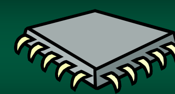
Processors

Part 2

1



Processors

What are they? Besides awesome!

2

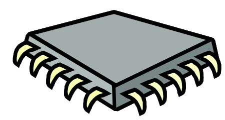
Computer Processors

- The *Central Processing Unit (CPU)* is the most complex part of a computer
- In fact, it is the computer!
- It works far different from a high-level language
- *Thousands* of processors have been developed

3

Some Famous Computer Processors

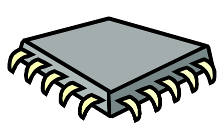
- RCA 1802
- Intel 8086
- Zilog Z80
- MOS 6502
- Motorola 68000
- ARM



4

Computer Processors

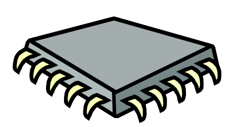
- Each processor functions differently
- Each is designed for a specific purpose – *form follows function*



5

Computer Processors

- But all share some basic properties and building blocks…
- Computer hardware is divided into two "units"
  1. Control Logic Unit
  2. Execution Unit



6

Control Logic Unit (CLU)

- *Control Logic Unit (CLU)* controls the processor
- Determines when instructions can be executed
- Controls internal operations
  - fetch & decode instructions
  - invisible to running programs

7

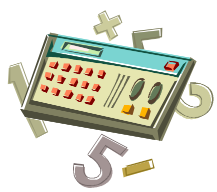
Execution Unit

- *Execution Unit (EU)* contains the hardware that *executes* tasks (your programs)
- Different in many processors
- Modern processors often use multiple execution units to execute instructions in parallel to improve performance

8

Execution Unit – The ALU

- *Arithmetic Logic Unit* is part of the Execution Unit and performs all calculations and comparisons
- Processor often contains special hardware for integer and floating point



9

Registers

Where the work is done

10

Registers

- In high level languages, you put active data into variables
- However, it works quite different on processors
- All computations are performed using *registers*

11

What – exactly – is a register?

- A *register* is a location, on the processor itself, that is used to store temporary data
- Think of it as a special global "variable"
- Some are accessible and usable by a programs, but many are hidden

12

What are registers used for?

- Registers are used to store <u>anything</u> the processor needs to keep to track of
- Designed to be *<u>fast!</u>*
- Examples:
  - the result of calculations
  - status information
  - memory location of the running program
  - and much more…

13

General Purpose Registers

- *General Purpose Registers (GPR)* don't have a specific purpose
- They are designed to be used by programs – however they are needed
- Often, you must use registers to perform calculations

14

Special Registers

- There are a number of registers that are used by the Control Logic Unit and cannot be accessed by your program
- This includes registers that control how memory works, your program execution thread, and much more.

15

Special Registers

- *Instruction Pointer (IP)*
  - also called *the program counter*
  - keeps track of the address of your running program
  - think it as the "line number" in your Java program – the one is being executed
  - it can be changed, but only indirectly *(using control logic – which we will cover later)*

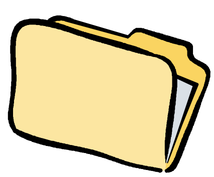
16

Special Registers

- *Status Register*
  - contains Boolean information about the processors current state
  - we will use this later, indirectly
- *Instruction Register (IR)*
  - stores the current instruction (being executed)
  - used internally and invisible to your program

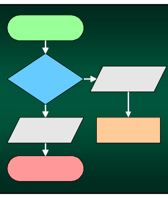
17

Register Files



- All the related registers are grouped into a *register file*
- Different processors access and use their register files in very different ways
- Sometimes registers are implied or hardwired

18



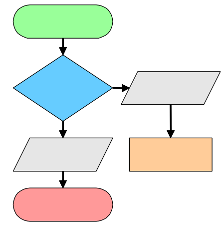
Instructions

It's all just a bunch of bytes

19

Instructions

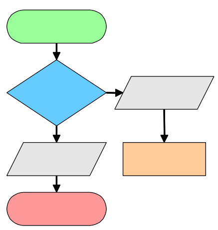
- You are used to writing programs in high level programming languages
- Examples:
  - C#
  - Java
  - Python
  - Visual Basic



20

High-Level Programming

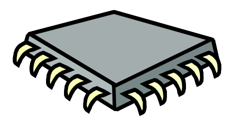
- These are *third-generation languages*
- They are designed to isolate you from architecture of the machine
- This layer of abstraction makes programs "portable" between systems



