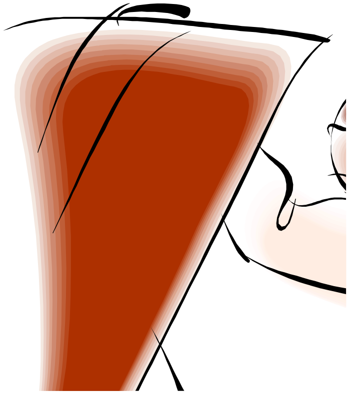
Assembly Language: Part 1

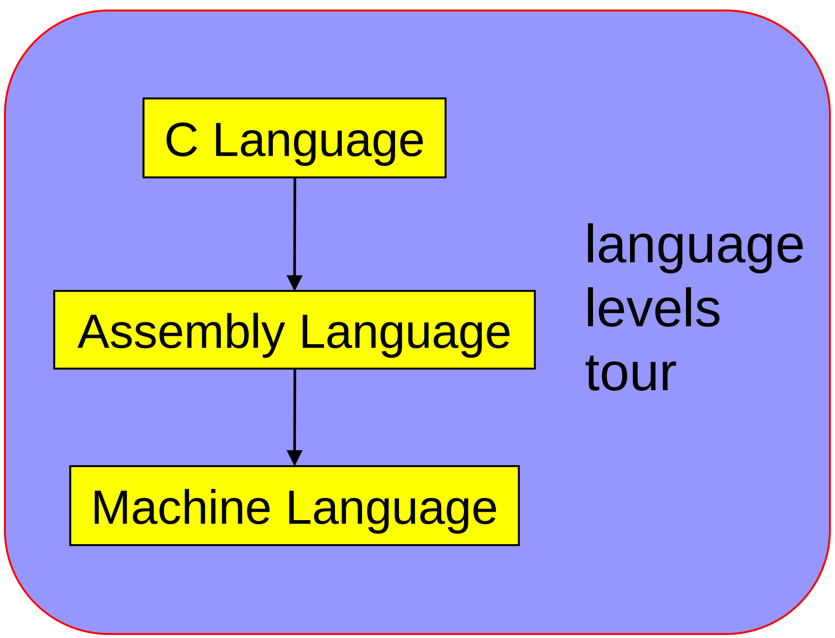


Context of this Lecture

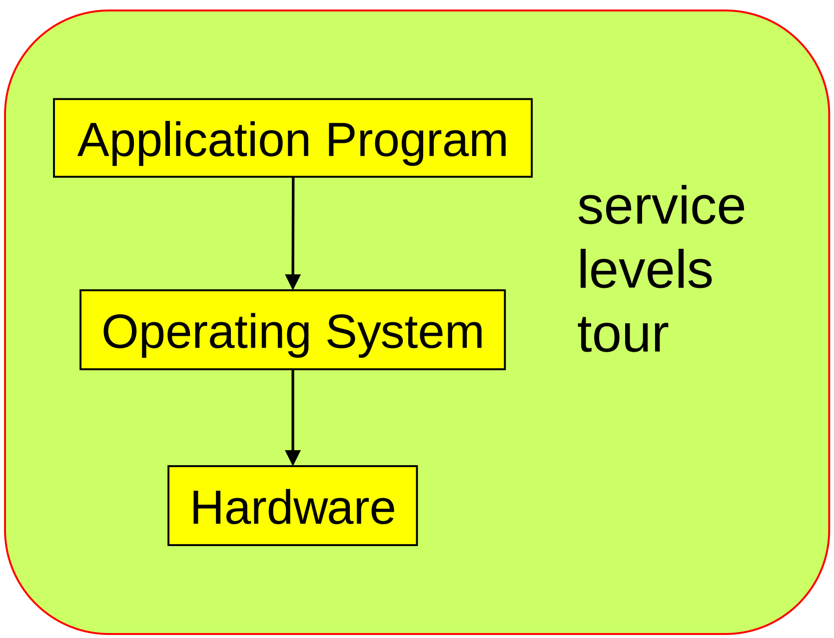
First half of the semester: "Programming in the large"

Second half: "Under the hood"

Starting Now



Afterward



Von Neumann Architecture

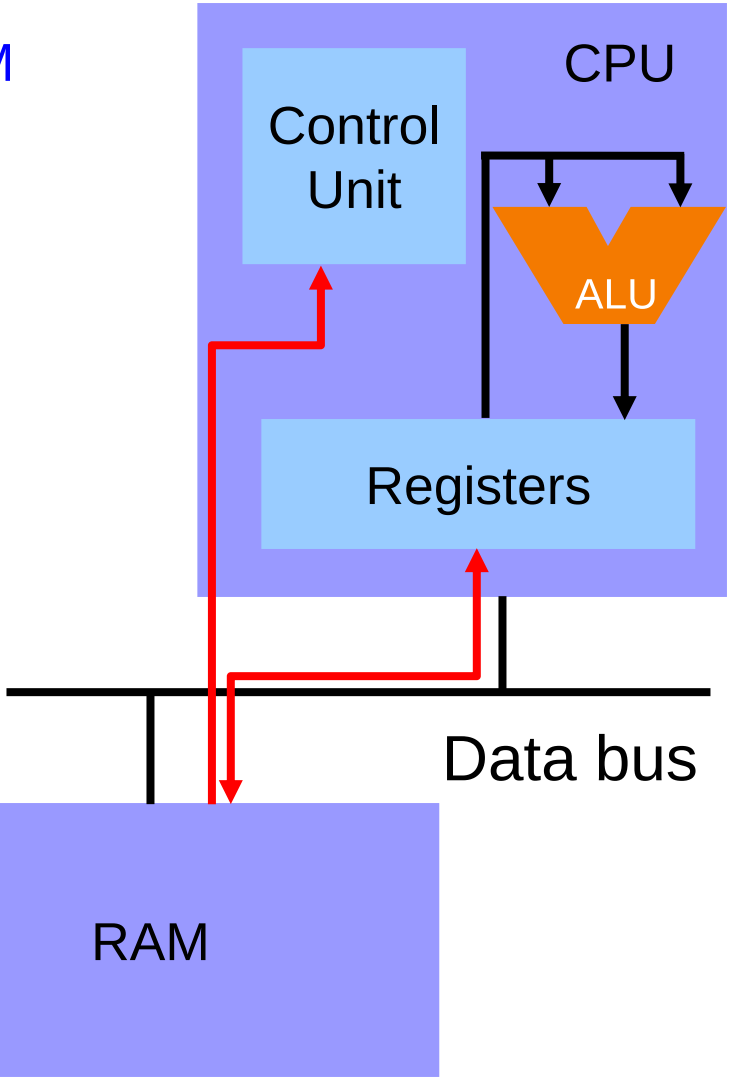
Instructions are fetched from RAM

- (encoded as bits)

Control unit interprets instructions

- to shuffle data between registers and RAM

- to move data from registers through ALU (arithmetic+logic unit) where operations are performed



Agenda

Language Levels

Instruction-Set Architecture (ISA)

Assembly Language: Performing Arithmetic

Assembly Language: Control-flow instructions

High-Level Languages

Characteristics

- Portable
  - To varying degrees
- Complex
  - One statement can do much work
- Structured

  while (...) {...}   if () ... else ...
- 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
- Unstructured
- 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

