



# Assembly Language: Overview

1



## Goals of this Lecture

- Help you learn:
  - The basics of computer architecture
  - The relationship between C and assembly language
  - IA-32 assembly language, through an example

2



## Three Levels of Languages

3



## High-Level Language

- Examples: C, C++, Java, Pascal, ...
- Make programming easier by describing operations in a semi-natural language
- Increase the portability of the code
- One line of code may involve many low-level operations

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

4

## Assembly Language



- E.g., IA-32 from Intel
- Tied to specifics of the underlying machine
- Commands and names represent bit patterns, to make code readable, writable by humans
- Hand-coded assembly may be more efficient than what compiler generates

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

## Machine Language



- Also tied to the underlying machine
- What hardware sees and deals with
- Every instruction is a sequence of one or more numbers
- All stored in memory on the computer, and read and executed
- Unreadable by humans

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

## Why Learn Assembly Language?



- Write faster code (even in high-level language)
  - By understanding which high-level constructs are more efficient at the machine level
- Understand how things work underneath
  - Learn the basic organization of the underlying machine
  - Learn how the computer actually runs a program
  - Design better computers in the future
- Some software is written in assembly language
  - Code that really needs to run quickly
  - Code for embedded systems, network processors, etc.

7

## Why Learn Intel IA-32 Assembly?



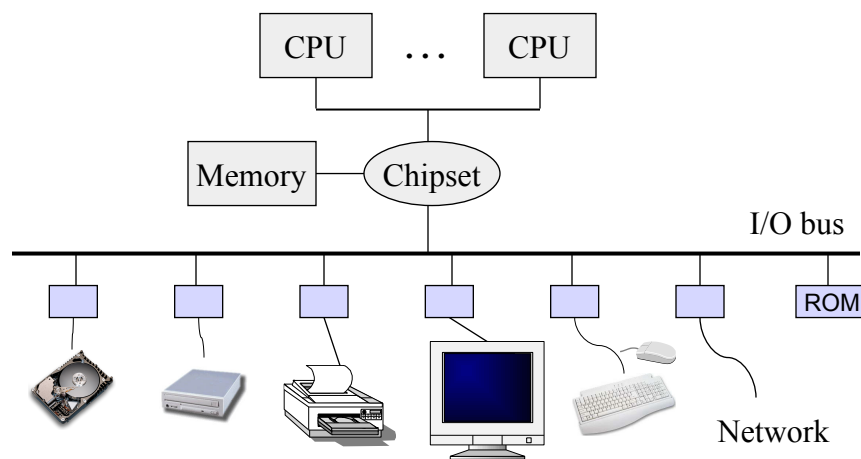
- Program natively on our computing platform
  - Rather than using an emulator to mimic another machine
- Learn instruction set for the most popular platform
  - Most likely to work with Intel platforms in the future
- But, this comes at some cost in complexity
  - IA-32 has a large and varied set of instructions
  - More instructions than are really useful in practice
- Fortunately, you won't need to use everything

8



# Computer Architecture

## A Typical Computer

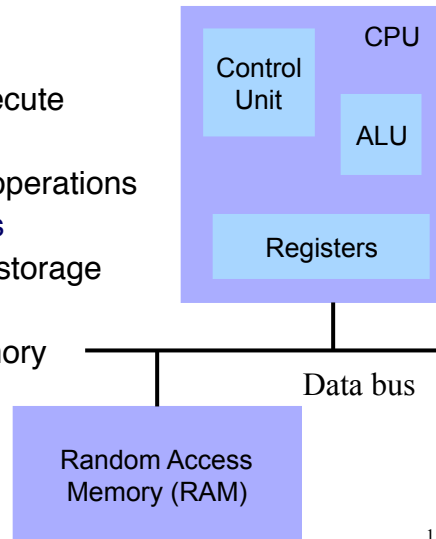


# Von Neumann Architecture



## • Central Processing Unit

- Control unit
  - Fetch, decode, and execute
- Arithmetic and logic unit
  - Execution of low-level operations
- General-purpose registers
  - High-speed temporary storage
- Data bus
  - Provide access to memory



