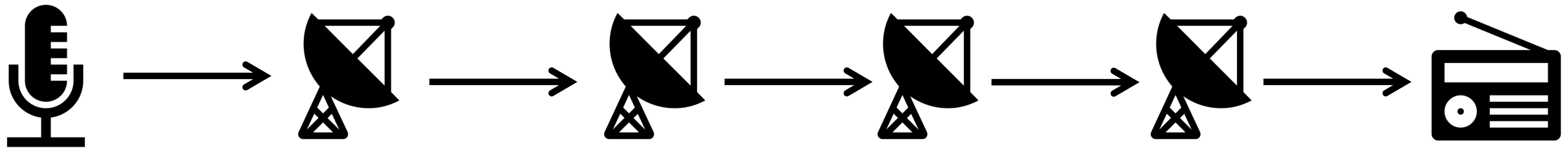
High performance parallel systems

Lecture 1 – Basics of Computer Architecture

Kenneth Skovhede, NBI, 2020-11-17

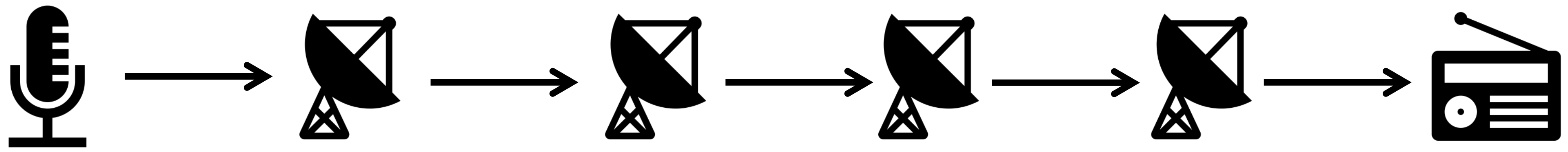
UNIVERSITY OF COPENHAGEN

Analog signals



Analog signals
- have (theoretically) unlimited resolution
- invariably degrade (noise)
- accumulate noise (signal-to-noise ratio increases)
- eventually loose the signal in noise

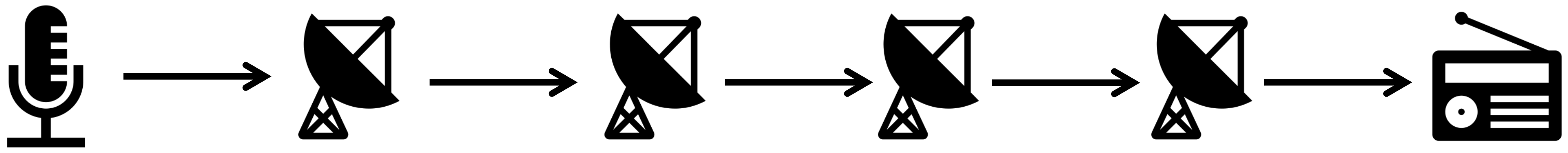
Digital signals

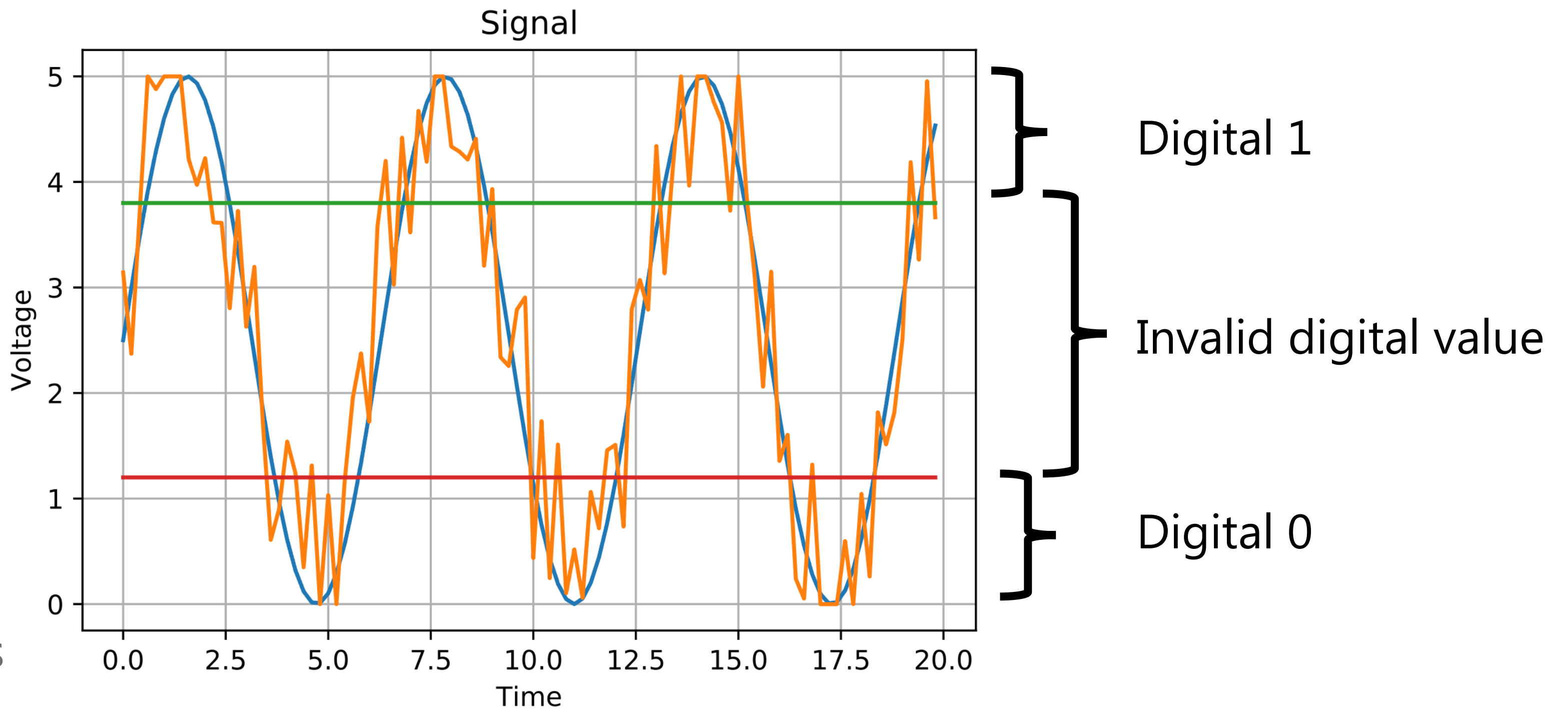


Digital signals

- have on/off resolution
- tolerate degradation (noise)
- removes noise when forwarded
- can be repeated infinitely (theoretically)
    - (bit flips occur, but for other reasons)

Digital signals





We can use other interpretations than binary but it generally just increases complexity

A game of telephone

Unlimited vocabulary

$$\begin{tikzcd}
\boxed{\text{Horse}} \arrow[r] & \boxed{\text{Worse}} \arrow[r] & \boxed{\text{Loose}} \arrow[r] & \boxed{\text{Woos}} \arrow[r] & \boxed{\text{Wool}}
\end{tikzcd}$$

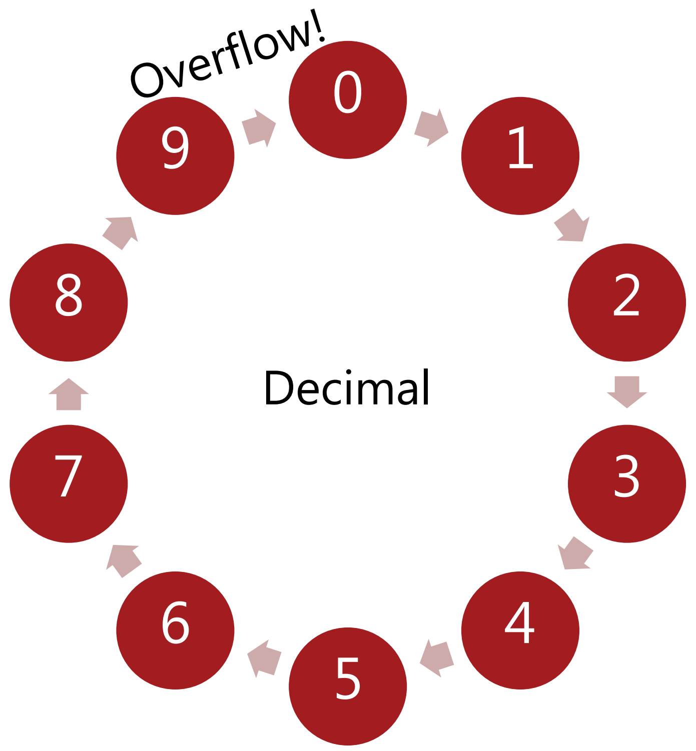
Binary vocabulary

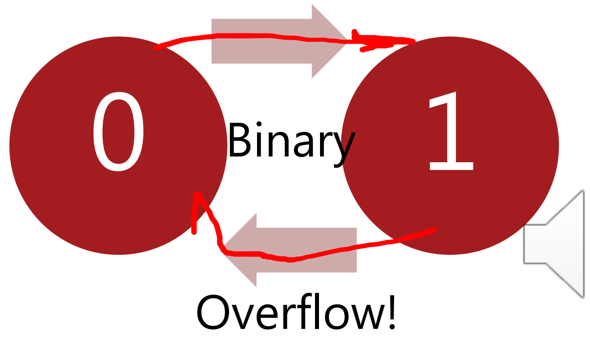
$$\begin{tikzcd}
\boxed{\text{Yes}} \arrow[r, "\text{Less?}"'] & \boxed{\text{Yes}} \arrow[r, "\text{Mess?}"'] & \boxed{\text{Yes}} \arrow[r, "\text{Res?}"'] & \boxed{\text{Yes}} \arrow[r] & \boxed{\text{Yes}}
\end{tikzcd}$$

Numbers with a digital system

Digital and binary

Binary (0b)	Decimal	Binary (0b)	Decimal
0000	0	1011	11
0001	1	1100	12
0010	2	1101	13
0011	3	1110	14
0100	4	1111	15
0101	5	10000	16
0110	6	10001	17
0111	7	10010	18
1000	8	10011	19
1001	9	10100	20
1010	10	…	…



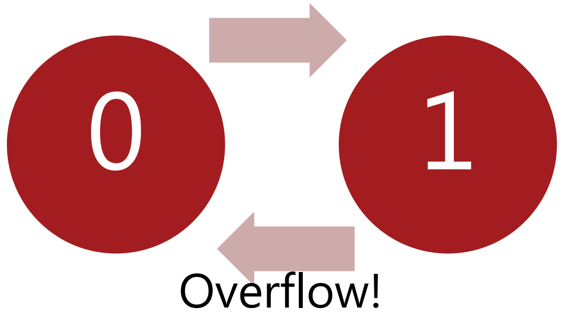


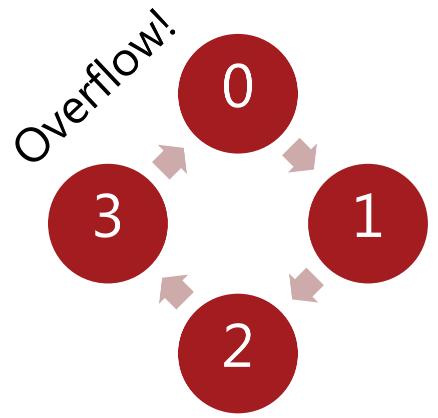
Try it yourself – base2 & base4 addition

To get a feel for math with numbers that do not use base10, try these basic math problems:

Binary numbers:	Result (in binary)
0b0010 + 0b0010 =	
0b0101 + 0b1010 =	
0b0111 + 0b0111 =	

Base4 numbers:	Result (in base4)
0010 + 0010 =	
0301 + 1030 =	
0333 + 0333 =	





Do not convert into decimal before doing the addition!

