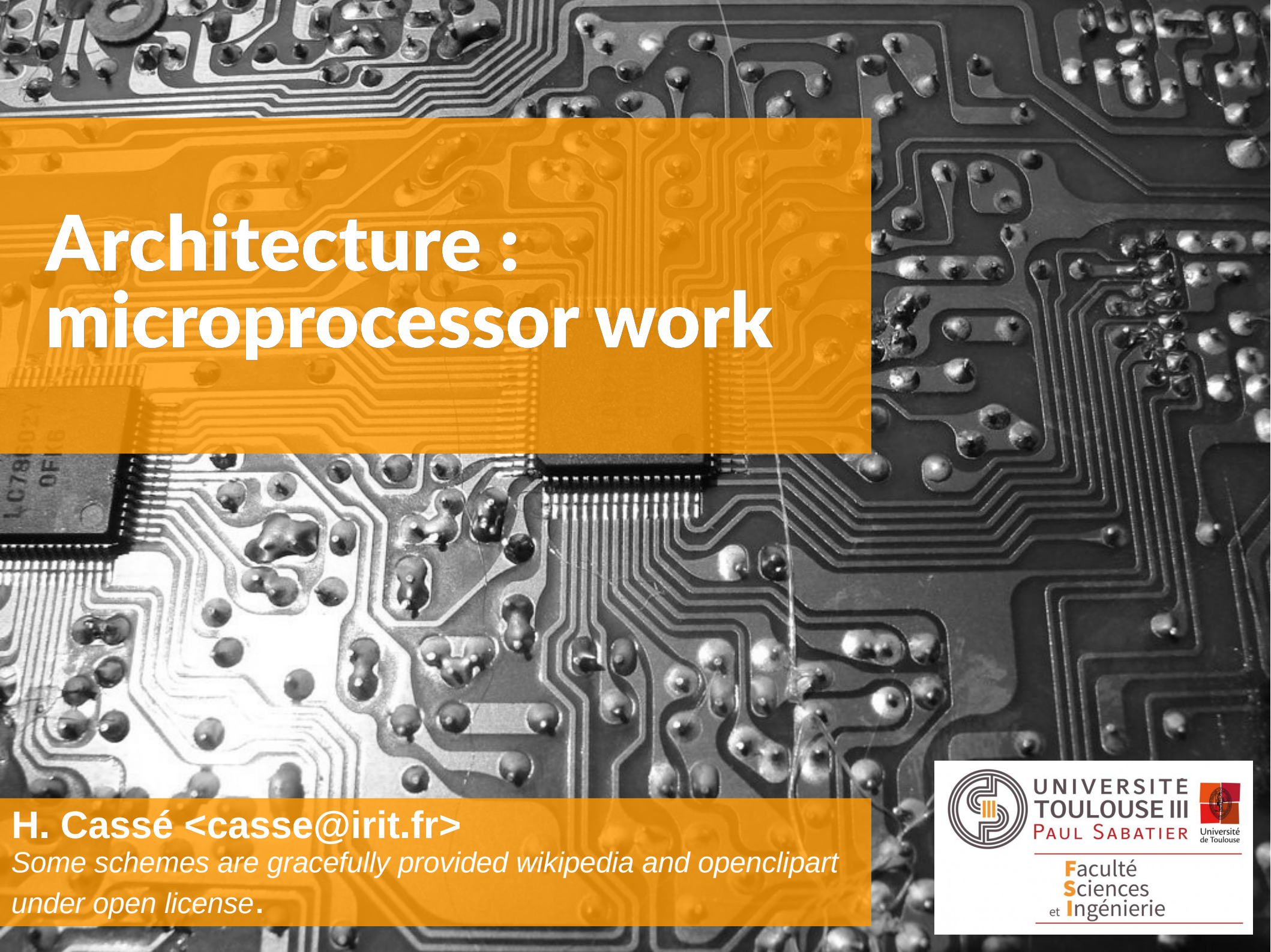
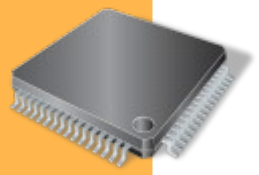


Architecture : microprocessor work

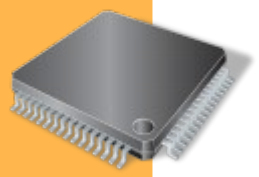
H. Cassé <casse@irit.fr>

UNIVERSITÉ TOULOUSE III PAUL SABATIER
Université de Toulouse
Faculté Sciences et Ingénierie



Introduction

- today, any device
  - automotive, aeronautics, space
  - domestic appliances, traffice lights, ...
  - contains a microprocessor!
- embedded application
  - software → smart device
  - hardware running the software
  - based on transistors + memory technologies



Embedded microprocessor market

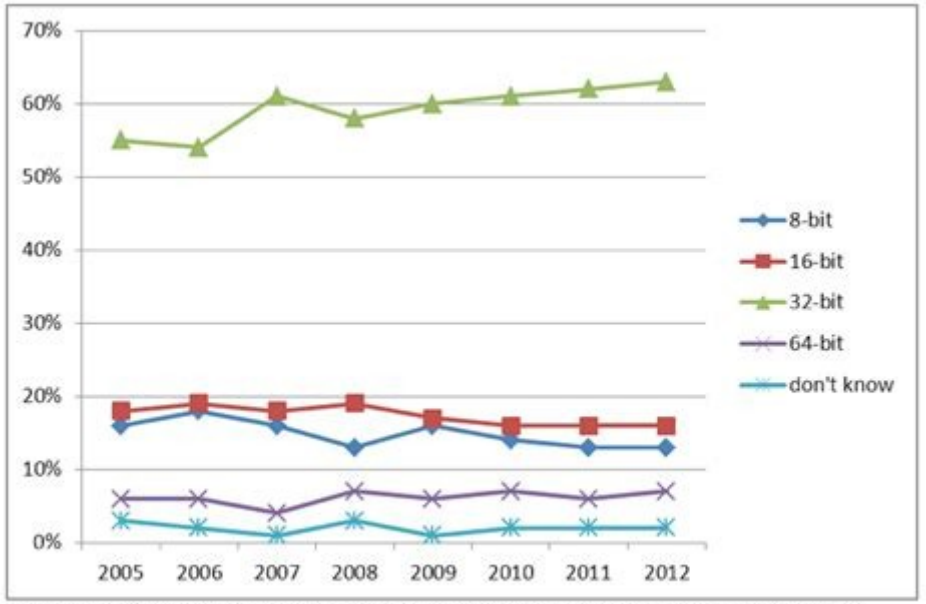
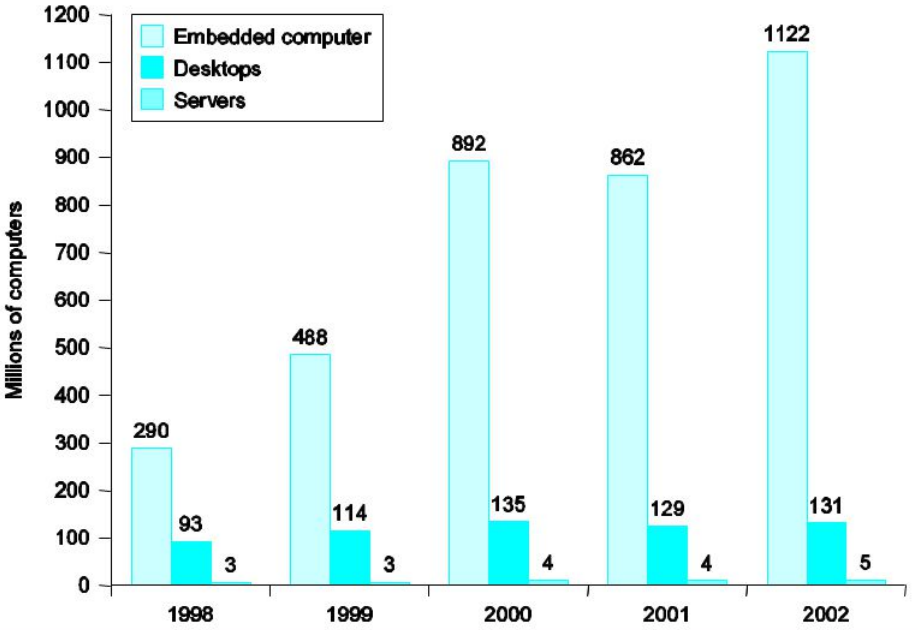
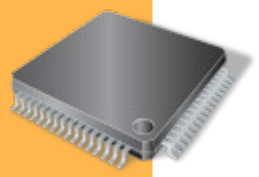


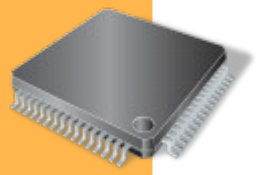
Figure 1: Responses to "My current embedded project's main processor is ...".



