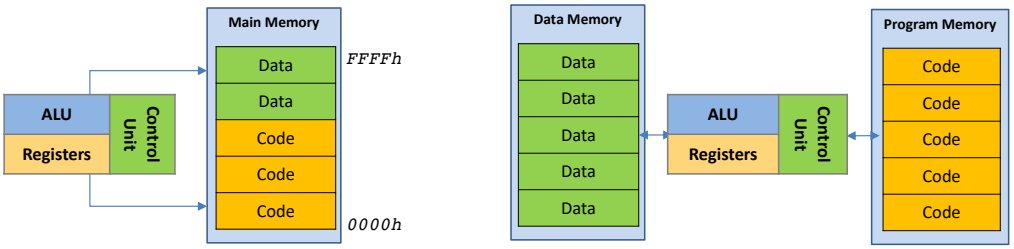
1. A must if you will write compilers or operating Systems for a living
2. Complete control over a system's resources: Performance!
3. Direct access to hardware, critical for security
4. Understanding processor and memory function
5. Transparent Execution (WYSIWYG). Easy to Debug

"machine language makes it possible to manipulate hardware directly, address critical issues concerning performance and also provide access to special instructions for processors. Uses of machine language include coding device drivers, real-time systems, low-level embedded systems, boot codes, reverse engineering and more" -Techopedia

Memory

Memory is a collection of hardware devices that store data and instructions in a computer.

A programmer views memory as a continuous array of cells of some fixed width, also called words or locations, each having a unique address.



Von Neumann Architecture
Shared address space for data and code

Harvard Architecture
Separate address space for data and code

Processor

The processor core includes registers, an Arithmetic Logic Unit and controllers.

The processor performs:
1. Arithmetic and logic operations
2. Memory access operations
3. Branching operations

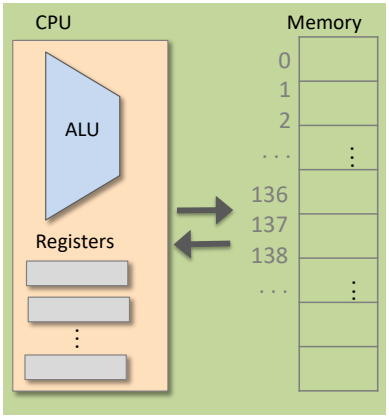
The operands of these operations are binary values.
The results of CPU operations are stored in registers or main memory

Registers

The register is a high-speed local memory and is the fundamental storage area in the processor

The features of the register are:
- Proximity to the processor
- Manipulate data and instructions quickly.
- Reduce the use of memory access commands to speed up program execution

The limitations of the registers are:
- Size in the order of bits
- Finite number of registers



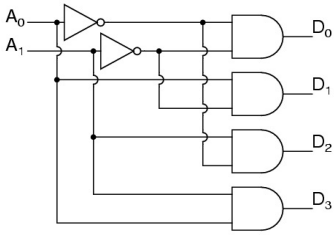
Registers

Most registers are general purpose and can store any type of information:
data – e.g. timer value, constants
address – e.g. ASCII table, stack

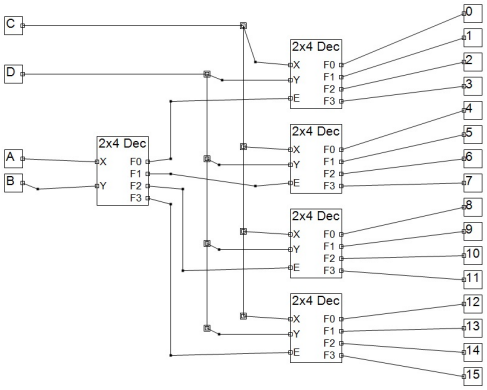
Some are reserved for specific purpose:
program counter (r15 in ARM)
program status register (CPSR in ARM)

Memory size vs. memory access

The larger the memory, the longer its address and therefore the wider (and slower) the decoder circuit.