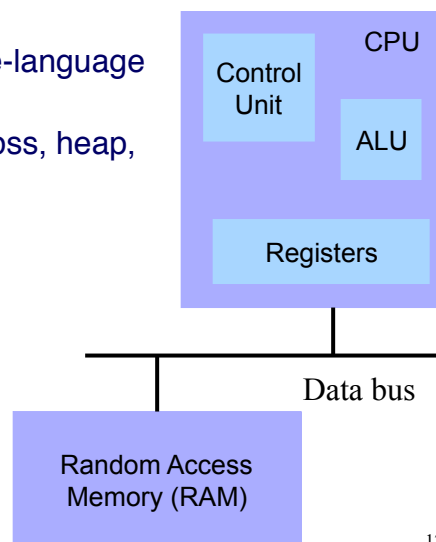
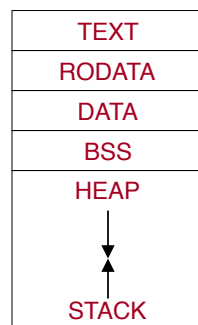
11

# Von Neumann Architecture



## • Memory

- Store executable machine-language instructions (text section)
- Store data (rodata, data, bss, heap, and stack sections)

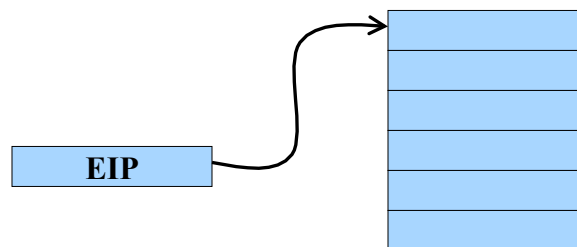


12

## Control Unit: Instruction Pointer



- Stores the location of the next instruction
  - Address to use when reading machine-language instructions from memory (i.e., in the text section)
- Changing the instruction pointer (EIP)
  - Increment it to go to the next instruction
  - Or, load a new value into EIP to “jump” to a new location

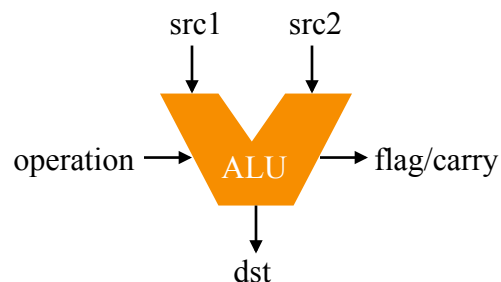


13

## Control Unit: Instruction Decoder



- Determines what operations need to take place
  - Translate the machine-language instruction
- Control what operations are done on what data
  - E.g., control what data are fed to the ALU
  - E.g., enable the ALU to do multiplication or addition
  - E.g., read from a particular address in memory



14

## Registers



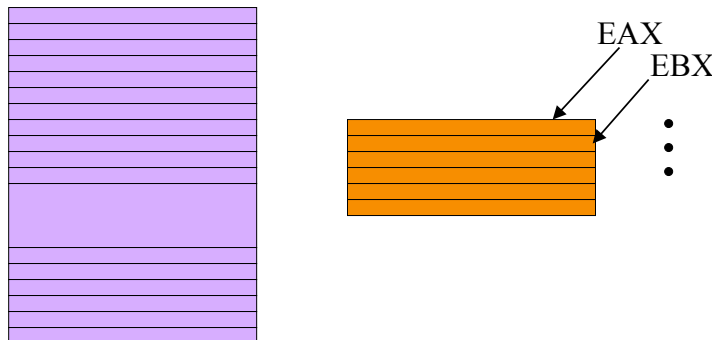
- Small amount of storage on the CPU
  - Can be accessed more quickly than main memory
  - Each register has a name, which assembly code uses
- Instructions move data in and out of registers
  - Loading (into) registers from main memory
  - Storing (from) registers to main memory
- Instructions manipulate the register contents
  - Registers essentially act as temporary variables
  - For efficient manipulation of the data
- Registers are the top of the memory hierarchy
  - Ahead of caches, main memory, disk, tape, ...