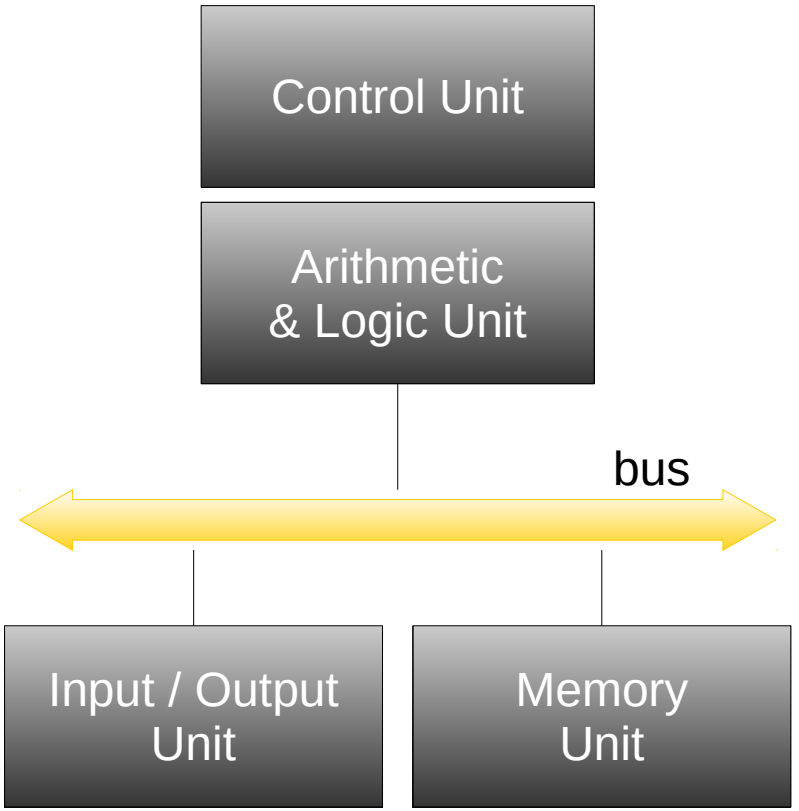


Comparison

- Laptop/Desktop
  - 64-bit
  - 1-3 GHz
  - 4-8 GB
  - x86 – Intel, AMD
- Mobile
  - 32/64-bits
  - 1-3 Ghz
  - 0.5-2 Gb
  - ARM – Nvidia, Qualcom
- Embedded
  - 8, 16, 32, 64-bit
  - 10-500 MHz
  - 1–100 MB
  - AVR, TriCore, ARM, Sparc, Mips, x86 (32-bit), PowerPC, ...
  - DSP, FPGA

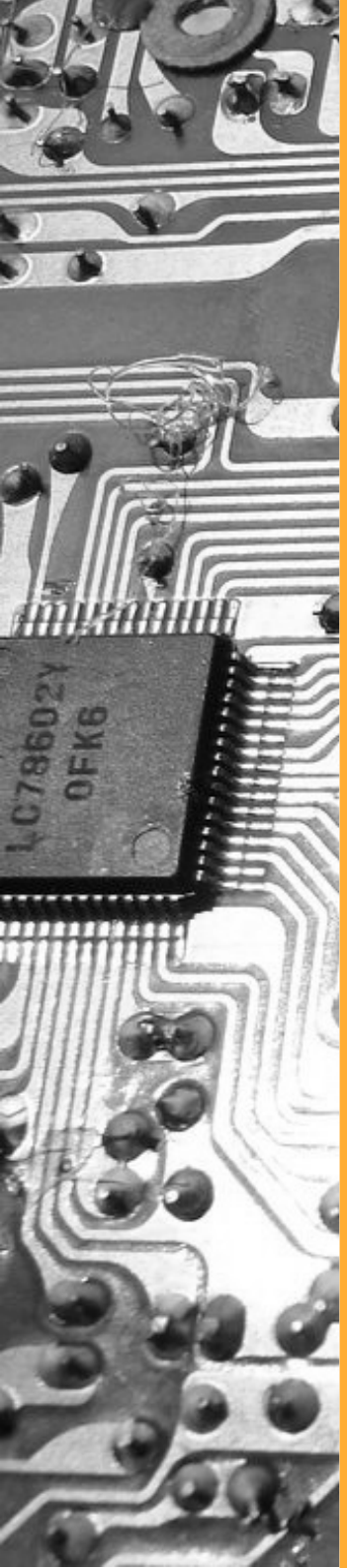


Von Neuman's machine



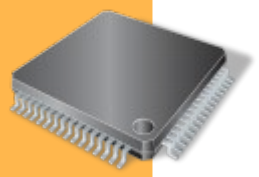
loop

(0) read instruction
(1) read operands
(2) compute
(3) write result



Overview

- Introduction
- **Making basic circuits**
- Making the memory
- Making the processor
- Conclusion

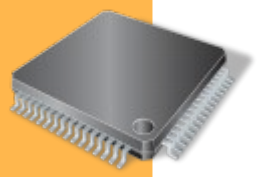


How to implement it?

- number encoding – bit (Binary digIT)
  - base-2 numbers – 0 or 1
- natural number encoding

  $N = (B_n B_{n-1} \ldots B_1 B_0)_2$

  $= B_n 2^n + B_{n+1} 2^{n+1} + \ldots + B_1 2^1 + B_0 2^0$
- integer encoding – 2's complement
  - (-N) s.t. N + (-N) = 0
  - $(-N)_2 = (\overline{B}_n \overline{B}_{n-1} \ldots \overline{B}_1 \overline{B}_0) \pm 1$
- note
  - A + B = OR, A . B = AND, $\overline{A}$ = NOT



Designing circuits

- B = {0, 1} → Boolean algebra
- designing circuit = designing Boolean function(s)
  - identifying boolean input
  - writing a truth table
  - express the output in con(dis)junctive normal form
- (B, NOT, AND, OR) complete group

  any function can be computed as a combination of these operators
- minimization
  - Karnaugh table (≤ 6 inputs)
  - Quine – Mc Cluskey algorithm

Example: Adder 1-bit

$$2 \times C_{out} + S = A + B + C_{in}$$

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$F_{cout}(A, B, C_{in}) = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

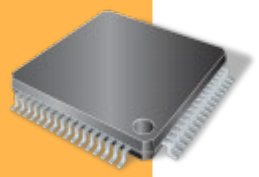
$$F_{S}(A, B, C_{in}) = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$$

Electronic notation

$\overline{A}$ = NOT A

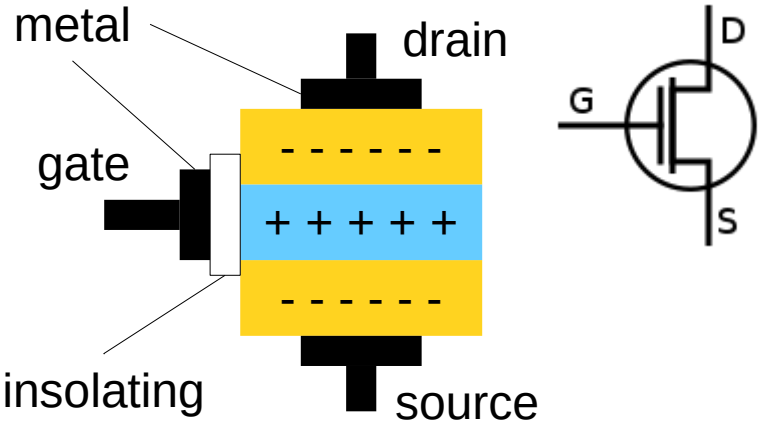
A . B = A AND B

A + B = A OR B

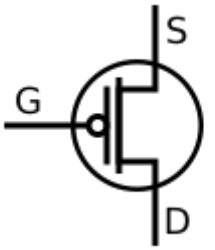


How to implement Boolean operator as NOT?

