

# 198:211 Computer Architecture

## Lecture 11 Fall 2010

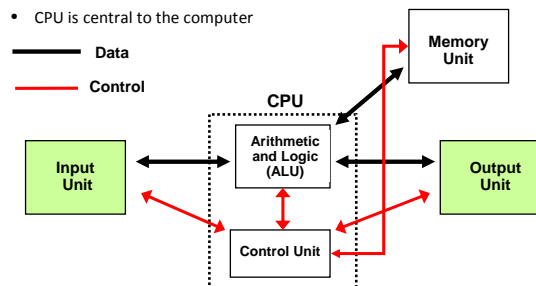
- Topics: Chapter 3
  - Assembly Language 3.2
  - Register Transfer 3.4
  - ALU 3.5

## Assembly level Programming

- We are now familiar with high level programming languages such as C and Java
- Computers execute machine code
- Compilers generate machine code from source code
- We need to understand how the code actually executes on various machines (architectures)
  - Studying binary code is not very easy
- At the same time, we need a language that mimics the machine level instructions and still readable (textual format)
- Enter Assembly: compilers can generate an intermediate step of code called assembly code
- Assemblers then convert assembly code to machine code

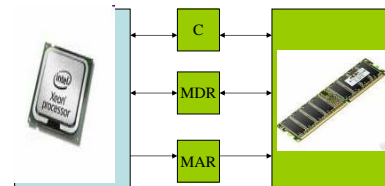
## (recall) Von Neumann Architecture

- Model of a computer that used stores programs
  - Both Data and Program stored in memory
  - Allows the computer to be "Re-programmed"
- CPU is central to the computer



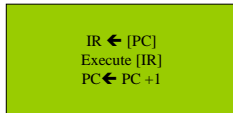
## Simplified hardware view

- Store: Store contents of MDR to Memory address specified by MAR
- Load: Load into MDR contents of Memory address specified by MAR



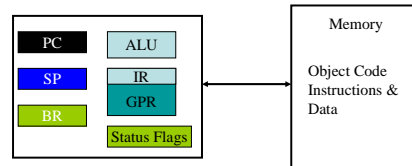
MAR: Memory Address Register also known as Program Counter  
MDR: Memory Data Register  
C: Control Switch; 0 is Load Or 1 is Store

## Fetch-execute cycle



- Notation: [X] or (X) contents stored at location (memory address) contained in Register X
- IR executes the instructions
- Where are the operands?
- As part of the execution, data may be transferred among various registers in the CPU as well as memory
- Typical data movement instruction is MOV

## Assembly programmer view



- ALU: Arithmetic Logic Unit
- IR: Instruction register
- GPR: General Purpose Registers
- PC: Program Counter
- SP: Stack Pointer
- BR: Base Register

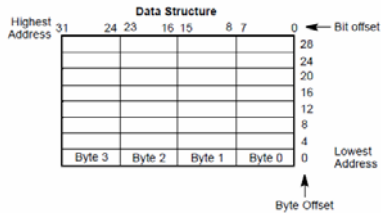
## MOV instruction

- Most common instruction is data transfer instruction
  - Notation: `mov S, D` (contents of S becomes contents of D)
- `mov` data from memory to register and register to memory
- `mov` data between registers
  - Notation: registers are preceded by % sign
- `mov` data to and from stack
- `mov` constants to registers
  - Notation: constant preceded by \$

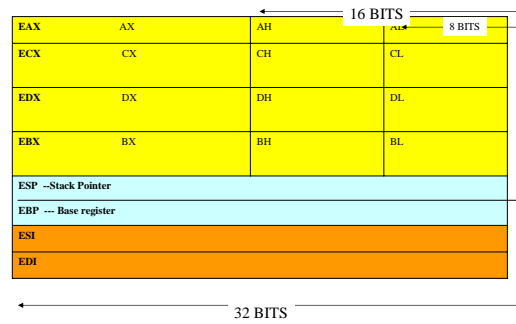
## Data formats

- Data is represented in different sizes
- Byte ... 8 bits
  - E.g., Char
- Word .. 16 bits (2 bytes)
  - E.g., Short int
- Double Word (long or dword).. 32 bits ( 4 bytes)
  - E.g., float
- QWORD .. 64 bits (8 bytes)
  - E.g., double
- Instructions can operate on any data size
  - `movl, movw, movb`
  - Operates on doubleword, word, byte respectively

