


Princeton University
Computer Science 217: Introduction to Programming Systems

**Assembly Language:
Part 1**



1

Context of this Lecture

First half of the semester: "Programming in the large"
Second half: "Under the hood"

Starting Now

C Language
↓
Assembly Language
↓
Machine Language

language levels tour

Afterward

Application Program
↓
Operating System
↓
Hardware

service levels tour

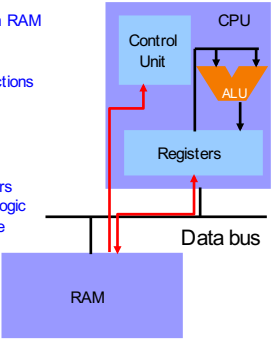
2

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



3

Agenda

Language Levels

Instruction-Set Architecture (ISA)
Assembly Language: Defining global data
Assembly Language: Performing Arithmetic
Assembly Language: Control-flow instructions

4

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;
}
```

5

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
0003	0001	0002	0003	0004	0005	0006	0007
0008	0009	000A	000B	000C	000D	000E	000F
0000	0000	0000	F010	F00E	C0FE	A0ED	C0EE
1234	5678	9ABC	DEF0	0000	0000	F00D	0000
0000	0000	EEEE	1111	EEEE	1111	0000	0000
0102	0103	0000	0000	0000	0000	0000	0000

6

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

Translation: C to x86-64

```

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

```

count ← r10d
n ← r11d
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
else:  jmp  endif
      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: Defining global data
- 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!

The diagram shows a purple box representing the CPU, containing a 'Control Unit', 'Registers', and an 'ALU'. Below the CPU is a blue box representing 'RAM'. A horizontal line labeled 'Data bus' connects the CPU to the RAM.

13

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"

A black and white portrait of John Von Neumann, a middle-aged man with glasses, wearing a suit and tie.

14

Von Neumann Architecture

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

The diagram shows the same CPU and RAM as in slide 13. A red arrow points from the RAM to the 'Registers' in the CPU, with the text 'Instructions are fetched from RAM' next to it. Another red arrow points from the 'Registers' to the 'Control Unit'.

15

Registers

Registers

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

The diagram shows the CPU and RAM from slide 15. The 'Registers' box within the CPU is highlighted with a red border.

16

Registers (x86-64 architecture)

General purpose registers:

63	31	15	7	0
RAX	EAX	AX	AL	
RBX	EBX	BX	BL	
RCX	ECX	CX	CL	
RDX	EDX	DX	DL	

17

Registers (x86-64 architecture)

General purpose registers (cont.):

63	31	15	7	0
RSI	ESI	SI	SIL	
RDI	EDI	DI	DIL	
RBP	EBP	BP	BPL	
RSP	ESP	SP	SPL	

RSP is unique; see upcoming slide

18

Registers (x86-64 architecture)

General purpose registers (cont.):

63	31	15	7	0
R8	R8D	R8W	R8B	
R9	R9D	R9W	R9B	
R10	R10D	R10W	R10B	
R11	R11D	R11W	R11B	
R12	R12D	R12W	R12B	
R13	R13D	R13W	R13B	
R14	R14D	R14W	R14B	
R15	R15D	R15W	R15B	

19

Registers summary

16 general-purpose 64-bit pointer/long-integer registers
 rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15

sometimes used as a "frame pointer" or "base pointer" (points to rbp)
 "stack pointer" (points to rsp)

If you're operating on 32-bit "int" data, use these names instead:
 eax, ebx, ecx, edx, esi, edi, ebp, esp, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d

it doesn't really make sense to put 32-bit ints in the stack pointer (points to rsp)

20

RSP Register

RSP (Stack Pointer) register

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

low memory
 high memory
 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

TEXT section

23

Registers summary

16 general-purpose 64-bit pointer/long-integer registers
 rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15

sometimes used as a "frame pointer" or "base pointer" (points to rbp)
 "stack pointer" (points to rsp)

If you're operating on 32-bit "int" data, use these names instead:
 eax, ebx, ecx, edx, esi, edi, ebp, esp, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d

it doesn't really make sense to put 32-bit ints in the stack pointer (points to rsp)

2 special-purpose registers: `eflags` ("condition codes"), `rip` ("program counter")

24

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

25

ALU

ALU (Arithmetic Logic Unit)

- Performs arithmetic and logic operations

26

Control Unit

Control Unit

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

27

CPU

CPU (Central Processing Unit)

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

28

Agenda

- Language Levels
- Architecture
- Assembly Language: Defining global data**
- Assembly Language: Performing Arithmetic
- Assembly Language: Control-flow instructions

29

RAM

RAM (Random Access Memory)

30

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

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

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

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**
- 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.)
- byte ⇒ operands are one-byte entities
- word ⇒ operands are two-byte entities
- long ⇒ operands are four-byte entities
- quad ⇒ operands are eight-byte entities

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

37

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**

38

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	Memory
EAX: 14	length: 5
R10:	width: 2
...	perim: 14
...	...

39

Performing Arithmetic: Byte Data

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

