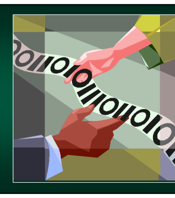
Data & Memory

Part 1

1



Binary Numbers

Bit of This and a Bit of That

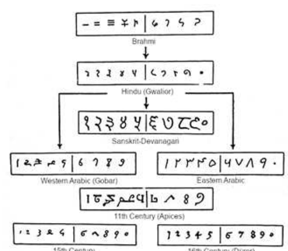
2

What is a Number?

- *Hindu-Arabic Number System*
  - positional grouping system
  - each position represents an increasing power of 10
  - used throughout the World
- Binary numbers
  - based on the same system
  - use powers of **2** rather than 10

3

Evolution of a Genius System



4

Base 10 Number

The number 1783 is ...

10^4	10^3	10^2	10^1	10^0
10000	1000	100	10	1
0	**1**	**7**	**8**	**3**

1000 + 700 + 80 + 3 = 1783

5

Binary Number Example

The number 0100 1010 is ...

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	**1**	**0**	**0**	**1**	**0**	**1**	**0**

64 + 8 + 2 = 74

6

Binary Number Example

The number 1101 1011 is ...

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	**1**	**0**	**1**	**1**	**0**	**1**	**1**

128 + 64 + 16 + 8 + 2 + 1 = 219

7

Hexadecimal Numbers

- Writing out long binary numbers is cumbersome and error prone
- As a result, computer scientists often write computer numbers in hexadecimal
- Hexadecimal is base-16
  - we only have 0 ... 9 to represent digits
  - So, hex uses **A ... F** to represent **10 ... 15**

8

Hexadecimal Numbers

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111

Hex	Decimal	Binary
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

9

Hex Example

The number 7AC is ...

16^4	16^3	16^2	16^1	16^0
65536	4096	256	16	1
0	**0**	**7**	**A**	**C**

$(7\times256) + (10\times16) + (12\times1) = 1964$

10

Converting Binary to Hex = Easy

- Since $16^1 = 2^4$, a single hex character can represent a total of 4 bits
- Convert every 4-bits to a single hexadecimal digit

A				**7**			
1	0	1	0	0	1	1	1

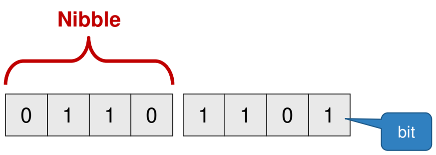
11

Bits and Bytes

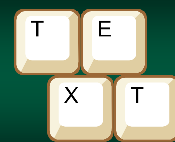
- Everything in a *modern* computer is stored using combination of ones and zeros
- *Bit* is one binary digit
  - either 1 or 0
  - shorthand for a bit is b
- *Byte* is a group of 8 bits
  - e.g. `1101 0100`
  - shorthand for a byte is B

12

The Byte



13



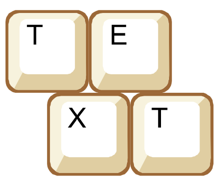
Text in Programming Languages

Press Any Key to Continue

14

How Text Is Stored

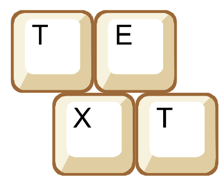
- Computer often store and transmit textual data
- Examples:
  - punctuation
  - numerals 0 – 9
  - letter
- Each of these symbols is called a *character*



15

Characters

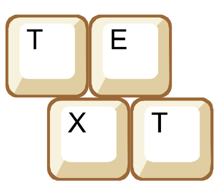
- Processors rarely know what a "character" is, and instead store each as an integer
- In this case, each character is given a unique value
- For instance
  - "A", could have the value of 1
  - "B" is 2
  - "C" is 3, etc…



16

Characters

- Characters and their matching values are a *character set*
- There have been many characters sets developed over time



17

Character Sets

- ASCII
  - 7 bits – 128 characters
  - uses a full byte, one bit is not used
  - created in the 1967
- EBCDIC
  - Alternative system used by old IBM systems
  - Not used much anymore

18

ASCII Chart

Control characters

	0	1	2	3	4		6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

19

ASCII Codes

- Each character has a unique value
- The following is how "OMG" is stored in ASCII

	Decimal	Hex	Binary
O	79	4F	0100 1111
M	77	4D	0100 1101
G	71	47	0100 0111

20

ASCII Codes

- ASCII is laid out very logically
- Alphabetic characters (uppercase and lowercase) are 32 "code points" apart

	Decimal	Hex	Binary
A	65	41	01000001
a	97	61	01100001

21

ASCII Codes

- $32^1 = 2^5$
- 1-bit difference between upper and lowercase letters
- Printers can easily convert between the two

	Decimal	Hex	Binary
A	65	41	01000001
a	97	61	01100001

22

ASCII: Number Characters

- ASCII code for 0 is 30h
- Notice that the actual value of a number character is stored in the lower nibble
- So, the characters 0 to 9 can be easily converted to their binary values

0	0011 0000
1	0011 0001
2	0011 0010
3	0011 0011
4	0011 0100
5	0011 0101
6	0011 0110
7	0011 0111
8	0011 1000
9	0011 1001

23

ASCII: Number Characters

- Character → Binary
  - clear the upper nibble
  - Bitwise And: 0000 1111
- Binary → Character
  - set the upper nibble to 0011
  - Bitwise Or: 0011 0000

0	0011 0000
1	0011 0001
2	0011 0010
3	0011 0011
4	0011 0100
5	0011 0101
6	0011 0110
7	0011 0111
8	0011 1000
9	0011 1001

24

Evolution of ASCII

From ASCII to Unicode

25

Evolution of ASCII

- The ASCII Character Set has gone through a few minor revisions this it was first created
- But, the 1967 standard is still used today

26

Evolution of ASCII

- Back in the 1960's – there were no computer monitors
- Nor did you enter programs using a keyboard
- Instead…
  - programs were inputted using punched cards or tape
  - all output was printed

27

Evolution of ASCII

- The initial 1963 version didn't include lowercase letters
- As shocking as it sounds, there was a time where lowercase letters were considered obsolete
- All-caps was called "modern"

28

Original 1963 ASCII Proposal

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOM	EOA	EOM	EOT	WRU	RU	BEL	FE0	HT	LF	VT	FF	CR	SO	SI
1	DC0	DC1	DC2	DC3	DC4	ERR	SYN	LEM	S0	S1	S2	S3	S4	S5	S6	S7
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
6																
7													ACK		ESC	DEL

29

1967 Revision Added Lowercase

Changed

New

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

30

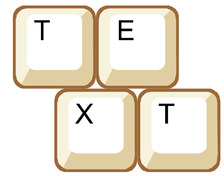
Control Characters

- Many of the control characters were designed for transferring data (SOH, STX, FS, GS, etc…)
- Other control characters we designed for printing

31

Evolution of ASCII

- Also note that very useful characters like "→" were ultimately removed from the 1963 design
- And a new character, Backspace (BS), was added
- Why?



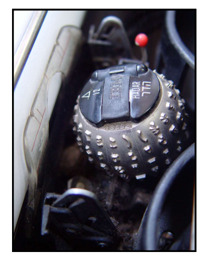
32

Backspace Control Character

- Printers, at the time, were basically classic typewriters
- The backspace character (BS) was used to print 2+ symbols on top of each other
- This would essentially create a new character on the paper

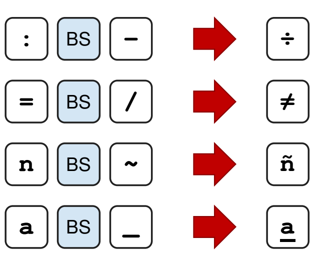
33

IBM Ball-Printers



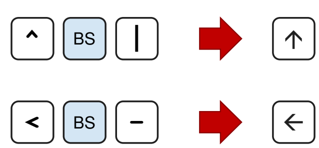
34

Power of the Backspace



35

Recreating the Removed Arrows



36

DEL Control Character

- You might have noticed an odd control character located at 7F
- This is the "Deleted" control character and was used with punched cards and tape

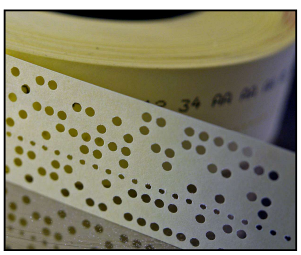
37

ASCII Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

38

Punched Tape



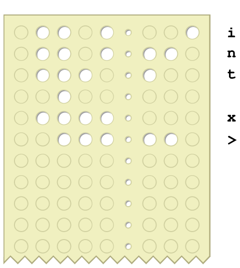
39

Bill Gates – Original Altair BASIC

Intel 8080 machine code

40

Example Punched Tape: `int x;`

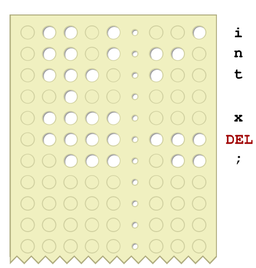


Student punched the wrong hole! They meant `;`

Do they have to punch an entire new tape?

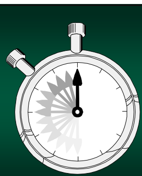
41

Example Punched Tape: `int x;`



Just punch all remaining holes. It becomes DEL and is ignored by the card or tape reader.

42



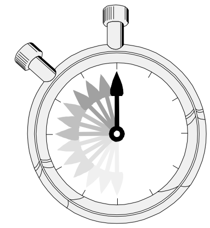
Times have changed

A New Character Set Enters the Scene

43

Times have changed…

- Computers have changed quite a bit since the 1960's
- As a result, most of these clever control characters are no longer needed
- Backspace, DEL, and numerous others are obsolete



44

Only Control Characters Still Used

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

45

Unicode Character Set

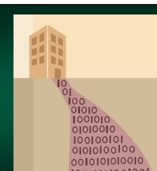
- ASCII is only good for the United States
  - Other languages need additional characters
  - Multiple competing character sets were created
- Unicode was created to support every spoken language
- Developed in Mountain View, California

46

Unicode Character Set

- Originally used 16 bits
  - that's over 65,000 characters!
  - includes every character used in the World
- Expanded to 21 bits
  - 2 million characters!
  - now supports every character ever created
  - … and emojis
- Unicode can be stored in different formats

47



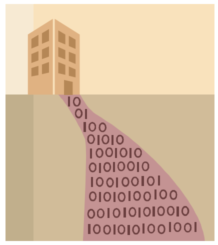
Computer Memory

Its… um…. I forgot….

48

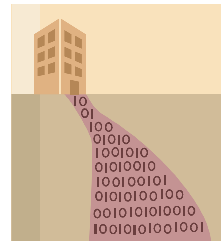
Computer Memory

- Programs access and manipulate memory far more than you realize
- So, understanding it…
  - is vital to becoming a great assembly programmer
  - and understanding computer architecture



What is Memory?

- Memory is essentially an enormous array
- It is also, sometimes, referred to as *storage*
- It stores both running programs and their related data



Memory Addresses

- Memory is divided into a storage locations that can hold 1 byte (8 bits)
- Each an be accessed using an *address*
  - unique value that refers to that specific byte
  - used to locate the exact byte the processor wants

Memory	
0	01000100
1	01000011
2	01101111
3	01101111
4	01101011

What is Memory?

- Each address is conceptually the same as an "index" in arrays
- … and you will write access memory as would an array

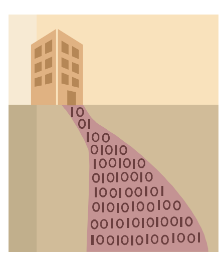
Memory	
0	01000100
1	01000011
2	01101111
3	01101111
4	01101011

Memory is a Hardware Array

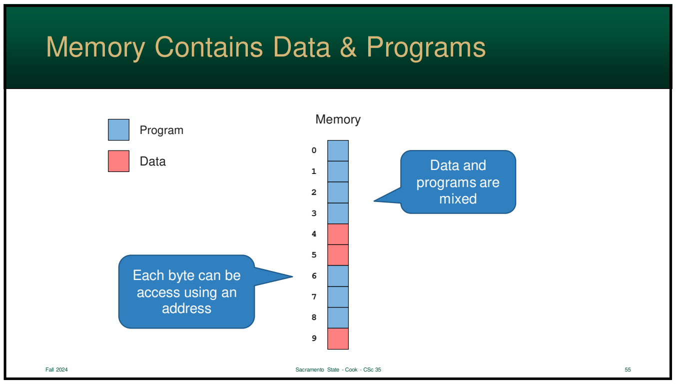
Memory

00000000	
00000001	
00000002	
00000003	
⋮	
FFFFFFFF	

Memory Contains Data & Programs



- Data and programs are just binary numbers (stored in a series of bytes)
- …and are stored together
- Appreciating this is vital to understanding computer architecture

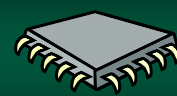

Memory Contains Data & Programs
Program
Data
Memory
0
1
2
3
4
5
6
7
8
9
Data and programs are mixed
Each byte can be access using an address
Fall 2024
Sacramento State - Cook - CSc 35
55

55

Processors

Part 2

1



Processors

What are they? Besides awesome!

2

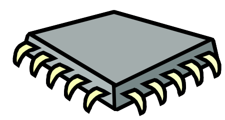
Computer Processors

- 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

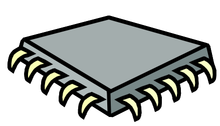
- RCA 1802
- Intel 8086
- Zilog Z80
- MOS 6502
- Motorola 68000
- ARM



4

Computer Processors

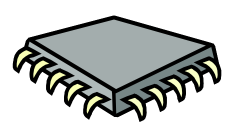
- Each processor functions differently
- Each is designed for a specific purpose – *form follows function*



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

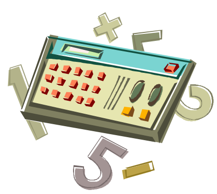
Execution Unit

- *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 anything the processor needs to keep track of
- Designed to be *fast!*
- 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)*

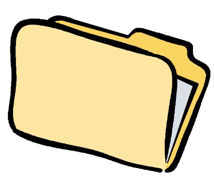
16

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

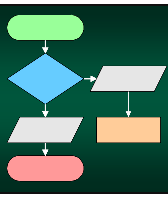
17

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



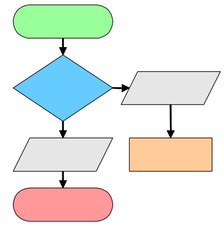
Instructions

It's all just a bunch of bytes

19

Instructions

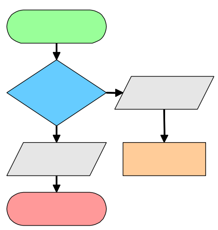
- You are used to writing programs in high level programming languages
- Examples:
  - C#
  - Java
  - Python
  - Visual Basic



20

High-Level Programming

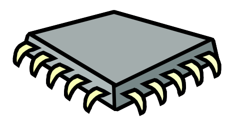
- These are *third-generation languages*
- They are designed to <u>isolate</u> you from architecture of the machine
- This layer of abstraction makes programs "portable" between systems



21

Instructions

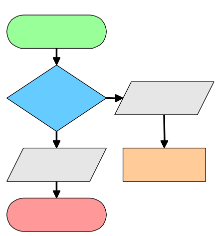
- Processors <u>do not</u> have the constructs you find in high-level languages
- Examples:
  - Blocks
  - If Statements
  - While Statements
  - … etc



22

Instructions

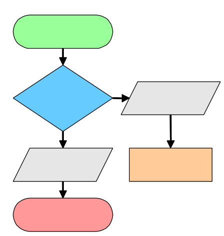
- Processors can only perform a series of simple tasks
- These are called *instructions*
- *Examples:*
  - add two values together
  - copy a value
  - jump to a memory location



23

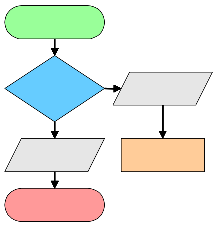
Instructions

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



24

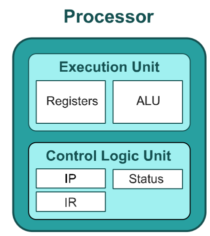
Processor Instruction Set



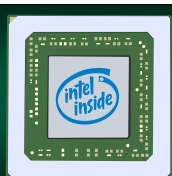
- A processor's *instruction set* defines all the available instructions
- The instructions and their respective formats are very different for each processor

25

Components of a Processor



26



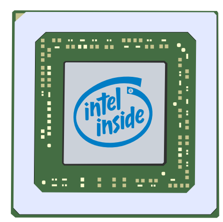
The Intel 8086

It was simple at first…

27

The Intel 8086

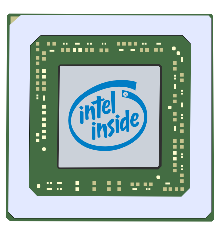
- The Intel x64 is the main processor used by servers, laptops, and desktops
- It has evolved continuously over a 40+ year period



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

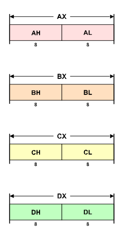
Original x86 Registers

- 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

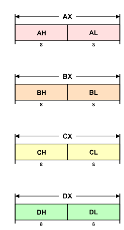
- However, back then (and now too) it is very useful to store 8-bit values
- So, Intel chopped 4 of the registers in half
- These registers have generic names of A, B, C, D



33

Original General-Purpose Registers

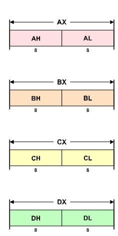
- The first and second byte can be used separately or used together
- Naming convention
  - high byte has the suffix "H"
  - low byte has the suffix "L"
  - for both bytes, the suffix is "X"



