



Assembly Language: Part 1

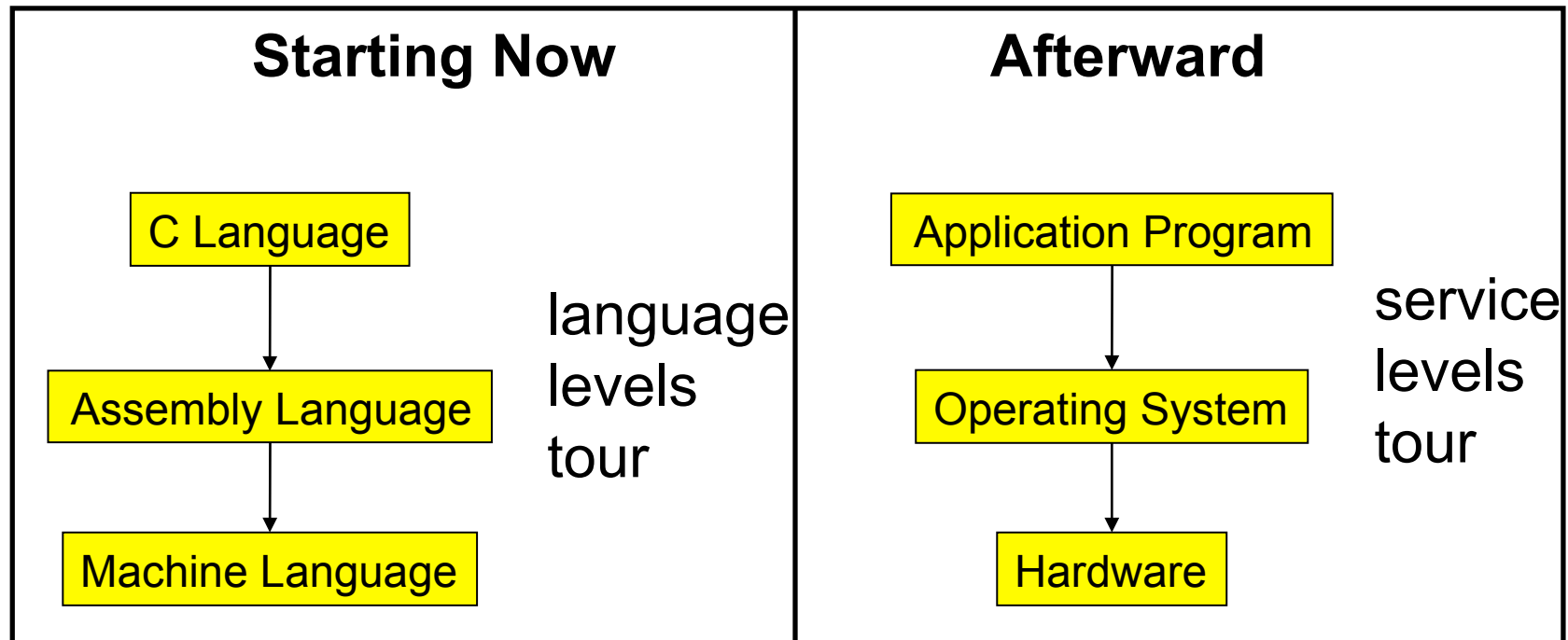




Context of this Lecture

First half lectures: “Programming in the large”

Second half lectures: “Under the hood”





Goals of this Lecture

Help you learn:

- Language levels
- The basics of IA-32 **architecture**
 - Enough to understand IA-32 assembly language
- The basics of IA-32 **assembly language**
 - Instructions to define global data
 - Instructions to transfer data and perform arithmetic



Lectures vs. Precepts

Approach to studying assembly language:

| Precepts | Lectures |
|---|---|
| Study complete pgms | Study partial pgms |
| Begin with small pgms; proceed to large ones | Begin with simple constructs; proceed to complex ones |
| Emphasis on writing code | Emphasis on reading code |



Agenda

Language Levels

Architecture

Assembly Language: Defining Global Data

Assembly Language: Performing Arithmetic



High-Level Languages

Characteristics

- Portable
 - To varying degrees
- Complex
 - One statement can do much work
- Expressive
 - To varying degrees
 - Good (code functionality / code size) ratio
- Human readable

```
count = 0;
while (n>1)
{ count++;
  if (n&1)
    n = n*3+1;
  else
    n = n/2;
}
```



Machine Languages

Characteristics

- Not portable
 - Specific to hardware
- Simple
 - Each instruction does a simple task
- Not expressive
 - Each instruction performs little work
 - Poor (code functionality / code size) ratio
- Not human readable
 - Requires lots of effort!
 - Requires tool support

| | | | | | | | |
|------|------|------|------|------|------|------|------|
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 9222 | 9120 | 1121 | A120 | 1121 | A121 | 7211 | 0000 |
| 0000 | 0001 | 0002 | 0003 | 0004 | 0005 | 0006 | 0007 |
| 0008 | 0009 | 000A | 000B | 000C | 000D | 000E | 000F |
| 0000 | 0000 | 0000 | FE10 | FACE | CAFE | ACED | CEDE |
| | | | | | | | |
| | | | | | | | |
| 1234 | 5678 | 9ABC | DEF0 | 0000 | 0000 | F00D | 0000 |
| 0000 | 0000 | EEEE | 1111 | EEEE | 1111 | 0000 | 0000 |
| B1B2 | F1F5 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |



Assembly Languages

Characteristics

- Not portable
 - Each assembly lang instruction maps to one machine lang instruction
- Simple
 - Each instruction does a simple task
- Not expressive
 - Poor (code functionality / code size) ratio
- **Human readable!!!**

```
        movl    $0, %ecx
loop:   cmpl    $1, %edx
        jle    endloop

        addl    $1, %ecx

        movl    %edx, %eax
        andl   $1, %eax
        je     else

        movl    %edx, %eax
        addl    %eax, %edx
        addl    %eax, %edx
        addl    $1, %edx

else:   jmp     endif

        sarl    $1, %edx

endif:  jmp     loop

endloop:
```




Why Learn Assembly Language?