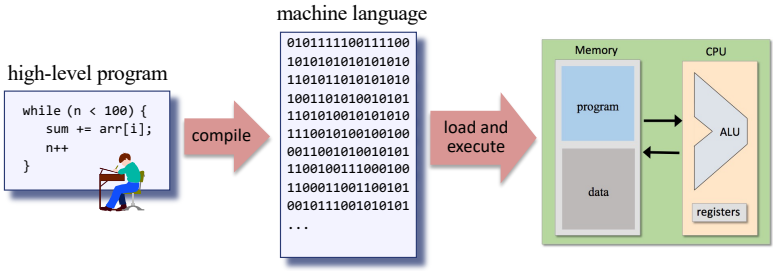


2 to 4 decoder for a 4 Block Memory



4 to 16 decoder for a 16 Block Memory

Compilation



How to read Mnemonics

Instructions:

1011	000011	000010
add	R3	R2

Interpretation 1:

- The symbolic form add R_3 R_2 doesn't really exist
- It is just a convenient mnemonic that can be used to present machine language instructions to humans.

How to read Mnemonics

Instructions:

1011	000011	000010
add	R3	R2

Interpretation 2:

- Allow humans to write symbolic machine language instructions, using assembly language
- Use an assembler program to translate the symbolic code into binary form.

Instruction classes

Instructions can be broadly separated into three basic classes:

1. Data Movement
   Memory load/store
   Register Transfers
2. Data Operation
   Arithmetic
   Logical
   Comparison and test
3. Flow Control
   Branch
   Conditional execution

Terminologies

Instruction set architecture: The parts of a processor design that one needs to understand or write assembly/machine code.

Intel: x86, IA32, Itanium, x86-64

ARM: Used in almost all mobile phones

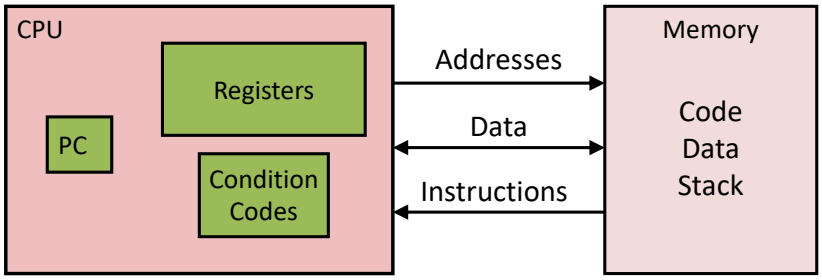
Microarchitecture: Implementation of the architecture.

e.g.: cache sizes and core frequency.

Machine Code: The byte-level programs that a processor executes

Assembly Code: A text representation of machine code

Assembly/Machine Code View



Programmer-Visible State

PC: Program counter

Address of next instruction

Register file

Heavily used program data

Condition codes

Store status information about most recent arithmetic or logical operation.

Used for conditional branching.

Memory

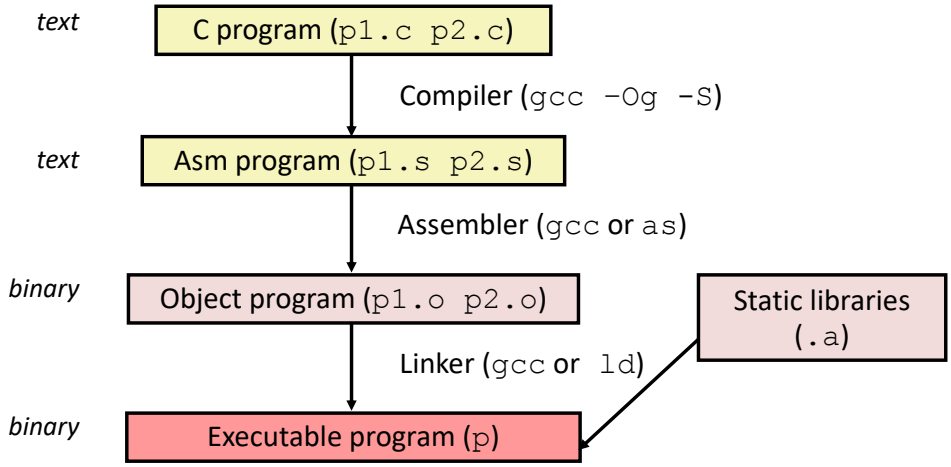
- Byte addressable array
- Code and user data
- Stack to support procedures

From C to Object Code

Code in files **`p1.c p2.c`**

Compile with command: **`gcc –Og p1.c p2.c -o p`**

Use basic optimizations (**`-Og`**) [New to recent versions of GCC]