21

Instructions

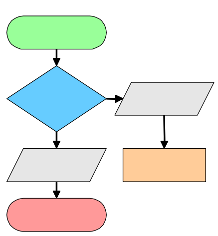
- Processors do not have the constructs you find in high-level languages
- Examples:
  - Blocks
  - If Statements
  - While Statements
  - … etc



22

Instructions

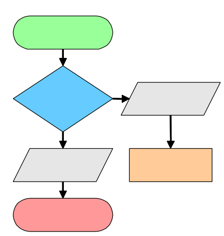
- Processors can only perform a series of simple tasks
- These are called *instructions*
- *Examples:*
  - add two values together
  - copy a value
  - jump to a memory location



23

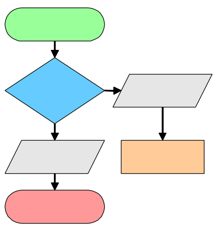
Instructions

- These instructions are used to create all logic needed by a program
- We will cover how to do this during the semester



24

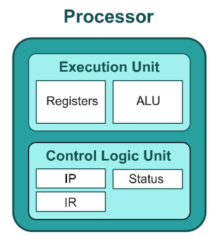
Processor Instruction Set



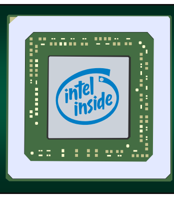
- A processor's *instruction set* defines all the available instructions
- The instructions and their respective formats are very different for each processor

25

Components of a Processor



26



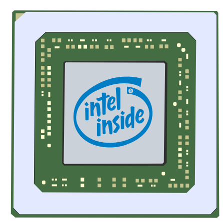
The Intel 8086

It was simple at first…

27

The Intel 8086

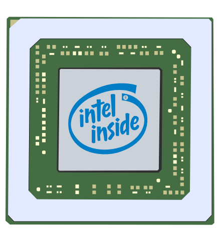
- The Intel x64 is the main processor used by servers, laptops, and desktops
- It has evolved continuously over a 40+ year period



28

The Original x86

- First "x86" was the 8086
- Released in 1978
- Attributes:
  - 16-bit registers
  - 16 registers
  - could access of 1MB of RAM *(in 64KB blocks using a special "segment" register)*



29

Original x86 Registers

- The original x86 contained 16 registers
- 8 can be used by your programs
- The other 8 are used for memory management

30

Original x86 Registers

- The x86 processor has evolved continuously over the last 4 decades
- It first jumped to 32-bit, and then, again, to 64-bit
- This has resulted in many of the registers have strange names

31

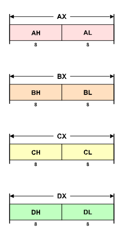
Original x86 Registers

- 8 Registers can be used by your programs
  - Four General Purpose: AX, BX, CX, DX
  - Four pointer index: SI, DI, BP, SP
- The remaining 8 are restricted
  - Six segment: CS, DS, ES, FS, GS, SS
  - One instruction pointer: IP
  - One status register – used in computations

32

Original General-Purpose Registers

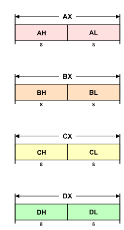
- However, back then (and now too) it is very useful to store 8-bit values
- So, Intel chopped 4 of the registers in half
- These registers have generic names of A, B, C, D



33

Original General-Purpose Registers

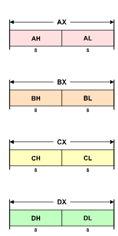
- The first and second byte can be used separately or used together
- Naming convention
  - high byte has the suffix "H"
  - low byte has the suffix "L"
  - for both bytes, the suffix is "X"



34

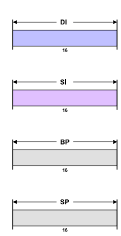
Original General-Purpose Registers

- This essentially doubled the number of registers
- So, there are:
  - four 16-bit registers or
  - eight 8-bit registers
  - *…and any combination you can think off*



35

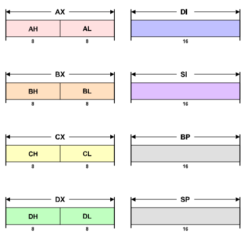
Last the 4 Registers



- The remaining 4 registers were not cut in half
- Used for storing indexes (for arrays) and pointers
- Their purpose
  - DI – destination index
  - SI – source index
  - BP – base pointer
  - SP – stack pointer