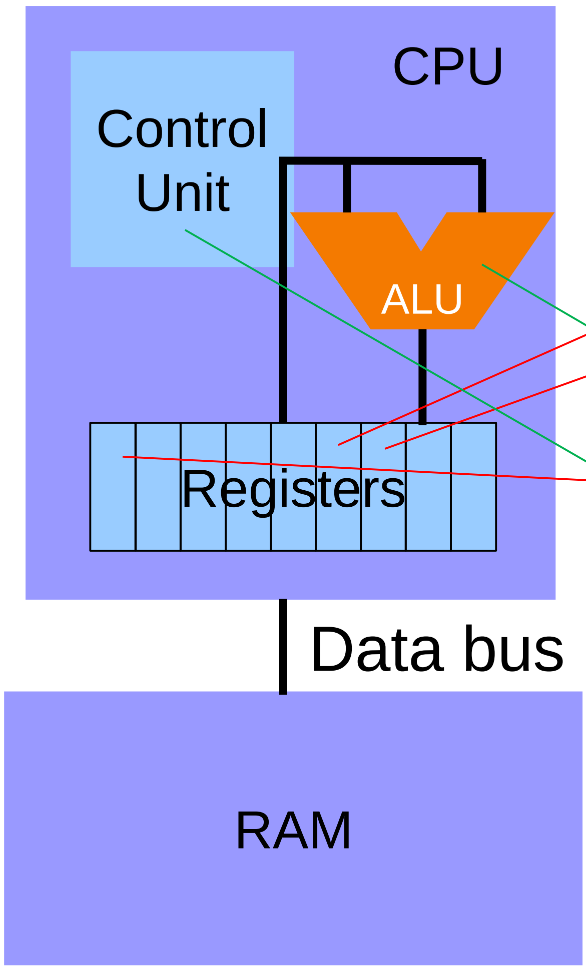
Characteristics

- Not portable
  - Each assembly lang instruction maps to one machine lang instruction
- Simple
  - Each instruction does a simple task
- Unstructured
- **Human readable!!!**

(well, in the same sense that Hungarian is human readable, if you know Hungarian).

```
        movl    $0, %r10d

loop:
        cmpl    $1, %r11d
        jle     endloop

        addl    $1, %r10d

        movl    %r11d, %eax
        andl    $1, %eax
        je      else

        movl    %r11d, %eax
        addl    %eax, %r11d
        addl    %eax, %r11d
        addl    $1, %r11d

        jmp     endif
else:
        sarl    $1, %r11d

endif:

        jmp     loop
endloop:
```

Computer: CPU + RAM




```
        movl    $0, %r10d
loop:
        cmpl    $1, %r11d
        jle     endloop

        addl    $1, %r10d

        movl    %r11d, %eax
        andl    $1, %eax
        je      else

        movl    %r11d, %eax
        addl    %eax, %r11d
        addl    %eax, %r11d
        addl    $1, %r11d

        jmp     endif
else:
        sarl    $1, %r11d
endif:
        jmp     loop
endloop:
```