Machine Instruction Example

```
*dest = t;
```

C Code

Store value **t** where designated by **dest**

```
movq %rax, (%rbx)
```

Assembly

Move 8-byte value to memory

Operands:

t: Register **%rax**

dest: Register **%rbx**

***dest:** Memory **M[%rbx]**

```
0x40059e: 48 89 03
```

Object Code

3-byte instruction

Stored at address **0x40059e**

Compiling Into Assembly (text)

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

`gcc -Og -S sum.c the` Produces file `sum.s`

Assembly to Object Code (Binary)

Code for sumstore

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

Assembler

- Translates `.s` into `.o`
- Binary encoding of each instruction

Linker

- Resolves references between files
- Combines with static run-time libraries

e.g., code for **`malloc`**, **`printf`**

- Linking occurs when program begins execution

Disassembling Object Code to Assemble

```
0000000000400595 <sumstore>:
  400595:  53               push   %rbx
  400596:  48 89 d3         mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff   callq  400590 <plus>
  40059e:  48 89 03         mov    %rax,(%rbx)
  4005a1:  5b               pop    %rbx
  4005a2:  c3               retq
```

Disassembler

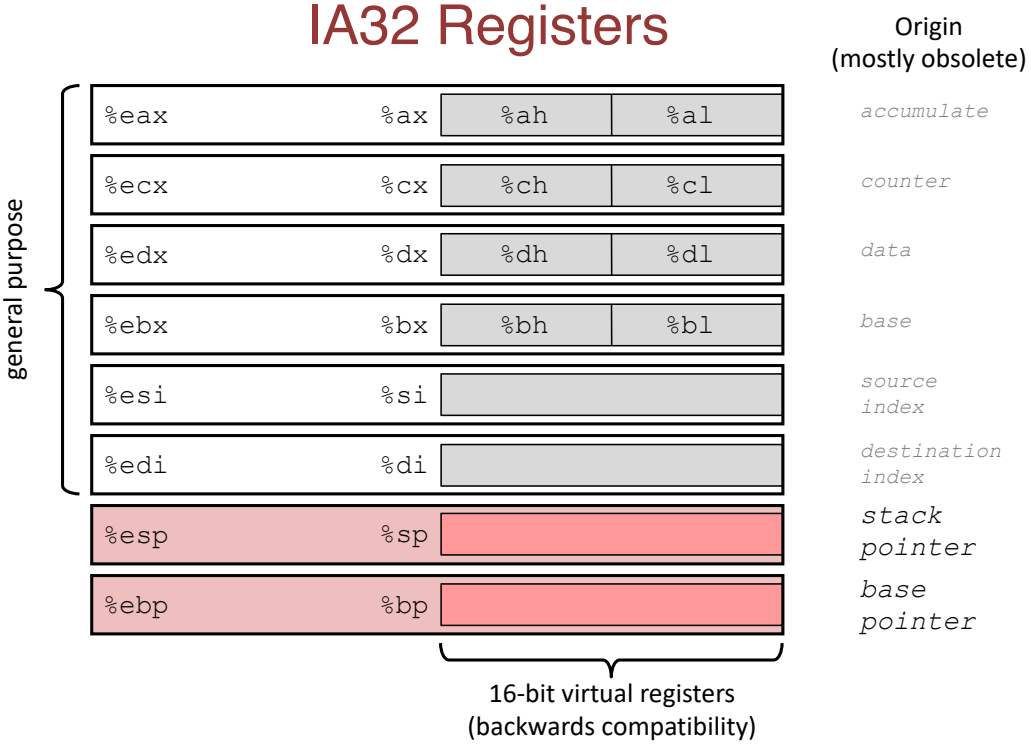
`objdump -d sum`

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either `a.out` (complete executable) or `.o` file

IA 32 registers

- An IA32 CPU has 08 registers of 32-bit each.
- These registers store integer data as well as pointers.
- The register names all begin with %e.
- The first six registers are general-purpose registers.
- The last two registers (%ebp and %esp) contain pointers.
- The low-order 2 bytes of the first four registers can be independently read or written by the byte operation instructions.

