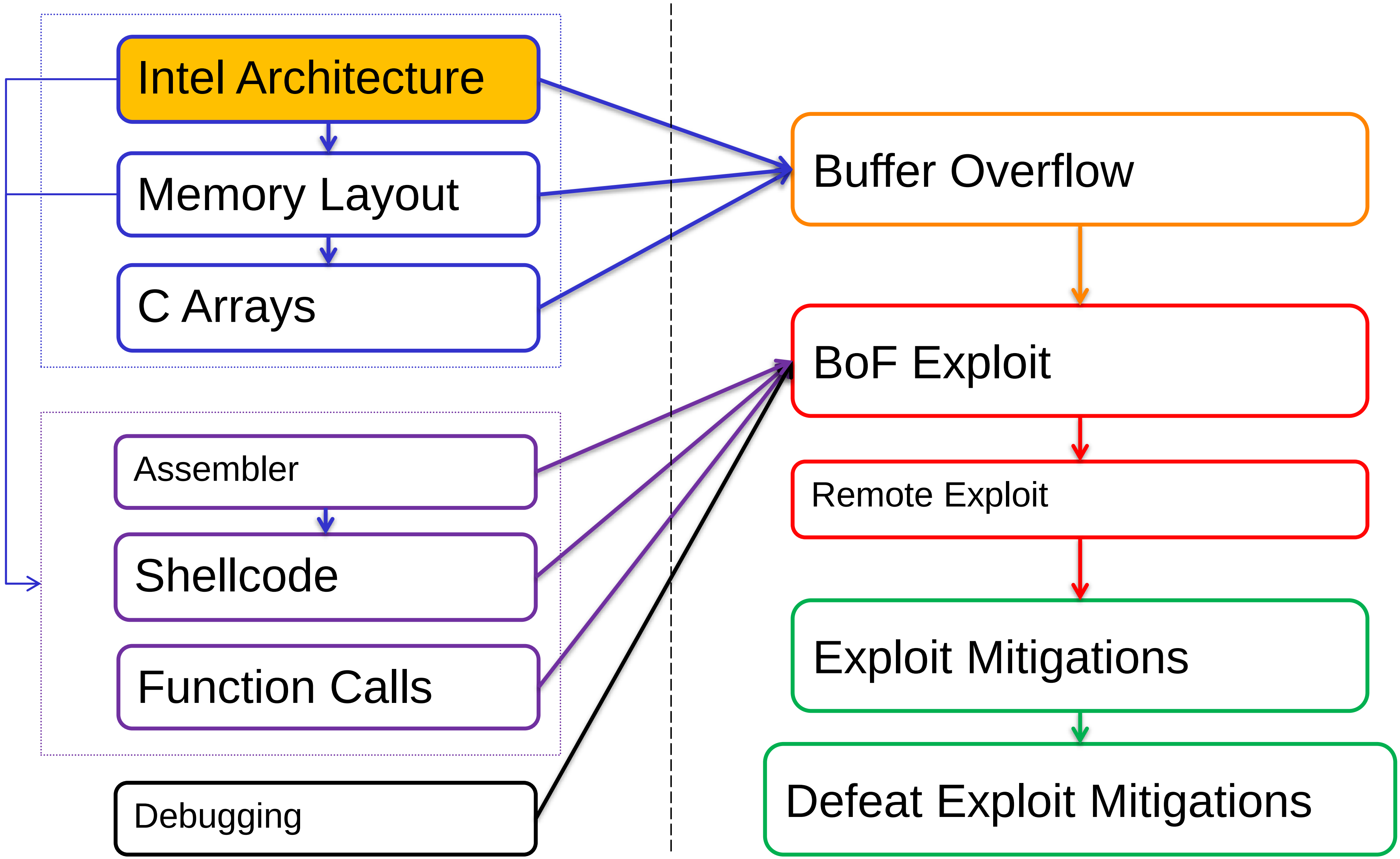
Intel Architecture

Content

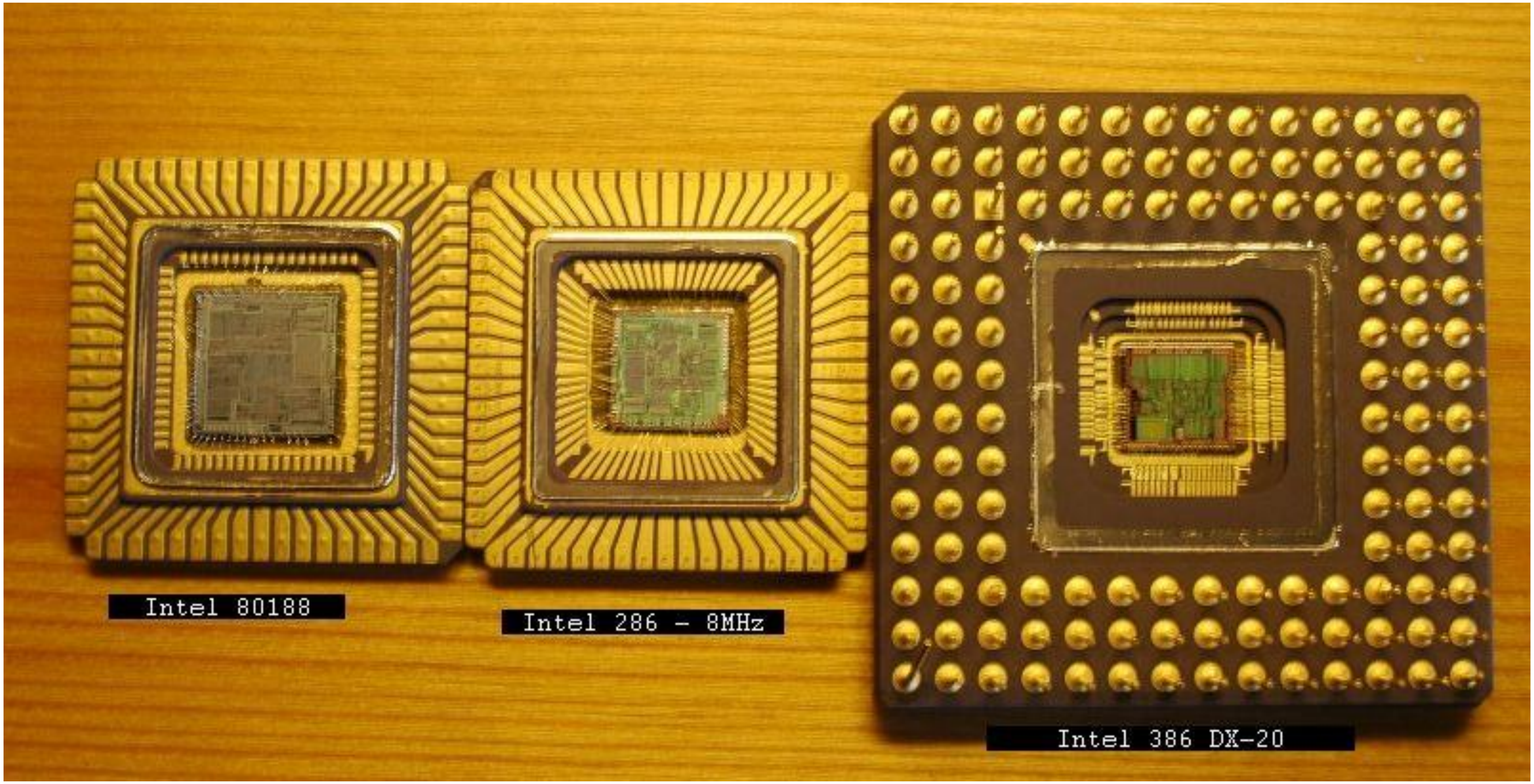

Intel Architecture
Memory Layout
C Arrays
Assembler
Shellcode
Function Calls
Debugging
Buffer Overflow
BoF Exploit
Remote Exploit
Exploit Mitigations
Defeat Exploit Mitigations

Intel Architecture

Intel CPU

Intel CPU

Intel CPU

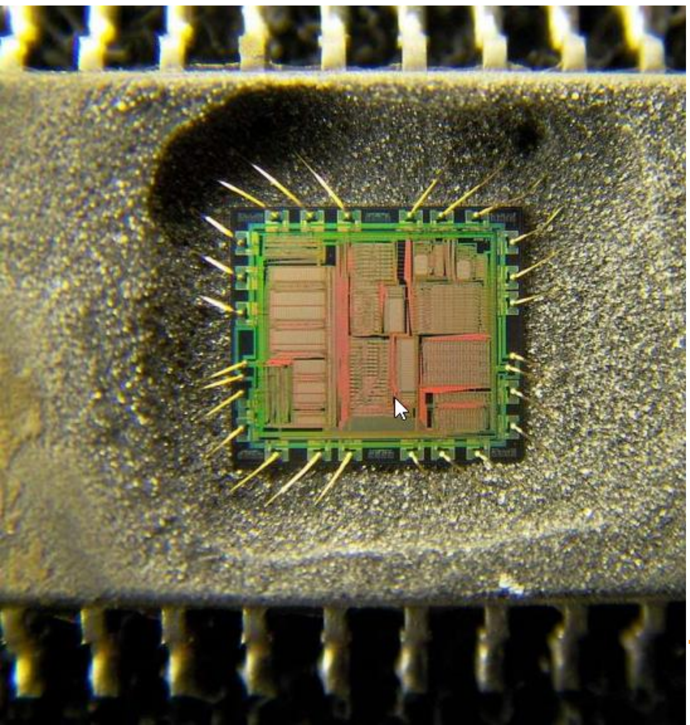


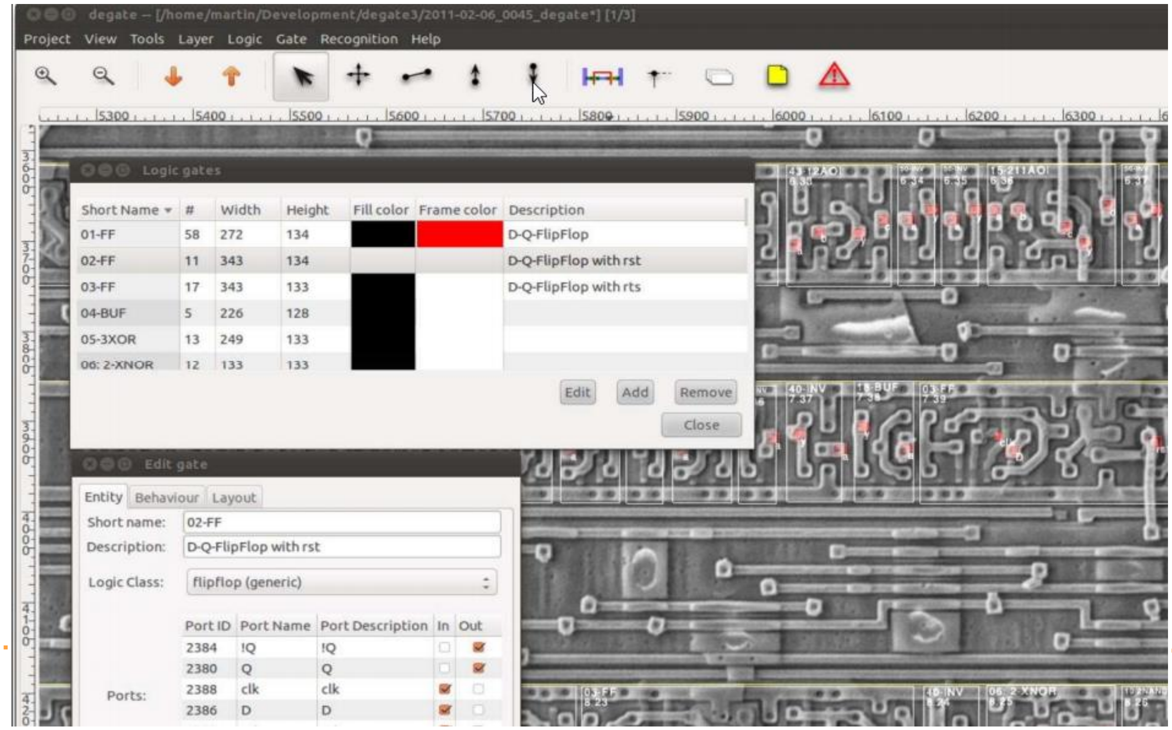
Intermezzo: Chip De-caping and Analysis

http://www.bluehatil.com/files/Extracting%20Secrets%20from%20Silicon%20%E2%80%93%20A%20New%20Generation%20of%20Bug%20Hunting.pdf

