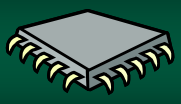
Part 2

Processors



Computer Processors

The Brains of the Box

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

Components of a Processor

- *Execution Unit (EU)*
  - performs calculations & logic
  - registers hold data
- *Control Logic Unit (CLU)*
  - reads and decodes instructions
  - talks to other components

Execution Unit

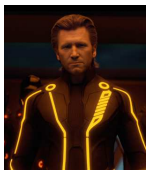
- Contains the hardware that *executes* tasks (your programs)
- Different in many processors
- Modern processors often use multiple execution units to execute instructions in parallel to improve performance

Execution Unit – The ALU

- The *Arithmetic Logic Unit* performs all calculations and comparisons
- Processor often contains special hardware for integer and floating point

Control Logic Unit (CLU)

- Controls the processor
- Determines when instructions can be executed
- Controls internal operations
  - fetch and execute each instruction
  - and store result of each instruction



CLU Over Time

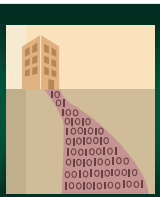
- In early processors…
  - CLU was a very small fraction of the hardware
  - EU and the registers took most of the space
- New processors…
  - complex control unit one of the more difficult parts of a processor to design
  - has increased in its percentage of the processor hardware

Computer Processors

- Over time, thousands of processors were developed
- Examples:
  - Intel x86
  - IBM PowerPC
  - MOS 6502
  - ARM

High-Level Programming

- You are used to writing in high level programming languages such as C#, Java, Visual Basic, etc…
- These are *third-generation languages* – and are designed to isolate you from architecture of the machine

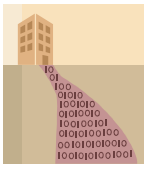


Registers

Where the work is done

Registers

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



What are registers used for?

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

What – exactly – is a register?

- A register is a memory location located on the processor itself
- Think of it as a special "variable"
- Designed to be fast
- Some are accessible and usable by a programs, but many are hidden.

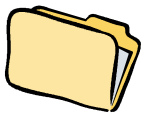
General Purpose Registers

- *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

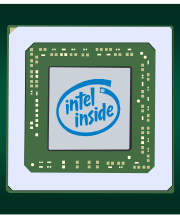
Some Special Registers

- Program counter
  - controls the current memory location of the running program
  - privileged – only the processor and OS can change it
- Stack pointer
  - tracks the top of the system stack
  - you can modify this … *with care…*

Register Files



- All the related registers are grouped into a *register file*
- Different processors access and use their register files in very different ways
- Some processors support multiple files

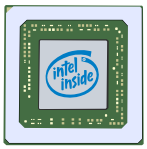


Original x86 Registers

It was simple at first…

Original x86 Registers

- First "x86" was the Intel 8086 released in 1978
- Attributes:
  - 16-bit processor (registers were 16-bit)
  - 16 registers
  - can access of 1MB of RAM

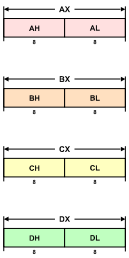


Intel x86 Registers

- 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

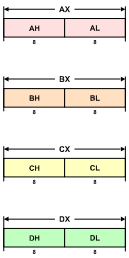
Original General Purpose Registers

- 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



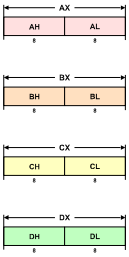
Original General Purpose Registers

- 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"