IA32 Registers



x86-64 Integer Registers

64	31	64	31
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

x86-64 Integer Registers (another view)

63	31	15	8 7	0	
%rax	%eax	%ax	%ah	%al	Return value
%rbx	%ebx	%bx	%bh	%bl	Callee saved
%rcx	%ecx	%cx	%ch	%cl	4th argument
%rdx	%edx	%dx	%dh	%dl	3rd argument
%rsi	%esi	%si		%sil	2nd argument
%rdi	%edi	%di		%dil	1st argument
%rbp	%ebp	%bp		%bpl	Callee saved
%rsp	%esp	%sp		%spl	Stack pointer
%r8	%r8d	%r8w		%r8b	5th argument
%r9	%r9d	%r9w		%r9b	6th argument
%r10	%r10d	%r10w		%r10b	Callee saved
%r11	%r11d	%r11w		%r11b	Used for linking
%r12	%r12d	%r12w		%r12b	Unused for C
%r13	%r13d	%r13w		%r13b	Callee saved
%r14	%r14d	%r14w		%r14b	Callee saved
%r15	%r15d	%r15w		%r15b	Callee saved

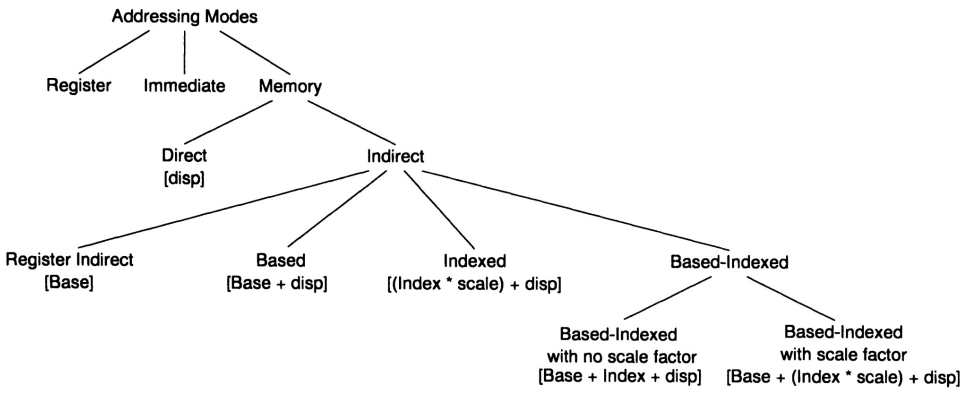
Textbook figure 3.2

Operands and addressing modes

Operands can be immediate constants, register values, or values from memory. The scaling factors must be either 1, 2, 4, or 8.

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$\mathrm{R}[\mathrm{E}_a]$	Register
Memory	Imm	$\mathrm{M}[Imm]$	Absolute
Memory	(E_a)	$\mathrm{M}[\mathrm{R}[\mathrm{E}_a]]$	Indirect
Memory	$Imm(\mathrm{E}_b)$	$\mathrm{M}[Imm + \mathrm{R}[\mathrm{E}_b]]$	Base + displacement
Memory	$(\mathrm{E}_b, \mathrm{E}_i)$	$\mathrm{M}[\mathrm{R}[\mathrm{E}_b] + \mathrm{R}[\mathrm{E}_i]]$	Indexed
Memory	$Imm(\mathrm{E}_b, \mathrm{E}_i)$	$\mathrm{M}[Imm + \mathrm{R}[\mathrm{E}_b] + \mathrm{R}[\mathrm{E}_i]]$	Indexed
Memory	$(, \mathrm{E}_i, s)$	$\mathrm{M}[\mathrm{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm(, \mathrm{E}_i, s)$	$\mathrm{M}[Imm + \mathrm{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed
Memory	$(\mathrm{E}_b, \mathrm{E}_i, s)$	$\mathrm{M}[\mathrm{R}[\mathrm{E}_b] + \mathrm{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathrm{E}_b, \mathrm{E}_i, s)$	$\mathrm{M}[Imm + \mathrm{R}[\mathrm{E}_b] + \mathrm{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed

Addressing Modes



Address Computation

Here are some practical examples:

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Effective Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + (4*0x100)`	`0xf400`
`0x80(,%rdx,2)`	`(2*0xf000) + 0x80`	`0x1e080`

Activity 3

Assume the values in table A are stored at the indicated memory addresses and registers.
Fill in table C showing the values for the indicated operands