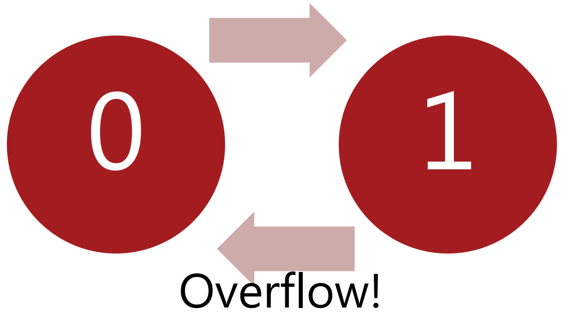
Try it yourself – base2 & base4 addition

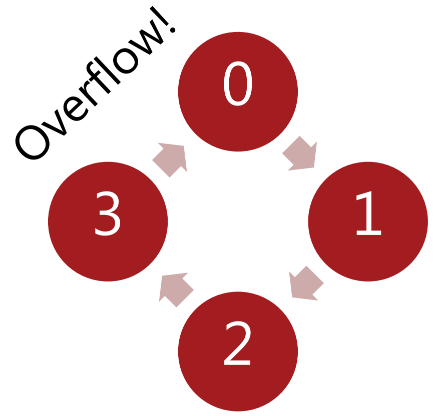
To get a feel for math with numbers that do not use base10, try these basic math problems:

Binary numbers:	Result (in binary)
0b0010 + 0b0010 =	0b0100
0b0101 + 0b1010 =	0b1111
0b0111 + 0b0111 =	0b1110

Base4 numbers:	Result (in base4)
0010 + 0010 =	0020
0301 + 1030 =	1010
0333 + 0333 =	1332

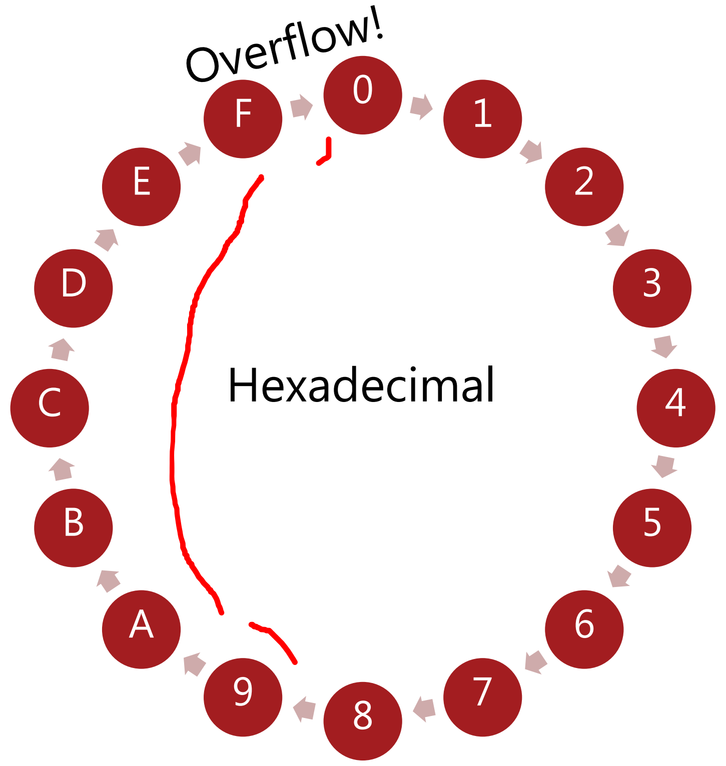
Do not convert into decimal before doing the addition!

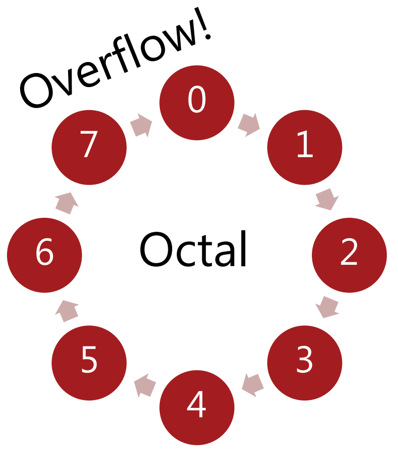




Other numeric bases

Binary (0b)	Decimal	Octal (0c)	Hexadecimal (0x)
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F
10000	16	20	10

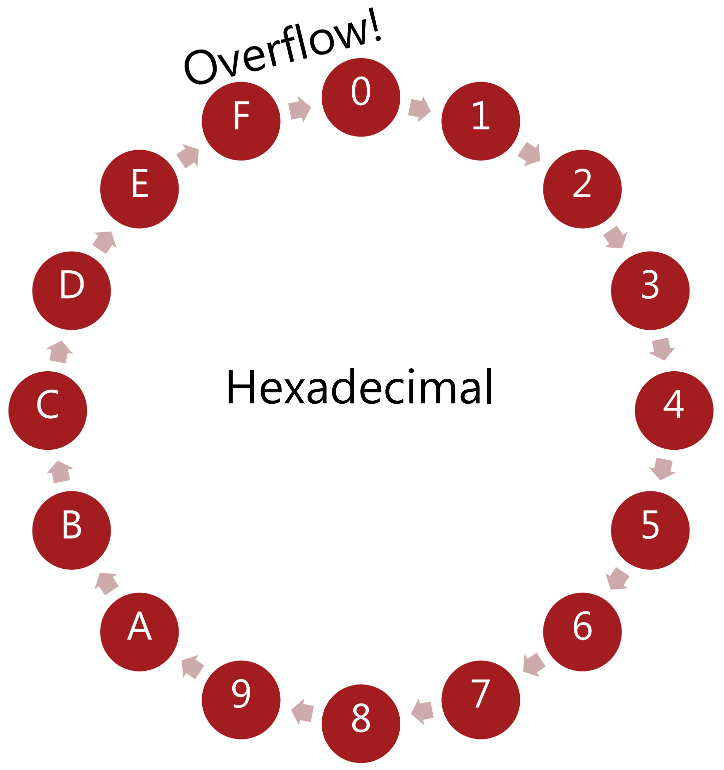




Try it yourself – base16 addition

To get a feel for math with numbers that do not use base10, try these basic math problems:

Hex numbers:	Result (in hex)
0xA + 0x1 =	
0x1F + 0x1F =	
0xBCBC + 0x1234 =	

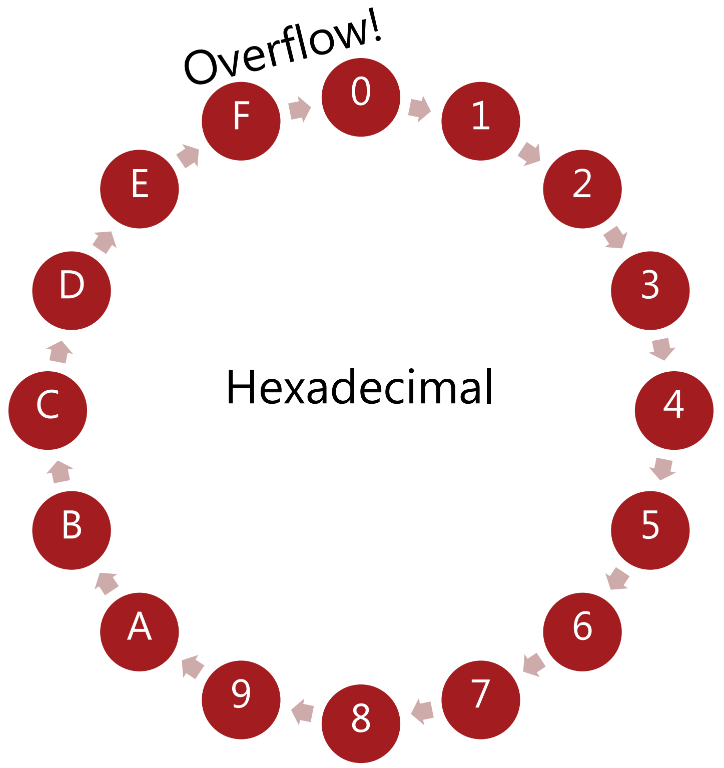


Do not convert into decimal before doing the addition!

Try it yourself – base16 addition

To get a feel for math with numbers that do not use base10, try these basic math problems:

Hex numbers:	Result (in hex)
0xA + 0x1 =	0xB
0x1F + 0x1F =	0x3E
0xBCBC + 0x1234 =	0xCEF0



Do not convert into decimal before doing the addition!