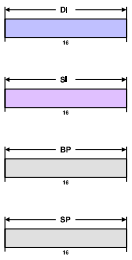


Original General Purpose Registers

- 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)*

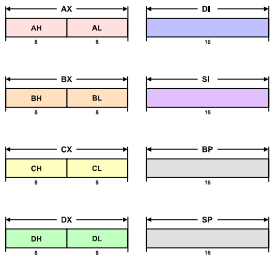


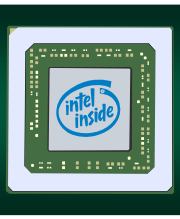
Last the 4 Registers



- 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

Original 16-Bit Registers



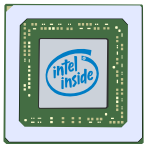


Evolution to 64 Bit Registers

This is going to hurt…

Original x86 Registers

- The x86 processor has evolved continuously over the last (nearly) 4 decades
- It jumped to 32-bit, and then finally 64-bit
- The result is many of the registers have strange names



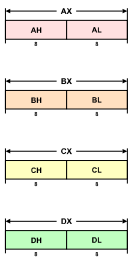
Evolution to 32-bit

- When the x86 moved into the 32-bit era, Intel expanded the registers to 32-bit
  - the 16-bit ones still exist
  - … but also have a 32-bit version
  - they have the prefix *"e" for extended*
- New instructions were added (to use them)

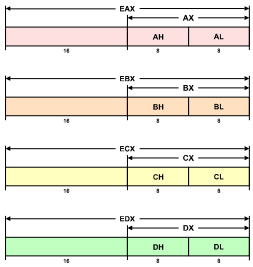
Evolution to 64-bit

- Once again, the processor evolved – now to 64-bit
- The registers were extended again
  - the 64-bit have the prefix *"r" for register*
  - 8 additional registers were added
  - also, it is now possible to get 8-bit values from all registers (hardware is more consistent!)

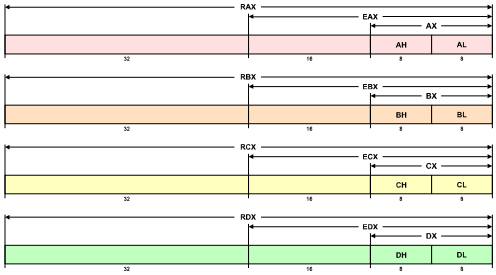
Original Registers



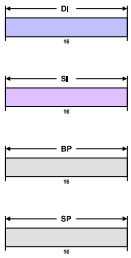
Expansion to 32-bit



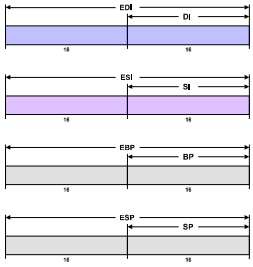
Expansion to 64-bit



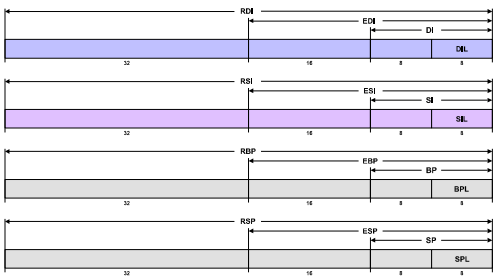
Original Registers



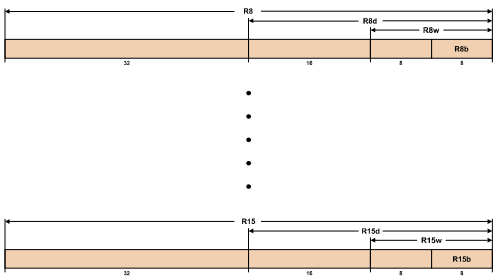
Expansion to 32-bit



Expansion to 64-bit



New 64-bit Registers

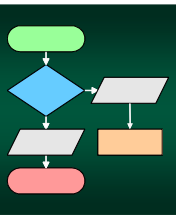


64-Bit Register Table

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		bsl

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

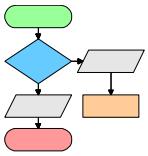


Instructions

Your programs are simple

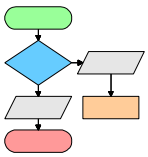
Instructions

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



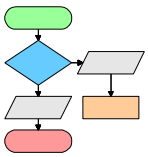
Instructions

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

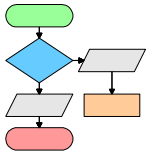


Instructions

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



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

What – exactly – is an instruction?

