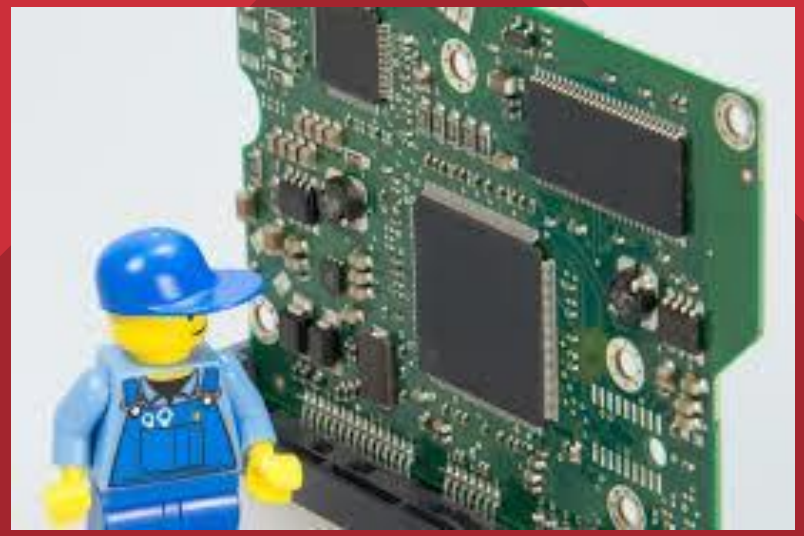
Høyskolen Kristiania

Høyskolen Kristiania

TK1100
Digital Technology

Computer architecture



Learning objectives covered so far (1)

- What is in the computer system
  - computer assembly and main components
- The computer language, alphabet,
- The Binary Numbering system.

Today: Computer Architecture

Boolean algebra
- NOT, AND, OR, XOR
- Sequential and combinatorial circuits.

Instruction Set Architecture (ISA).
- What instructions?
- Word size (number of bits)
- Addressing mode
- Register
- Data formats
- What the CPU is suitable of doing

Organization (microarchitecture on CPU).
- How data is being moved (buses architecture)
- How data is stored, e.g. Cache.
- …

Now it is time to learn about computer architecture.
How the computer brain, CPU, in the hardware works with the alphabet.
how the spirit of numbers goes to the body of the computer architecture

Boolean algebra

Logic Gates

x \ y	$\wedge$ 0	$\wedge$ 1
0	0	0
1	0	1

x \ y	$\vee$ 0	$\vee$ 1
0	0	1
1	1	1

x \ y	$\rightarrow$ 0	$\rightarrow$ 1
0	1	1
1	0	1

x \ y	$\oplus$ 0	$\oplus$ 1
0	0	1
1	1	0

Figure 1. Truth tables

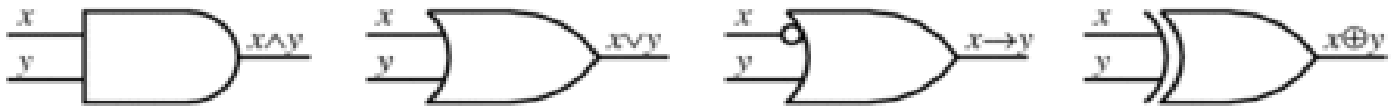


Figure 2. Logic gates

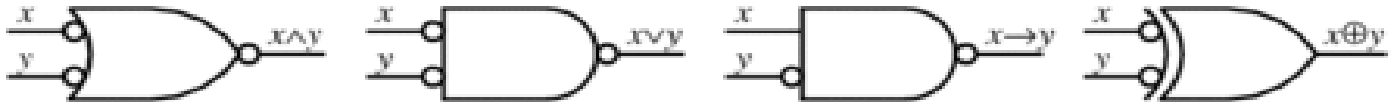


Figure 3. De Morgan equivalents

Figure 4. Venn diagrams

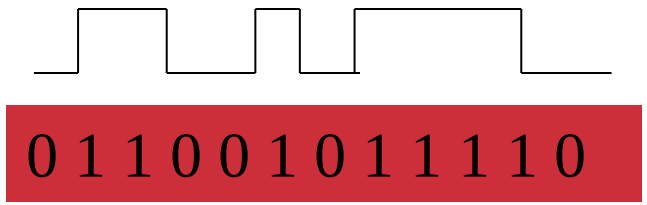
1	10
2	11
3	12
4	13
5	14
6	15
7	16
8	17
9	18

A logic gate is an idealized model of computation or **physical electronic device implementing a Boolean function**, a logical operation performed on one or more binary inputs that produces a single binary output.

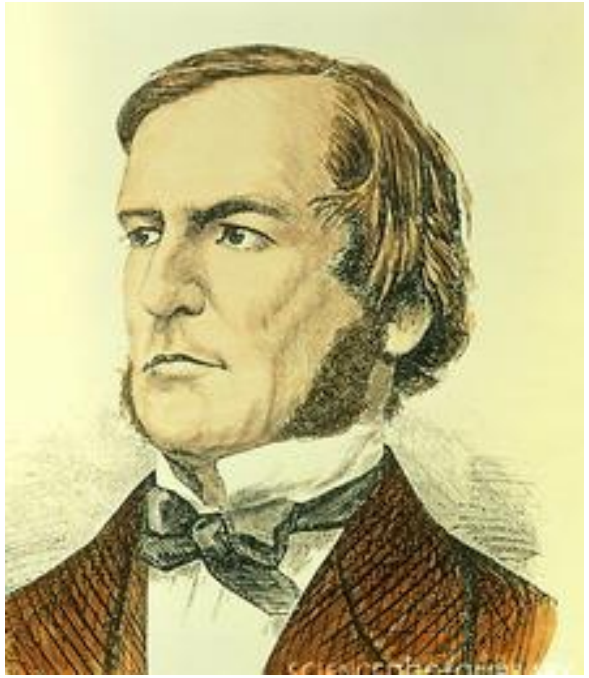
Bit and George Boole

Høyskolen Kristiania

- All logic in **digital** electronics is based on **bits**
- transistors are the building blocks
- Can be 0 or 1; high or low voltage.
- The logic used is called **Boolean Algebra**
  - **George Boole** (1815-1864)



Note! These 12 bits are normally just a snippet of a 32/64 bit word.

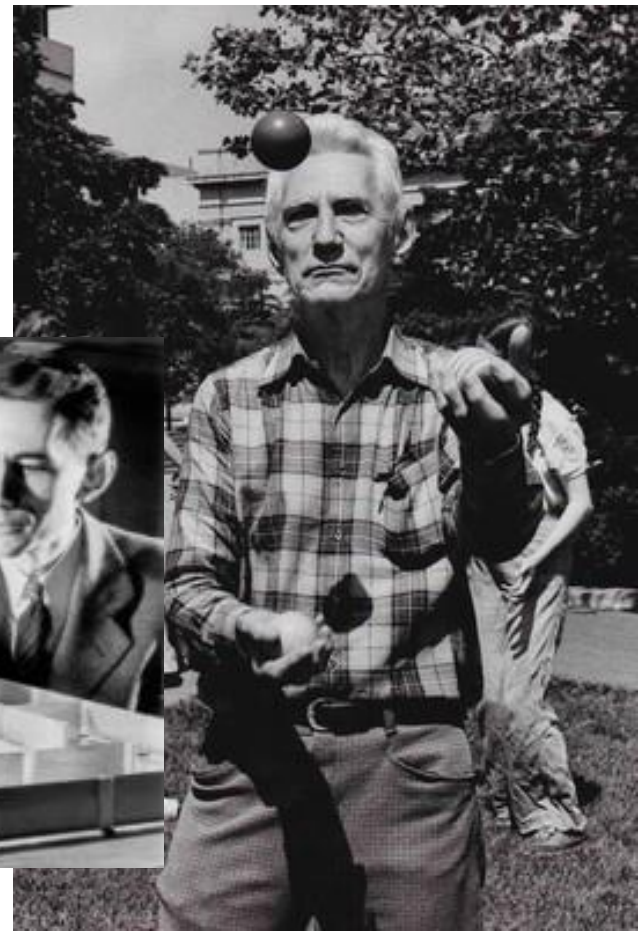


Boolean algebra

- Consists of 3 (4) basic operations (gates).
  - (NOT)
  - (AND)
  - (OR)
  - (XOR)
- Rediscovered in 1939 by Claude Elwood Shannon
- Computer electronics most often

Prefer «negative» logic

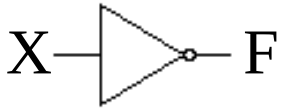
  - NAND, NOR, XNOR



he tried to solve the maze in one of the first experiments in artificial intelligence.

NOT

10010110

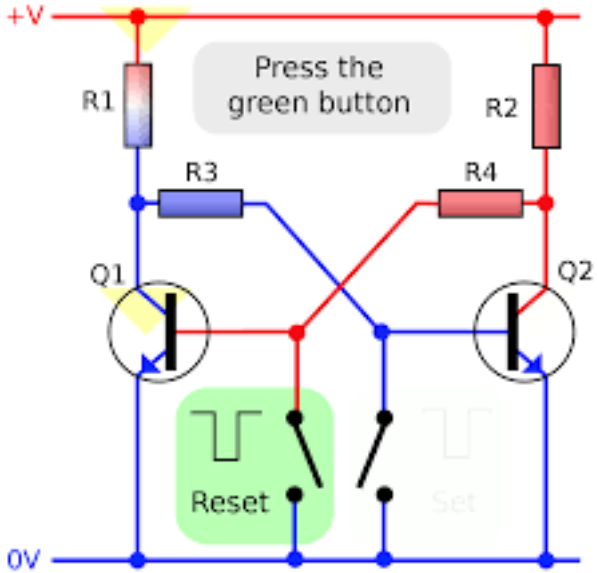


01101001

NOT

Truth table

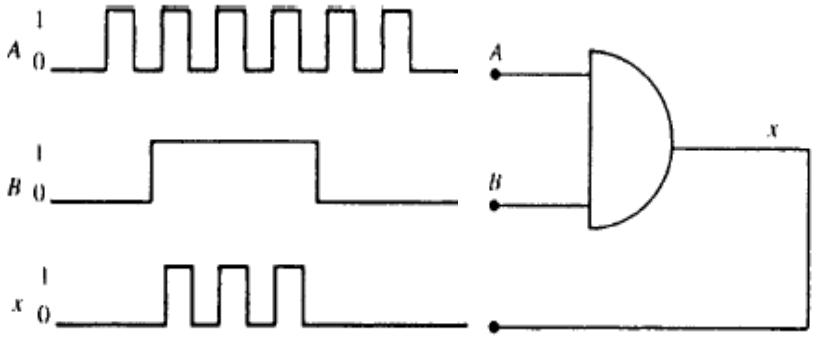
X	F
0	1
1	0



- Flip-flops and latches are fundamental building blocks of digital electronics systems.
- Flip-flops and latches are used as data storage elements.
- A flip-flop is a device which stores a single *bit* (binary digit) of data

https://upload.wikimedia.org/wikipedia/commons/a/a5/Transistor_Bistable_interactive_animated-en.svg

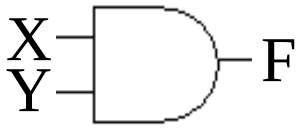
AND and NAND



Høyskolen Kristiania

10010101

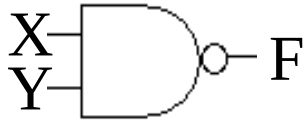
10101100



10000100

AND

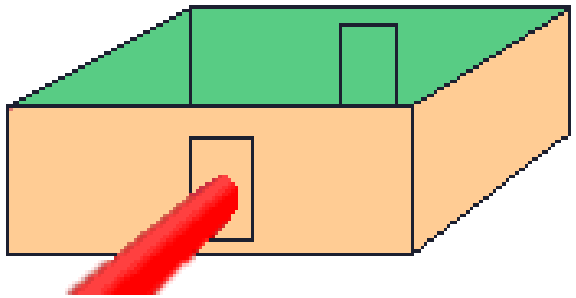
X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1



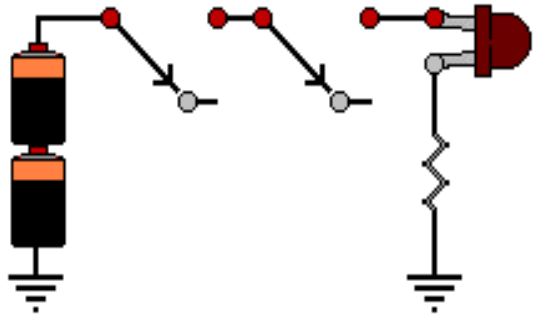
NAND

X	Y	F
0	0	1
0	1	1
1	0	1
1	1	0

Example: AND



A	B	C
0	0	0
0	1	0
1	0	0
1	1	1



OR and NOR

Høyskolen Kristiania

10111001

10001110

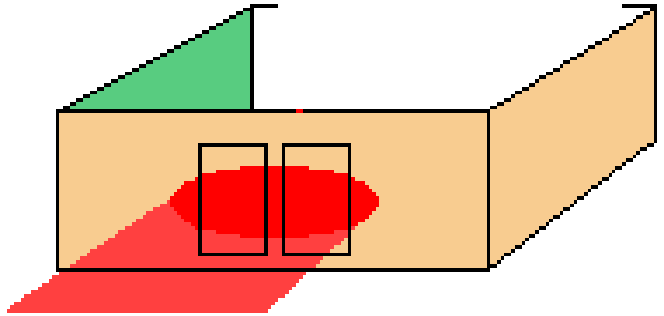
OR

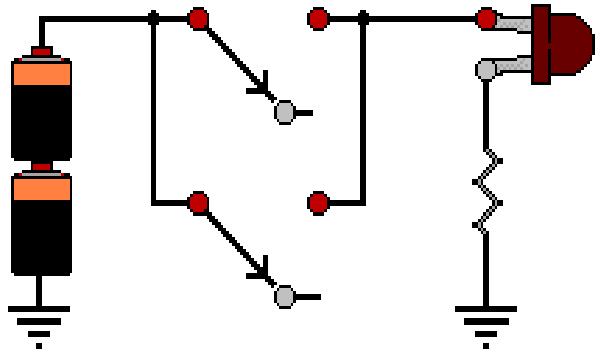
X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

NOR

X	Y	F
0	0	1
0	1	0
1	0	0
1	1	0

Example: OR



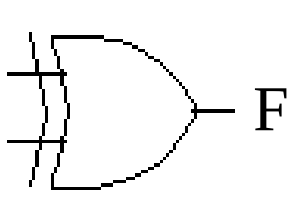


A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

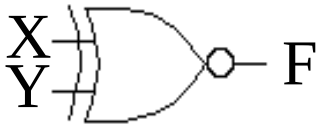
XOR and XNOR

10110010

11001110



01111100



XOR

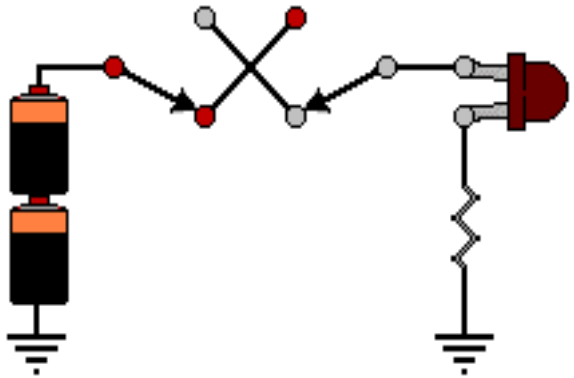
X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

X	Y	F
0	0	1
0	1	0
1	0	0
1	1	1

Example: XOR

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0



Examples of Boolean algebra using binary system
computer arithmatics

Binary Operations

- We learn the binary operations using truth tables

X	Y	AND
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	OR
0	0	0
0	1	1
1	0	1
1	1	1

X	NOT
0	1
1	0

X	Y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

- Given two bits, apply the operator
  - 1 AND 0 = 0
  - 1 OR 0 = 1
  - 1 XOR 0 = 1
- Apply the binary (Boolean) operators *bitwise* (in columns) to binary numbers as in
  - 10010011 AND 00001111 = 00000011

Examples

Høyskolen Kristiania

- AND – if both bits are 1 the result is 1, otherwise 0
  - 11111101 AND 00001111 = 00001101
  - 01010101 AND 10101010 = 00000000
  - 00001111 AND 00110011 = 00000011
- OR – if either bit is 1 the result is 1, otherwise 0
  - 10101010 OR 11100011 = 11101011
  - 01010101 OR 10101010 = 11111111
  - 00001111 OR 00110011 = 00111111
- NOT – flip (negate) each bit
  - NOT 10101011 = 01010100
  - NOT 00001111 = 11110000
- XOR – if the bits differ the result is 1, otherwise 0
  - 10111100 XOR 11110101 = 01001001
  - 11110000 XOR 00010001 = 11100001
  - 01010101 XOR 01011110 = 00001011

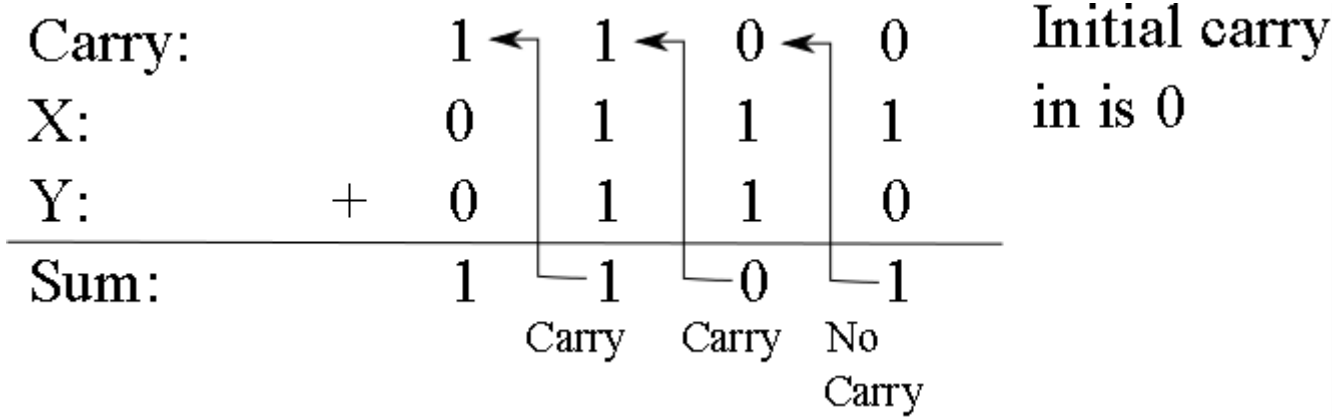
Binary Addition - (to remember)

Høyskolen Kristiania

- To add 2 bits, there are four possibilities
  - 0 + 0 = 0
  - 1 + 0 = 1
  - 0 + 1 = 1
  - 1 + 1 = 2 – we can’t write 2 in binary, but 2 is 10 in binary, so write a 0 and carry a 1
- To compute anything useful (more than 2 single bits), we need to add binary numbers
- This requires that we chain together carrys
  - The carry out of one column goes to be added in the other column to its left

Binary Addition Continued

- With 3 bits (the two bits plus the carry), we have 4 possibilities:
  - 0 + 0 + 0 = 0
  - 2 zeroes and **1 one** = 1
  - **2 ones** and 1 zero = 2 (carry of 1, sum of 0
  - **3 ones** = 3 (carry of 1 and sum of 1)
- Example:



Check your work, convert to decimal!