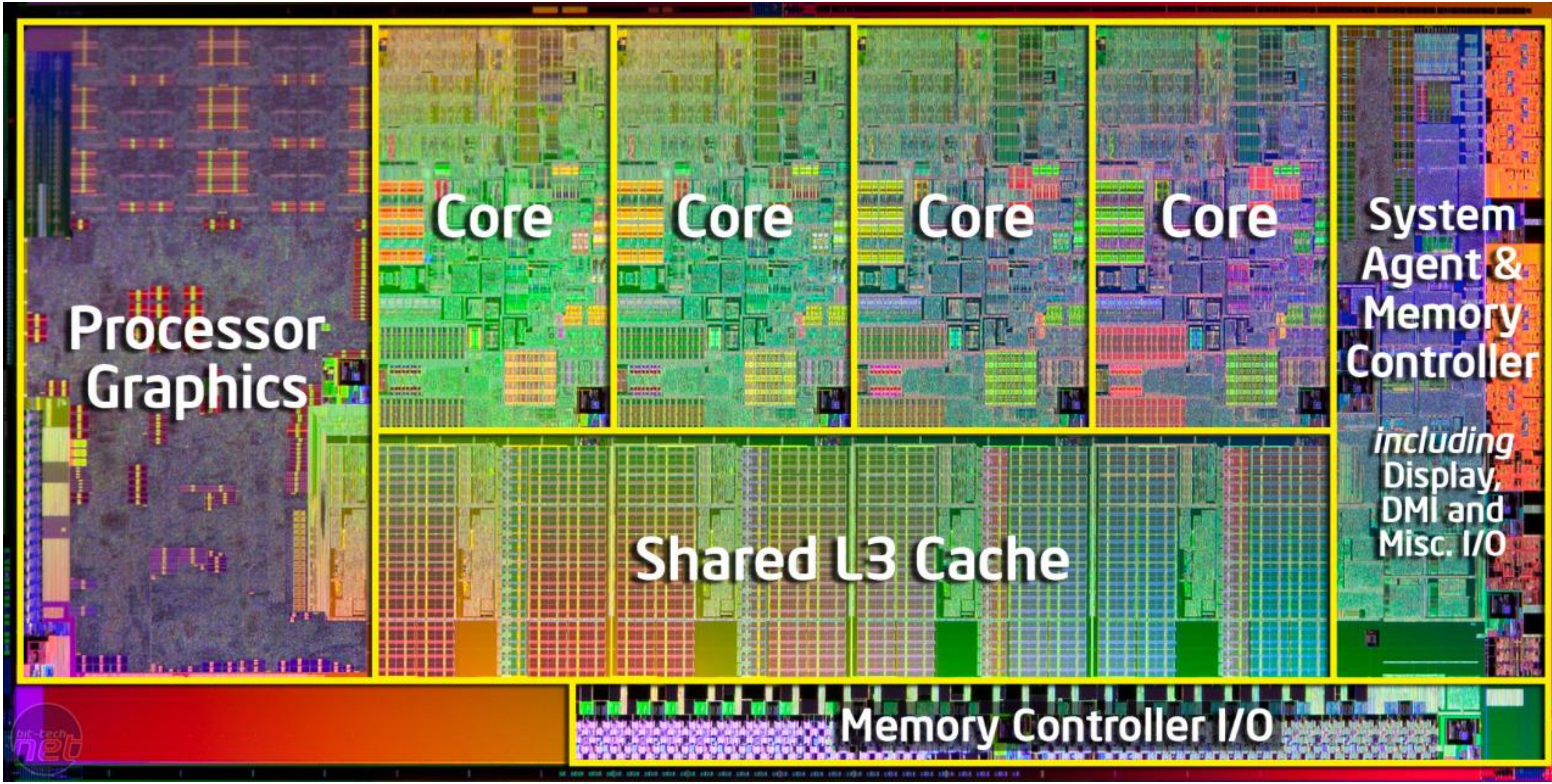
Extracting Secrets from Silicon – A New Generation of Bug Hunting

- Gunter Ollmann, Microsoft
- Blue Hat Security



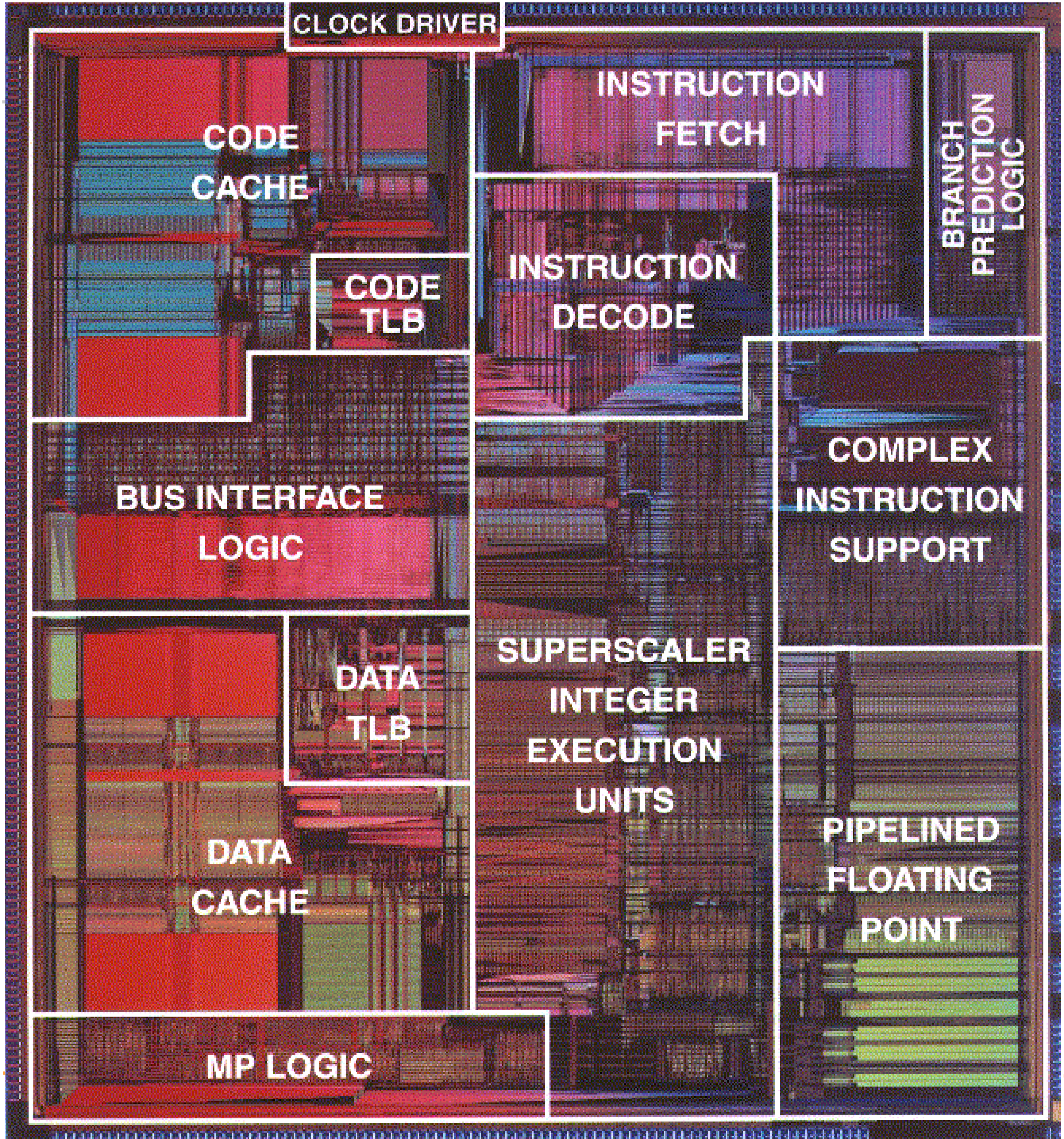


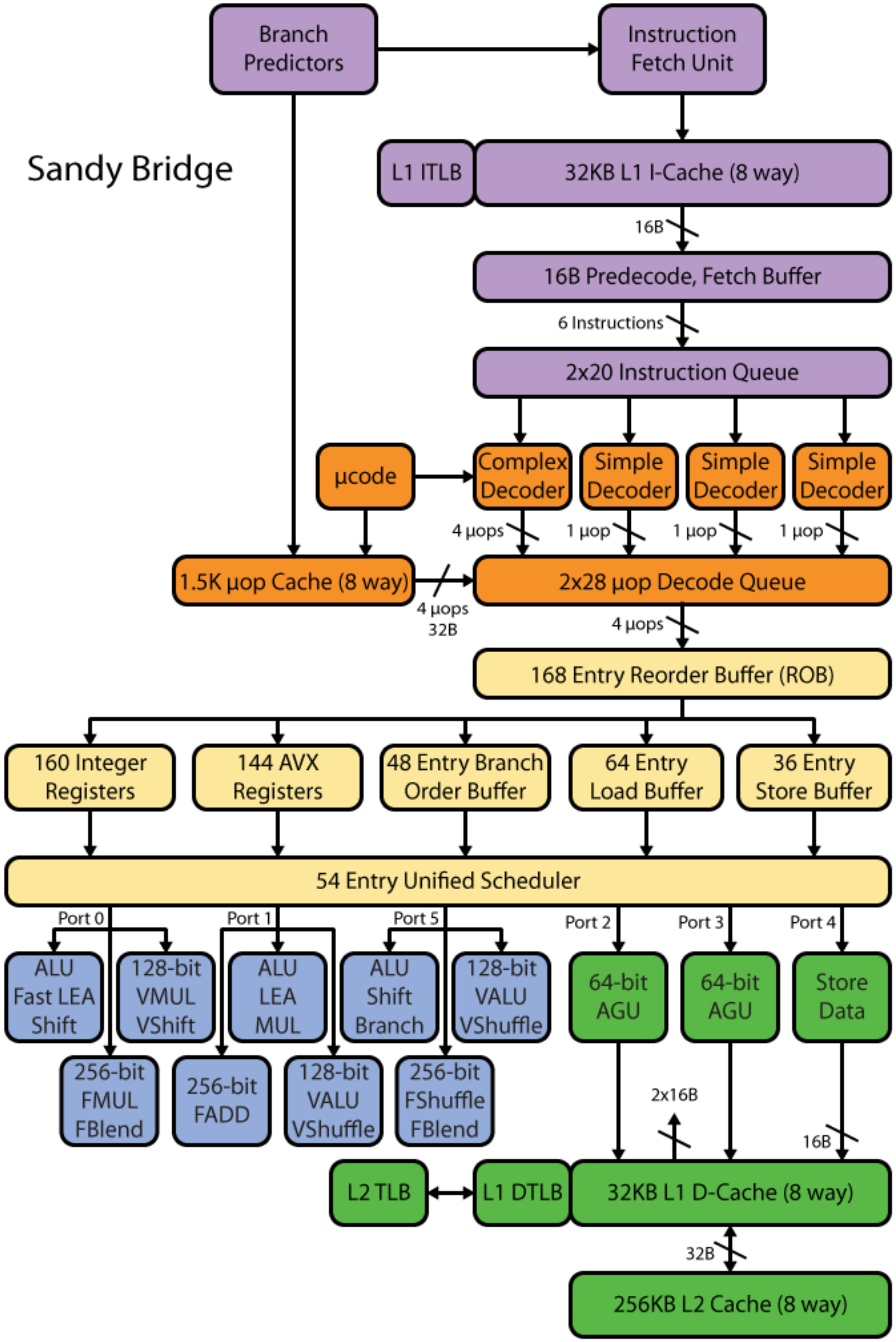
Intel CPU


Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller
including Display, DMI and Misc. I/O
Shared L3 Cache
Memory Controller I/O