15

## Keeping it Simple: All 32-bit Words



- Simplifying assumption: all data in four-byte units
  - Memory is 32 bits wide
  - Registers are 32 bits wide



- In practice, can manipulate different sizes of data

16





## C Code vs. Assembly Code

17

## Kinds of Instructions



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

- **Reading and writing data**
  - count = 0
  - n
- **Arithmetic and logic operations**
  - Increment: count++
  - Multiply: n \* 3
  - Divide: n/2
  - Logical AND: n & 1
- **Checking results of comparisons**
  - Is (n > 1) true or false?
  - Is (n & 1) non-zero or zero?
- **Changing the flow of control**
  - To the end of the while loop (if “n > 1”)
  - Back to the beginning of the loop
  - To the else clause (if “n & 1” is 0)

18

## Variables in Registers



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

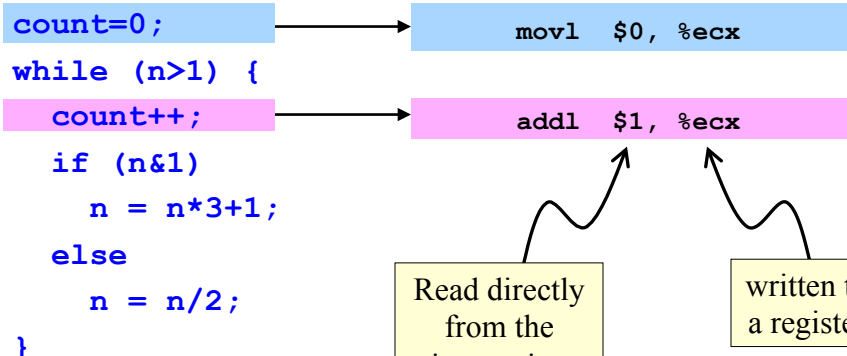
### Registers

**n**            **%edx**  
**count**      **%ecx**

Referring to a register: percent sign (“%”)

19

## Immediate and Register Addressing



Referring to a immediate operand: dollar sign (“\$”)

20

## Immediate and Register Addressing



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

```
movl %edx, %eax
andl $1, %eax
```

Computing intermediate value in register EAX

21

## Immediate and Register Addressing



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

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

Update %edx this time, since changing value of n

22

## Immediate and Register Addressing



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

→

```
sarl $1, %edx
```

23

## Changing Program Flow



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

- Cannot simply run next instruction
  - Check result of a previous operation
  - Jump to appropriate next instruction
- Flags register (EFLAGS)
  - Stores the status of operations, such as comparisons, as a side effect
  - E.g., last result was positive, negative, zero, etc.
- Jump instructions
  - Load new address in instruction pointer
- Example jump instructions
  - Jump unconditionally (e.g., “}”)
  - Jump if zero (e.g., “while (n&1)”)
  - Jump if greater/less (e.g., “if (n>1)”) 24