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Table A

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Table B

Operand	Value
%eax	
0x104	
$0x108	
(%eax)	
$0xAC	
4(%eax)	
9(%eax,%edx)	
260(%ecx,%edx)	
0xFC(,%ecx,4)	
(%eax,%edx,4)	

Table C

Activity 3 solution

Assume the values in table A are stored at the indicated memory addresses and registers.
Fill in table C showing the values for the indicated operands

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Table A

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Table B

Operand	Value
%eax	0x100
0x104	0xAB
$0x108	0x108
(%eax)	0xFF
$0xAC	0xAC
4(%eax)	0xAB
9(%eax,%edx)	0x11
260(%ecx,%edx)	0x13
0xFC(,%ecx,4)	0xFF
(%eax,%edx,4)	0x11

Table C

Instruction classes

Instructions can be broadly separated into three basic classes:

1. Data Movement
   - Memory load/store
   - Register Transfers
2. Data Operation
   - Arithmetic
   - Logical
   - Comparison and test
3. Flow Control
   - Branch
   - Conditional execution

Data movement

An x86 CPU can address up to 2^{32} Bytes of memory.

The data movement instructions are:

- mov
- push
- pop

Data movement

Moving Data

`movq Source, Dest:`

Operand Types

Immediate: Constant integer data

- Example: `$0x400`, `$-533`
- Like C constant, but prefixed with `'$'`

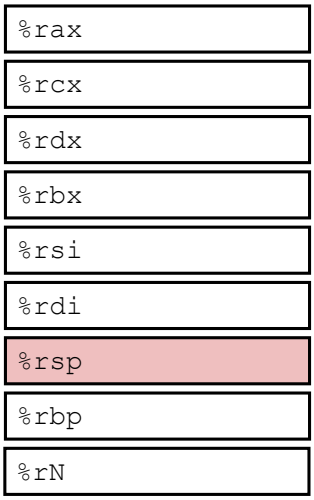
Register: One of 16 integer registers

- Example: `%rax`, `%r13`
- But `%rsp` reserved for special use

Memory:

8 consecutive bytes of memory at address given by register

- Example: `(%rax)`
- Various other “address modes”



The mov instruction

The mov instruction copies the data referred in the first operand to the location referred in the second operand.

What types of mov operations can you make?

register-to-register: yes ☺

memory-to-register: yes ☺

memory-to-memory: no ☹

Syntax:	`mov <reg>, <reg>`
	`mov <reg>, <mem>`
	`mov <mem>, <reg>`
	`mov <imm>, <reg>`
	`mov <imm>, <mem>`

Examples:	`mov %ebx, %eax /*copy the value in EBX into EAX*/` `movb $5, var(,1) /*store the value 5 into the byte at location var*/`

mov instruction: example

The mov instruction moves a source to destination

mov (%ebx), %eax	Load 4 bytes from the memory address in EBX to EAX.
mov %ebx, var(,1)	Move the contents of EBX to the 4 bytes at memory address var. (var is a 32-bit constant)
mov -4(%esi), %eax	Move 4 bytes at memory address ESI + (-4) to EAX.
mov %cl, (%esi,%eax,1)	Move the contents of CL to the byte at address ESI+EAX.
mov (%esi,%ebx,4), %edx	Move the 4 bytes of data at memory address ESI+4*EBX to EDX.

The push instruction

The push instruction places its operand on the top of the hardware stack in memory.

push first decrements the stack pointer (ESP) by 4, then places its operand into the contents of the 32-bit location at address (%esp).

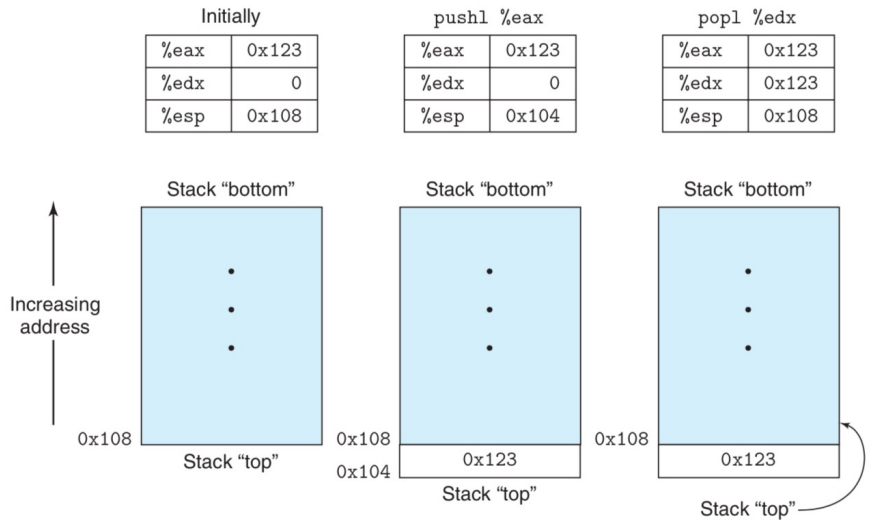
push decrements the ESP because the x86 stack grows down.

