CSC 2400: Computer Systems

Towards the Hardware

1

Towards the Hardware

- High-level language (Java)
  - High-level language (C) →
    assembly language →
    machine language (IA-32)

2

High-Level Language

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

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

3

Assembly Language

- Tied to the specifics of the underlying machine
- Commands and names to make the code readable and writeable by humans
- E.g., IA-32 from Intel

```
      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 the computer 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
```

5

Lecture Goals

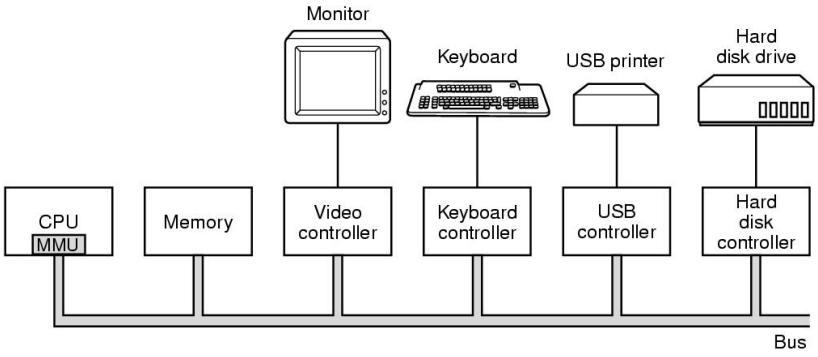
- Help you learn…
  - Basics of computer architecture
  - Relationship between C and assembly language
  - IA-32 assembly language through an example
- Why do you need to know this?

6

Computer Architecture

7

A Typical Computer

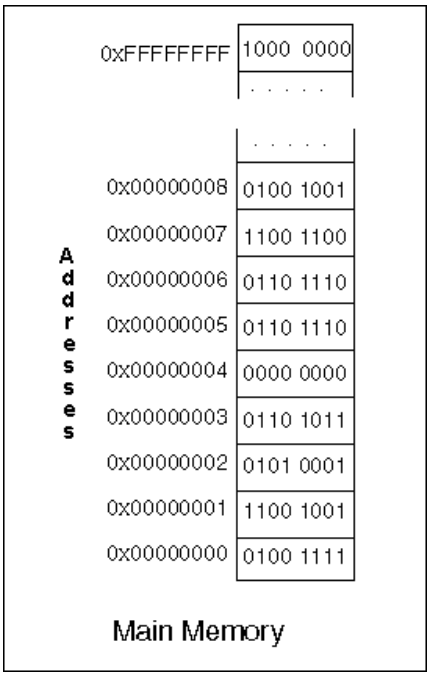


This model provides a good study framework.

8

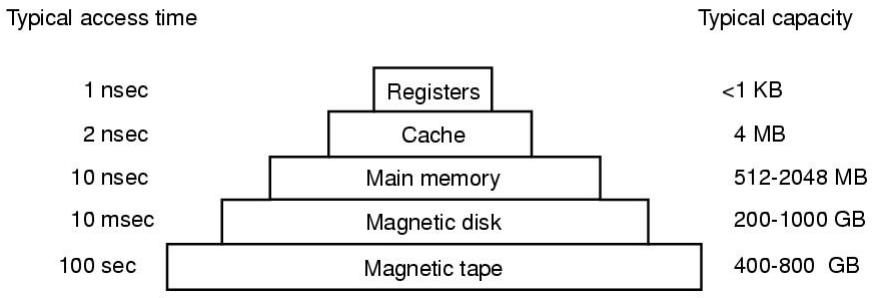
Memory

- The only large storage area that CPU can access directly
- Hence, any program executing must be in memory