Q: Why learn assembly language?

A: Knowing assembly language helps you:

- Write faster code
 - In assembly language
 - In a high-level language!
- Understand what's happening “under the hood”
 - Someone needs to develop future computer systems
 - Maybe that will be you!



Why Learn IA-32 Assembly Lang?

Why learn IA-32 assembly language?

Pros

- IA-32 is the most popular processor
- Nobel computers are IA-32 computers
 - Program natively on nobel instead of using an emulator

Cons

- IA-32 assembly language is **big**
 - Each instruction is simple, but...
 - There are **many** instructions
 - Instructions differ widely

We'll study a popular subset

- As defined by Bryant & O'Hallaron Ch 3 and precept **IA-32 Assembly Language** document



Agenda

Language Levels

Architecture

Assembly Language: Defining Global Data

Assembly Language: Performing Arithmetic



John Von Neumann (1903-1957)

In computing

- Stored program computers
- Cellular automata
- Self-replication

Other interests

- Mathematics
- Nuclear physics (hydrogen bomb)

Princeton connection

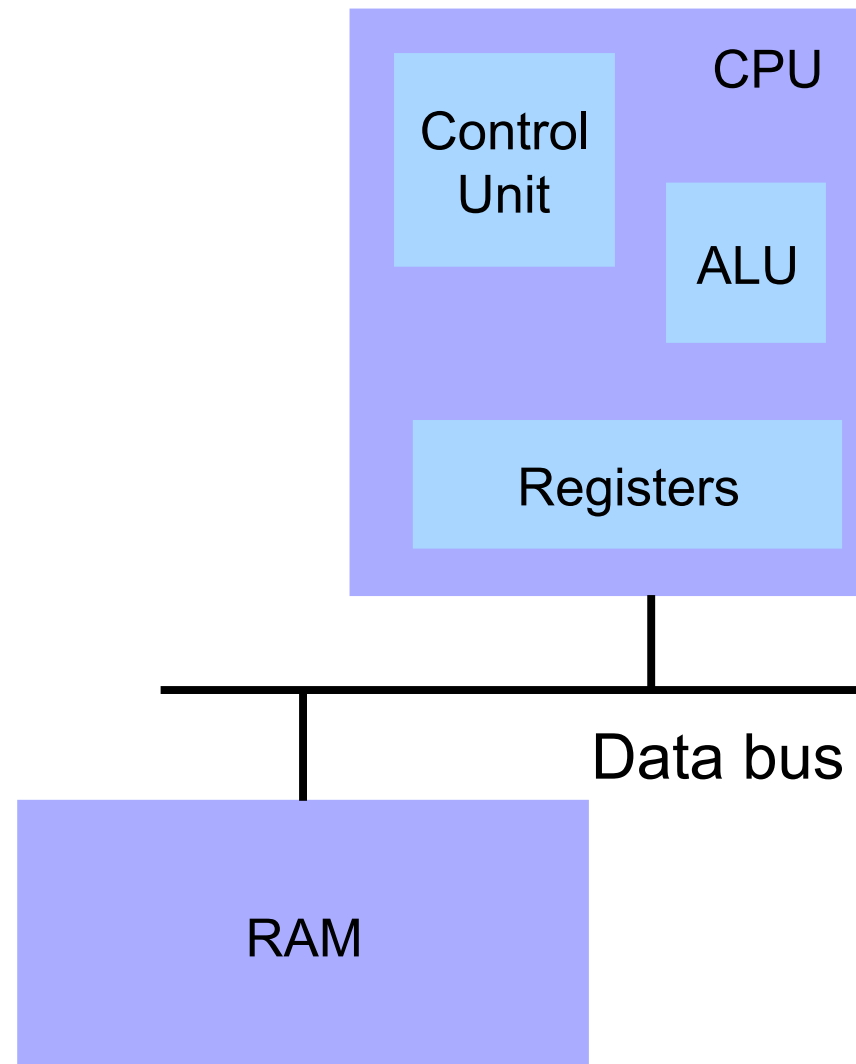
- Princeton Univ & IAS, 1930-death

Known for “Von Neumann architecture”

- In contrast to less successful “Harvard architecture”