Note: A stack grows from high addresses to lower addresses.

Syntax:	`push <reg32>`
	`push <mem>`
	`push <con32>`

Examples:	`push %eax /* push eax on the stack */` `push var(,1) /* push the 4 bytes at address` *var* `onto the stack */`

Stack operations



The pop instruction

The pop instruction removes the 4-byte data element from the top of the hardware stack into the specified operand (register or memory location).

It first moves the 4 bytes located at memory location (%esp) into the specified register or memory location, and then increments ESP by 4.

Syntax:	`pop <reg32>` `pop <mem>`
Examples:	`pop %edi /*pop the top element of the stack into EDI*/` `pop (%ebx) /*pop the top element of the stack into memory at the four bytes starting at location EBX.*/`

x-64 instructions

"byte" refers to a one-byte integer (suffix b) 8-bit

"word" refers to a two-byte integer (suffix w) 16-bit

"doubleword" refers to a four-byte integer (suffix l) 32-bit

"quadword" refers to an eight-byte value (suffix q) 64-bit

Instructions, like mov, use a suffix to show how large the operands is.
e.g., `movq %rax, %rbx` /*move a quadword from %rax to %rbx*/

The movq instruction

Copy a quadword from the source operand to the destination operand

	Source	Dest	Src, Dest	C
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

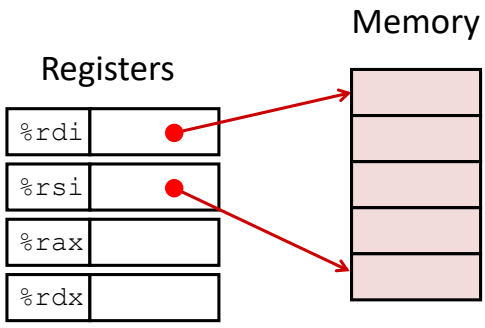
Example of Addressing Modes: Swap()

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

Example of Addressing Modes: Swap()

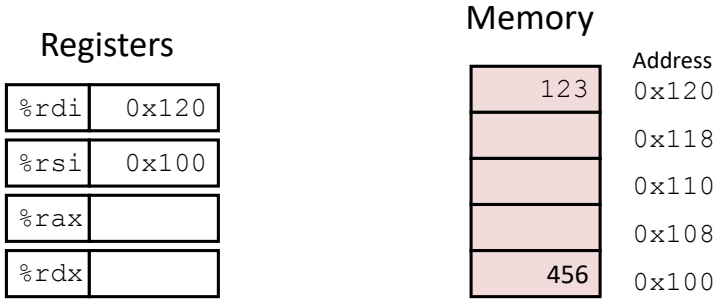
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()

