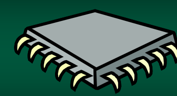
Processors

Part 2

1



Processors

What are they? Besides awesome!

2

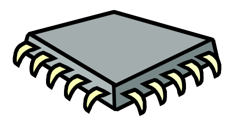
Computer Processors

- 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

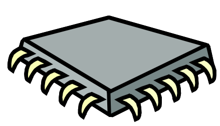
- RCA 1802
- Intel 8086
- Zilog Z80
- MOS 6502
- Motorola 68000
- ARM



4

Computer Processors

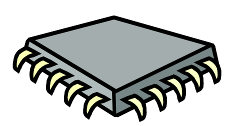
- Each processor functions differently
- Each is designed for a specific purpose – *form follows function*



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

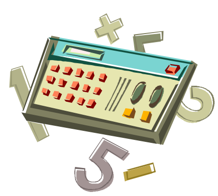
Execution Unit

- *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

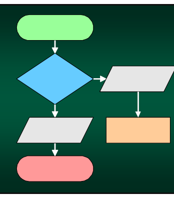
8

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



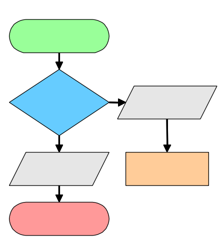
Instructions

It's all just a bunch of bytes

10

Instructions

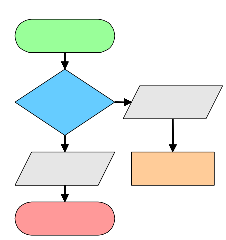
- You are used to writing programs in high level programming languages
- Examples:
  - C#
  - Java
  - Python
  - Visual Basic



11

High-Level Programming

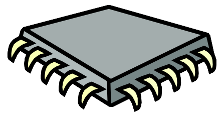
- These are *third-generation languages*
- They are designed to isolate you from architecture of the machine
- This layer of abstraction makes programs "portable" between systems



12

Instructions

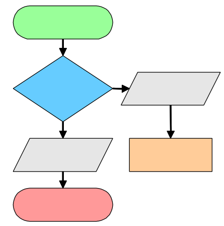
- Processors do not have the constructs you find in high-level languages
- Examples:
  - Blocks
  - If Statements
  - While Statements
  - … etc



13

Instructions

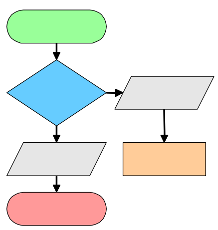
- Processors can only perform a series of simple tasks
- These are called *instructions*
- *Examples:*
  - add two values together
  - copy a value
  - jump to a memory location



14

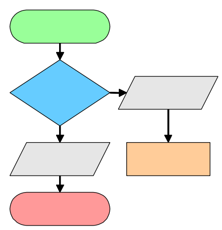
Instructions

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



15

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

16

Registers

Where the work is done

17

Registers

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

18

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

19

What are registers used for?

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

20

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

21

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.

22

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)*

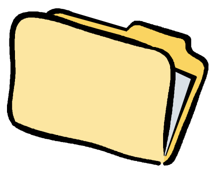
23

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

24

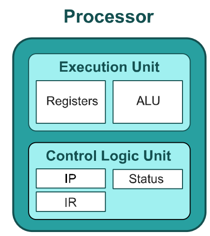
Register Files



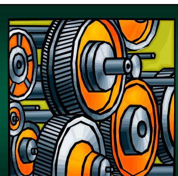
- All the related registers are grouped into a *register file*
- Different processors access and use their register files in very different ways
- Sometimes registers are implied or hardwired

25

Components of a Processor



26



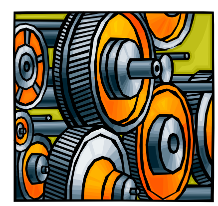
Machine Language

The raw bytes of your program

27

Machine Language

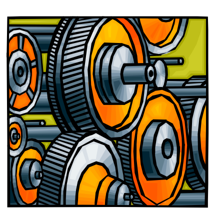
- The instructions, that are *actually* executed on the processor, are just bytes
- In this raw binary form, instructions are stored in *Machine Language (aka Machine Code)*



28

Machine Language

- Each instruction is *encoded* (stored) is in a compact binary form
- Easy for the processor to interpret and execute
- Some instructions may take more bytes than others – not all are equal in complexity