Convention: 0 = 0V, 1 = 1.8V or 3.3 or 5V)



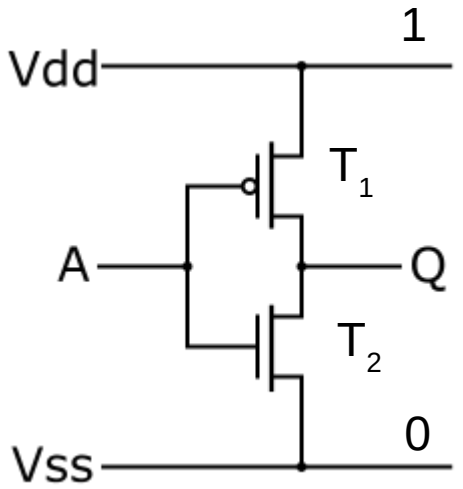
p-channel variant



MOSFET (Metal-Oxid-Semiconductor Field-Effect Transistor) n-channel

gate = 0 ==> no current (off)
gate = 1 ==> current flows from source to drain (on)

How to implement Boolean operator as NOT?



NOT implementation

Vdd = 1

Vss = 0

when A = 1

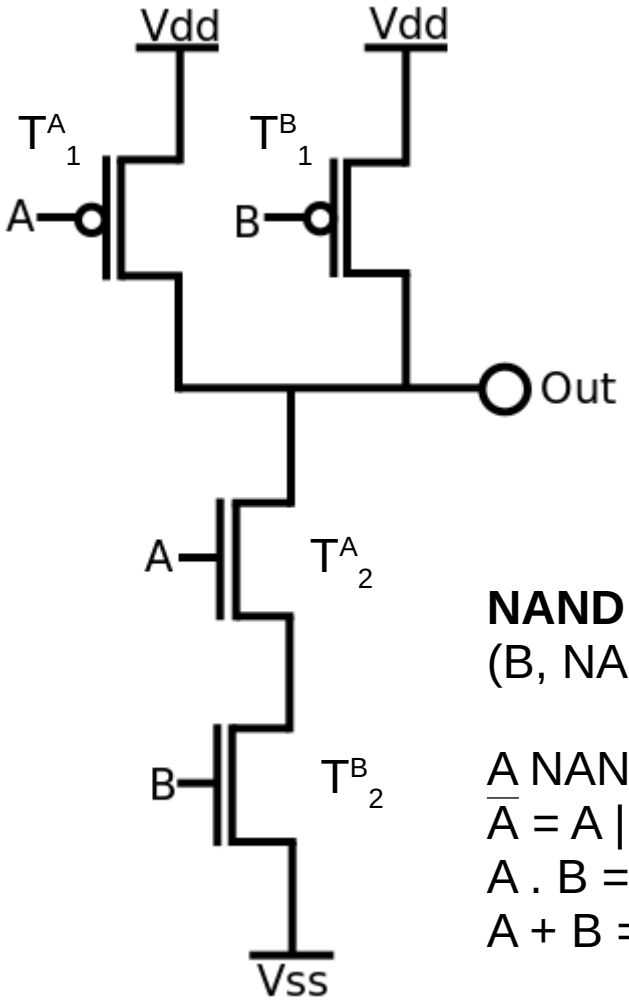
- T_1 off
- T_2 on
- Q = 0

when A = 0

- T_1 on
- T_2 off
- Q = 1

CMOS (Complementary Metal–Oxide–Semiconductor)

Implementing NAND (NOR)



NAND implementation
(B, NAND) complete group

A NAND B = A | B = $\overline{A \,.\, B}$
$\overline{A}$ = A | A
A . B = (A | A) | (B | B)
A + B = (A | B) | (A | B)

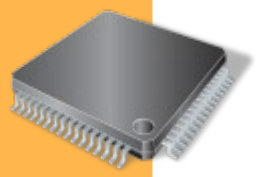
The same with NOR.

Vdd = 1
Vss = 0
when A = 1 and B = 1
- T^A_1, T^B_1 off
- T^A_2, T^B_2 on
- Out = 0

when A = 0 and B = 1
- T^A_1, T^B_2 on
- T^B_1, T^A_2 off
- Out = 1

when A = 1 and B = 0
- T^A_1, T^B_2 off
- T^B_1, T^A_2 on
- Out = 1

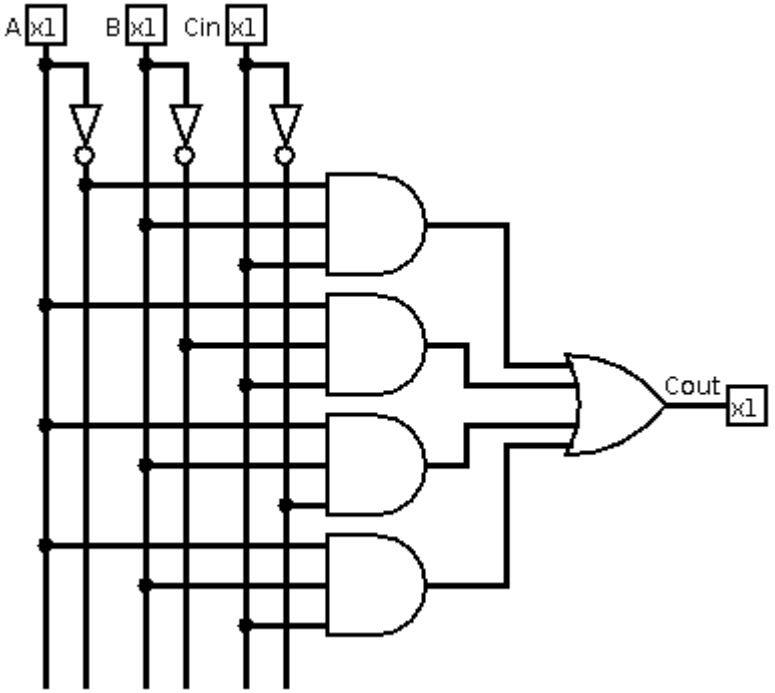
when A = 0 and B = 0
- T^A_1, T^B_1 on
- T^A_2, T^B_2 off
- Out = 1