Bases with "power of 2" properties

Binary	Decimal	Octal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F

Hexadecimal	Binary
0xAB	1010 1011
0xF1	1111 0001
0xABCDEF09	1010 1011 1100 1101 1110 1111 0000 1001

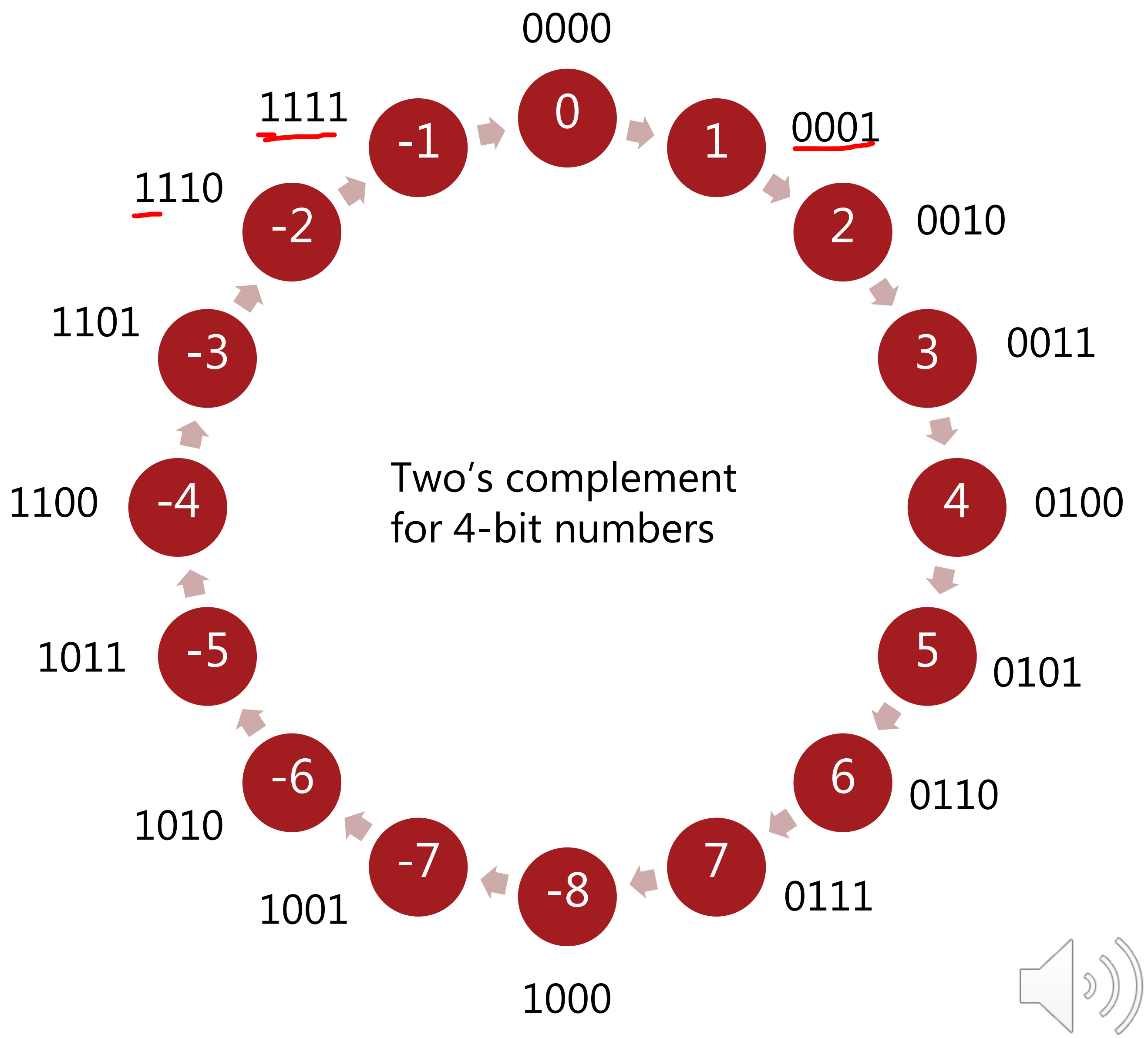
Octal	Binary
0c56	101 110
0c71	111 001
0x6701	110 111 000 001

Negative numbers

Bits	Min	Max
4	-8	7
8	-128	127
16	-16384	16383
32	-2147483648	2147483647

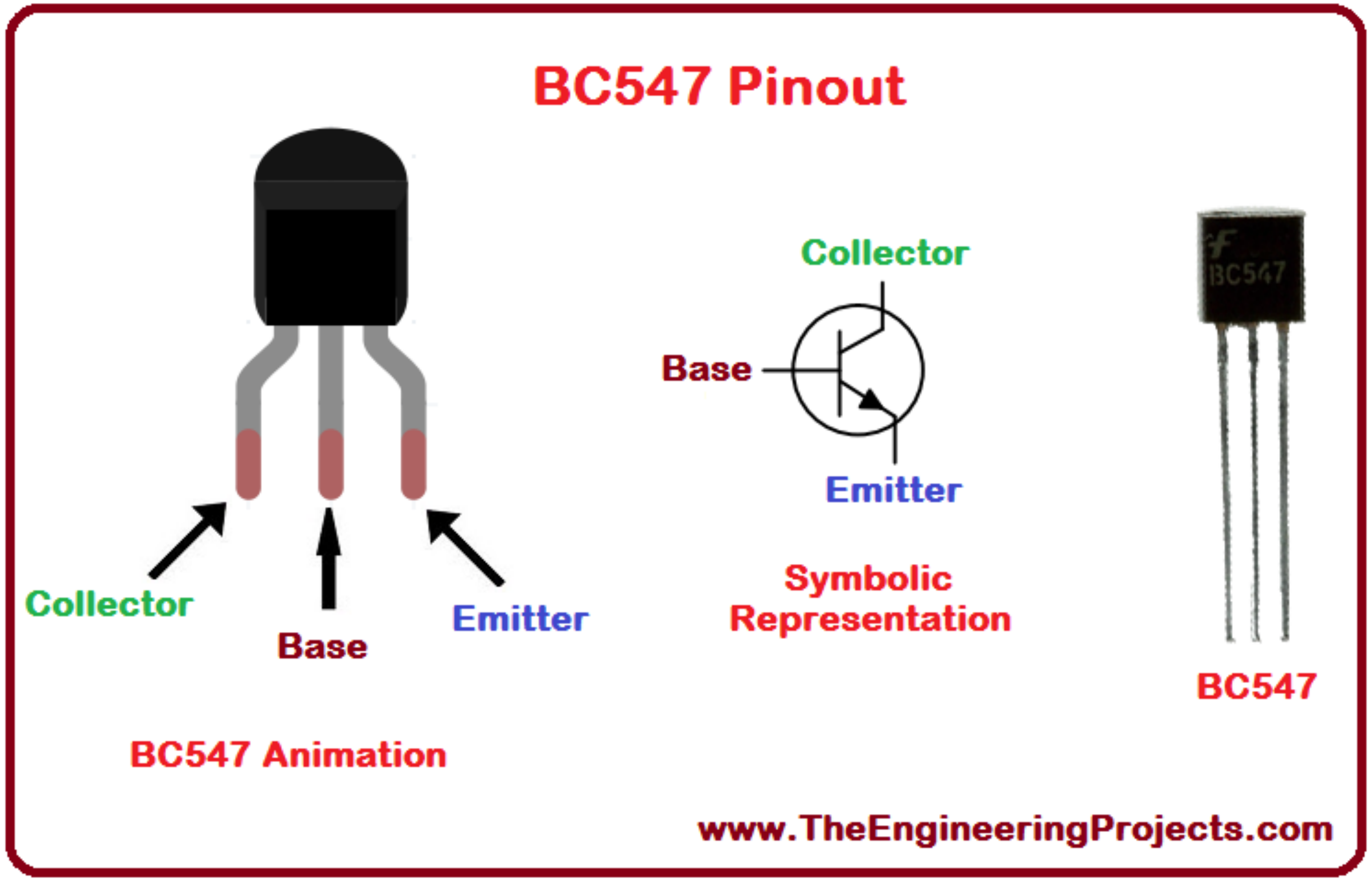
Odd behavior due to non-symmetry

- -(-8) = -8
- Abs(-8) = -8
- -8 * -1 = -8



Digital circuits

Building blocks – A transistor

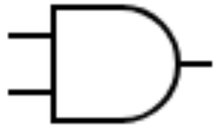


Note that the image on the right is the transistor with a plastic house, which is obviously not present inside a CPU core

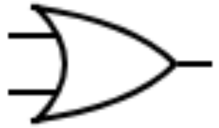
Basic gates

AND	0	1
0	0	0
1	0	1

AND

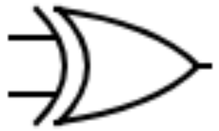


OR

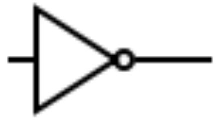


OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0



XOR

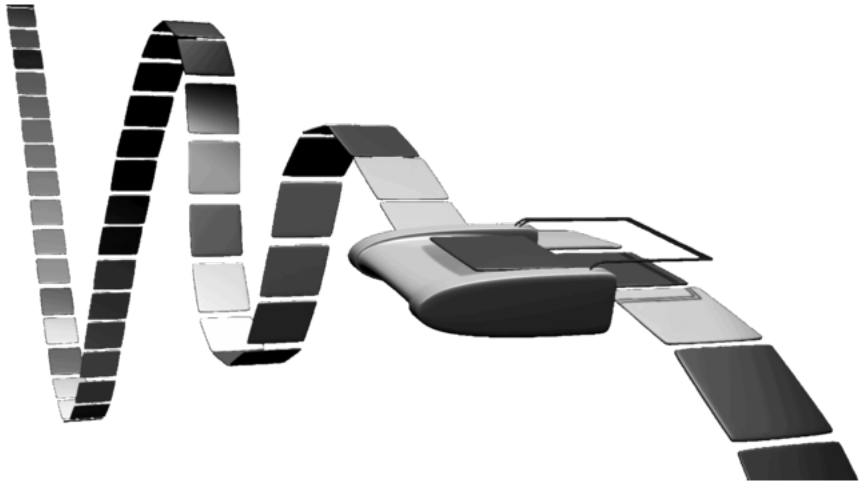


Not

NOT	0	1
	1	0

A conceptual computer

The Turing Machine



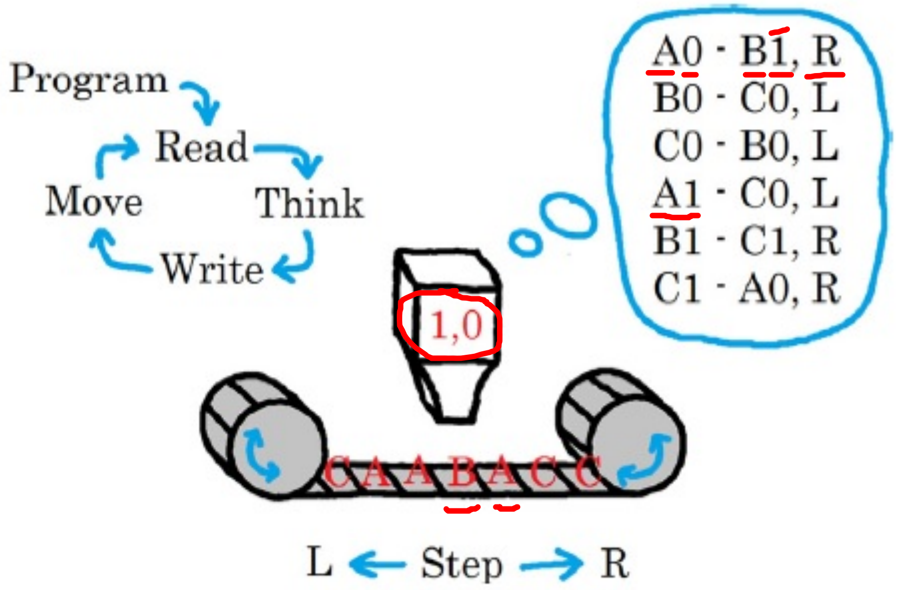


Image from: https://phys.org/news/2013-03-artificial-muscle-universal-turing-machine.html