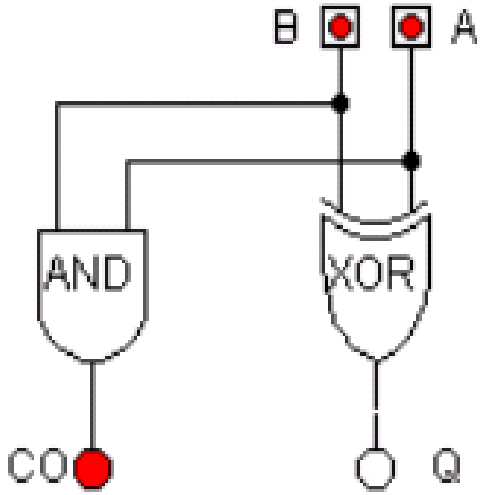
Addition Using AND, OR, XOR

- To implement addition in the computer, convert addition to AND, OR, NOT and XOR
- Input for any single addition is two binary numbers and the carry in from the previous (to the right) column
  - For column i, we will call these X_i, Y_i and C_i
- Compute sum and carry out for column i (S_i, C_{i+1})
- $S_i = (X_i \text{ XOR } Y_i) \text{ XOR } C_i$
  - Example: if 1 + 0 and carry in of 1
    - sum = (1 XOR 0) XOR 1 = 1 XOR 1 = 0
- $C_{i+1} = (X_i \text{ AND } Y_i) \text{ OR } (X_i \text{ AND } C_i) \text{ OR } (Y_i \text{ AND } C_i)$
  - Example: if 1 + 0 and carry in of 1
    - carry out = (1 AND 0) OR (1 AND 1) OR (0 AND 1) = 0 OR 1 OR 0 = 1

- Try it out on the previous problem

Practical example: Half adder

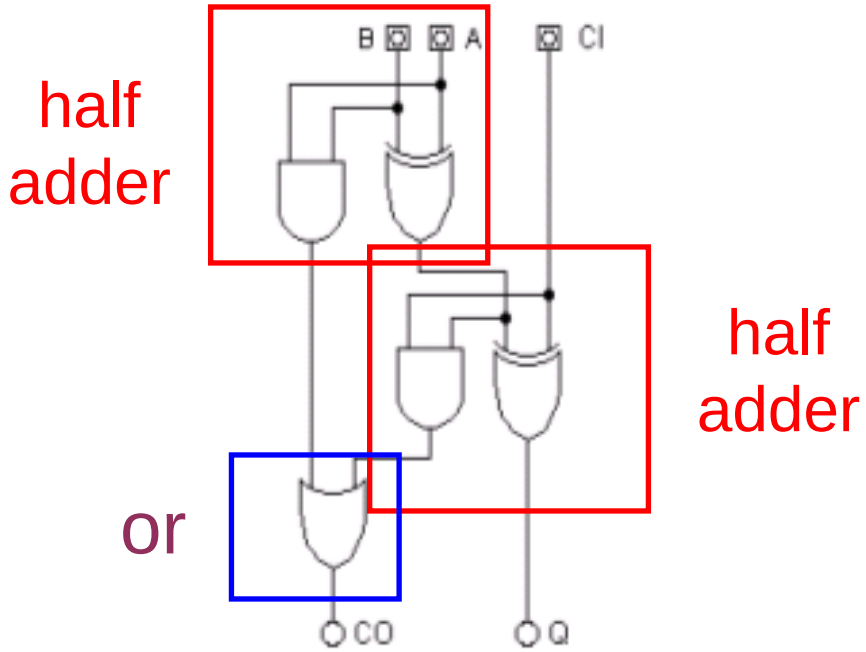
- Adds two bits, A and B
- Gets the answer in **Q** and carry in **CO**
  - Q = A XOR B
  - CO = A AND B



A	B	CO	Q
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

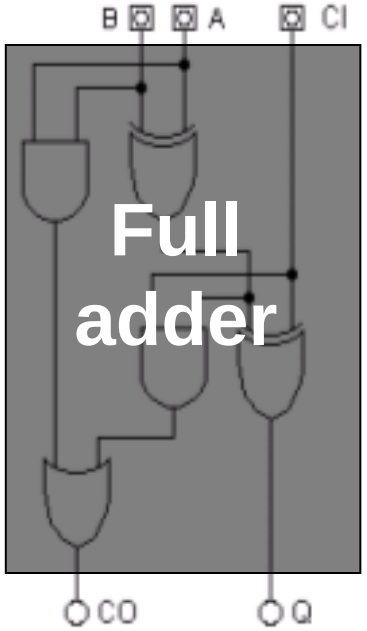
Practical example: Full adder

- Adds two bits together, **A** and **B**, and possibly **CI** (carry in)
- Gets the answer in **Q** with carry in **CO** (carry out)

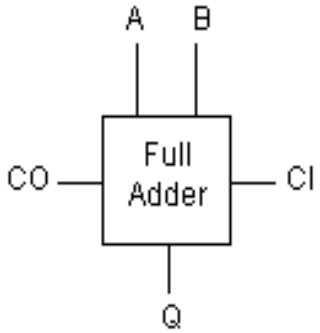


A	B	CI	Q	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

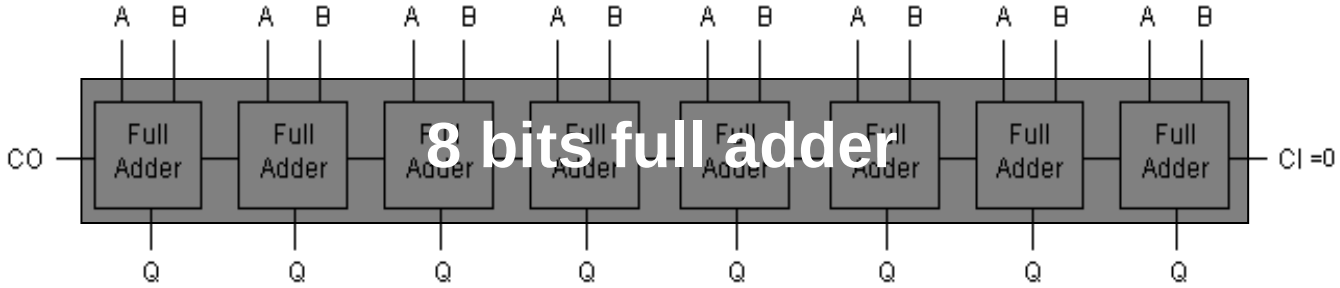
Practical example: 8 bit full adder



’black box’



Adds 1 bit numbers

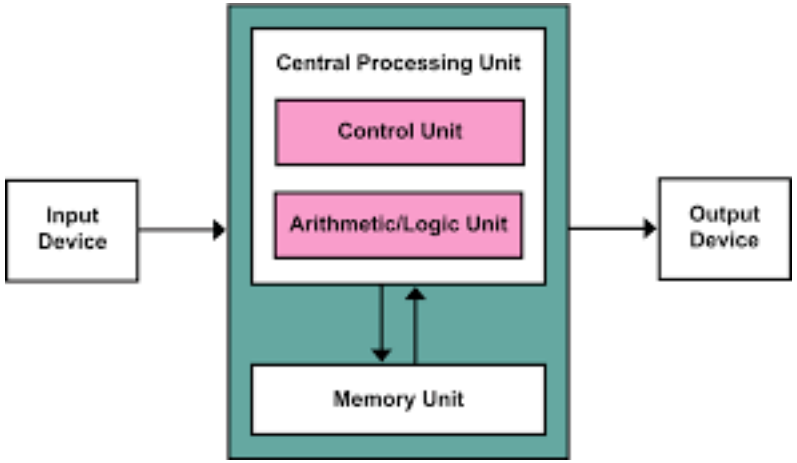


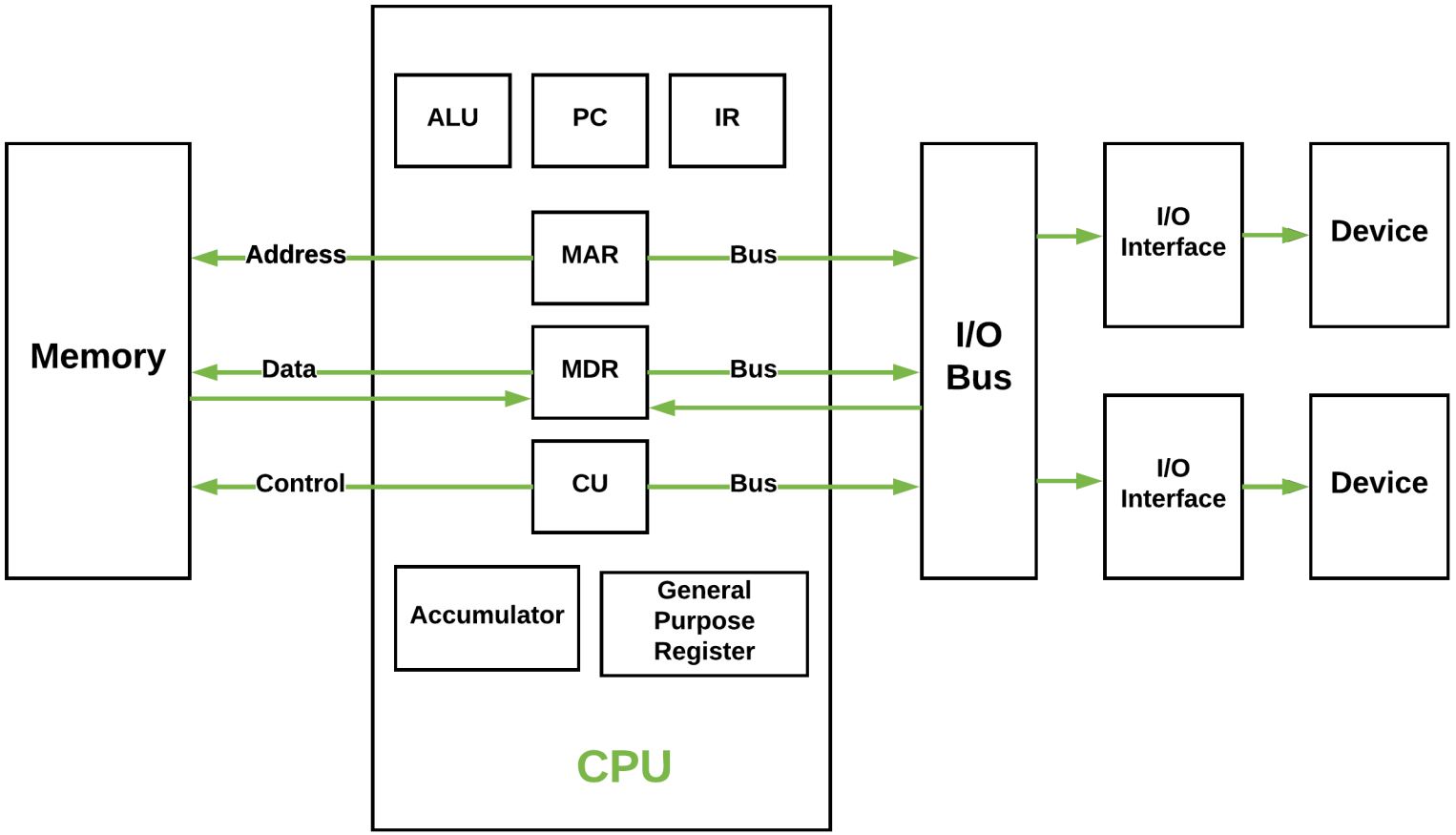
8 full adders connected together: Adds 8 bit numbers

Høyskolen Kristiania

Computer architecture

Von Neumann model





https://youtu.be/-SADbPS8UgA

Central Processing Unit

- Areas of use
- Working speed (clock frequency)
- Instruction set (ISA)
- Register
- Arithmetic Logical Unit and Float Point Unit
- Techniques to increase performance: pipelining and parallelization

Processor Classification

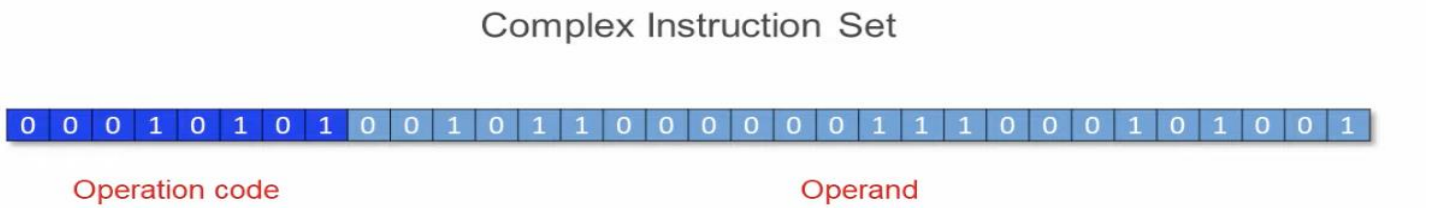
Høyskolen Kristiania

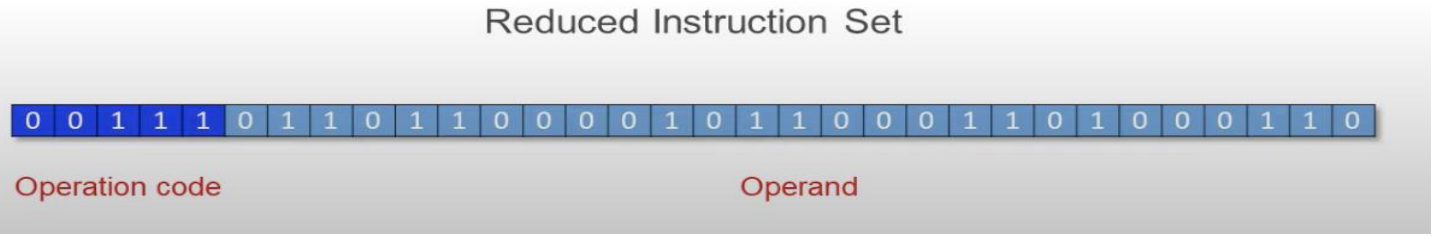
- Instruction Set Architecture (ISA)
  - RISC vs CISC
- Area of use
  - Embedded <-> Desktop <-> supercomputers
- Clock frequency
- **Registry file size**
- Number of functional units **arithmetic logic unit (**ALU) and floating-point unit (FPU)
- Superscalarity and parallelism
- Socket-type
- Energy consumption and cooling needs

Different Processor (CPU) Design Methodologies

Høyskolen Kristiania

- RISC vs CISC
- Complex Instruction Set Computing
- Reduced Instruction Set Computing





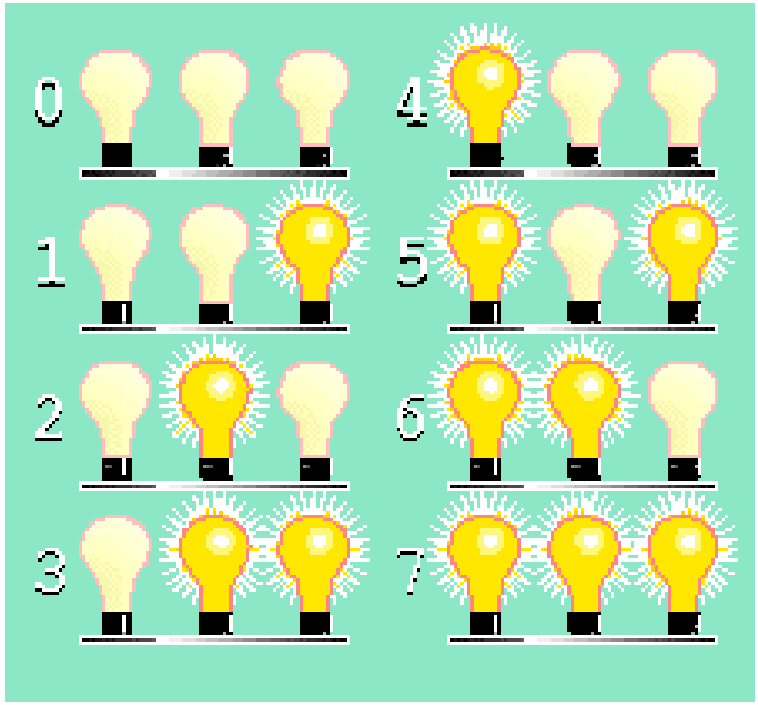
https://www.youtube.com/watch?v=6Rxade2nEjk&t=8s

Binary number representation (recall)

- With 3 lamps we can represent $2^3 = 8$ combinations of on / off
- With Off = 0 and On = 1 we get

0 = 000	4 = 100
1 = 001	5 = 101
2 = 010	6 = 110
3 = 011	7 = 111

- This can be extended to use more lamps (transistors), e.g. 8, 16, 32, 64, ...



Manufacturers-Instruction Set Architecture (ISA)

Høyskolen Kristiania

Mac book pro intel

- Intel (x86) and AMD
  - Complex Instruction Set Computing
  - Constructs and produses CPU, chipset etc.
  - Focus: **Desktop**, **server**, mobile, embedded.

other Mac apple M1 ARM

- MIPS(32 registers) and ARM (16 registers)(apple)
  - Reduced Instruction Set Computing
  - Constructs and licenses CPU cores etc. to various manufacturers
  - Focus: **Embedded**, **mobile**

- Now x86 and ARM processors bothe have features of RISC and CISC

Complex Instruction Set (CISC)

Reduced Instruction Set (RISC)

https://www.youtube.com/watch?v=6Rxade2nEjk&t=8s

Area of use

Høyskolen Kristiania

- Embedded systems
- IoT Devices
- Supercomputer
- Mobile phones use simple and "cheap" CPUs.
- Desktop and Laptops

Embedded and IoT systems

Høyskolen Kristiania

IoT vs Embedded Systems

	IoT	Embedded Systems
Definition	Things that contain computers, software and networking capabilities.	Things that contain computers and software.

- Made for specific purposes
- Controlled by a microprocessor /- controls
- The software is ready-made, often a single program, and is usually in firmware.
  - Can be configured frequently.
  - Does not (often) need OS.
- Example
  - Washing machines, home routers, clock radios, guitar tuners, smart cards, SIM cards,…

- In the last couple of years, the Arduino and Raspberry Pi have opened up for much simpler home development of various embedded systems.

https://www.youtube.com/watch?v=p40OetppIDg

Supercomputers

Kilde: http://top500.org/

Tianhe-2

- 3.120.000 cores
- 33.863 Tflop/s
- 17.808 kW
- 1.024.000 GB RAM
- Kylin Linux

The world's largest and fastest: CHINA

Mobile phones

- Traditionally, mobile phones use simple and "cheap" CPUs.
  - **ARM** or **MIPS** ISA and core
  - The mobile manufacturers then produce custom versions themselves.
- Intel has ambitions in the market.
- "Smart phones" are (perhaps) most similar to a full-fledged PC with its own GSM card

Desktop ≈ x86

Høyskolen Kristiania

- **x86**
  - x86 is a generic term for CPUs with the same set of instructions as Intel developed.
- Apple started with Motorola 68000 RISC CPU, continued with IBM / Freescale PowerPC CPUs.
  - Went over to x86 i 2005
- Intel and AMD
  - Still developing the x86 series.