29

Instruction Encoding

- Each instruction must contain everything the processor needs to know to do something
- Think of them as functions in Java: they need a name and arguments to work

30

Instruction Encoding

- For example: if you want it to add 2 things…
- The instruction needs:
  - something to tell the processor to add
  - something to identify the two "things"
  - destination to save the result

31

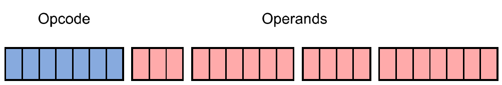
Operation Codes

- Each instruction has a unique *operation code (Opcode)*
- This is a value that specifies the exact operation to be performed by the processor
- Assemblers use friendly names called *mnemonics*

32

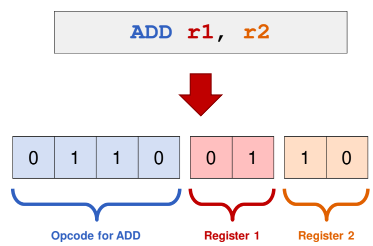
Typical Instruction Format

- The opcode is, typically, followed by various *operands* – what data is to be used
- These can be register codes, addressing data, literal values, etc…

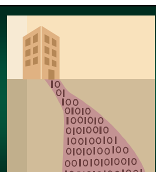


33

Machine Code Example (not x86)



34



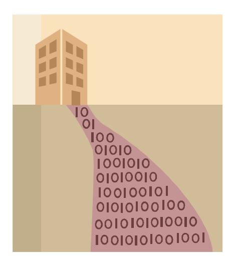
Accessing Data

Filling the registers

35

Accessing Data

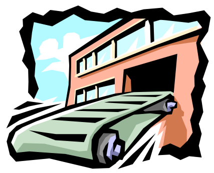
- Processors use registers to hold data being computed
- So, how is data put into the registers to begin with?
- Data can come from two major sources



36

Immediates

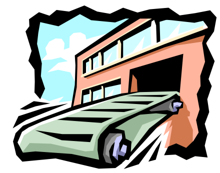
- In programming, it is common to assign a constant to a variable
- As you can imagine, this will also be quite common with instructions



37

Immediates

- When a constant is stored as part of instruction, it is called an *immediate*
- Once the instruction is loaded by the processor, it is "immediately" available from the IR – hence, the name



38

Load Immediate

- A *Load Immediate* instruction, stores a constant into a register
- The instruction must store the destination register and the immediate value

Opcode	Register	Immediate
Load Immediate	Destination	Value

39

Load Immediate Example (not x86)

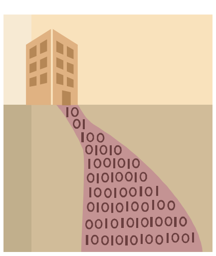
```
LDI r2, 123
```

1	1	0	0	1	0	0	1	1	1	1	0	1	1

$\underbrace{1100}_{\text{Opcode for LDI}}\ \underbrace{10}_{\text{Register 2}}\ \underbrace{01111011}_{\text{123 in binary}}$

40

Copying Data



- Processors have a number of instructions that can copy data
- Each has a unique name –*not surprising since each does something different*

41

Transfer

- A *Transfer* instruction, copies the contents of one instruction into another
- The instruction must store both the destination and source register

Opcode	Register	Register
Transfer	Destination	Source

42

Transfer Example (not x86)

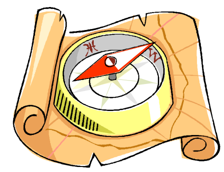
```
TRA r1, r2
```

1	1	1	0	0	1	1	0
Opcode for Transfer				Register 1		Register 2	

43

Effective Addresses

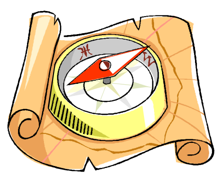
- Processors also have the ability to access memory
- Since memory is a massive array, the processor needs to know the address



44

Effective Addresses

- The *effective address* is used to access memory
- Often it is created by combining multiple values
- … but we will cover that later in the semester



45

Load

- A *Load* instruction, reads data from memory (at a specified address)
- This data is then stored into the destination register

Opcode	Register	Immediate
Load	Destination	Source Address

46

Store

- A *Store* instruction, writes data from a register into the specified address
- Note: the structure is identical to Load