Von Neumann Architecture

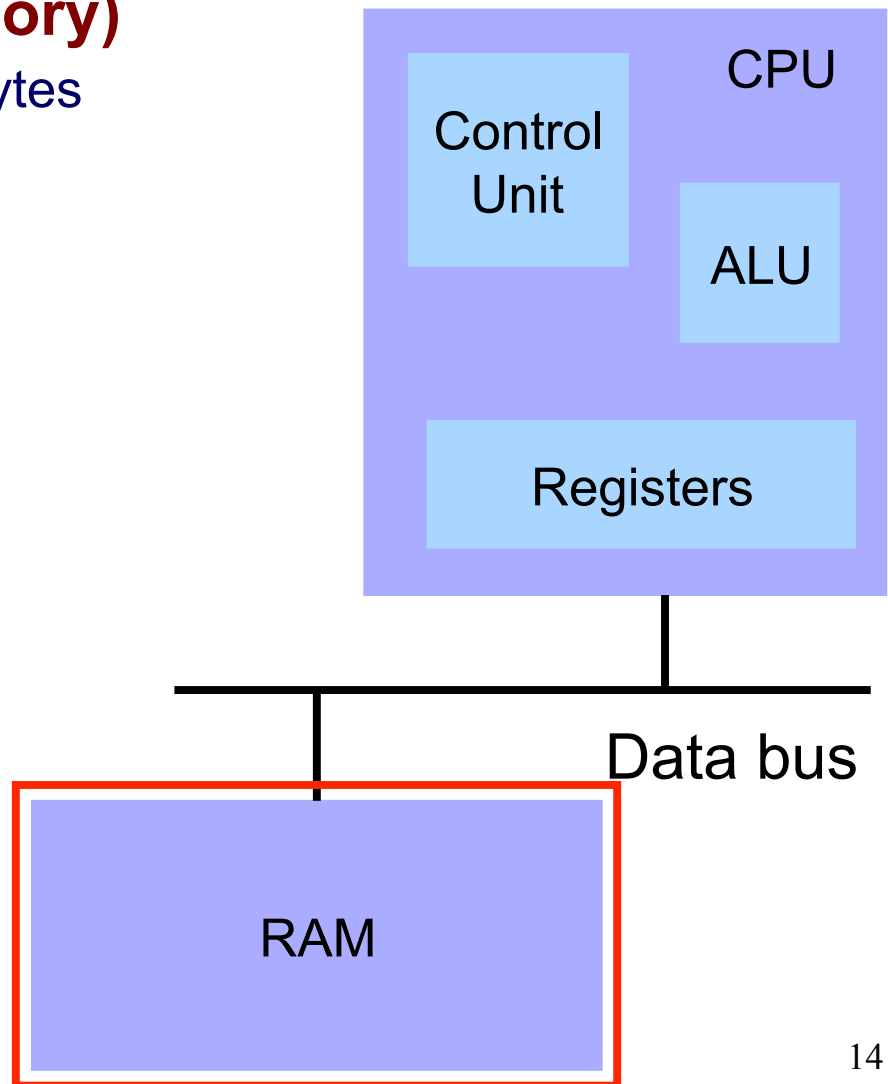
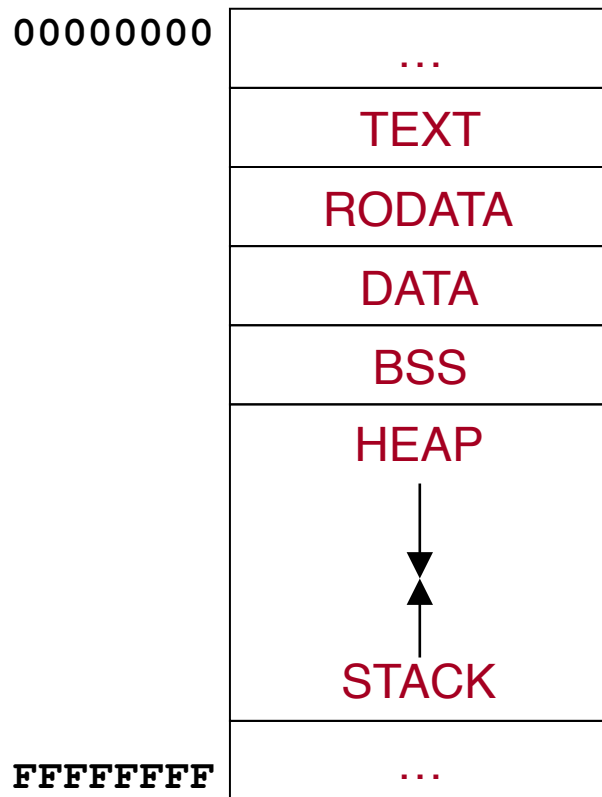




RAM

RAM (Random Access Memory)