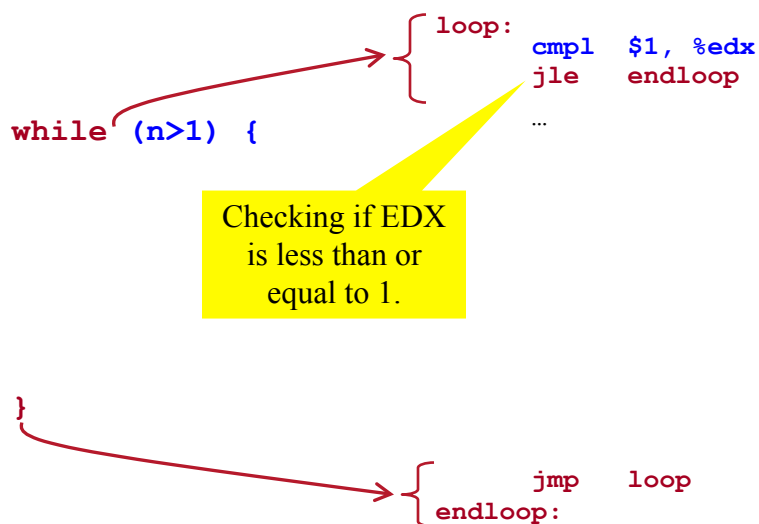
## Conditional and Unconditional Jumps



- Comparison `cmp` compares two integers
  - Done by subtracting the first number from the second
    - Discarding the results, but setting flags as a side effect
  - Example:
    - `cmp $1, %edx` (computes `%edx - 1`)
    - `jle endloop` (checks whether result was 0 or negative)
- Logical operation `and` compares two integers
  - Example:
    - `and $1, %eax` (bit-wise AND of `%eax` with 1)
    - `je else` (checks whether result was 0)
- Also, can do an unconditional branch `jmp`
  - Example:
    - `jmp endif` and `jmp loop`

25

## Jump and Labels: While Loop



26

## Jump and Labels: While Loop



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

```
loop:
    movl $0, %ecx
    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
    jmp endif
else:
    sarl $1, %edx
endif:
    jmp loop
endloop:
```

Diagram illustrating the mapping between C code and assembly code for a while loop. Red arrows point from the C code to the corresponding assembly instructions. The C code shows a while loop with an if-else statement. The assembly code shows the equivalent instructions, including labels for the loop and the end of the loop.

27

## Jump and Labels: If-Then-Else



```
if (n&1)
    ...
else
    ...
```

```
movl %edx, %eax
andl $1, %eax
je else
...
jmp endif
else:
...
endif:
```

Diagram illustrating the mapping between C code and assembly code for an if-then-else statement. Red arrows point from the C code to the corresponding assembly instructions. The C code shows an if-then-else statement. The assembly code shows the equivalent instructions, including labels for the 'then' block, the 'else' block, and the end of the if-then-else statement.

28

## Jump and Labels: If-Then-Else



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

“then” block

“else” block

```
loop:
    movl $0, %ecx
    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
        jmp  endif
    else:
        sarl $1, %edx
    endif:
    jmp  loop
endloop:
```

29

## Making the Code More Efficient...



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

Replace with  
“jmp loop”

```
loop:
    movl $0, %ecx
    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
        jmp  endif
    else:
        sarl $1, %edx
    endif:
    jmp  loop
endloop:
```

30

## Complete Example

n %edx  
count %ecx



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

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

31

## Reading IA-32 Assembly Language



- Referring to a register: percent sign (“%”)
  - E.g., “%ecx” or “%eip”
- Referring to immediate operand: dollar sign (“\$”)
  - E.g., “\$1” for the number 1
- Storing result: typically in the second argument
  - E.g. “addl \$1, %ecx” increments register ECX
  - E.g., “movl %edx, %eax” moves EDX to EAX
- Assembler directives: starting with a period (“.”)
  - E.g., “.section .text” to start the text section of memory
- Comment: pound sign (“#”)
  - E.g., “# Purpose: Convert lower to upper case”

32



## Conclusions



- **Assembly language**
  - In between high-level language and machine code
  - Programming the “bare metal” of the hardware, but mnemonically and not just with bits (machine language)
  - Loading and storing data, arithmetic and logic operations, checking results, and changing control flow
- **To get more familiar with IA-32 assembly**
  - Read more assembly-language examples
    - Chapter 3 of Bryant and O’Hallaron book
  - Generate your own assembly-language code
    - `gcc217 -S -O2 code.c`