Opcode	Register	Immediate
Store	Source	Destination Address

47

Load Example (not x86)

```
LDR r3, [123]
```

1	1	0	0	1	1	0	1	1	1	1	0	1	1
Opcode for LOAD				Register 3		123 in binary							

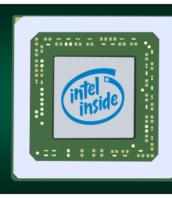
48

Intro to the Intel x64

Part 3

49

The Intel x64

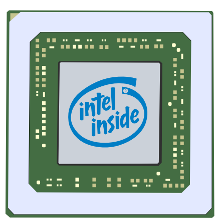


It was simple at first…

50

The Intel x64

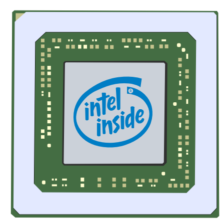
- The Intel x64 is the main processor used by servers, laptops, and desktops
- It has evolved continuously over a 40+ year period



51

The Original x86

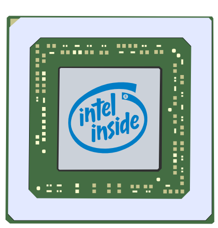
- First "x86" was the 8086
- Released in 1978
- Attributes:
  - 16-bit registers
  - 16 registers
  - could access of 1MB of RAM *(in 64KB blocks using a special "segment" register)*



52

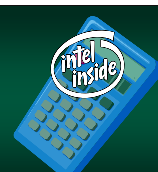
What to call the processor

- The classic term "x86" refers to the 32-bit and 16-bit processor family
- With move to 64-bit, the term "x64" is used to differentiate the newest design from the previous



53

Original x86 Registers



It was simple at first…

54

Original x86 Registers

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

55

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
- The result is many of the registers have strange names

56

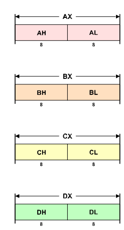
Original x86 Registers

- 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

57

Original General-Purpose Registers

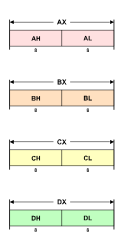
- However, back then (and now too) it is very useful to store 8-bit values
- So, Intel chopped 4 of the registers in half
- These registers have generic names of A, B, C, D



58

Original General-Purpose Registers

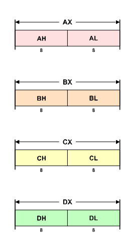
- The first and second byte can be used separately or used together
- Naming convention
  - high byte has the suffix "H"
  - low byte has the suffix "L"
  - for both bytes, the suffix is "X"



59

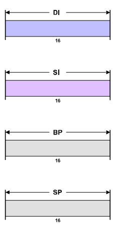
Original General-Purpose Registers

- 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*



60

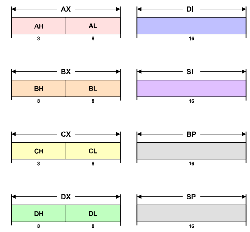
Last the 4 Registers



- 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

61

Original 16-Bit Registers



62

Evolution to 64-Bit Registers

This is going to hurt…

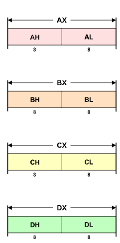
63

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

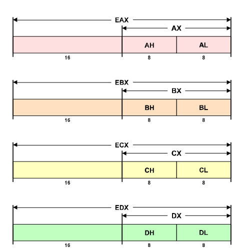
64

Original Registers



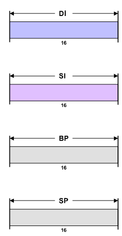
65

Expansion to 32-bit



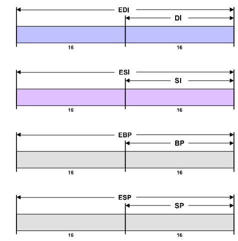
66

Original Registers



67

Expansion to 32-bit



68

Evolution to 64-bit

- At this point, Intel had decided to abandon the x86 in lieu of their new Itanium Processor
- The Itanium was a radically different design and was completely incompatible
- Advanced Micro Devices (AMD), to Intel's chagrin, decided to – *once again* – extend the x86