34

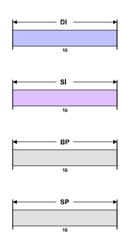
Original General-Purpose Registers

- 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

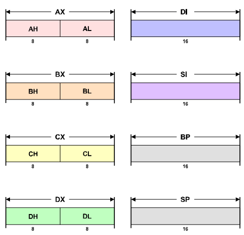
Last the 4 Registers



- 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



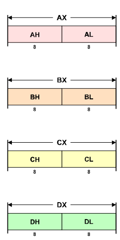
37

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

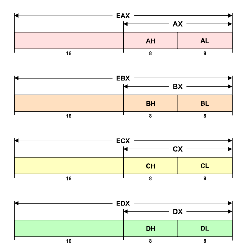
38

Original Registers



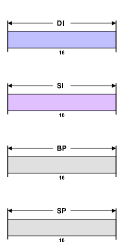
39

Expansion to 32-bit



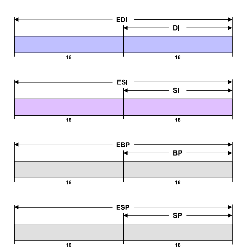
40

Original Registers

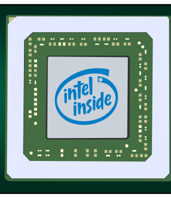


41

Expansion to 32-bit



42



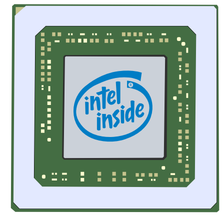
Chaotic Move to 64-Bit

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

The 64-bit Era

Intel vs. AMD

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

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

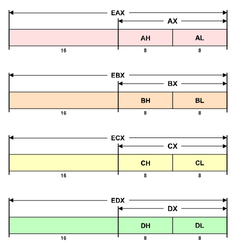
x64 Registers

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

x64 Simplified Hardware (best it could)

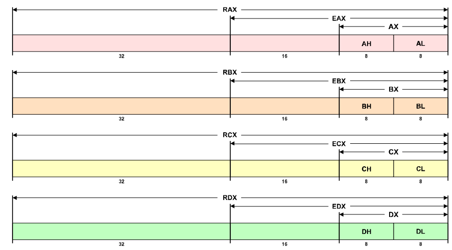
- It is now possible to get 8-bit values from all registers
- This makes the hardware simpler and more consistent
- Also, many, many archaic, x86 instructions were dropped

Expansion to 64-bit



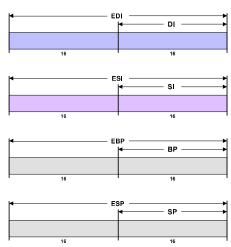
55

Expansion to 64-bit



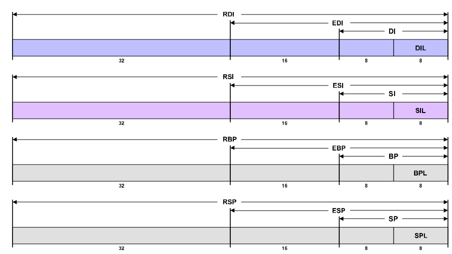
56

Expansion to 64-bit



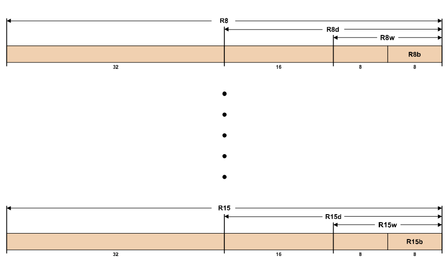
57

Expansion to 64-bit



58

New 64-bit Registers: R8…R15



59

64-Bit Register Table

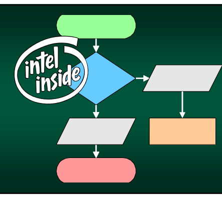
Register	32-bit	16-bit	8-bit High	8-bit Low
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si		sil
rdi	edi	di		dil
rbp	ebp	bp		bpl
rsp	esp	sp		spl

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



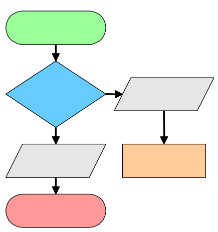
Basic Intel x86 Instructions

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

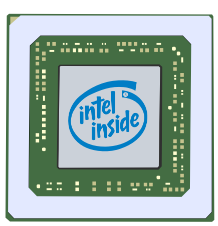
Types of Operands

- 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

```
ADD target , value
```

Immediate, Register, Memory

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

```
AND target , value
```

Immediate, Register, Memory

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
```

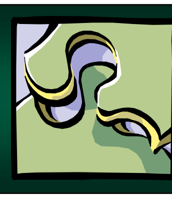
This name is an address

82

Programs

Part 3

1



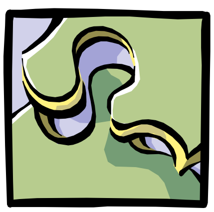
Compilers, Assemblers & Linkers

Programs, Coding, and Nerds… oh my!

2

Compilers & Assemblers

- When you hit "compile" or "run" (e.g. in your Java IDE), many actions take place *"behind the scenes"*
- You are usually only aware of the work that the parser does



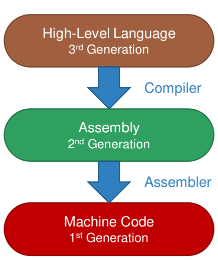
3

Development Process

1. Write program in high-level language
2. Compile program into assembly
3. Assemble program into objects
4. Link multiple objects programs into one executable
5. Load executable into memory
6. Execute it

4

From Abstract to Machine



5

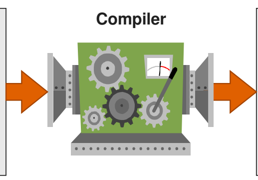
Compiler

- Convert programs from high-level languages (such as C or C++) into assembly language
- Some create machine-code directly…
- *Interpreters*, however…
  - never compile code
  - Instead, they run parts of their own program

6

Compilers: 3rd → 2nd Generation

```
x = 1846;
x += 42;
n = 3;
a[n] = x;
```




```
mov r8, 1846
add r8, 42
mov r9, 3
mov [a+r9*8], r8
```


7

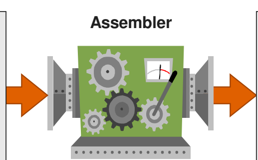
Assembler

- Converts assembly into the binary representation used by the processor
- Often the result is an *object file*
  - usually not executable - yet
  - contains computer instructions and information on how to "link" into other executable units
  - file may include: relocation data, unresolved labels, debugging data

8

Assembler: 2nd → 1st Generation

```
mov r8, 1846
add r8, 42
mov r9, 3
mov [a+r9*8], r8
```

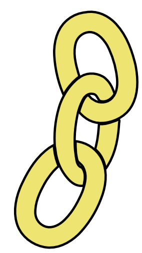



```
01000100
01100101
01110110
01101001
01101110
```


9

Linkers

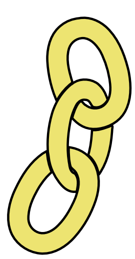
- Often, parts of a program are created separately
- Happens *more often than you think* – almost always
- Different parts of a program are called *objects*
- A *linker* joins them into a single file



10

What a Linker Does

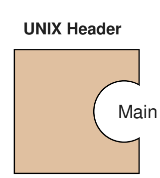
- Connects labels (identifiers) - used in one object - to the object that defines it
- So, one object can call another object
- A linker will show an error if there are label conflicts or missing labels



11

Linking your program

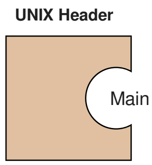
- UNIX header is defined by crt1.o and crti.o
- They are supplied behind the scenes, *so you don't need to worry about them*



12

Linking your program

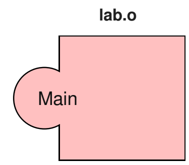
- It references a subroutine called Main
- But… it is not defined in the header
- It is used to start your program (main in Java)



13

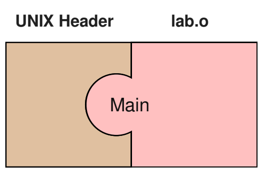
Linking your program

- Your program supplies this subroutine
- The linker connects the two, so the header calls your subroutine



14

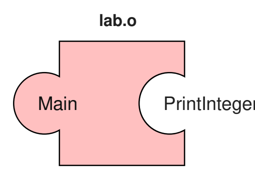
Linking to the UNIX Header



15

You will use my library

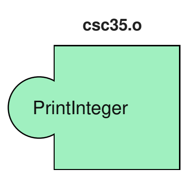
- To make labs easier, you will use my library
- Your program will reference its subroutines



16

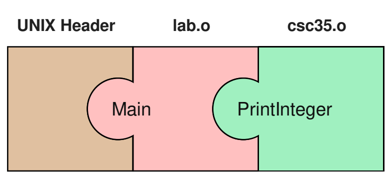
You will use my library

- Once the object file "csc35.o" is linked, the program is complete

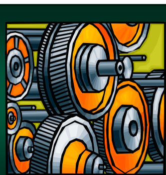


17

You will use my library



18



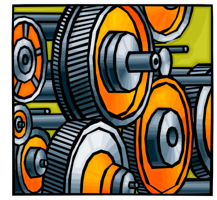
Assembly Basics

The beautiful language of the computer

19

Assembly Language

- *Assembly* allows you to write machine language programs using easy-to-read text
- Assembly programs is based on a specific processor architecture
- So, it won't "port"



20

Assembly Benefits

1. Consistent way of writing instructions
2. Automatically counts bytes and allocates buffers
3. *Labels* are used to keep track of **addresses** which prevents common machine-language mistakes

21

1. Consistent Instructions

- Assembly combines related machine instructions into a single notation *(and name)* called a *mnemonic*
- For example, the following machine-language actions are different, but related:
  - register → memory
  - register → register
  - constant → register

22

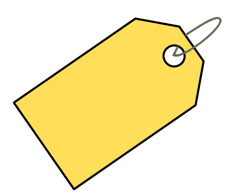
2. Count and Allocate Buffers

- Assembly automatically counts bytes and allocates buffers
- Miscounts (when done by hand) can be very problematic – and can lead to hard to find errors

23

3. Labels & Addresses

- Assembly uses *labels* to store **addresses**
- Used to keep track of locations in your programs
  - data
  - subroutines (functions)
  - …and much more



24

Battle of the Syntax

- The basic concept of assembly's notation and syntax hasn't changed
- However, there are two major competing notations
- They are *just* different enough to make it confusing for students and programmers *(who are used to the other notation)*

25

Battle of the Syntax

- AT&T Syntax
  - dominate on UNIX / Linux systems
  - registers prefixed by %, values with $
  - receiving register is last
- Intel Syntax
  - *actually created by Microsoft*
  - dominate on DOS / Windows systems
  - neither registers or values have a prefix
  - receiving register is first

26

AT&T Example

```
# Just a simple add

mov $42, %rbx        #rbx = 42
mov value, %rdx      #rdx = value
add %rbx, %rax       #rax += rbx
```


27

Intel Example

```
# Just a simple add

mov rbx, 42          #rbx = 42
mov rdx, value       #rdx = value
add rax, rbx         #rax += rbx
```


28

Assembly Program Structure

How these little beasties are organized

29

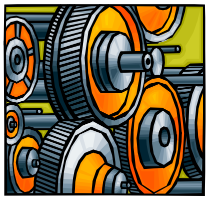
Assembly Programs

- Assembly programs are divided into two sections
- *data section* allocate the bytes to store your constants, variables, etc…
- *text section* contains the instructions that will make up your program

30

Directives

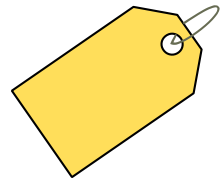
- A *directive* is a special command for the assembler
- Notation: starts with a period
- What they do:
  - allocate space
  - define constants
  - start the text or data section
  - make labels "global" for the linker



31

Labels

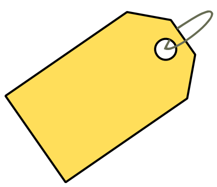
- You can define *labels* by following an identifier with a colon
- As the assembler is reading your program, it is generating machine code instructions and storage



32

Labels

- When the assembler sees a label declaration, it will save the current address (at that point) into a table
- Anytime you use a label, it is replaced by that address
- Labels are addresses



33

Hello World – Using csc35.o

```
.intel_syntax noprefix
.data
message:
   .ascii "Hello World!\n\0"

.text
.global Main

Main:
    lea   rdx, message
    call  PrintString

    call  EndProgram
```


34

Hello World – Using csc35.o

```
.intel_syntax noprefix
.data
message:
   .ascii "Hello World!\n\0"

.text
.global Main

Main:
    lea   rdx, message
    call  PrintString

    call  EndProgram
```

Data Section

35

Data Section

Use Intel format

No prefix characters

```
.intel_syntax noprefix
.data
message:
   .ascii "Hello World!\n\0"
```

Start data section

36

Data Section

```
.intel_syntax noprefix
.data
message:
    .ascii "Hello World!\n\0"
```

Create a label called 'message'. It will store an address.

Allocate the bytes required to store text

37

Hello World – x86, Linux

```
.intel_syntax noprefix
.data
message:
    .ascii "Hello World!\n\0"

.text
.global Main

Main:
    lea   rdx, message
    call  PrintString

    call  EndProgram
```

Text / Code Section

38

Text / Code Section

```
.text
.global Main

Main:
    lea   rdx, message
    call  PrintString

    call  EndProgram
```

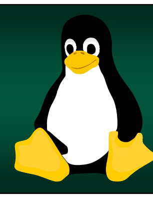
Start text section

Make label visible to the linker. Header will call Main

Loads the Effective Address 'message' into rdx

Call the library subroutine (it needs an address in rdx)

39



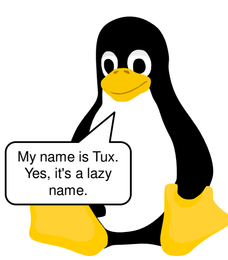
Basics of UNIX

Feel the pow-wah of the dark side

40

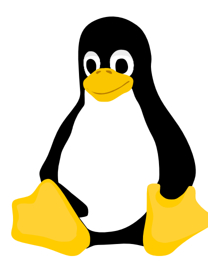
Basics UNIX

- UNIX was developed at AT&T's Bell Labs in 1969
- Design goals:
  - operating system for mainframes
  - stable and powerful
  - but not exactly easy to use – GUI hadn't been invented yet



41

Basics UNIX



- There are versions of UNIX with a nice graphical user interface
- A good example is all the various versions of Linux
- However, all you need is a command line interface

42

Command Line Interface

- Command line interface is text-only
- But, you can perform all the same functions you can with a graphical user interface
- This is how computer scientists have traditionally used computers

```
>gcc hello.c
>ls
a.out hello.c

>a.out
Hello world!
```


43

Command Line Interface

- Each command starts with a name followed by zero or more arguments
- Using these, you have the same abilities that you do in Windows/Mac

Spaces separate name & arguments

```
name argument1 argument2 …
```


44

`ls` Command

- Short for *List*
- Lists all the files in the current directory
- It has arguments that control how the list will look
- Notation:
  - directory names have a slash suffix
  - programs have an asterisk suffix

45

`ls` Command

```
> ls
a.out*  csc35/  html/  mail/
test.asm
```


46

`ll` Command

- Short for *List Long*
- This command is a shortcut notation for **`ls -l`**
- Besides the filename, its size, access rights, etc… are displayed

47

`ll` Command

```
> ll

-rwx------  1 cookd othcsc  4650 Sep 10 17:44 a.out*
drwx------  2 cookd othcsc  4096 Sep  5 17:49 csc35/
drwxrwxrwx 10 cookd othcsc  4096 Sep  6 11:04 html/
drwxrwxrwx  2 cookd othcsc  4096 Jun 20 17:58 mail/
-rw-------  1 cookd othcsc    74 Sep 10 17:44 test.asm
```


48

`nano` Application

- Nano is the UNIX text editor (well, the best one – that is)
- It is very similar to Windows Notepad – but can be used on a terminal
- You will use this to write your programs

49

`nano` Application

- Nano will open and edit the filename provided
- If the file doesn't exist, it will create it

```
nano filename
```


50

`as` Command

- This is the GNU assembler
- It will take an assembly program and convert it into an object
- You will be alerted of any syntax errors or unrecognized mnemonics (typos)

51

`as` Command

- The **–o** specifies the next name listed is the output file
- So, the second is the output file (object)
- The third is your input (assembly)

```
as –o lab.o lab.asm
```


52

`as` Command

- **Be very careful** – anything after –o will be destroyed
- There is no "undo" in UNIX!
- Check the two extensions for "o" then "asm"

```
as –o lab.o lab.asm
```


53

`as` Command

```
> ls
lab.asm

> as –o lab.o lab.asm

> ls
lab.asm  lab.o
```


54

`ld` Command

- This is the GNU linker
- It will take one (or more) objects and link them into an executable
- You will be alerted of any unresolved labels

55

`ld` Command

- The `-o` specifies the next name is the output
- The second is the output file (executable)
- The third is your input objects (1 or more)

```
ld -o a.out csc35.o lab.o
```


56

`ld` Command

- Be very careful – if you list your input file (an object) first, it will be destroyed
- I will provide the "csc35.o" file

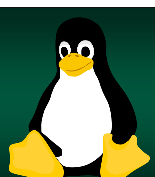
```
ld -o a.out csc35.o lab.o
```


57

`ld` Command