Memory Hierarchy

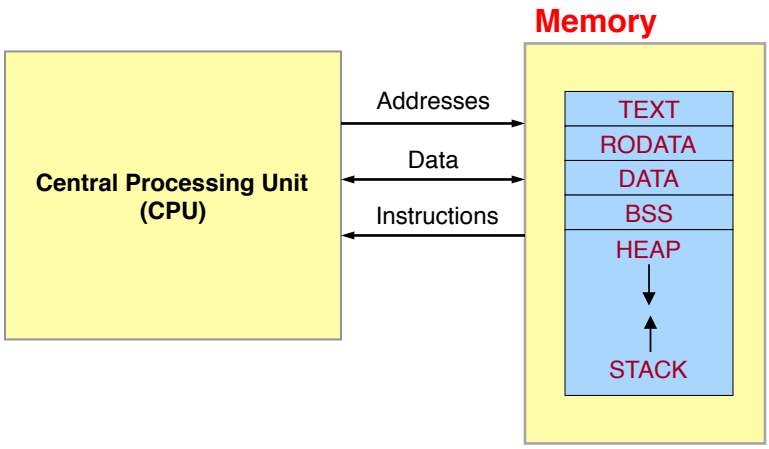
A typical memory hierarchy.
The numbers are very rough approximations.



Cache Principle
The more frequently data is accessed, the faster the access should be.

10

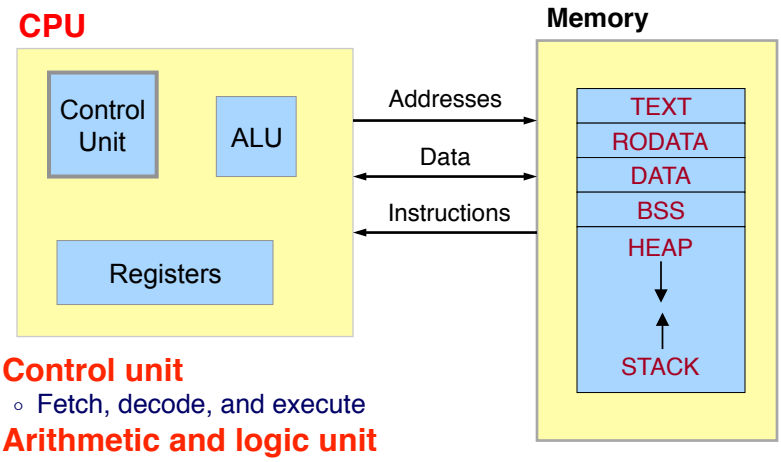
Memory (Unix Process Layout)



- Stores executable machine-language instructions (TEXT)
  Stores data (rodata, data, bss, heap, and stack sections)

11

Central Processing Unit (CPU)



- **Control unit**
  - Fetch, decode, and execute
- **Arithmetic and logic unit**
  - Execution of low-level operations
- **Registers**
  - High-speed temporary storage