Area of use

Høyskolen Kristiania

- Embedded systems
  - The dominant CPU architecture for embedded
  - ARM, which is always described as RISC.
  - Other players are x86 [CISC] and MIPS [RISC].
- IoT Devices
  - Most Arduinos (ARM, etc) are considered RISC
- Supercomputer
  - RISC offers some advantages over CISC and have even been used in the current world’s fastest supercomputer (February 10, 2020).
- Mobile phones use simple and "cheap" CPUs.
  - ARM or MIPS ISA and core
- Desktop and Laptops

x86 instruction set (CISC)

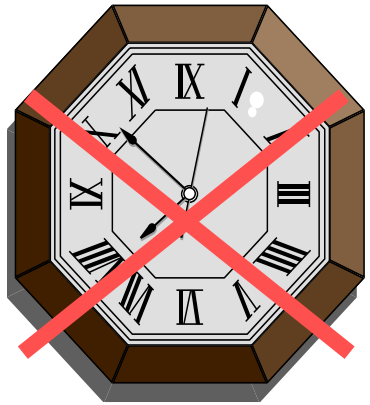
Høyskolen Kristiania

- Data relocation:
  - Move data between registers and to / from RAM.
  - move, push, pop, …
- Control transfer:
  - Perform condition statements (if, while) and method calls
  - je, jg, jmp, call, ret
- Arithmetic / Logic:
  - Perform operations on data in registers.
  - cmp, add, sub, inc, mul, imul, not, and, or, xor
- Input/Output: in, out
  - Writing / reading to ports (and memory addresses)
- Debug and Interrupt handling:
  - int, sti, hlt, nop
- **Floathing point numbers** has their own instructions
- **Single instruction, multiple data** (**SIMD**) vector and matrix instructions.
  - MMX, 3D Now!, SSE 1-4
- Separate instructions for changing the processor state (system)

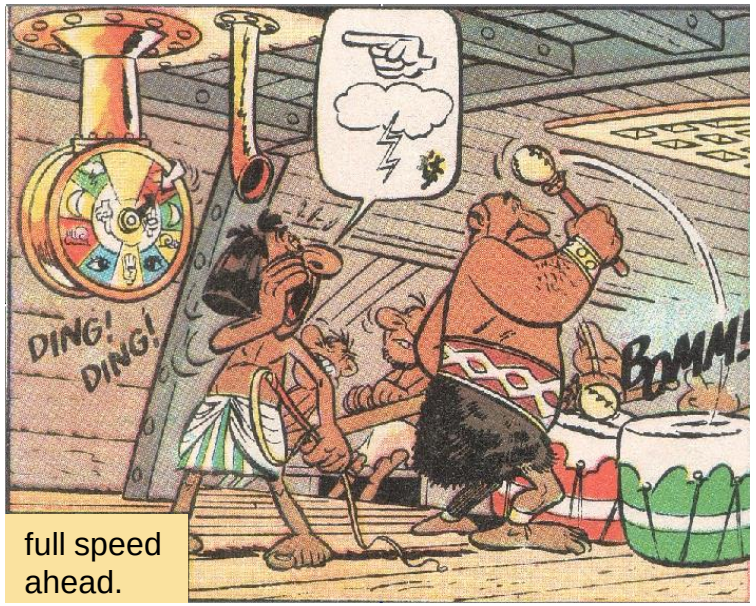
Techniques and concepts

System clocks (Hz)

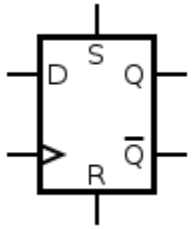
- The system clock keeps steps, does not show the time!

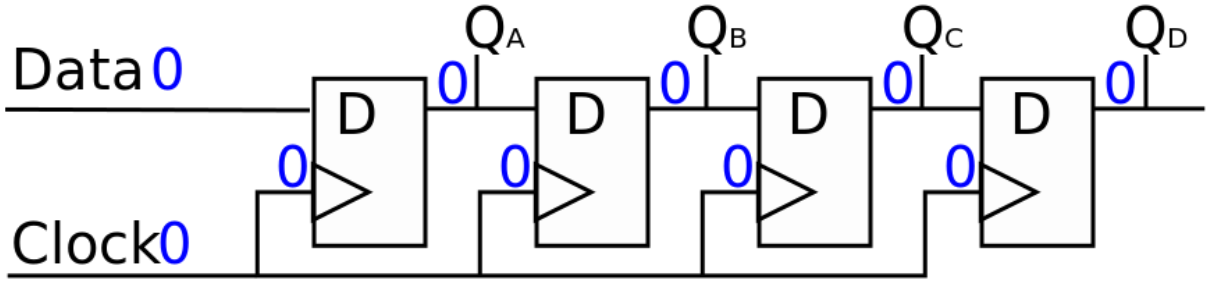


3.2 GHz executes 3.2 billion cycles per second. (Older CPUs had speeds measured in megahertz, or millions of cycles per second.)



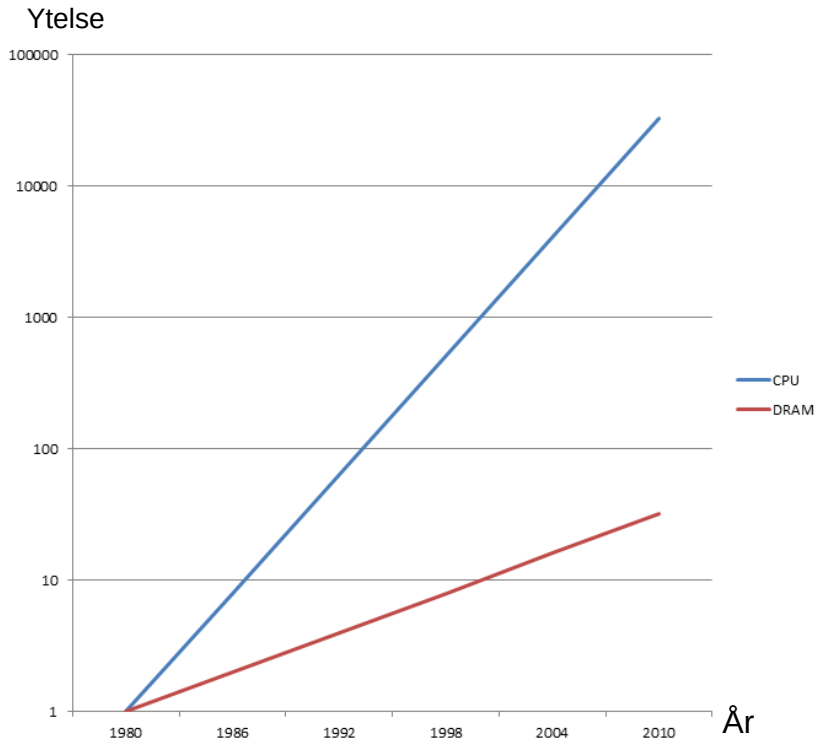
full speed ahead.





CPU vs RAM (DRAM)

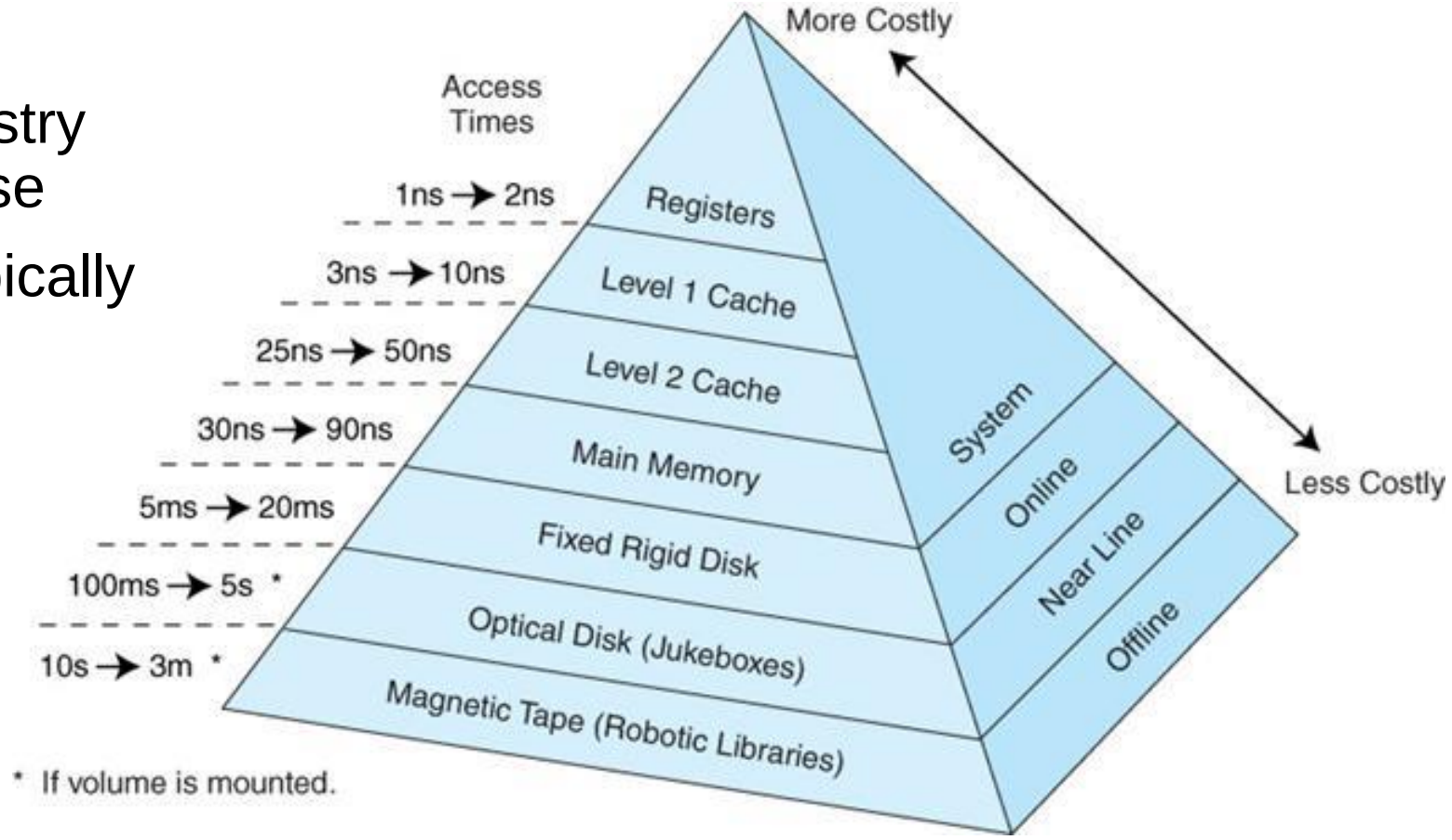
- CPU performance / speed have doubled every other year
- DRAM performance / speed every six years!



- The complete program is in RAM
- Only single instructions and single data are on the CPU
- -> PROBLEM !!!
- The challenge is to keep the pipeline full

Memory-hierarchy

- Doing something in a registry typically takes a clock pulse
- Reading / writing RAM typically 150-300.



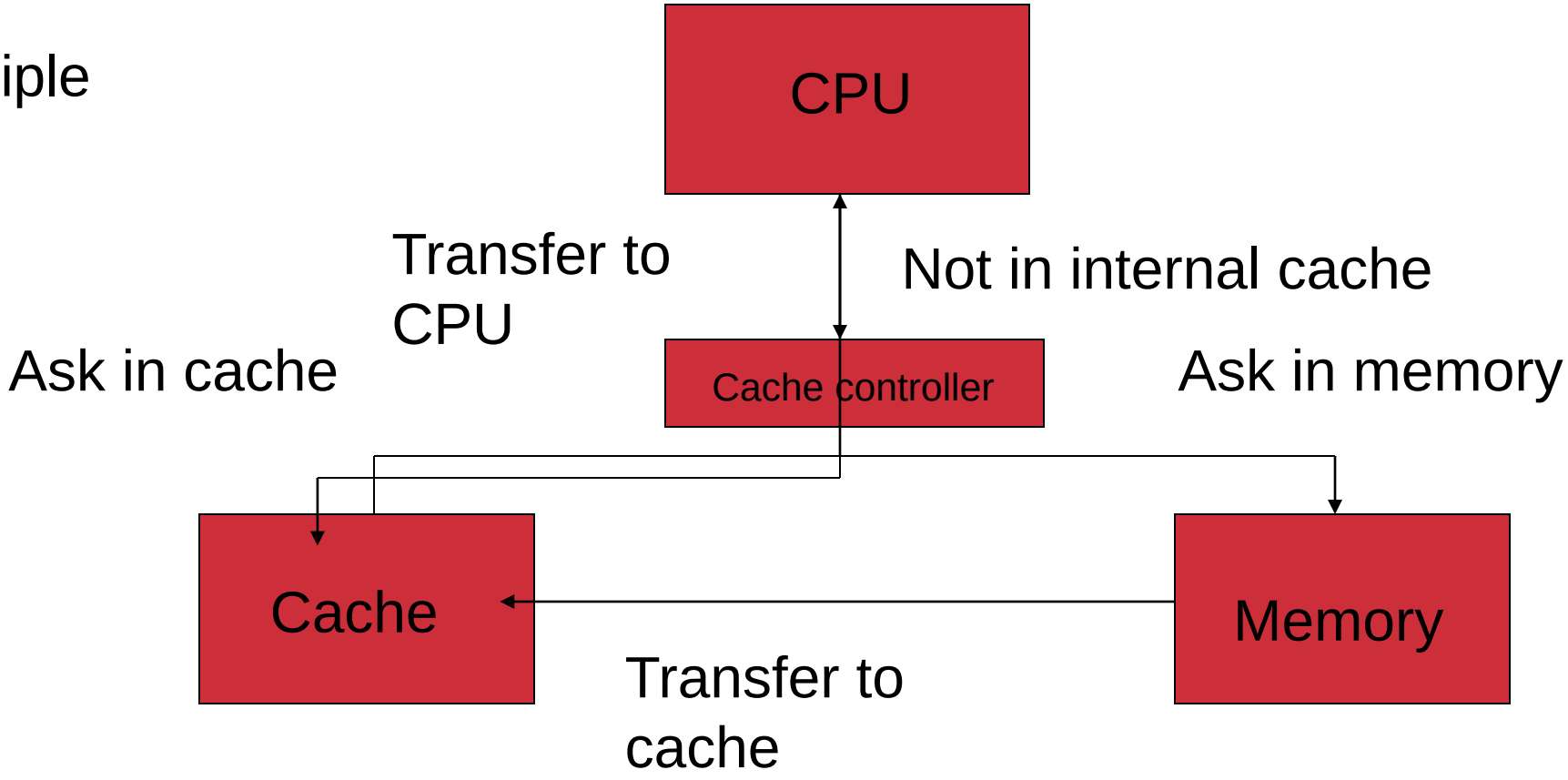
Cache

Høyskolen Kristiania

- The locality principle
  - About 5% of the code is run about 95% of the time.
  - Instructions to be executed are stored in the order in which they are going to be executed.
- Cacheing takes advantage of this by caching the most recently read instructions
  - Examples: keeping loops make it easy to access …
- + Pageing
  - We typically do not read individual instructions to Cache, but memory blocks of 4 KiB

Cache

- In principle



More "cores"

- A **multi-core processor** is a computer processor on a single integrated circuit with two or more separate processing units, called cores, each of which reads and executes program instructions
- The instructions are ordinary CPU instructions (such as add, move data, and branch) but the single processor can run instructions on separate cores at the same time, increasing overall speed for programs that support multithreading or other parallel computing techniques.
- The CPU contains several full-fledged "sub-CPUs", which can run software in parallel.
- you must have programs that can be parallelized and must not be run sequentially!

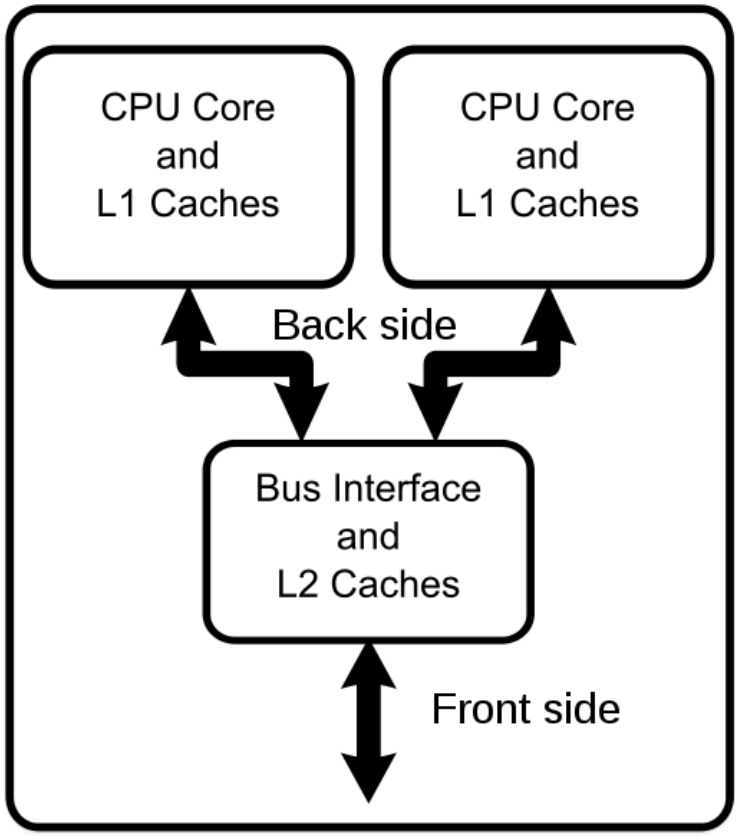


Diagram of a generic dual-core processor with CPU-local level-1 caches and a shared, on-die level-2 cache

Units and sizes

Bit (b) - 0 or 1
Byte (B) = 8 bits
Kilo = 10^3 = 1000 ≈ 1024 = 2^{10}
1 km = 1000 m, 1 mm = 1/1000 m
For the sake of simplicity, we "cheat" a little!

Heart (Hz) = events per second
MIPS = Mega instructions per second.
FLOPS = Mega floating point operations per second
kbps = 1000 bit / second (bit rate, «bandwidth»)

System of Units (SI)			Binary Numeral				%
Factor	Name	Symbol	Factor	Name	Symbol	# of Bytes	Difference
10^3	kilobyte	KB	2^{10}	kibibyte	KiB	1,024	2.4%
10^6	megabyte	MB	2^{20}	mebibyte	MiB	1,048,576	4.9%
10^9	gigabyte	GB	2^{30}	gibibyte	GiB	1,073,741,824	7.4%
10^{12}	terabyte	TB	2^{40}	tebibyte	TiB	1,099,511,627,776	10.0%
10^{15}	petabyte	PB	2^{50}	pebibyte	PiB	1,125,899,906,842,624	12.6%
10^{18}	exabyte	EB	2^{60}	exbibyte	EiB	1,152,921,504,606,846,976	15.3%
10^{21}	zettabyte	ZB	2^{70}	zebibyte	ZiB	1,180,591,620,717,411,303,424	18.1%
10^{24}	yottabyte	YB	2^{80}	yobibyte	YiB	1,208,925,819,614,629,174,706,176	20.9%

Differences on x86 16, 32 and 64 bit?