## Bit and byte order



## x86 General purpose registers (8)



## Registers

- Registers are 32 bit (operations can access 16 bits, 8 bits within the register)
  - Operations involving registers are typically single cycle (nano seconds)...
- Types of registers
- Data registers (EAX, EBX, ECX, EDX)
  - Holds operands
- Pointer and Index registers (EBP, ESP, EIP,ESI,EDI)
  - Holds references to addresses as well as indexes
- Segment registers
  - Holds starting address of program segments (CS,DS,SS,ES)

## Assembly Language Characteristics

- Minimal Data Types
  - "Integer" data of 1, 2, or 4 bytes
    - Data values
    - Addresses (untyped pointers)
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures
    - Just contiguously allocated bytes in memory
- Primitive Operations
  - Perform arithmetic function on register or memory data
  - Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
  - Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches

## Assembly level instructions

- Opcode byte | Addressing byte | Other bytes
- Instruction length varies from 1 byte to several bytes
- Instruction consists of
  - First byte is the Opcode byte and has short names called mnemonic
    - Opcode byte is typically 1 byte
    - More recent instructions set (floating point, multimedia, parallel) have multiple opcode bytes
  - Second byte is the Addressing byte (for data handling instructions)
  - Specifies the type of Operands which one is a register, a memory and the type of addressing
  - For many instructions operands are implicitly specified by designated registers
    - push

## Machine Instruction Example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression  
x += y

```
addl 8(%ebp),%eax
```

```
0x401046: 03 45 08
```

- C Code
  - Add two signed integers
- Assembly
  - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    - x: Register %eax
    - y: Memory M[%ebp+8]
    - t: Register %eax
      - Return function value in %eax
- Object Code
  - 3-byte instruction
  - Stored at address 0x401046

## Sample code in C, assembly, machine code on a SUN Sparc System

```
#include <stdio.h>
#include <stdlib.h>
main()
{ int x=1,
  y=2;
  int sum;
  sum=x+y;
}

mov    1, %o0
st     %o0, [%fp-20]
mov    2, %o0
st     %o0, [%fp-24]
ld     [%fp-20], %o0
ld     [%fp-24], %o1
add    %o0, %o1, %o0
st     %o0, [%fp-28]
```

```
1059c: 90 10 20 01  mov 1, %o0
105a0: d0 27 bf ec  st %o0, [%fp-20]
105a4: 90 10 20 02  mov 2, %o0
105a8: d0 27 bf e8  st %o0, [%fp-24]
105ac: d0 07 bf ec  ld [%fp-20], %o0
105b0: d2 07 bf e8  ld [%fp-24], %o1
105b4: 90 02 00 09  add %o0, %o1, %o0
105b8: d0 27 bf e4  st %o0, [%fp-28]
```

gcc -s sum.c

objdump -d sum.o > machinecode.txt

## Sample code in C, assembly, machine code on an intel x86 machine

```
#include <stdio.h>
#include <stdlib.h>
main()
{ int x=1,
  y=2;
  int sum;
  sum=x+y;
}

movl   $1, -4(%ebp)
movl   $2, -8(%ebp)
movl   -8(%ebp), %eax
addl   -4(%ebp), %eax
movl   %eax, -12(%ebp)

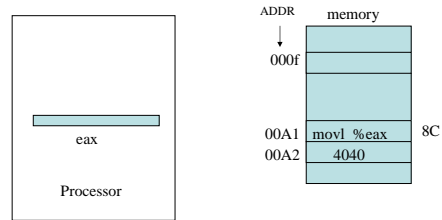
8048350: c7 45 fc 01 00 00 00  movl $0x1,0xffffffff(%ebp)
8048357: c7 45 f8 02 00 00 00  movl $0x2,0xffffffff8(%ebp)
804835e: 8b 45 f8             mov  0xffffffff8(%ebp),%eax
8048361: 03 45 fc             add  0xffffffffc(%ebp),%eax
8048364: 89 45 f4             mov  %eax,0xffffffff4(%ebp)
```

## Addressing byte

- The second byte indicates how to get the data
- Operand Types
  - Immediate: Constant integer data
    - Like C constant, but prefixed with '\$'
    - E.g., \$0x400, \$-533
    - Encoded with 1, 2, or 4 bytes
  - Register: One of 8 integer registers
    - But %esp and %ebp reserved for special use
    - Others have special uses for particular instructions
  - Memory: 4 consecutive bytes of memory
    - Various "address modes"

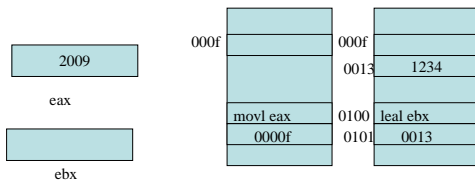
## Immediate addressing

- Operand is immediate
  - Operand value is found immediately following the instruction
  - movl \$x4040, %eax