Intel CPU

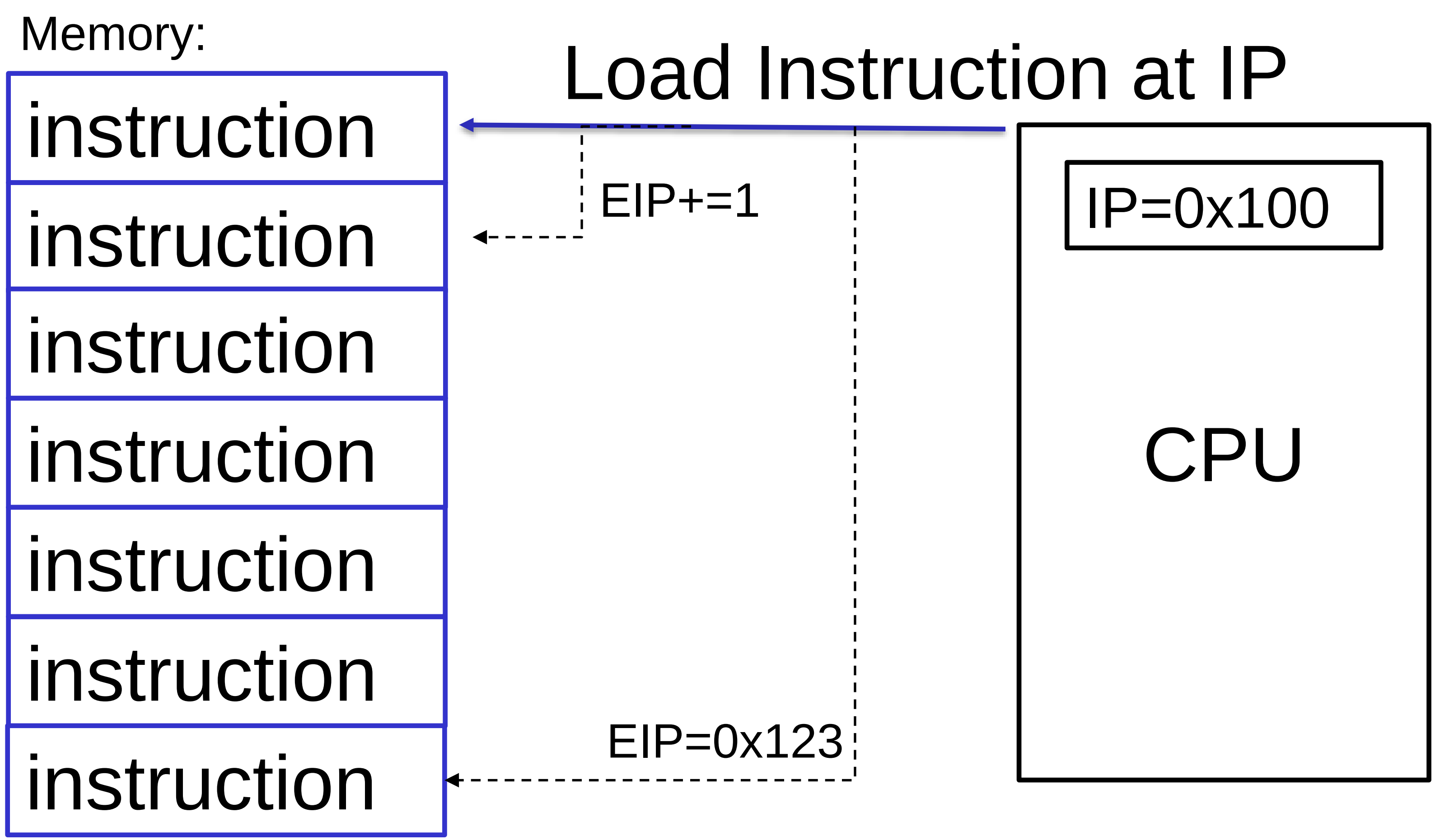
Pentium Die

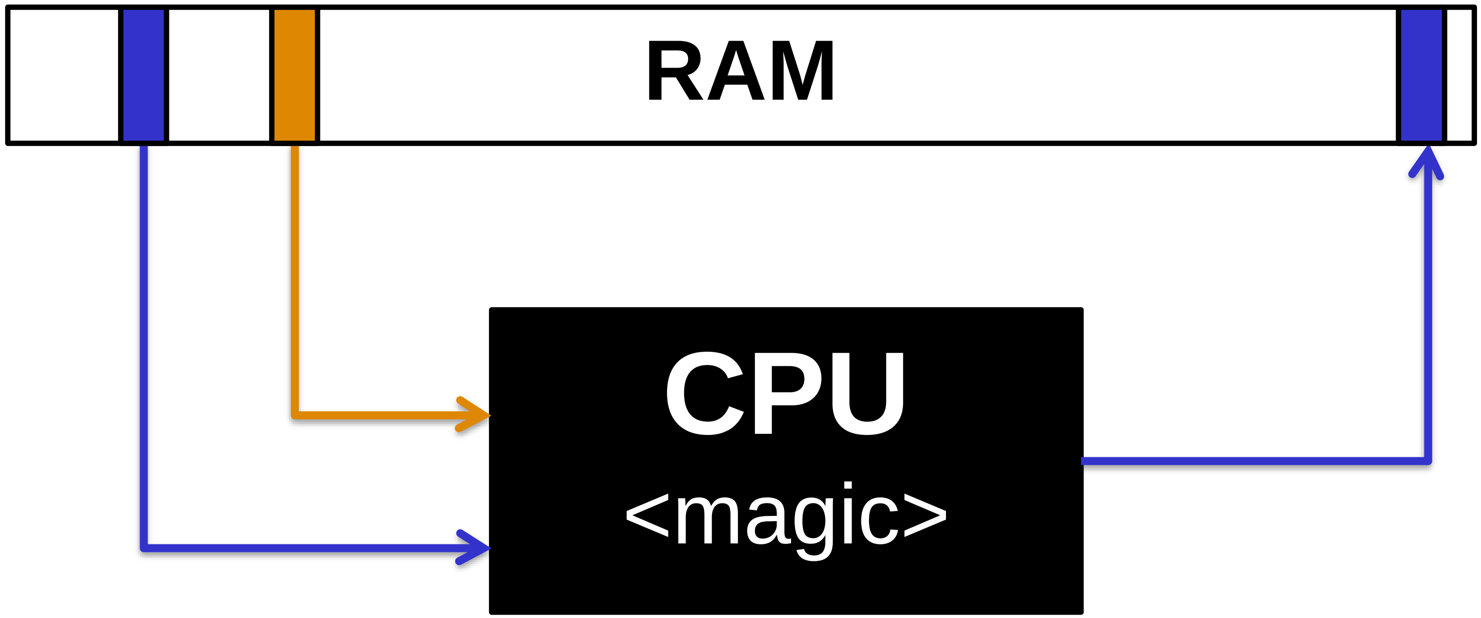




Sandy Bridge
Branch Predictors
Instruction Fetch Unit
L1 ITLB
32KB L1 I-Cache (8 way)
16B
16B Predecode, Fetch Buffer
6 Instructions
2x20 Instruction Queue
µcode
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
4 µops
1 µop
1 µop
1 µop
1.5K µop Cache (8 way)
4 µops 32B
2x28 µop Decode Queue
4 µops
168 Entry Reorder Buffer (ROB)
160 Integer Registers
144 AVX Registers
48 Entry Branch Order Buffer
64 Entry Load Buffer
36 Entry Store Buffer
54 Entry Unified Scheduler
Port 0
Port 1
Port 5
Port 2
Port 3
Port 4
ALU Fast LEA Shift
128-bit VMUL VShift
ALU LEA MUL
ALU Shift Branch
128-bit VALU VShuffle
64-bit AGU
64-bit AGU
Store Data
256-bit FMUL FBlend
256-bit FADD
128-bit VALU VShuffle
256-bit FShuffle FBlend
2x16B
16B
L2 TLB
L1 DTLB
32KB L1 D-Cache (8 way)
32B
256KB L2 Cache (8 way)

Overview: Computerz



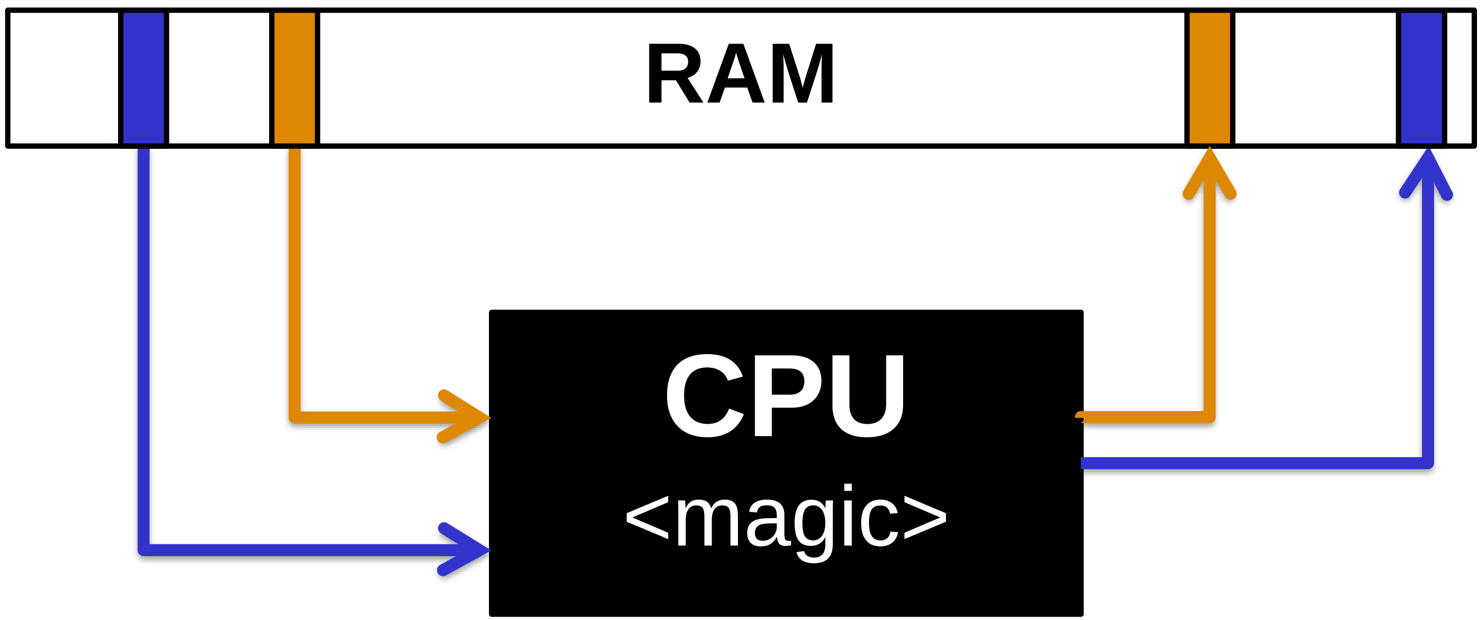


Read:
- Data
- Instructions

Write:
- Data

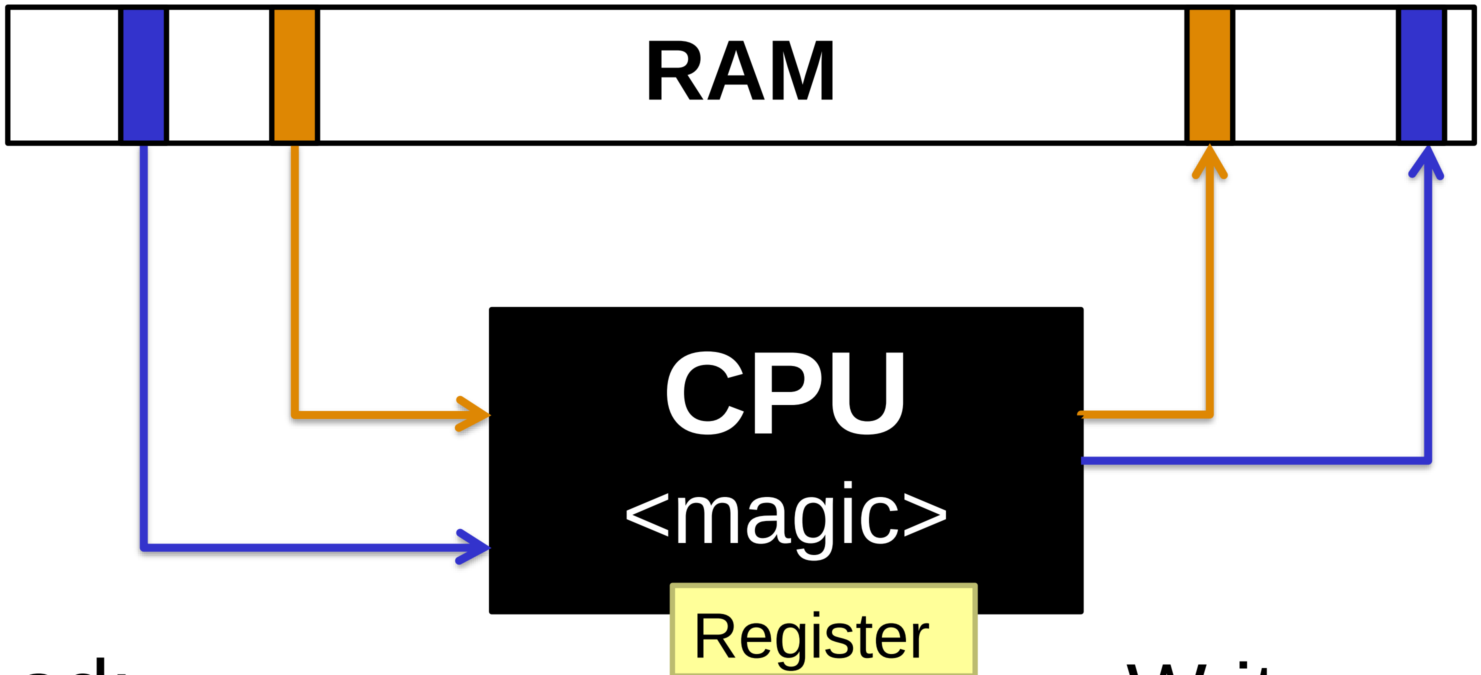
Overview: Computerz

von Neumann Architecture



Read:
- Data
- Instructions

Write:
- Data
- Instructions



Read:

- Data
- Instructions

Write:

- Data
- Instructions

Register are the "variables" on the CPU

Immediate access for the CPU

Cannot write Memory -> Memory

- Always: Memory -> Register -> Memory

Register: <1 cycle
L1: ~3
L2: ~14
RAM: ~240

Register are of a certain size

- e.g. 64 bit (can hold a number of size 8 bytes)

Register can hold:

- Data (numbers)
- Addresses (also numbers, but with a different meaning)

Registers can be used to:

- Perform computations(add, multiply, xor..)
- Read / Write memory(address in register)
- Execute instructions(special registers)