- An instruction is a series of bytes that contain everything the processor needs to know to do something
- An instruction must specify:
  - *operation* – what to do
  - *operands* – what data is to be used

Operation Codes

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

Types of Operations (Opcodes)

- Data Transfer
- Program Flow Control
- Arithmetic and Logic operations
- Input and Output Instructions

Operations: Data Transfer

- One of the most common tasks is moving data to and from registers
- Classified into three categories:
  - *loading* a register with data in memory
  - *storing* data in a register into memory
  - *transferring* data between registers

Operations: Control Flow

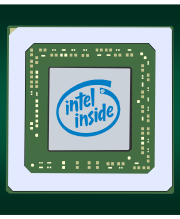
- Processors do not support blocks, If Statements, etc…
- Instead, you must jump around code you don't want to execute
- This is the same of idea of GoTo Statements

Operations: Arithmetic and Logic

- Many operations are used to modify data such as arithmetic, comparisons, and shifting
- Comparison is in this category
  - when two operands are compared – often one is subtracted from the other
  - result sets Boolean flags – *more on this later!*

Operations: Input and Output

- There are also a instructions that are designed to talk to ports, hard drives and other components
- However, in modern systems, these are privileged and only usable by the operating system
- You will use interrupts to tell the operating system to input/output data

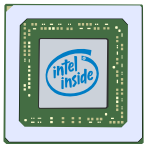


Basic Intel x86 Instructions

Feel the pow-wah of the x86!

Basic Intel x86 Instructions

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

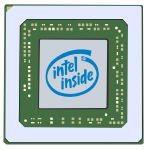


Types of Operands

- Registers
- Memory address
- Register pointing to memory
- A constant stored with the instruction – this is called an *immediate*

Intel x86 Instruction Limits

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



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

Instruction: Move

- The x86 Move Instruction combines load, store, and register transfer logic
- It is one of the most common instructions used in programs (true of all processors)
- Remember how often you use the assignment statement in C / Java?

Instruction: Move

Immediate, Register, Memory

```
MOV source , destination
```

Register, Memory

Example: Move immediate

Source is a immediate constant

```
MOV $42, %rax
```

Destination is rax

Example: "A" Register

```
# So many options!

mov $42, %al     #low byte
mov $13, %ah     #high byte
mov $47, %ax     #both bytes
```


Example: Move register to register

Source is rax

```
MOV %rax, %rbx
```

Destination is rbx

Example: Move register to memory

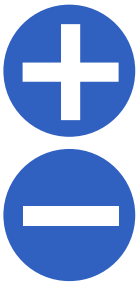
Source is rax

```
MOV %rax, counter
```

Memory location named 'Counter'

Instruction: Add & Subtract

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



Instruction: Add

Immediate, Register, Memory

```
ADD value , target
```

Register, Memory

Example: Move register to memory

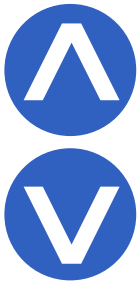
Move memory into rax

```
MOV counter, %rax
ADD $2, %rax
```

rax += 2

Instruction: And & Or

- The Logical And and Logical 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…



Instruction: Logical And

```
AND value , target
```

Example: Logical Or

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

0011 0000

Instruction: Call

- The Call Instruction transfers control to a memory location (a subroutine)
- Subroutines are analogous to the functions you wrote in Java
- Once it completes, execution continues after the call

Instruction: Call

CALL ***address***

Usually a label – an constant that holds an address

Example: Print an integer

```
#This is using the CSC35 library

MOV   $42, %rax
CALL  PrintInt
```

This name is an address