## Direct addressing

- Address of operand is found immediately after the instruction
  - Also known as direct addressing or absolute address
  - movl x000F, %eax
  - leal x0013, %ebx
  - Load effective address; load address directly

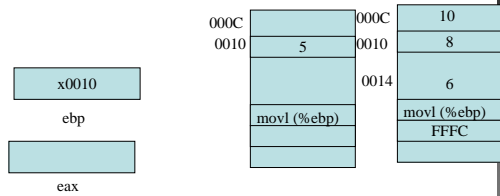


## Register-mode addressing

- One or more operands are found in registers
  - movl %eax, %ebx
  - Move contents of eax to ebx
  - add %ebx, %eax
  - Add contents of ebx and eax and store result in eax
  - mov %ea, %eb
  - Move 16 bit LSB of eax to 16 bit LSB of ebx
  - mov %al, %bl
  - Move 8 bit LSB of eax to 8 bit LSB of ebx

## Indirect mode addressing

- Content of operand is an address
- Offset can be specified as immediate mode
  - `movl (%ebp), %eax`
  - `movl -4(%ebp), %eax`



## Indexed mode addressing

- Consider the problem of initializing an array `a[5]` to 0
- One could use `movl $0, x1040, movl, $0, x1044, ...`
- Each instruction is a repeat of the other except the address
  - E.g., `for loop`
- Execute the same instruction with a different operand each time
- Alter the address of operand by adding the contents of another register
  - `movl (eab,esi), %eax`
  - `movl 8(eab,esi), %eax`

## Scaled index mode

- Same piece of code need to iterate over multiple byte sizes
- `movl (eab,esi,4), %eax`
  - Move content of address= $eab+esi*4$  to `eax`
- `leal (%edx,%edx,4), %eax`
  - Copy effective address =  $%edx + %edx*4=5*%edx$  to `eax`

## Address Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x100</code>

Expression	Computation	Address
<code>0x8(%edx)</code>	$0xf000 + 0x8$	<code>0xf008</code>
<code>(%edx,%ecx)</code>	$0xf000 + 0x100$	<code>0xf100</code>
<code>(%edx,%ecx,4)</code>	$0xf000 + 4*0x100$	<code>0xf400</code>
<code>0x80(,%edx,2)</code>	$2*0xf000 + 0x80$	<code>0x1e080</code>

## movl Operand Combinations

	Source	Destination	C Analog
movl	Imm	Reg	movl \$0x4,%eax    temp = 0x4;
		Mem	movl \$-147,(%eax)    *p = -147;
	Reg	Reg	movl %eax,%edx    temp2 = temp1;
		Mem	movl %eax,(%edx)    *p = temp;
	Mem	Reg	movl (%eax),%edx    temp = *p;
	<ul style="list-style-type: none"> <li>Cannot do memory-memory transfers with single instruction</li> </ul>		

## Stack operations

- %esp contains the address of top of stack
  - pushl %eax, popl %ebx
- Pushl stores operand in top of stack
- Popl stores top of stack in destination operand



## Using Simple Addressing Modes

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

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Annotations for the assembly code:

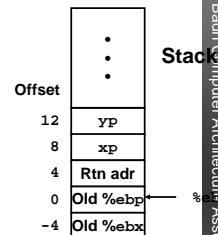
- Set Up: pushl %ebp, movl %esp,%ebp, pushl %ebx
- Body: movl 12(%ebp),%ecx, movl 8(%ebp),%edx, movl (%ecx),%eax, movl (%edx),%ebx, movl %eax,(%edx), movl %ebx,(%ecx)
- Finish: movl -4(%ebp),%ebx, movl %ebp,%esp, popl %ebp, ret

## Understanding Swap

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

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx
```



### Understanding Swap

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Offset	Address	
	123 0x124	
	456 0x120	
		0x11c
		0x118
		0x114
YP 12	0x120	0x110
xp 8	0x124	0x10c
4	Rtn adr	0x108
%ebp → 0		0x104
-4		0x100

```

movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx

```

@2010 Badri Computer Architecture Assembly 29

### Understanding Swap

%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Offset	Address	
	123 0x124	
	456 0x120	
		0x11c
		0x118
		0x114
YP 12	0x120	0x110
xp 8	0x124	0x10c
4	Rtn adr	0x108
%ebp → 0		0x104
-4		0x100

```

movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx

```

@2010 Badri Computer Architecture Assembly 30

### Understanding Swap

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Offset	Address	
	123 0x124	
	456 0x120	
		0x11c
		0x118
		0x114
YP 12	0x120	0x110
xp 8	0x124	0x10c
4	Rtn adr	0x108
%ebp → 0		0x104
-4		0x100