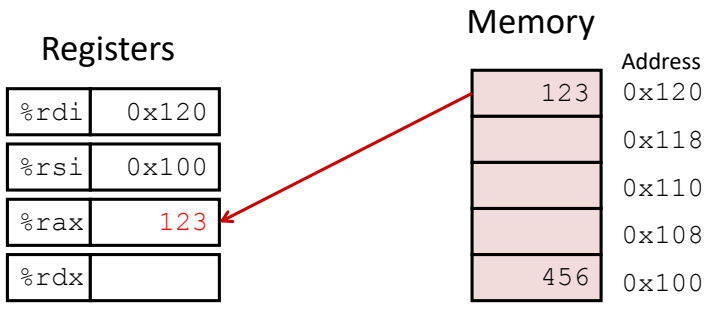



```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

REGISTERS	
%rdi	0x126
%rsi	0x300
%rax	0x200
%rdx	0x312

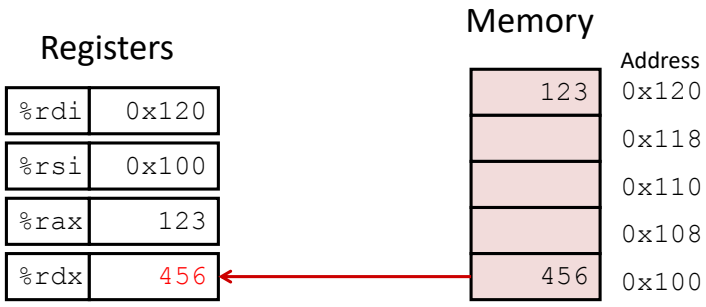
MEMORY	
Address	Content
0x100	
0x108	
0x110	
0x118	
0x120	

Understanding Swap()



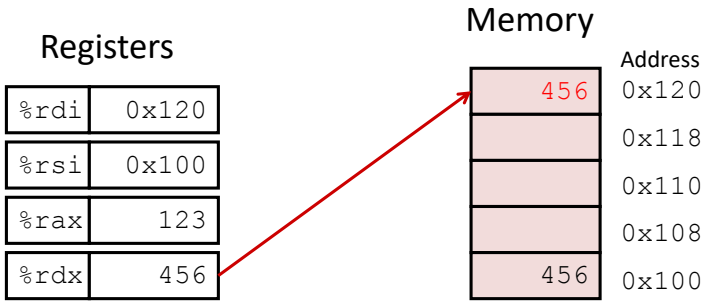

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding Swap()



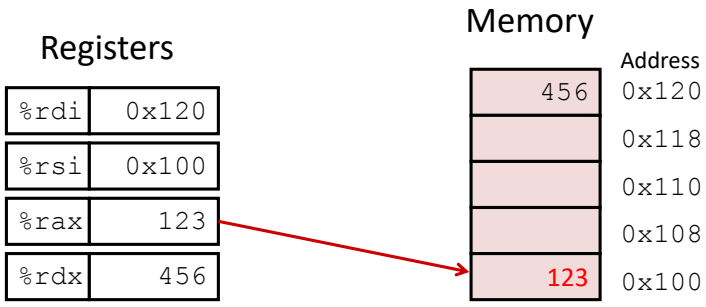

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding Swap()