Translation: C to x86-64

count↔r10d
n↔r11d

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

```
        movl    $0, %r10d
loop:
        cmpl    $1, %r11d
        jle     endloop
        addl    $1, %r10d
        movl    %r11d, %eax
        andl    $1, %eax
        je      else
        movl    %r11d, %eax
        addl    %eax, %r11d
        addl    %eax, %r11d
        addl    $1, %r11d
        jmp     endif
else:
        sarl    $1, %r11d
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 x86-64 Assembly Lang?

Why learn **x86-64** assembly language?

Pros

- X86-64 is widely used
- CourseLab computers are x86-64 computers
  - Program natively on CourseLab instead of using an emulator

Cons

- X86-64 assembly language is **big and ugly**
  - There are **many** instructions
  - Instructions differ widely

Agenda

Language Levels

Architecture

Assembly Language: Performing Arithmetic

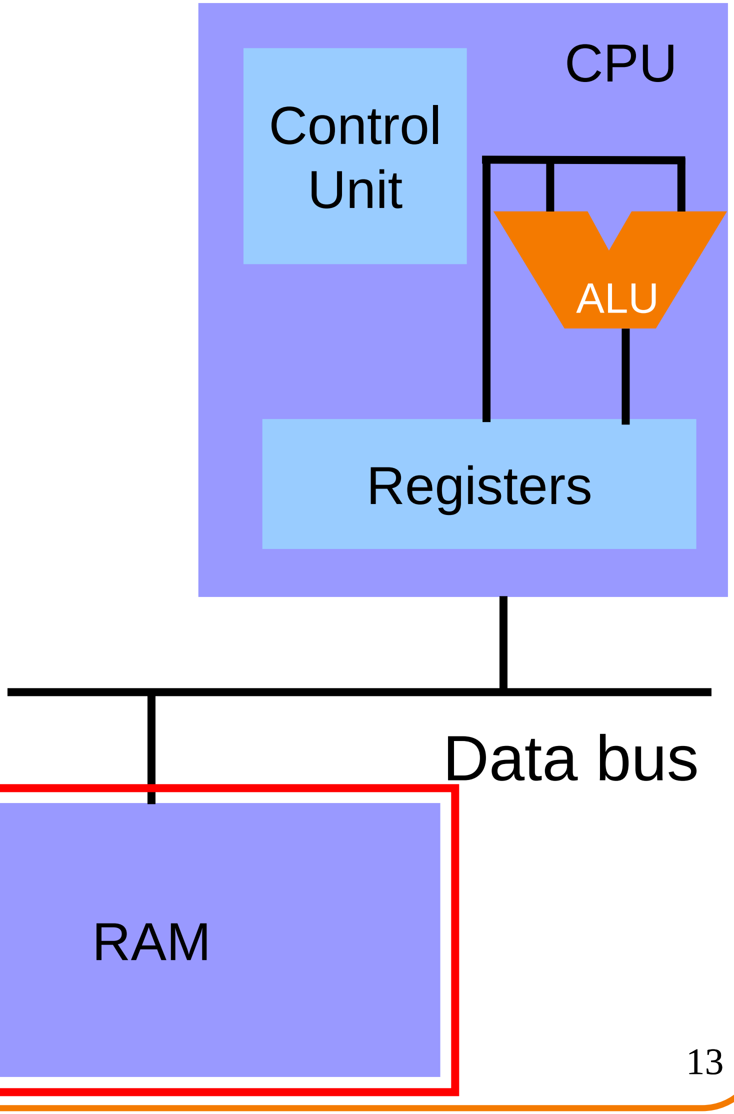
Assembly Language: Control-flow instructions

RAM

RAM (Random Access Memory)

Conceptually: large array of bytes

- Contains data
  (program variables, structs, arrays)
- and the program!



John Von Neumann (1903-1957)

In computing

- Stored program computers
- Cellular automata
- Self-replication

Other interests

- Mathematics
- Inventor of game theory
- Nuclear physics (hydrogen bomb)

Princeton connection