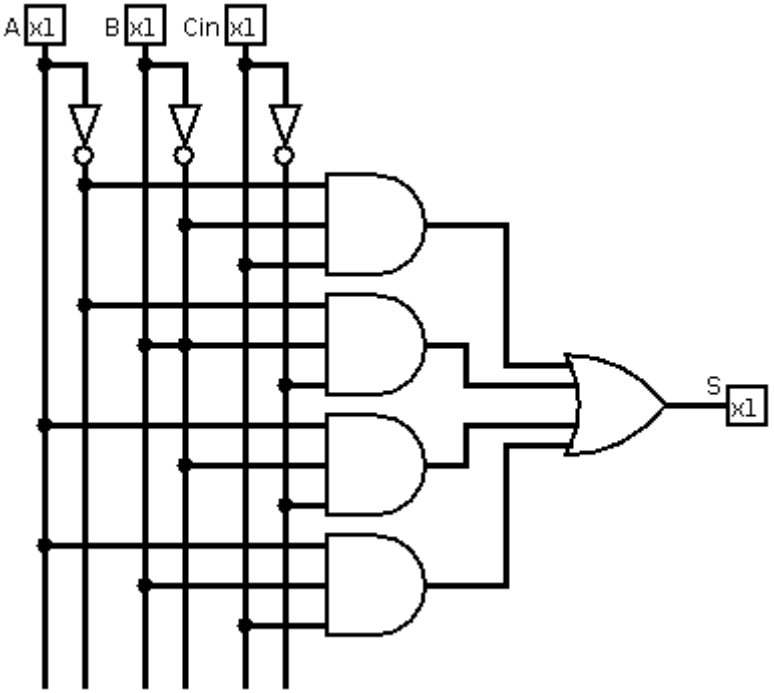


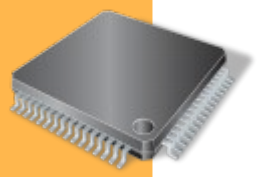
Composing gates to make a circuit

$$F_{cout}(A, B, C_{in}) = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

$$F_{S}(A, B, C_{in}) = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$$



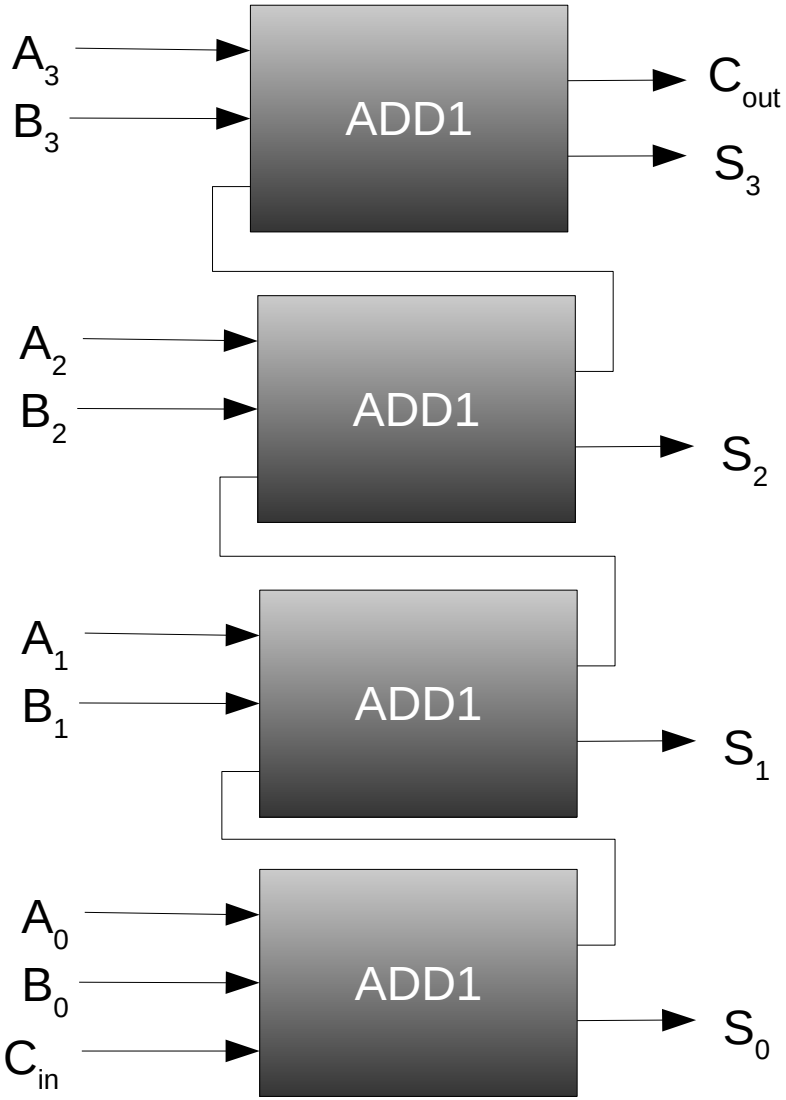


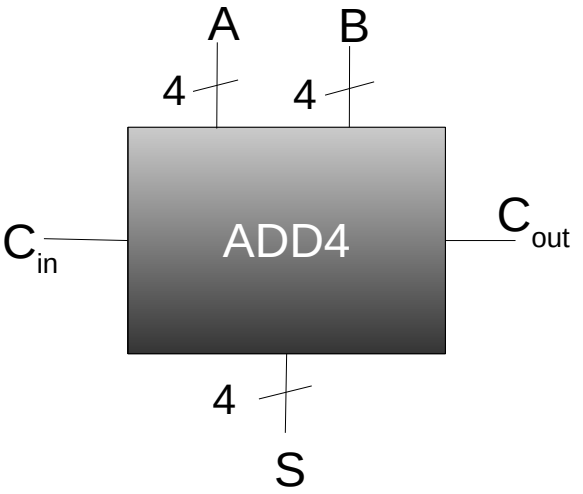


Designing a basic circuit

- identify inputs
- identify outputs → function to implement
- write a truth table for each function / output
- write the equation for each function
  (using conjonctive or disjonctive normal form)
- implement the circuit using AND / OR / NOT gates
  (automatic)
- build the circuit using transistors
  (synthesis – mostly automatic)

Composing circuits to make bigger circuits

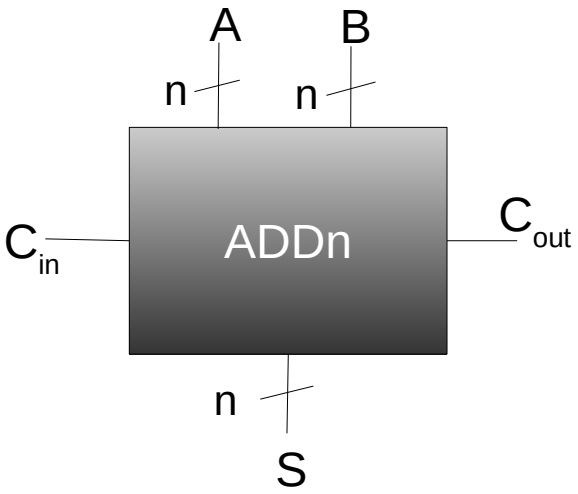


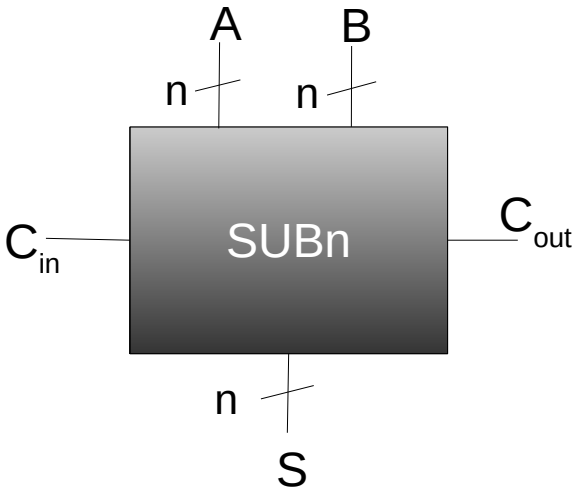


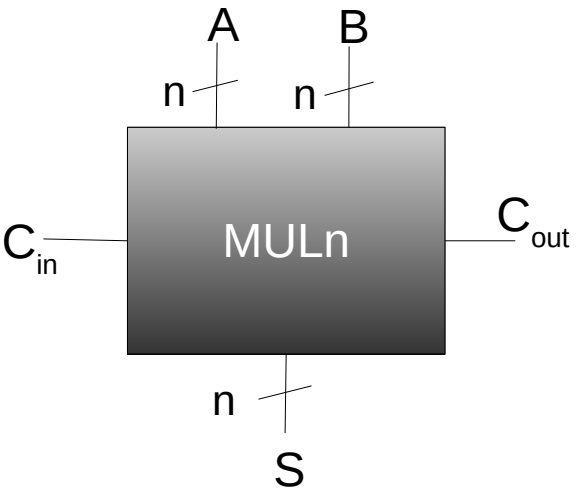
$A = A_3\ A_2\ A_1\ A_0$
$B = B_3\ B_2\ B_1\ B_0$
$S = S_0\ S_2\ S_1\ S_0$

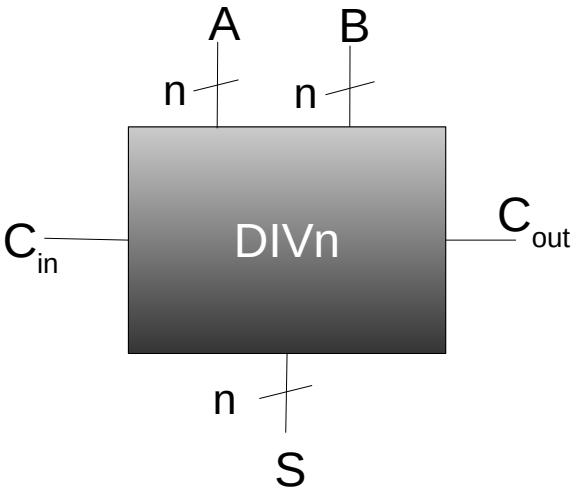
$S_0 = A_0 + B_0 + C_{in}$
$S_1 = A_1 + B_1 + C_{out,0}$
$S_2 = A_2 + B_2 + C_{out,1}$
$S_3 = A_3 + B_3 + C_{out,2}$