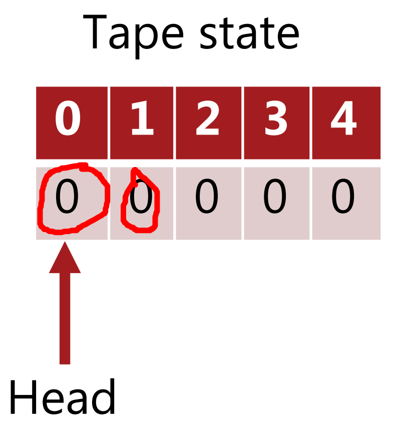
Turing Machine Example

- The state can be any of: A, B, C, ~~D~~
  - The state D means program termination
  - Initialized to state A
- The tape can contain either zero or one
  - The tape is initialized to all zeros
- Write a program that emits the sequence 101010···

Input	Action
A0	B1 R
B0	A0 R
C0	unreachable
A1	unreachable
B1	unreachable
C1	unreachable

New state = B, Write 1, Move right

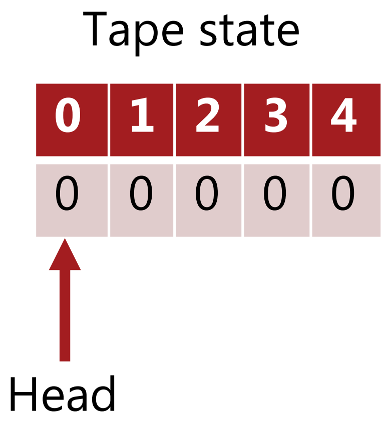
New state = A, Write 0, Move right



Try it yourself – Turing Machine

- The state can be any of: A, B, C, D
  - The state D means program termination
  - Initialized to state A
- The tape can contain either zero or one
  - The tape is initialized to all zeros
- Write a program that emits the sequence 1100000···

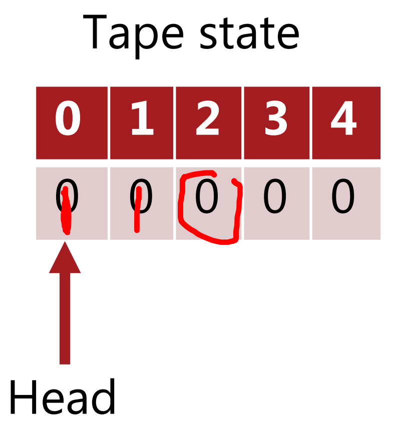
Input	Action
A0	
B0	
C0	
A1	
B1	
C1	



Try it yourself – Turing Machine

- The state can be any of: A, B, C, D
  - The state D means program termination
  - Initialized to state A
- The tape can contain either zero or one
  - The tape is initialized to all zeros
- Write a program that emits the sequence 1100000···

Input	Action
A0	B1, R
B0	C1, R
C0	C0, R
A1	
B1	
C1	



The halting problem

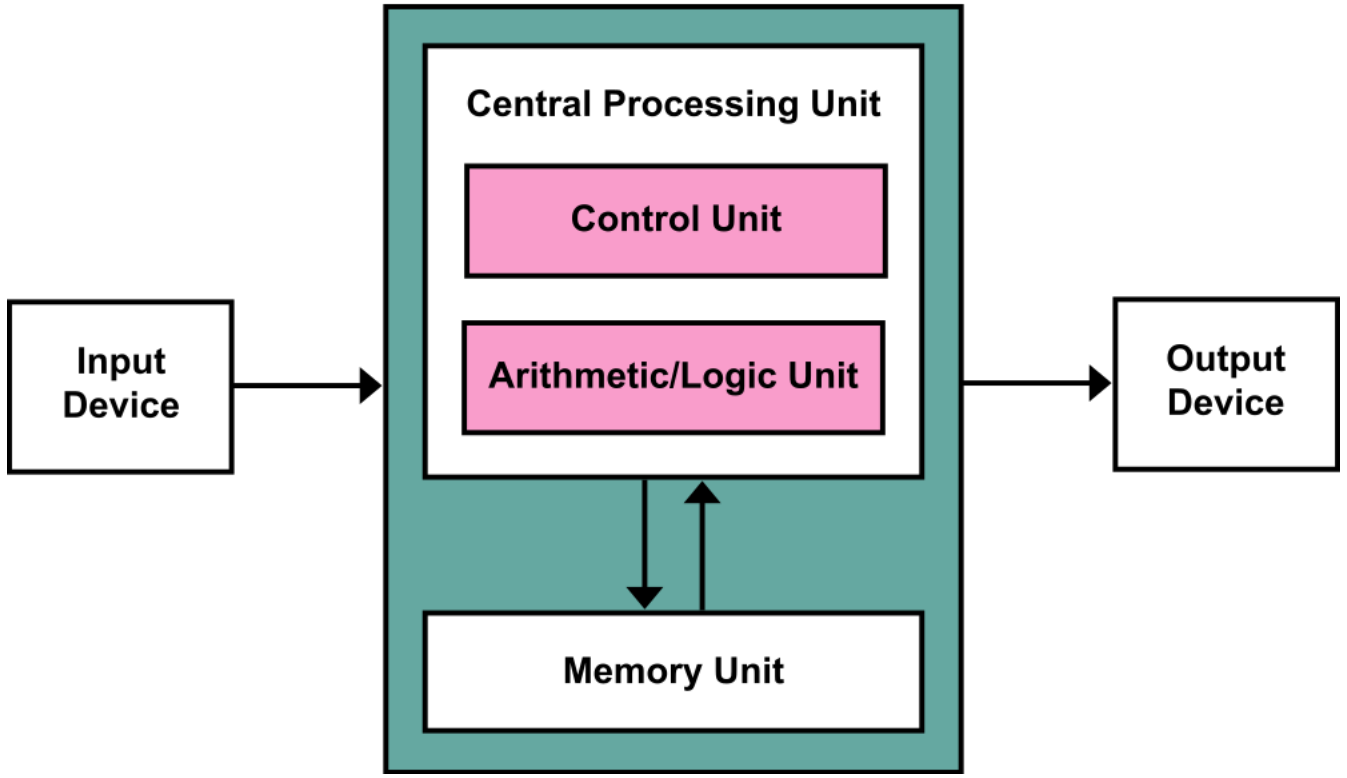
```
void easy_halting1(int x) {
  return;
}

void easy_halting2(int x) {
  while(1);
}
```

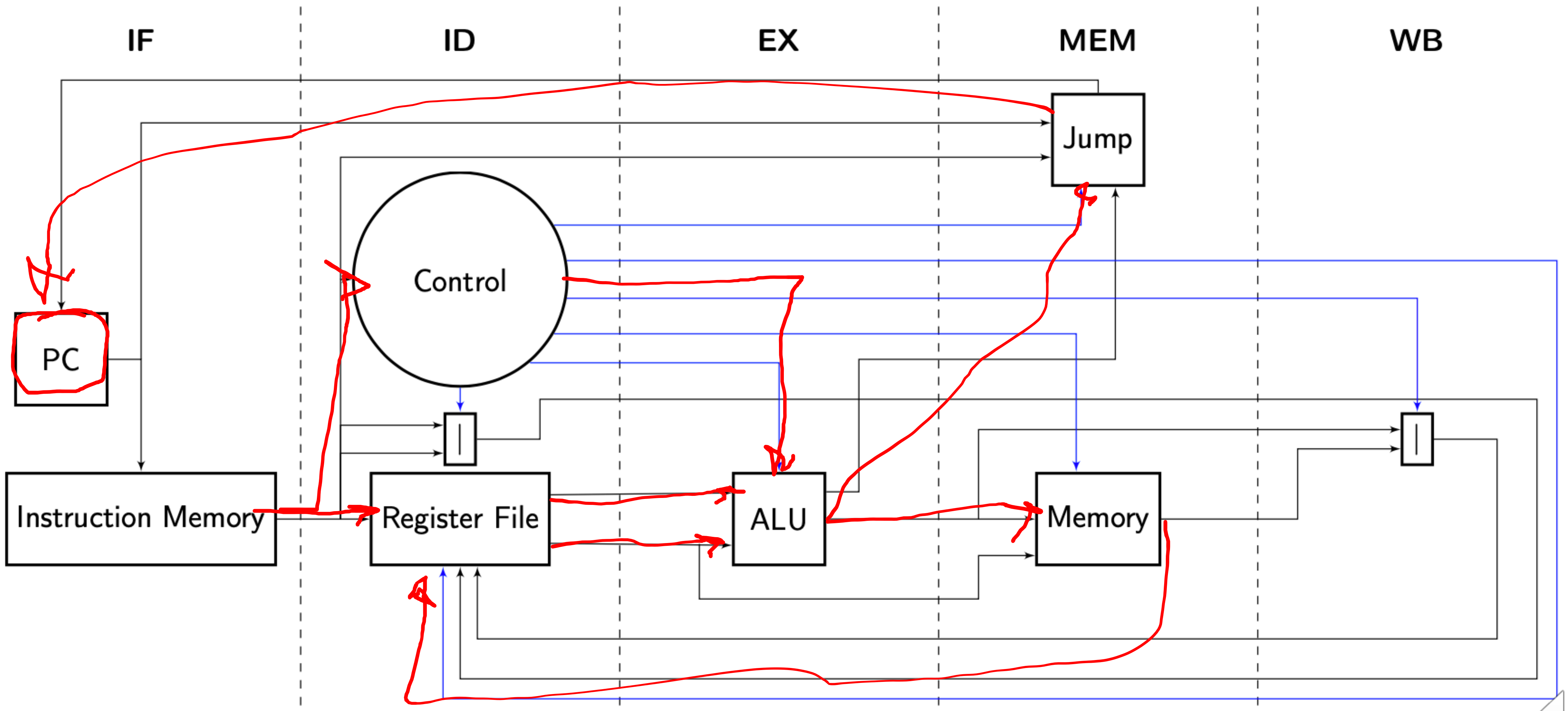
```
void hard_halting(int x) {
  while (x != 0) {
    x = (x * (x + 1)) % 73;
  }
}
```

```
Trying 0 - 0 rounds
Trying 1 - 12 rounds
Trying 2 - 11 rounds
…
```