Overview: CPU Registers

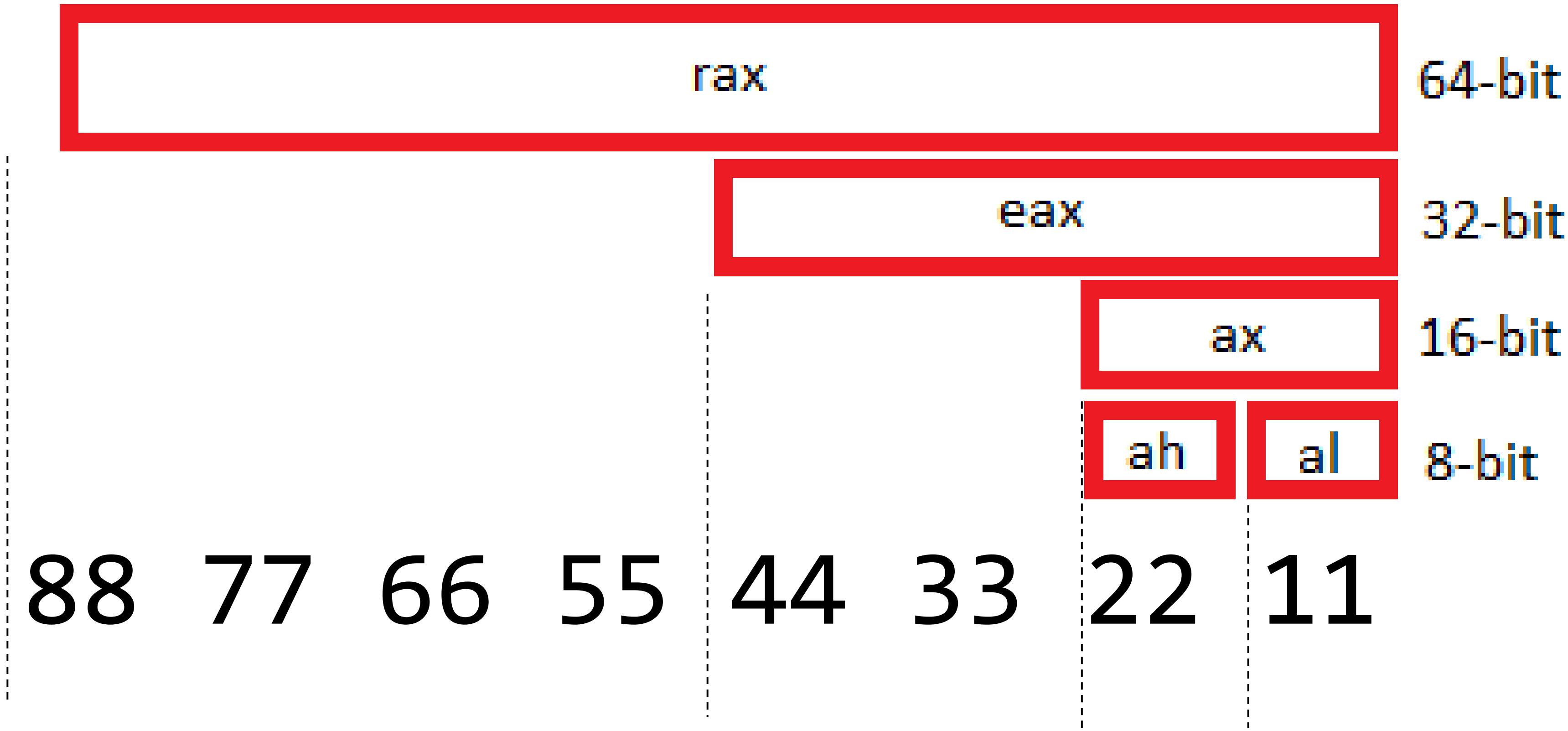
32	64	Acronym	What?
EAX	RAX	Accumulator	Adding stuff
EBX	RBX	Base	Referencing stuff
ECX	RCX	Count	Counting stuff
EDX	RDX	Data	Stuff
ESI	RSI	Source Index	Points to a source
EDI	RDI	Destination Index	Points to a destination
	R8-R15		General Purpose

Overview: CPU Registers

32	64	Acronym	Points to?
EIP	RIP	Instruction Pointer	Next instruction to be executed
ESP	RSP	Stack Pointer	Top of Stack
EBP	RBP	Base Pointer	Current Stack Frame (Bottom)

Print this slide and stick it on your bathroom mirror

Overview: CPU Registers



Overview: CPU Registers

Fun Fact: Current Intel CPU’s are compatible to the 8086

8086:

- From 1978
- 5-10mhz

Overview: CPU Registers

Recap:

- CPU work with **registers**
- Registers can hold **data**
- Registers can also hold **addresses** of memory locations (to write/read data)
- They can be 32 bit (**E**AX) or 64 bit (**R**AX)
- Some registers are multi-purpose
- Some registers are special (RIP, RBP, RSP)

How a CPU interprets instructions

CPU Instructions

A CPU in a few lines of pseudo code:

```
instr = [ 0x01 0xA0 0xB0 0x02 0xA1 0xA2 .. ]
ip = 0

while true:
  switch instr[ip]:
      case 0x01:
           add( instr[ip+1], instr[ip+2] )
           ip = ip + 3
           break
      case 0x02:
           sub( instr[ip+1], instr[ip+2] )
  …
```

CPU Instructions

83 c4 08	**add** $0x8,%rsp
83 c3 01	**add** $0x1,%rbx
83 44 24 0c 01	**addl** $0x1,0xc(%rsp)
83 05 41 94 2c 00 01	**addl** $0x1,0x2c9441(%rip)

CPU Opcodes

Assemble ("compile") ←

Disassemble →

Assembler Instructions

Fraunhofer
FKIE
FRAUNHOFER-INSTITUT FÜR KOMMUNIKATION, INFORMATIONSVERARBEITUNG UND ERGONOMIE FKIE

x86 Opcode Structure and Instruction Overview

1st \ 2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD						PUSH ES	POP ES	OR						PUSH CS	TWO BYTE
1	ADC						PUSH SS	POP SS	SBB						PUSH DS	POP DS
2	AND						ES SEGMENT OVERRIDE	DAA	SUB						CS SEGMENT OVERRIDE	DAS
3	XOR						SS SEGMENT OVERRIDE	AAA	CMP						DS SEGMENT OVERRIDE	AAS
4	INC								DEC							
5	PUSH								POP							
6	PUSHAD	POPAD	BOUND	ARPL	FS SEGMENT OVERRIDE	GS SEGMENT OVERRIDE	OPERAND SIZE (SIZE OVERRIDE)	ADDRESS SIZE (SIZE OVERRIDE)	PUSH	IMUL	PUSH	IMUL	INS		OUTS	
7	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
	Jcc															
8	ADD/ADC/AND/XOR OR/SBB/SUB/CMP				TEST		XCHG		MOV REG				MOV SREG	LEA	MOV SREG	POP
9	NOP	XCHG EAX							CWD	CDQ	CALLF	WAIT	PUSHFD	POPFD	SAHF	LAHF
A	MOV EAX				MOVS		CMPS		TEST		STOS		LODS		SCAS	
B	MOV															
C	SHIFT IMM		RETN		LES	LDS	MOV IMM		ENTER	LEAVE	RETF		INT3	INT IMM	INTO	IRETD
D	SHIFT 1 (ROL/ROR/RCL/RCR/SHL/SHR/SAL/SAR)		SHIFT CL		AAM	AAD	SALC	XLAT	FPU							
E	LOOPNZ (CONDITIONAL LOOP)	LOOPZ	LOOP	JECXZ	IN IMM		OUT IMM		CALL	JMP	JMPF	JMP SHORT	IN DX		OUT DX	
F	LOCK (EXCLUSIVE ACCESS)	ICE BP	REPNE (CONDITIONAL REPETITION)	REPE	HLT	CMC	TEST/NOT/NEG [i]MUL/[i]DIV		CLC	STC	CLI	STI	CLD	STD	INC DEC	INC/DEC CALL/JMP PUSH

1st \ 2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	{L,S}LDT {L,S}TR VER{R,W}	{L,S}GDT {L,S}IDT {L,S}MSW	LAR	LSL			CLTS		INVD	WBINVD		UD2		NOP		
1	*SSE{1,2,3}*								Prefetch *SSE1*	HINT_NOP						
2	MOV CR/DR								*SSE{1,2}*							
3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC *SMX*	MOVBE / THREE BYTE		THREE BYTE *SSE4*					
4	CMOV															
5	*SSE{1,2}*															
6	*MMX, SSE2*															
7	*MMX, SSE{1,2,3}, VMX*												*MMX, SSE{2,3}*			
8	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
	Jcc SHORT															
9	SETO	SETNO	SETB	SETNB	SETE	SETNE	SETBE	SETA	SETS	SETNS	SETPE	SETPO	SETL	SETGE	SETLE	SETG
	SETcc															
A	PUSH FS	POP FS	CPUID	BT	SHLD				PUSH GS	POP GS	RSM	BTS	SHRD		*FENCE	IMUL
B	CMPXCHG		LSS	BTR	LFS	LGS	MOVZX		POPCNT	UD	BT BTS BTR BTC	BTC	BSF	BSR	MOVSX	
C	XADD		*SSE{1,2}*					CMPXCHG	BSWAP							
D	*MMX, SSE{1,2,3}*															
E	*MMX, SSE{1,2}*															
F	*MMX, SSE{1,2,3}*															

Arithmetic & Logic
Memory
Stack
Control Flow & Conditional
Prefix
System & I/O
No Operation (NOP) / Multiple Instructions / Extended Instruction Set

General Opcode Structure

Element Information # of bytes	Prefix 0-4	Opcode 1-3	AddrMode (mod, reg, r/m) 0-1	SIB Byte (scale, index, base) 0-1	Displacement 0/1/2/4	Immediate Data 0/1/2/4
Bit structure		O O O O O O D L	MOD MOD REG REG REG RM RM RM	S S I I I B B B		

Main Opcode bits
Direction bit
Operand length bit
Addressing mode
Register/Opcode modifier, defined by primary opcode
r/m field
Scale field
Index field
Base field

Addressing Modes

mod	00		01		10		11
r/m	16bit	32bit	16bit	32bit	16bit	32bit	r/m // REG
000	[BX+SI]	[EAX]	[BX+SI]+disp8	[EAX]+disp8	[BX+SI]+disp16	[EAX]+disp32	AL / AX / EAX
001	[BX+DI]	[ECX]	[BX+DI]+disp8	[ECX]+disp8	[BX+DI]+disp16	[ECX]+disp32	CL / CX / ECX
010	[BP+SI]	[EDX]	[BP+SI]+disp8	[EDX]+disp8	[BP+SI]+disp16	[EDX]+disp32	DL / DX / EDX
011	[BP+DI]	[EBX]	[BP+DI]+disp8	[EBX]+disp8	[BP+DI]+disp16	[EBX]+disp32	BL / BX / EBX
100	[SI]	*SIB*	[SI]+disp8	*SIB*+disp8	[SI]+disp16	*SIB*+disp32	AH / SP / ESP
101	[DI]	disp32	[DI]+disp8	[EBP]+disp8	[DI]+disp16	[EBP]+disp32	CH / BP / EBP
110	disp16	[ESI]	[BP]+disp8	[ESI]+disp8	[BP]+disp16	[ESI]+disp32	DH / SI / ESI
111	[BX]	[EDI]	[BX]+disp8	[EDI]+disp8	[BX]+disp16	[EDI]+disp32	BH / DI / EDI

SIB Byte Structure

encoding	scale (2bit)	Index (3bit)	Base (3bit)
000	$2^0=1$	[EAX]	EAX
001	$2^1=2$	[ECX]	ECX
010	$2^2=4$	[EDX]	EDX
011	$2^3=8$	[EBX]	EBX
100	--	none	ESP
101	--	[EBP]	disp32 / disp8+ [EBP] / disp32 + [EBP]
110	--	[ESI]	ESI
111	--	[EDI]	EDI
SIB value = index * scale + base			

v1.0 – 30.08.2011
Contact: Daniel Plohmann – +49 228 73 54 228 – daniel.plohmann@fkie.fraunhofer.de

Source: Intel x86 Instruction Set Reference
Opcode table presentation inspired by work of Ange Albertini

Add

Opcode	Mnemonic	Description
04 ib	ADD AL, imm8	Add imm8 to AL
05 iw	ADD AX, imm16	Add imm16 to AX
05 id	ADD EAX, imm32	Add imm32 to EAX
80 /0 ib	ADD r/m8, imm8	Add imm8 to r/m8
81 /0 iw	ADD r/m16, imm16	Add imm16 to r/m16
81 /0 id	ADD r/m32, imm32	Add imm32 to r/m32
83 /0 ib	ADD r/m16, imm8	Add sign-extended imm8 to r/m16
83 /0 ib	ADD r/m32, imm8	Add sign-extended imm8 to r/m32
00 /r	ADD r/m8, r8	Add r8 to r/m8
01 /r	ADD r/m16, r16	Add r16 to r/m16
01 /r	ADD r/m32, r32	Add r32 to r/m32
02 /r	ADD r8, r/m8	Add r/m8 to r8
03 /r	ADD r16, r/m16	Add r/m16 to r16
03 /r	ADD r32, r/m32	Add r/m32 to r32

Description

Adds the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

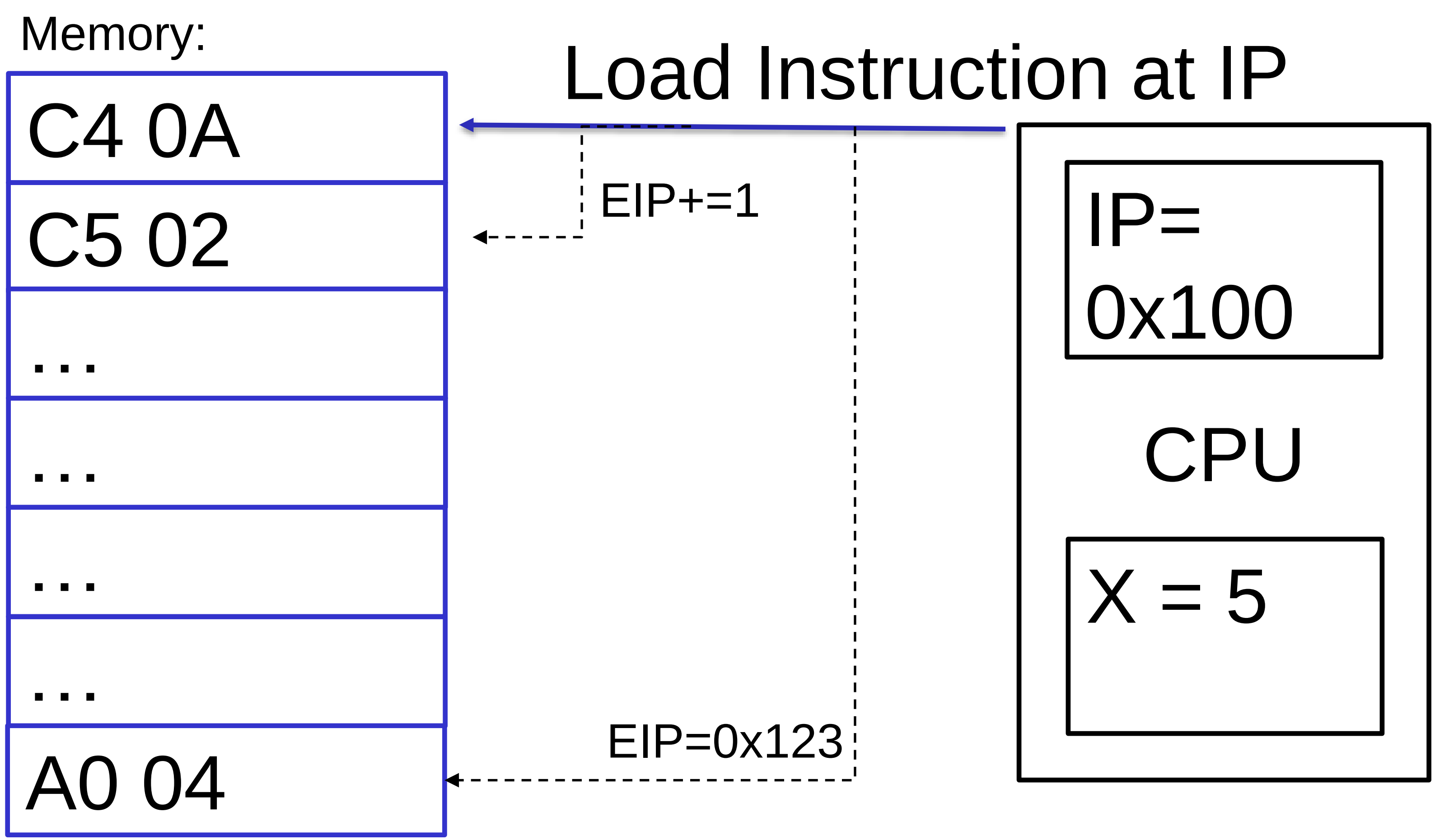
```
Destination = Destination + Source;
```

