# Assembly Language: Part 1

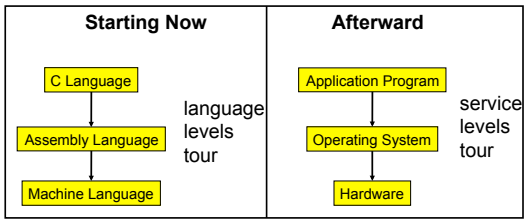
Jennifer Rexford

1

## Context of this Lecture

First half lectures: "Programming in the large"

Second half lectures: "Under the hood"




2

## Goals of this Lecture

Help you learn:

- Language levels
- The basics of x86-64 **architecture**
  - Enough to understand x86-64 assembly language
- The basics of x86-64 **assembly language**
  - Instructions to define global data
  - Instructions to transfer data and perform arithmetic

3

## 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 |

4

## Agenda

**Language Levels**

Architecture

Assembly Language: Defining Global Data

Assembly Language: Performing Arithmetic

5

## High-Level Languages

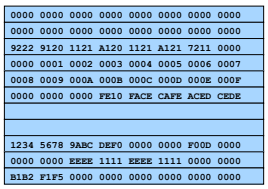
Characteristics

- 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;
}
```

6

## 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
```

7

## 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, %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:
```

8

## 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!

9

## Why Learn x86-64 Assembly Lang?

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

Pros
- X86-64 is popular
- FC010 computers are x86-64 computers
  - Program natively on FC010 instead of using an emulator

Cons
- X86-64 assembly language is **big**
  - Each instruction is simple, but…
  - There are **many** instructions
  - Instructions differ widely

10

## x86-64 Assembly Lang Subset

We'll study a popular subset
- As defined by precept ***x86-64 Assembly Language*** document

We'll study programs define functions that:
- Do not use floating point values
- Have parameters that are integers or addresses (but not structures)
- Have return values that are integers or addresses (but not structures)
- Have no more than 6 parameters

Claim: a reasonable subset

11

## Agenda

Language Levels

**Architecture**

Assembly Language: Defining Global Data

Assembly Language: Performing Arithmetic

12

## John Von Neumann (1903-1957)

In computing
- Stored program computers
- Cellular automata
- Self-replication

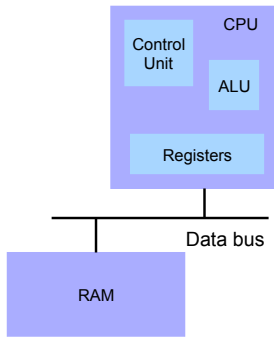
Other interests
- Mathematics
- Nuclear physics (hydrogen bomb)

Princeton connection
- Princeton Univ & IAS, 1930-death

Known for "Von Neumann architecture"
- In contrast to less successful "Harvard architecture"

13

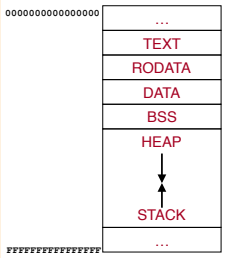
## Von Neumann Architecture

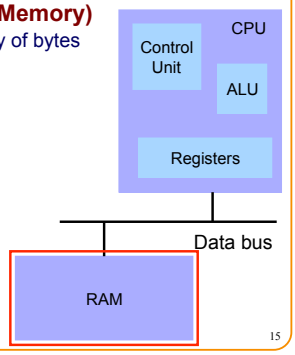



14

## RAM

**RAM (Random Access Memory)**
- Conceptually: large array of bytes

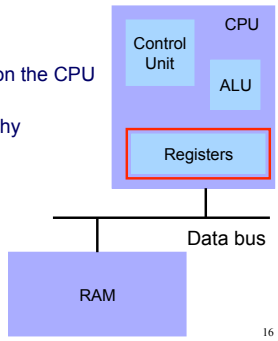



15

## Registers

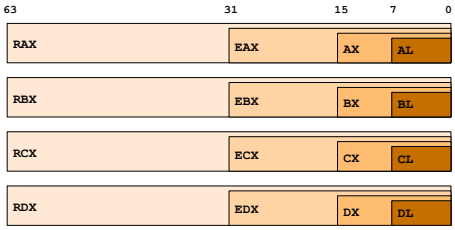
**Registers**
- Small amount of storage on the CPU
- Much faster than RAM
- Top of the storage hierarchy
  - Above RAM, disk, …




16

## Registers

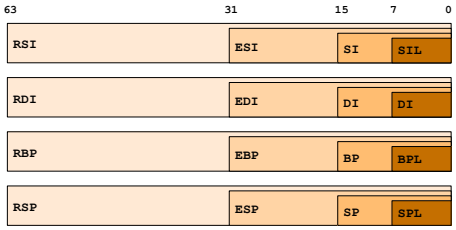
**General purpose registers:**




17

## Registers

**General purpose registers (cont.):**

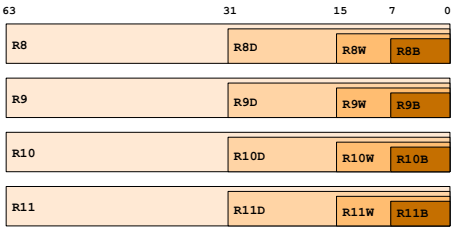



RSP is unique; see upcoming slide

18

## Registers

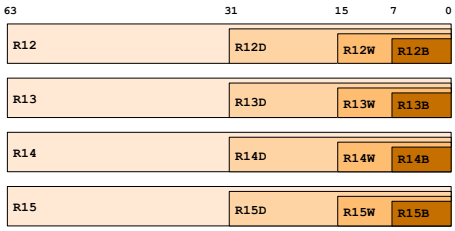
**General purpose registers (cont.):**




19

## Registers

**General purpose registers (cont.):**

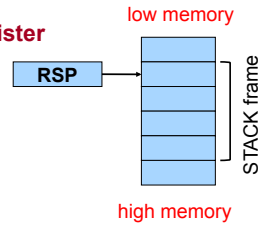



20

## 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)

21

## 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)

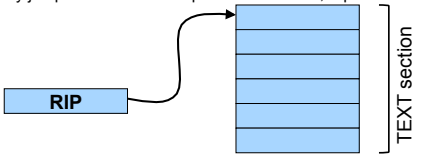
22

## 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




23

## Registers and RAM

Typical pattern:

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

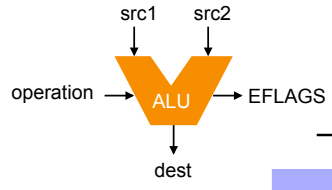
Many instructions combine steps

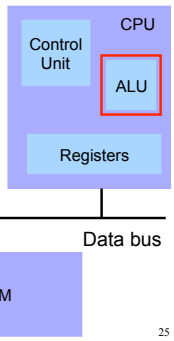
24

## ALU

**ALU (Arithmetic Logic Unit)**

- Performs arithmetic and logic operations



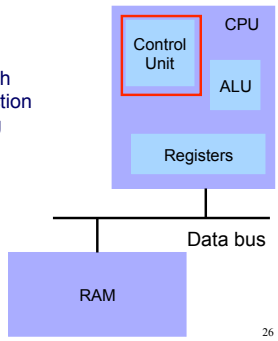




25

## Control Unit

**Control Unit**

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

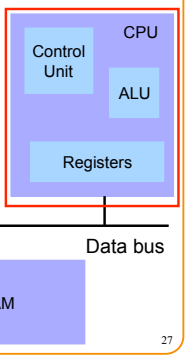



26

## CPU

**CPU (Central Processing Unit)**

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




27

## Agenda

Language Levels

Architecture

**Assembly Language: Defining Global Data**

Assembly Language: Performing Arithmetic

28

## Defining Data: DATA Section 1