- Conceptually: large array of bytes

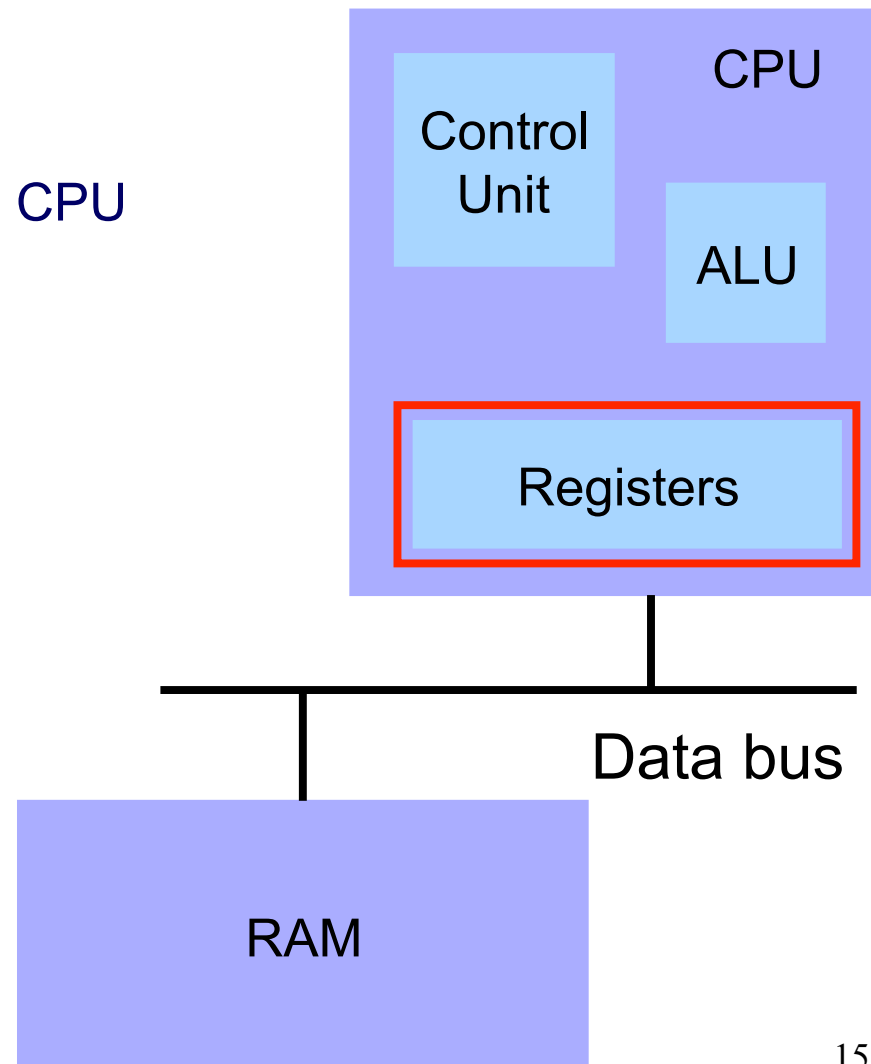




Registers

Registers

- Small amount of storage on the CPU
- Much faster than RAM
- Top of the storage hierarchy
 - Above RAM, disk, ...





Registers

General purpose registers

| 31 | 15 | 8 7 | 0 | 16-bit | 32-bit |
|----|----|-----|----|--------|--------|
| | AH | | AL | AX | EAX |
| | BH | | BL | BX | EBX |
| | CH | | CL | CX | ECX |
| | DH | | DL | DX | EDX |
| | SI | | | | ESI |
| | DI | | | | EDI |
| | BP | | | | EBP |
| | SP | | | | ESP |



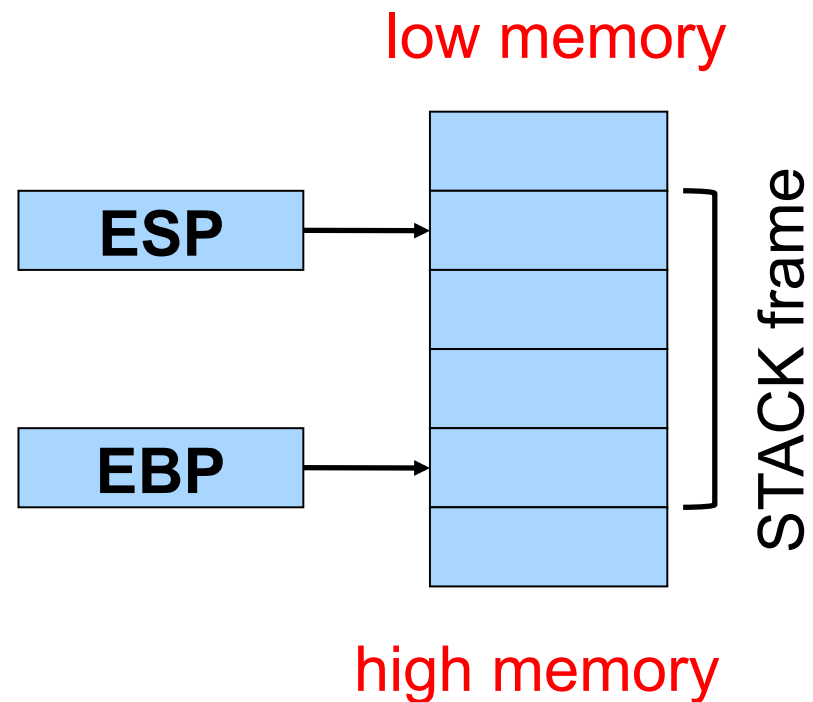
ESP and EBP Registers

ESP (Stack Pointer) register

- Contains address of top (low address) of current function's stack frame

EBP (Base Pointer) register

- Contains address of bottom (high address) of current function's stack frame



Allow effective use of the STACK section of memory

(See **Assembly Language: Function Calls** lecture)



EFLAGS Register

Special-purpose register...

EFLAGS (Flags) register

- Contains **CC (Condition Code) bits**
- Affected by compare (**cmp**) instruction
 - And many others
- Used by conditional jump instructions
 - **je, jne, jl, jg, jle, jge, jb, jbe, ja, jae, jb**

(See **Assembly Language: Part 2** lecture)