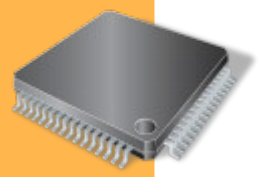
Arithmetic circuits



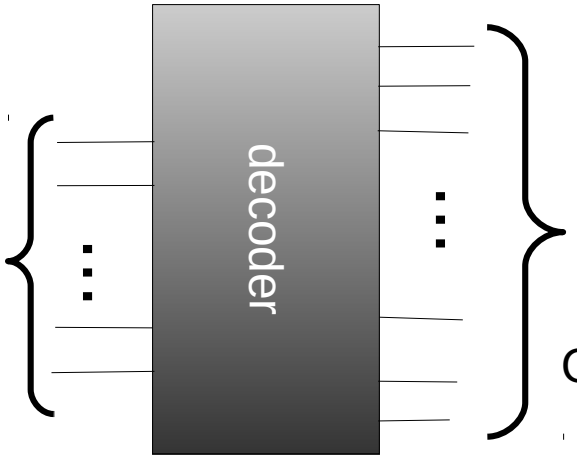


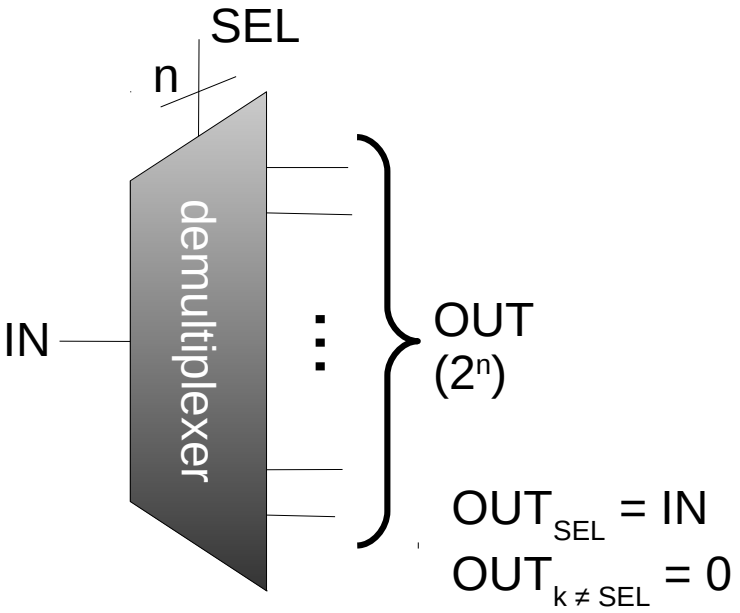


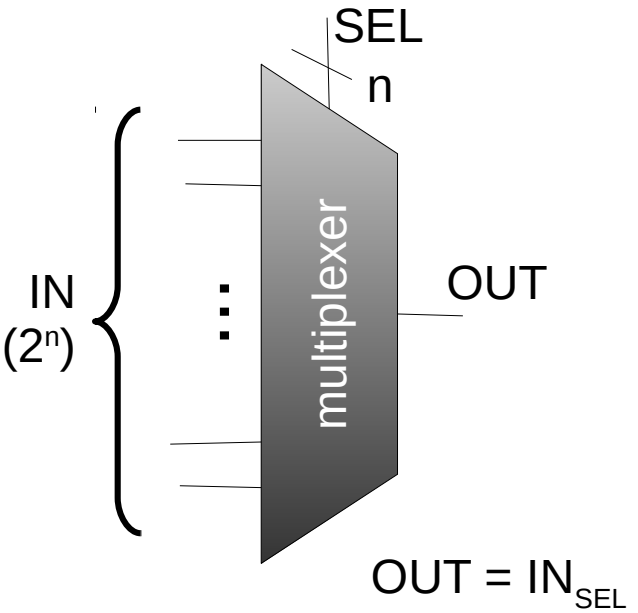


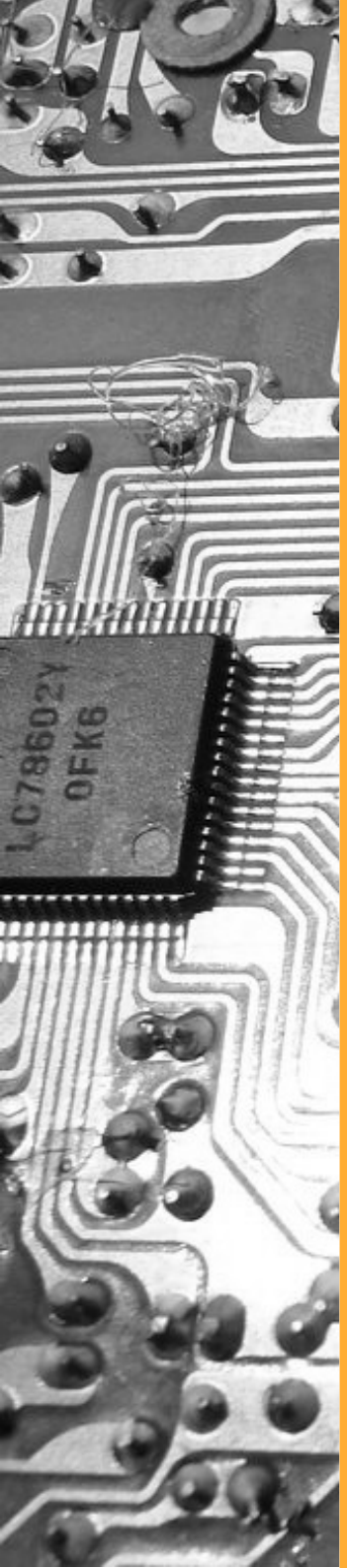


Useful circuit





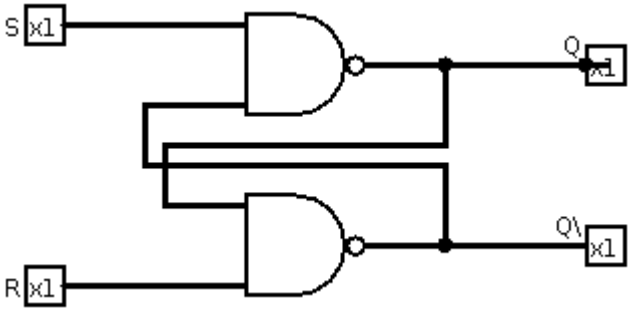




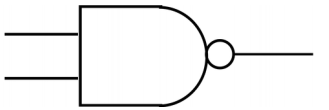
Overview

- Introduction
- Making basic circuits
- **Making the memory**
- Making the processor
- Conclusion

1-bit memory: flip-flop

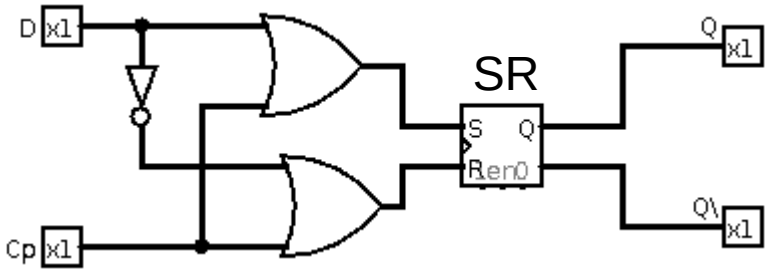


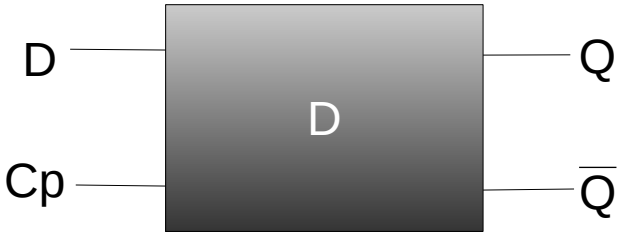
S^t	R^t	Q^{t+1}	$\overline{Q}^{t+1}$
1	1	Q^t	$\overline{Q}^t$
1	0	0	1
0	1	1	0
0	0	X	X



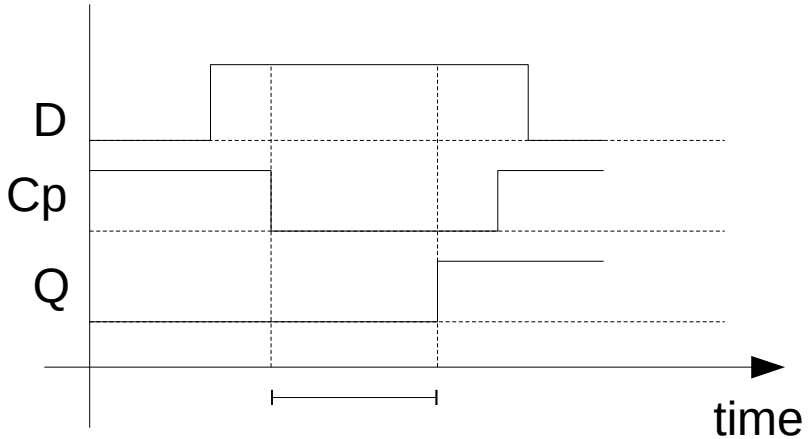
A NAND B = $\overline{A \cdot B}$

D flip-floop





D^t	Cp^t	Q^{t+1}	$\overline{Q}^{t+1}$
X	1	Q^t	$\overline{Q}^t$
0	0	0	1
1	0	1	0

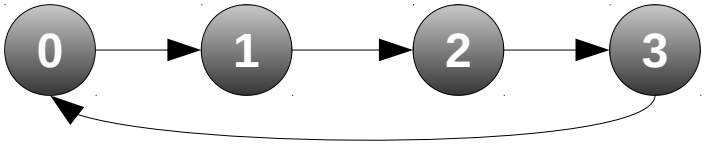


setup time (t_D)