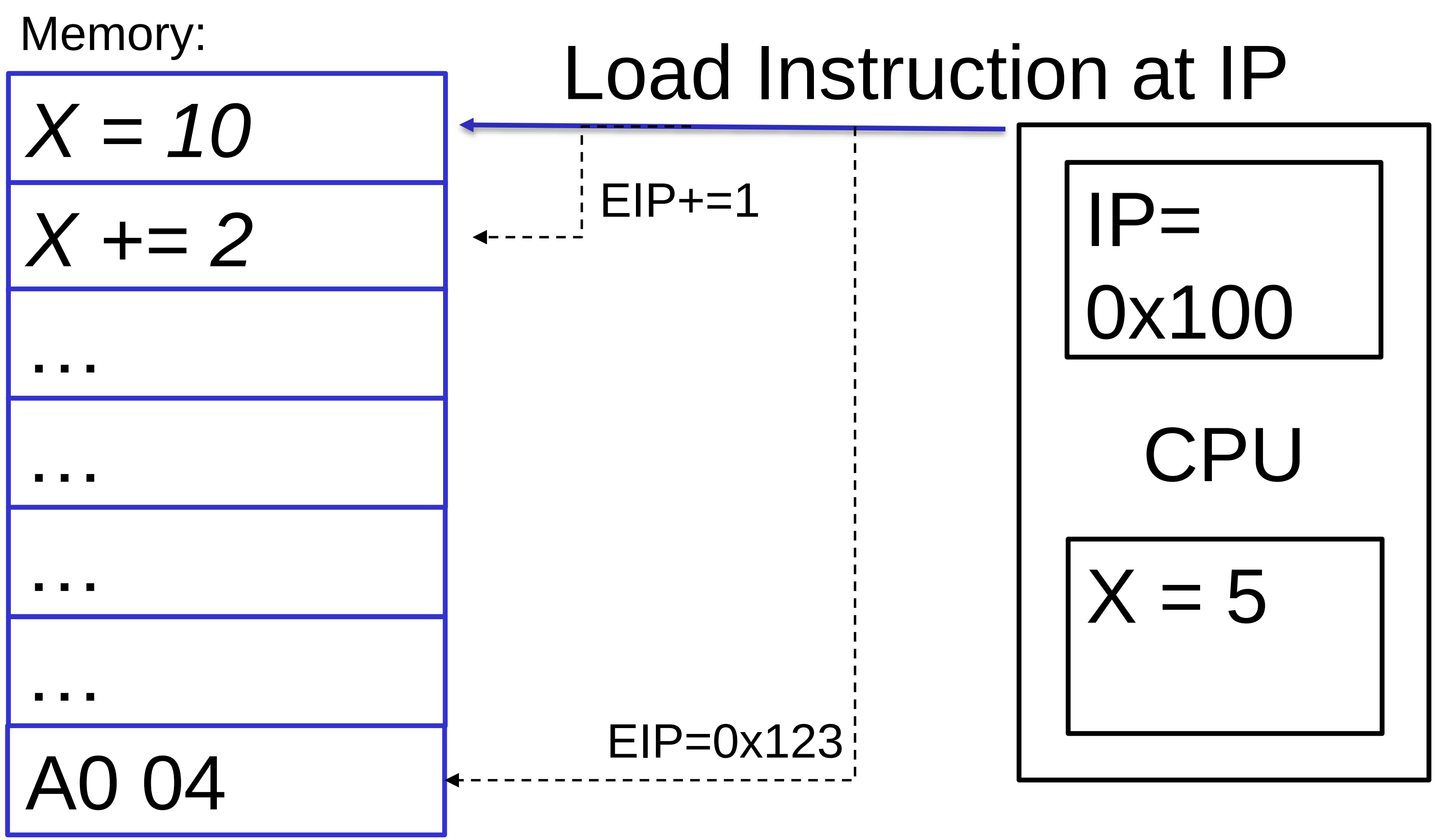
CPU Instructions

Recap:

- CPU looks at bytes, and then decides what to execute (“do”) based on them

Overview: Computerz




Memory:
X = 10
X += 2
...
...
...
...
A0 04
Load Instruction at IP
EIP+=1
EIP=0x123
IP=
0x100
CPU
X = 5

Hex Numbers, and Little Endian

A guide to understand the rest of my slides

Hex Numbers, and Little Endian

Intel CPU’s

- 1 Byte = 8 Bit
- Little endian

Hex Numbers, and Little Endian

How many bits in a byte?

Intel CPU's

- 1 Byte = 8 Bit
- Little endian

Others:

- CDC 6000: 18, 24 and 60 bit
- PDP1/9/15: 18 bit words
- Apollo Guidance Computer: 15 bit

- ARM and other RISC: 8 bit Bytes, but in Big Endian

Decimal: 0 1 2 3 4 5 6 7 8 9

1 decimal digit: 10 values

2 decimal digits: 100 values

10 * 10 = 100

Hex Numbers

Hex: 0 1 2 3 4 5 6 7 8 9 **A B C D E F**

1 hex digit: 16 values (4 bit, 2^4)

*2 hex digits: 256 values (8 bit, 2^8 = 2^4 * 2^4)*

16 * 16 = 256

1 Byte = 8 Bit = 256 values!

Hex numbers

0x00 = 0

0x01 = 1

0x0f = 15

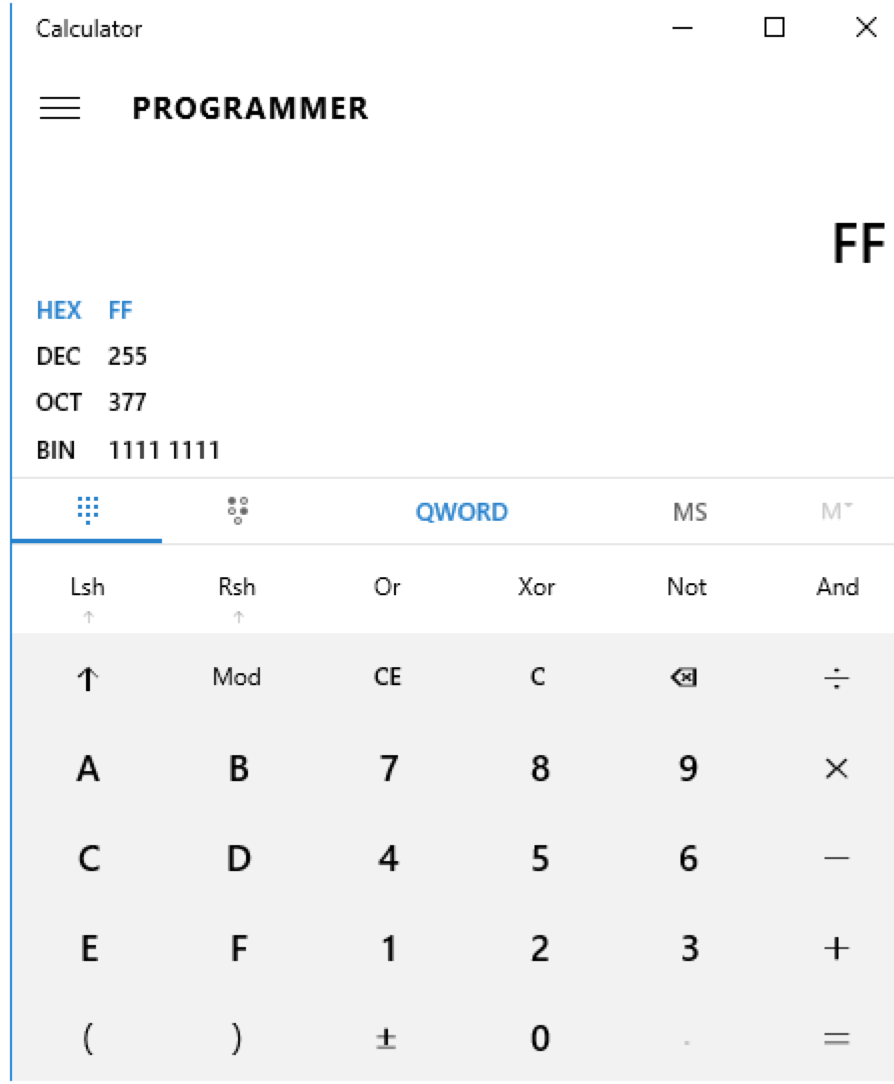
0x10 = 16

0x11 = 17

0x20 = 32

0xf0 = 240

0xff = 255



Hex Numbers

Base 10 6975

Base 16 0x1B3F

Nibbles 0001 1011 0011 1111

Hex Numbers

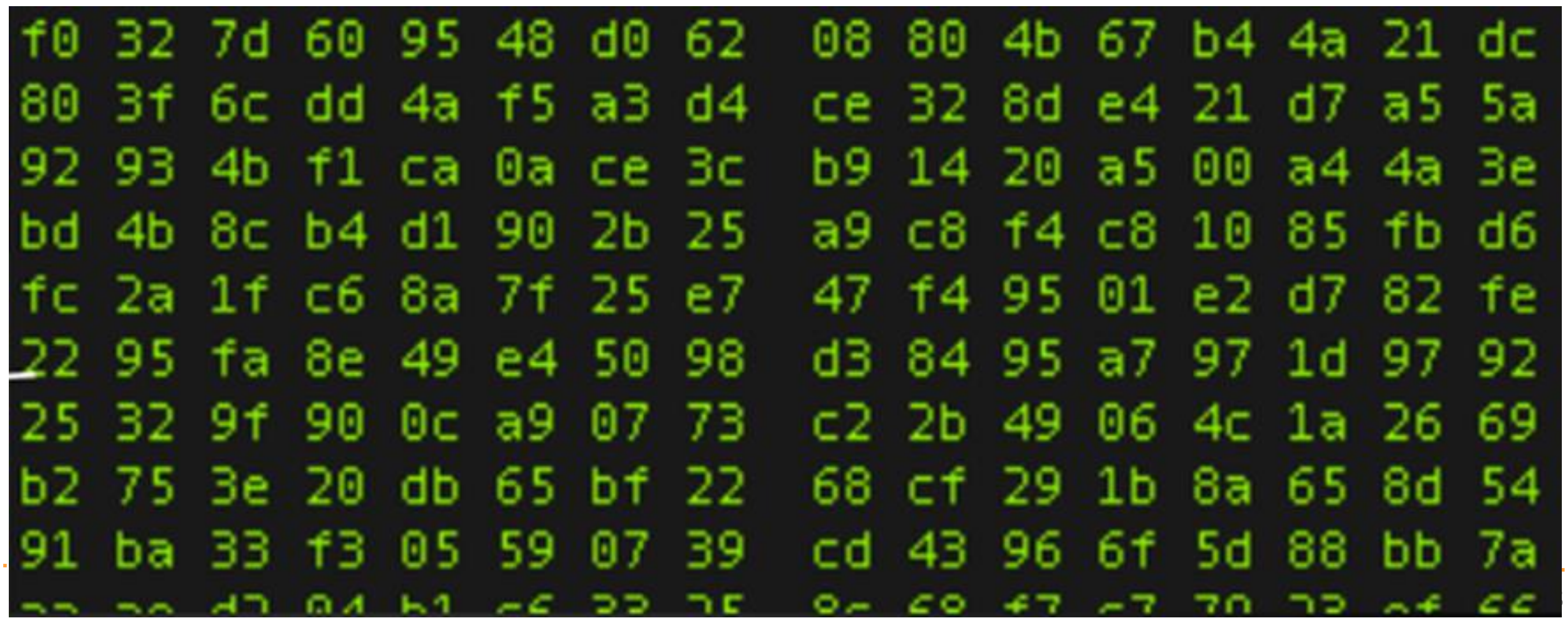
Base 10	6975
Base 16	0x1B3F
Nibbles	0001 1011 | 0011 1111
Bytes	0x1B | 0x3F