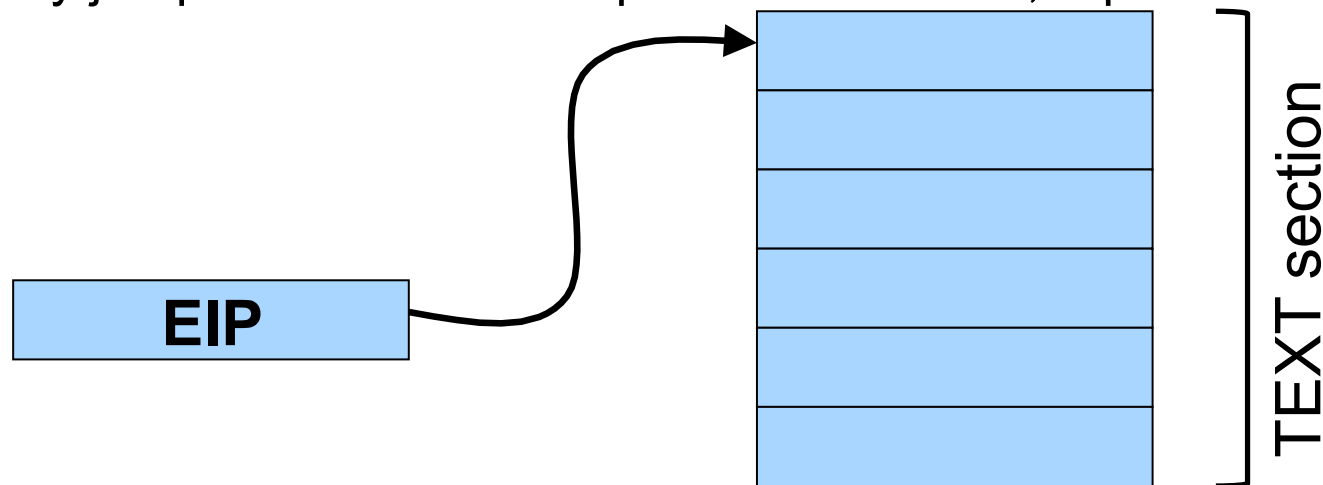


EIP Register

Special-purpose register...

EIP (Instruction Pointer) register

- Stores the location of the next instruction
 - Address (in TEXT section) of machine-language instructions to be executed next
- Value changed:
 - Automatically to implement sequential control flow
 - By jump instructions to implement selection, repetition





Registers and RAM

Typical pattern:

- **Load** data from RAM to registers
- **Manipulate** data in registers
- **Store** data from registers to RAM

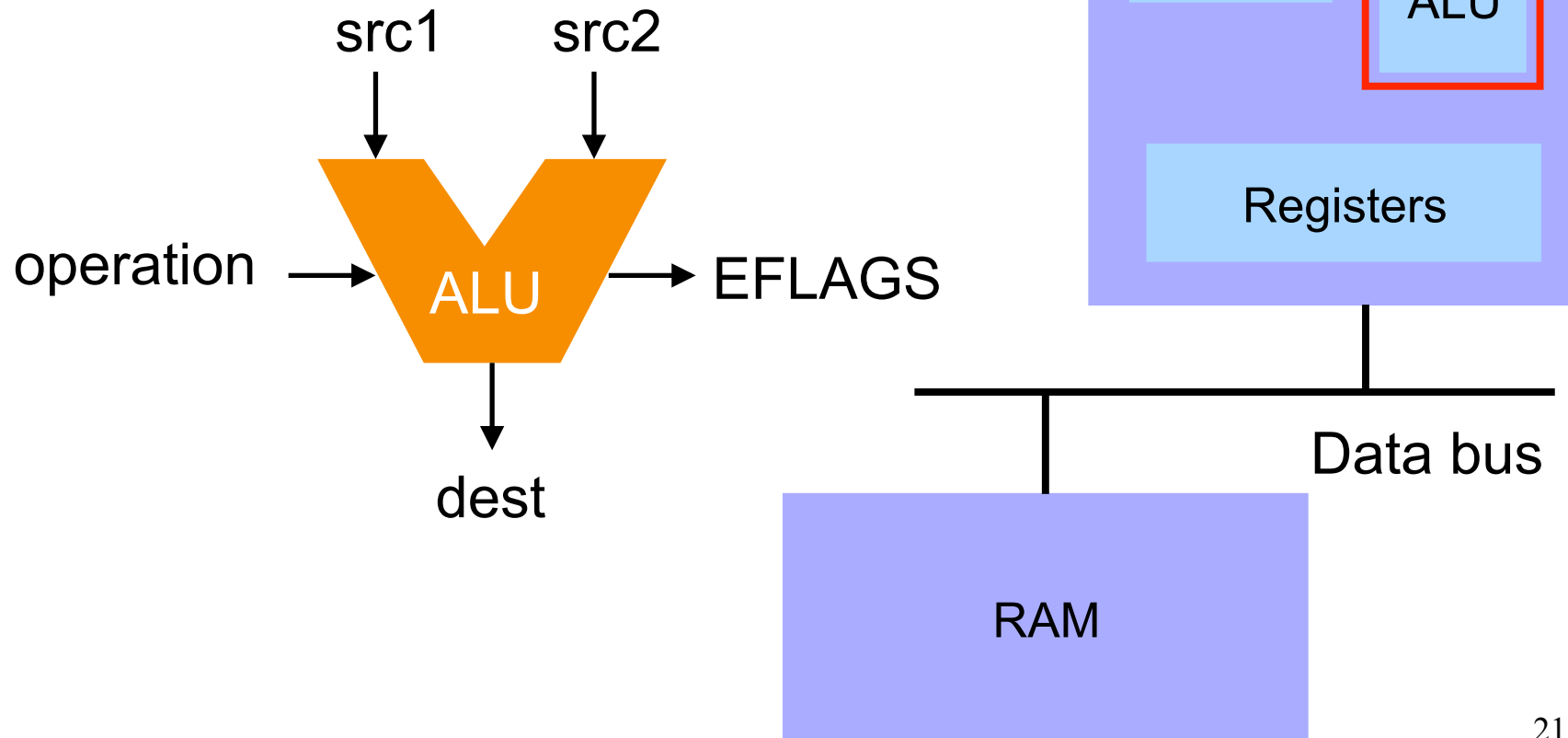
Many instructions combine steps



ALU

ALU (Arithmetic Logic Unit)