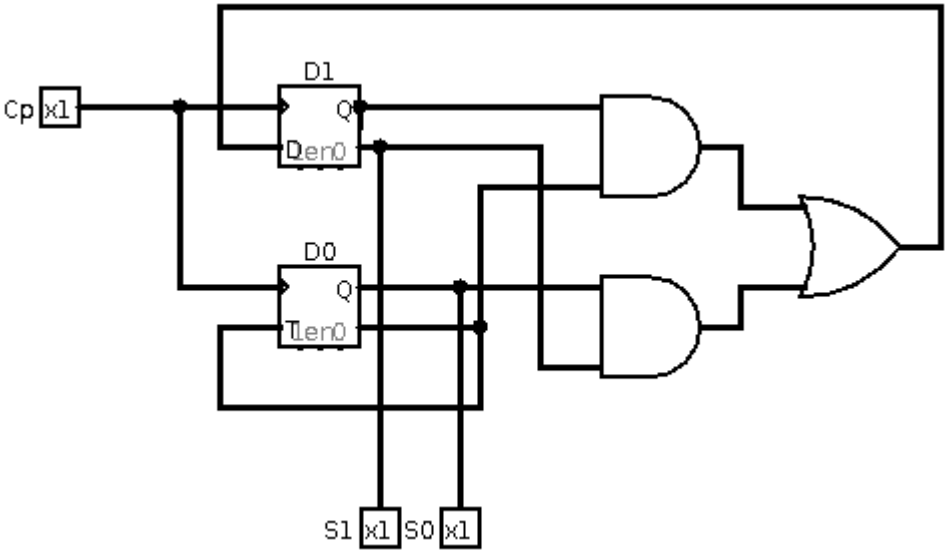
(chornogram)

Von Neuman machine: control unit automaton

loop
- (0) read instruction
- (1) read operands
- (2) compute
- (3) write result



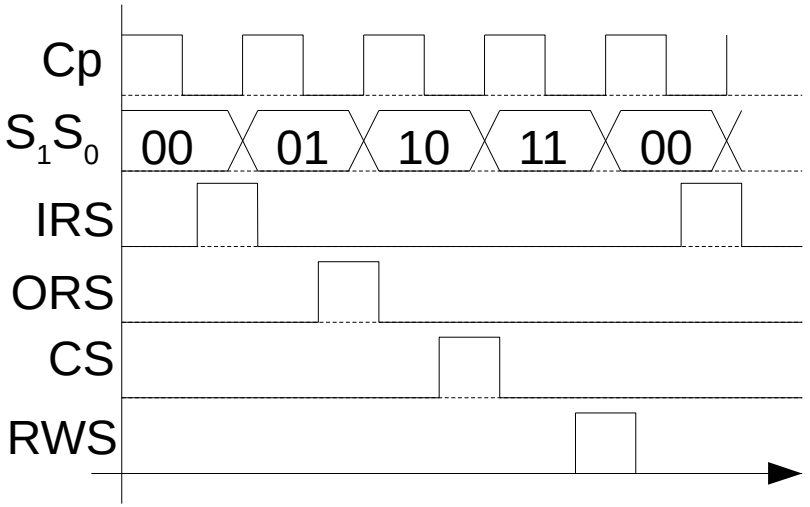
Q^t_1	Q^t_0	Q^{t+1}_1	Q^{t+1}_0	D^t_1	D^t_0
0	0	0	1	0	1
0	1	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	0

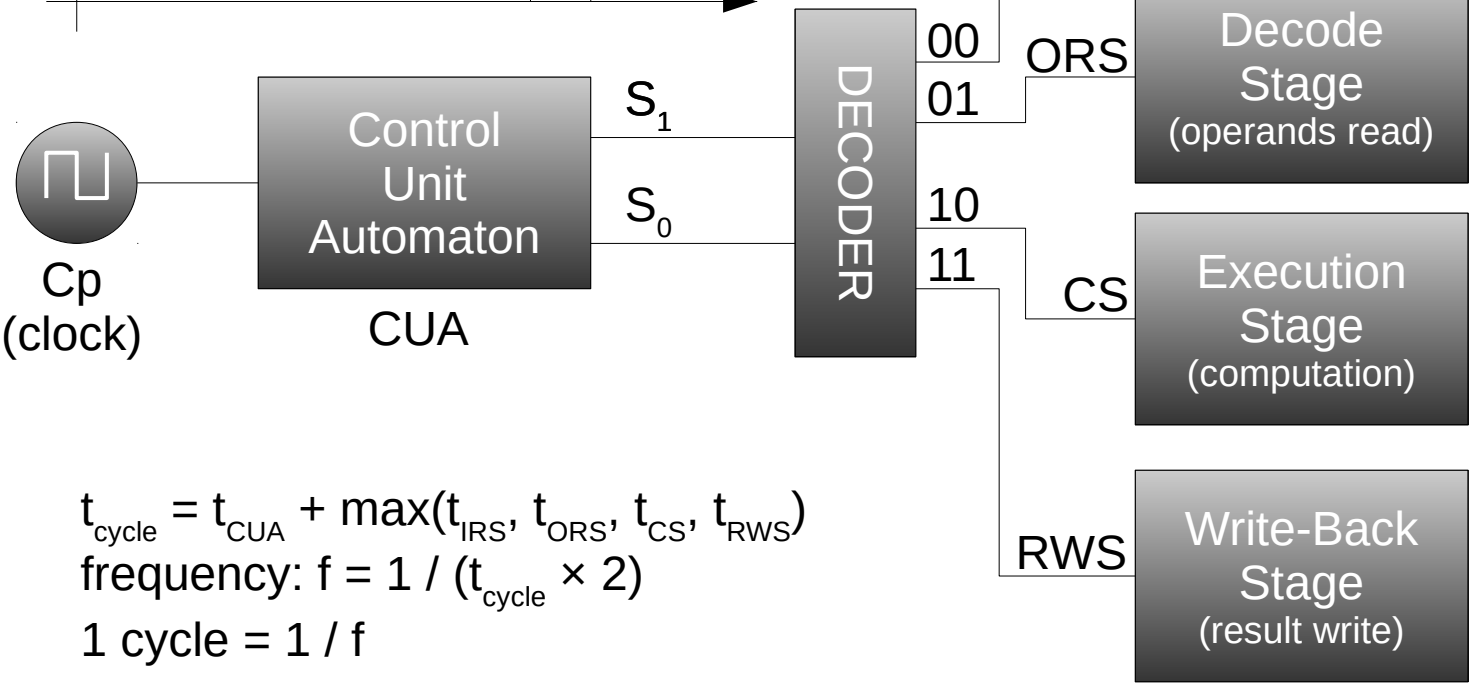


$$D^t_0 = \overline{Q}^t_0$$

$$D^t_1 = \overline{Q}^t_1 \, Q^t_0 + Q^t_1 \, \overline{Q}^t_0$$

Von Neuman machine: control unit

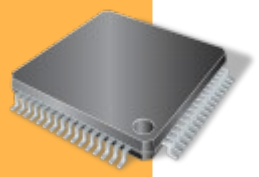




$t_{cycle} = t_{CUA} + \max(t_{IRS}, t_{ORS}, t_{CS}, t_{RWS})$

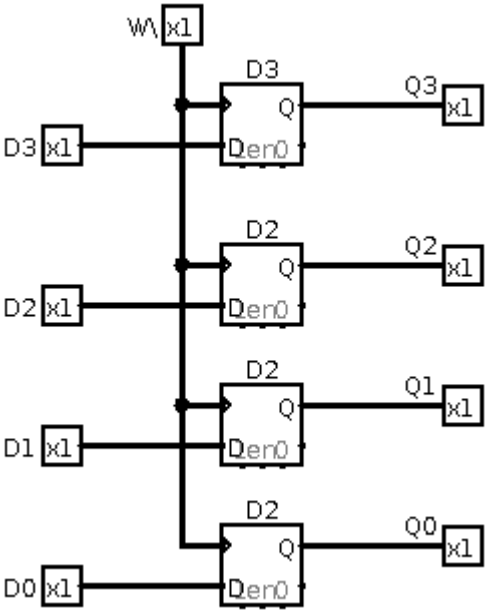
frequency: $f = 1 / (t_{cycle} \times 2)$