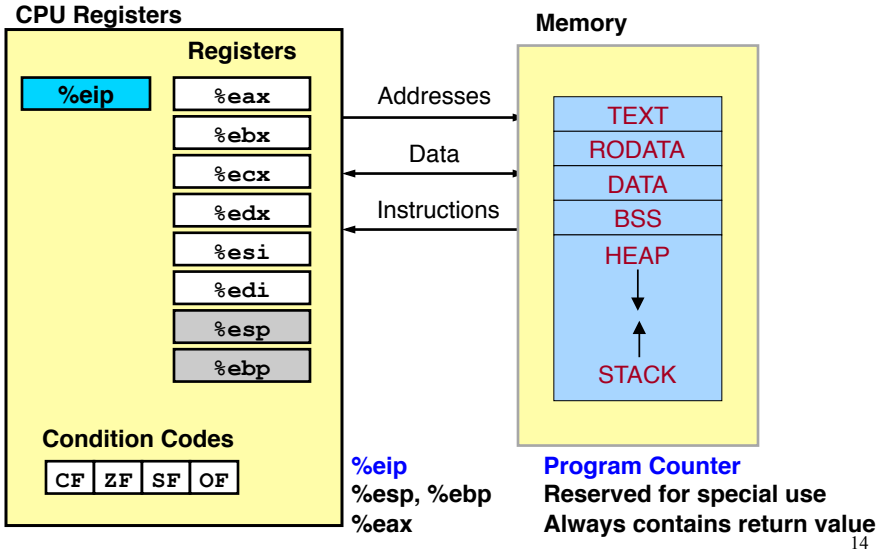
12

Registers

- Small amount of storage on the CPU
  - Can be accessed more quickly than main memory
- Instructions move data in and out of registers
  - Loading registers from main memory
  - Storing 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 main memory, disk, tape, …

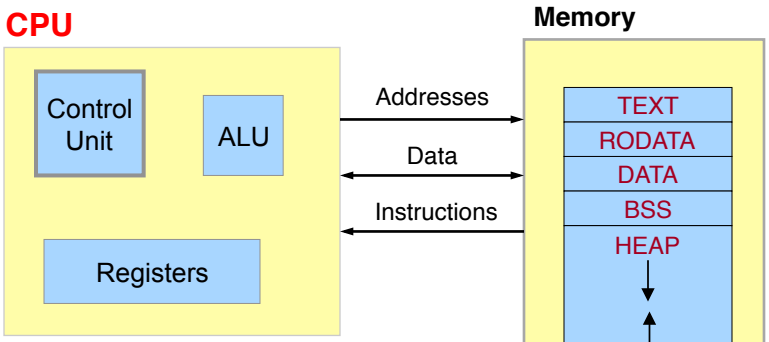
13

IA-32 Architecture



14

CPU – Control Unit and ALU

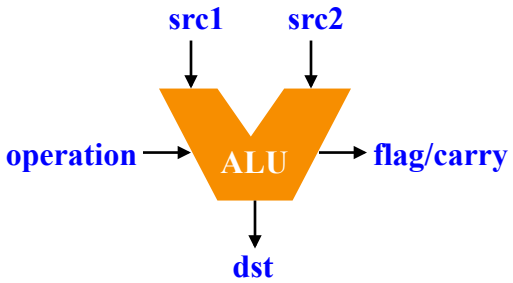


- **Control unit**
  - Fetch, decode, and execute
- **Arithmetic and logic unit (ALU)**
  - Execution of low-level operations

15

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

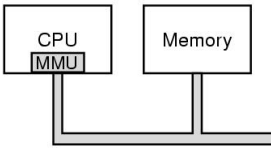


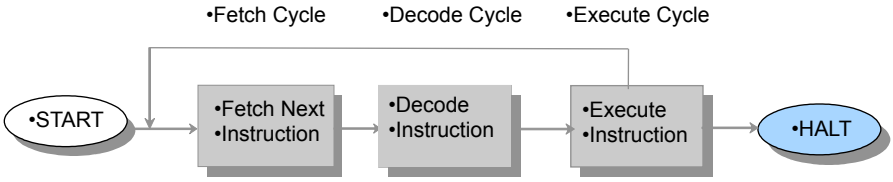
16

Central Processing Unit (CPU)

- Runs the loop

  Fetch-Decode-Execute





- Fetch the next instruction from memory
- Decode the instruction to figure out what to do
- Execute the instruction and store the result

17

Fetch-Decode-Execute Cycle

- Where is the “next instruction” held in the machine?
  - a CPU register called the Program Counter (PC) holds the address of the instruction to be fetched next
- Fetch cycle
  - Copy instruction from memory into Instruction Register (IR)
- Decode cycle
  - Decode instruction and fetch operands, if necessary
- Execute cycle
  - Execute the instruction
  - Increment PC by the instruction length after execution (assuming that all instructions are the same length)

18

C Code vs. Assembly Code

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)