- 16 bit architecture
  - 16 bit register size (word)
  - 20 bit addressing (max 1 MiB «RAM»)
  - Segmented addressing (CS: IP e.g)
  - 16 bit words (typically 16 bit int e.g)
  - No distinction between running kernel code in OS and regular programs.
- 32 bit (386->Pentium)
  - 32 bit register (word size)
  - 32 bit addressing (4 GiB(**gibibyte**) Virtual Memory)
  - Flat and support for virtual memory
  - 32 bit words
  - Distinguish between kernel and user runtime.
- 64 bit (now)
  - 64 bit registers and multiple work registers
  - Typical 48 bit addressing (16 EiB(**exbibyte**) virtual memory)
  - Often multi-core
  - Even more instructions

A mebibyte is equal to 2^{20} or 1,048,576 bytes. A megabyte is equal to 10^6 1,000,000 bytes. **One mebibyte equals 1.048576 megabytes**.
The relatively small difference between them and so they are often used synonymously.

End

Point

- CISC
  - COMPLEX Instruction Set Computer
  - Many different instructions
  - Instructions have different lengths in bits / bytes
- RISC
  - REDUCED Instruction Set Computer
  - Only those instructions that (empirically) are used a lot.
  - Provides simpler electronics and thus faster execution.
  - See byte code on JVM.
- Moderne Intel/AMD CPU
  - "both"
  - CISC externally
  - RISC microinstructions on the processor itself

More points

- The processor executes the instructions in the order in which they are stored in memory («**sequentially**»).
- On **different data types** completely **different instructions** are used
  - 7 + 5:
    ```
    mov AX,7
    add AX,5
    ```
    results in 12 being in the AX register
  - 7.0 + 5.0:
    ```
    FLD «The memory address where 7.0 is located»
    FLD 0012AC45
    FADD ST, ST(1)
    FSTP «the address the answer should be added»
    ```
  - ‘Hei’ + ‘Hå!’
    loads of instructions that move both strings into the common memory area and add the end cursor (typically ASCII 0x00).

More points

Høyskolen Kristiania

- What is a variable??
  - At the instruction level: **the address** of a place memory (or a register).
- Why do variables have «scope»?
  - Calling a method means that you start running instructions somewhere else in memory
  - When you jump back to where the method was called, everything other than what the method returns disappears (typically an address to a value…)

What should we know?

Høyskolen Kristiania

- Boolean algebra
  - NOT, AND, OR, XOR
- What an instruction is.
  - how built up
  - what types
- What a register is.
  - the types of registers found on the CPU
  - The difference between address and answer register.
- Von Neumann model (again)
  - smaller «black box»
- The difference between CISC and RISC
- A little about x86 and options

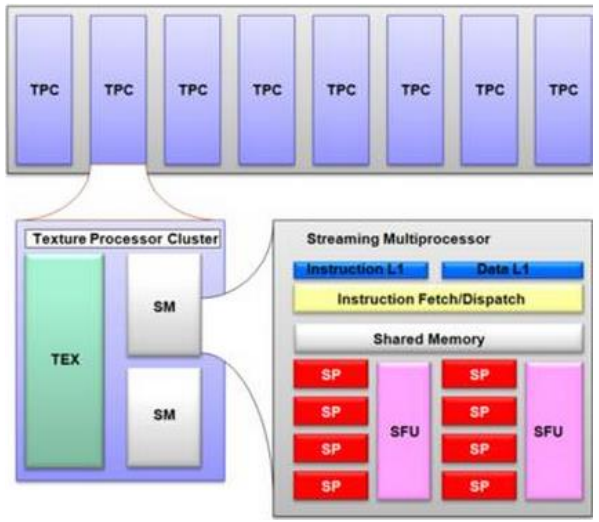
For optional self-study

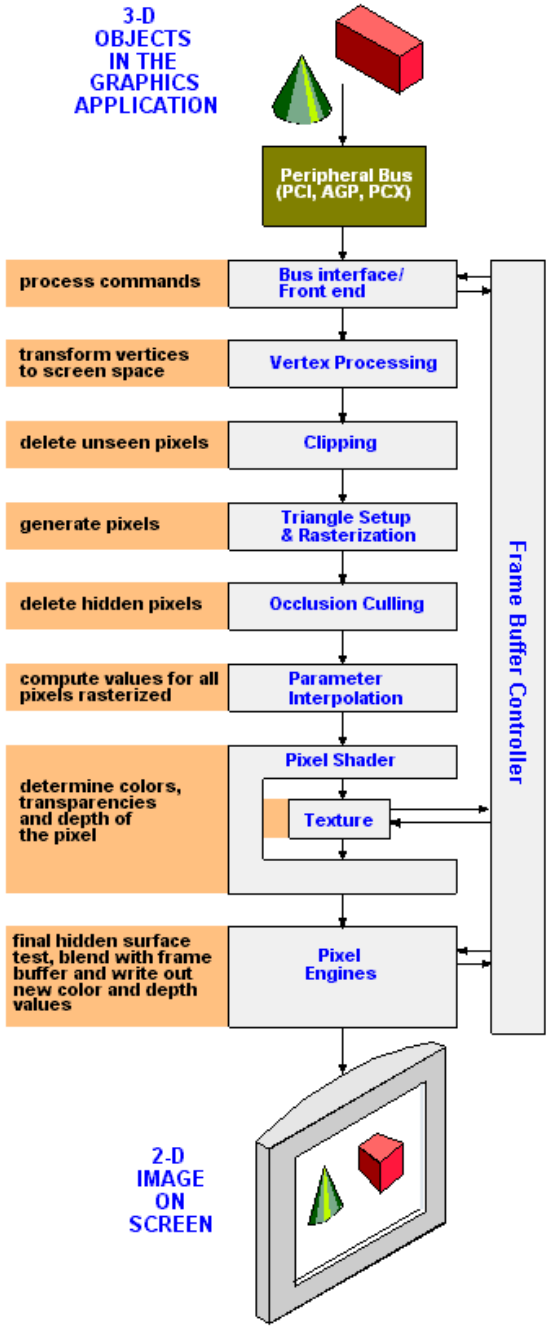
For those who want to learn some topics more in depth to better understand, here are some extra topics related to today's teaching, it must be expected some personal work to understand these topics.

There will be no questions on the exam from these, and this is therefore not considered to be part of the syllabus.

Graphics Processing Unit

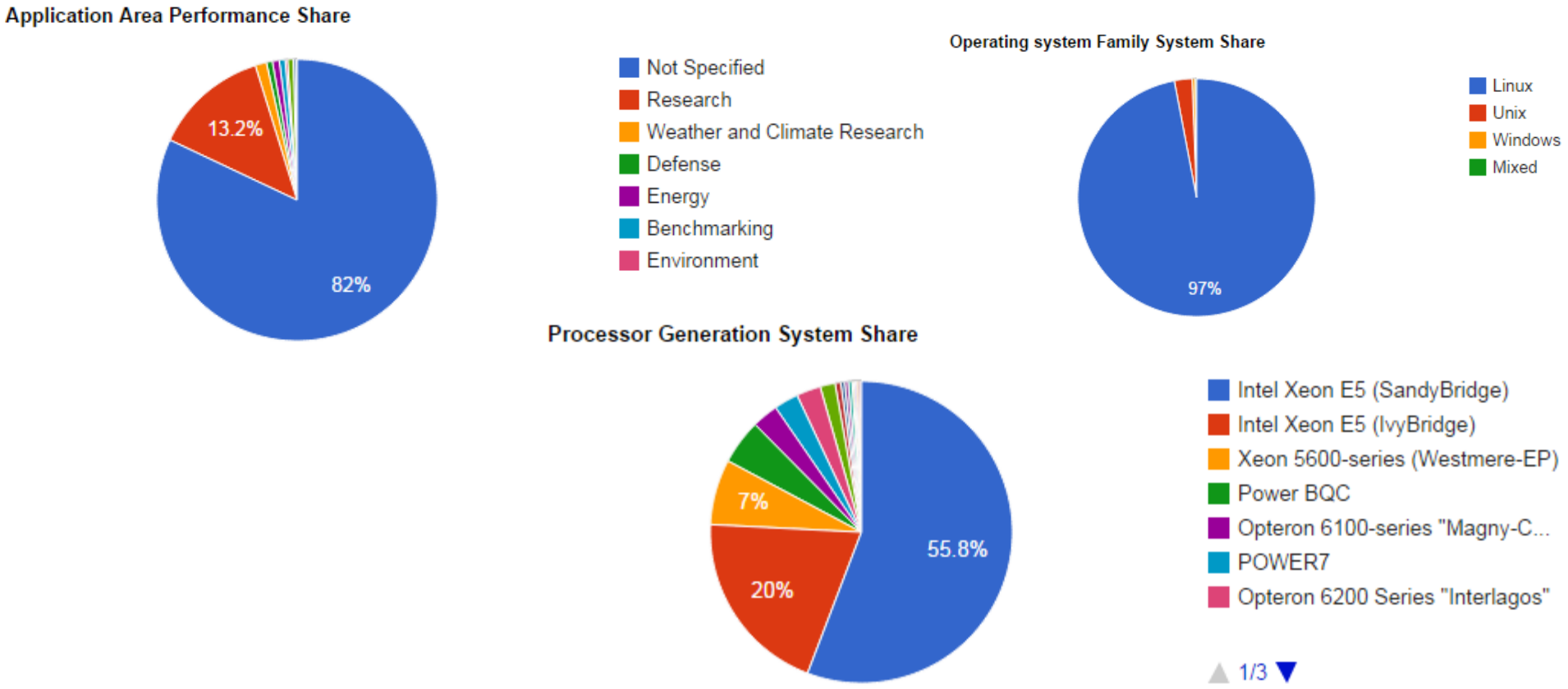
- Either integrated on motherboard, or mounted on graphics card
- Massively parallel
  - (now) ~ 100s of parallel arithmetic units
  - Calculates matrices (tables) of floating point numbers
  - Transforms mathematical models into screenshots (pixels in framebuffer)



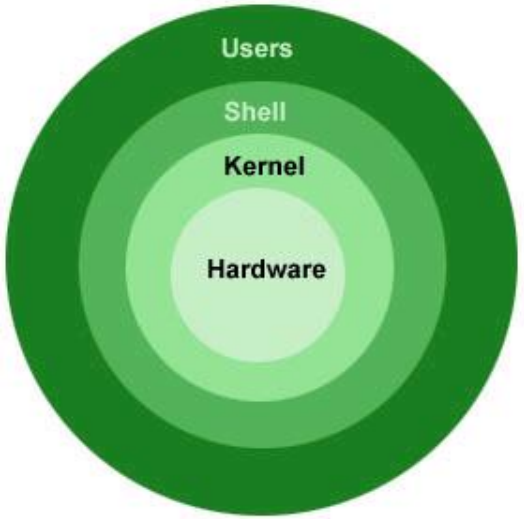


Heavy rain

As of June 2014, there were various Intel Xeon models, which run Linux in Clusters dominate



Software vs Hardware



Applications

- Shell + file manager
- User programs
- Compilers

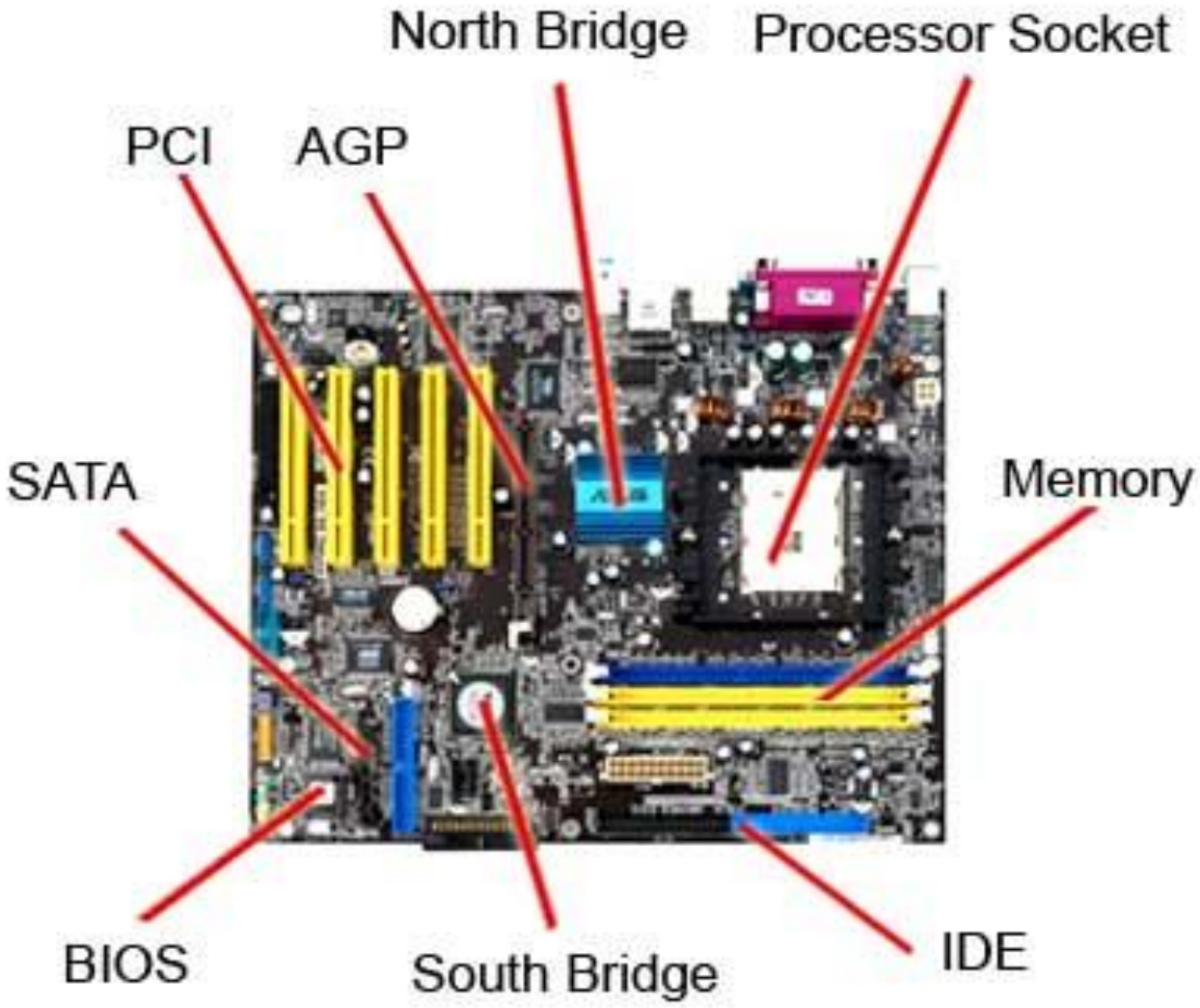
Hardware

- CPU, RAM, I/O-controllers
- Peripheral buses
- Additional card
- Hard disk …

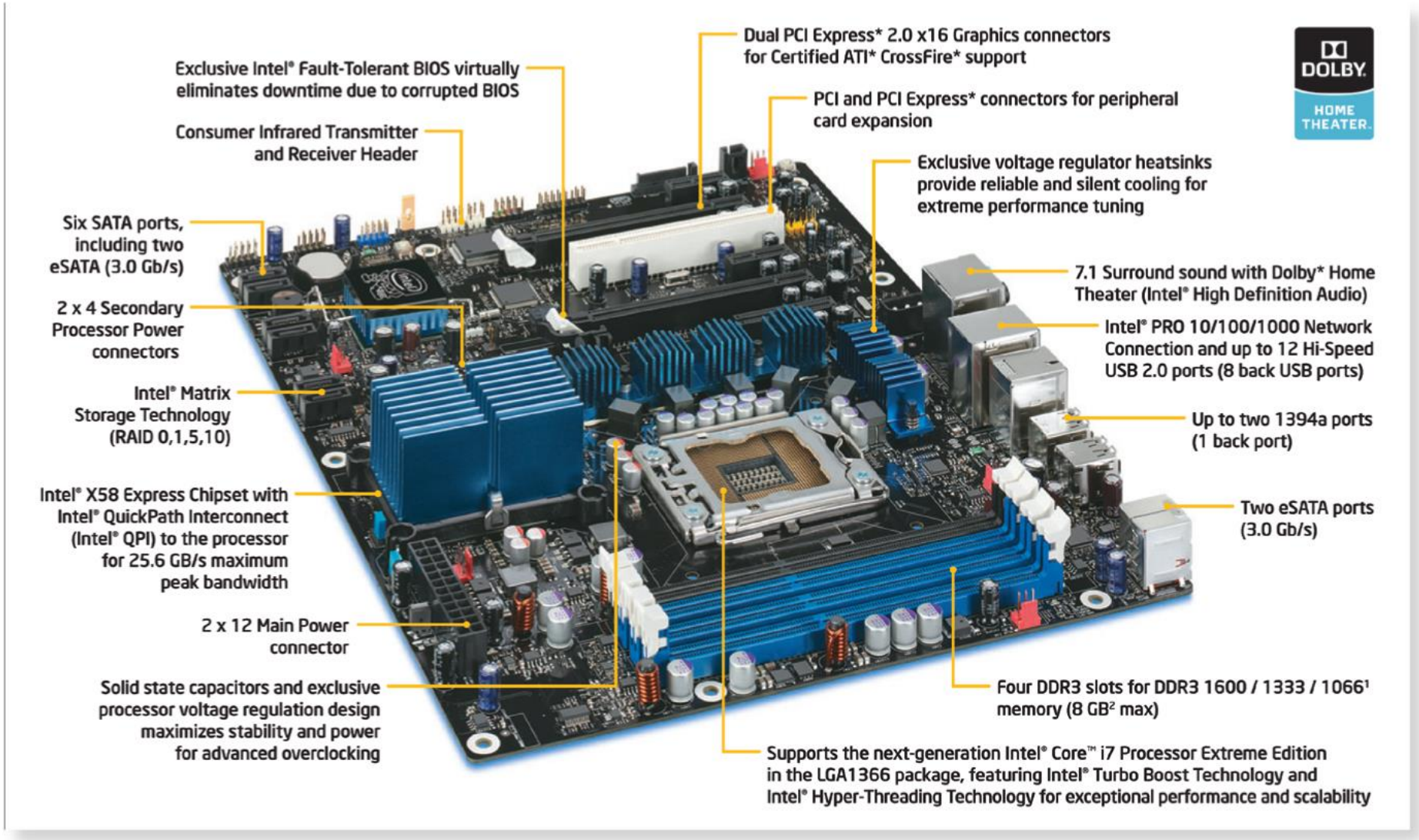
Operating system

- Manage access to CPU, Memory, Equipment and files.
- Makes it easier to use hardware.
- Abstraction layer

Motherboard approx 2004 (?)

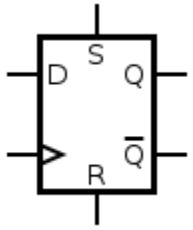


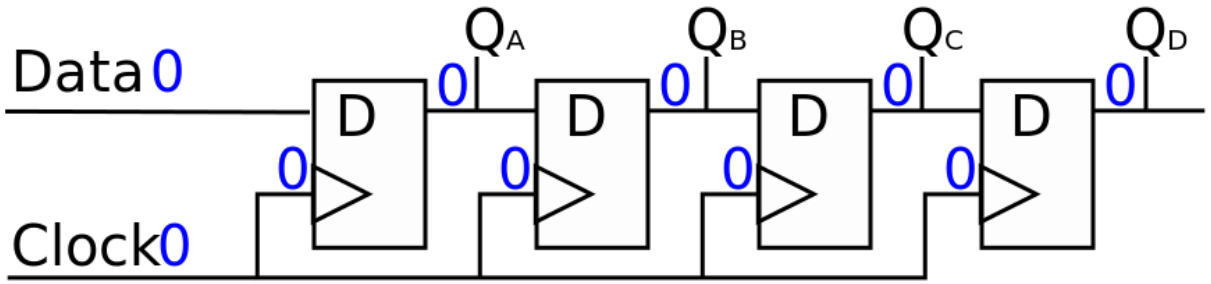
Motherboard 2012



Sequential vs. Combinatorial

- So far we have only looked at what is called combinatorial logic
- Output is determined solely by **Input** (and time).
- Sequential logic also has «memory»
  - Output is determined by **Input and stored bits.**
  - Used i.a. to build register (memory cells).