Endianness

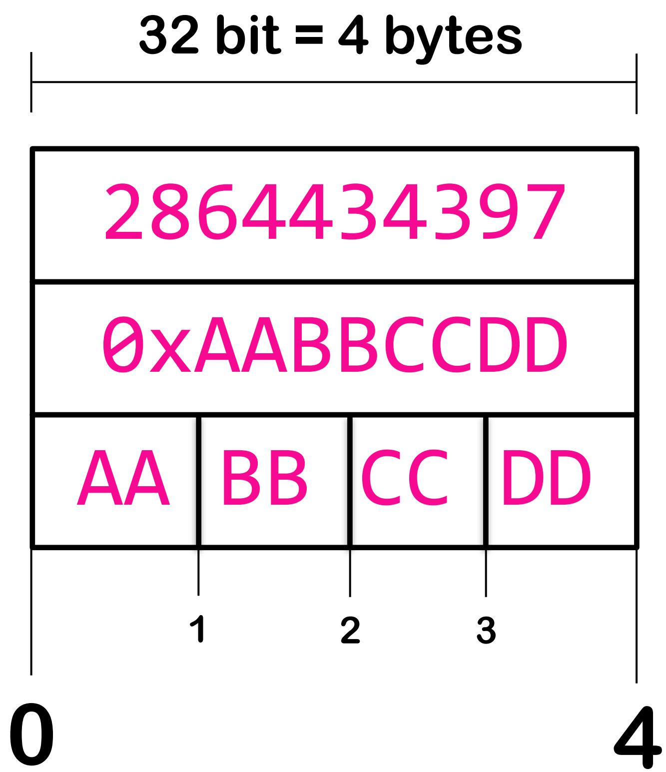
Number: 0x1B3F

Big Endian: 0x1B 0x3F -> 0x1B3F

Little Endian: 0x3F 0x1B -> 0x3F1B



Endianness: Big Endian (ARM)

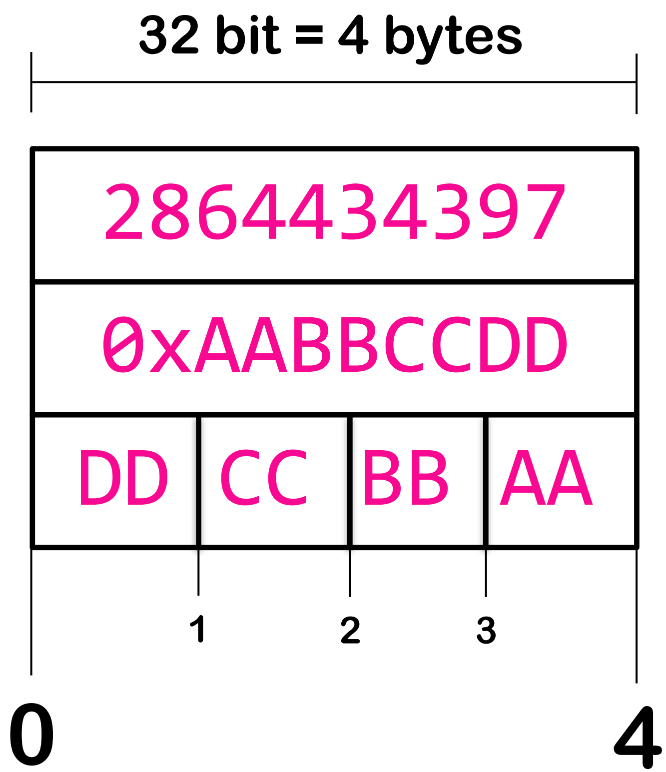


Number in Decimal (10)

Number in Hex (16)

Big Endian Storage

Endianness: Little Endian (Intel)



Number in Decimal (10)

Number in Hex (16)

Little Endian Storage

We don’t know what type the bytes in memory are

DD	CC	BB	AA

Four 8 bit numbers:

- DD
- CC
- BB
- AA

Two 16 bit numbers:

- 0xCCDD
- 0xAABB

A 32 bit number:

- 0xAABBCCDD

??

Number:

0x1122334455667788

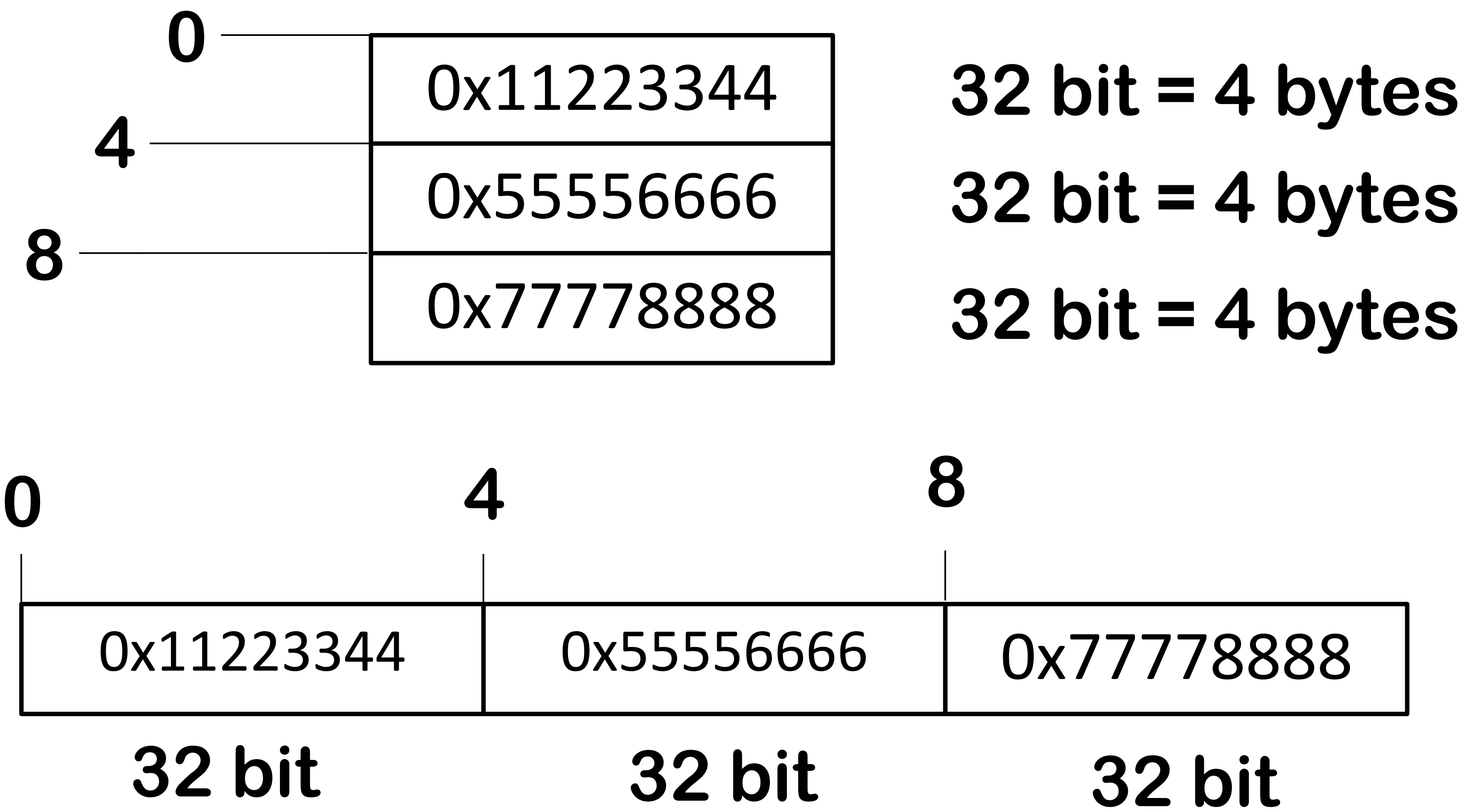
Little Endian:

88	77	66	55	44	33	22	11
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Numbers in memory

0	0x11223344	32 bit = 4 bytes
4	0x55556666	32 bit = 4 bytes
8	0x77778888	32 bit = 4 bytes

Numbers in memory



Numbers in memory

In memory:

0	4	8
0x44 0x33 0x22 0x11	0x66 0x66 0x55 0x55	0x88 0x88 0x77 0x77

For humans:

0	4	8
0x11223344	0x55556666	0x77778888
32 bit	**32 bit**	**32 bit**

ASCII

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

ASCII - Hexdump

```
0008b400  25 00 e9 cc fd ff ff 66  0f 1f 84 00 00 00 00 00  |%......f........|
0008b410  e8 2b 21 01 00 e9 c4 fe  ff ff 66 0f 1f 44 00 00  |.+!.......f..D..|
0008b420  44 3b 2d f1 fe 25 00 0f  8f 5f fe ff ff 8b 05 d5  |D;-..%..._......|
0008b430  fe 25 00 3b 05 fb fd 25  00 0f 8d 4d fe ff ff 83  |.%.;...%...M....|
0008b440  c0 01 89 05 c0 fe 25 00  e9 3f fe ff ff 0f 1f 00  |......%..?......|
0008b450  41 57 41 56 41 55 49 89  fd 41 54 55 53 48 83 ec  |AWAVAUI..ATUSH..|
0008b460  08 8b 1d cd fd 25 00 8b  6f 20 8d 43 01 4c 63 e3  |.....%..o .C.Lc.|
0008b470  89 ef 89 2d b8 dd 25 00  89 05 b6 fd 25 00 48 8b  |...-..%.....%.H.|
```

Numbers in memory

Recap:

- Numbers can be displayed in decimal, or hex (0-9, a-f)
- Numbers are stored as 16, 32 or 64 bit values, mostly as little endian
- If we look at little endian numbers as bytes, they are reversed
- If we look at numbers in memory, we can't know if they are 8, 16, 32 or 64 bit
- We can try to interpret bytes as ASCII

Numbers test!

0x07 + 0x07 = ?

0x0a + 0x0a = ?

0x22 – 0x10 = ?

0x22 – 0x0a = ?

0x 0b b4 c0 0d in little endian = 0x????????

ASCII 0x42 0x46 0x48 = «???»

The string «leet» in ASCII = 0x?? 0x?? 0x?? 0x??

Operating System Basics

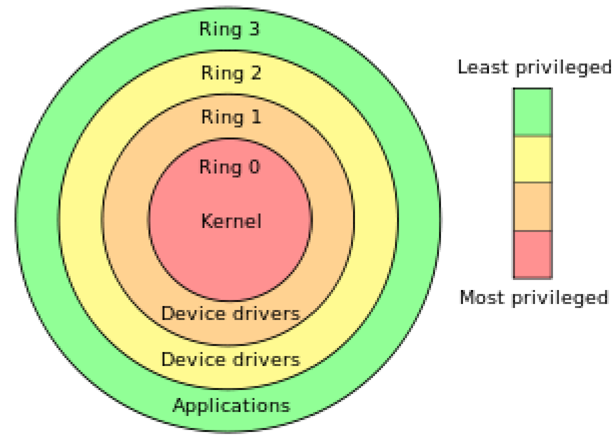
OS Basics: Rings

Ring 0: Kernel (Kernelspace)

- Not covered here
- Can be interacted with by using “syscalls”

Ring 3: Userspace

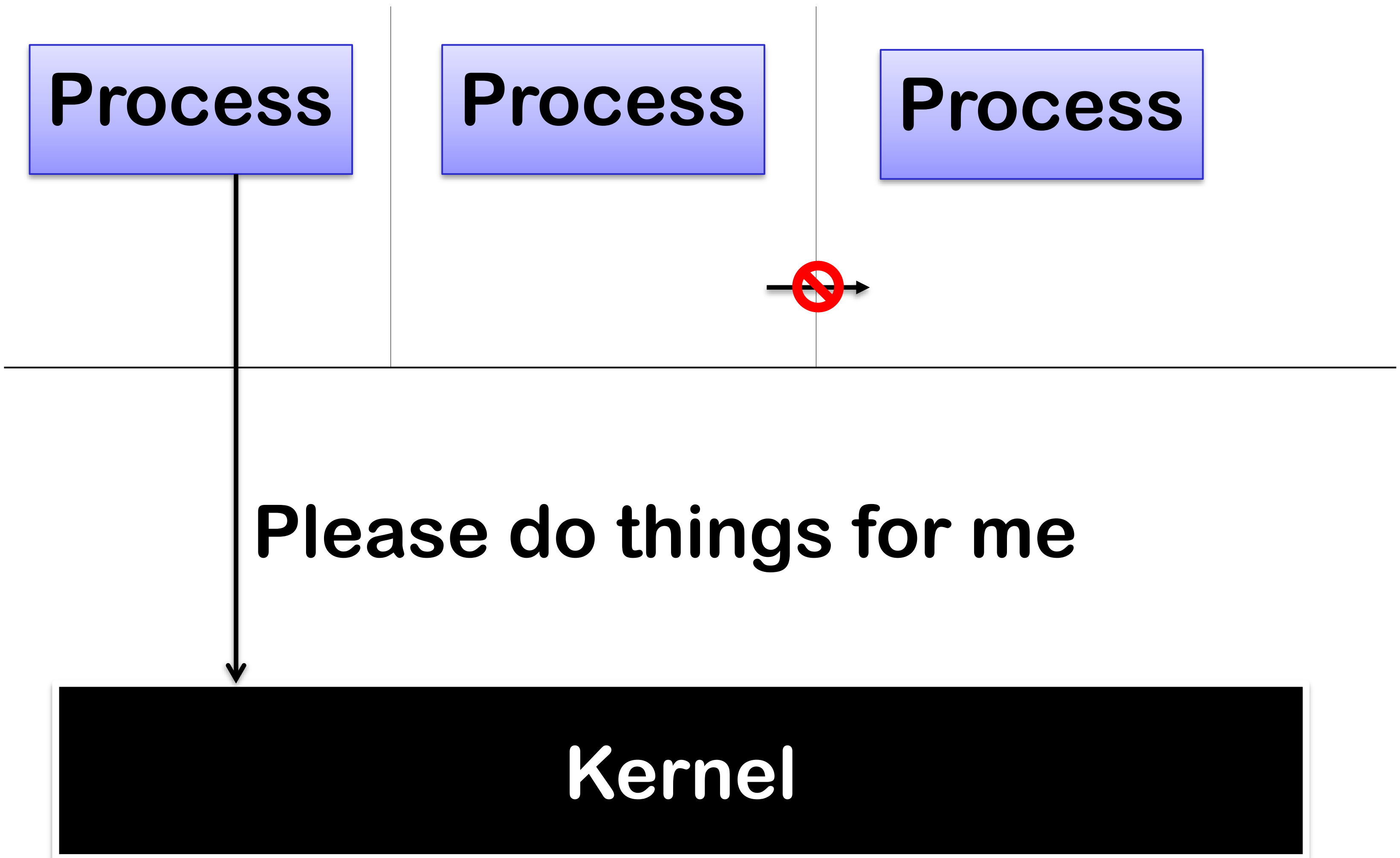
- Where all programs run
- ls, Bash, Vim, Apache, Xorg, Firefox, …



How to transit from userspace to kernelspace?