Variables in Registers

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

Registers

```
count   %ecx
n       %edx
```

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

21

Immediate and Register Addressing

```
count   %ecx
n       %edx
```

```
count=0;                    movl  $0, %ecx
while (n>1) {
  count++;                  addl  $1, %ecx
  if (n&1)
    n = n*3+1;
  else
    n = n/2;
}
```

Read directly from the instruction

written to a register

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

22

General Syntax

op *Src, Dest*

Perform operation op on Src and Dest
Save result in Dest

23

Immediate and Register Addressing

```
count   %ecx
n       %edx
```

```
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

24

Immediate and Register Addressing

```
count   %ecx
n       %edx
```

```
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
```

Adding n twice is cheaper than multiplication!

25

Immediate and Register Addressing

```
count   %ecx
n       %edx
```

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

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

Shifting right by 1 bit is cheaper than division!

26

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
- **Jump instructions**
  - Load new address in instruction pointer
- **Example jump instructions**
  - Jump unconditionally (e.g., "}")
  - Jump if zero (e.g., "n&1")
  - Jump if greater/less (e.g., "n>1")

27

Jump Instructions

- Jump depends on the result of previous arithmetic instruction.

Jump	Description
`jmp`	**Unconditional**
`je`	**Equal / Zero**
`jne`	
`js`	
`jns`	
`jg`	
`jge`	
`jl`	
`jle`	

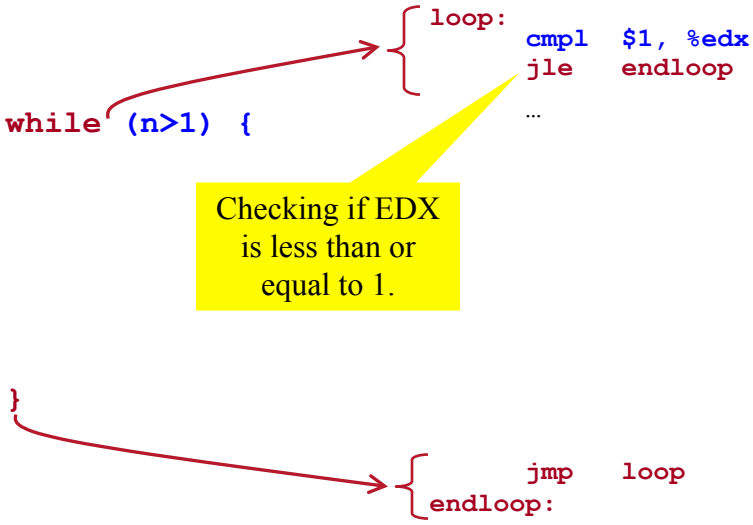
28

Conditional and Unconditional Jumps

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

Jump and Labels: While Loop

```
count   %ecx
n       %edx
```



Jump and Labels: While Loop

count	%ecx
n	%edx

```
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

Jump and Labels: If-Then-Else

count	%ecx
n	%edx

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

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

"then" block

"else" block

32

Jump and Labels: If-Then-Else

count	%ecx
n	%edx

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

"then" block

"else" block

```
      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:
```

33

Code More Efficient…

count	%ecx
n	%edx

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

Replace with "jmp loop"

```
      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:
```

34

Complete Example

```
count   %ecx
n       %edx
```

```
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:
```

35

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”

36

X86 → C Example

`int Example(int x){ ??? }`

```
0x80483c0     push %ebp
0x80483c1     mov  %esp,%ebp
0x80483c3     mov  0x8(%ebp)%ecx
0x80483c6     xor  %eax,%eax
0x80483c8     xor  %edx,%edx
0x80483ca     cmp  %ecx,%edx
0x80483cc     jge  0x80483d7
0x80483ce     mov  %esi,%esi
0x80483d0     add  %edx,%eax
0x80483d2     inc  %edx
0x80483d3     cmp  %ecx,%edx
0x80483d5     jl   0x80483d0
0x80483d7     mov  %ebp,%esp
0x80483d9     pop  %ebp
0x80483da     ret
```