The von Neumann Architecture



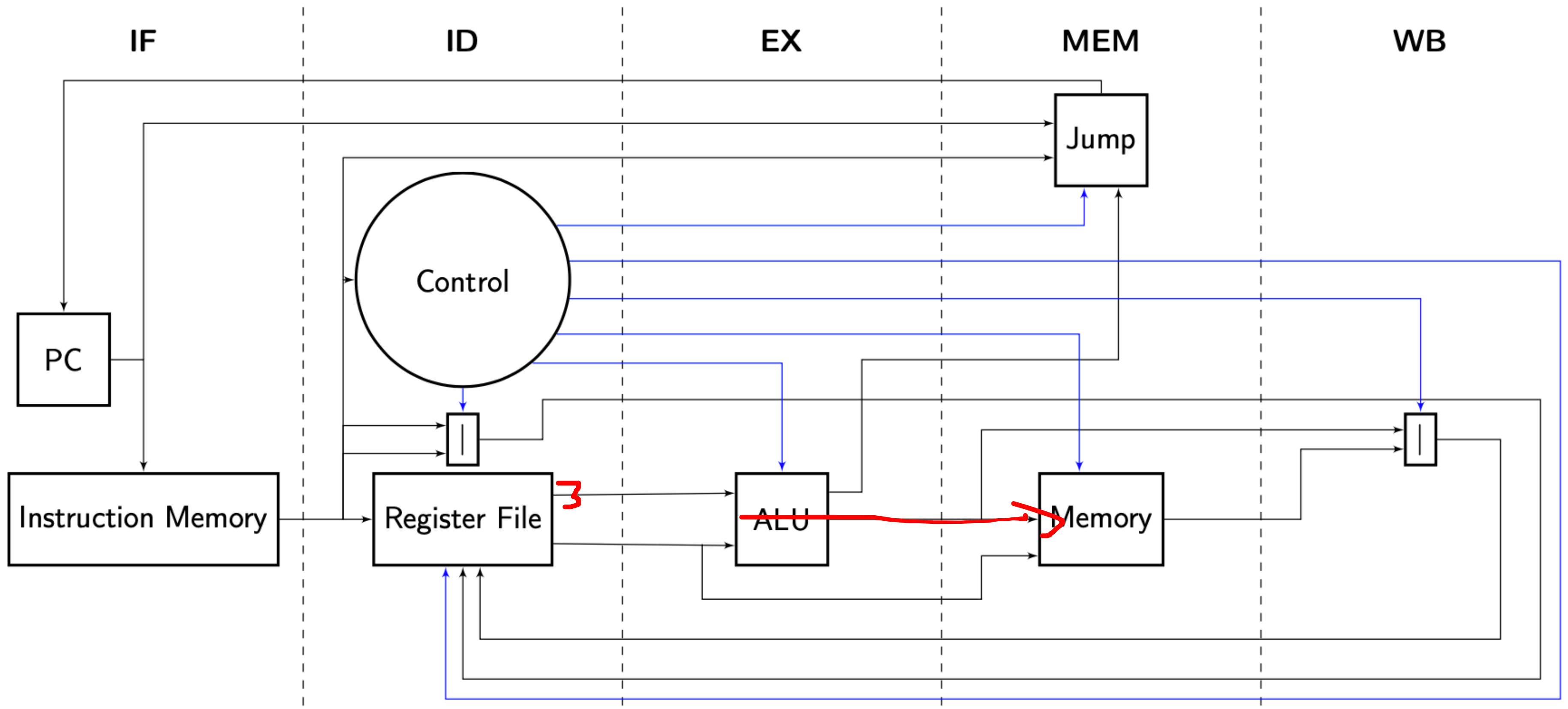
A simple CPU



Drawing by Carl-Johannes Johnsen

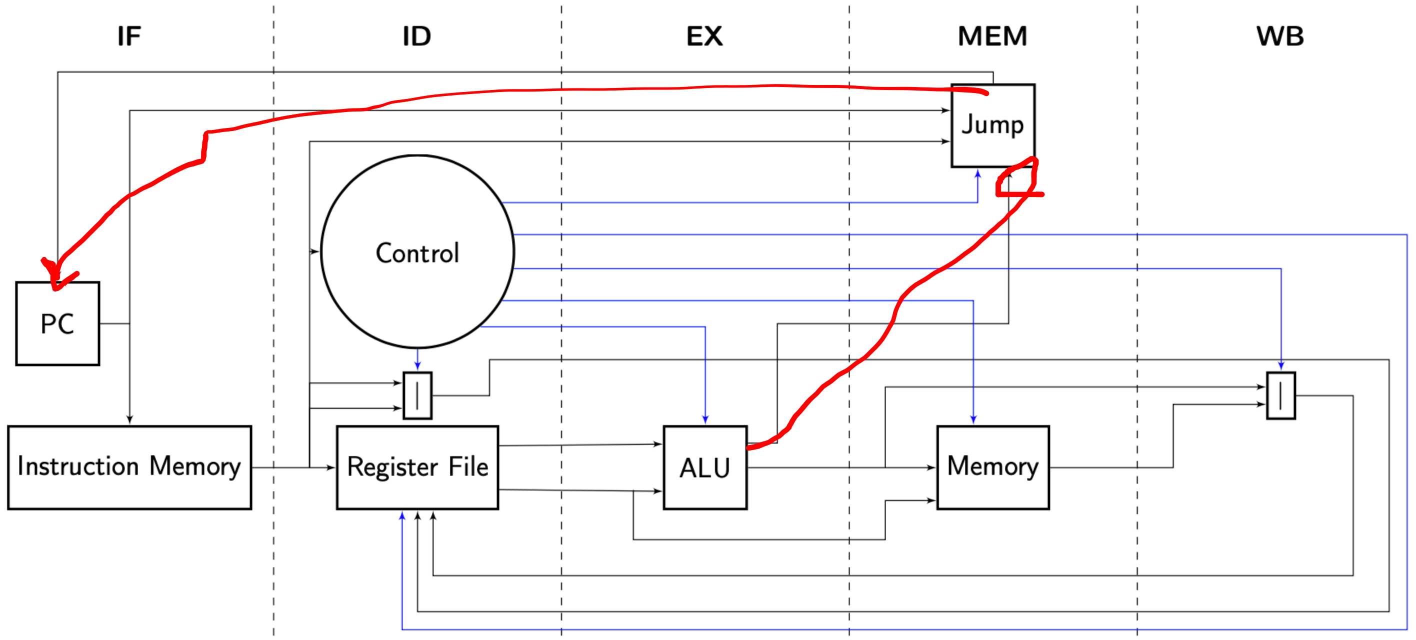
A simple CPU program

```
LOAD_MEM #1234, $1  //Load from memory location 1234 into register 1
LOAD_CONST 1, $2    // Load the value 1 into register 2
ADD $1, $2, $3      // Add register 1 with register 2 and store in register 3
WRITE_MEM $3, #1234 // Write value in register 3 into memory location 1234
```



A simple CPU program with a loop

```
LOAD_MEM #1234, $1    // Load from memory location 1234 into register 1
COPY $1, $2           // Copy the value from register 1 into register 2
LOAD_C 4, $3          // Load the value 3 into register 2
ADD $1, $2, $2        // Add register 1 with register 2 and store in register 2
LOAD_C 1, $4          // Load the value 1 into register $4
SUB $3, $4, $3        // Subtract register 4 from register 3 and store in register 3
JNZ $3, -3            // Jump three instructions back, if register 3 is not zero
WRITE_MEM $2, #1234   // Write value in register 2 into memory location 1234
```



Try it yourself – Modify the program

```
LOAD_MEM #1234, $1   // Load from memory location 1234 into register 1
COPY $1, $2          // Copy the value from register 1 into register 2
LOAD_C 4, $3         // Load the value 3 into register 3
ADD $1, $2, $2       // Add register 1 with register 2 and store in register 2
LOAD_C 1, $4         // Load the value 1 into register $4
SUB $3, $4, $3       // Subtract register 4 from register 3 and store in register 3
JNZ $3, -3           // Jump three instructions back, if register 3 is not zero
WRITE_MEM $2, #1234  // Write value in register 2 into memory location 1234
```

Modify the above program to allow multiplication with a value read from memory

Try it yourself – Modify the program

```
LOAD_MEM #1234, $1    // Load from memory location 1234 into register 1
COPY $1, $2           // Copy the value from register 1 into register 2
LOAD_MEM #1235, $3    // Load from memory location 1234 into register 3
ADD $1, $2, $2        // Add register 1 with register 2 and store in register 2
LOAD_C 1, $4          // Load the value 1 into register $4
SUB $3, $4, $3        // Subtract register 4 from register 3 and store in register 3
JNZ $3, -3            // Jump three instructions back, if register 3 is not zero
WRITE_MEM $2, #1234   // Write value in register 2 into memory location 1234
```