```
> ls
lab.o   csc35.o

> ld -o a.out lab.o csc35.o

> ls
lab.o   csc35.o   a.out*
```


58



UNIX Directories

Feel the pow-wah of the dark side

59

`cd` Command

- Short for *Change Directory*
- Allows you to change your current working directory
- If you specify a folder name, you will move into it
- If you use .. (two dots), you will go to the parent folder

60

`cd` Command

```
> cd csc35

> cd ..
```

Move into csc35 folder

Return to parent folder

61

`pwd` Command

- Short for *Print Working Directory*
- It displays the path your current directory (the one you are looking at).
- Slashes separate the directory names

62

`pwd` Command

```
> pwd

/gaia/class/student/cookd
```


63

`mkdir` Command

- Short for *Make Directory*
- Essentially the same as making a new subfolder in Windows or Mac-OS
- You may want to create one to store your CSc 35 work

64

`mkdir` Command

```
> ls
a.out*  html/  mail/  test.asm

> mkdir csc35

> ls
a.out*  csc35/  html/  mail/  test.asm
```


65

`rm` Command

- Short for *Remove*
- It essentially deletes a file
- Be careful…
  - files don't go into a "recycle bin"
  - they are gone forever!
- It can also delete multiple files using patterns

66

rm Command

```
> ls
a.out*  html/  mail/  test.asm

> rm a.out

> ls
html/  mail/  test.asm
```

Fall 2024 Sacramento State - Cook - CSc 35 67

67

Control Logic

Part 4

1

Intel x86 Jump Instructions

Fly over code

2

Operations: Program Flow Control

- Unlike high-level languages, processors don't have fancy expressions or blocks
- Programs are controlled by jumping over blocks of code

3

Operations: Program Flow Control

- The processor moves the instruction pointer *(where your program is running in memory)* to a new address and execution continues

4

Types of Jumps: Unconditional

- *Unconditional jumps* simply transfers the running program to a new address
- Basically, it just "gotos" to a new line
- These are used extensively to recreate the blocks we use in 3GLs (like Java)

5

Instruction: Jump

```
JMP  address
```

Usually a label – a constant that holds an address

6

Infinite Loop

```
.intel_syntax noprefix
.data
message:
   .ascii "I'm getting dizzy!\n\0"

.text
.global _start

_start:
    lea  rax, message
Loop:
    call  PrintString
    jmp   Loop
```


7

Infinite Loop

```
_start:
    lea  rdx, message
Loop:
    call PrintString
    jmp  Loop
```


8

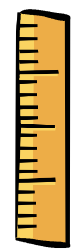
Conditional Jumps

- *Conditional jumps* (aka *branching*) will only jump if a certain condition is met
- What happens
  - processor jumps if and only if a specific status is set
  - otherwise, it simply continues with the next instruction

9

Instruction: Compare

- Performs a comparison operation between two arguments
- The result of the comparison is used for conditional jumps
- We will get into how this works a tad later



10

Instruction: Compare

Register, Memory

CMP ***arg1* , *arg2***

Immediate, Register, Memory

11

Conditional Jumps

- x86 contains a large number of conditional jump instructions
- x86 assembly has several names for the same instruction – which adds readability

12

Conditional Jumps

Jump	Description
JE	Jump Equal
JNE	Jump Not equal
JG	Jump Greater than
JGE	Jump Greater than or Equal
JL	Jump Less than
JLE	Jump Less than or Equal

13

Conditional Jump Example

```
_start:
    cmp   rax, 13
    je    Equal
    ...
Equal:
    ...
```

rax = 13?

14

Conditional Jump Example

```
_start:
    mov   rax, 42
    cmp   rax, 13
    jge   Bigger
    ...
Bigger:
    add   rax, 5
```

rax >= 13?

15

If Statements on the x86

How to we conditionally execute code?

16

If Statements in Assembly

- High-level programming language have easy to use If-Statements
- However, processors handle all branching logic using jumps
- You basically jump over true and else blocks

17

If Statements in Assembly

- Converting from an If Statement to assembly is easy
- Let's look at If Statements…
  - block is only executed if the expression is true
  - so, if the expression is false your program will skip over the block
  - this is a jump…

18

If Statement jumps over code

```
rax = 18;
if (rax >= 21)
{
   //true part
}
rbx = 12;
```

False

19

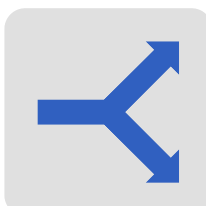
Converting an If Statement

- Compare the two values
- If the result is *false* …
  - then jump over the true block
  - you will need label to jump to
- To jump on false, reverse your logic
  - `a < b` → `not (a >= b)`
  - `a >= b` → `not (a < b)`

20

Please Note…

- Following examples use *very* generic label names
- In your program, each label you create must be unique
- So, please don't think that each label (as it is typed) is "the" label you need to use



21

Converting an If Statement

```
if (rax >= 21)
{
   //true block
}
//end
```

Greater-Than or Equal
So, jump on Less-Than

22

Jump over true part

```
      cmp   rax, 21
      jl    End

      #true block

   End:
```

Branch when false. JL (Jump Less Than) is the opposite of JGE

23

Jump over true part

```
      cmp   rax, 21
      jl    End

      #true block

   End:
```

Jumps over true part

24

Else Clause

- The Else Clause is a tad more complex
- You need to have a true block and a false block
- Like before…
  - you must jump over instructions
  - just remember… *the program will continue with the next instruction unless you jump!*

25

Else Clause

```
if (rax >= 21)
{
   //true block
}
else
{
   //false block
}
//end
```


26

Jump over true part

```
    cmp  rax, 21
    jl   Else

    #true block
    jmp  End
Else:
    #false block
End:
```

Jump to false block

False block flows down to End

27

Jump over true part

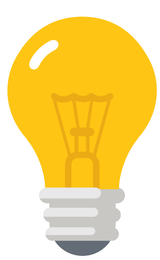
```
    cmp  rax, 21
    jl   Else

    #true block
    jmp  End
Else:
    #false block
End:
```

If we run the true block, we have to jump over the false block

28

Alternative Approach

- In these examples, I put the False Block first and used inverted logic for the jump
- You can construct If Statements without inverting the conditional jump, but the format is layout is different



29

If Statement – No Else

```
    cmp  rax, 21
    jge  Then
    jmp  End
Then:
    #true block
End:
```

Jumps to true block

30

If Statement – No Else

```
        cmp   rax, 21
        jge   Then
        jmp   End
    Then:
        #true block
    End:
```

Jump to end if false (it didn't jump with JGE)

31

If Statement with Else

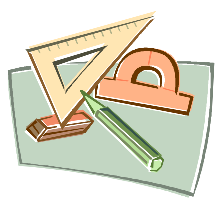
```
        cmp   rax, 21
        jge   Then

        #false block
        jmp   End
    Then:
        #true block
    End:
```

Notice that this is identical to the last slide – the false block is just empty

32

3 Rules of Engineering

1. If it works… *it works!*
2. If it ain't broke… *don't fix it!*
3. Reread rules 1 and 2 you moron!



33

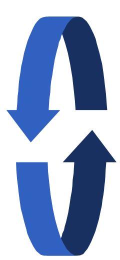
While Loops

Doing the same thing again and again … and again

34

While Statement

- Processors do not have While Statements – just like If Statements
- Looping is performed much like an implementing an If Statement
- A While Statement is, in fact, the same thing as an If Statement



35

Converting a While Statement

- To create a While Statement
  - start with an If Statement and…
  - add an unconditional jump at the end of the block that jumps to the beginning
- You will "branch out" of an infinite loop
- Structurally, this is almost identical to what you did before
- However, you do need another label :(

36

Converting an While Statement

```
while (rax < 21)
{
   //true block
}
//end
```

Less-Than.
So, jump on
Greater-Than or Equal

37

Converting an While Statement

```
While:
    cmp  rax, 21
    jge  End

    #true block
    jmp  While
End:
```

Loop after
block executes

38

Converting an While Statement

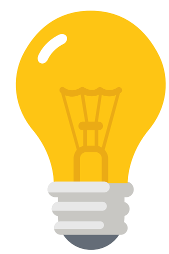
```
While:
    cmp  rax, 21
    jge  End

    #true block
    jmp  While
End:
```

Escape infinite
loop

39

Alternative Approach

- Before, we created an If Statement by inverting the branch logic (jump on false)
- You can, also implement a While Statement without inverting the logic
- Either approach is valid – use what you think is best



40

Alternative Approach

```
while (rax < 21)
{
   //true block
}
//end
```


41

Alternative Approach

```
While:
    cmp  rax, 21
    jl   Do
    jmp  End
Do:
    #true block
    jmp  While
End:
```

Jumps to Do
Block

42

Alternative Approach

```
While:
    cmp  rax, 21
    jl   Do
    jmp  End
Do:
    #true block
    jmp  While
End:
```

jl didn't jump, so jump out of the loop

43

Alternative Approach

```
While:
    cmp  rax, 21
    jl   Do
    jmp  End
Do:
    #true block
    jmp  While
End:
```

Repeat the loop

44

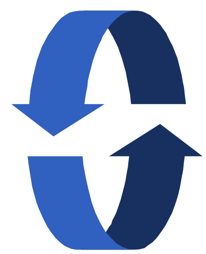
Do Loops

Post-Test While Loops

45

Do Loops

- Programming languages also support post-test loop statements
- Many programming languages use the keyword "repeat" or "do"
- Easier than While Statements



46

Converting Do Loops

```
do
{
   //true block
}
while (rax < 21);
//end
```

We jump UP when TRUE

47

Converting Do Loops

```
Do:
    #true block

    cmp  rax, 21
    jl   Do
```

Positive logic

48

Alternative Approach

- You can also implement Do Loops using negative logic
- But it requires a few an extra label and jump statement

49

Alternative Approach

```
Do:
    #true block

    cmp  rax, 21
    jge  End
    jmp  Do
End:
```

Negative logic

50

Alternative Approach

```
Do:
    #true block

    cmp  rax, 21
    jge  End
    jmp  Do
End:
```

Infinite loop

51

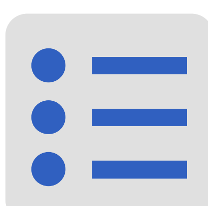
Switch Statements on the x86

Reason for the C, Java, and C# design

52

Switch Statements on the x86

- You might have noticed the strange behavior of Switch statements in C, Java, and C#
- Java and C# inherited their behavior from C



53

Switch Statements on the x86

- C, in turn, was designed for embedded systems
- Language creates very efficient assembly code
- The Switch Statement converts easily to efficient code

54

Switch Statement

- It is very efficient because…
  - it is restricted to integer constants
  - once a case is matched, no others are checked
  - they can fall through to match multiple values
- So, how?
  - start of the statement sets up just 1 register
  - compared to each "case" constant
  - jumps to a label created for each

55

Switch Statement Syntax

```
switch (integer)
{
  case value:
     Statements

  default:
     Statements
}
```

integer expression

You can have as many of these as needed

Executed if nothing matched

56

C/Java Code

```
switch (month)
{
   case 10:
      Halloween();
   case 11:
      Thanksgiving();
   default:
      Christmas();
}
```


57

Assembly Code

```
    mov  rax, month
    cmp  rax, 10
    je   case_10
    cmp  rax, 11
    je   case_11
    jmp  default

case_10:
    call Halloween
case_11:
    call Thanksgiving
default:
    call Christmas
```


58

Assembly Code

```
    mov  rax, month
    cmp  rax, 10
    je   case_10
    cmp  rax, 11
    je   case_11
    jmp  default

case_10:
    call Halloween
case_11:
    call Thanksgiving
default:
    call Christmas
```

Jump header

59

Assembly Code: Jump Header

```
mov  rax, month
cmp  rax, 10
je   case_10
cmp  rax, 11
je   case_11
jmp  default
```

case 10:

case 11:

default:

60

Assembly Code

```
    mov  rax, month
    cmp  rax, 10
    je   case_10
    cmp  rax, 11
    je   case_11
    jmp  default

case_10:
    call Halloween
case_11:
    call Thanksgiving
default:
    call Christmas
```

Case Body

61

Assembly Code: The Case Body

```
case_10:
    call Halloween
case_11:
    call Thanksgiving
default:
    call Christmas
```

Each "falls through". They are just labels!

62

Fall-Through Labels

```
10
Halloween
Thanksgiving
Christmas
```


63

Break Statement

- Even in the last example, we still fall-through to the default
- The "Break" Statement is used exit a case
- Semantics
  - simply jumps to a label after the last case
  - so, break converts directly to a single jump

64

Java Code

```
switch (month)
{
  case 10:
     Halloween();
     break;
  case 11:
     Thanksgiving();
     break;
  default:
     Christmas();
}
```

Let's jump to the end

65

Assembly Code: The Cases

```
case_10:
    call Halloween
    jmp  End
case_11:
    call Thanksgiving
    jmp  End
default:
    call Christmas
End:
```

Break jumps to the end

66

When Fallthrough Works

- The fallthrough behavior of C was designed for a reason
- It makes it easy to combine "cases" – make a Switch Statement match multiple values
- … and keeps the same efficient assembly code

67

Java Code: Primes from 1 to 10

```
switch (number)
{
  case 2:
  case 3:
  case 5:
  case 7:
     result = True;
     break;
  default:
     result = False;
}
```

Match Multiple

68

Primes: Jump Header

```
mov  rax, number

cmp  rax, 2
je   case_2
cmp  rax, 3
je   case_3
cmp  rax, 5
je   case_5
cmp  rax, 7
je   case_7

jmp  default
```

These are our primes

69

Assembly Code: The Cases

```
case_2:
case_3:
case_7:
case_9:
     movq  result, 1
     jmp   End
default:
     movq  result, 0
```

All these labels will be at the same address. You, of course, would write prettier code.

70

Arithmetic Logic Unit

Part 5

1

Adding Binary Integers

1 + 1 = 10

2

Adding Binary Integers

- Computer's add binary numbers the same way that we do with decimal
- Columns are aligned, added, and "1's" are carried to the next column
- In computer processors, this component is called an *adder*

3

Adding Base 10 Numbers

	1	1		
	2	7	8	1
+	3	7	2	1
	6	5	0	2

4

Adding Binary Example

		1	1	1		1	1		
118		0	1	1	1	0	1	1	0
51	+	0	0	1	1	0	0	1	1
169		1	0	1	0	1	0	0	1

5

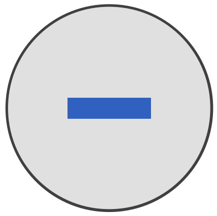
Negative Binary Integers

Have a positive attitude about negatives

6

Negative Binary Numbers

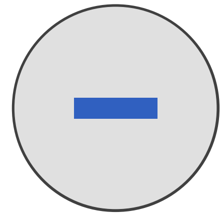
- When we write a negative number, we generally use a "-" as a prefix character
- However, binary numbers can only store ones and zeros



7

Negative Binary Numbers

- So, how we store a negative a number?
- When a number can represent both positive and negative numbers, it is called a *signed integer*
- Otherwise, it is *unsigned*



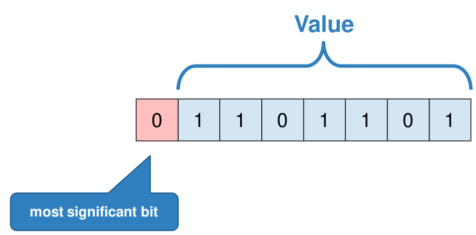
8

Signed Magnitude

- One approach is to use the most significant bit (msb) to represent the negative sign
- If positive, this bit will be a zero
- If negative, this bit will be a 1
- This gives a byte a range of -127 to 127 rather than 0 to 255

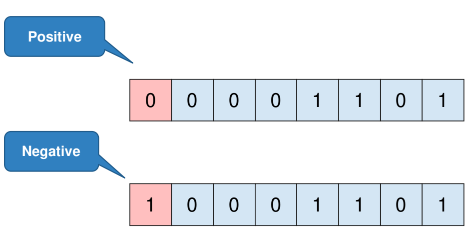
9

Signed Magnitude



10

Signed Magnitude: 13 and -13



11

Signed Magnitude Drawback #1

- When two numbers are added, the system needs to check and sign bits and act accordingly
- For example:
  - if both numbers are positive, add values
  - if one is negative subtract it from the other
  - etc…
- There are also rules for subtracting

12

Signed Magnitude Drawback #2

- Also, signed magnitude also can store a positive <u>and</u> negative version of zero
- Yes, there are two zeroes!
- Imagine having to write Java code like…