- Princeton Univ & IAS, 1930-1957

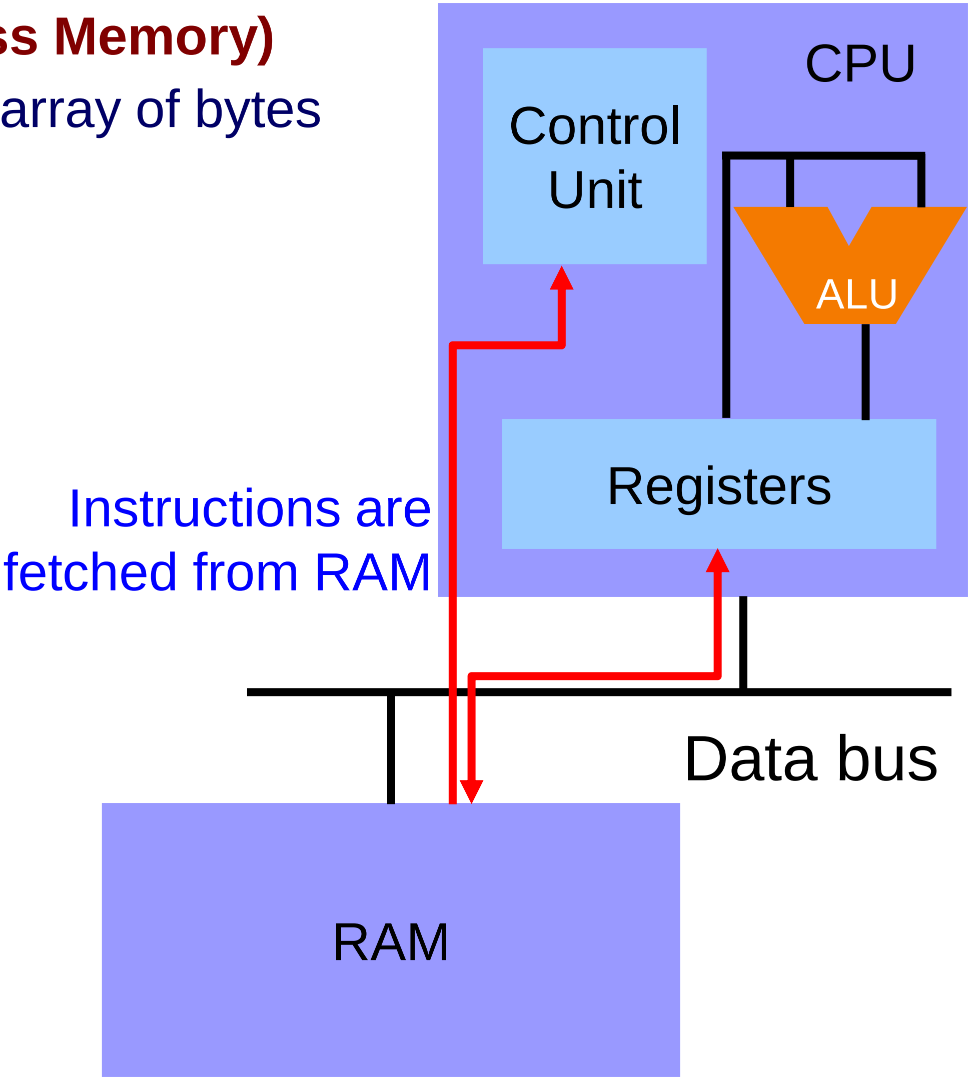
Known for “Von Neumann architecture (1950)”

- In which programs are just data in the memory
- Contrast to the now-obsolete “Harvard architecture”

Von Neumann Architecture

RAM (Random Access Memory)

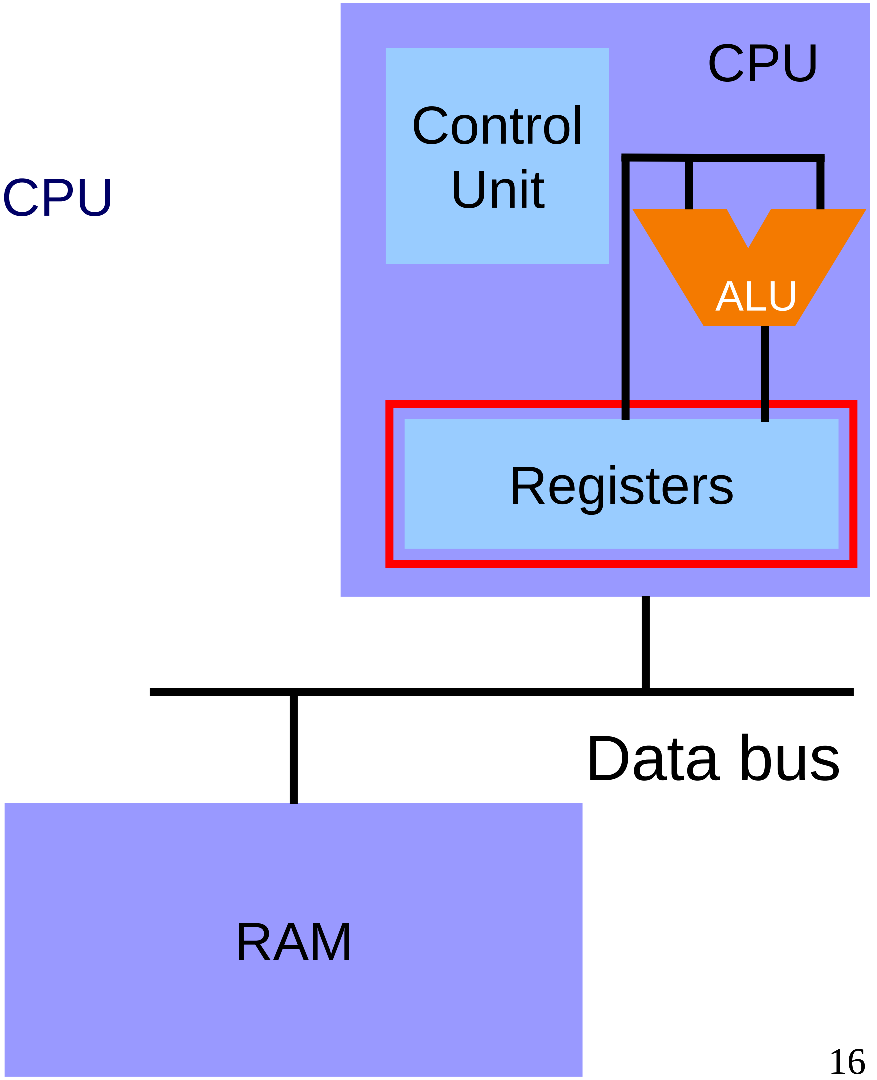
Conceptually: large array of bytes



Registers

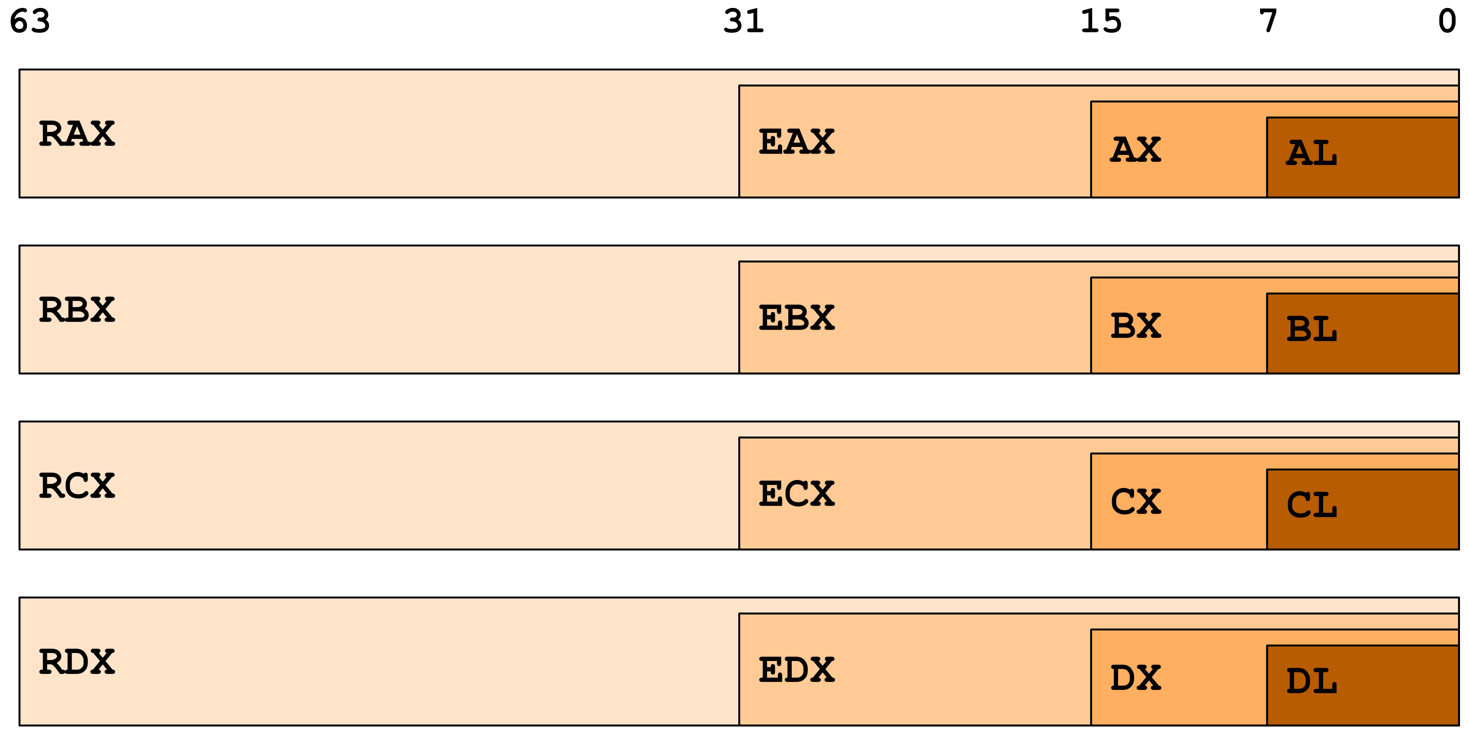
Registers

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



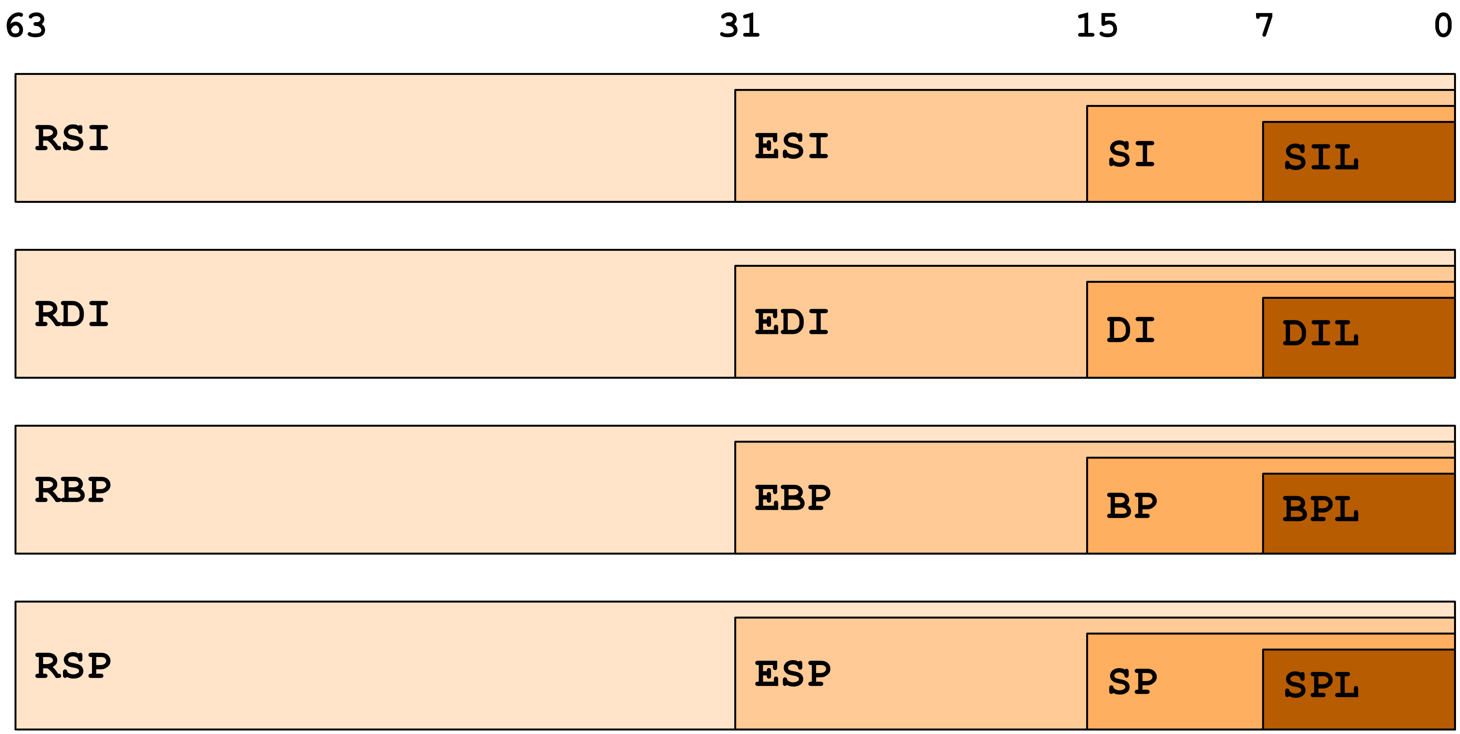
Registers (x86-64 architecture)

General purpose registers:



Registers (x86-64 architecture)

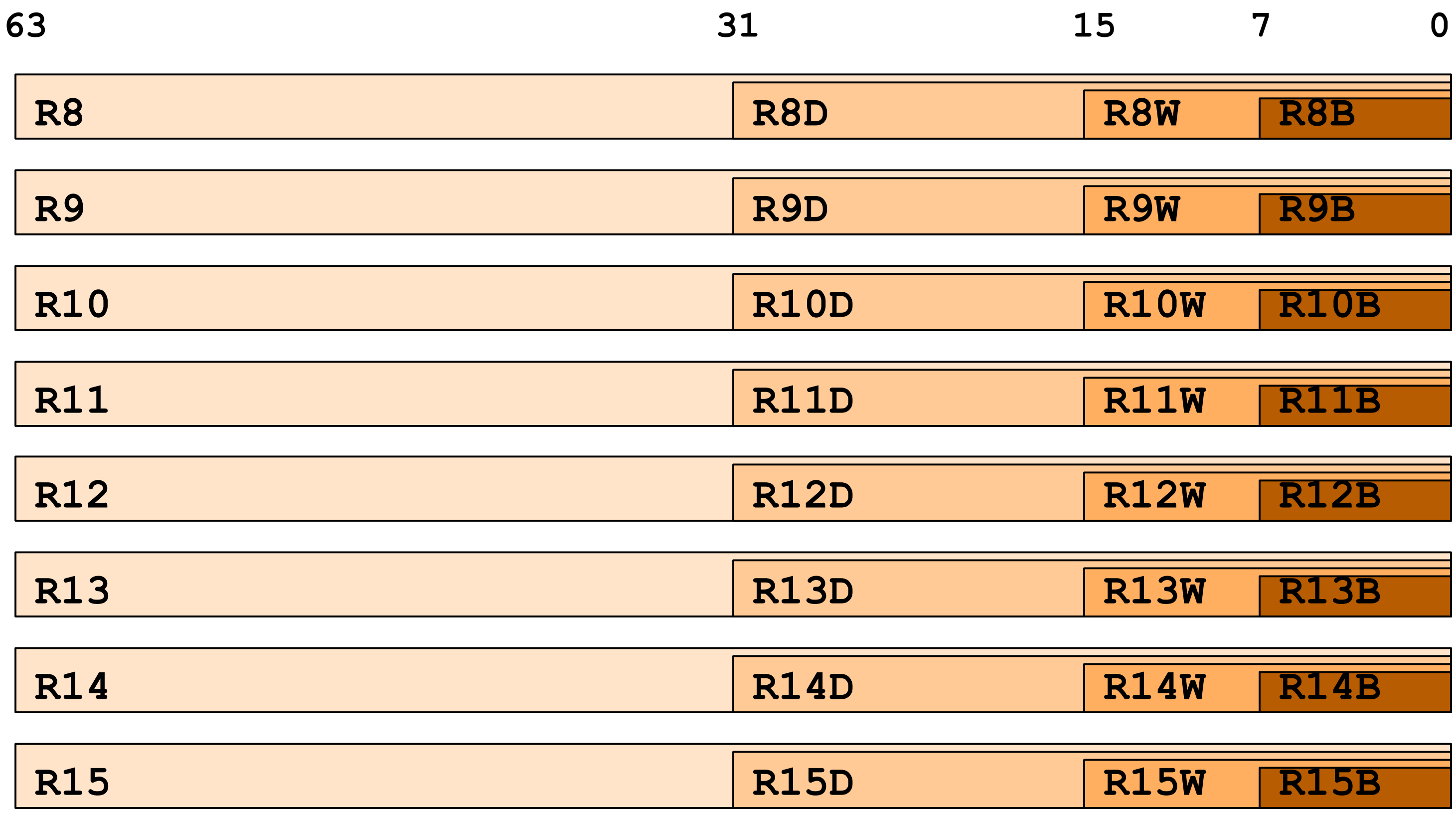
General purpose registers (cont.):



RSP is unique; see upcoming slide

Registers (x86-64 architecture)

General purpose registers (cont.):



Registers summary

16 general-purpose 64-bit pointer/long-integer registers, many with stupid names:

rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15

rbp: sometimes used as a “frame pointer” or “base pointer”

rsp: “stack pointer”

If you’re operating on 32-bit “int” data, use these stupid names instead:

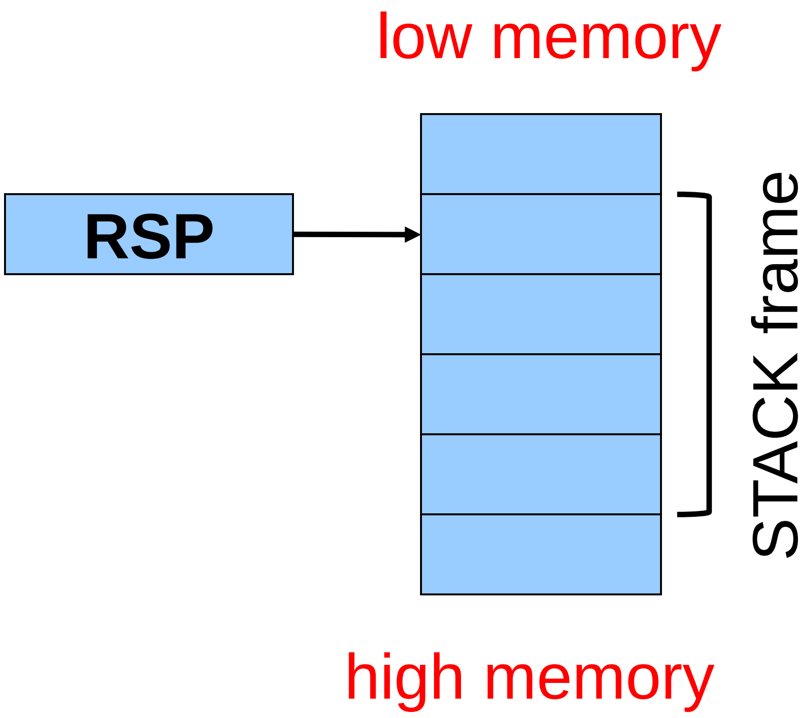
eax, ebx, ecx, edx, esi, edi, ebp, rsp, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d

rsp: it doesn’t really make sense to put 32-bit ints in the stack pointer

RSP Register

RSP (Stack Pointer) register