- System Calls (syscall)

OS Basics: Processes



A Process:

- Is a **running** program
  - Program lives on disk (static, dead)
  - Process lives in memory (interpreted, alive)
- Process thinks he “owns” the hardware
  - RAM
  - CPU

Multiple processes can run simultaneously

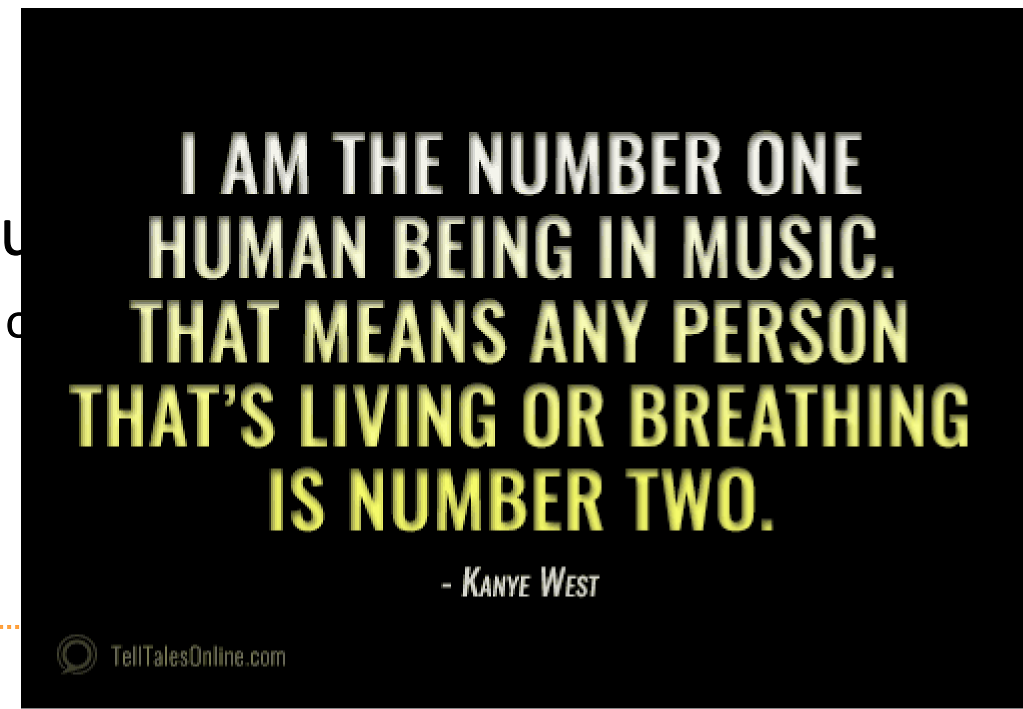
- Everyone thinks he is the only one
- Like Kanye West

A Process:

- Is a *running* program
  - Program lives on disk (static)
  - Process lives on memory (alive)
- Process thinks he "owns" the hardware
  - RAM
  - CPU

Multiple processes can ru

- Everyone thinks he is the c
- Like Kanye West



Processes can address:

- **4 GB of memory** in 32bit OS
  - (2-3 GB actually)

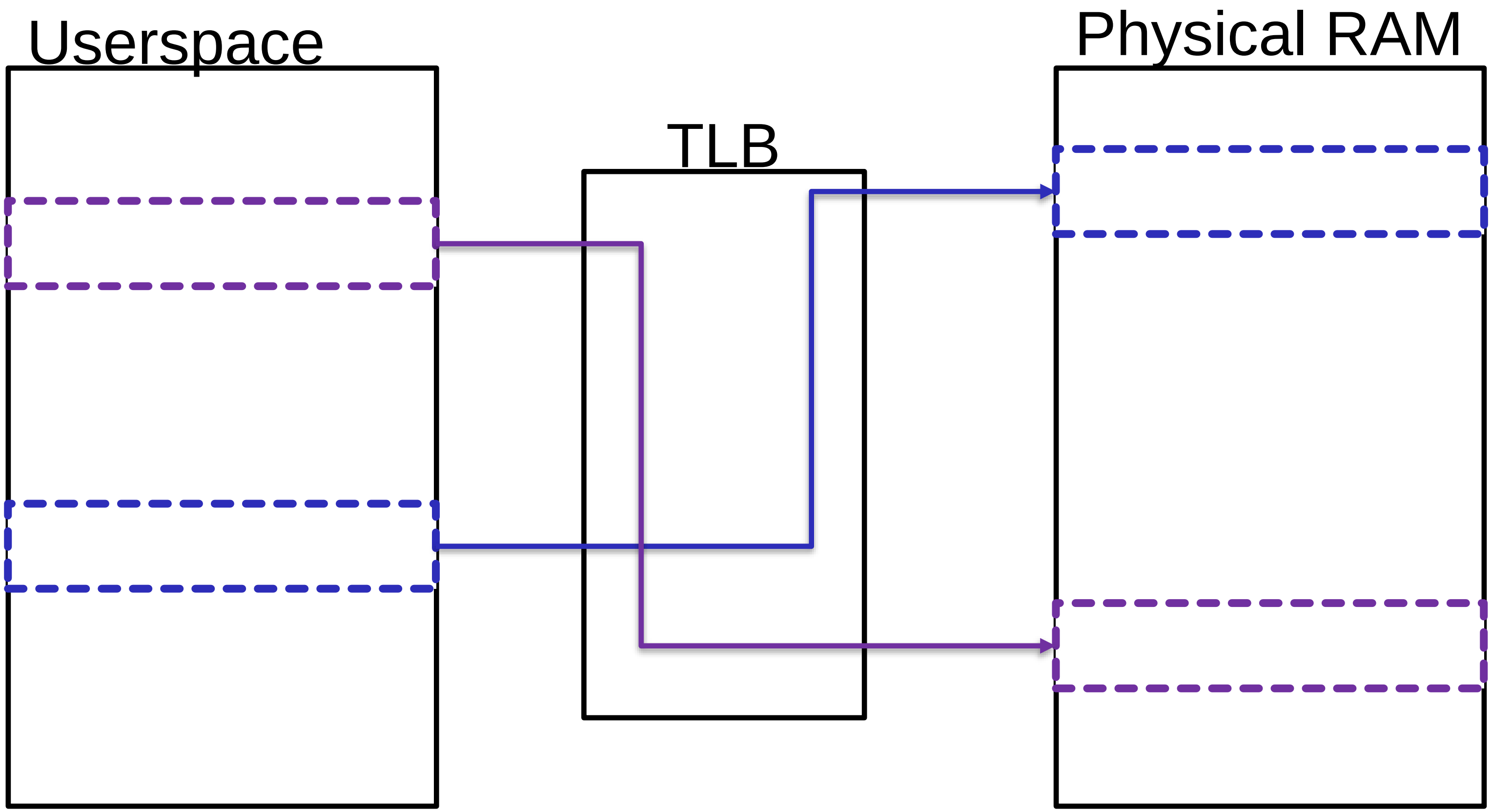
- Independent on how much memory (RAM) there really is

What if we have:

- Only 2 GB RAM?
  - OOM (Out Of Memory) when too much memory is used
- 8 GB RAM?
  - 2 Processes can use all their 4GB!

OS Basics – Kernelspace/Userspace

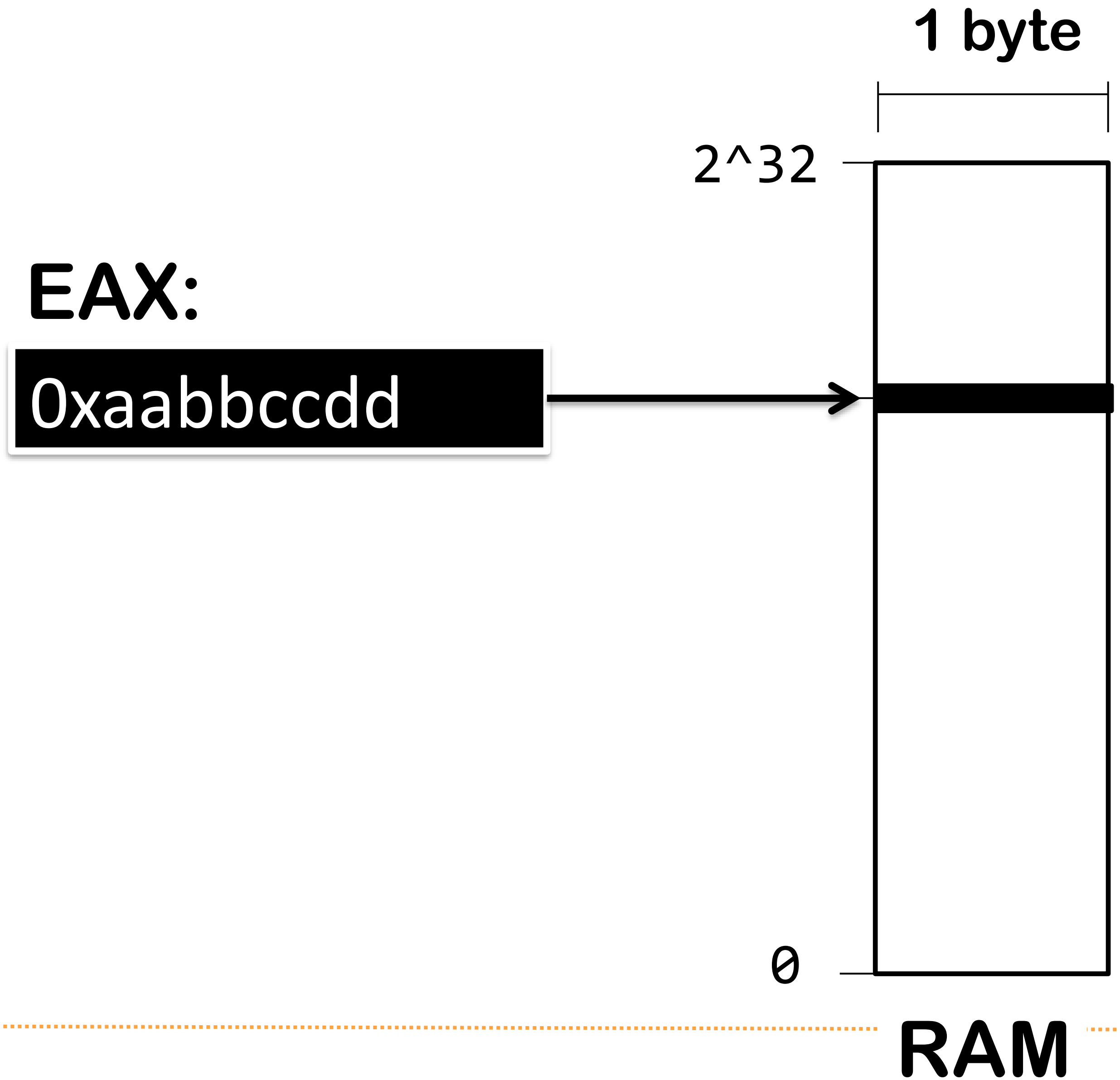
OS/CPU manages mapping between physical pages and process (virtual) pages – called **paging**



Why 4 GB?