```

movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx

```

@2010 Badri Computer Architecture Assembly 31

### Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Offset	Address	
	123 0x124	
	456 0x120	
		0x11c
		0x118
		0x114
YP 12	0x120	0x110
xp 8	0x124	0x10c
4	Rtn adr	0x108
%ebp → 0		0x104
-4		0x100

```

movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx

```

@2010 Badri Computer Architecture Assembly 32

### Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Address	123	0x124
	456	0x120
		0x11c
		0x118
		0x114
Offset		
YP	12	0x120
XP	8	0x124
	4	Rtn adr
%ebp	→ 0	0x108
	-4	0x104
		0x100

```

movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax   # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax,(%edx)   # *xp = eax
movl %ebx,(%ecx)   # *yp = ebx
  
```

33

### Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Address	456	0x124
	456	0x120
		0x11c
		0x118
		0x114
Offset		
YP	12	0x120
XP	8	0x124
	4	Rtn adr
%ebp	→ 0	0x108
	-4	0x104
		0x100

```

movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax   # eax = *yp (t1)
movl (%edx),%ebx   # ebx = *xp (t0)
movl %eax,(%edx)  # *xp = eax
movl %ebx,(%ecx)   # *yp = ebx
  
```

34

### Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Address	456	0x124
	123	0x120
		0x11c
		0x118
		0x114
Offset		
YP	12	0x120
XP	8	0x124
	4	Rtn adr
%ebp	→ 0	0x108
	-4	0x104
		0x100

```

movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax   # eax = *yp (t1)
movl (%edx),%ebx   # ebx = *xp (t0)
movl %eax,(%edx)   # *xp = eax
movl %ebx,(%ecx)  # *yp = ebx
  
```

35

### Some Arithmetic Operations

Format	Computation	
• Two Operand Instructions		
addl Src, Dest	Dest = Dest + Src	
subl Src, Dest	Dest = Dest - Src	
imull Src, Dest	Dest = Dest * Src	
sall Src, Dest	Dest = Dest << Src	Also called shll
sarl Src, Dest	Dest = Dest >> Src	Arithmetic
shrl Src, Dest	Dest = Dest >> Src	Logical
xorl Src, Dest	Dest = Dest ^ Src	
andl Src, Dest	Dest = Dest & Src	
orl Src, Dest	Dest = Dest   Src	

36

## Shift right

- sarl arithmetic right Shift
- sarl k, D
  - Shift right k bits (D), retaining MSB
  - sarl 1, 10001 → ?
- shril logical right shift
  - Shift right k bits (D), fill with 0s from the right
  - shril 1, 10001 → ?

## Some Arithmetic Operations: unary

Format                      Computation

- One Operand Instructions

incl Dest                      Dest = Dest + 1

decl Dest                      Dest = Dest - 1

negl Dest                      Dest = - Dest    *twos complement negation*

*Replaces the value of the operand with its twos complement (equivalent to subtracting from 0)*

notl Dest                      Dest = ~ Dest

*Bitwise negation of all the bits (each 1 is replaced by 0 and each 0 is replaced by 1)*

## Using leal for Arithmetic Expressions

```

arith:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax
    movl %ebp,%esp
    popl %ebp
    ret
    
```

} Set Up  
 } Body  
 } Finish

## Understanding arithmetic

```

int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
    
```

Offset	•	Stack
16	z	
12	y	
8	x	
4	Rtn adr	
0	Old %ebp	← %ebp

```

movl 8(%ebp),%eax      # eax = x
movl 12(%ebp),%edx     # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx # edx = 3*y
sall $4,%edx           # edx = 48*y (t4)
addl 16(%ebp),%ecx     # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax        # eax = t5*t2 (rval)
    
```

## Understanding arith

```
int arith
(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
# eax = x
movl 8(%ebp),%eax
# edx = y
movl 12(%ebp),%edx
# ecx = x+y (t1)
leal (%edx,%eax),%ecx
# edx = 3*y
leal (%edx,%edx,2),%edx
# edx = 48*y (t4)
sall $4,%edx
# ecx = z+t1 (t2)
addl 16(%ebp),%ecx
# eax = 4+t4+x (t5)
leal 4(%edx,%eax),%eax
# eax = t5*t2 (rval)
imull %ecx,%eax
```

## Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192, 2^{13} - 7 = 8185$

```
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

logical:

```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
movl %ebp,%esp
popl %ebp
ret
```

```
eax = x
eax = x^y (t1)
eax = t1>>17 (t2)
eax = t2 & 8185
```

} Set  
Up  
} Bod  
} Fin