- Performs arithmetic and logic operations

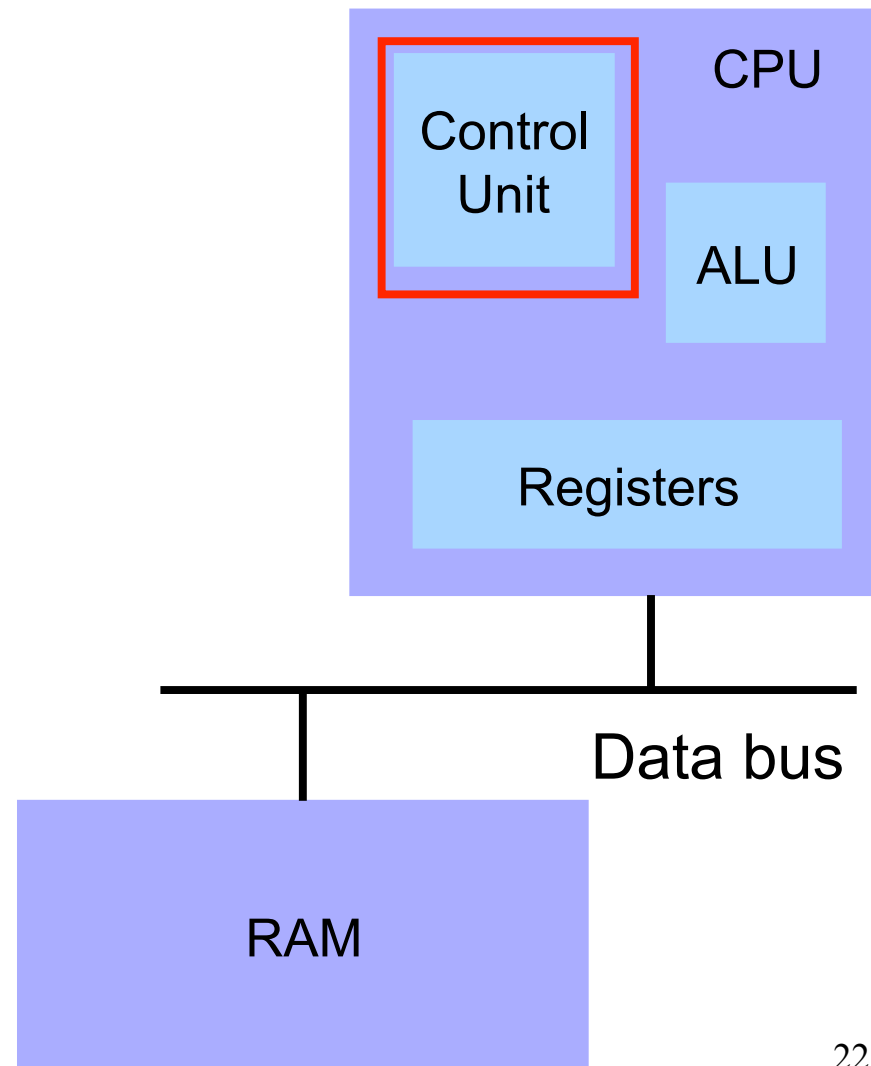




Control Unit

Control Unit

- Fetches and decodes each machine-language instruction
- Sends proper data to ALU

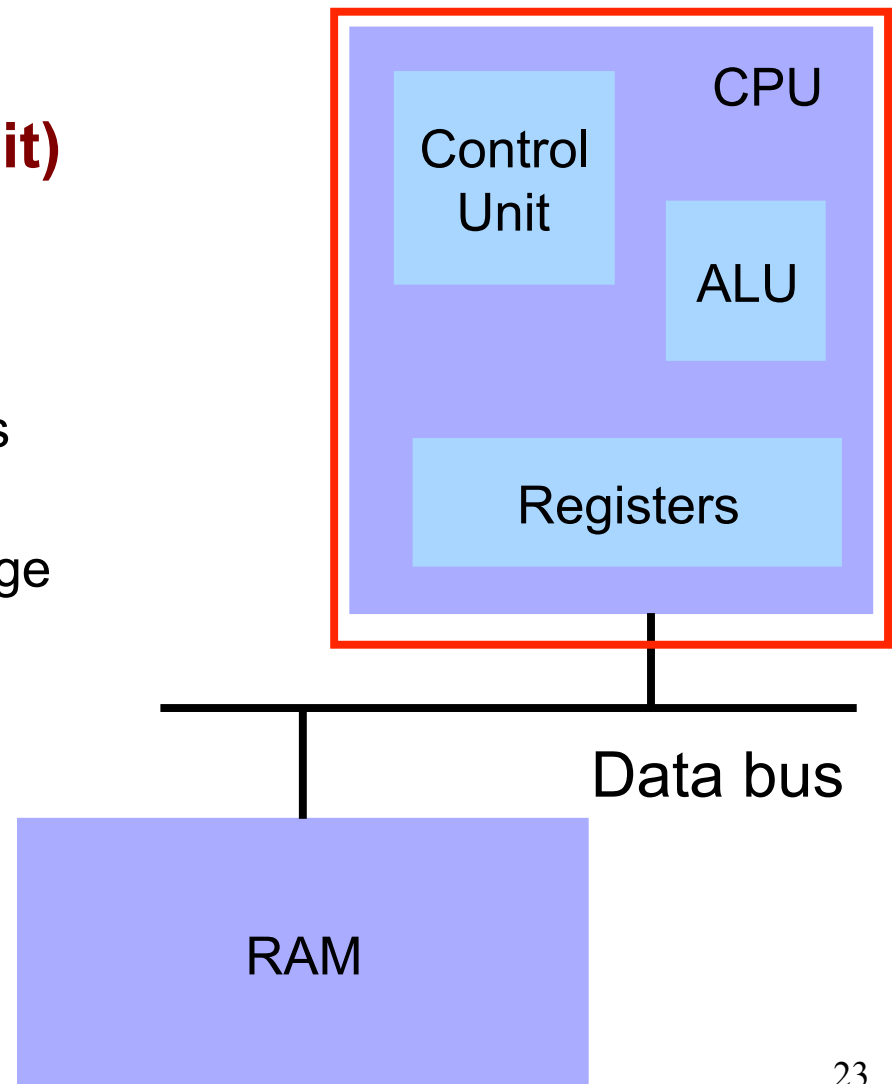


CPU



CPU (Central Processing Unit)