Modify the above program to allow multiplication with a value read from memory

The speed of a CPU

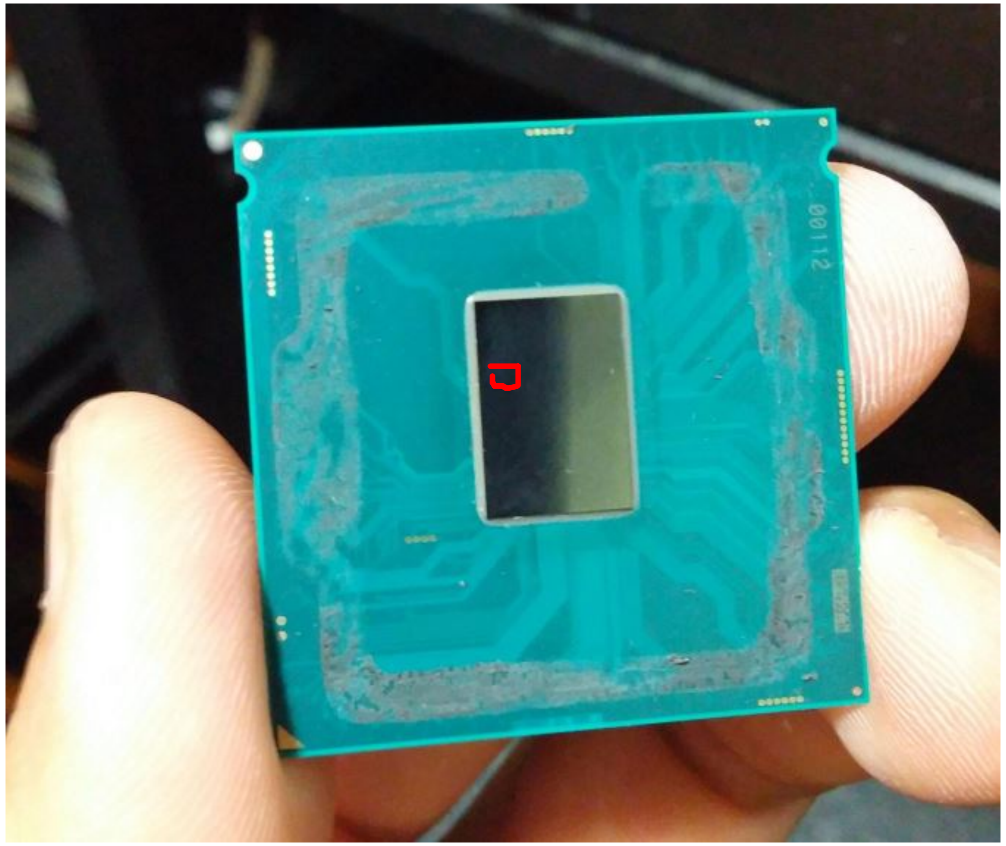


Image from Splave

Intel® Core™ i7-3960X Processor Die Detail

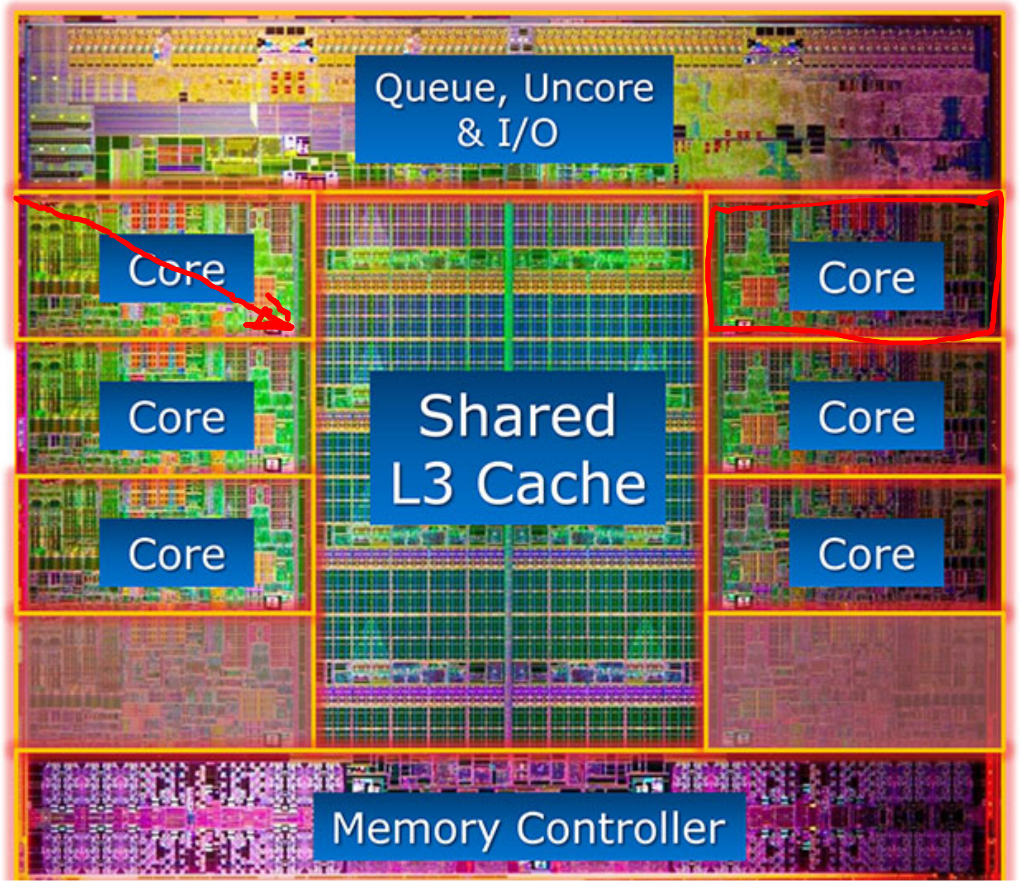
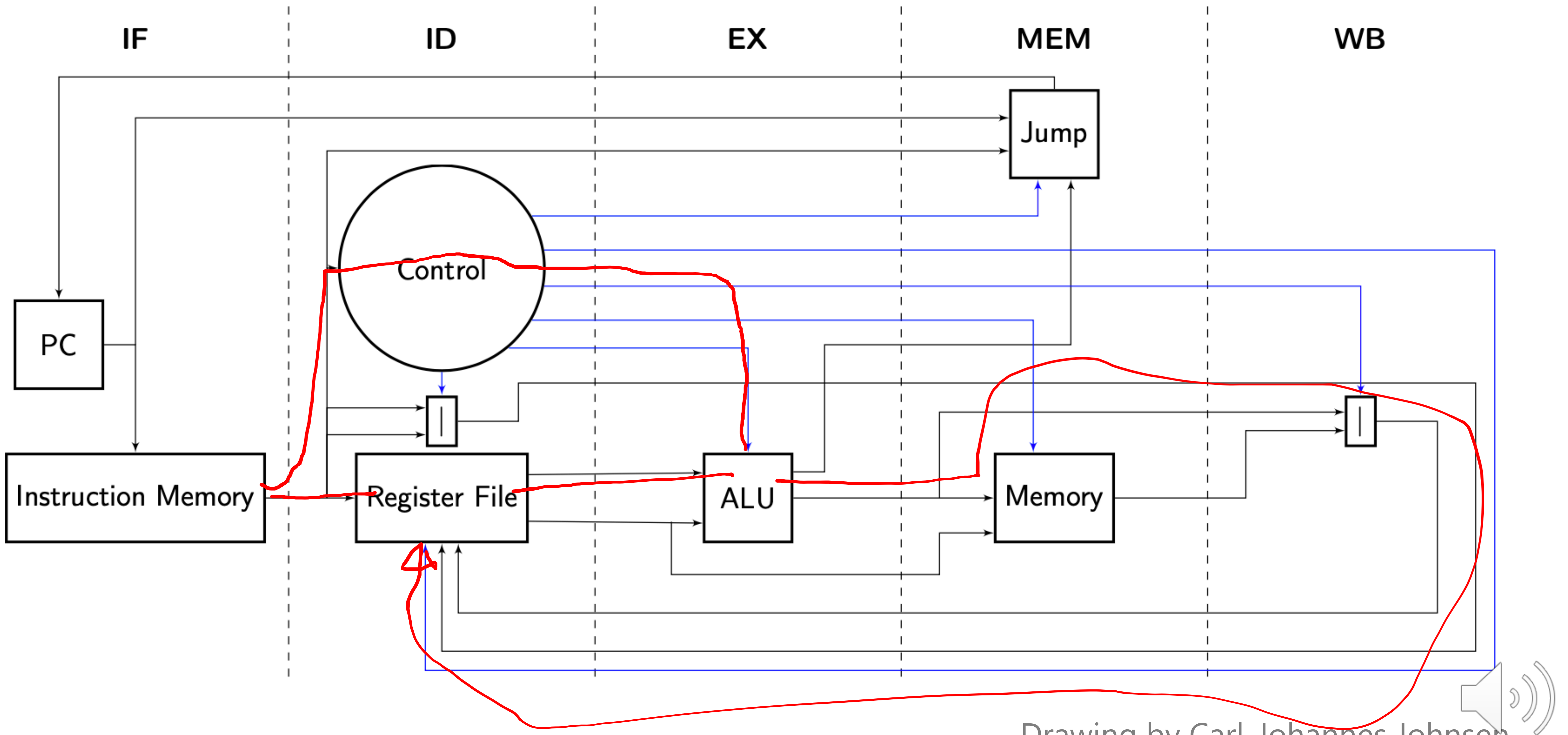