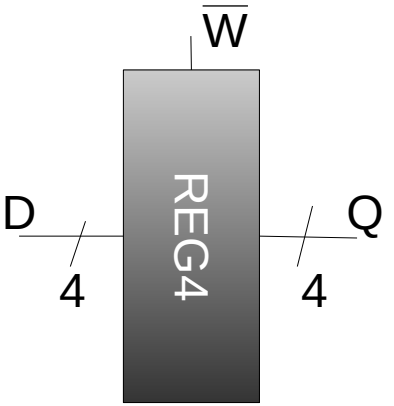
1 cycle = 1 / f



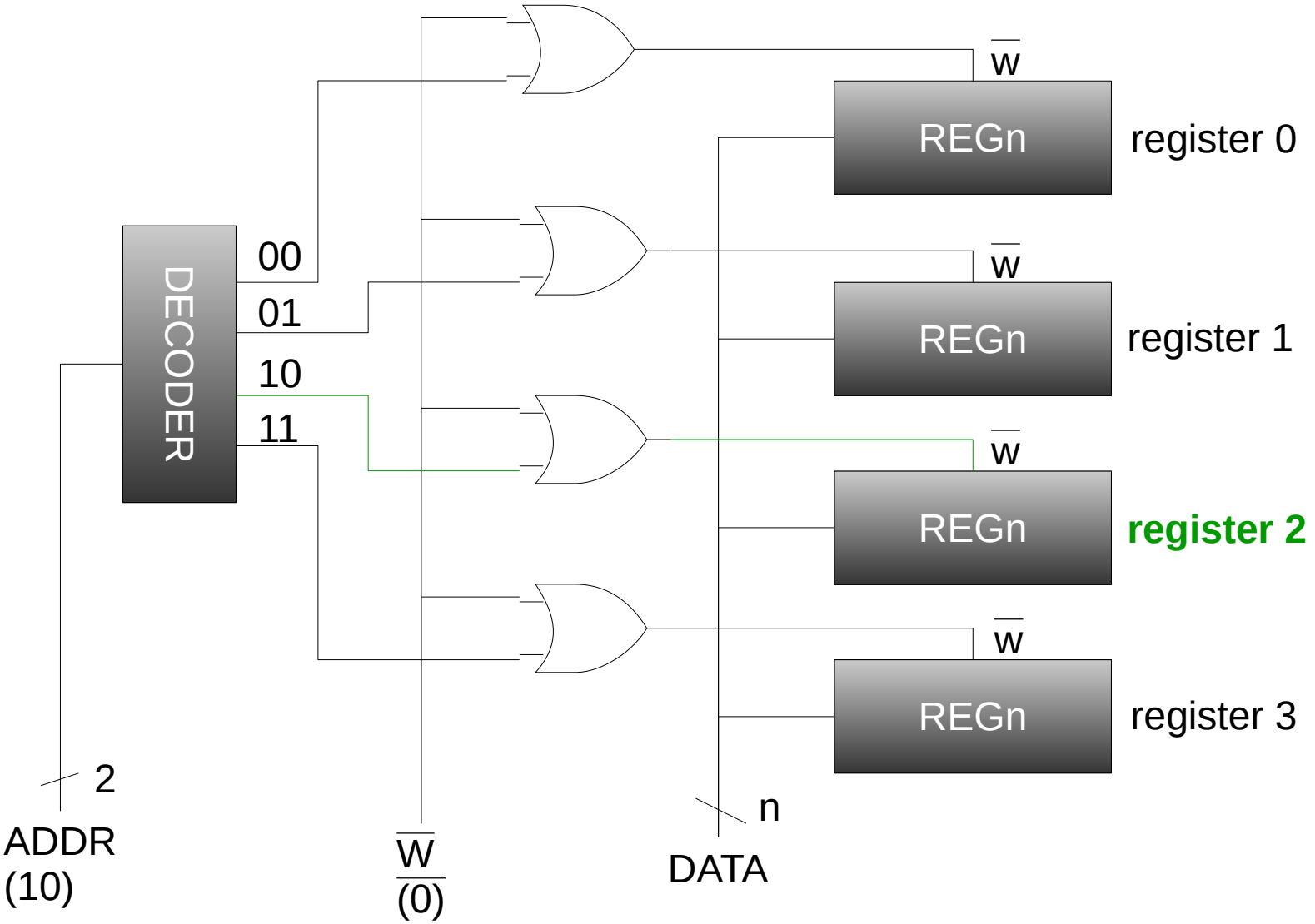
Hardware register

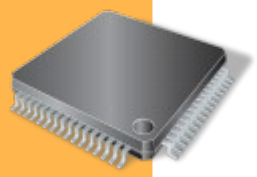
- machine word = n bit
  - n = 64 in laptop
  - n = 8 / 16 / 32 /64 in embedded systems
- hardware register
  - set of D flip-flop
  - storage for a word
  - Cp = set signal
- read operation
- write operation
  - set value on D
  - set 0 on $\overline{W}$





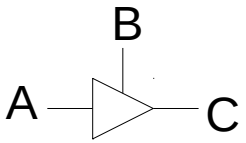
Register bank (write operation)



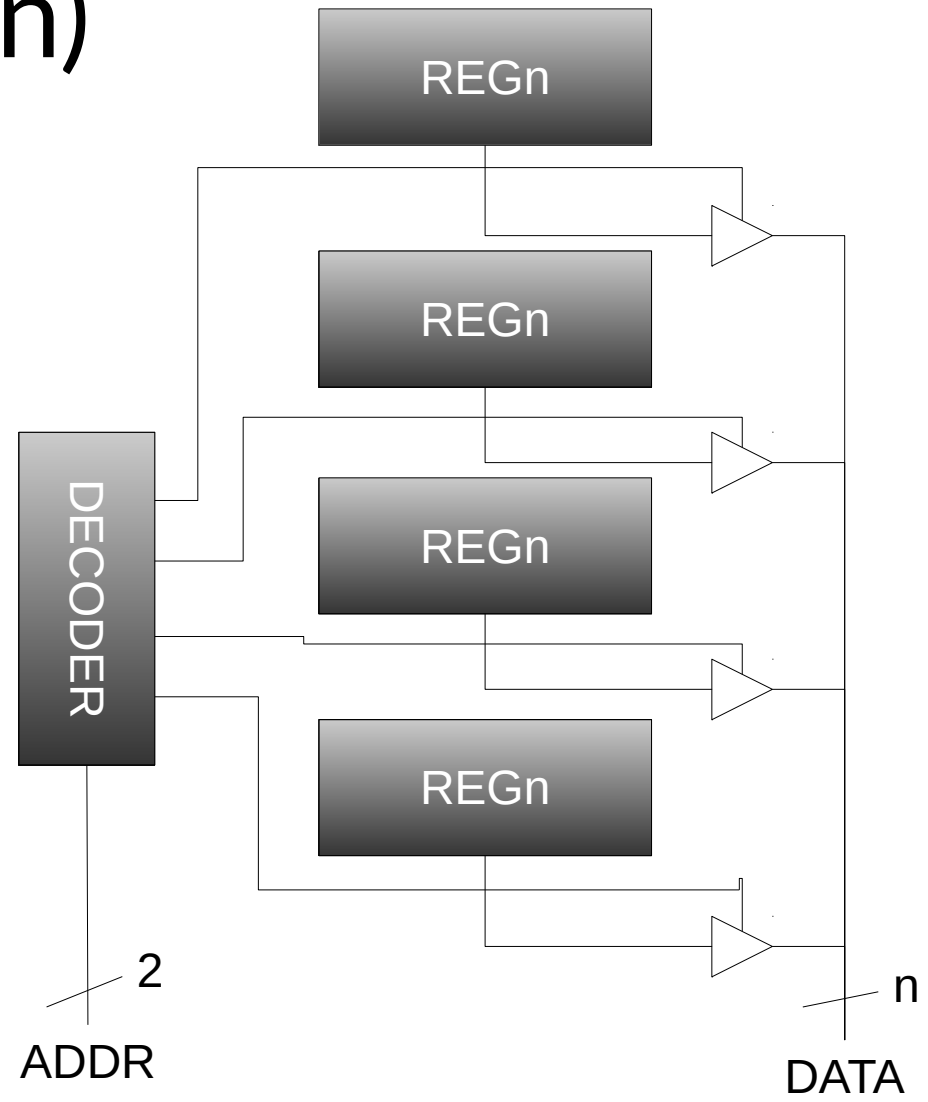


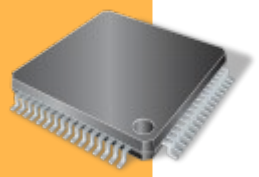
Register bank (read operation)