```
if (x == +0 || x == -0)
```


13

Oh noes! Two zeros?

+0

0	0	0	0	0	0	0	0

−0

1	0	0	0	0	0	0	0

14

1's Complement

- Rather than use a sign bit, the value can be made negative by *inverting* each bit
  - each 1 becomes a 0
  - each 0 becomes a 1
- Result is a "complement" of the original
- This is logically the same as subtracting the number from 0

15

Advantages / Disadvantages

- Advantages over signed magnitude
  - very simple rules for adding/subtracting
  - numbers are simply added:
    5 - 3 is the same as 5 + -3
- Disadvantages
  - positive and negative zeros still exist
  - so, it's not a perfect solution

16

1's Complement: 13 and -13

Positive

0	0	0	0	1	1	0	1

Negative

1	1	1	1	0	0	1	0

17

1's Complement Has Two Zeros

+0

0	0	0	0	0	0	0	0

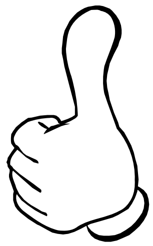
−0

1	1	1	1	1	1	1	1

18

2's Complement

- Practically all computers use *2's Complement*
- Similar to 1's complement, but after the number is inverted, 1 is added to the result
- Logically the same as:
  - subtracting the number from 2^n
  - where n is the total number of bits in the integer



19

2's Complement Advantages

- Since negatives are subtracted from 2^n
  - they can simply be added
  - the extra carry 1 (if it exists) is discarded
  - this simplifies the hardware considerably since the processor only has to add
- The +1 for negative numbers…
  - makes it so there is only one zero
  - values range from -128 to 127

20

2's Complement: 13 and -13

Positive	0	0	0	0	1	1	0	1
Negative	1	1	1	1	0	0	1	1

Add 1

21

Just One Zero!

0	0	0	0	0	0	0	0	0
−1	1	1	1	1	1	1	1	1

22

Adding 2's Complement

	1	1	1	1			1	1
91	0	1	0	1	1	0	1	1
+ -13	1	1	1	1	0	0	1	1
78	0	1	0	0	1	1	1	0

23

Unsigned or Signed?

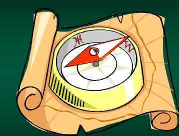
- In reality, processors don't know *(or care)* if a number if unsigned or signed
- The hardware works the same either way
- It's your responsibility to keep track if it's signed/unsigned

24

It's Your Responsibility

- In ma... cases, you must use the c... t instruction - based ... *you* are treating ... ned or
- ... *great programming* ... *er comes great* ... *onsibility*

25



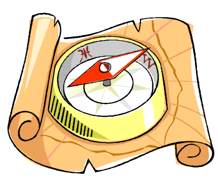
Relative Addressing

Jumping of the instruction pointer

26

Relative Addressing

- In *relative addressing*, a value is added to a instruction pointer (e.g. program counter)
- This allows access a fixed number of bytes *up or down* from the instruction pointer



27

Relative Addressing

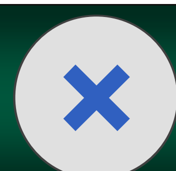
- Often used in conditional jump statements
  - jumps are often short – not a large number of instructions
  - so, the instruction only stores the value to add to the program counter
  - practically all processors us this approach
- Also used to access local data – load/store

28

Relative Addressing Advantages

- The instruction can just store the *difference* (in bytes) from the current instruction address
- It takes less storage than a full 64-bit address
- It also allows a program to be stored anywhere in memory – *and it will still work!*

29



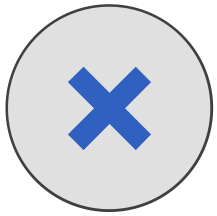
Multiplying Binary Numbers

$11 \times 11 = 1001$

30

Multiplying Binary Numbers

- Many processors today provide complex mathematical instructions
- However, the processor only needs to know how to add
- Historically, multiplication was performed with successive additions



31

Multiplying Scenario

- Let's say we have two variables: A and B
- Both contain integers that we need to multiply
- Our processor can *only* add (and subtract using 2's complement)
- How do we multiply the values?

32

Multiplying: The Bad Way

- One way of multiplying the values is to create a For Loop using one of the variables – A or B
- Then, inside the loop, continuously add the other variable to a running total

33

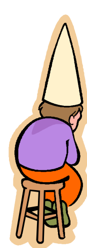
Multiplying: The Bad Way

```
total = 0;
for (i = 0; i < A; i++)
{
   total += B;
}
```


34

Multiplying: The Bad Way

- If A or B is large, then it could take a long time
- This is incredibly inefficient
- Also, given that A and B could contain drastically different values – the number of iterations would vary
- Required time is not constant



35

Multiplying: The Best Way

- Computers can multiply by using long multiplication – *just like you do*
- Number of additions is fixed to 8, 16, 32, 64 depending on the size of the integer
- The following example multiplies 2 unsigned 4-bit numbers

36

Unsigned Integer: 13 × 10

```
     1101
×    1010
---------
     0000
+
---------
```


37

Unsigned Integer: 13 × 10

```
     1101
×    1010
---------
     0000
    1101
+
---------
```


38

Unsigned Integer: 13 × 10

```
     1101
×    1010
---------
     0000
    1101
   0000
+
---------
```


39

Unsigned Integer: 13 × 10

```
     1101
×    1010
---------
     0000
    1101
   0000
+ 1101
---------
```


40

Unsigned Integer: 13 × 10

```
        1101
×       1010
------------
        0000
       1101
      0000
+    1101
------------
130 10000010
```


41

Multiplication Doubles the Bit-Count

- When two numbers are multiplied, the product will have twice the number of digits
- Examples:
  - 8-bit × 8-bit → 16-bit
  - 16-bit × 16-bit → 32-bit
  - 32-bit × 32-bit → 64-bit
  - 64-bit × 64-bit → 128-bit

42

Multiplication Doubles the Bit-Count

- So, how do we store the result?
- It is often too large to fit into any single existing register
- Processors can…
  - fit the result in the original bit-size *(and raise an overflow if it does not fit)*
  - …or store the new double-sized number

43

x86 Mathematics

Complex Math is Complex

44

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…

45

Addition

Immediate, Register, Memory

```
ADD target, value
```

Register, Memory

46

Subtraction

Immediate, Register, Memory

```
SUB target, value
```

Register, Memory

47

Negate (2's complement)

```
NEG register
```


48

Example: Simple Add

```
MOV   rax, 17
ADD   rax, 2
```

Move value into RAX

RAX += 2

49

x86 Multiplication

Complex Math is Complex

50

Multiplication & Division

- The x86 treats multiplication quite differently than add/subtract
- Why? Intel was designed as a business processor and high-precision math is paramount

51

Multiplication Review

- Remember: when two *n* bit numbers are multiplied, result will be *2n* bits
- So…
  - two 8-bit numbers → 16-bit
  - two 16-bit numbers → 32-bit
  - two 32-bit numbers → 64-bit
  - two 64-bit numbers → 128-bit

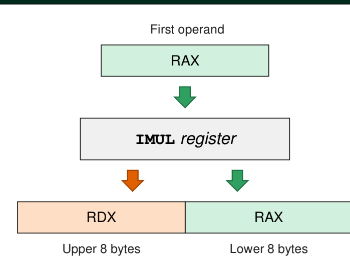
52

Multiplication on the x86

- Intel stores the product into <u>two</u> registers
  - RAX will contain the lower 8 bytes
  - RDX will contain the upper 8 bytes
- This maintains the high-precision result
- Instruction inputs are strange
  - first operand is <u>must</u> be stored in RAX
  - second operand <u>must</u> be a register or memory

53

x86 Multiplication



54

Multiply - Signed

```
IMUL operand
```

Register or Memory only

55

Multiply - Unsigned

```
MUL operand
```

Register or Memory only

56

Signed Multiply: 1846 by 42

```
MOV   rax, 1846     #First operand
MOV   rbx, 42       #Need register for MUL
IMUL  rbx           #RAX gets low 8 bytes
                    #RDX gets high 8 bytes
```


57

Multiplication Tips

- Even though you are just using RAX as input, both RAX and RDX will change
- Be aware that you might lose important data, and backup to memory if needed

58

Additional x86 Multiply Instructions

- Over time, designers requested a low-precision version of multiplication
- Intel added "short" IMUL instructions that store into a single register
- Please Note: these do not exist for MUL

59

IMUL (few more combos)

Immediate, Register, Memory

```
IMUL target, value
```

Register

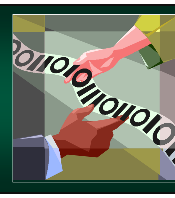
60

Signed Multiply: 1846 by 42

```
MOV   rax, 1846
IMUL  rax, 42
```

This works, but could cause an overflow

61



Extending Byte Size

Converting from 8-bit to 16-bit and more

62

Extending Unsigned Integers

- Often in programs, data needs to moved to a integer with a larger number of bits
- For example, an 8-bit number is moved to a 16-bit representation

63

Extending Unsigned Integers

- For unsigned numbers is fairly easy – just add zeros to the left of the number
- This, naturally, is how our number system works anyway: 456 = 000456

64

Unsigned 13 Extended

0	0	0	0	1	1	0	1

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

65

Extending Signed Integers

- When the data is stored in a signed integer, the conversion is a little more complex
- Simply adding zeroes to the left, will *convert a negative value to a positive one*
- Each type of signed representation has its own set of rules

66

2's Complement Incorrectly Done

-13

1	1	1	1	0	0	1	1

243

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

67

Sign Magnitude Extension

- In signed magnitude, the most-significant bit (msb) stores the negative sign
- The new sign-bit needs to have this value
- Rules:
  - copy the old sign-bit to the new sign-bit
  - fill in the rest of the new bits with zeroes – *including the old sign bit*

68

Sign Magnitude Extended: +77

0	1	0	0	1	1	0	1

69

Sign Magnitude Extended: +77

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

70

Sign Magnitude Extended: -77

1	1	0	0	1	1	0	1

71

Sign Magnitude Extended: -77

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

72

1's & 2's Complement Extension

- 1's and 2's Complement is very simple to convert to a larger representation
- Remember that we inverted the bits and added 1 to get a negative value
- Rule: copy the old most-significant bit to all the new bits

73

1's & 2's Complement Extended: +77

0	1	0	0	1	1	0	1

74

1's & 2's Complement Extended: +77

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

75

1's & 2's Complement Extended: -77

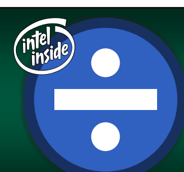
1	0	1	1	0	0	1	1

76

1's & 2's Complement Extended: -77

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

77



x86 Division

Complex Math is Complex

78

Division on the x86

- Division on the x86 is very interesting
- Since multiplication stores into to two registers, divide uses these as the numerator
- Numerator is fixed as:
  - RAX contains the lower 8 bytes
  - RDX contains the upper 8 bytes

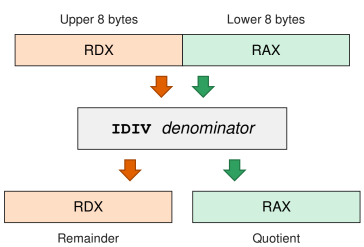
79

Division on the x86

- These two registers are also used for the result
- The output contains:
  - RAX will contain the quotient (the whole number)
  - RDX will contain the remainder

80

x86 Division



81

Divide - Signed

```
IDIV denominator
```

Register or Memory only

82

Divide - Unsigned

```
DIV denominator
```

Register or Memory only

83

Dividing Rules

- The numerator **must** be expanded to the destination size (twice the original)
- Why? Multiplication doubles the number of digits; division does the opposite
- This must be done before the division - otherwise the result will be incorrect

84

On the Intel…

- You <u>must</u> setup RDX <u>before</u> you divide
- For unsigned: store 0 into it
- For signed-division:
  - RAX needs must be *sign-extended* into RDX
  - there are special instructions

85

Sign Extend Example

RAX

`00 00 00 00 00 00 00 00`

RDX	RAX
	`00 00 00 00 00 00 00 00`

86

Sign Extend Example

RAX

`00 00 00 00 00 00 00 00`

RDX	RAX
`00 00 00 00 00 00 00 00`	`00 00 00 00 00 00 00 00`

87

Sign Extend Example

Hex 8 = 1000 binary
This is a negative number

RAX

`80 00 00 00 00 00 00 00`

RDX	RAX
	`80 00 00 00 00 00 00 00`

88

Sign Extend Example

Hex 8 = 1000 binary
This is a negative number

RAX

`80 00 00 00 00 00 00 00`

RDX	RAX
`FF FF FF FF FF FF FF FF`	`80 00 00 00 00 00 00 00`

89

CWD (16 bit): Extend AX → DX

```
CWD
```

Convert Word to Double

90

CDQ (32 bit): Extend EAX → EDX

CDQ

Convert Double to Quad

91

CQO (64 bit): Extend RAX → RDX

Use this one!

CQO

Convert Quad to Oct

92

Divide 64-bit: -1846 by 42

```
MOV  rax, -1846     #RAX is the dividend
MOV  rbx, 42        #Divisor
CQO                 #Sign extend to RDX
IDIV rbx            #RAX gets quotient
                    #RDX gets remainder
```


93

How Compare Works

It's all math

94

Behind the scenes…

- The second argument is subtracted from the first
- The result of this computation is used to determine how the operands compare
- This subtraction result is discarded

95

But… why subtract?

- Why subtract the operands?
- The result can tell you which is larger
- For example: A and B are both positive…
  - A – B → positive number → A was larger
  - A – B → negative number → B was larger
  - A – B → zero → both numbers are equal

96

Instruction: Compare

Register, Memory

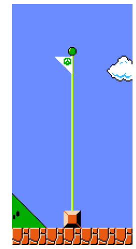
CMP ***value1, value2***

Immediate, Register, Memory

97

Flags

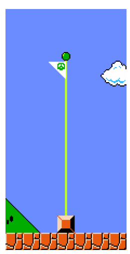
- A *flag* is a Boolean value that indicates the result of an action
- These are set by various actions such as calculations, comparisons, etc…



98

Flags

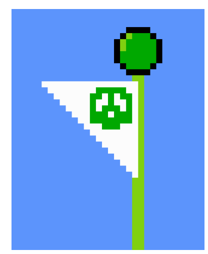
- Flags are typically stored as individual bits in the *Status Register*
- You can't change the register directly, but numerous instructions use it for control and logic



99

Zero Flag (ZF)

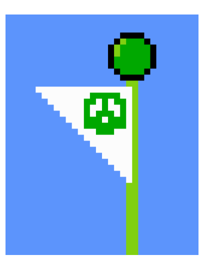
- True if the last computation resulted in zero (all bits are 0)
- For compare, the zero flag indicates the two operands are equal
- Used by quite a few conditional jump statements



100

Sign Flag (SF)

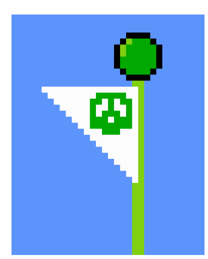
- True of the *most significant bit* of the result is 1
- This would indicate a negative 2's complement number
- Meaningless if the operands are interpreted as unsigned



101

Carry Flag (CF)

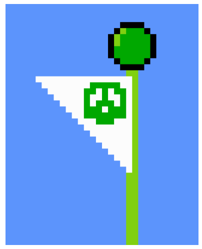
- True if a 1 is "borrowed" when subtraction is performed
- …or a 1 is "carried" from addition
- For unsigned numbers, it indicates:
  - exceeded the size of the register on addition
  - or an underflow (too small value) on subtraction



102

Overflow Flag (OF)

- Also known as "signed carry flag"
- True if the sign bit changed *when it shouldn't have*
- For example:
  - (negative – positive) should be negative
  - a positive result will set the flag
- For signed numbers, it indicates:
  - exceeded the register size
  - i.e. the value was too big/small



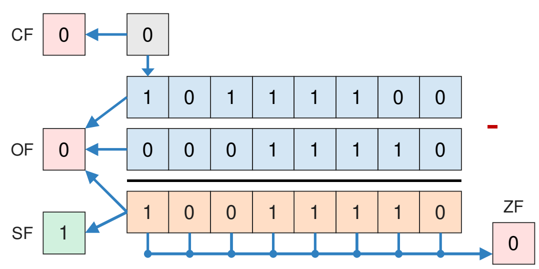
103

x86 Flags Used by Compare

Name	Description	When True
CF	Carry Flag	If a bit was "carried" or "borrowed" during math.
ZF	Zero Flag	All the bits in the result are zero.
SF	Sign Flag	If the most significant bit is 1.
OF	Overflow Flag	If the sign-bit changed when it shouldn't have.

104

-68 vs. 30 (if interpreted as signed)
188 vs. 30 (if interpreted as unsigned)



105

Jump on Equality

Jump	Description	When True
JE	Equal	ZF = 1
JNE	Not equal	ZF = 0

106

Signed Jump Instructions

Jump	Description	When True
JG	Jump Greater than	SF = OF, ZF = 0
JGE	Jump Greater than or Equal	SF = OF
JL	Jump Less than	SF ≠ OF, ZF = 0
JLE	Jump Less than or Equal	SF ≠ OF

107

Unsigned Jumps

Jump	Description	When True
JA	Jump Above	CF = 0, ZF = 0
JAE	Jump Above or Equal	CF = 0
JB	Jump Below	CF = 1, ZF = 0
JBE	Jump Below or Equal	CF = 1

108

Unsigned Conditional Jump Example

```
_start:
    mov  rax, 42
    cmp  rax, 13
    jae  Bigger
    ...
                    rax >= 13?
Bigger:
    add  rax, 5
```


109

Addressing

Part 6

1

Buffers

Creating your own space

2

Buffers

- A *buffer* is any allocated block of memory that contains data
- This can hold anything:
  - text
  - image
  - file
  - etc.…

3

Buffers

- There are several assembly *directives* which will allocate space
- We have covered a few of them, but there are many – all with a specific purpose

4

A few directives that create space

Directive	What it does
`.ascii`	Allocate enough space to store an ASCII string
`.quad`	Allocate 8-byte blocks with initial value(s)
`.byte`	Allocate byte(s) with initial value(s)
`.space`	Allocate any ***size*** of empty bytes (with initial values).

5

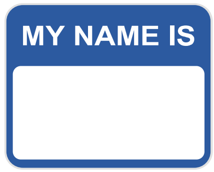
Labels are addresses

- Labels are used to keep track of memory locations
- They are stored, by the assembler, in a table
- Whenever a label is used in the program, the assembler substitutes the address

6

Labels are addresses

- The table of labels is stored in the *object file*
- That way the linker can resolve any unknown labels
- After the program is linked into an executable, only addresses exist. No labels.



7

Quad Directive

Let's assume Value = 2000

```
Value:
    .quad 74
```

2000	4A
2001	00
2002	00
2003	00
2004	00
2005	00
2006	00
2007	00

8

ASCII Directive Creates a Buffer

```
Greeting:
    .ascii "Hello\0"
```

This label will store an address... once the assembler finds where to store it.

Creates 6 bytes to store Hello. They are stored consequently.

9

Bytes are stored consecutively

Let's assume Greeting = 2000

```
Greeting:
    .ascii "Hello\0"
```

2000	48	H
2001	65	e
2002	6C	l
2003	6C	l
2004	6F	o
2005	00	\0

10

Same Thing!