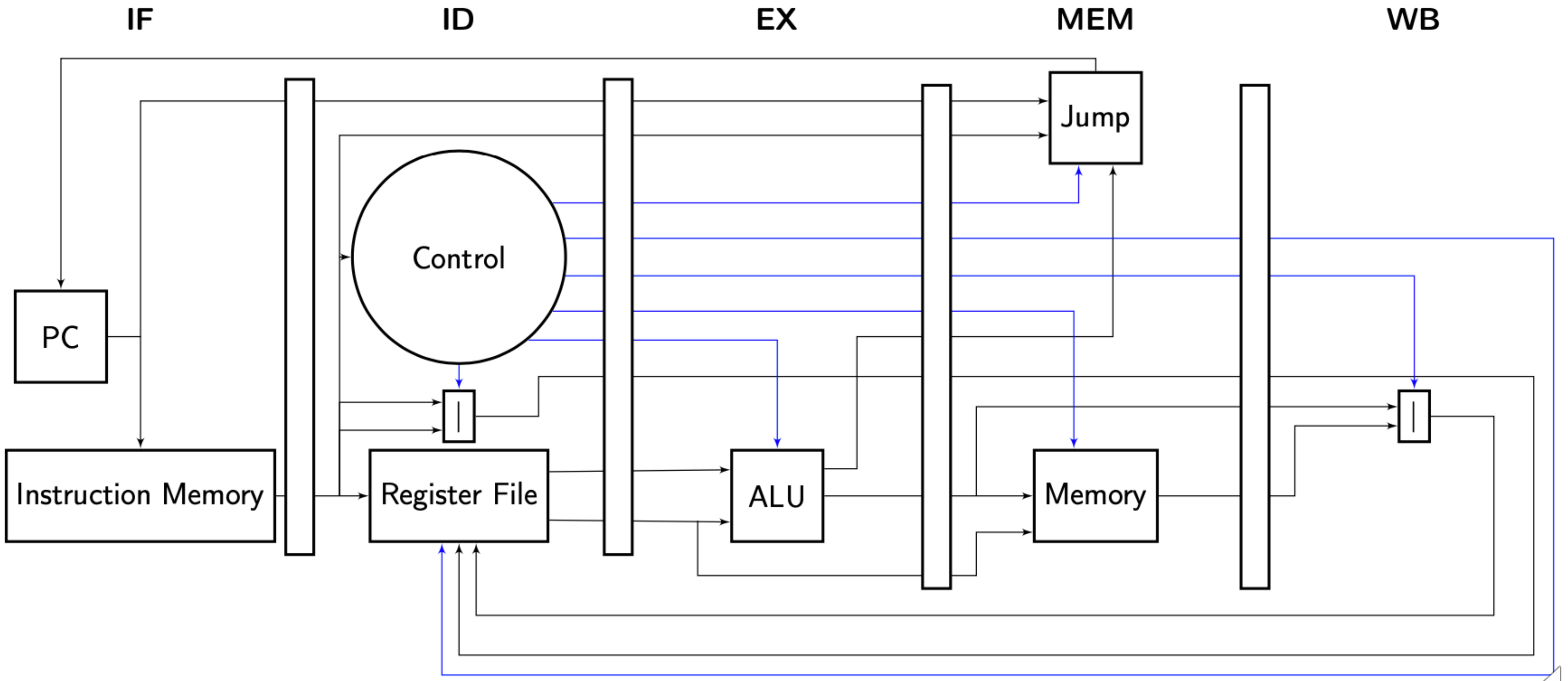
Image from Intel

A simple CPU



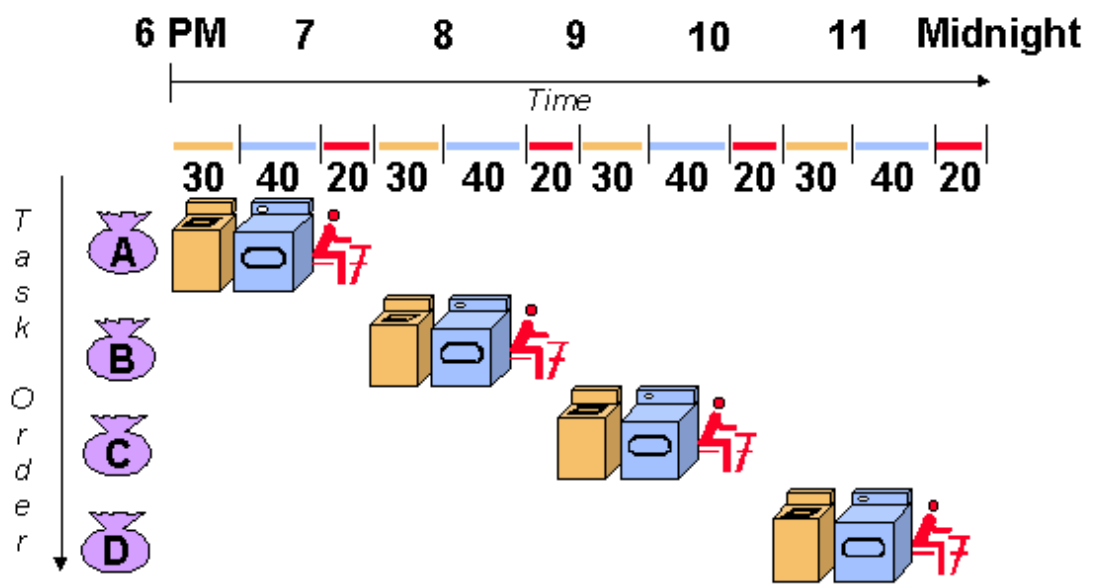
Drawing by Carl-Johannes Johnsen

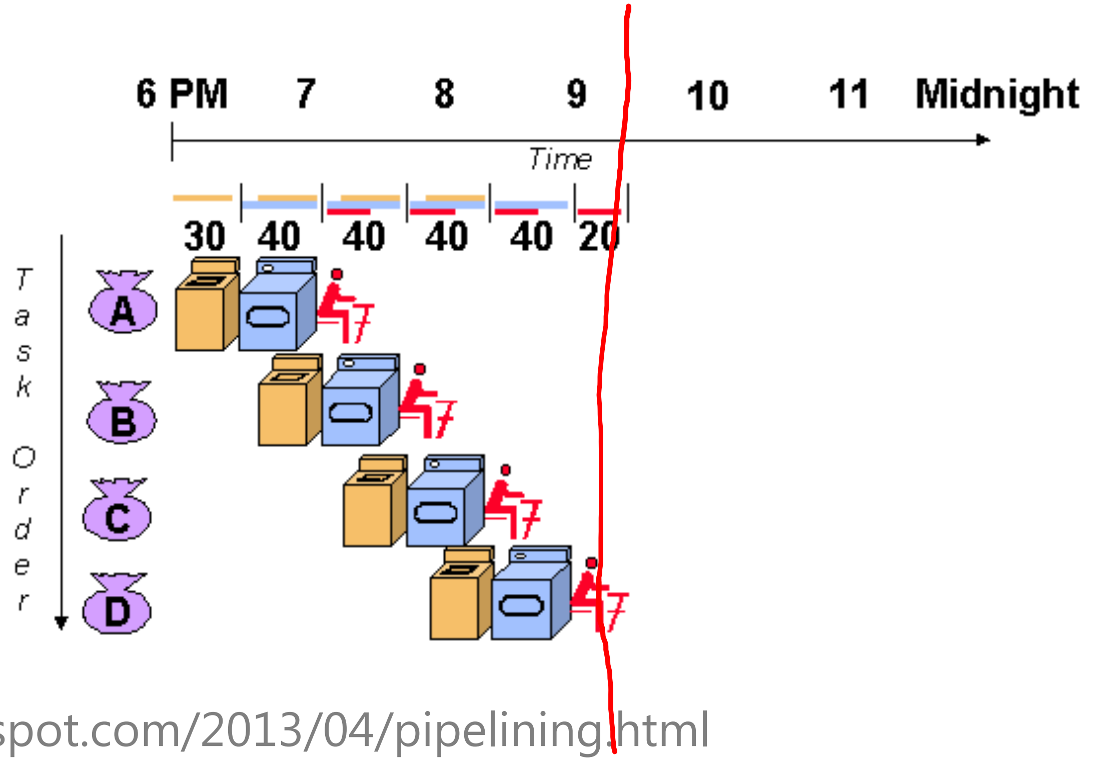
A pipelined CPU



Drawing by Carl-Johannes Johnsen

Pipelining – Useful in practice

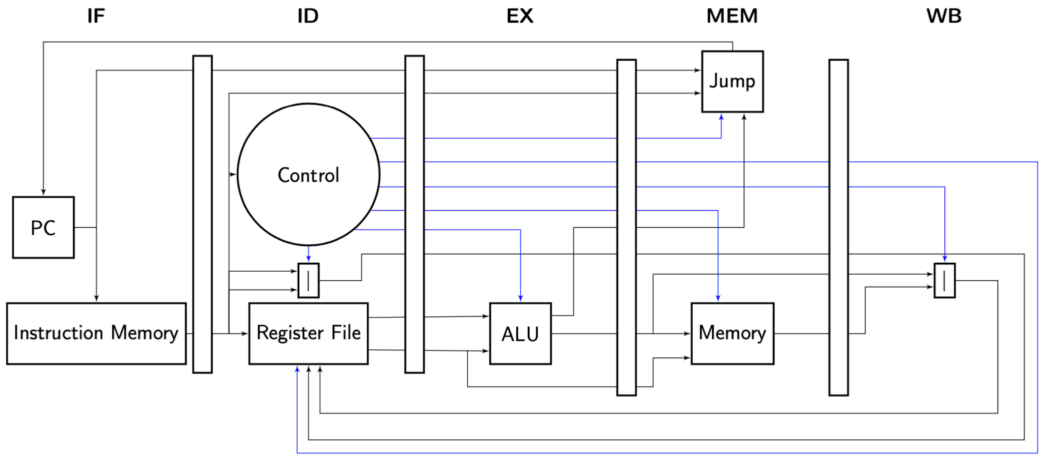




Images from https://project-navi.blogspot.com/2013/04/pipelining.html

A pipelined CPU – Data hazards

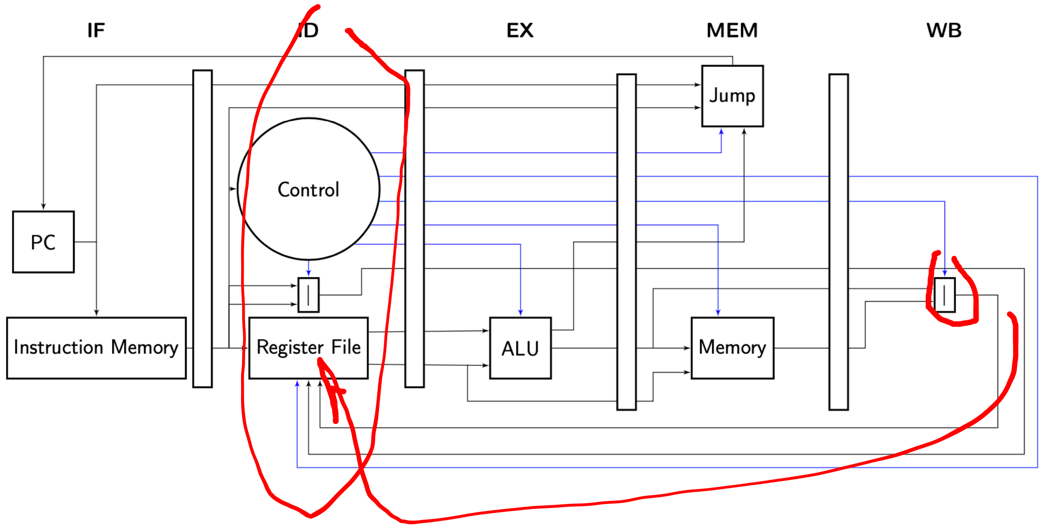
```
LOAD_MEM #1234, $1
LOAD_CONST 1, $2
ADD $1, $2, $3
WRITE_MEM $3, #1234
```



Cycle	IF	ID	EX	MEM	WB
0	LOAD, $1				
1	LOADC, $2	LOAD $1			
2	ADD $1, $2, $3	LOADC, $2	LOAD $1		
3	WRITE $3	ADD $1, $2, $3	LOADC, $2	LOAD $1	
4		WRITE $3	ADD $1, $2, $3	LOADC, $2	LOAD $1
5			WRITE $3	ADD $1, $2, $3	LOADC, $2
6				WRITE $3	ADD $1, $2, $3
7					WRITE $3

A pipelined CPU – Data hazards

```
LOAD_MEM #1234, $1
LOAD_CONST 1, $2
ADD $1, $2, $3
WRITE_MEM $3, #1234
```



Cycle	IF	ID	EX	MEM	WB
0	LOAD, $1				
1	LOADC, $2	LOAD $1			
2	ADD $1, $2, $3	LOADC, $2	LOAD $1		
3	WRITE $3	ADD $1, $2, $3	LOADC, $2	LOAD $1	
4		WRITE $3	ADD $1, $2, $3	LOADC, $2	LOAD $1
5			WRITE $3	ADD $1, $2, $3	LOADC, $2
6				WRITE $3	ADD $1, $2, $3
7					WRITE $3

We read $1 in cycle 3

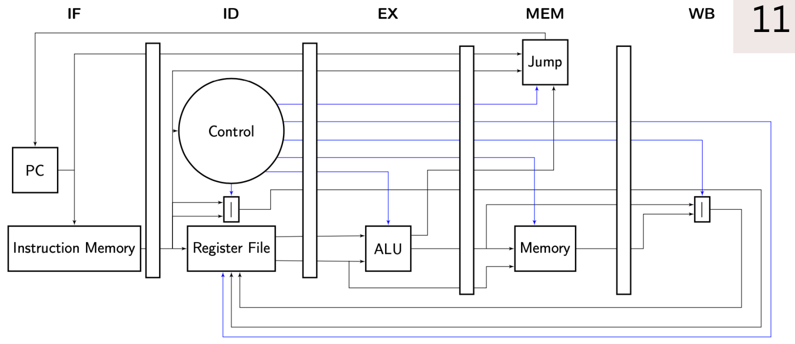
But $1 is written in cycle 4, and cannot be read before cycle 5