```
.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
```

Registers: EAX [A][][][] Memory: grade [A][B][D][0]

Note:

- Comment**
- movb instruction**
- subb instruction**
- decb instruction**

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

40

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

41

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

42

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)
movzql 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		unsigned	
3	0011 _b	3	0011 _b
+ -4	+ 1100 _b	+ 12	+ 1100 _b
--	----	--	----
-1	1111 _b	15	1111 _b

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 _b
= 11	00000101 _b
-44 / 2 ²	11101010 _b
= -11	11101011 _b

copies of sign bit

unsigned (logical right shift)

44 / 2 ²	00010110 _b
= 11	00000101 _b
468 / 2 ²	11101010 _b
= 117	00110101 _b

zeros

Multiplication & Division

Signed

```

imulq srcRM    reg[RDX:RAX] = reg[RDX:RAX] * src
imull srcRM    reg[RDX:RAX] = reg[RDX:RAX] * src
imulw srcRM    reg[DX:AX] = reg[AX] * src
imulb srcRM    reg[AX] = reg[AL] * src
idivq srcRM    reg[RAX] = reg[RDX:RAX] / src
    
```

Unsigned

```

mulq srcRM    reg[RDX:RAX] = reg[RDX:RAX] * src
mull srcRM    reg[RDX:RAX] = reg[RDX:RAX] * src
mulw srcRM    reg[DX:AX] = reg[AX] * src
mulb srcRM    reg[AX] = reg[AL] * src
divq srcRM    reg[RAX] = reg[RDX:RAX] / src
    
```

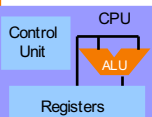
See Bryant & O' Hallaron book for description of signed vs. unsigned multiplication and division

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

else:
    jmp endif

endif:
    sarl $1, %r11d

    jmp loop
endloop:
    
```

Agenda

- Language Levels
- Architecture
- Assembly Language: Defining global data
- Assembly Language: Performing Arithmetic
- Assembly Language: Control-flow instructions

Control Flow with Signed Integers

Comparing (signed or unsigned) integers

```
cmp(q,l,w,b) srcImm, destImm    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`

49

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

50

Assembly lang. Machine lang.

<pre> loop: movl \$0, %r10d cmpl \$1, %r1ld jle endloop addl \$1, %r10d movl %r1ld, %eax andl \$1, %eax je else movl %r1ld, %eax addl %eax, %r1ld addl %eax, %r1ld addl \$1, %r1ld else: jmp endif sarl \$1, %r1ld endif: jmp loop endloop: </pre>	<pre> address: contents (in hex) 1000: 41ba00000000 1006: 4183fb01 100a: 7e20 20 = 102c - 100c (hex) 100c: 4183c201 1010: 4489d8 1013: 83e001 1016: 740f 1018: 4489d8 101b: 4101c3 101e: 4101c3 1021: 4183c301 1025: eb03 1027: 41d1fb 102a: ebda 102c: </pre>
---	---

51

Label stands for an address

<pre> loop: movl \$0, %r10d cmpl \$1, %r1ld jle endloop addl \$1, %r10d movl %r1ld, %eax andl \$1, %eax je else movl %r1ld, %eax addl %eax, %r1ld addl %eax, %r1ld addl \$1, %r1ld else: jmp endif sarl \$1, %r1ld endif: jmp loop endloop: </pre>	<pre> address: contents (in hex) 1000: 41ba00000000 1006: 4183fb01 100a: 7e20 20 = 102c - 100c (hex) 100c: 4183c201 1010: 4489d8 1013: 83e001 1016: 740f 1018: 4489d8 101b: 4101c3 101e: 4101c3 1021: 4183c301 1025: eb03 1027: 41d1fb 102a: ebda 102c: </pre>
---	---

52

Translation: C to x86-64

<pre> count = 0; while (n > 1) { count++; if (n & 1) n = n * 3 + 1; else n = n / 2; } </pre>	<pre> loop: movl \$0, %r10d cmpl \$1, %r1ld jle endloop addl \$1, %r10d movl %r1ld, %eax andl \$1, %eax je else movl %r1ld, %eax addl %eax, %r1ld addl %eax, %r1ld addl \$1, %r1ld else: jmp endif sarl \$1, %r1ld endif: jmp loop endloop: </pre>
---	---

53

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`

54

Appendix

Big-endian vs little-endian byte order

55

Byte Order

x86-64 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	0000101
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	0000101

56

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

57

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
```

x86-64 is little endian, so what will be the value of grade?

What would be the value of grade if x86-64 were big endian?

58

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?

59