Write comments

```
              # ecx = x
              # eax = 0
              # edx = 0
              # if (edx-x >=0)
              # goto L1
              # nop
L2:           # eax += edx
              # edx++
              # if (edx-x < 0)
              # goto L2
L1:
```

37

Name the variables

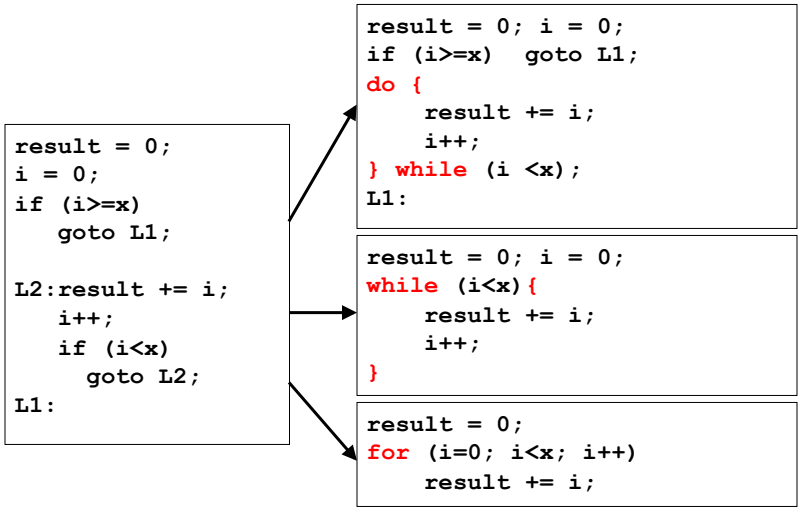
`eax ← result, edx ← i`

```
0x80483c0     push %ebp
0x80483c1     mov  %esp,%ebp
0x80483c3     mov  0x8(%ebp),%ecx
0x80483c6     xor  %eax,%eax
0x80483c8     xor  %edx,%edx
0x80483ca     cmp  %ecx,%edx
0x80483cc     jge  0x80483d7
0x80483ce     mov  %esi,%esi
0x80483d0     add  %edx,%eax
0x80483d2     inc  %edx
0x80483d3     cmp  %ecx,%edx
0x80483d5     jl   0x80483d0
0x80483d7     mov  %ebp,%esp
0x80483d9     pop  %ebp
0x80483da     ret
```

```
              # ecx = x
              # result = 0
              # i = 0
              # if (i >= x)
              # goto L1
              # nop
L2:           # result += i
              # i++
              # if (i < x)
              # goto L2
L1:
```

38

Identify the Loop



39

C Code

```
int Example(int x)
{
     int result=0;
     int i;
     for (i=0; i<x; i++)
        result += i;
     return result;
}
```

40

Exercise

int F(int x, int y){ ??? }

```
        push %ebp
        movl %esp,%ebp
        movl 8(%ebp),%ecx
        movl 12(%ebp),%edx
        xorl %eax,%eax
        cmpl %edx,%ecx
        jle .L1
.L2:    decl %ecx
        incl %edx
        incl %eax
        cmpl %edx,%ecx
        jg .L2
.L1:    incl %eax
        movl %ebp,%esp
        popl %ebp
        ret
```

```
                # ecx = x
                # edx = y
.L2:

.L1:
```

41

C Code

```
int F(int x, int y)
```

42

Conclusions

- Hardware
  - Memory is the only storage area CPU can access directly
  - Executables are stored on the disk
  - Fetch-Decode-Execute Cycle for running executables
- Assembly language
  - In between high-level language and machine code
  - Programming the “bare metal” of the hardware
  - Loading and storing data, arithmetic and logic operations, checking results, and changing control flow
- To get more familiar with IA-32 assembly
  - Generate your own assembly-language code
    - gcc –S –O2 code.c