```
Greeting:
    .byte 'H'
    .byte 'e'
    .byte 'l'
    .byte 'l'
    .byte 'o'
    .byte 0
```

Created byte by byte

Null character. Marks the end of a string

11

This works too!

```
Greeting:
    .ascii "Hello"
    .byte 0
```

Directives just create space. So, this creates a byte after the ASCII text.

12

Create a Buffer of Any Size

```
EmptyBuffer:
    .space 30
```

Create 30 bytes (defaults to 0x20 which is a space)

13

Create a Buffer of Any Size

Create 30 bytes. All of which are 0.

```
EmptyBuffer:
    .space 30, 0
```


14

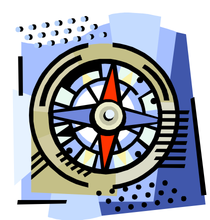
Addressing Modes Basics

How to interact with memory

15

Addressing Modes

- Processor instructions often need to access memory to read values and store results
- So far, we have used registers to read and store single values
- However, we need to:
  - access items in an array
  - follow pointers
  - and more!



16

Addressing Modes

- *How* the processor can locate and read data is called an *addressing mode*
- Information combined from registers, immediates, etc… to create a target address
- Modes vary greatly between processors

17

4 Basic Addressing Modes

- Immediate Addressing
- Register Addressing
- Direct Addressing
- Indirect Addressing

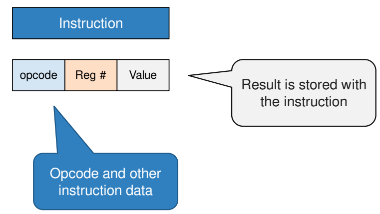
18

Immediate Addressing

- Immediate addressing is one of the most basic modes found on a processor
- Often a value is stored as part of the instruction
- As the result, it is *immediately* available
- Very common for assigning constants

19

Immediate Addressing



20

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

21

Example: Immediate Addressing

immediate

```
mov   rcx,  1947
```


22

Register & Immediate in Java

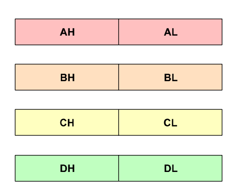
- The following, for comparison, is the equivalent code in Java
- The register file (for rcx) is set to the value 1947.

```
// rcx = 1947;
mov rcx, 1947
```


23

Register Addressing

- *Register addressing* is used in practically all computer instructions
- A value is read from or stored into one of the processor's registers



24

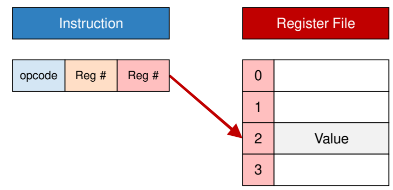
Transfer

- 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

25

Register Addressing



26

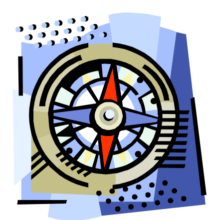
Load & Store

Saving and retrieving values

27

Load & Store

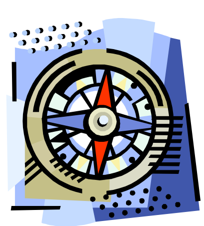
- Often data is accessed from memory
- Memory is an important part of von Neuman architecture
- As such, there are many ways of accessing it



28

Load & Store

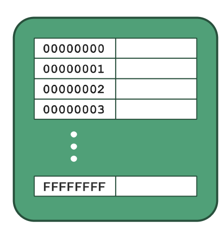
- On some processors, only Load and Store can access memory
- The Intel processor allows multiple instructions to have load/store capabitilies



29

Load

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



30

Load

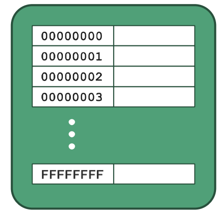
- A load needs to store the destination register as well as the address in memory
- Note that this is stored as an immediate

Opcode	Register	Immediate
Load	Destination	Source Address

31

Store

- A *Store* instruction, writes data from a register into the specified address
- So, it's the opposite of the Load



32

Store

- Like Load, the Store instruction needs to specify an address
- Note: the structure is identical to Load

Opcode	Register	Immediate
Store	Source	Destination Address

33

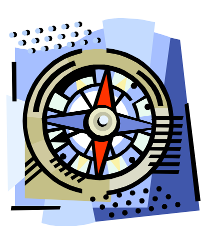
Direct Addressing

Using Memory for "Variables"

34

Direct Addressing

- In *direct addressing*, the processor reads data directly from an address
- Commonly used to:
  - get a value from a "variable"
  - read items in an array
  - etc...



35

Direct Addressing

36

Direct in Java

- Note: this a shortcut notation
- The full notation would use square brackets
- The assembler recognizes the difference automatically

```
// rdx = Memory[total];
mov rdx, total
```


37

Direct in Java (alternative notation)

- You can use the square-brackets if you want
- This way it explicitly show *how* the label is being used – it's a matter of preference

```
// rdx = Memory[total];
mov rdx, [total]
```


38

Example: Direct Load

```
.intel_syntax noprefix
.data
funds:
    .quad 100

.text
.global Main
Main:
    mov rdx, funds
```

64 bit integer with an initial value of 100.

Read 8 bytes at this address. Doesn't store the address in rdx.

39

Example: Direct

```
.intel_syntax noprefix
.data
funds:
    .quad 100

.text
.global Main
Main:
    mov rdx, [funds]
```

A bit more descriptive

40

Example: Direct Store

```
.intel_syntax noprefix
.data
funds:
    .quad 200

.text
.global Main
Main:
    mov  rcx, 2500
    mov  funds, rcx
```

Store rcx into Address "funds"

41

Example: Direct Store 2

```
.intel_syntax noprefix
.data
funds:
    .quad 100

.text
.global Main
Main:
    call ScanInteger
    mov  funds, rdx
```

You can store inputted values.

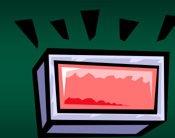
42

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

43



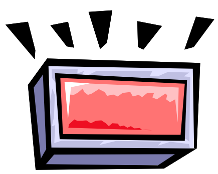
When to use **mov** and **lea**

The difference is huge!

44

When to use **mov** and **lea**

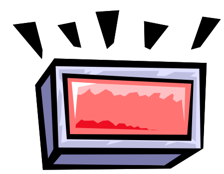
- Knowing when to use an address **or** the data *located at that address* is vital
- Using the wrong one can cause your program to malfunction or crash



45

Cause of the Segmentation Fault

- This is one of the most common mistakes in assembly programming



46

Using Move Correctly

```
.intel_syntax noprefix
.data
Year:
    .quad 1947

.text
.global Main
Main:
    mov  rdx, Year
    call PrintInteger
```

Creates 8 bytes

mov loads the data located at the address Year

47

Using move Correctly: Output

```
1947
```

Correct output. mov loaded the data from an address

48

Using `lea` by accident

```
.intel_syntax noprefix
.data
Year:
    .quad 1947

.text
.global Main
Main:
    lea  rdx, Year
    call PrintInteger
```

Creates 8 bytes

lea is going to store the address Year into rdx

49

Using `lea` by accident

```
6293248
```

That's wrong... very, very wrong

50

Why it Failed

```
.intel_syntax noprefix
.data
Year:
    .quad 1947

.text
.global Main
Main:
    lea  rdx, Year
    call PrintInteger
```

6293232	
6293240	
6293248	1947
6293256	
6293264	

1947 was being stored at this address

51

Sometimes, You Need the Address

- Of course, sometimes, you do need an address
- For example, PrintString
  - needs to know where the string is located so it can print a series of characters
  - so, it requires an address
  - `lea` is necessary

52

Using `lea` correctly

```
.intel_syntax noprefix
.data
Message:
    .ascii "Hello!!\0"

.text
.global Main
Main:
    lea  rdx, Message
    call PrintString
```

Loads the effective address into rdx

53

Using `lea` correctly: Output

```
Hello!!
```

Correct output. PrintString went to the address and printed characters

54

Using the mov rather than lea

```
.intel_syntax noprefix
.data
Message:
    .ascii "Hello!!\0"

.text
.global Main
Main:
    mov  rdx, Message
    call PrintString
```

Creates 8 bytes using ASCII values

Using mov rather than lea. rdx is 64-bit (8 bytes)

55

Using the mov rather than lea

```
Segmentation Fault (core dumped)
```

It crashed! We attempted to access memory we don't have permission to.

56

Cause of the Segmentation Fault

```
.intel_syntax noprefix
.data
Message:
    .ascii "Hello!!\0"

.text
.global Main
Main:
    mov  rdx, Message
    call PrintString
```

Message		
	48	H
	65	e
	6C	l
	6C	l
	6F	o
	21	!
	21	!
	00	\0

57

Cause of the Segmentation Fault

```
.intel_syntax noprefix
.data
Message:
    .ascii "Hello!!\0"

.text
.global Main
Main:
    mov  rdx, Message
    call PrintString
```

Grabs 8 bytes and creates a HUGE value

Message		
	48	H
	65	e
	6C	l
	6C	l
	6F	o
	21	!
	21	!
	00	\0

58

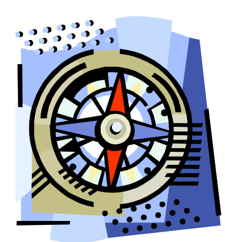
Indirect Addressing

The Power of Pointers

59

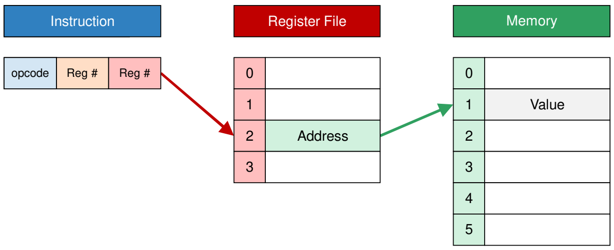
Indirect Addressing

- *Register Indirect* reads data from an address stored in register
- Same concept as a *pointer*
- Benefits:
  - it is just as fast as direct addressing
  - processor already has the address
  - … and very common



60

Register Indirect Addressing



61

Load Effective Address

- Load Effective Address stores the address into a register
- It computes the address (as it if was going to read from memory), but just stores that value

```
// rbx = total;
lea rbx, total
```


62

Load Effective Address

- So, just like normal direct addressing, the brackets are implied

```
// rbx = total;
lea rbx, [total]
```


63

Indirect in Java

- The following, for comparison, is the equivalent code in Java
- The value in rbx is used as the address to read from memory.
- *The brackets here are necessary!*

```
// rcx = Memory[rbx];
mov rcx, [rbx]
```


64

Example: Indirect

```
.intel_syntax noprefix
.data
total:
    .quad 451

.text
.global Main
Main:
    lea  rax, total
    mov  rbx, [rax]
```

64 bit integer. With an initial value of 451.

Load the address into rax

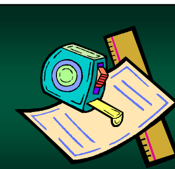
rbx gets the data from the address stored in rax

65

Memory Indexing

Part 7

1



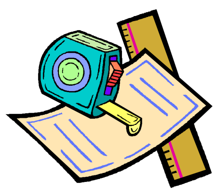
Sizing Instructions

How many bytes are you using?

2

Sizing Instructions

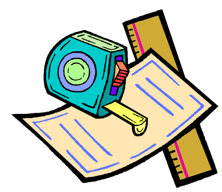
- The Intel can load/store 1-byte, 2-byte, 4-byte or 8-byte values
- Whenever a processor accesses memory, the instruction specifies how many bytes to access



3

Sizing Instructions

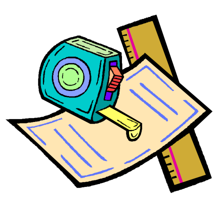
- The assembler will automatically fill this information in for you.
- *How?* If a register is used, the assembly can assume it by looking at size of the register



4

Sizing Instructions

- However, sometimes the number of bytes (1, 2, etc..) can't be determined
- In this case, the assembler will report an error
- … since it doesn't know how to encode the instruction



5

Example: How Many Bytes?

```
.intel_syntax noprefix
.data
total:
    .quad 0

.text
.global Main
Main:
    mov  total, 50
```

total is a target address. It doesn't have any implied size.

6

Example: How Many Bytes?

```
.intel_syntax noprefix
.data
total:
    .quad 0

.text
.global Main
Main:
    mov  total, 50
```

How many bytes is this? The value 50 can be stored in 1, 2, 4, or 8 bytes.

7

How Many Bytes?

- If the assembler can't infer how many bytes to access, it'll will report *"ambiguous operand size"*
- To address this issue…
  - GAS assembly allows you places a single character after the instruction's mnemonic
  - this suffix will tell the assembler how many bytes will be accessed during the operation

8

How Many Bytes

Suffix	Name	Size
b	byte	1 byte
s	short	2 bytes
l	long	4 bytes
q	quad	8 bytes

9

Example: Suffix Used

```
.intel_syntax noprefix
.data
total:
    .quad 0

.text
.global Main
Main:
    movq total, 50
```

Note the q. Now the assembler knows you mean "move quad".

10

Behind the Scenes of Arrays

All the mystery is revealed!

11

Arrays

- Computers do not have an 'array' data type
- So, how do you have array variables?
- When you create an array…
  - you allocate a block of memory
  - each element is located sequentially in memory – one right after each other

12

Arrays

- Every byte in memory has an address
- This is just like an array
- To get an array element
  - we merely need to compute the address
  - we must also remember that some values take multiple bytes – *so there is math*

13

Array Math Example

- Let's again assume that our buffer starts at address 2000
- The first array element is located at address 2000
- Arrays consists of bytes…
  - the second is 2001
  - the third is 2002
  - the fourth 2003
  - etc…

2000	H
2001	e
2002	l
2003	l
2004	o

14

Array Math Example – 16 bit

- First element uses 2000… 2001
- Since each array element is 2 bytes…
  - second address is 2002
  - third address is 2004
  - fourth address is 2006
  - etc…

2000	F0A3
2002	042B
2004	C1F1
2006	0D0B
2008	9C2A

15

Array Math Example – 64 bit

- First element uses 2000 to 2007
- Second address is 2008
- Third address is 2016
- Fourth address is 2024
- etc…

2000	446576696E20436F
2008	6F6B000000000000
2016	53616372616D656E
2024	746F205374617465
2032	4353433335000000

16

Behind the Scenes…

- So, when an array element is read, internally, a mathematical equation is used
- It uses the start of the first element, the array index, and the size of each element

```
start address + (index × size)
```


17

Behind the Scenes…

- *This is why the C Programming Languages uses zero as the first array element*
- If zero is used with this formula, it gets the start of the buffer

```
start address + (index × size)
```

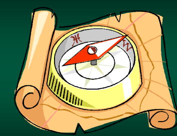

18

Behind the Scenes…

- Java uses zero-indexing because C does
- … and C does so it can create efficient assembly!