36

Original 16-Bit Registers



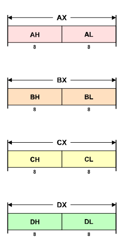
37

Evolution to 32-bit

- When the x86 moved to 32-bit era, Intel expanded the registers to 32-bit
  - the 16-bit ones still exist
  - they have the prefix "e" for extended
- New instructions were added to use them

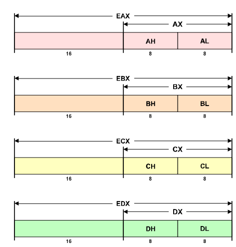
38

Original Registers



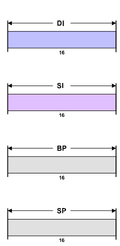
39

Expansion to 32-bit



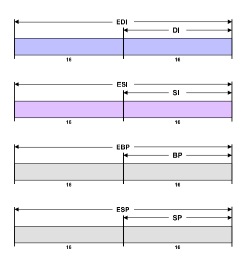
40

Original Registers



41

Expansion to 32-bit



42

Chaotic Move to 64-Bit

Intel vs. AMD

43

The Move to 64-Bit

- By the year 2000, Intel needed to move to 64-bit
- Intel could have, yet again, extended the x86
- However, Intel decided to abandon the x86 in lieu of new design

44

The Itanium

- The Itanium was a radically different from the 8086.
- However, it was completely incompatible with existing x86 programs
- Old programs would have to run through an emulator

45

AMD's Response to the Itanium

- Advanced Micro Devices (AMD), to Intel's chagrin, decided to – *once again* – extend the x86
- It could run old programs without emulation

46

Itanium's Problems

1. The AMD-64 could run existing programs without emulation
2. The Itanium design made it difficult for compilers to make optimized machine code

47

Itanium's Downfall

"The Itanium approach...was supposed to be so terrific until it turned out that the wished-for compilers were basically impossible to write."

– Donald Knuth

48

The Result

- The AMD-64 was a huge commercial *success*
- The Itanium was a huge commercial *failure*
- Intel, dropped the Itanium and started making 64-bit x86 using AMD's design

49

The 64-bit Era

Intel vs. AMD

50

The 64-bit Era

- After the Itanium's disastrous flop – Intel resorted to making AMD-64 compatible processors.
- The classic term "x86" refers to the 32-bit and 16-bit processor family

51

The 64-bit Era

- The term "x64" is used to refer to the AMD's 64-bit extension
- However, the two terms, x86 and x64, are often used interchangeably

52

x64 Registers

- Existing registers were extended by adding 32-bits
- 8 additional registers were added – needed by this era
- 64-bit registers have the prefix *"r"*

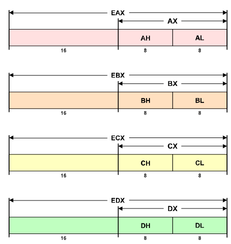
53

x64 Simplified Hardware (best it could)

- It is now possible to get 8-bit values from all registers
- This makes the hardware simpler and more consistent
- Also, many, many archaic, x86 instructions were dropped