Boolean as data type

Høyskolen Kristiania

- In many contexts, there is no correspondence between the values 0 and 1 and Boolean data types.
  - Often used for statement logic, where we are interested in "true" and "false", not 1/0
  - The statement 4 == 4 has truth value true
  - The statement 3 == 5 has truth value false
  - How "true" and "false" are actually coded binary is language / compiler dependent.
- In Java (and other C-languages) you have different logical operators depending on whether it is truth value or bit-combination you want to perform.
  - & er bitwise, && are logical
    - Thus becomes **3&4 = 0** (0011 & 0100 = 0000),
    - But **3&&4** gives «true», «syntax error», or 4…

Programmering

Høyskolen Kristiania

Bitwise AND	writes	&
Bitwise OR	writes	|
Bitwise XOR	writes	^
Bitwise NOT	writes	~
Logical NOT	writes	!
Logical AND	writes	&&
Logical OR	writes	||

LET US DESIGN A SIMPLE COMPUTER!

Høyskolen Kristiania

- SIMPLE!!! (Understand the principles)
- Primitive
- We do not make electronics
- The machine can still do some things
- Further description can be found in **Compendium 1**

What was architecture again?

Høyskolen Kristiania

- How to build the instructions that the computer should execute (execute)
- What types of instructions should we have?
- How should addressing take place?

- Once we have determined the **I**nstruction **S**et **A**rchitecture, we can design a CPU

- In practice, this is an interaction between machine design and ISA design
- We have a transistor budget

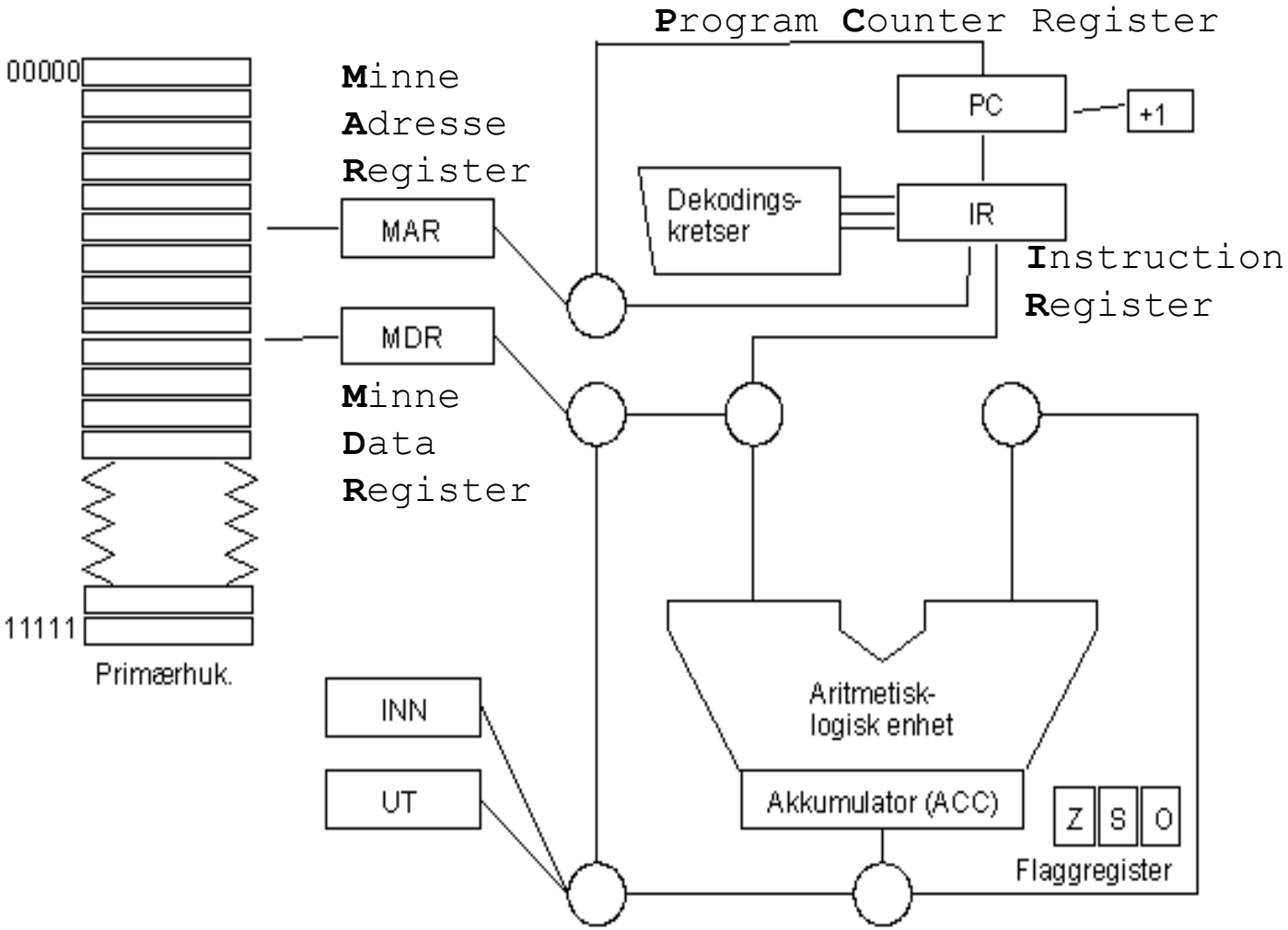
Register

Høyskolen Kristiania

- All bits on the CPU must be entered in a register
- A register is addressable memory (SRAM) that resides on the CPU.

- In Assembly language, the registers typically have their own "names" and roles
- The registers used to manipulate variables and values are referred to as work registers.
- Example. In X86-64
  - **RIP** is the register that contains the address of where in memory the next instruction is to be retrieved from.
  - **RAX** is the register most often used to store results in (accumulator)
  - **RCX** most often used for countdown (think for-loop)
  - + many more (memory processing, driving,…)

PrimRisc v. 0.1 layout



- Accumulator-based
- Only one work register
- 8 bit addressing
- Uses only 5
- Load / store
- Everything goes via RAM
- Single
- Can "really" just add

Register-«file»

- **PC (Program Counter).**
- This register will at all times contain the address of the location in the primary warehouse from which the next instruction is to be retrieved.
- **IR (Instruction Register).**
- This register stores instructions after they have been retrieved from the primary memory and before being sent to the decoder circuits.
- **MAR (address register).**
- When we want to read something from, or put something in the primary warehouse, we must first place the address in MAR.
- **MDR (data register).**
- When we do a read operation from the primary memory, the result is placed here. If we are to write to the primary memory, we must first place what we are to write in MDR
- **ACC (Accumulator).**
- This register is used primarily in connection with calculation operations. It is also possible to read the contents of a memory address into the accumulator with the instruction LOAD, and save the contents of the accumulator with the instruction STORE.
- **FLAG register**
- we need it in order to be able to make conditional jumps (if, while and the like)
- **The Z flag** (Zero flag) is set to 1 if the result of a calculation operation is zero.
- **The S-flag** (Sign-flag) is set to 1 if the result of a calculation operation is negative.
- **The O-flag** (Overflow-flag) is set to 1 if the result of a calculation operation becomes too large (overflow) to fit in the ACC register

Instructions

Mnemonic	Opkode	Beskrivelse	Funksjon
STOP	0000	Stopp maskinen	
LOAD	0001	Les inn i akkumulator	(ACC) ← (MDR)
STORE	0010	Lagre akkumulator	(MDR) ← (ACC)
ADD	0011	Adder til akkumulator	(ACC) ← (ACC) + (MDR)
SUB	0100	Subtraher fra akkumulator	(ACC) ← (ACC) - (MDR)
BRANCH	0101	Hopp uansett	(PC) ← IR<4:0>
BRZERO	0110	Hopp hvis resultat = 0	(PC) ← IR<4:0> eller (PC) ← (PC) + 1
BRNEG	0111	Hopp hvis resultat < 0	(PC) ← IR<4:0> eller (PC) ← (PC) + 1
BROVFL	1000	Hopp hvis overflyt	(PC) ← IR<4:0> eller (PC) ← (PC) + 1
IN	1001	Les inn fra tastatur	(ACC) ← INN
OUT	1010	Skriv til skjerm	UT ← (ACC)

- All instructions are 16 bit (2 bytes)
- 1 byte to encode
- Uses only 4 bits (in this version)
- 1 byte in the instruction to addresses
- Uses only 5 bits (in this version)

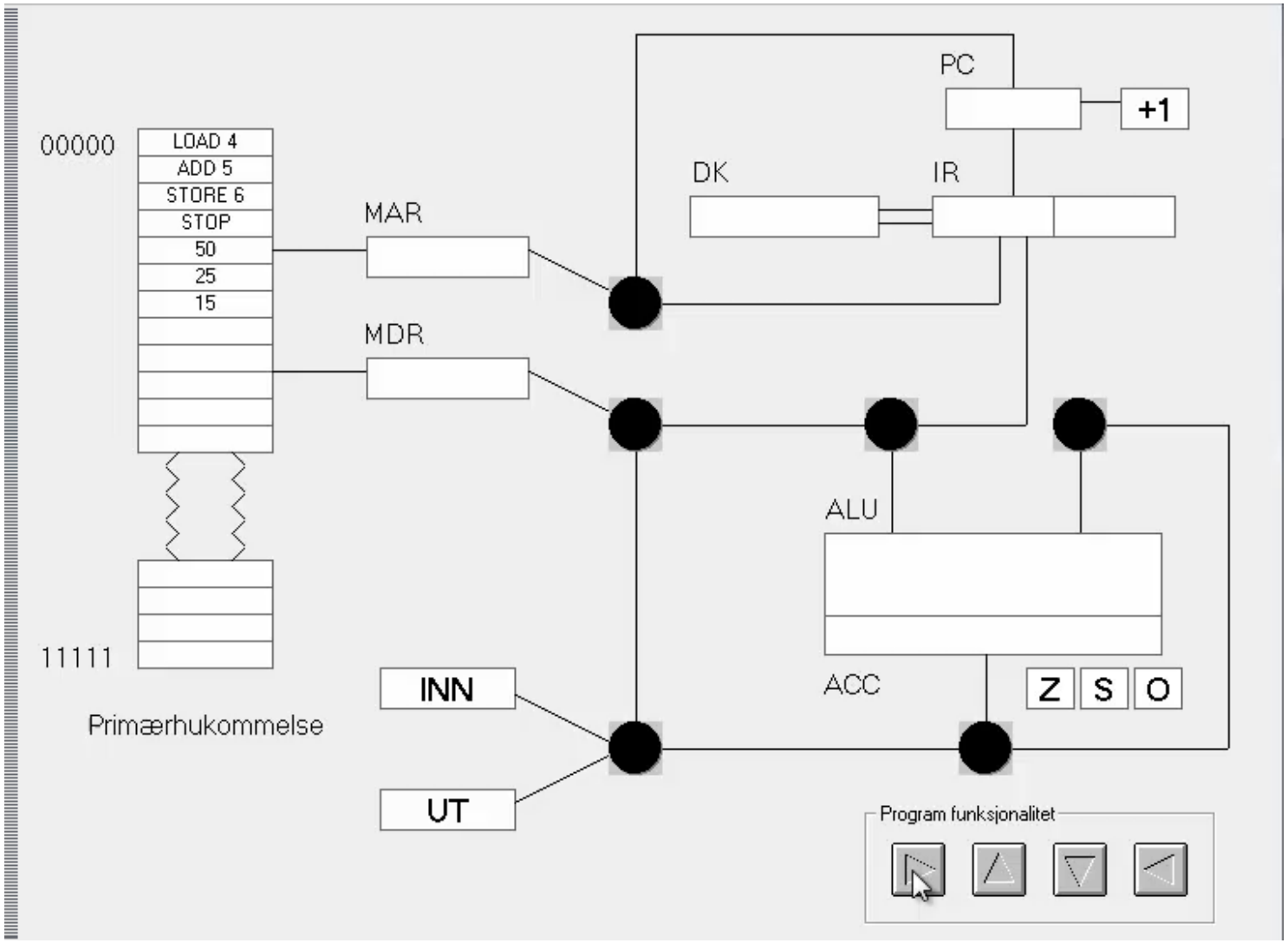
Example of a program in memory

Minneadresse binært/(desimalt)	Maskinkode (opkode+adresse) binært/(desimalt)	Asembly-instruksjon
00000 (0)	0001 0000 0000 0100 (1)(4)	LOAD 4
00001 (1)	0011 0000 0000 0101 (3)(5)	ADD 5
00010 (2)	0010 0000 0000 0110 (2)(6)	STORE 6
00011 (3)	0011 0000 0000 0000 (0)(-)	STOP
00100 (4)	0000 0000 0000 0001 (0)(1)	
00101 (5)	0000 0000 0000 0010 (-)(2)	
00110 (6)	0000 0000 0000 0000 (-)(0)	

```
/* Load in the accumulator data from address 4,
add data from address 5, store at the address
located at address 6,
End the run */
```

- *Can use a simulator of this architecture (buggy) on today's exercise* (Take a look at http://youtu.be/igthwn_qGVI)

Test


PC
+1
DK
IR
00000
LOAD 4
ADD 5
STORE 6
STOP
50
25
15
MAR
MDR
ALU
11111
ACC
Z
S
O
INN
UT
Primærhukommelse
Program funksjonalitet

Analysis / Criticism of PrimRISC 1

Høyskolen Kristiania

- Ready to demonstrate a Fetch-Execute cycle and how instructions can be structured
- Missing Pipelining
- Poor instruction set
- 8 against x86-64 sine ~ 1000
- Does not support System Programming (OS), multitasking or virtual memory
- «Quite small, but still so simple»

Criticism 2

Høyskolen Kristiania

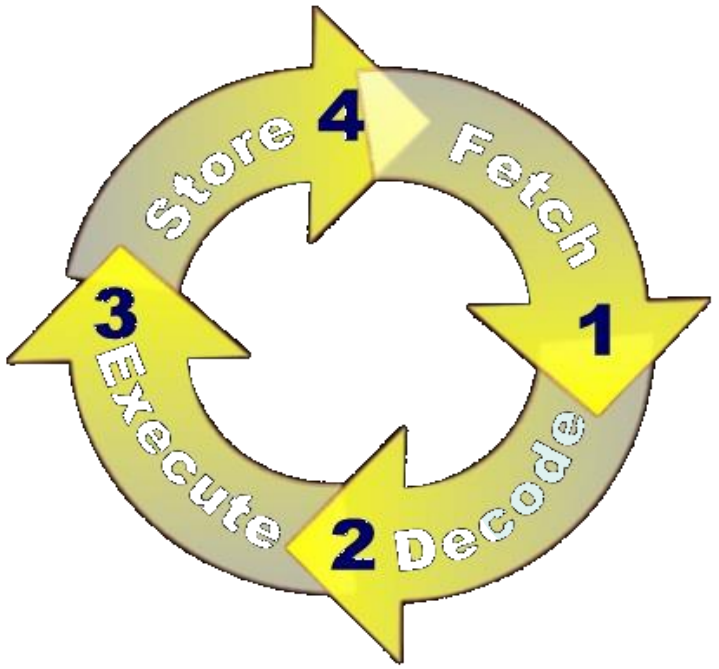
- Has support for conditional jumps
- Can implement if, for, while etc.

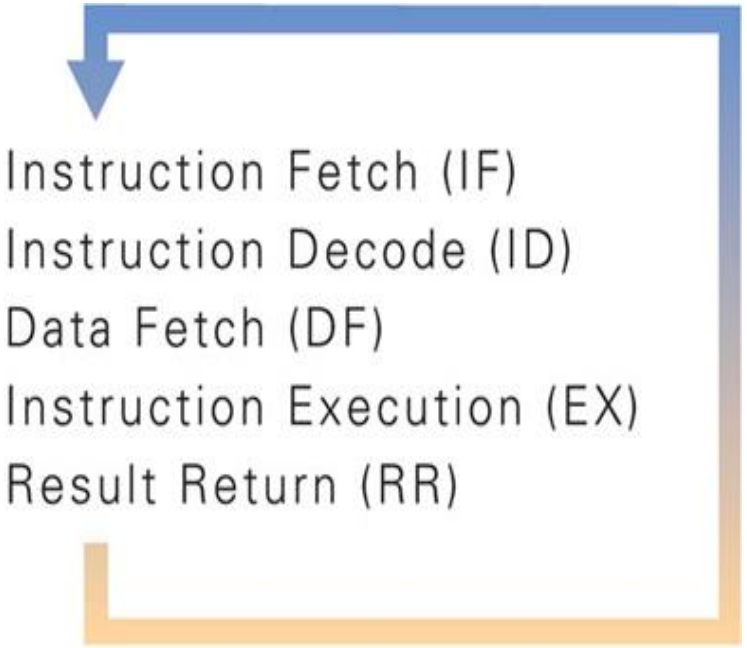
- Missing (proper) HW support for methods !!!
- In Intel / AMD, this is solved with separate registers (SS and SP) and instructions (call) that support the use of a separate memory area as a stack.

- Does not support interupt!
- More about this in the upcoming lectures.