```
start address + (index × size)
```


19



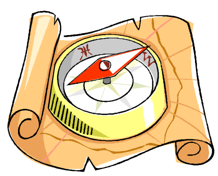
Indexing on the x64

Grabbing any byte

20

Indexing on the x64

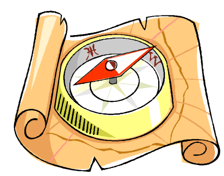
- The Intel x64 supports direct, indirect, indexing and scaling
- So, the Intel is very versatile in how it can access memory
- This is typical of CISC-ish architectures



21

Effective Addresses

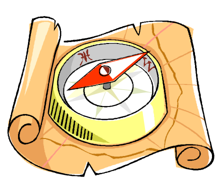
- Processors have the ability to create the *effective address* by combining data
- How it works:
  - starts with a base address
  - then adds a value (or values)
  - finally, uses this temporary value as the actual address



22

Effective Addresses

- Using the addresses stored in memory, registers, etc… is useful in programs
- Often programs contain *groups* of data
  - fields in an abstract data type
  - elements in an array
  - entries in a large table etc…



23

Terminology

- *Base-address* is the initial address
- *Displacement (aka offset)* is a constant (immediate) that is added to the address
- *Index* is a register added to the address
- *Scale* used to multiply the index before adding it to the address

24

x64 Effective Address Formula

Signed Constant

1, 2, 4 or 8

```
displacement + base + (index × scale)
```

Any Register

Any Register

25

Behind the Scenes…

- But wait, doesn't that formula look familiar?
- The addressing term "scale" is basically equivalent to "size" in this example
- Addressing and arrays work together flawlessly

```
start address + (index × size)
```


26

Addressing Notation in Assembly

- Intel Notation *(Microsoft actually created it)* allows you to specify the full equation
- The notation is very straight forward and mimics the equation used to compute the effective address
- Parts of the equation can be omitted, and the assembler will understand

27

Intel Notation

Register

Signed constant

```
[Base + Index * Scale + Displacement]
```

Register

1, 2, 4, 8

28

Notation (reg = register)

Mode	Syntax	Java Equivalent
Immediate	`value`	`value`
Register	`register`	`register`
Direct	`label`	`Memory[label]`
Direct Indexed	`[label + reg]`	`Memory[label + reg]`
Indirect	`[reg]`	`Memory[reg]`
Indirect Indexed	`[reg + reg]`	`Memory[reg + reg]`
Indirect Indexed Scale	`[reg + reg * scale]`	`Memory[reg + reg × scale]`

29

Addressing Notation in Assembly

- When you write an assembly instruction…
  - you specify all 4 four addressing features
  - however, notation fills in the "missing" items
- For example: for direct addressing…
  - Displacement → Address of the data
  - Base → Not used
  - Index → Not used
  - Scale → 1, irrelevant without an Index

30

Indexing Examples

- The following examples use addressing modes modify an ASCII buffer
- Let's assume that the start of the buffer **Talk** is **5000**

`Talk = 5000`

5000	48	H
5001	65	e
5002	6C	l
5003	6C	l
5004	6F	o

31

Example: Direct Index

```
# Talk = 5000

mov  rdi, 1
movb [Talk + rdi], 33
```

Using the rsi register for indexing, but you can use any register

ASCII 33 → !

5000	48	H
5001	33	!
5002	6C	l
5003	6C	l
5004	6F	o

32

Example: Direct Index (Scale 2)

```
# Talk = 5000

mov  rdi, 1
movb [Talk + rdi * 2], 33
```

5000	48	H
5001	65	e
5002	33	!
5003	6C	l
5004	6F	o

33

Example: Direct Index (Scale 4)

```
# Talk = 5000

mov  rdi, 1
movb [Talk + rdi * 4], 33
```

5000	48	H
5001	65	e
5002	6C	l
5003	6C	l
5004	33	!

34

Example: Register Indirect

```
# Talk = 5000

lea  rax, Talk
movb [rax], 33
```

Indirect. Base is rax

5000	33	!
5001	65	e
5002	6C	l
5003	6C	l
5004	6F	o

35

Example: Register Indirect Index

```
# Talk = 5000

lea  rax, Talk
mov  rdi, 1
movb [rax + rdi], 33
```

Base Index

5000	48	H
5001	33	!
5002	6C	l
5003	6C	l
5004	6F	o

36

Ex: Register Indirect Index (Scale 2)

```
# Talk = 5000

lea  rax, Talk
mov  rsi, 1
movb [rax + rsi * 2], 33
```

Scale

5000	48	H
5001	65	e
5002	33	!
5003	6C	l
5004	6F	o

37

Ex: Register Indirect Index (Scale 4)

```
# Talk = 5000

lea  rax, Talk
mov  rsi, 1
movb [rax + rsi * 4], 33
```

5000	48	H
5001	65	e
5002	6C	l
5003	6C	l
5004	33	!

38

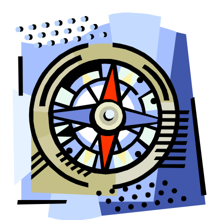
Addressing & Loops

They were made for each other … *literally*

39

Addressing & Loops

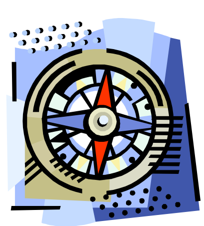
- When you use arrays in Java, often the index is a variable
- This allows you to use a For Loop to analyze very element in the array
- This is more common than you think in assembly



40

Addressing & Loops

- So, processors allow a register to be used as an index
- This allows you to:
  - copy strings (copying arrays)
  - search through a list
  - and much more…



41

For Loop: 0 to 4 - Before

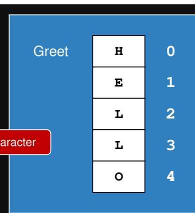
```
.intel_syntax noprefix
.data
Greet:
   .ascii "HELLO"

.text
.global Main

Main:
```

Greet		
	H	0
	E	1
	L	2
	L	3
	O	4

42

For Loop: 0 to 4

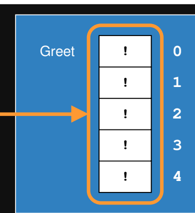
```
        mov   rax, 0

    Loop:
        cmp   rax, 4
        jg    End

        movb  [Greet + rax], 33
        add   rax, 1
        jmp   Loop
    End:
```



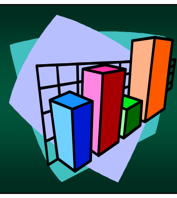
43

For Loop: 0 to 4 - After

```
        mov   rax, 0

    Loop:
        cmp   rax, 4
        jg    End

        movb  [Greet + rax], 33
        add   rax, 1
        jmp   Loop
    End:
```



44



Tables

How to Organize Data

45

Tables

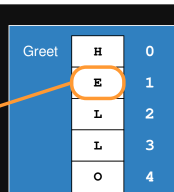
- In assembly, you have full control of memory
- You can take advantage of these to create tables
- They can contain any data – from integers, to characters, to addresses

46

Accessing Each element

Use register to hold table index

```
mov    rsi, 1
movb   ah,  [Greet + rsi]
```



47

Tables of Integers

- Tables can contain *anything!*
- Often, they are used to store integers & addresses (8 bytes on a 64-bit system)
- Just make sure to use the scale feature!

48

Table of Long Integers

```
Years:
    .quad 1776
    .quad 1783
    .quad 1846
    .quad 1850
    .quad 1947
```

8 Bytes each

49

Assuming Years is 6000

```
Years:
    .quad 1776
    .quad 1783
    .quad 1846
    .quad 1850
    .quad 1947
```

6000	1776
6008	1783
6016	1846
6024	1850
6032	1947

50

Assuming Years is 6000

Table index 1

```
mov  rsi, 1
mov  rcx, [Years + rsi * 8]
```

Note the scale!

6000	1776
6008	1783
6016	1846
6024	1850
6032	1947

51

Table of Addresses. Assume Names is 3000

```
Sutter:
    .ascii "John Sutter\0"

Marshal:
    .ascii "James Marshal\0"

Names:
    .quad Sutter
    .quad Marshal
```

3000	Sutter *(address)*
3008	Marshal *(address)*

52

Assuming Names is 3000

Note: mov is used. We want the data from the table (which is an address)

```
mov  rsi, 1
mov  rdx, [Names + rsi * 8]

call PrintString
```

3000	Sutter *(address)*
3008	Marshal *(address)*

53

Buffer Overflow

With Great Power
Comes Great Responsibility

54

Buffer Overflow

- Operating systems protect programs from having their memory / code damaged by *other* programs
- However…operating systems don't protect programs from damaging *themselves*

55

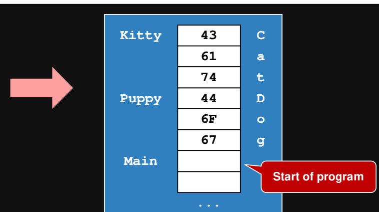
Buffers & Programs

- In memory, a running program's data is often stored next to its instructions
- This means…
  - if the end of a buffer of exceeded, the program can be read/written
  - this is a common hacker technique to modify a program *while it is running!*

56

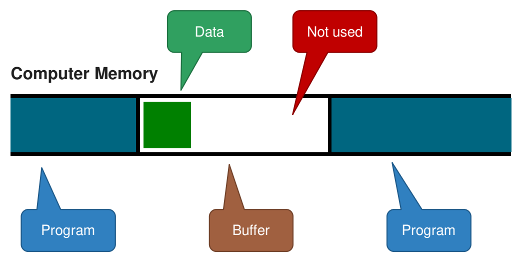
Example Program

```
.data
Kitty:
    .ascii "Cat"

Puppy:
    .ascii "Dog"

.text
.global Main
Main:
```



57

Buffer Overflow – How it Works



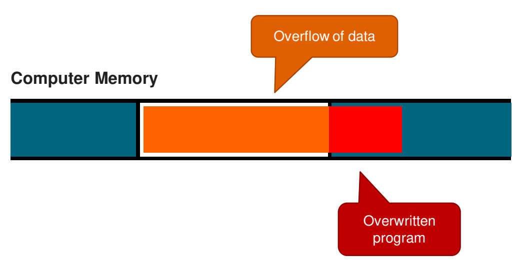
58

Buffer Overflow

- It is possible to store too much information – resulting in a *buffer overflow*
- The extra bytes will overwrite part of the running program – changing it!

59

Buffer Overflow – How it Works



60

Bad Indexing

- It is possible to accidentally change data stored in the different buffers
- In assembly, you have full control over your allocated memory
- *With great power comes great responsibility*

61

Wrong Buffer Changed

```
.intel_syntax noprefix
.data
Kitty:
   .ascii "Cat\0"
Puppy:
   .ascii "Dog\0"

.text
.global Main
Main:
    mov  rdi, 4
    movb [Kitty + rdi], 72
```

4 bytes. Character indexes from 0 to 3

72 is ASCII 'H' In hex it's 48

62

Wrong Buffer Changed

```
.intel_syntax noprefix
.data
Kitty:
   .ascii "Cat\0"
Puppy:
   .ascii "Dog\0"

.text
.global Main
Main:
    mov  rdi, 4
    movb [Kitty + rdi], 72
```

Kitty	43	C
	61	a
	74	t
	00	
Puppy	44	D
	6F	o
	67	g
	00	

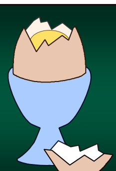
63

Wrong Buffer Changed

```
.intel_syntax noprefix
.data
Kitty:
   .ascii "Cat\0"
Puppy:
   .ascii "Dog\0"

.text
.global Main
Main:
    mov  rdi, 4
    movb [Kitty + rdi], 72
```

Kitty	43	C
	61	a
	74	t
	00	
Puppy	48	H
	6F	o
	67	g
	00	

64



Endianness

The "proper" order of things

65

So Many Bytes…

- On a 64-bit system, each word consists of 8 bytes
- So, when any 64-bit value is stored in memory, each of those 8 bytes must be stored
- However, question remains: *What order do we store them?*

66

Example Unsigned Integer (4 Byte)

1,188,852,977

46	DC	74	F1

Most significant Byte (MSB)

Least significant Byte (LSB)

67

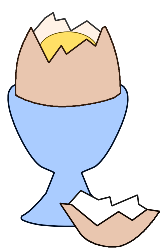
So Many Bytes…

- Do we store the least-significant byte (LSB) first, or the most-significant (MSB)?
- As long as a system always follows the same format, then there are no problems
- … but different system use different approaches

68

Big Endian vs. Little Endian

- Big-Endian approach
  - store the MSB first
  - used by Motorola & PowerPC
- Little-Endian approach
  - store the LSB first
  - used by Intel



69

Big Endian vs. Little Endian

46	DC	74	F1

Big Endian

0	46
1	DC
2	74
3	F1

Little Endian

0	F1
1	74
2	DC
3	46

70

Assuming Value is located at 2000

```
Value:
    .quad 74
```

2000	4A
2001	00
2002	00
2003	00
2004	00
2005	00
2006	00
2007	00

Least Significant Byte (LSB)

Little Endian

71

No "End" to Problems

- *There is a problem...* if two systems use different formats, data will be interpreted incorrectly!
- If how the read differs from how it is stored, the data will be mangled

72

No "End" to Problems

- For example:
  - a little-endian system reads a value stored in big-endian
  - a big-endian system reads a value stored in little-endian
- Programmers must be conscience of this whenever binary data is accessed

73

No "End" to Problems

- So, whenever data is read from secondary storage, you cannot assume it will be in your processor's format
- This is compounded by file formats (gif, jpeg, mp3, etc…) which are also inconsistent

74

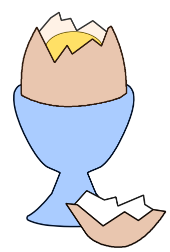
Example File Format Endianness

File Format	Endianness
Adobe Photoshop	Big Endian
Windows Bitmap (.bmp)	Little Endian
GIF	Little Endian
JPEG	Big Endian
MP4	Big Endian
ZIP file	Little Endian

75

So… who is correct?

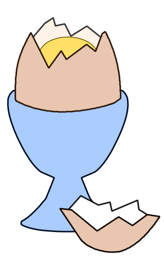
- So, what is the correct and superior format?
- Is it Intel (little endian)?
- …or the PowerPC (big endian) correct?



76

So… who is correct?

- In reality neither side is superior
- Both formats are equally correct
- Both have minor advantages in assembly… but nothing huge



77

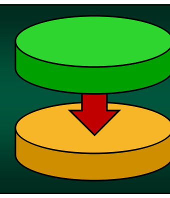
Gulliver's Travels

78

Subroutines & Operating Systems

Part 8

1



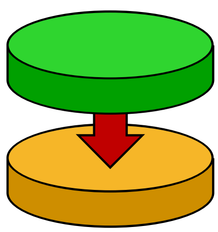
The System Stack

Pile of… Data

2

The System Stack

- The processor maintains a stack in memory
- It allows *subroutines*
  - analogous to the "functions" you use in Java and other third-generation languages
  - but, much more simple



3

Examples of Stacks

- Page-visited "back button" history in a web browser
- Undo sequence in a text editor
- Deck of cards in Windows Solitaire

4

Implementing in Memory

- On a processor, the stack stores integers
  - size of the integer the bit-size of the system
  - 64-bit system → 64-bit integer
- Stacks is stored in memory
  - A fixed location pointer (S0) defines the bottom of the stack
  - A *stack pointer* (SP) gives the location of the top of the stack

5

Approaches

- Growing upwards
  - Bottom Pointer (S0) is the *lowest* address in the stack buffer
  - stack grows towards *higher* addresses
- Grow downwards
  - Bottom Pointer (S0) is the *highest* address in the stack buffer
  - stack grows towards *lower* addresses

6

Size of the Stack

- As an abstract data structure…
  - stacks are assumed to be infinitely deep
  - so, an arbitrary amount of data can be stored
- However…
  - stacks are implemented using memory buffers
  - which are finite in size
- If the data exceeds the allocated space, a *stack overflow* error occurs

7

Subroutine Call Basics

Organizing Your Program

8

Subroutine Call

- The stack is essential for subroutines to work
- How?
  - used to save the return addresses for call instructions
  - backup and restore registers
  - pass data between subroutines

9

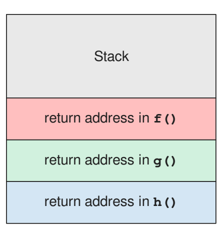
When you call a subroutine…

1. Processor pushes the instruction pointer (IP) – an address – on the stack
2. IP is set to the address of the subroutine
3. Subroutine executes and ends with a "return" instruction
4. Processor pops & restores the original IP
5. Execution continues after the initial call

10

Nesting is Possible

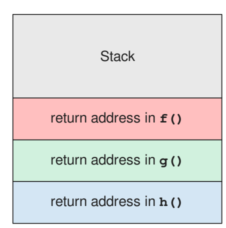
- Subroutines can call other subroutines
- `f()` calls `g()` which then calls `h()`, etc…
- The stack stores the return addresses of the callers
- Just like the "history button" in your web browser, you can store many return addresses



11

Nesting is Possible

- Each time a subroutine completes, the processor pops the top of the stack
- …then returns to the *caller*
- This allows normal function calls and recursion (a powerful tool)



12

x64 Subroutines

Organizing Your Programs … with Intel

13

Instruction: Call

- The *Call Instruction* transfers control to a subroutine
- Other processors call it different names such as JSR (Jump Subroutine)
- The stack is used to save the current IP

14

Instruction: Call

Usually, a label (which is an address)

```
CALL address
```


15

Instruction: Return

- The Return Instruction is used mark the end of subroutine
- When the instruction is executed…
  - the old instruction pointer is read from the system stack
  - the current instruction pointer is updated – restoring execution after the initial call

16

Instruction: Return

- Do not forget this!
- If you do…
  - execution will simply continue, in memory, until a return instruction is encountered
  - often is can run past the end of your program
  - …and run data!

17

Instruction: Return

No arguments!

```
RET
```


18

Subroutine Example

```
Main:
    mov  rcx, 4
    mov  rbx, 12
    call AddIt
    add  rbx, 1
    ...
AddIt:
    add  rbx, rcx
    ret
```


19

Saving Registers & Lost Data

Making subroutines clean

20

Saving Registers & Lost Data

- Each subroutine will use the registers as it needs
- So, when a subroutine is called, *it may modify the caller's registers*

21

Subroutine Example

```
StingersUp:
    .ascii "Stingers Up!\n\0"

 ...
Main:
    mov  rdx, 1947
    call Hornet
    call PrintInteger
```

We will create a subroutine called Hornet to print this

22

```
Main:
    mov  rdx, 1947
    call Hornet
    call PrintInteger
    ...