```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding `Swap()`




```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Instruction classes

Instructions can be broadly separated into three basic classes:

1. Data Movement
   - Memory load/store
   - Register Transfers
2. Data Operation
   - Arithmetic
   - Logical
   - Comparison and test
3. Flow Control
   - Branch
   - Conditional execution

Data operation

Arithmetic Operations

1. Unary Operations
2. Binary operations
3. Shift operations

Comparison and Test Instructions

Data operation: unary

A unary operation is a one-operand instruction.

Instruction	Operation	Description
inc dst	dst = dst + 1	Increment
dec dst	dst = dst - 1	Decrement
neg dst	dst = -dst	Negate
not dst	dst = ~dst	Bitwise complement

The inc and dec instructions

The inc instruction increments the contents of its operand by one.
The dec instruction decrements the contents of its operand by one.

Syntax:	`inc <reg>`
	`inc <mem>`
	`dec <reg>`
	`dec <mem>`

Examples:	`dec %eax – subtract one from the contents of EAX` `incl var(,1) – add one to the 32-bit integer stored at location var`

The neg instruction

The neg instruction Performs the two's complement negation of the operand contents.

Syntax:	`neg <reg>` `neg <mem>`
Examples:	`neg %eax – EAX is set to (- EAX)`

The not instruction

The not instruction logically negates the operand contents (flips all bit values in the operand).

Syntax:	`not <reg>` `not <mem>`
Examples:	`not %eax – flip all the bits of EAX`

Data operation: binary

A binary operation is a two-operand instruction.

Instruction	Operation	Description
lea S, D	D = address S	Load effective address of source into destination
add S, D	D = D + S	Add source to destination
sub S, D	D = D - S	Subtract source from destination
imul S, D	D = D * S	Multiply destination by source (signed)
idiv S, D	D = D / S	Divide destination by source (signed)
xor S, D	D = D $\oplus$ S	Bitwise XOR destination by source
or S, D	D = D $\vee$ S	Bitwise OR destination by source
and S, D	D = D $\wedge$ S	Bitwise AND destination by source

The lea instruction

The lea instruction places the address specified by its first operand into the register specified by its second operand.

Note: the contents of the memory location are not loaded, only the effective address is computed and placed into the register.

This is useful to get a pointer into a memory region or to perform simple arithmetic operations.

Syntax:	`lea <mem>, <reg32>`
Examples:	`lea (%ebx,%esi,8), %edi – the quantity EBX+8*ESI is placed in EDI.` `lea val(,1), %eax – the value val is placed in EAX.`

The add instruction

The add instruction adds together its two operands, storing the result in its second operand.

Note:

- Both operands may be registers.
- At most one operand may be a memory location.

Syntax:	`add <reg>, <reg>`
	`add <mem>, <reg>`
	`add <reg>, <mem>`
	`add <con>, <reg>`
	`add <con>, <mem>`

Examples:	`add $10, %eax – EAX is set to EAX + 10` `addb $10, (%eax) – add 10 to the single byte stored at memory address stored in EAX`

The sub instruction

The sub instruction subtracts the first operand from the second operand and stores the result in the second operand, storing the result in its second operand.

Note:

Both operands may be registers.

At most one operand may be a memory location.

Syntax:	sub <reg>, <reg>
	sub <mem>, <reg>
	sub <reg>, <mem>
	sub <con>, <reg>
	sub <con>, <mem>

Examples:	sub %ah, %al – AL is set to AL - AH sub $216, %eax – subtract 216 from the value stored in EAX

The imul instruction

The imul instruction can have two operands or three operands

2-operands: multiply the two operands together and stores the result in the second operand.

3-operands: multiply the second and third operands and store the result in the last operand.

Syntax:	imul <reg32>, <reg32> imul <mem>, <reg32> imul <con>, <reg32>, <reg32> imul <con>, <mem>, <reg32>
Examples:	imul (%ebx), %eax – multiply the contents of EAX by the 32-bit contents of the memory at location EBX. Store the result in EAX. imul $25, %edi, %esi – ESI is set to EDI * 25

The idiv instruction

The idiv instruction divides the contents of the 64 bit integer EDX:EAX by the specified operand value.

The quotient result of the division is stored into EAX, and the remainder is stored in EDX.

Syntax:	idiv <reg32> idiv <mem>
Examples:	idiv %ebx - divide the contents of EDX:EAX by the contents of EBX. Place the quotient in EAX and the remainder in EDX. idivw (%ebx) – divide the contents of EDX:EAS by the 32-bit value stored at the memory location in EBX. Place the quotient in EAX and the remainder in EDX.

The and, or, xor instructions

These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.

Syntax:	and | or | xor <reg>, <reg> and | or | xor <mem>, <reg> and | or | xor <reg>, <mem> and | or | xor <con>, <reg> and | or | xor <con>, <mem>
Examples:	and $0x0f, %eax – clear all but the last 4 bits of EAX. xor %edx, %edx – set the contents of EDX to zero.

Data operation: shift

A shift operation moves the bits in a binary number.

Instruction	Description
shl	Logical left shift destination by k bits
shr	Logical right shift destination by k bits

The shl, shr instructions

The shl and shr instructions logically shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros.

The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL.

Syntax:	`shl | shr <con8>, <reg>`
	`shl | shr <con8>, <mem>`
	`shl | shr %cl, <reg>`
	`shl | shr %cl, <mem>`

Examples:	`shl $1, eax - Multiply the value of EAX by 2 (if the most significant bit is 0)` `shr %cl, %ebx - Store in EBX the floor of result of dividing the value of EBX by 2n (n is the value in CL).`

Comparison and Test Instructions

A shift operation moves the bits in a binary number.

Instruction	Description
cmp S_2 , S_1	Set condition codes according to S_1 - S_2
tst S_2 , S_1	Set condition codes according to S_1 & S_2

The cmp instruction

The cmp instruction compares the values of the two specified operands, setting the condition codes in the machine status word appropriately.

This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand.

Syntax:	`cmp <reg>, <reg>`
	`cmp <mem>, <reg>`
	`cmp <reg>, <mem>`
	`cmp <con>, <reg>`

Examples:	`cmpb $10, (%ebx)`
	`jeq loop`
	`If the byte stored at the memory location in EBX is equal to the integer constant 10, jump to the location labeled loop.`