69

Evolution to 64-bit

- Registers were extended again
  - 64-bit registers have the prefix *"r"*
  - 8 additional registers were added
  - also, it is now possible to get 8-bit values from all registers (hardware is more consistent!)
- Some old, archaic, features were dropped

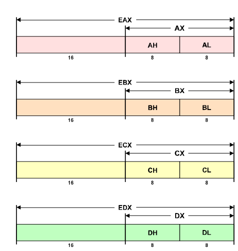
70

Evolution to 64-bit

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

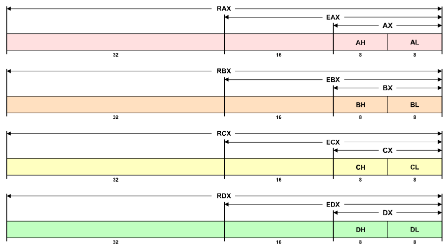
71

Expansion to 64-bit



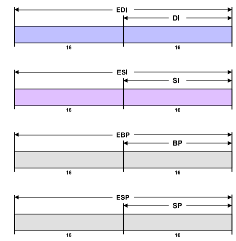
72

Expansion to 64-bit



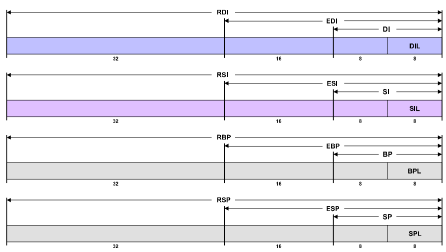
73

Expansion to 64-bit



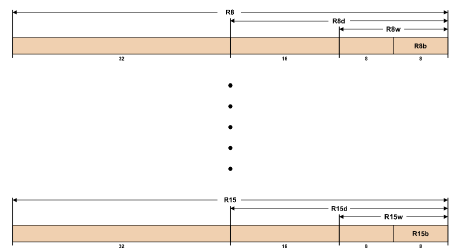
74

Expansion to 64-bit



75

New 64-bit Registers: R8…R15



76

64-Bit Register Table

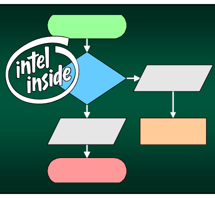
Register	32-bit	16-bit	8-bit High	8-bit Low
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si		sil
rdi	edi	di		dil
rbp	ebp	bp		bpl
rsp	esp	sp		spl

77

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

78



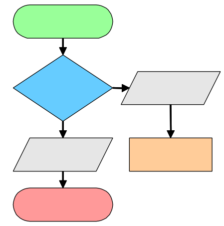
Basic Intel x86 Instructions

Feel the pow-wah of the x86!

79

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



80

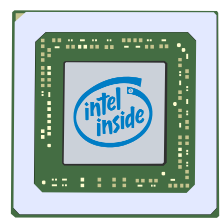
Types of Operands

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

81

Intel x86 Instruction Limits

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



82

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

83

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

84

Instruction: Move

Immediate, Register, Memory

MOV *destination, source*

Register, Memory

85

Example: Move immediate

Source is a immediate constant

```
MOV rax, 42
```

Same as Java
`rax = 42;`

Destination is rax

86

Example: Transfer

Source is rax

```
MOV rbx, rax
```

Same as Java
`rbx = rax;`

Destination is rbx

87

Example: Load

"total" is memory location

```
MOV rax, total
```

Destination is rax

88

Example: Store

Source is rax

```
MOV counter, rax
```

Memory location named 'Counter'

89

Example: "A" Register

```
# So many options!

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


90

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…

91

Instruction: Add

```
ADD target , value
```

Immediate, Register, Memory

Register, Memory

92

Example: Move register to memory

Move memory into rax

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

Same as Java `rax += 2;`

93

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…

94

Instruction: Logical And

```
AND target , value
```

Immediate, Register, Memory

Register, Memory

95

Example: Logical Or

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

0011 0000

96

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

97

Call Instruction

```
CALL address
```

Usually a label – a constant that holds an address

98

Call Instruction

- The *Call instruction* doesn't change any of the general-purpose registers
- It only stores an address – where execution will continue

Opcode	Immediate
Call	Subroutine Address

99

Example: Print an integer

```
#Using the CSC35 library

MOV   rcx, 1846
CALL  PrintInt
```

This name is an address

100