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



Allows 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`**

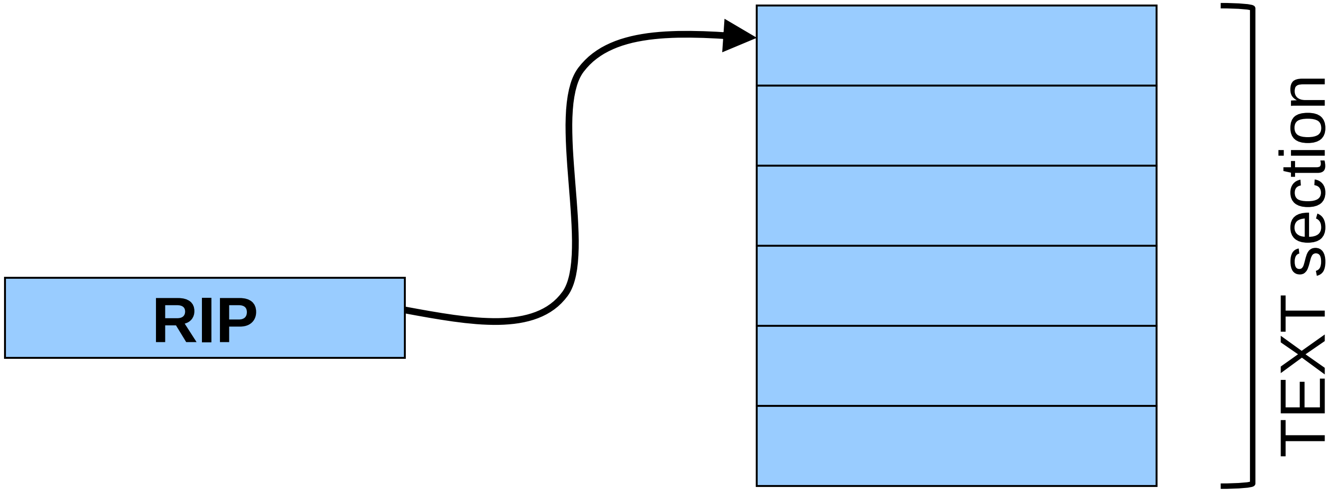
(See **Assembly Language: Part 2** lecture)

RIP Register

Special-purpose register…

RIP (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 summary

16 general-purpose 64-bit pointer/long-integer registers, many with stupid names:

rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15

rbp: sometimes used as a “frame pointer” or “base pointer”

rsp: “stack pointer”

If you’re operating on 32-bit “int” data, use these stupid names instead:

eax, ebx, ecx, edx, esi, edi, ebp, rsp, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d

rsp: it doesn’t really make sense to put 32-bit ints in the stack pointer

2 special-purpose registers: eflags rip

eflags: “condition codes”

rip: “program counter”

Registers and RAM

Typical pattern:

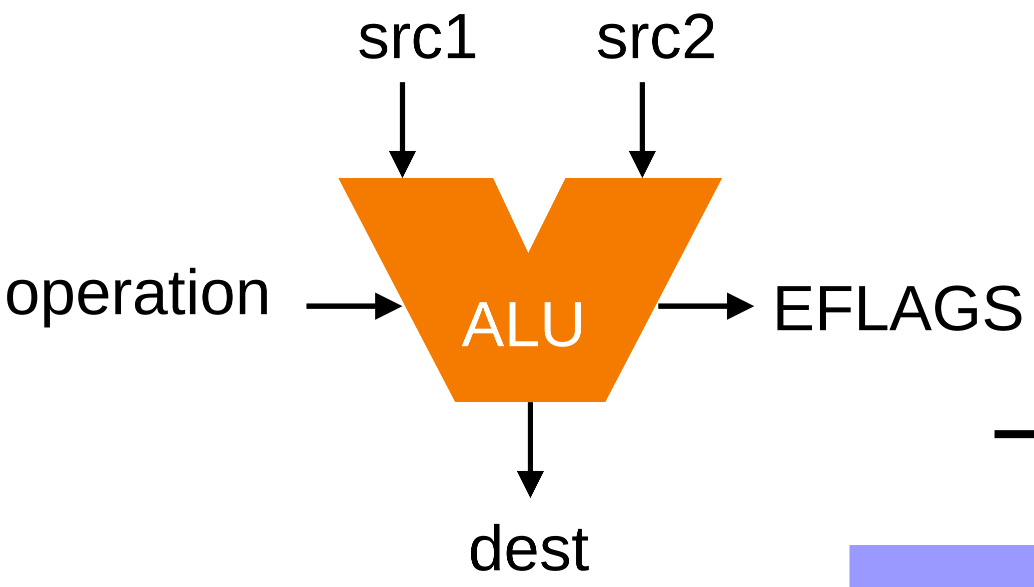
- **Load** data from RAM to registers
- **Manipulate** data in registers
- **Store** data from registers to RAM

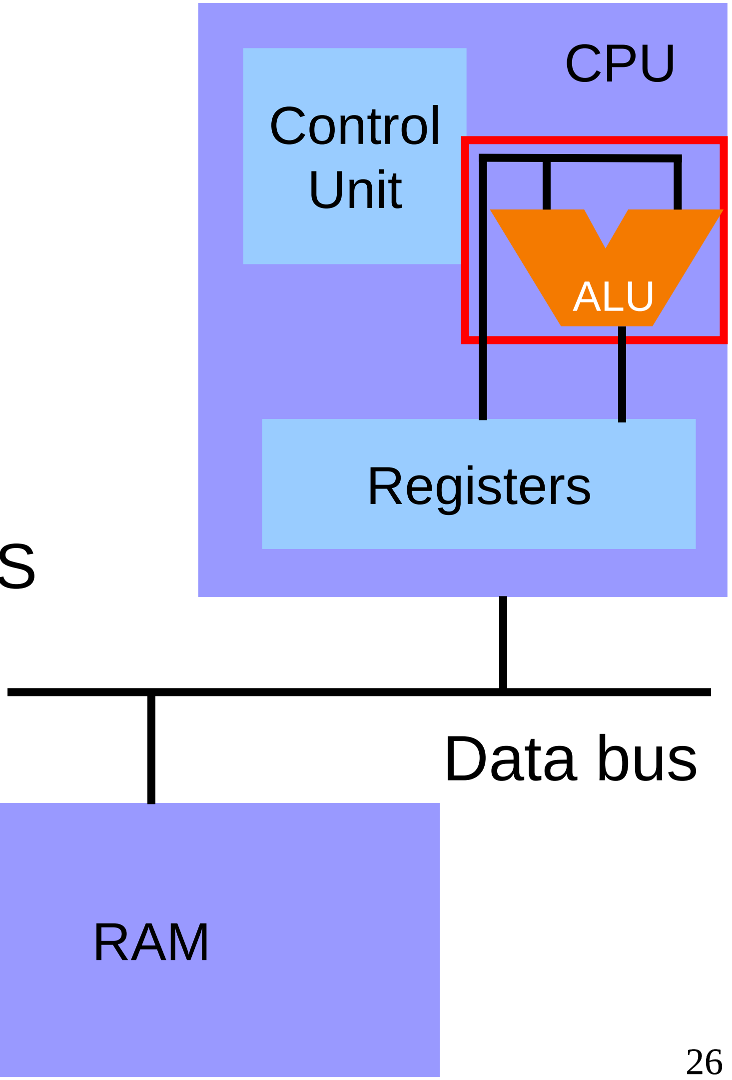
Many instructions combine steps

ALU

ALU (Arithmetic Logic Unit)

- Performs arithmetic and logic operations

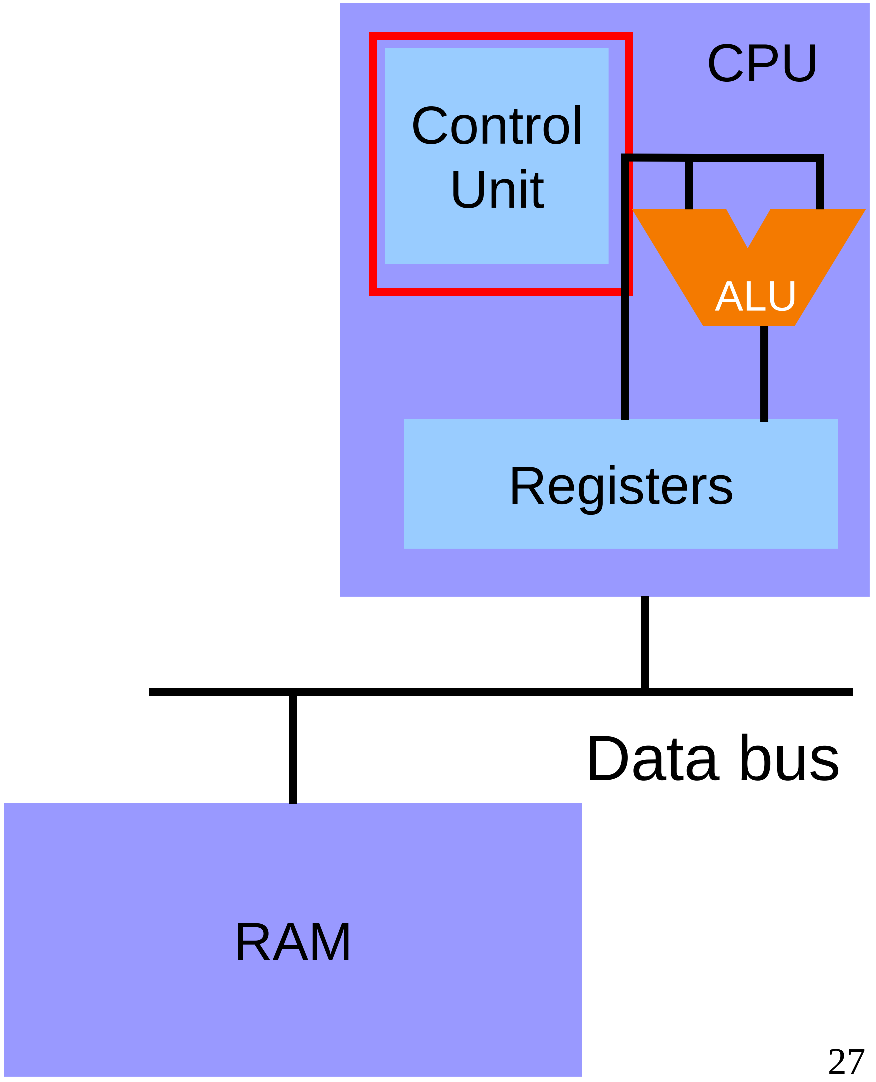




Control Unit

Control Unit

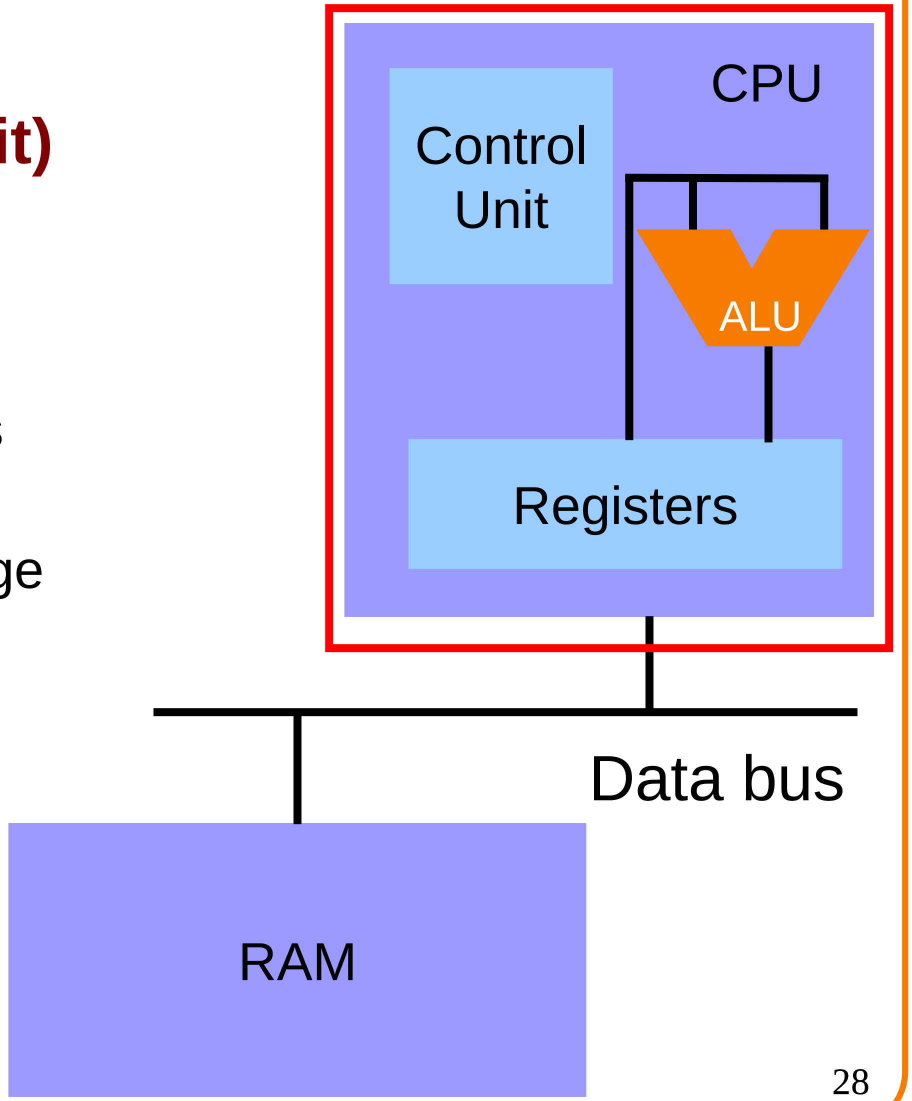
- Fetches and decodes each machine-language instruction
- Sends proper data to ALU



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: Performing Arithmetic

Assembly Language: Control-flow instructions

Instruction Format

Many instructions have this format:

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

- **name**: name of the instruction (**`mov`**, **`add`**, **`sub`**, **`and`**, etc.)

- **b**yte ⇒ operands are one-byte entities
- **w**ord ⇒ operands are two-byte entities
- **l**ong ⇒ operands are four-byte entities
- **q**uad ⇒ operands are eight-byte entitles

Instruction Format

Many instructions have this format:

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

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

Instruction Format

Many instructions have this format:

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

- **dest: destination operand**
  - The destination of data
  - Can be
    - **Register operand**: **`%rax`**, **`%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)

Registers

EAX	14
R10	

…

Memory

length	5
width	2
perim	14

Performing Arithmetic: Byte Data

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

Registers — EAX: | A | | | |

Memory — grade: | A | A | D | 0 |

Note:
- Comment
- `movb` instruction
- `subb` instruction
- `decb` instruction

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