```
static char c = 'a';
static short s = 12;
static int i = 345;
static long l = 6789;
```

```
    .section ".data"
c:
    .byte 'a'
s:
    .word 12
i:
    .long 345
l:
    .quad 6789
```

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)
`.quad` instruction (8 bytes)

Note:

Best to avoid "word" (2 byte) data

29

## Defining Data: DATA Section 2

```
char c = 'a';
short s = 12;
int i = 345;
long l = 6789;
```

```
    .section ".data"
    .globl c
c: .byte 'a'
    .globl s
s: .word 12
    .globl i
i: .long 345
    .globl l
l: .quad 6789
```

Note:

Can place label on same line as next instruction
`.globl` instruction

30

## Defining Data: BSS Section

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

```
    .section ".bss"
c:
    .skip 1
s:
    .skip 2
i:
    .skip 4
l:
    .skip 8
```

Note:
`.section` instruction (to announce BSS section)
`.skip` instruction

31

## Defining Data: RODATA Section

```
…
…"hello\n"…;
…
```

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

Note:
`.section` instruction (to announce RODATA section)
`.string` instruction

32

## Agenda

Language Levels

Architecture

Assembly Language: Defining Global Data

**Assembly Language: Performing Arithmetic**

33

## 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

34

## 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`

35

## 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**

36

## 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)

37

## Performing Arithmetic: Byte Data

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

```
   .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
```

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

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

38

## 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
- `%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

39

## Generalization: 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

40

## 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

41

## 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
```

42

## Generalization: Signed Mult & Div

Signed multiplication and division instructions

```
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
```

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

43

## Generalization: Unsigned Mult & Div

Unsigned multiplication and division instructions

```
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

44

## 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)
```

45

## Summary

Language levels

The basics of computer architecture
- Enough to understand x86-64 assembly language

The basics of x86-64 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`

46

## Appendix

Big-endian vs. little-endian byte order

47

## 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 |

48

## Byte Order Example 1

```
#include <stdio.h>
int main(void)
{  unsigned int i = 0x00337ff;
   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
```

49

## Byte Order Example 2

Note:
Flawed code; uses "b" instructions to manipulate a four-byte memory area

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

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

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

50

## Byte Order Example 3

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

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

What would happen?

51