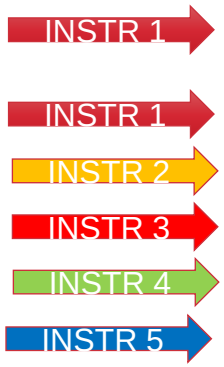
Fetch Execute Cycle

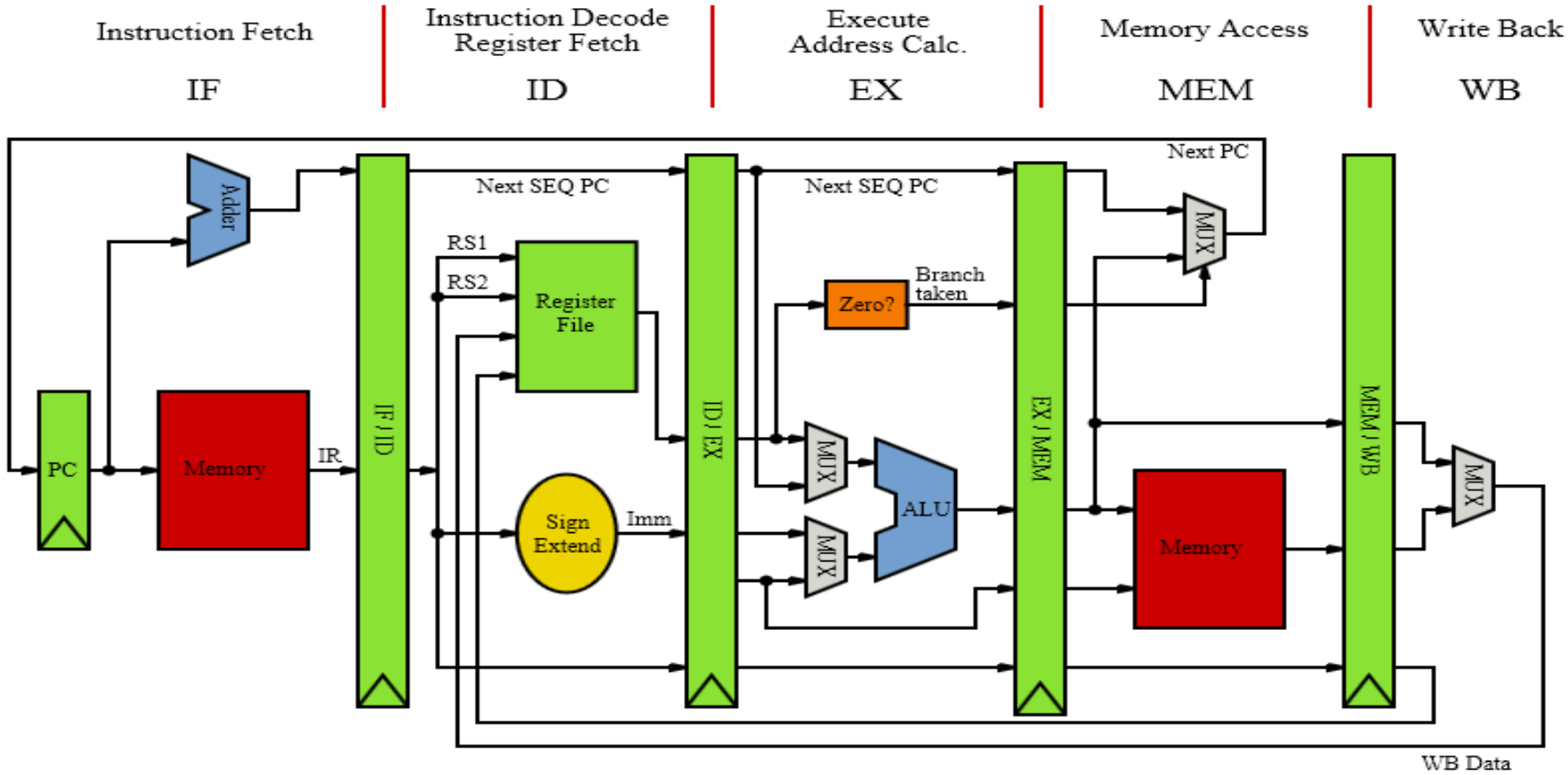
1. **Fetch**: retrieve instruction from memory and into registry
2. **Decode**: Interpret the instruction and set up the circuits that execute it
3. **Execute**: Execute the instruction on data / registers (change state)
4. **Store**: Save result (back in memory)



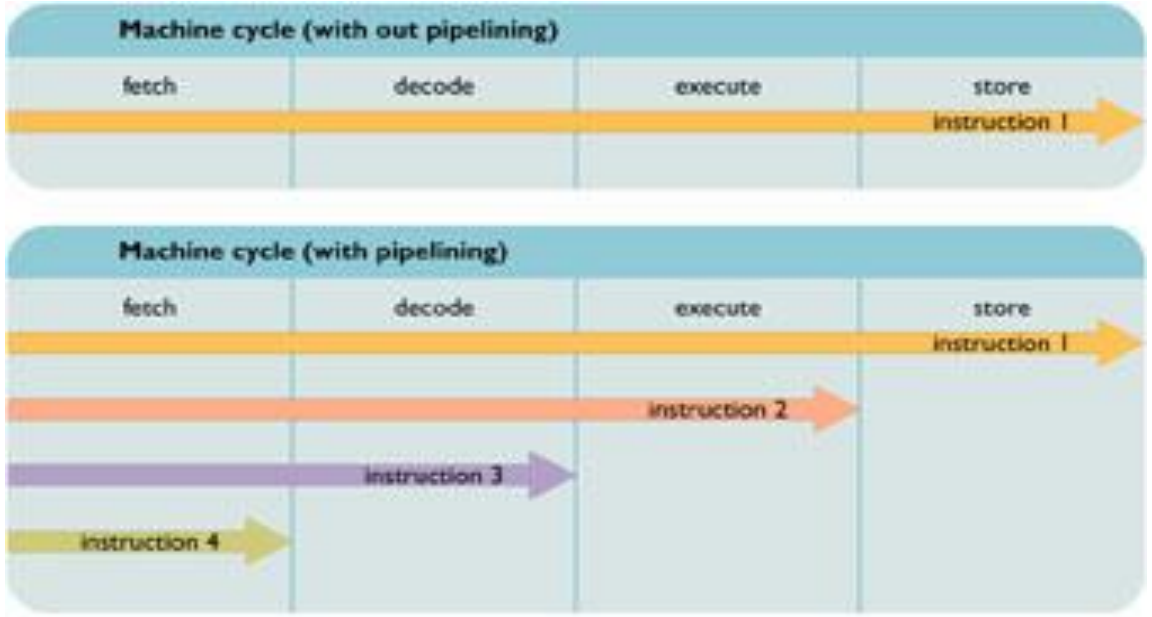


Ex: MIPS


INSTR 1
INSTR 1
INSTR 2
INSTR 3
INSTR 4
INSTR 5

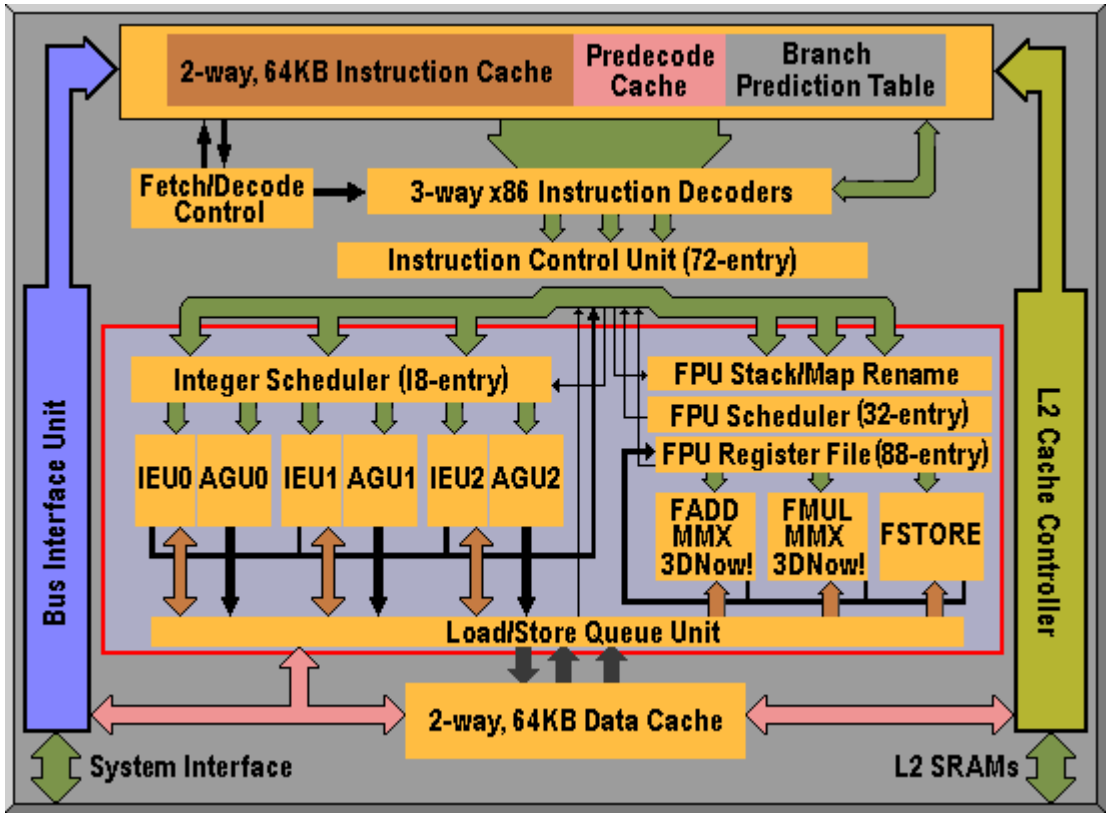

Instruction Fetch
IF
Instruction Decode
Register Fetch
ID
Execute
Address Calc.
EX
Memory Access
MEM
Write Back
WB
Adder
Next SEQ PC
Next SEQ PC
Next PC
MUX
RS1
RS2
Register
File
Zero?
Branch
taken
PC
Memory
IR
IF / ID
ID / EX
EX / MEM
MEM / WB
MUX
MUX
ALU
Sign
Extend
Imm
Memory
MUX
WB Data

Pipelining



- Since there are different devices on the CPU that retrieve, decode, execute and store these can be run in parallel ("assembly line")
  - Can start retrieving a new instruction as soon as it is left for decoding.
- In modern Intel / AMD CPU there are «deep and long» pipelines.
  - 15-25 steps ->

Super scale

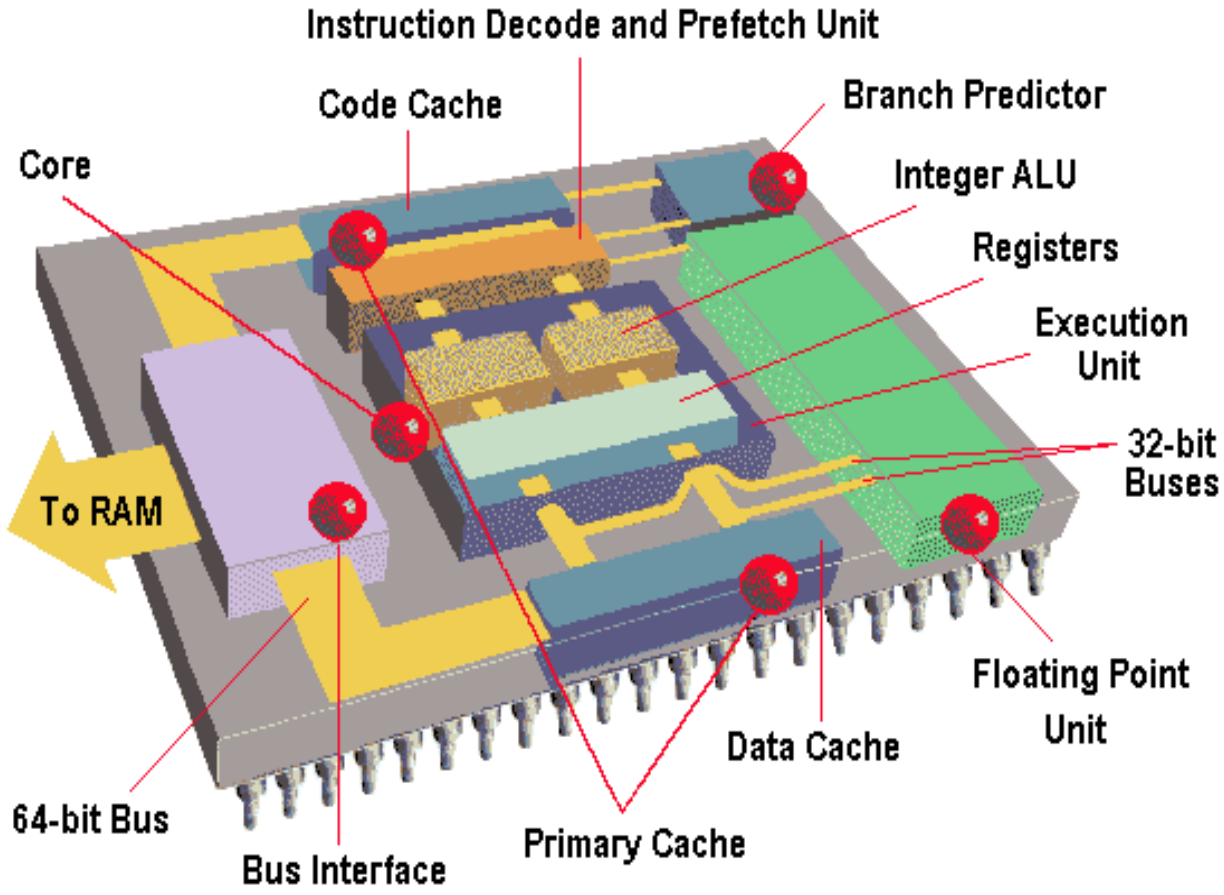


AMD Athlon hadde tre ALUer
og en SIMD flyttallenhet

- Superscalarity is a form of parallelization where we have several (parallel) **ALU** and **FPU** available on CPU
- Can then perform (even) more instructions at the same time!

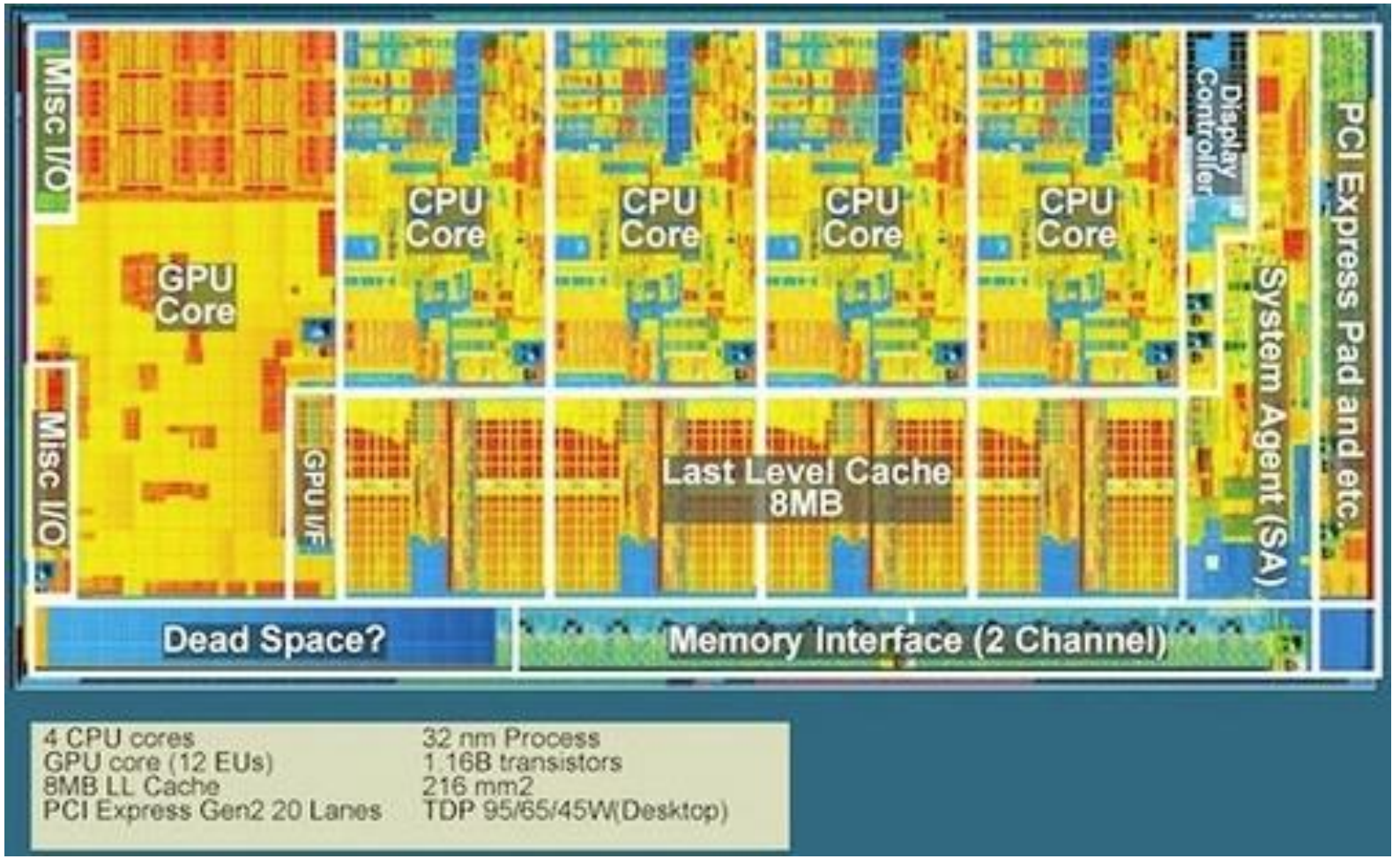
Pentium (IA32) layout (simplified)

- Branch Prediction
  - «"Guess" which instructions to run afterwards and retrieve them on the CPU»
- Cacheing
  - Instructions in own cache (should never return to RAM!)
  - Data in own cache

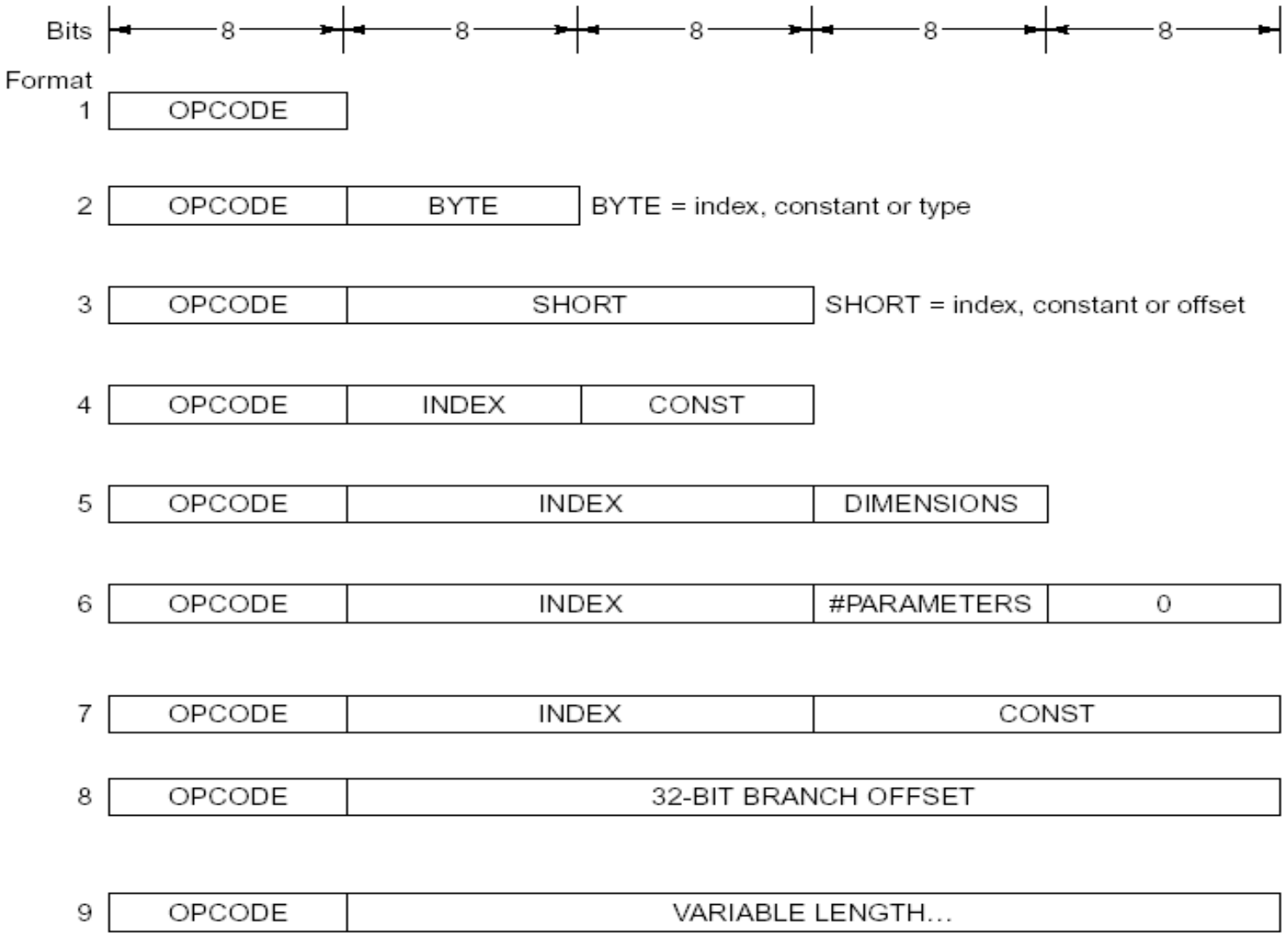


Sandy Bridge

- Multiple "cores" -> parallel running of program (threads)
- Moved Northbridge (and GPU) «on die»