```
      .section ".data"
grade:  .byte 'B'
        .byte 'A'
        .byte 'D'
        .byte 0
…
      .section ".text"
…
      # Option 1
      movb grade, %al
      subb $1, %al
      movb %al, grade
      …
      # Option 2
      subb $1, grade
      …
      # Option 3
      decb grade
```

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

- **`%rax`** ⇒ read from (or write to) register RAX
- 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

*if you're lucky

Notation

Instruction notation:

- q ⇒ quad (8 bytes); 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

```
mov{q,l,w,b} srcIRM, destRM    dest = src
movsb{q,l,w} srcRM, destR      dest = src (sign extend)
movsw{q,l} srcRM, destR        dest = src (sign extend)
movslq srcRM, destR            dest = src (sign extend)
movzb{q,l,w} srcRM, destR      dest = src (zero fill)
movzw{q,l} srcRM, destR        dest = src (zero fill)
movzlq srcRM, destR            dest = src (zero fill)

cqto             reg[RDX:RAX] = reg[RAX] (sign extend)
cltd             reg[EDX:EAX] = reg[EAX] (sign extend)
cwtl             reg[EAX] = reg[AX] (sign extend)
cbtw             reg[AX] = reg[AL] (sign extend)
```

`mov` is used often; others less so

Generalization: Arithmetic

Arithmetic instructions