Hornet:
    lea  rdx, StingersUp
    call PrintString
    ret
```

We want to print 1947 after "Stingers Up!"

Hornet is called before PrintInteger

But, rdx is changed here

23

Subroutine Wrong Output

```
Stingers Up!
4202698
```

That's not 1947. The subroutine changed rdx to an address.

24

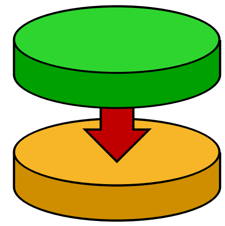
Saving Registers & Lost Data

- Some processors have few registers – this is very likely
- This can lead to hard-to-fix bugs if caution is not used – *e.g. subroutine changes the callers loop counter*

25

Use the Stack!

- Subroutine saves registers it will change
- How? Registers are pushed onto the state at the beginning of the subroutine
- Before it returns, it pops (and restores) the old values



26

Saving Registers… How nice! :-)

```
DoSomething:
    push rax
    push rbx
    push rcx

    ...

    pop rcx
    pop rbx
    pop rax
    ret
```

Backup registers

Your code

Restore them.
Note the reverse order

27

Fixed Subroutine

```
Hornet:
     push rdx

     lea  rdx, StingersUp
     call PrintString

     pop rdx
     ret
```

Push the value on the stack
`stack.push(rdx)`

Restore the value
`rdx = stack.pop()`

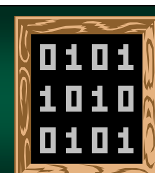
28

Subroutine Correct Output

```
Stingers Up!
1947
```

The subroutine worked perfectly

29



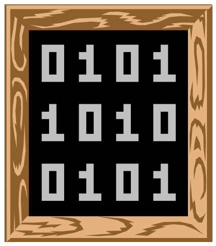
Stack Frames

The Jenga of data!

30

Stack Frames

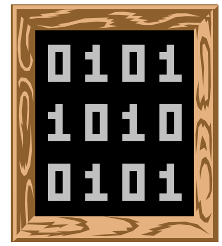
- *Stack Frames* is an compiler approach where subroutine arguments are passed using the system stack
- The stack is also used to store local variables



31

Why is this needed?

1. Need to support **any** number of parameters – *even if it exceeds the available number of registers*
2. Need support local variables



32

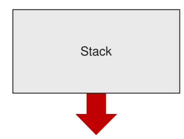
Stack Frame Contents

- Contains all the information needed by subroutine
- Includes:
  - calling program's return address
  - input parameters to the subroutine
  - the subroutine's local variables
  - space to backup the caller's register file

33

Nesting is Possible

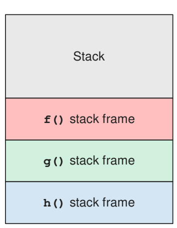
- Stack is LIFO (last in first out), so subroutines can call subroutines
- This approach allows recursion and all the features found in high-level programming languages



34

Nesting is Possible

- For example, subroutine `f()` calls `g()` which then calls `h()`
- Since the stack is used, the stack frames follow the LIFO behavior



35

How it Starts Up

- Caller
  - pushes the subroutine's arguments onto the stack
  - caller calls the subroutine
- Subroutine then…
  - uses the stack to backup registers
  - and *"carve"* out local variables

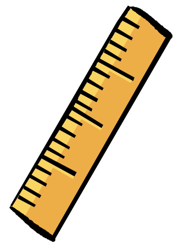
36

How it Finishes

- Subroutine…
  - restores the original register values
  - removes the local variables from the stack
  - calls the processor "return" instruction
- Caller, then…
  - removes its arguments from the stack
  - handles the result – which can be passed either in a register or on the stack

37

Stack Frame Size Varies

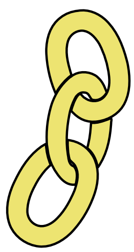


- The number of input arguments and local variables varies from subroutine to subroutine
- The arrangement of data within the stack frame also varies from compiler to compiler
- Stack frames is a *concept* – and it is used with various differences

38

What About Different Object Files?

- Programs are often created from multiple object files
- These can be created by different compilers and linked separately –
- So, how do we make sure that these are all compatible?



39

Calling Convention

- A *calling convention* is defined by a programming system (e.g. a language) to define *how* data will be passed
- In particular, it defines the structure of the stack frame and how data is returned

40

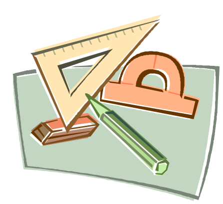
Calling convention

- For example:
  - Is the first argument pushed first? Or last?
  - Is the result in a register? Or the stack?
- If all subroutines follow the same format
  - caller can use the same format for each
  - subroutines can also be created separately and linked together

41

Compatibility

- If two different compilers use the same calling convention, the resulting object files will be compatible
- This means, large applications can be created in different programming languages



42

Stack Frames on the x64

Pretty much how its done on all processors

43

Stack Frames on the x64

- Stack frames on the x64 are accomplished pretty much the same way as other processors
- How it is done in real life is not simple – and is one of the hardest concepts to understand

44

Stack Frames on the x64

- On the x64, we will use the Base Pointer (RBP) to access elements in the stack frame
- This is a pointer register
- We will use it as an "anchor" in our stack frame

45

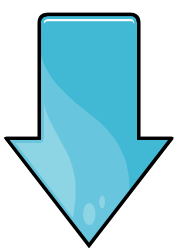
Stack Frames on the x64

- As we build the stack frame, we will set RBP to fixed address in the stack frame
- Our parameters and local variables will be accessed by looking at memory *relative* to the RBP
- So, we will look *x* many bytes above and below the "anchor"

46

Stack on the x64

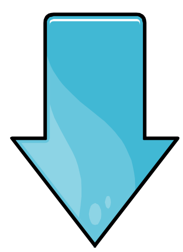
- The stack base on the x64 is stored in high memory and grows downwards towards 0
- So, as the size of the stack increases, the stack pointer (RSP) will decrease in value



47

Stack on the x64

- On a 64-bit system, it will decrease by increments of 8 bytes
- So, each of our values (local variables and parameters) will be offsets of 8



48

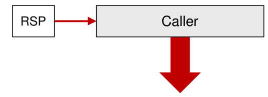
Stack Frame Start Steps

1. Caller pushes arguments
2. Call the subroutine
3. Backup (push) the Base Pointer
4. Set the Base Pointer
5. "Carve" Local Variables
6. Backup (push) local variables

49

Structure of a Stack Frame

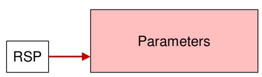
- In this example, the stack is growing downward
- The stack pointer (RSP) is always on the bottom of the stack frame



50

1. Push Parameters

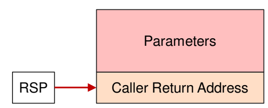
- Caller starts by pushing each of the parameters onto the stack
- Order parameters are pushed is defined in the *calling convention*



51

2. Call the subroutine

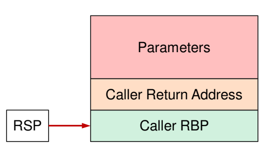
- The caller then uses the *Call Instruction* to pass control to the subroutine
- The processor pushes the IP (instruction pointer) on the stack
- Subroutine now runs



52

3. Backup the Old Base Pointer

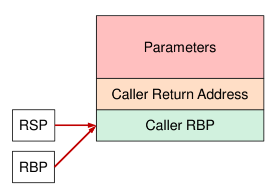
- We need to set the Base Pointer (RBP)
- So, the old version needs to be saved (so it can be restored)
- Old RBP is pushed



53

4. Set the Base Pointer

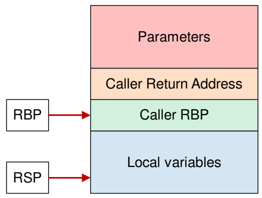
- Then, it sets the Base Pointer (RBP) to the current stack pointer (RSP) address
- RBP is an "anchor" that we will use



54

5. "Carve" Local Variables

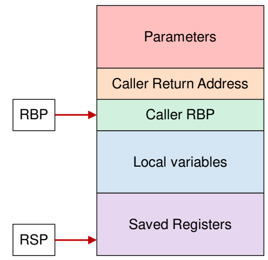
- Subroutine now creates local variables on the stack
- Their initial values can be simply pushed



55

6. Backup Registers

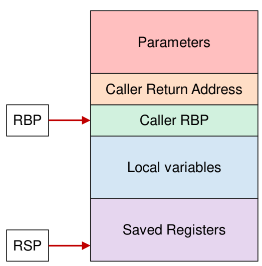
- Finally, the subroutine saves all the registers (that will change) on the stack
- It will restore them at the end



56

The Completed Stack Frame

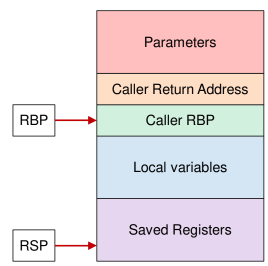
- The Stack Frame contains all the information a subroutine needs
- We just need to be able to access the data programmatically



57

Parameters & Local Variables

- Now, RBP is set to an address between the parameters and the local variables
- We can use *offsets* from RBP to access each



58

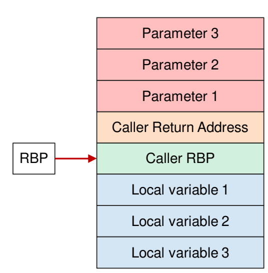
Compiler Friendly… Not Human

- The offset values have to count the bytes from the RBP register
- These are not easy to read since both parameters and local variables are just offsets

59

Size of Stack Values – 64 bit

- x64 stack grows downward, so…
- Local variables will have a negative offset
- Parameters will have a positive offset



60

Size of Stack Values – 32 bit

- On a 32-bit system, each word is 4 bytes
- So, each value on the stack is 4 bytes
- Offsets increase and decrease by 4

+16	Parameter 3
+12	Parameter 2
+8	Parameter 1
+4	Caller Return Address
RBP →	Caller RBP
−4	Local variable 1
−8	Local variable 2
−12	Local variable 3

61

Size of Stack Values – 64 bit

- On a 64-bit system, each word is 8 bytes
- So, each value on the stack is 8 bytes
- Offsets increase and decrease by 8

+32	Parameter 3
+24	Parameter 2
+16	Parameter 1
+8	Caller Return Address
RBP →	Caller RBP
−8	Local variable 1
−16	Local variable 2
−24	Local variable 3

62

Caller Example

```
push  42
push  rax
call  Subroutine
```

1. Push parameters (16 bytes added)

2. Call subroutine

63

Subroutine: Setup Example

```
push  rbp
mov   rbp, rsp
push  1
push  2
push  rax
push  rbx
```

3. Backup RBP

3. Set new RBP

4. Local variables

5. Backup registers

64

Subroutine Data Example

Offset from pointer

```
mov  rax, [rbp + 16]
add  rax, [rbp + 24]

mov  [rbp – 8], rax
```

Parameter 1

Parameter 2

Local variable 1

65

Stack Frame Ending Steps

1. Restore registers (pop)
2. Restore the stack pointer (set it to the base pointer)
3. Restore the old base pointer
4. Return and delete parameters

66

Stack Frame Return

- The Return can also be used to clean up the caller's stack items
- You can specify the number of bytes to pop (and discard) after the return
- Alternatively, the caller can clean up the stack

67

Stack Frame Return

```
RET  byteCount
```

Bytes to discard after return

68

Subroutine: Ending Example

```
pop   rbx
pop   rax

mov   rsp, rbp
pop   rbp
ret   16
```

1. Restore registers (pop in reverse order)

2. Set RSP to RBP
Effectively deletes all local variables

3. Restore RBP

4 Return after the stack is restored

69

```
MySub:
    push  rbp
    mov   rbp, rsp
    push  1
    push  2
    push  rax
    push  rbx

    ...

    pop   rbx
    pop   rax
    mov   rsp, rbp
    pop   rbp
    ret   16
```

Setup base pointer

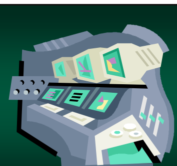
Local variables (with initial values)

backup registers

Restore registers

Restore base pointer

70



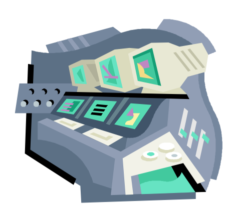
Operating Systems

The master software

71

What is an operating system?

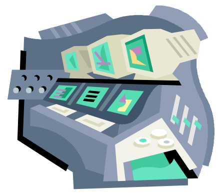
- The operating system is simply a series of programs
- These programs, however, run with special privileges which are needed by the OS
- Processors support two modes for executing programs



72

Execution Modes

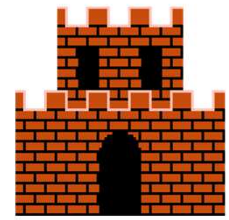
- *Privileged (supervisor) mode*
  - can run special instructions
  - can talk to all the hardware
  - etc…
- *User mode*
  - can only execute certain instructions
  - can't talk to all the hardware



73

Vector Tables

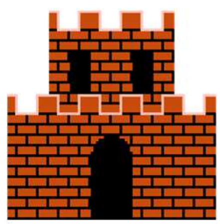
- Programs (and hardware) often need to talk to the operating system
- Examples:
  - software needs talk to the OS
  - USB port notifies the OS that a device was plugged in



74

Vector Tables

- But how does this happen?
- The processor can be *interrupted* – alerted – that something must be handled
- It then runs a special program that handles the event



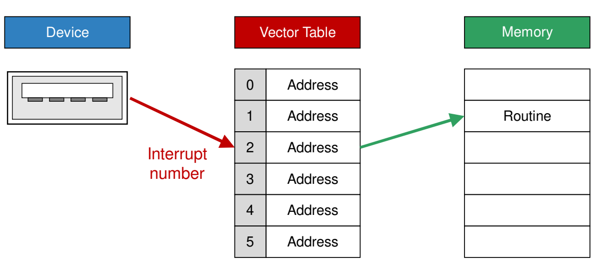
75

Vector Table

- During an interrupt, the device sends the processor an *interrupt number*
- The processor looks up the number in the *vector table*
- Table contains the address of *Interrupt Service Routine (ISR)* to execute

76

How It Works



77

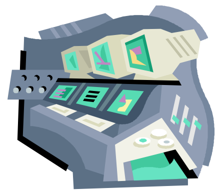
What Happens…

1. Device interrupts the processor
2. Current program state is saved
   - register file
   - instruction pointer
3. Processor executes ISR using the Vector Table address
4. Current program state is restored

78

The Kernal

- All these Interrupt Service Routines belong to the *kernal* – the core of the operating system
- Vast majority of the operating system is hidden from the end user



79

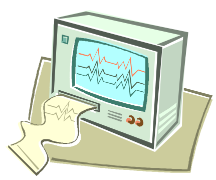
Interact with Applications

How do WE talk to the OS

80

Interact with Applications

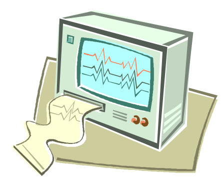
- Software also needs to talk to the operating system
- For example:
  - draw a button
  - print a document
  - close this program
  - etc…



81

Interact with Applications

- Software can interrupt itself with a specific number
- This interrupt is *designated specifically for software*
- The operating system then handles the software's request



82

Application Program Interface

- Programs "talk" to the OS using *Application Program Interface (API)*
- Application → Operating System → IO
- Benefits:
  - makes applications faster and smaller
  - also makes the system more secure since apps do not directly talk to IO

83

Instruction: syscall (64-bit)

```
SYSCALL
```

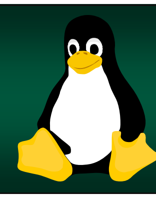
Calls interrupt number reserved for programs needing attention

84

Subroutine vs. Interrupt

Subroutine	Interrupt
Executes code	Executes code
Returns when complete	Returns when complete
Called by the application	Executed by the processor
Part of the application	Handles events for the OS

85



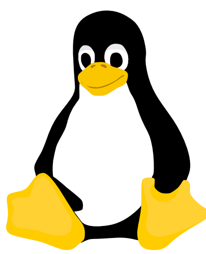
Linux System Calls

How software and hardware "talk"

86

Interrupts on the Linux

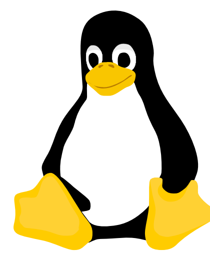
- Linux, like other operating systems communicate with applications using *interrupts*
- Applications do not know where (in memory) to contact the kernal – so they ask the processor to do it



87

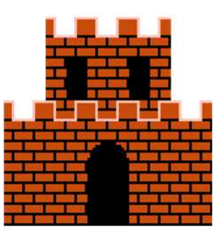
How It Works

1. Fill the registers
2. Interrupt using syscall (or INT 0x80 if on 32-bit)
3. Any results will be stored in the registers



88

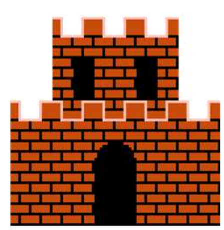
How to Call Linux – 64 bit



- The **rax** register must contain the *system call number*
- This number indicates what you asking the OS to do
- There are only 329 total calls in the entire 64-bit UNIX operating system!

89

How to Call Linux – 64 bit

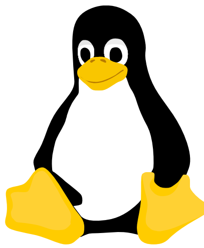


- Different registers are used to hold data
- The order is also quite odd: **rdi**, **rsi**, **rdx**, **r10**, **r8**

90

Kernals are Simple!

- Linux only has **1** write and **1** read system call
- The location, number of bytes, and device only change *"write x many bytes from address y to device z"*
- So, writing to the screen, a file, a port, etc…use the same call!



91

Some Linux 64 Calls

System Call	rax	rdi	rsi	rdx
read	0	file descriptor	address	max bytes
write	1	file descriptor	address	count
open	2	address	flags	mode
close	3	file descriptor		
get pid	39			
exit	60	error code		

92

Linux 64: Sys Write

```
mov  rax, 1
mov  rdi, 1
lea  rsi, address
mov  rdx, length
syscall
```

Linux command for WRITE

1 = Screen

Call Linux

93

Linux 64: Sys Read

```
mov  rax, 0
mov  rdi, 0
lea  rsi, address
mov  rdx, maxBytes
syscall
```

Linux command for READ

0 = Keyboard

Maximum number of bytes to read

Call Linux

94

Write Example