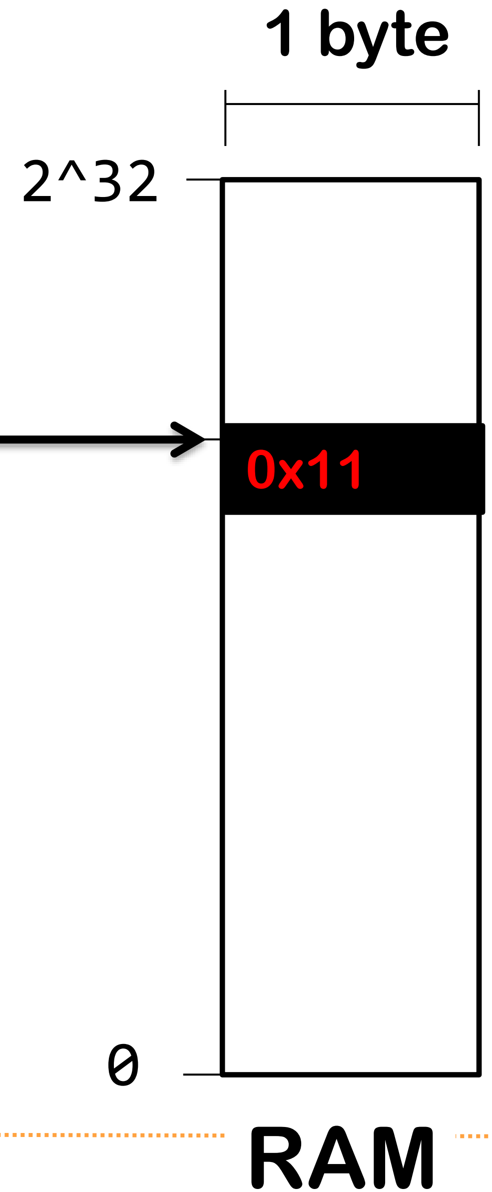
- 32 bit register size in Intel CPU
- Register are used to address memory
- 2^32 = 4 billion = 4 gigabyte

A process has therefore access to **4 billion one-byte** memory locations


1 byte
2^32
EAX:
0xaabbccdd
0
RAM

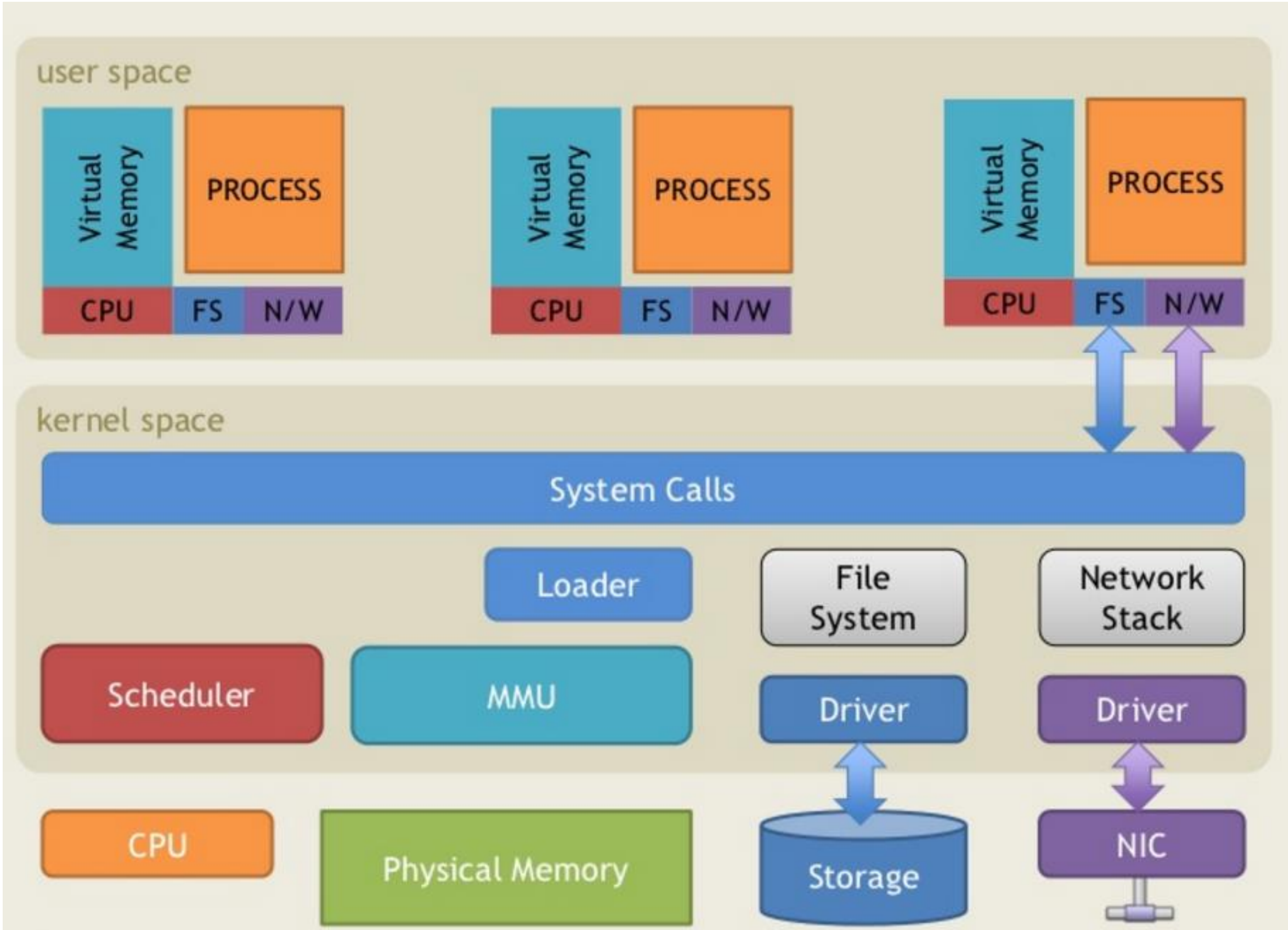
EAX:

0xaabbccdd




```
"C like":
eax = 0xaaabbccdd
*eax = 0x11
```

OS Overview

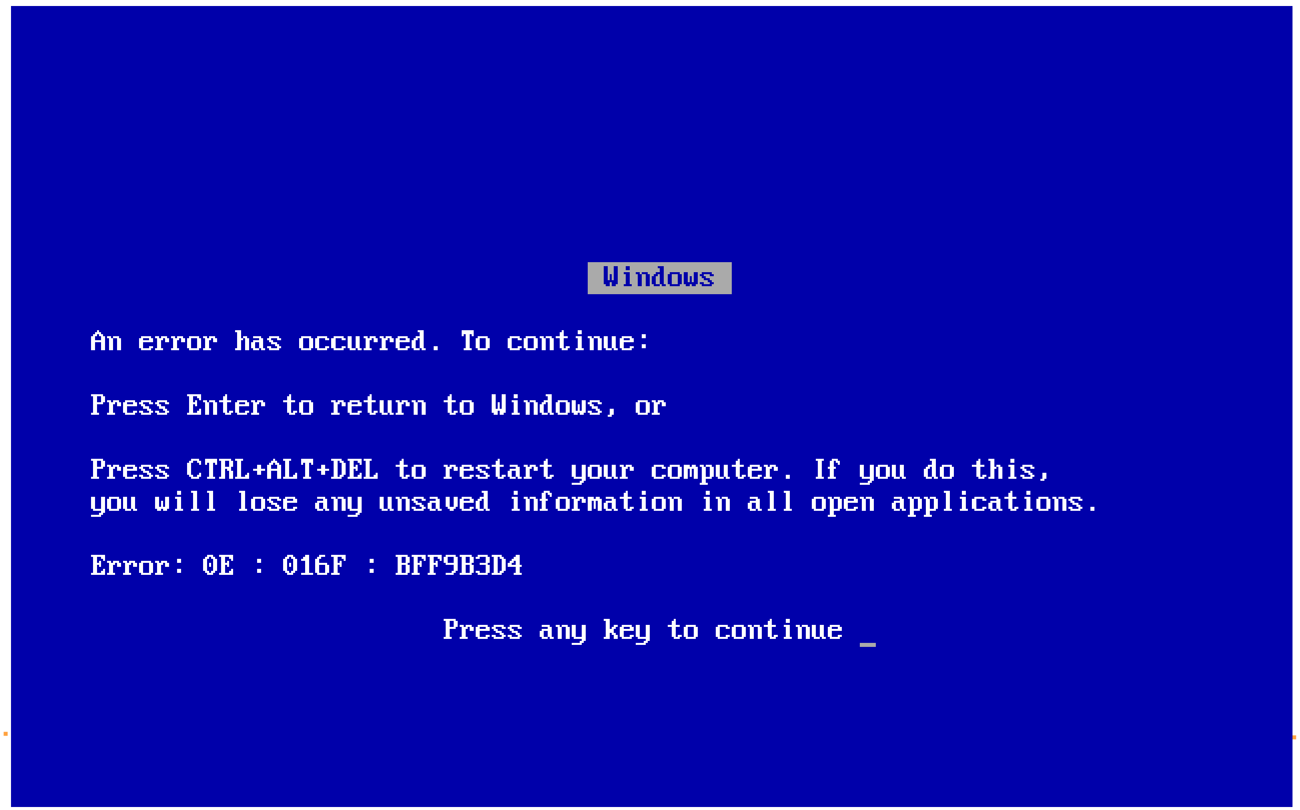


http://www.slideshare.net/saumilshah/operating-systems-a-primer

OS Overview

History lesson: "The good old times"

- Windows did not have true protected memory until windows NT/2000
  - Including all of DOS, Windows 3.1, Windows 95, 98, ME
- Every process could write into all all other processes, or even the OS
- "Blue screen of death"



There's only one CPU, how can:

- Multiple programs run at the same time?
- The OS and the programs run at the same time?

Solution: Interrupts

- Timer interrupts
- Interrupts are handled by the kernel "callbacks"
  - Time / clock
  - Network interface
  - USB devices
- Kernel schedules the different processes

OS Overview

Recap:

- Processes are programs which are alive in the RAM
- Every process thinks he owns the computer (including all the RAM)
- Every process has access to 2^32 (~4 billion) memory locations of 1 byte size

32 bit vs 64 bit

This chapter may expect some know-how from later chapters

its more like a reference

32bit vs 64bit

From 32 to 64 bit

- You all are probably too young for this
- But it was kinda big thing
- AMD invented the current 64 bit architecture
  - Intel wanted a new one: Itanium. Failed hard.
  - (AMD was better than Intel in many respects back then. Sigh).
- x86 to x64/amd64
  - 8086, 80286, 80386, 80486, 80586 aka Pentium

- “Is windows 64 bit twice as good/fast than windows 32 bit?”
  - Width of the CPU registers define the amount of addressable memory

32bit vs 64bit

64 bit pros:

- Can address more than 4 gb of memory per computer
  - 18’446’744’073’709’551’616 bytes theoretically
- 64 bit calculations are maybe a bit faster

64 bit cons:

- Programs use more space
  - Because pointers and data-types (integer) are twice as big
  - On disk, memory and most importantly: cache

32bit vs 64bit

64 bit registers are prefixed with “R” (RAX, RIP, …)

New registers: R8-R15

Pointers are 64 bit

Push/Pop are 64 bit

For 64 bit:

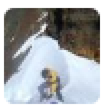
- 64 bit are 18 exabytes
- Only 47 bit are used (=140 terabytes)
- `< 0x00007fffffffffff`

32bit vs 64bit

For 64 bit:

- 64 bit are 18 exabytes
- Only 47 bit are used (=140 terabytes)
- `< 0x00007fffffffffff`

halvarflake Retweeted



Anders Fogh @anders_fogh · 7h

Yay. We're getting 57 bit physical address space.

> **Giuseppe `N3mes1s`** @gN3mes1s
> 5-Level Paging and 5-Level EPT - software.intel.com/sites/default/...

2 11 8 •••

intel

5-Level Paging and 5-Level EPT

White Paper

Revision 1.0

December 2016

32bit vs 64bit

Linux (and Windows) can execute 32 bit processes on a 64 bit OS

- C:\Program Files
- C:\Program Files (x86)

- /lib/lib
- /lib/lib64

The 32 bit process does not realize he’s on a 64 bit system

- But needs a 32 bit runtime

32bit vs 64 bit

For this presentation:

32 bit is “old” and “dead”

- But its much easier to create and explain exploits in it

Old plan was: “Lets be modern, 64 bit only”

Current plan: “Lets be modern, but still use 32 bit to exploit stuff”

Sorry…

32bit vs 64bit

Recap

- There are some differences between 32 and 64 bit
- A 32 bit process can run on a 64 bit system as 32 bit

Excursion: C64

Commodore 64

1 process only

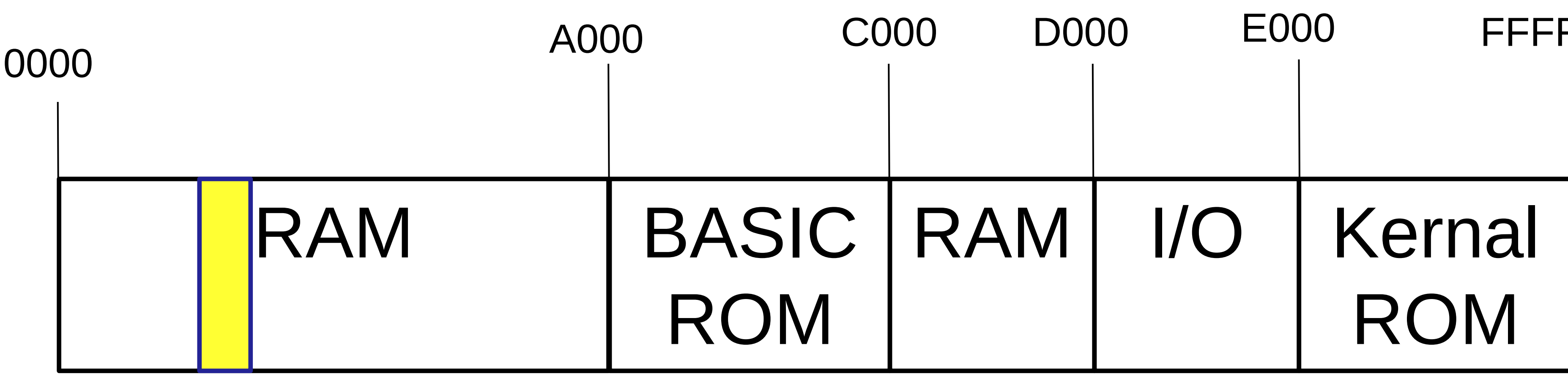
~1 MHz, 64kb RAM (2^16)

3 general purpose registers, 8 bit: A, X, Y

Stack Pointer: 8 bit, IP: 16 bit

Screen memory mapping: 0x0400 (1000 bytes, 40x25)

```
00 01 02 03
40 41 42 43
```



Videos, for the curious

https://media.ccc.de/v/34c3-9064-the_ultimate_apollo_guidance_computer_talk

https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/DEFCON-25-Christopher-Domas-Breaking-The-x86-ISA.pdf