```
add{q,l,w,b} srcIRM, destRM   dest += src
sub{q,l,w,b} srcIRM, destRM   dest -= src
inc{q,l,w,b} destRM           dest++
dec{q,l,w,b} destRM           dest--
neg{q,l,w,b} destRM           dest = -dest
```

Q: Is this adding signed numbers or unsigned?

A: Yes! [remember properties of 2's complement]

signed 2's complement

```
   3       0011B
+ -4     + 1100B
  --       ----
  -1       1111B
```

unsigned

```
   3       0011B
+ 12     + 1100B
  --       ----
  15       1111B
```

Generalization: Bit Manipulation

Bitwise instructions

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

signed (arithmetic right shift)

$44 / 2^2$	$00010110 0_B$
$= 11$	$00 0001011_B$
$-44 / 2^2$	$11101010 0_B$
$= -11$	$11 1110101_B$

copies of sign bit

unsigned (logical right shift)

$44 / 2^2$	$00010110 0_B$
$= 11$	$00 0001011_B$
$468 / 2^2$	$11101010 0_B$
$= 117$	$00 1110101_B$

zeros

Multiplication & Division

Signed

```
imulq srcRM    reg[RDX:RAX] = reg[RAX]*src
imull srcRM    reg[EDX:EAX] = reg[EAX]*src
imulw srcRM    reg[DX:AX] = reg[AX]*src
imulb srcRM    reg[AX] = reg[AL]*src
idivq srcRM    reg[RAX] = reg[RDX:RAX]/src
               reg[RDX] = reg[RDX:RAX]%src
idivl srcRM    reg[EAX] = reg[EDX:EAX]/src
               reg[EDX] = reg[EDX:EAX]%src
idivw srcRM    reg[AX] = reg[DX:AX]/src
               reg[DX] = reg[DX:AX]%src
idivb srcRM    reg[AL] = reg[AX]/src
               reg[AH] = reg[AX]%src
```

Unsigned

```
mulq srcRM     reg[RDX:RAX] = reg[RAX]*src
mull srcRM     reg[EDX:EAX] = reg[EAX]*src
mulw srcRM     reg[DX:AX] = reg[AX]*src
mulb srcRM     reg[AX] = reg[AL]*src
divq srcRM     reg[RAX] = reg[RDX:RAX]/src
               reg[RDX] = reg[RDX:RAX]%src
divl srcRM     reg[EAX] = reg[EDX:EAX]/src
               reg[EDX] = reg[EDX:EAX]%src
divw srcRM     reg[AX] = reg[DX:AX]/src
               reg[DX] = reg[DX:AX]%src
divb srcRM     reg[AL] = reg[AX]/src
               reg[AH] = reg[AX]%src
```

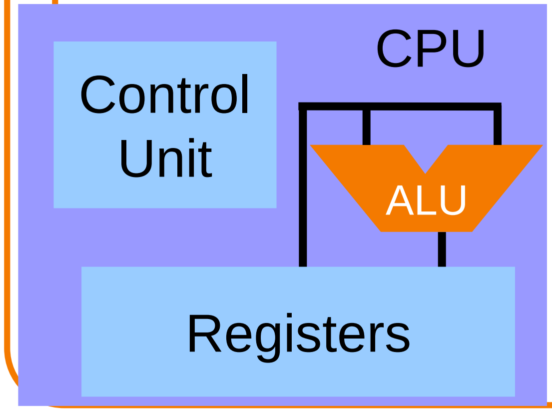
See Bryant & O’Hallaron book for description of signed vs. unsigned multiplication and division

Translation: C to x86-64

count↔r10d
n↔r11d

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

```
        movl    $0, %r10d
loop:
        cmpl    $1, %r11d
        jle     endloop

        addl    $1, %r10d

        movl    %r11d, %eax
        andl    $1, %eax
        je      else

        movl    %r11d, %eax
        addl    %eax, %r11d
        addl    %eax, %r11d
        addl    $1, %r11d

        jmp     endif
else:
        sarl    $1, %r11d
endif:
        jmp     loop
endloop:
```



Agenda

Language Levels

Architecture

Assembly Language: Performing Arithmetic

Assembly Language: Control-flow instructions

Control Flow with Signed Integers

Comparing (signed or unsigned) integers

`cmp{q,l,w,b} srcIRM, destRM`	Compare dest with src

- Sets condition-code bits in the EFLAGS register
- Beware: operands are in counterintuitive order
- Beware: many other instructions set condition-code bits
  - Conditional jump should **immediately** follow **`cmp`**

Control Flow with Signed Integers

Unconditional jump

```
jmp X   Jump to address X
```

Conditional jumps after comparing signed integers

```
je  X        Jump to X if equal
jne X        Jump to X if not equal
jl  X        Jump to X if less
jle X        Jump to X if less or equal
jg  X        Jump to X if greater
jge X        Jump to X if greater or equal
```

- Examine condition-code bits in EFLAGS register

Assembly lang. Machine lang.

```
        movl    $0, %r10d
loop:
        cmpl    $1, %r11d
        jle     endloop

        addl    $1, %r10d

        movl    %r11d, %eax
        andl    $1, %eax
        je      else

        movl    %r11d, %eax
        addl    %eax, %r11d
        addl    %eax, %r11d
        addl    $1, %r11d

        jmp     endif
else:
        sarl    $1, %r11d
endif:

        jmp     loop
endloop:
```

address:	contents (in hex)
1000:	41ba00000000
1006:	4183fb01
100a:	7e25
100c:	4183c201
1010:	4489d8
1013:	83242500000000001
101b:	740f
101d:	4489d8
1020:	4101c3
1023:	4101c3
1026:	4183c301
102a:	eb03
102c:	41d1fb
102f:	83c301
1031:	

25 = 2f−0a (hex)

Label *stands for* an address

```
        movl    $0, %r10d
loop:
        cmpl    $1, %r11d
        jle     endloop
        addl    $1, %r10d
        movl    %r11d, %eax
        andl    $1, %eax
        je      else
        movl    %r11d, %eax
        addl    %eax, %r11d
        addl    %eax, %r11d
        addl    $1, %r11d
        jmp     endif
else:
        sarl    $1, %r11d
endif:
        jmp     loop
endloop:
```

```
address:  contents (in hex)

1000:   41ba00000000
1006:   4183fb01
100a:   7e25
100c:   4183c201
1010:   4489d8
1013:   83242500000000001
101b:   740f
101d:   4489d8
1020:   4101c3
1023:   4101c3
1026:   4183c301
102a:   eb03
102c:   41d1fb
102f:   83c301
1031:
```

$25 = 31 - 0c$ (hex)

Translation: C to x86-64

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

```
        movl    $0, %r10d
loop:
        cmpl    $1, %r11d
        jle     endloop

        addl    $1, %r10d

        movl    %r11d, %eax
        andl    $1, %eax
        je      else

        movl    %r11d, %eax
        addl    %eax, %r11d
        addl    %eax, %r11d
        addl    $1, %r11d

        jmp     endif
else:
        sarl    $1, %r11d
endif:
        jmp     loop
endloop:
```

Summary

Language levels

The basics of computer architecture

- Enough to understand x86-64 assembly language

The basics of x86-64 assembly language

- Registers
- Arithmetic
- Control flow

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`