JVM ISA: Format



JVM: 226 instructions

Loads

typeLOAD IND8	Push local variable onto stack
typeALOAD	Push array element on stack
BALOAD	Push byte from an array on stack
SALOAD	Push short from an array on stack
CALOAD	Push char from an array on stack
AALOAD	Push pointer from an array on "

Stores

typeSTORE IND8	Pop value and store in local var
typeASTORE	Pop value and store in array
BASTORE	Pop byte and store in array
SASTORE	Pop short and store in array
CASTORE	Pop char and store in array
AASTORE	Pop pointer and store in array

Pushes

BIPUSH CON8	Push a small constant on stack
SIPUSH CON16	Push 16-bit constant on stack
LDC IND8	Push constant from const pool
typeCONST_#	Push immediate constant
ACONST_NULL	Push a null pointer on stack

Arithmetic

typeADD	Add
typeSUB	Subtract
typeMUL	Multiple
typeDIV	Divide
typeREM	Remainder
typeNEG	Negate

Boolean/shift

ilAND	Boolean AND
ilOR	Boolean OR
ilXOR	Boolean EXCLUSIVE OR
ilSHL	Shift left
ilSHR	Shift right
ilUSHR	Unsigned shift right

Conversion

x2y	Convert x to y
i2c	Convert integer to char
i2b	Convert integer to byte

Stack management

DUPxx	Six instructions for duping
POP	Pop an int from stk and discard
POP2	Pop two ints from stk and discard
SWAP	Swap top two ints on stack

Miscellaneous

IINC IND8,CON8	Increment local variable
WIDE	Wide prefix
NOP	No operation
GETFIELD IND16	Read field from object
PUTFIELD IND16	Write field to object
GETSTATIC IND16	Get static field from class
NEW IND16	Create a new object
INSTANCEOF OFFSET16	Determine type of obj
CHECKCAST IND16	Check object type
ATHROW	Throw exception
LOOKUPSWITCH ...	Sparse multiway branch
TABLESWITCH ...	Dense multiway branch
MONITORENTER	Enter a monitor
MONITOREXIT	Leave a monitor

IND8/16 = index of local variable
CON8/16, D, ATYPE = constant

type, x, y = I, L, F, D
OFFSET16 for branch

Comparison

IF_ICMPrel OFFSET16	Conditional branch
IF_ACMPEQ OFFSET16	Branch if two ptrs equal
IF_ACMPNE OFFSET16	Branch if ptrs unequal
IFrel OFFSET16	Test 1 value and branch
IFNULL OFFSET16	Branch if ptr is null
IFNONNULL OFFSET16	Branch if ptr is nonnull
LCMP	Compare two longs
FCMPL	Compare 2 floats for <
FCMPG	Compare 2 floats for >
DCMPL	Compare doubles for <
DCMPG	Compare doubles for >

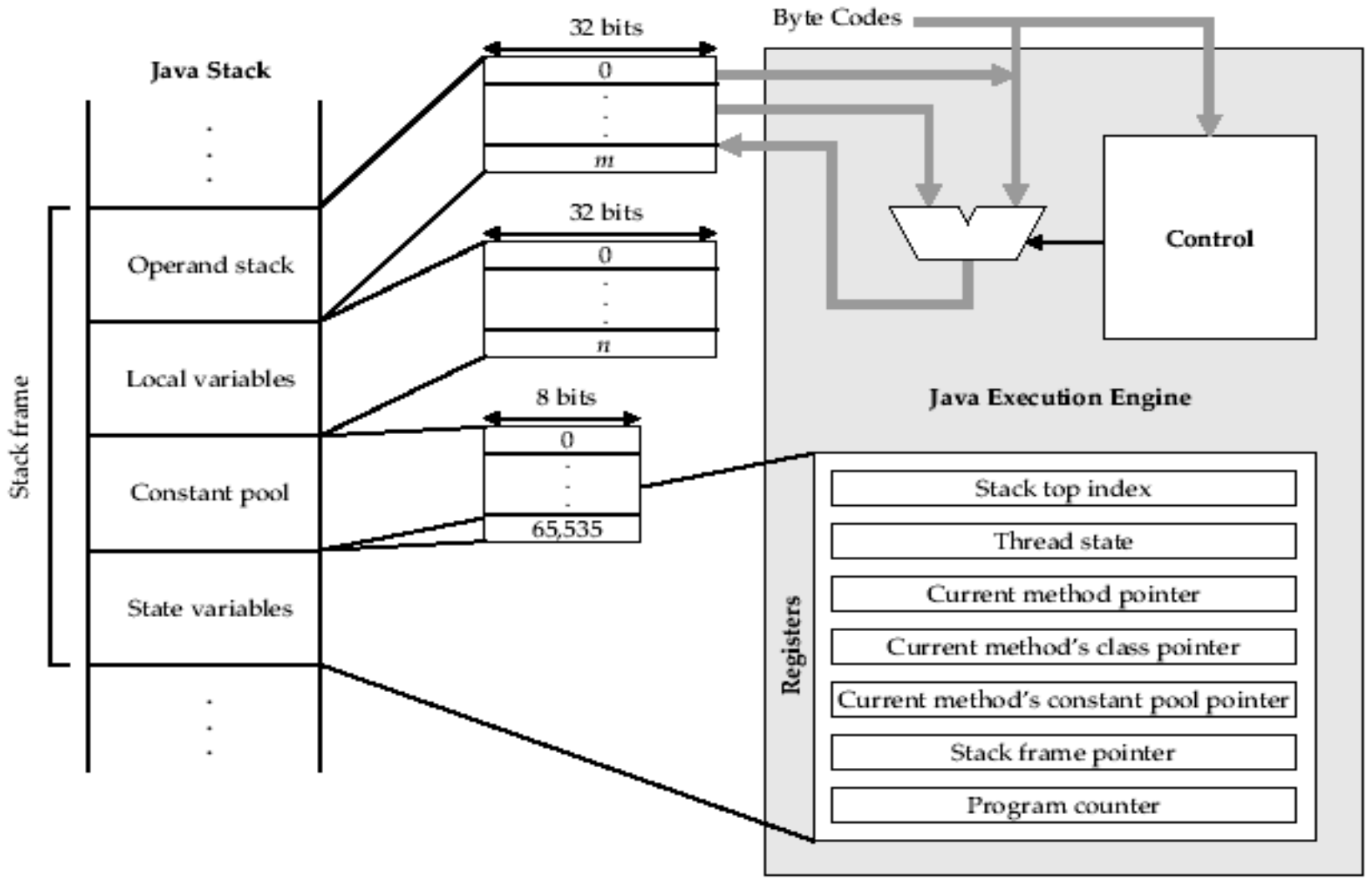
Transfer of control

INVOKEVIRTUAL IND16	Method invocation
INVOKESTATIC IND16	Method invocation
INVOKEINTERFACE ...	Method invocation
INVOKESPECIAL IND16	Method invocation
JSR OFFSET16	Invoke finally clause
typeRETURN	Return value
ARETURN	Return pointer
RETURN	Return void
RET IND8	Return from finally
GOTO OFFSET16	Unconditional branch

Arrays

ANEWARRAY IND16	Create array of ptrs
NEWARRAY ATYPE	Create array of atype
MULTINEWARRAY IN16,D	Create multidim array
ARRAYLENGTH	Get array length

Example: JVM (Java RunTime Enviroment)

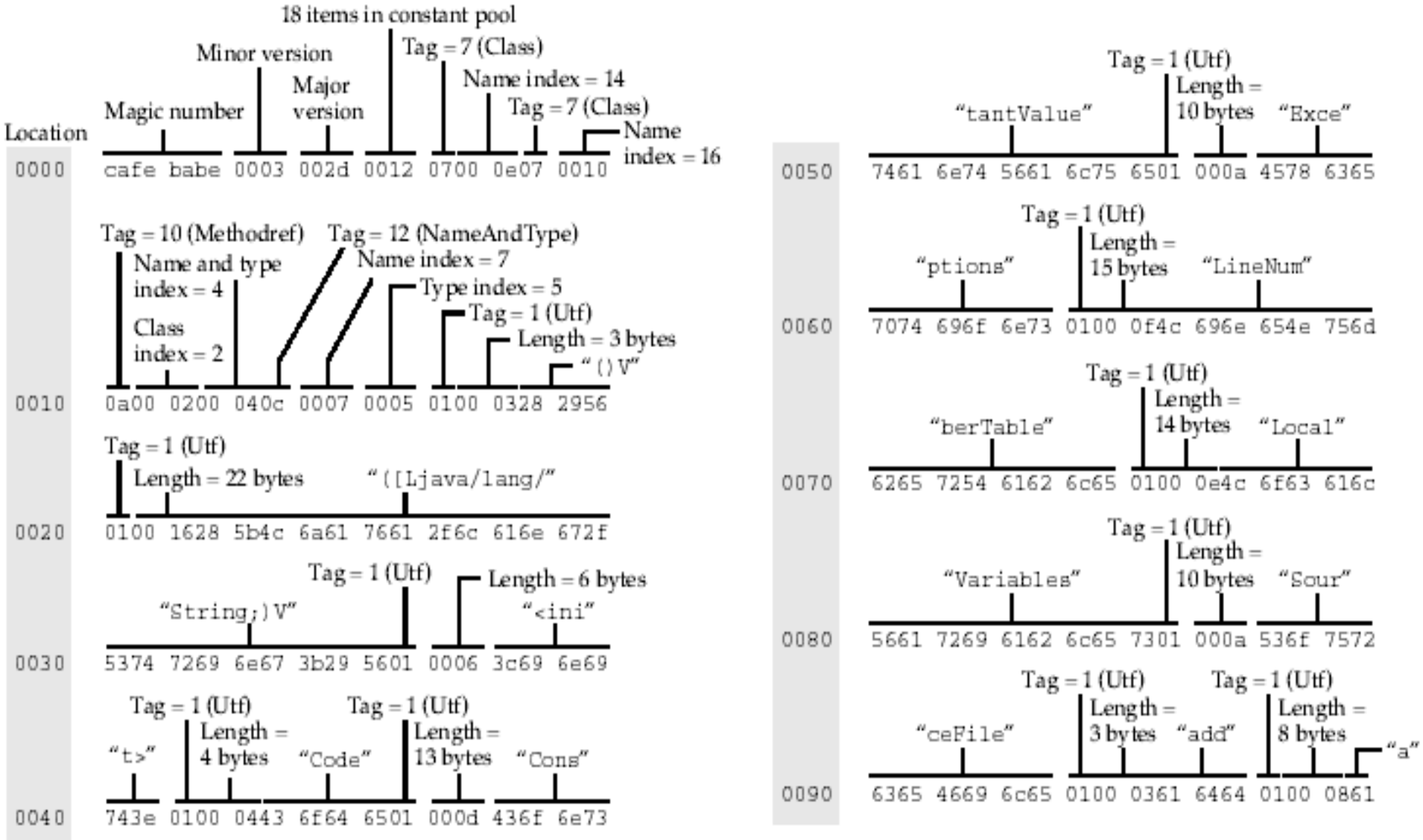


Example: Code and class

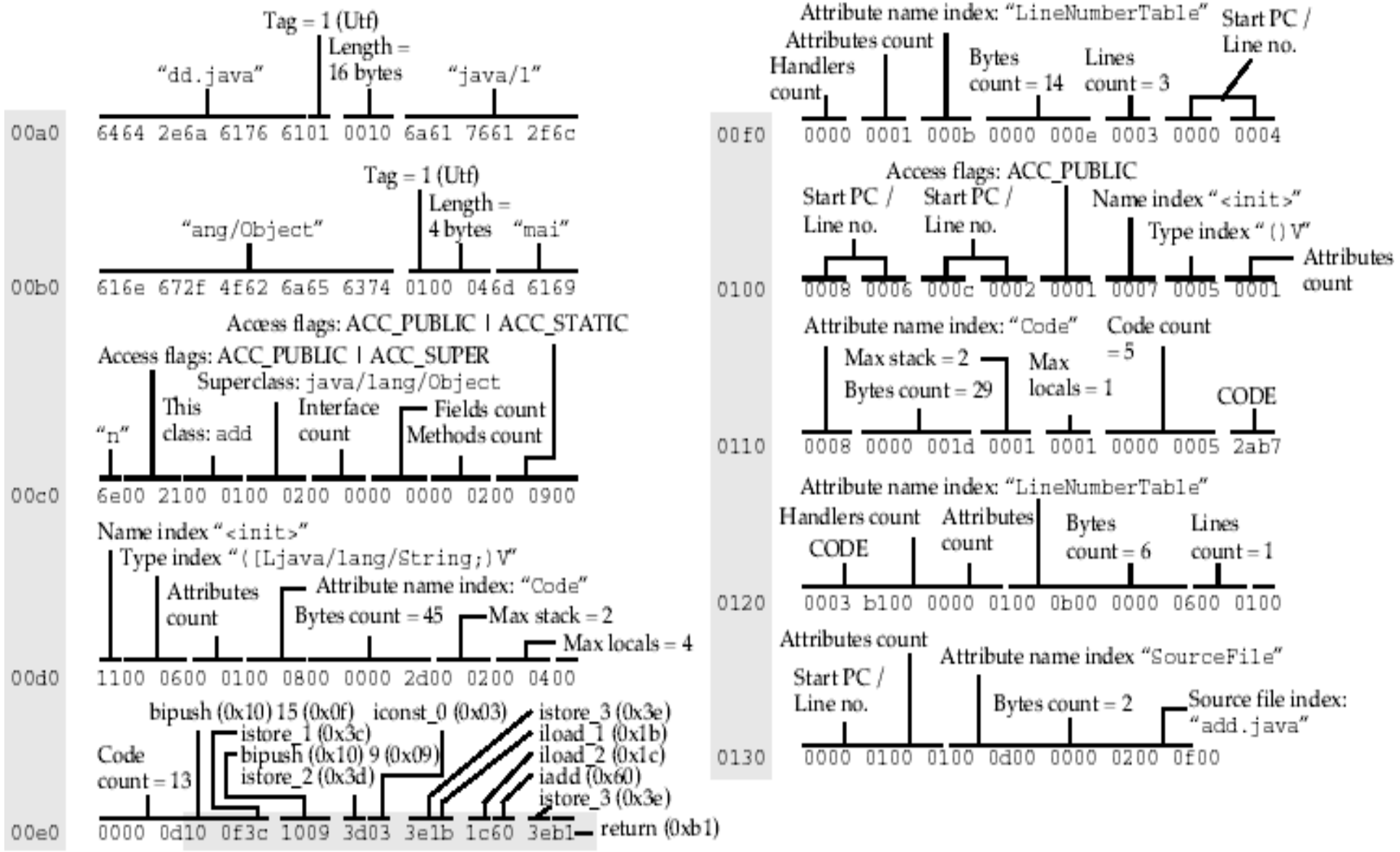
```
public class add {
    public static void main(String args[]) {
        int x=15, y=9, z=0;
        z = x + y;
        }
    }
```

```
0000 cafe babe 0003 002d 0012 0700 0e07 0010   ..............
0010 0a00 0200 040c 0007 0005 0100 0328 2956   ............()V
0020 0100 1628 5b4c 6a61 7661 2f6c 616e 672f   ...([Ljava/lang/
0030 5374 7269 6e67 3b29 5601 0006 3c69 6e69   String;)V...<ini
0040 743e 0100 0443 6f64 6501 000d 436f 6e73   t>...Code...Cons
0050 7461 6e74 5661 6c75 6501 000a 4578 6365   tantValue...Exce
0060 7074 696f 6e73 0100 0f4c 696e 654e 756d   ptions...LineNum
0070 6265 7254 6162 6c65 0100 0e4c 6f63 616c   berTable...Local
0080 5661 7269 6162 6c65 7301 000a 536f 7572   Variables...Sour
0090 6365 4669 6c65 0100 0361 6464 0100 0861   ceFile...add...a
00a0 6464 2e6a 6176 6101 0010 6a61 7661 2f6c   dd.java...java/l
00b0 616e 672f 4f62 6a65 6374 0100 046d 6169   ang/Object...mai
00c0 6e00 2100 0100 0200 0000 0000 0200 0900   n.!.............
00d0 1100 0600 0100 0800 0000 2d00 0200 0400   ..........-.....
00e0 0000 0d10 0f3c 1009 3d03 3e1b 1c60 3eb1   .....<..=.>..`>.
00f0 0000 0001 000b 0000 000e 0003 0000 0004   ................
0100 0008 0006 000c 0002 0001 0007 0005 0001   ................
0110 0008 0000 001d 0001 0001 0000 0005 2ab7   ..............*.
0120 0003 b100 0000 0100 0b00 0000 0600 0100   ................
0130 0000 0100 0100 0d00 0000 0200 0f00        ..............
```

Class file (1)


Location
18 items in constant pool
Minor version
Major version
Magic number
Tag = 7 (Class)
Name index = 14
Tag = 7 (Class)
Name index = 16
0000 cafe babe 0003 002d 0012 0700 0e07 0010
Tag = 10 (Methodref)
Name and type index = 4
Class index = 2
Tag = 12 (NameAndType)
Name index = 7
Type index = 5
Tag = 1 (Utf)
Length = 3 bytes
"()V"
0010 0a00 0200 040c 0007 0005 0100 0328 2956
Tag = 1 (Utf)
Length = 22 bytes
"([Ljava/lang/"
0020 0100 1628 5b4c 6a61 7661 2f6c 616e 672f
Tag = 1 (Utf)
"String;)V"
Length = 6 bytes
"<ini"
0030 5374 7269 6e67 3b29 5601 0006 3c69 6e69
Tag = 1 (Utf)
"t>"
Length = 4 bytes
"Code"
Tag = 1 (Utf)
Length = 13 bytes
"Cons"
0040 743e 0100 0443 6f64 6501 000d 436f 6e73
Tag = 1 (Utf)
"tantValue"
Length = 10 bytes
"Exce"
0050 7461 6e74 5661 6c75 6501 000a 4578 6365
Tag = 1 (Utf)
"ptions"
Length = 15 bytes
"LineNum"
0060 7074 696f 6e73 0100 0f4c 696e 654e 756d
Tag = 1 (Utf)
"berTable"
Length = 14 bytes
"Local"
0070 6265 7254 6162 6c65 0100 0e4c 6f63 616c
Tag = 1 (Utf)
"Variables"
Length = 10 bytes
"Sour"
0080 5661 7269 6162 6c65 7301 000a 536f 7572
Tag = 1 (Utf)
"ceFile"
Tag = 1 (Utf)
Length = 3 bytes
"add"
Tag = 1 (Utf)
Length = 8 bytes
"a"
0090 6365 4669 6c65 0100 0361 6464 0100 0861