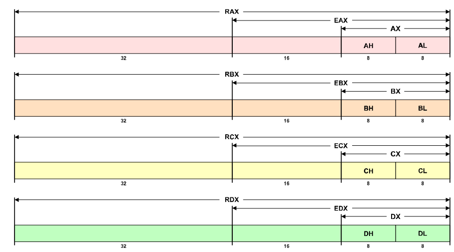
54

Expansion to 64-bit



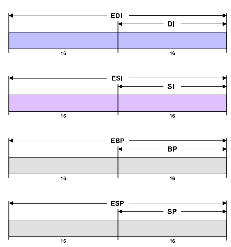
55

Expansion to 64-bit



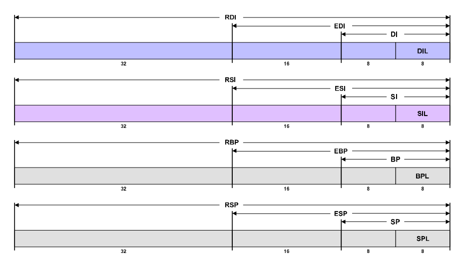
56

Expansion to 64-bit



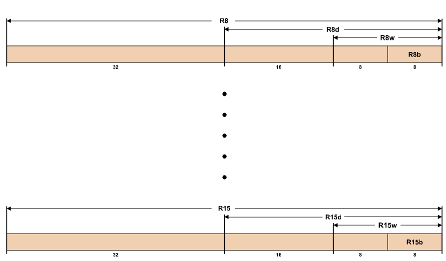
57

Expansion to 64-bit



58

New 64-bit Registers: R8…R15



59

64-Bit Register Table

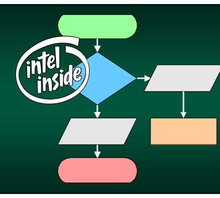
Register	32-bit	16-bit	8-bit High	8-bit Low
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si		sil
rdi	edi	di		dil
rbp	ebp	bp		bpl
rsp	esp	sp		spl

60

64-Bit Register Table

Register	32-bit	16-bit	8-bit High	8-bit Low
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

61



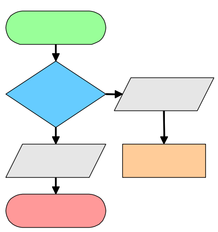
Basic Intel x86 Instructions

Feel the pow-wah of the x86!

62

Basic Intel x86 Instructions

- Each x86 instruction can have up to 2 operands
- Operands in x86 instructions are very versatile
- Each operand can be either a memory address, register or an immediate value



63

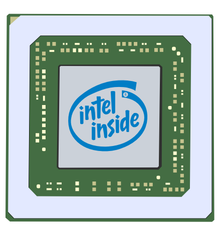
Types of Operands

- Registers
- Address in memory
- Register pointing to a memory address
- Immediate

64

Intel x86 Instruction Limits

- There are some limitations…
- Some instructions must use an immediate
- Some instructions require a *specific* register to perform calculations



65

Intel x86 Instruction Limits

- A register must always be involved
  - processors use registers for all activity
  - both operands cannot access memory at the same time
  - *the processor has to have it at some point!*
- Also, obviously, the receiving field cannot be an immediate value

66

Instruction: Move

- The Intel *Move Instruction* combines transfer, load and store instructions under one name
- … well, that's something the assembler does for us – but, we'll cover that soon
- "Move" is a tad confusing – it copies data

67

Instruction: Move

Immediate, Register, Memory

```
MOV destination, source
```

Register, Memory

68

Example: Move immediate

Source is a immediate constant

```
MOV rax, 42
```

Same as Java `rax = 42;`

Destination is rax

69

Example: Transfer

Source is rax

```
MOV rbx, rax
```

Same as Java `rbx = rax;`

Destination is rbx

70

Example: Load

"total" is memory location

```
MOV rax, total
```

Destination is rax

71

Example: Store

Source is rax

```
MOV counter, rax
```

Memory location named 'Counter'

72

Example: "A" Register

```
# So many options!

mov al, 42      #low byte
mov ah, 13      #high byte
mov ax, 1947    #both bytes
```


73

Instruction: Add & Subtract

- The Add and Subtract instructions take two operands and store the result in the first operand
- This is the same as the += and −= operators used in Visual Basic .NET, C, C++, Java, etc…

74

Instruction: Add

Immediate, Register, Memory

```
ADD target , value
```

Register, Memory

75

Example: Move register to memory

Move memory into rax

```
MOV rax, counter
ADD rax, 2
```

Same as Java
`rax += 2;`

76

Instruction: And & Or

- The Bitwise And & Bitwise Or instructions take two operands and stores the result in the second operand
- This is the same as the ^= and |= operators used in C, C++, Java, etc…

77

Instruction: Logical And

Immediate, Register, Memory

```
AND target , value
```

Register, Memory

78

Example: Logical Or

```
#Convert 5 to ASCII '5'
MOV   rax, 5
OR    rax, 0x30
```

0011 0000

79

Call Instruction

- The *Call Instruction* causes the processor to start running instructions at a specified memory location (a subroutine)
- Subroutines are analogous to the functions you wrote in Java
- Once it completes, execution returns from the subroutine and continues after the call

80

Call Instruction

CALL ***address***

Usually a label – a constant that holds an address

81

Example: Print an integer

```
#Using the CSC35 library

MOV   rdx, 1846
CALL  PrintInteger
```

This name is an address

82