```
SacState:
   .ascii "Stinger's up!\n"    #\n counts as 1 character
...
   mov  rax, 1              #1 = write
   mov  rdi, 1              #1 = screen
   lea  rsi, SacState
   mov  rdx, 14             #14 bytes
   syscall
```


95

Design Principles

Part 9

1

von Neumann Architecture

The Information Superhighway

2

von Neumann Machine Architecture

- Modern computers are based on the design of John von Neumann
- His design greatly simplified the construction of (and use) computers

3

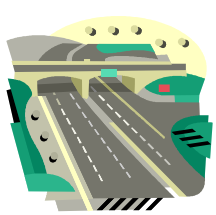
Some von Neumann Attributes

1. Programs are stored and executed in memory
2. Separation of processing from memory
3. Different system components communicate over a shared bus

4

The Bus

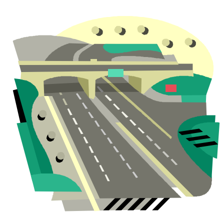
- Electronic pathway that transports data between components
- Think of it as a "highway"
  - data moves on shared paths
  - otherwise, the computer would be very complex



5

System Bus

- The information sent on the memory bus falls into 3 categories
- Three sets of signals
  - address bus
  - data bus
  - control bus



6

Address Bus

- Used by the processor to access a specific piece of data
- This "address" can be
  - a specific byte in memory
  - unique IO port
  - etc…
- The more bits it has, the more memory can be accessed

7

Address Bus Size Examples

- 8-bit → 2^{8} = 256 bytes
- 16-bit → 2^{16} = 64 KB (65,536 bytes)
- 32-bit → 2^{32} = 4 GB (4,294,967,296 bytes)
- 64-bit → 2^{64} = 18 EB (18,446,744,073,709,551,616)

8

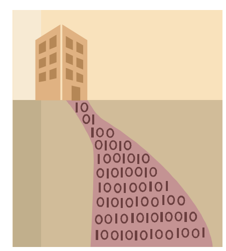
Historic Address Sizes

- Intel 8086
  - original 1982 IBM PC
  - 20-bit address bus (1 MB)
  - only 640 KB usable for programs
- MOS 6502 computers
  - Commodore 64, Apple II, Nintendo, etc…
  - 16-bit address bus (64 KB)

9

Data Bus

- The actual data travels over the *data bus*
- The number of bits that the processor uses – as its natural unit of data – is called a *word*



10

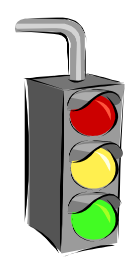
Data Bus

- Typically we define a system by word size
- Example:
  - 8-bit system uses 8 bit words
  - 16-bit system uses 16 bits (2 bytes) words
  - 32-bit system uses 32 bits (4 bytes) words
  - etc…

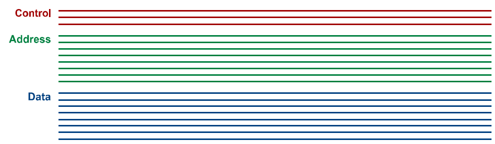
11

Control Bus

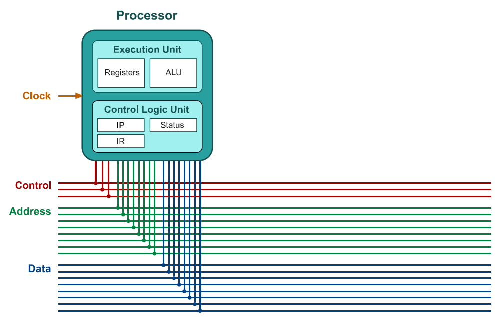
- The *control bus* controls the timing and synchronizes the subsystems
- Specifies what is happening
  - read data
  - write data
  - reset
  - etc…



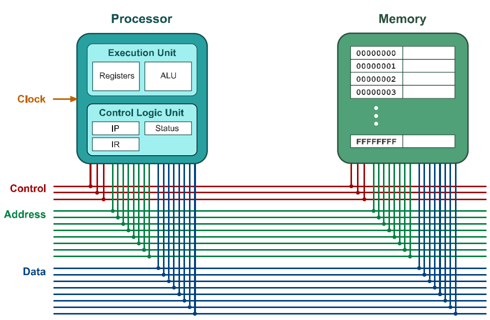
12



13



14



15

von Neumann Architecture Today

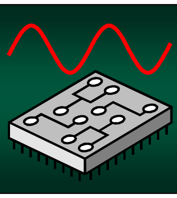
- Because of the emphasis on memory, most real-world systems use a modified version of his design
- In particular, they have a special high-speed bus between the processor and memory

16

von Neumann Architecture Today

- Think of it as a diamond-lane on a freeway
- … or as high-speed rail – which has a fixed source and destination and goes faster than the freeway

17



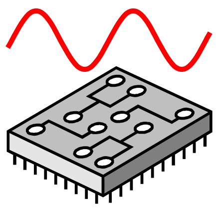
The System Clock

Tick-tock tick-tock tick-tock

18

The System Clock

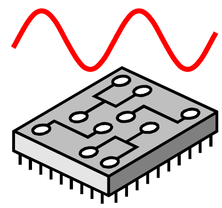
- The rate in which instructions are executed is controlled by the CPU clock
- The faster the clock rate, the faster instructions will be executed
- Measured in Hertz – number of oscillations per second



19

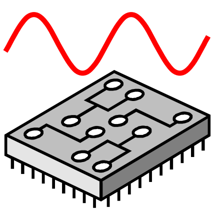
The Clock

- Computers are typically (and generically) labeled on the processor clock rate
- In the early 80's it was about 1 Megahertz – million clocks per second
- Now, it is terms of Gigahertz – billion clocks per second



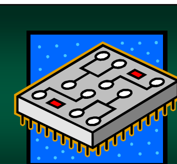
20

Clock and Instructions



- Not all instructions are "equal"
- Some require multiple clock cycles to execute
- For example:
  - add can take a single clock
  - but floating-point math could require a dozen

21



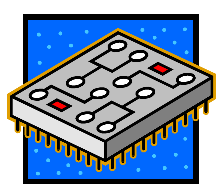
Technological Trends

Change is fast

22

Technological Trends

- Since the design of the integrated circuit, computers have advanced *dramatically*
- Home computer's today have more power than mainframes did 30 years ago
- A hand calculator has more power than the computer that took us to the Moon



23

Integrated Circuits Improved In…

- *Density* – total number transistors and wires can be placed in a fixed area on a silicon chip
- *Speed* – how quickly basic logic gates and memory devices operate
- *Area* – the physical size of the largest integrated circuit that can be fabricated

24

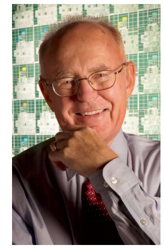
Rate of Improvement

- The increase in performance does not increase at a linear rate
- Speed & Density improves *exponentially*
  - from one year to the next… it has been a relatively constant fraction of the previous year's performance
  - …rather than constant absolute value

25

Moore's Law

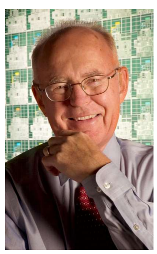
- Gordon Moore is one of the co-founders of Intel
- He first observed (and predicted) computer performance improves *exponentially*, not linearly



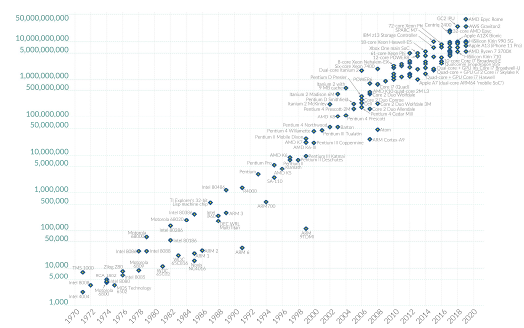
26

Moore's Law

- *Moore's Law* states the performance doubles every 18 months
- This law has held for nearly 50 years



27



28

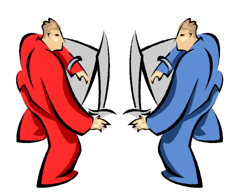
CISC vs. RISC

How do we tip the scales?

29

CISC vs. RISC

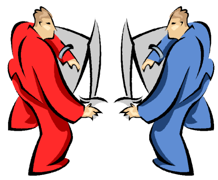
- There is, an often contentious, debate on how to design a processor
- For instance:
  - how is memory going to be accessed
  - what instructions are needed
  - how to encode/structure them



30

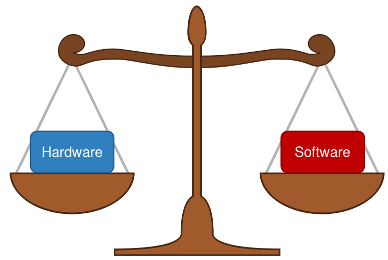
CISC vs. RISC

- Typically, the debate comes down to CISC vs. RISC
- Processors are typically put into these two categories
- Rarely is a processor "pure" RISC or CISC
- It's a *design philosophy* with a large "gray" area



31

Hardware vs. Software



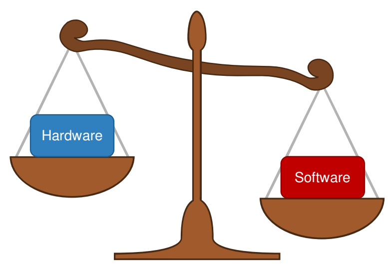
32

RISC

- Reduced Instruction Set Computer (RISC) emphasizes hardware simplicity
- Software should contain the complexity rather than hardware

33

RISC – Simple Hardware



34

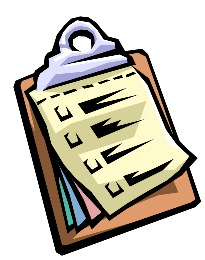
RISC

- So, RISC contains fewer instructions than CISC – only the minimum needed to work
- Minimalize memory access
  - only a few instructions can access memory
  - usually limited to register load and store instructions

35

RISC Characteristics

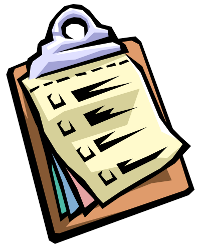
- Access to memory is restricted to load/store instructions
- Many registers – since all instructions can only use registers for calculations



36

RISC Characteristics

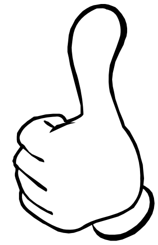
- Instructions typically take one clock cycle each
- The number of bytes, used by instructions, tend to fixed in size (all 32-bit, for example)



37

RISC Advantages

- Simpler instructions simplify hardware - makes processors easier to manufacture
- Also, produces less heat and requires less energy



38

RISC Advantages

- Fewer instructions means there is less to learn and master
- Memory access is minimalized

39

Example RISC Processors

- ARM
  - dominant processor used by smartphones
  - designed to reduce transistors
  - less cost, less heat, less power
- IBM PowerPC 601
  - developed in by IBM, Apple, and Motorola (AIM)
  - used by 1990's Macs

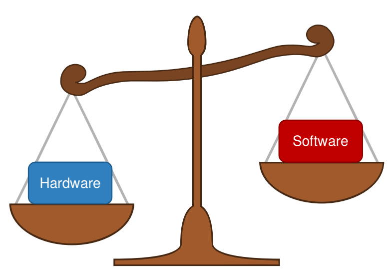
40

CISC

- Complex Instruction Set Computer (CISC) emphasizes flexibility in instructions
- Hardware should contain the complexity rather than the software

41

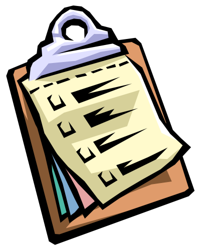
CISC – Hardware is more complex



42

CISC Characteristics

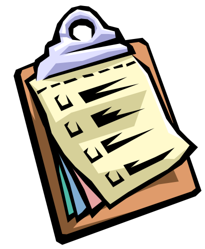
- Instructions can take multiple clocks – depending on how complex
- Operands are *generalized* – each can access memory, immediates or registers



43

CISC Characteristics

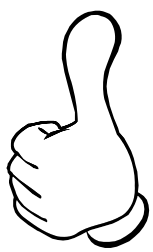
- Very few general-purpose registers
- The number of bytes, used by instructions, tend to vary in sizes



44

CISC Advantages

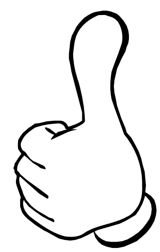
- Generally, requires fewer instructions than RISC to perform the same computation
- Software is easier to write given the flexibility



45

CISC Advantages

- Programs written for CISC architectures tend to take less space in memory
- Variable-sized instructions can make it possible for the processor "evolve" – i.e. add new instructions



46

Example CISC Processors

- Intel x86
  - evolved from the 8088 processor and contains 8-bit, 16-bit, and 32-bit instructions
  - dominant processor for PCs
- Motorola 68000
  - used in many 80's computers
  - …including the first Macintosh

47

Example CISC Processors

- VAX
  - contained even more addressing modes than we will cover
  - specialized instructions – even case blocks!
  - supported data types beyond float and int: variable-length strings, variable-length bit fields, etc…

48

RISC vs. CISC Comparison

CISC	RISC
Simple software, complex hardware	Simple hardware, complex software
Most operands can access memory	Load/Store instructions can access memory
Low number of registers	Higher number of registers
Instructions can have multiple clock cycles	Instructions tend towards one per clock cycle
Encoded instructions vary in size	Encoded instructions are all the same size

49

CISC Example (not x86)

```
# n  =  a * (b + c) - d

mov   R1, b
add   R1, c        # b + c

mul   R1, a        # a * (b + c)
sub   R1, d        # a * (b + c) - d

mov   n, R1
```

Note that ADD both loads & adds. This is 2 operations.

These too!

50

RISC Example (not x86)

```
# n  =  a * (b + c) - d

load   R1, b
load   R2, c
add    R2, R1      # b + c
load   R3, a
mul    R2, R3      # a * (b + c)
load   R4, d
sub    R2, R4      # a * (b + c) - d
store  n, R2
```

Note that all instructions (besides load & store) have two registers as operands.

51

Moore's Law & CISC

The pendulum swings

52

Moore's Law & CISC

- In the early 80's (the beginning of the Computer Revolution), memory was *expensive*
- In fact, it was the most expensive part of a new computer

53

Moore's Law & CISC

- Memory also ran at the same speed as the processor
- So, CISC was at a clear advantage – programs didn't take up as much memory as RISC

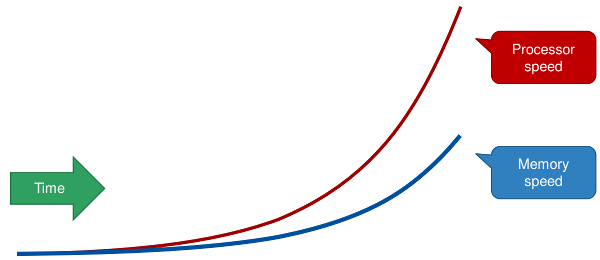
54

Moore's Law & CISC

- Computer speed through the 1980's grew exponentially
- However, …
  - rate of processor growth has been far greater than memory
  - so, memory *relative to the processor's speed* has gotten much slower

55

Moore's Law During the 80's



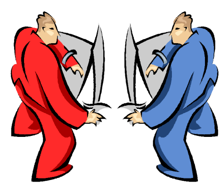
56

Memory is the Bottleneck

- CISC can access memory with nearly every instruction
- Memory is slow compared to register-to-register operations
- It is far more efficient (now) to do all work on the processor and use memory **only** when absolutely necessary

57

Latest Approach



- After the 1990s, RISC architectures have incorporated some of most useful complex instructions from CISC architectures
- Rely on micro-architecture to implement these instructions with little impact on the clock

58

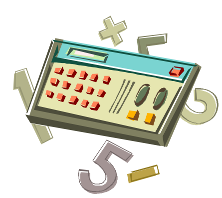
Instruction Operands

How much data does each need?

59

Instruction Operand

- The number of operands used in an instruction varies greatly by processor
- More operands give greater functionality, but require more bits to store in memory
- Typically processors contain 1, 2 or 3 operands



60

Single Operand Processors

- Single operand processors are also known as *accumulators*
- Operates similar to your hand calculator
- The accumulator register
  - used for all mathematical computations
  - other registers simply are used to compare and hold temporary data
- Examples: MOS 6502

61

Single Operand Instruction (CISC)

```
# z = 50 – (x + y)

lda x
add y          # x + y
sta temp
lda 50
sub temp       # 50 – temp

sta z
```

lda = Load accumulator

sta = store accumulator

62

Two Operand Processors

- Allows two operands to be specified
- For computations, both operands are typically treated as input, and one is used to store the result
- Examples:
  - x86 processors
  - PowerPC

63

Two Operand Instruction (CISC)

```
#  z  =  50 – (x + y)

mov   R1, x
add   R1, y     # x + y

mov   R2, 50
sub   R2, R1    # 50 – R1

mov   z, R2
```


64

Three Operand Processors

- Allows two input values like before, but also can specify a third output operand
- The third operand can also be used as a index for simple addressing
- Examples:
  - ARM
  - Intel Itanium

65

Three Operand Instruction (RISC)

```
#  z  =  50 – (x + y)

load  R1, x
load  R2, y
add   R3, R1, R2   # x + y
load  R4, 50
sub   R5, R4, R3
store z, R5
```


66

Three Operand Instruction (CISC)

```
#   z  =  50 - (x + y)

add  R1, x, y         # x + y
sub  z, 50, R1
```

Best of both worlds!

67