Class file (2)


Tag = 1 (Utf)
Length = 16 bytes
"dd.java"
"java/1"
00a0
6464 2e6a 6176 6101 0010 6a61 7661 2f6c
Tag = 1 (Utf)
Length = 4 bytes
"ang/Object"
"mai"
00b0
616e 672f 4f62 6a65 6374 0100 046d 6169
Access flags: ACC_PUBLIC | ACC_STATIC
Access flags: ACC_PUBLIC | ACC_SUPER
Superclass: java/lang/Object
"n"
This class: add
Interface count
Fields count
Methods count
00c0
6e00 2100 0100 0200 0000 0000 0200 0900
Name index "<init>"
Type index "([Ljava/lang/String;)V"
Attributes count
Attribute name index: "Code"
Bytes count = 45
Max stack = 2
Max locals = 4
00d0
1100 0600 0100 0800 0000 2d00 0200 0400
bipush (0x10) 15 (0x0f)
iconst_0 (0x03)
istore_3 (0x3e)
istore_1 (0x3c)
iload_1 (0x1b)
Code count = 13
bipush (0x10) 9 (0x09)
iload_2 (0x1c)
istore_2 (0x3d)
iadd (0x60)
istore_3 (0x3e)
00e0
0000 0d10 0f3c 1009 3d03 3e1b 1c60 3eb1
return (0xb1)
Attribute name index: "LineNumberTable"
Start PC / Line no.
Attributes count
Handlers count
Bytes count = 14
Lines count = 3
00f0
0000 0001 000b 0000 000e 0003 0000 0004
Access flags: ACC_PUBLIC
Start PC / Line no.
Start PC / Line no.
Name index "<init>"
Type index "()V"
Attributes count
0100
0008 0006 000c 0002 0001 0007 0005 0001
Attribute name index: "Code"
Code count = 5
Max stack = 2
Bytes count = 29
Max locals = 1
CODE
0110
0008 0000 001d 0001 0001 0000 0005 2ab7
Attribute name index: "LineNumberTable"
Handlers count
Attributes count
Bytes count = 6
Lines count = 1
CODE
0120
0003 b100 0000 0100 0b00 0000 0600 0100
Attributes count
Attribute name index "SourceFile"
Start PC / Line no.
Bytes count = 2
Source file index: "add.java"
0130
0000 0100 0100 0d00 0000 0200 0f00

Byte code

Høyskolen Kristiania

Location	Code	Mnemonic	Meaning
0x00e3	0x10	bipush	Push next byte onto stack
0x00e4	0x0f	15	Argument to bipush
0x00e5	0x3c	istore_1	Pop stack to local variable 1
0x00e6	0x10	bipush	Push next byte onto stack
0x00e7	0x09	9	Argument to bipush
0x00e8	0x3d	istore_2	Pop stack to local variable 2
0x00e9	0x03	iconst_0	Push 0 onto stack
0x00ea	0x3e	istore_3	Pop stack to local variable 3
0x00eb	0x1b	iload_1	Push local variable 1 onto stack
0x00ec	0x1c	iload_2	Push local variable 2 onto stack
0x00ed	0x60	iadd	Add top two stack elements
0x00ee	0x3e	istore_3	Pop stack to local variable 3
0x00ef	0xb1	return	Return

Java byte code vs x86 instructions

Høyskolen Kristiania

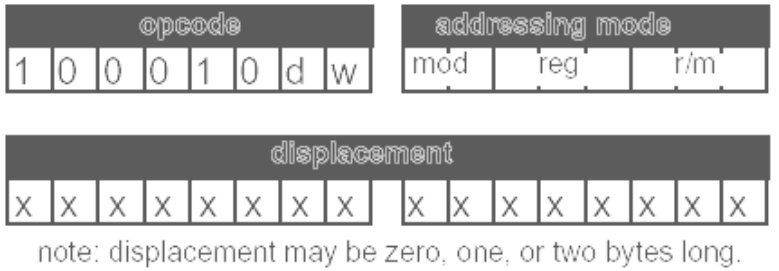
- Java bytecode can be called a RISC type instruction set
- x86 (Intel, as well as AMD) is CISC

- The next two files show:
- The oldest instructions still included (256 of them, there are many, many more more)
- Example of an IA-32 instruction

X86 – Basic Encoding Room

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD rmb,rb	ADD rmv,rv	ADD rb,rmb	ADD rv,rmv	ADD AL,ib	ADD eAX,iv	PUSH ES	POP ES	OR rmb,rb	OR rmv,rv	OR rb,rmb	OR rv,rmv	OR AL,ib	OR eAX,iv	PUSH CS	386 space
1	ADC rmb,rb	ADC rmv,rv	ADC rb,rmb	ADC rv,rmv	ADC AL,ib	ADC eAX,iv	PUSH SS	POP SS	SBB rmb,rb	SBB rmv,rv	SBB rb,rmb	SBB rv,rmv	SBB AL,ib	SBB eAX,iv	PUSH DS	POP DS
2	AND rmb,rb	AND rmv,rv	AND rb,rmb	AND rv,rmv	AND AL,ib	AND eAX,iv	ES:	DAA	SUB rmb,rb	SUB rmv,rv	SUB rb,rmb	SUB rv,rmv	SUB AL,ib	SUB eAX,iv	CS:	DAS
3	XOR rmb,rb	XOR rmv,rv	XOR rb,rmb	XOR rv,rmv	XOR AL,ib	XOR eAX,iv	SS:	AAA	CMP rmb,rb	CMP rmv,rv	CMP rb,rmb	CMP rv,rmv	CMP AL,ib	CMP eAX,iv	DS:	AAS
4	INC eAX	INC eCX	INC eDX	INC eBX	INC eSP	INC eBP	INC eSI	INC eDI	DEC eAX	DEC eCX	DEC eDX	DEC eBX	DEC eSP	DEC eBP	DEC eSI	DEC eDI
5	PUSH eAX	PUSH eCX	PUSH eDX	PUSH eBX	PUSH eSP	PUSH eBP	PUSH eSI	PUSH eDI	POP eAX	POP eCX	POP eDX	POP eBX	POP eSP	POP eBP	POP eSI	POP eDI
6	PUSHA PUSHAD	POPA POPAD	BOUND rv,m2v	ARPL rm2,r2	FS:	GS:	OpLen	AdLen	PUSH iv	IMUL rv,rmv,iv	PUSH ib	IMUL rv,rmv,ib	INSB	INSD	OUTSB	OUTSD
7	JO ib	JNO ib	JB ib	JAE ib	JE ib	JNE ib	JBE ib	JA ib	JS ib	JNS ib	JP ib	JNP ib	JL ib	JGE ib	JLE ib	JG ib
8	Immed rmb,ib	Immed rmv,iv		Immed rmv,ib	TEST rmb,rb	TEST rmv,rv	XCHG rmb,rb	XCHG rmv,rv	MOV rmb,rb	MOV rmv,rv	MOV rb,rmb	MOV rv,rmv	MOV rm,segr	LEA rv, m	MOV segr,rm	POP rmv
9	NOP	XCHG eAX,eCX	XCHG eAX,eDX	XCHG eAX,eBX	XCHG eAX,eSP	XCHG eAX,eBP	XCHG eAX,eSI	XCHG eAX,eDI	CWDE CBW	CWQ CDQ	CALL FAR s:m	FWAIT	PUSHF	POPF	SAHF	LAHF
A	MOV AL,[iv]	MOV eAX,[iv]	MOV [iv],AL	MOV [iv],eAX	MOVSB	MOVSD	CMPSB	CMPSD	TEST AL,rb	TEST eAX,iv	STOSB	STOSD	LODSB	LODSD	SCASB	SCASD
B	MOV AL,ib	MOV CL,ib	MOV DL,ib	MOV BL,ih	MOV AH,ib	MOV CH,ih	MOV DH,ih	MOV BH,ih	MOV eAX,iv	MOV eCX,iv	MOV eDX,iv	MOV eBX,iv	MOV eSP,iv	MOV eBP,iv	MOV eSI,iv	MOV eDI,iv
C	Shift rmb,ib	Shift rmv,ib	RET i2	RET	LES rm,rmp	LDS rm,rmp	MOV rmb, ib	MOV rmv, iv	ENTER i2, ih	LEAVE	RETF i2	RETF	INT 3	INT ih	INTO	IRET
D	Shift rmb, 1	Shift rmv, 1	Shift rmb,CL	Shift rmv,CL	AAM	AAD		XLATB	87 space	87 space	87 space	87 space	87 space	87 space	87 space	87 space
E	LOOPNE short	LOOPE short	LOOP short	JeCXZ short	IN AL,[ib]	IN eAX,[ib]	OUT [ib],AL	OUT [ib],eAX	CALL iv	JMP iv	JMP FAR s:m	JMP ib	IN AL,[DX]	IN eAX,[DX]	OUT [DX],AL	OUT [DX],eAX
F	LOCK		REPNE	REP REPE	HLT	CMC	Unary rmb	Unary rmv	CLC	STC	CLI	STI	CLD	STD	IncDec rmb	Indir rmv

MOV (386->) – mod-reg-r/m



d and **w** in encoding define the length and direction of the operation

Table 23: REG Bit Encodings

reg	w=0	16 bit mode w=1	32 bit mode w=1
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Table 24: MOD Encoding

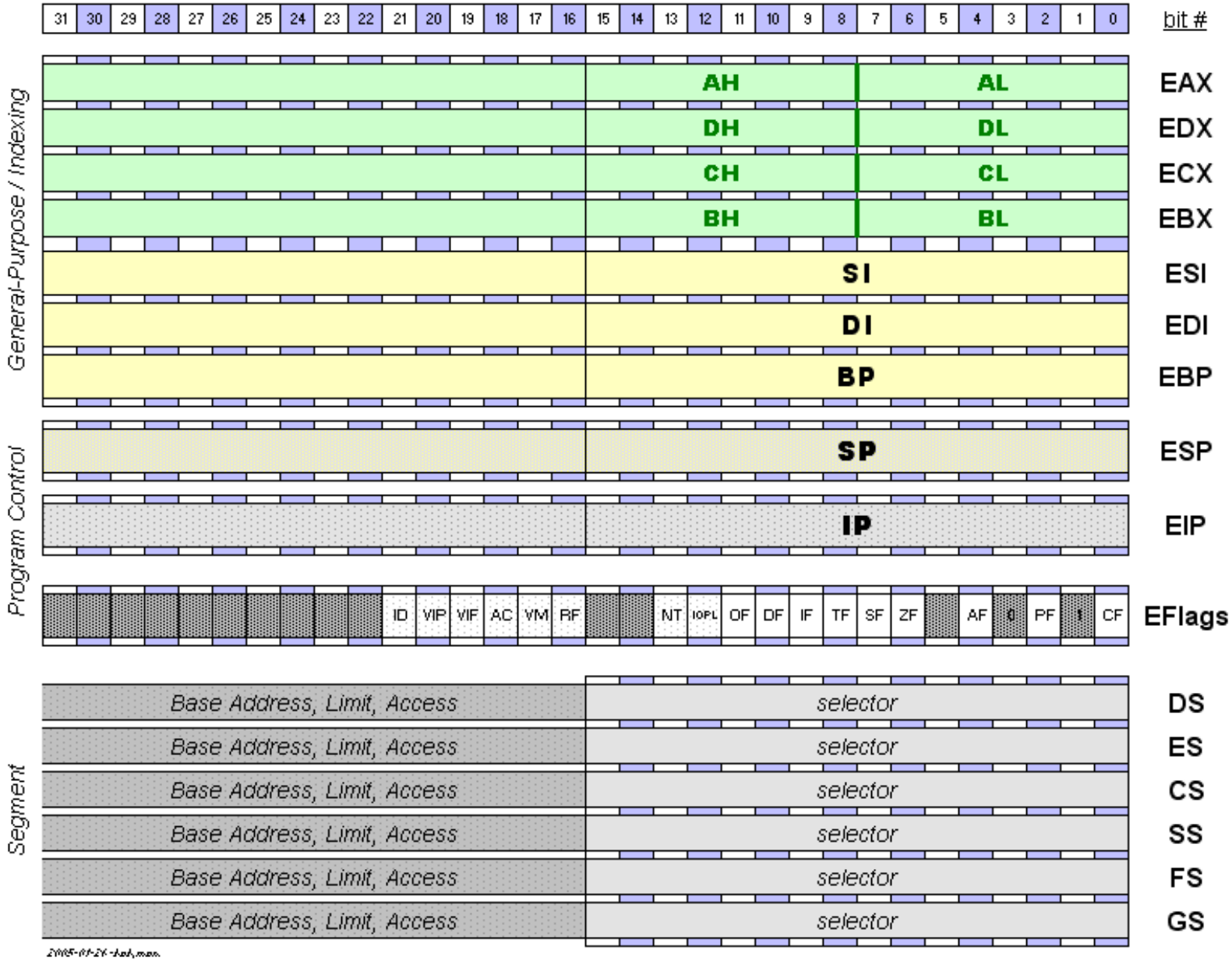
MOD	Meaning
00	The r/m field denotes a register indirect memory addressing mode or a base/indexed addressing mode (see the encodings for r/m) *unless* the r/m field contains 110. If MOD=00 and r/m=110 the mod and r/m fields denote displacement-only (direct) addressing.
01	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is an eight bit signed displacement following the mod/reg/rm byte.
10	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is a 16 bit signed displacement (in 16 bit mode) or a 32 bit signed displacement (in 32 bit mode) following the mod/reg/rm byte .
11	The r/m field denotes a register and uses the same encoding as the *reg* field

Table 25: R/M Field Encoding

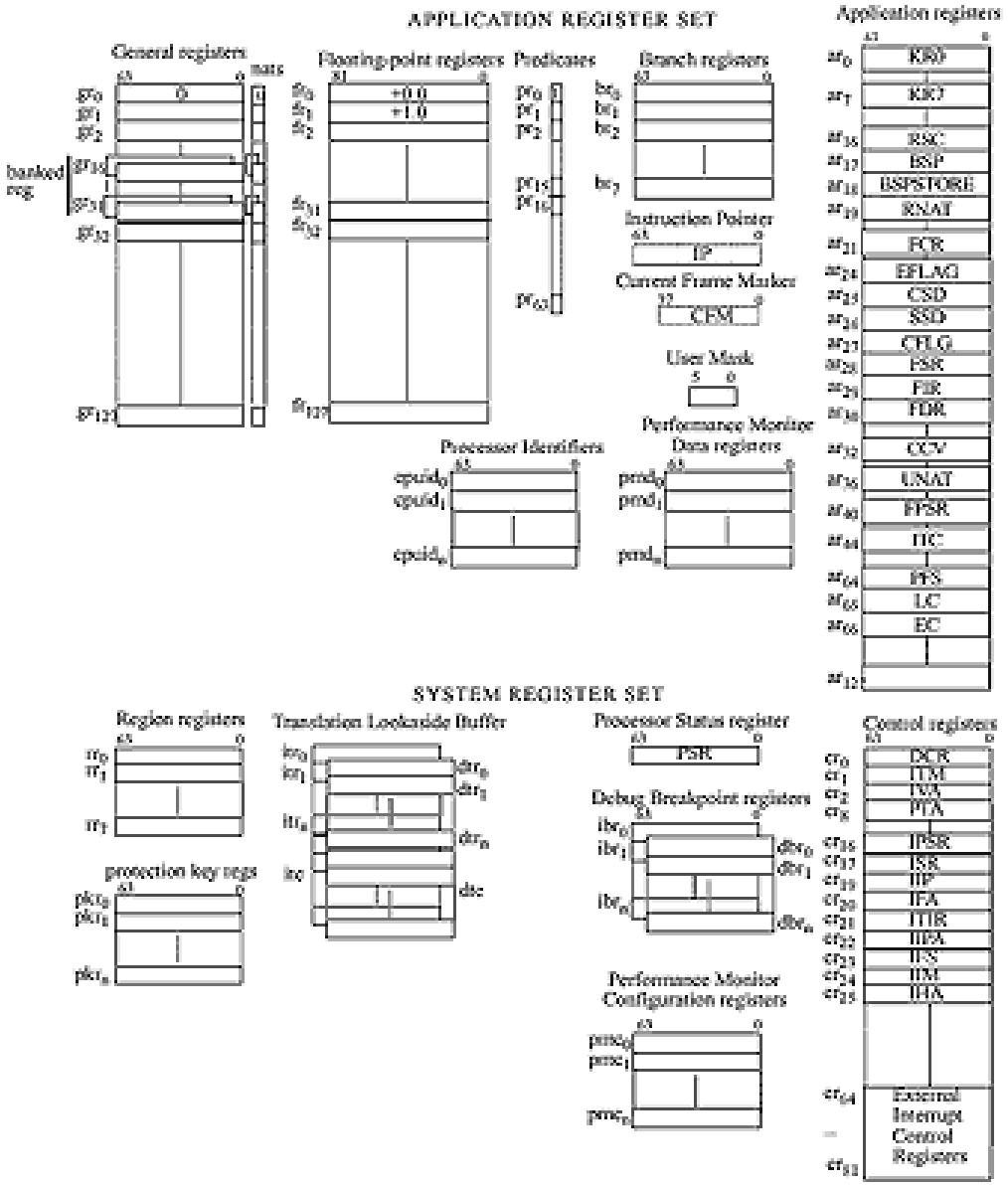
R/M	Addressing mode (Assuming MOD=00, 01, or 10)
000	[BX+SI] or DISP[BX][SI] (depends on MOD)
001	[BX+DI] or DISP[BX+DI] (depends on MOD)
010	[BP+SI] or DISP[BP+SI] (depends on MOD)
011	[BP+DI] or DISP[BP+DI] (depends on MOD)
100	[SI] or DISP[SI] (depends on MOD)
101	[DI] or DISP[DI] (depends on MOD)
110	Displacement-only or DISP[BP] (depends on MOD)
111	[BX] or DISP[BX] (depends on MOD)

```
[ebx][ebp]          ;Uses DS by default.
[ebp][ebx]          ;Uses SS by default.
[ebp*1][ebx]        ;Uses DS by default.
[ebx][ebp*1]        ;Uses DS by default.
[ebp][ebx*1]        ;Uses SS by default.
[ebx*1][ebp]        ;Uses SS by default.
es:[ebx][ebp*1]     ;Uses ES.
```

386 Registers (IA32) User Programs



X86-64 register



- Very many different registers
  -on each core!
- Extreme complexity
- How do we relate to complexity?
- **Abstract!**

Abstraction

Høyskolen Kristiania

- a concept or idea that is not associated with any specific instance
- to remove the unnecessary in a given context
- to focus on the features that are common
- To "hide" complexity

Example:

- In a Java program you can hide a complicated program inside a method…
- after that you just need to relate to the method
- High-level language abstracts away all the machine-complexity
- System.out.print ("Hello World!") Actually involves a few thousand instructions since is a method call from java.exe to the operating system's write-to-screen routines.

What is abstraction?

- Abstraction is one of the four cornerstones of Computer Science. It involves filtering out – essentially, ignoring - the characteristics that we don't need in order to concentrate on those that we do.