- Control unit
 - Fetch, decode, and execute
- ALU
 - Execute low-level operations
- Registers
 - High-speed temporary storage





Agenda

Language Levels

Architecture

Assembly Language: Defining Global Data

Assembly Language: Performing Arithmetic



Defining Data: DATA Section 1

```
static char c = 'a';  
static short s = 12;  
static int i = 345;
```

```
.section ".data"  
c:  
    .byte 'a'  
s:  
    .word 12  
i:  
    .long 345
```

Note:

- `.section` instruction (to announce DATA section)
- label definition (marks a spot in RAM)
- `.byte` instruction (1 byte)
- `.word` instruction (2 bytes)
- `.long` instruction (4 bytes)

Note:

Best to avoid “word” (2 byte) data



Defining Data: DATA Section 2

```
char c = 'a';  
short s = 12;  
int i = 345;
```

```
.section ".data"  
  
.globl c  
c: .byte 'a'  
  
.globl s  
s: .word 12  
  
.globl i  
i: .long 345
```

Note:

Can place label on same line as next instruction

.globl instruction



Defining Data: BSS Section

```
static char c;  
static short s;  
static int i;
```

```
.section ".bss"  
c:  
.skip 1  
s:  
.skip 2  
i:  
.skip 4
```

Note:

- .section** instruction (to announce BSS section)
- .skip** instruction



Defining Data: RODATA Section

```
...  
..."hello\n"...;  
...
```

```
.section ".rodata"  
helloLabel:  
.string "hello\n"
```

Note:

- `.section` instruction (to announce RODATA section)
- `.string` instruction



Agenda

Language Levels

Architecture

Assembly Language: Defining Global Data

Assembly Language: Performing Arithmetic



Instruction Format

Many instructions have this format:

```
name{b,w,l} src, dest
```

- **name**: name of the instruction (mov, add, sub, and, etc.)
- **byte** => operands are one-byte entities
- **word** => operands are two-byte entities
- **long** => operands are four-byte entities



Instruction Format

Many instructions have this format:

```
name{b,w,l} src, dest
```

- **src: source operand**
 - The source of data
 - Can be
 - **Register operand:** `%eax`, `%ebx`, etc.
 - **Memory operand:** `5` (legal but silly), `someLabel`
 - **Immediate operand:** `$5`, `$someLabel`



Instruction Format

Many instructions have this format:

```
name{b,w,l} src, dest
```

- **dest: destination operand**
 - The destination of data
 - Can be
 - **Register operand:** `%eax`, `%ebx`, etc.
 - **Memory operand:** `5` (legal but silly), `someLabel`
 - Cannot be
 - **Immediate operand**



Performing Arithmetic: Long Data

```
static int length;  
static int width;  
static int perim;  
...  
perim =  
    (length + width) * 2;
```

```
.section ".bss"  
length: .skip 4  
width: .skip 4  
perim: .skip 4  
...  
.section ".text"  
...  
movl length, %eax  
addl width, %eax  
sall $1, %eax  
movl %eax, perim
```

Note:

movl instruction

addl instruction

sall instruction

Register operand

Immediate operand

Memory operand

.section instruction (to announce TEXT section)



Performing Arithmetic: Byte Data

```
static char grade = 'B';  
...  
grade--;
```

Note:

Comment

movb instruction

subb instruction

decb instruction

What would happen if we use **movl** instead of **movb**?

```
.section ".data"  
grade: .byte 'B'  
...  
.section ".text"  
...  
# Option 1  
movb grade, %al  
subb $1, %al  
movb %al, grade  
...  
# Option 2  
subb $1, grade  
...  
# Option 3  
decb grade
```



Generalization: Operands

Immediate operands

- `$5` => use the number 5 (i.e. the number that is available immediately within the instruction)
- `$i` => use the address denoted by `i` (i.e. the address that is available immediately within the instruction)
- Can be source operand; cannot be destination operand

Register operands

- `%eax` => read from (or write to) register EAX
- Can be source or destination operand

Memory operands

- `5` => load from (or store to) memory at address 5 (silly; seg fault)
- `i` => load from (or store to) memory at the address denoted by `i`
- Can be source or destination operand (**but not both**)
- There's more to memory operands; see next lecture



Generalization: Notation

Instruction notation:

- l => long (4 bytes); w => word (2 bytes); b => byte (1 byte)

Operand notation:

- src => source; dest => destination
- R => register; I => immediate; M => memory



Generalization: Data Transfer

Data transfer instructions

| | |
|--|--|
| <code>mov{l,w,b} srcIRM, destRM</code> | <code>dest = src</code> |
| <code>movsb{l,w} srcRM, destR</code> | <code>dest = src (sign extend)</code> |
| <code>movswl srcRM, destR</code> | <code>dest = src (sign extend)</code> |
| <code>movzb{l,w} srcRM, destR</code> | <code>dest = src (zero fill)</code> |
| <code>movzwl srcRM, destR</code> | <code>dest = src (zero fill)</code> |
| <code>cld</code> | <code>reg[EDX:EAX] = reg[EAX]</code> <code>(sign extend)</code> |
| <code>cwtd</code> | <code>reg[DX:AX] = reg[AX]</code> <code>(sign extend)</code> |
| <code>cbtw</code> | <code>reg[AX] = reg[AL]</code> <code>(sign extend)</code> |