A pipelined CPU – Control hazards

```
LOAD_MEM #1234, $1
COPY $1, $2
LOAD_C 4, $3
ADD $1, $2, $2
LOAD_C 1, $4
SUB $3, $4, $3
JNZ $3, -3
WRITE_MEM $2, #1234
```

Cycle	IF	ID	EX	MEM	WB
0	LOAD, $1				
1	COPY $1, $2	LOAD $1			
2	LOADC, $3	COPY $1, $2	LOAD $1		
3	ADD, $1, $2	LOADC, $3	COPY $1, $2	LOAD $1	
4	LOADC, $4	ADD, $1, $2	LOADC, $3	COPY $1, $2	LOAD $1
5	SUB, $3, $4	LOADC, $4	ADD, $1, $2	LOADC, $3	COPY $1, $2
6	JNZ, $3	SUB, $3, $4	LOADC, $4	ADD, $1, $2	LOADC, $3
7	WRITE, $2	JNZ, $3	SUB, $3, $4	LOADC, $4	ADD, $1, $2
8		WRITE, $2	JNZ, $3	SUB, $3, $4	LOADC, $4
9			WRITE, $2	JNZ, $3	SUB, $3, $4
10				WRITE, $2	JNZ, $3
11					WRITE, $2

Do we jump?



RISC or CISC?

CISC – Complex Instruction Set Computer	RISC – Reduced Instruction Set Computer
Hardware focused	Software focused
Multitude of instructions with variations	Limited set of instructions
Fewer registers	More register
Complex addressing modes	Simple addressing modes
Uses microcode (instructions are emulated)	Compiler fixes problems
Instructions can take multiple cycles	Instructions take one cycle
Hard to pipeline	Easy to pipeline

Amdahls Law

Making things faster

$$\text{Speedup} = \frac{\text{execution time before}}{\text{execution time after}}$$

$$\text{Speedup} = \frac{1}{\boxed{\text{non-enhanced part}} + \boxed{\dfrac{\text{enhanced part}}{\text{improvement factor}}}}$$

Speedup – pipeline example

$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \arrow[r] & D \arrow[r] & E \end{tikzcd}$$

	A	B	C	D	E
Before	0.2	0.2	0.2	0.2	0.2
After	0.2	$\underline{0.02}$	0.2	0.2	0.2

$$\text{Speedup} = \frac{\boxed{1}}{\boxed{0.8} + \boxed{0.02}} = \underline{1.3\text{x}}$$

Amdahls Law – Bounds on parallelism

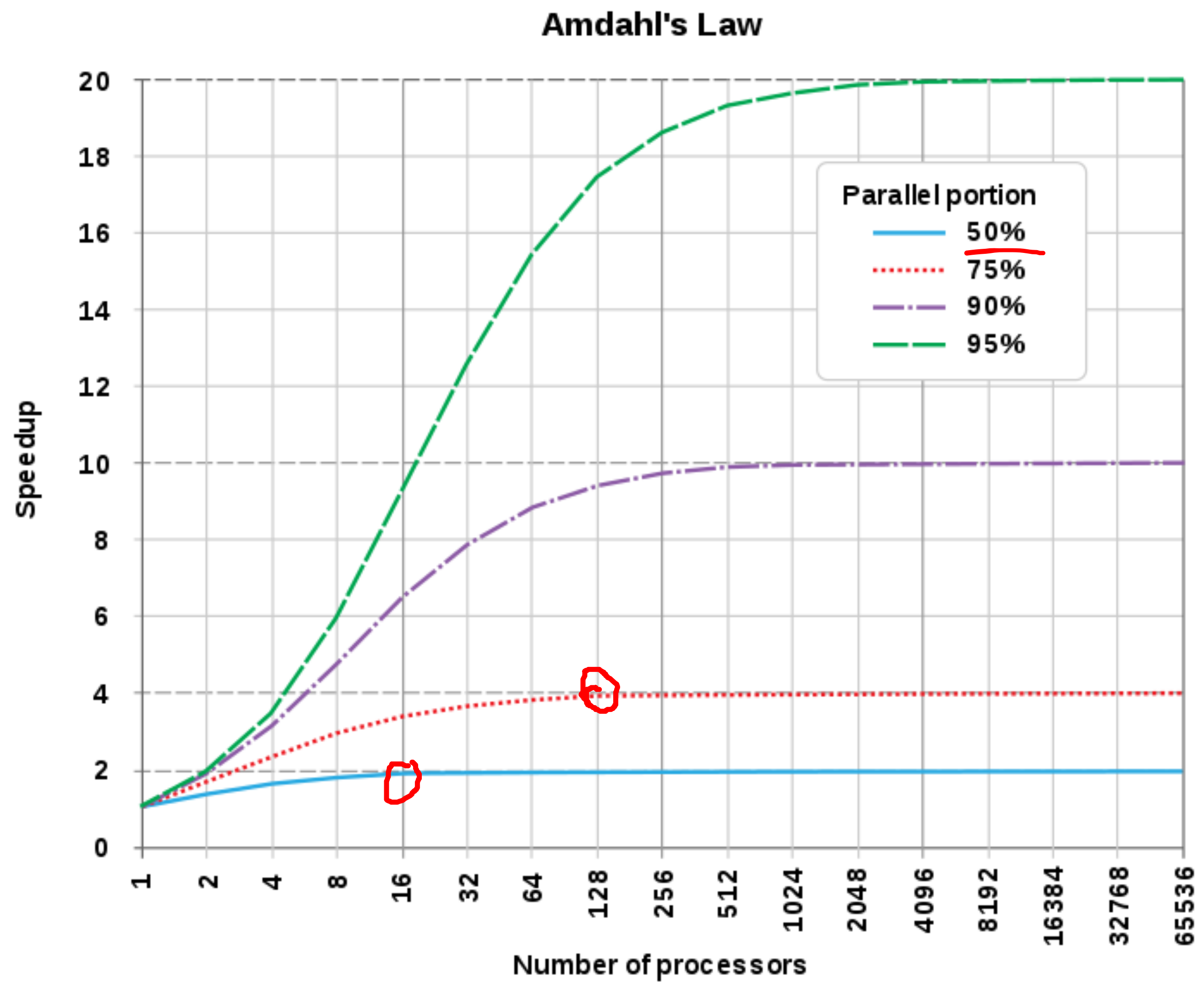
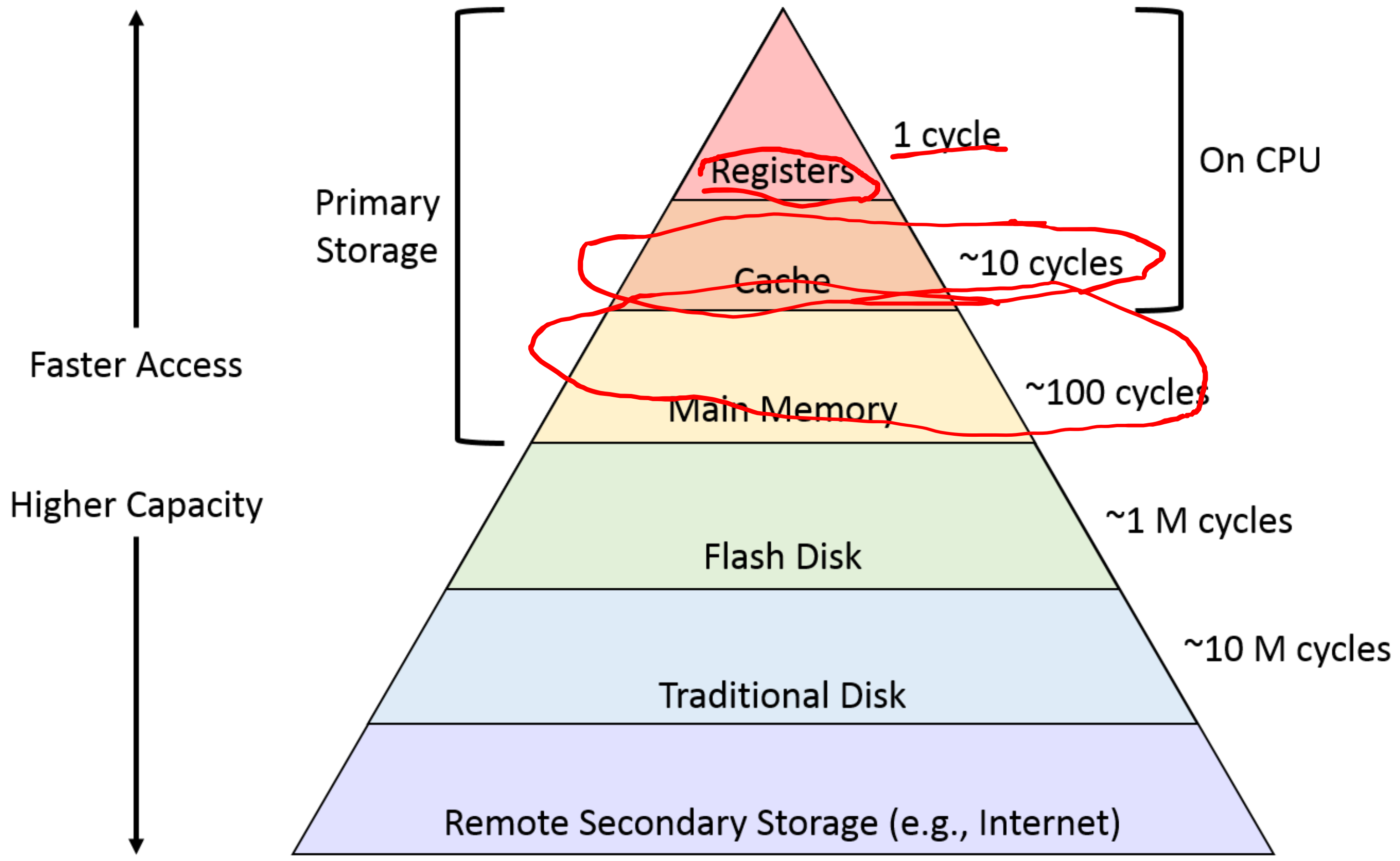


Image from: https://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg

Memory – slowing us down



The Memory Hierarchy

Image from: https://www.cs.swarthmore.edu/~kwebb/cs31/f18/memhierarchy/mem_hierarchy.html

Wrapping up