`mov` is used often; others rarely



Generalization: Arithmetic

Arithmetic instructions

| | |
|--|---------------------------|
| <code>add{l,w,b} srcIRM, destRM</code> | <code>dest += src</code> |
| <code>sub{l,w,b} srcIRM, destRM</code> | <code>dest -= src</code> |
| <code>inc{l,w,b} destRM</code> | <code>dest++</code> |
| <code>dec{l,w,b} destRM</code> | <code>dest--</code> |
| <code>neg{l,w,b} destRM</code> | <code>dest = -dest</code> |



Generalization: Signed Mult & Div

Signed multiplication and division instructions

| | |
|--------------------------|--|
| <code>imull srcRM</code> | <code>reg[EDX:EAX] = reg[EAX]*src</code> |
| <code>imulw srcRM</code> | <code>reg[DX:AX] = reg[AX]*src</code> |
| <code>imulb srcRM</code> | <code>reg[AX] = reg[AL]*src</code> |
| <code>idivl srcRM</code> | <code>reg[EAX] = reg[EDX:EAX]/src</code> <code>reg[EDX] = reg[EDX:EAX]%src</code> |
| <code>idivw srcRM</code> | <code>reg[AX] = reg[DX:AX]/src</code> <code>reg[DX] = reg[DX:AX]%src</code> |
| <code>idivb srcRM</code> | <code>reg[AL] = reg[AX]/src</code> <code>reg[AH] = reg[AX]%src</code> |

See Bryant & O'Hallaron book for description of signed vs. unsigned multiplication and division



Generalization: Unsigned Mult & Div

Unsigned multiplication and division instructions

| | |
|-------------------------|--|
| <code>mull srcRM</code> | <code>reg[EDX:EAX] = reg[EAX]*src</code> |
| <code>mulw srcRM</code> | <code>reg[DX:AX] = reg[AX]*src</code> |
| <code>mulb srcRM</code> | <code>reg[AX] = reg[AL]*src</code> |
| <code>divl srcRM</code> | <code>reg[EAX] = reg[EDX:EAX]/src</code> <code>reg[EDX] = reg[EDX:EAX]%src</code> |
| <code>divw srcRM</code> | <code>reg[AX] = reg[DX:AX]/src</code> <code>reg[DX] = reg[DX:AX]%src</code> |
| <code>divb srcRM</code> | <code>reg[AL] = reg[AX]/src</code> <code>reg[AH] = reg[AX]%src</code> |

See Bryant & O'Hallaron book for description of signed vs. unsigned multiplication and division



Generalization: Bit Manipulation

Bitwise instructions

| | |
|--|---|
| <code>and{l,w,b} srcIRM, destRM</code> | <code>dest = src & dest</code> |
| <code>or{l,w,b} srcIRM, destRM</code> | <code>dest = src dest</code> |
| <code>xor{l,w,b} srcIRM, destRM</code> | <code>dest = src ^ dest</code> |
| <code>not{l,w,b} destRM</code> | <code>dest = ~dest</code> |
| <code>sal{l,w,b} srcIR, destRM</code> | <code>dest = dest << src</code> |
| <code>sar{l,w,b} srcIR, destRM</code> | <code>dest = dest >> src (sign extend)</code> |
| <code>shl{l,w,b} srcIR, destRM</code> | <code>(Same as sal)</code> |
| <code>shr{l,w,b} srcIR, destRM</code> | <code>dest = dest >> src (zero fill)</code> |



Summary

Language levels

The basics of computer architecture

- Enough to understand IA-32 assembly language

The basics of IA-32 assembly language

- Instructions to define global data
- Instructions to perform data transfer and arithmetic

To learn more

- Study more assembly language examples
 - Chapter 3 of Bryant and O' Hallaron book
- Study compiler-generated assembly language code
 - `gcc217 -S somefile.c`



Appendix

Big-endian vs little-endian byte order



Byte Order

Intel is a **little endian** architecture

- **Least** significant byte of multi-byte entity is stored at lowest memory address
- “Little end goes first”

The int 5 at address 1000:

| | |
|------|----------|
| 1000 | 00000101 |
| 1001 | 00000000 |
| 1002 | 00000000 |
| 1003 | 00000000 |

Some other systems use **big endian**

- **Most** significant byte of multi-byte entity is stored at lowest memory address
- “Big end goes first”

The int 5 at address 1000:

| | |
|------|----------|
| 1000 | 00000000 |
| 1001 | 00000000 |
| 1002 | 00000000 |
| 1003 | 00000101 |



Byte Order Example 1

```
#include <stdio.h>
int main(void)
{
    unsigned int i = 0x003377ff;
    unsigned char *p;
    int j;
    p = (unsigned char *)&i;
    for (j=0; j<4; j++)
        printf("Byte %d: %2x\n", j, p[j]);
}
```

Output on a
little-endian
machine

Byte 0: ff
Byte 1: 77
Byte 2: 33
Byte 3: 00

Output on a
big-endian
machine

Byte 0: 00
Byte 1: 33
Byte 2: 77
Byte 3: ff



Byte Order Example 2

Note:

Flawed code; uses “b” instructions to manipulate a four-byte memory area

Intel is **little** endian, so what will be the value of grade?

What would be the value of grade if Intel were **big** endian?

```
.section ".data"
grade: .long 'B'
...
.section ".text"
...
# Option 1
movb grade, %al
subb $1, %al
movb %al, grade
...
# Option 2
subb $1, grade
```



Byte Order Example 3

Note:

Flawed code; uses "l" instructions to manipulate a one-byte memory area

What would happen?

```
.section ".data"
grade: .byte 'B'
...
.section ".text"
...
# Option 1
movl grade, %eax
subl $1, %eax
movl %eax, grade
...
# Option 2
subl $1, grade
```