- combination of register outputs?
  - wired OR – does not work (electronic properties)
  - n multiplexers (for n-bits)
  - tristate buffer – OK

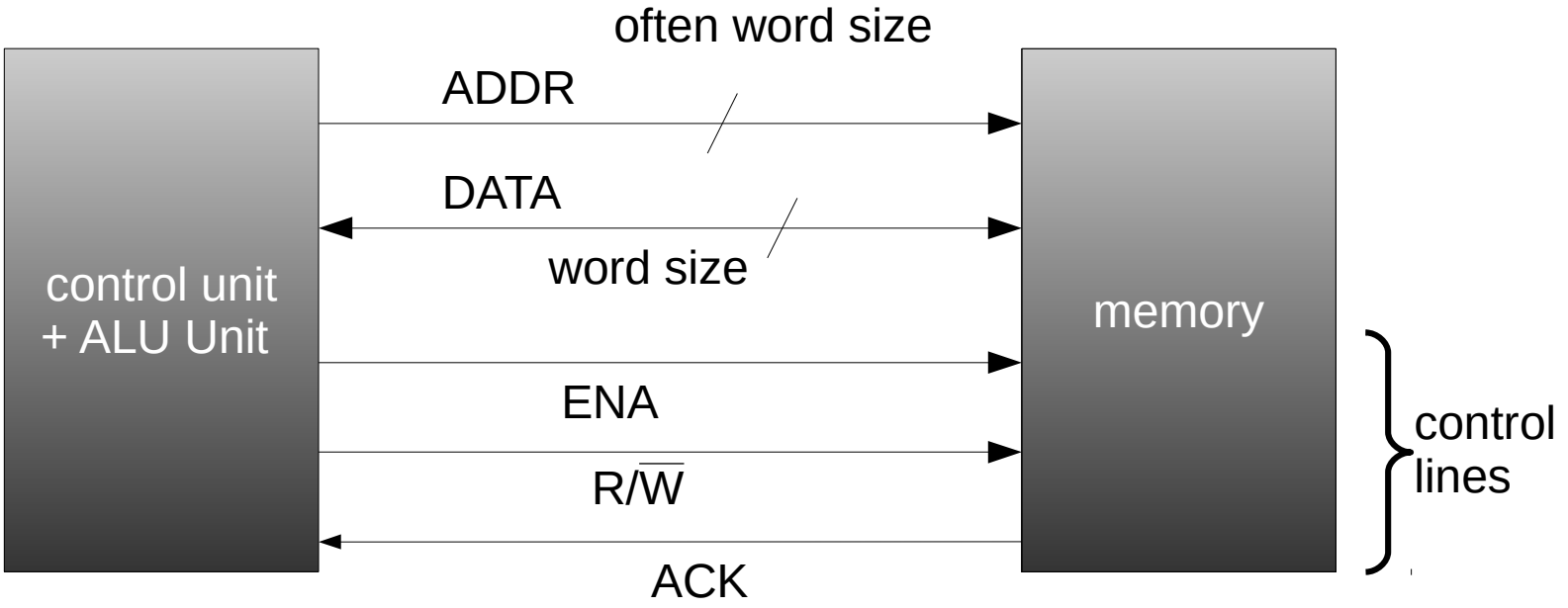


A	B	C
X	0	Z (high imp)
0	1	0
1	1	1





System bus



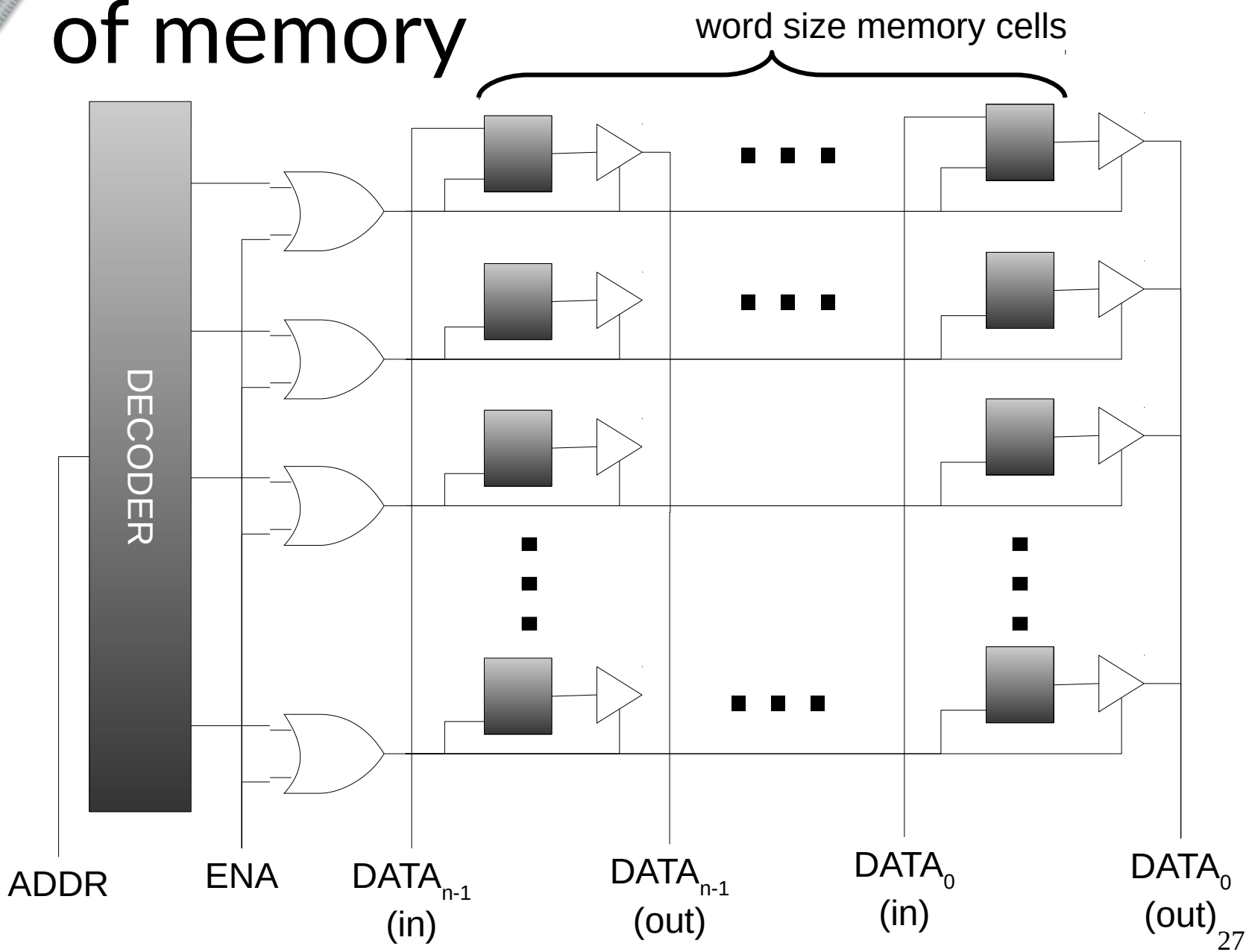
write word *w* to address *a*

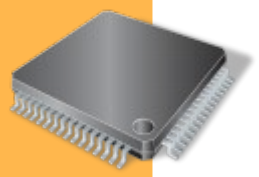
- put *w* on DATA, *a* on ADDR and R/$\overline{W}$ to 0
- then set ENA to 1
- wait for ACK set to 1 and reset ENA

read word *w* to address *a*

- put a on ADDR and R/$\overline{W}$ to 1
- then set ENA to 1
- wait for ACK set to 1
- then reset ENA and read w from DATA

Architecture of memory



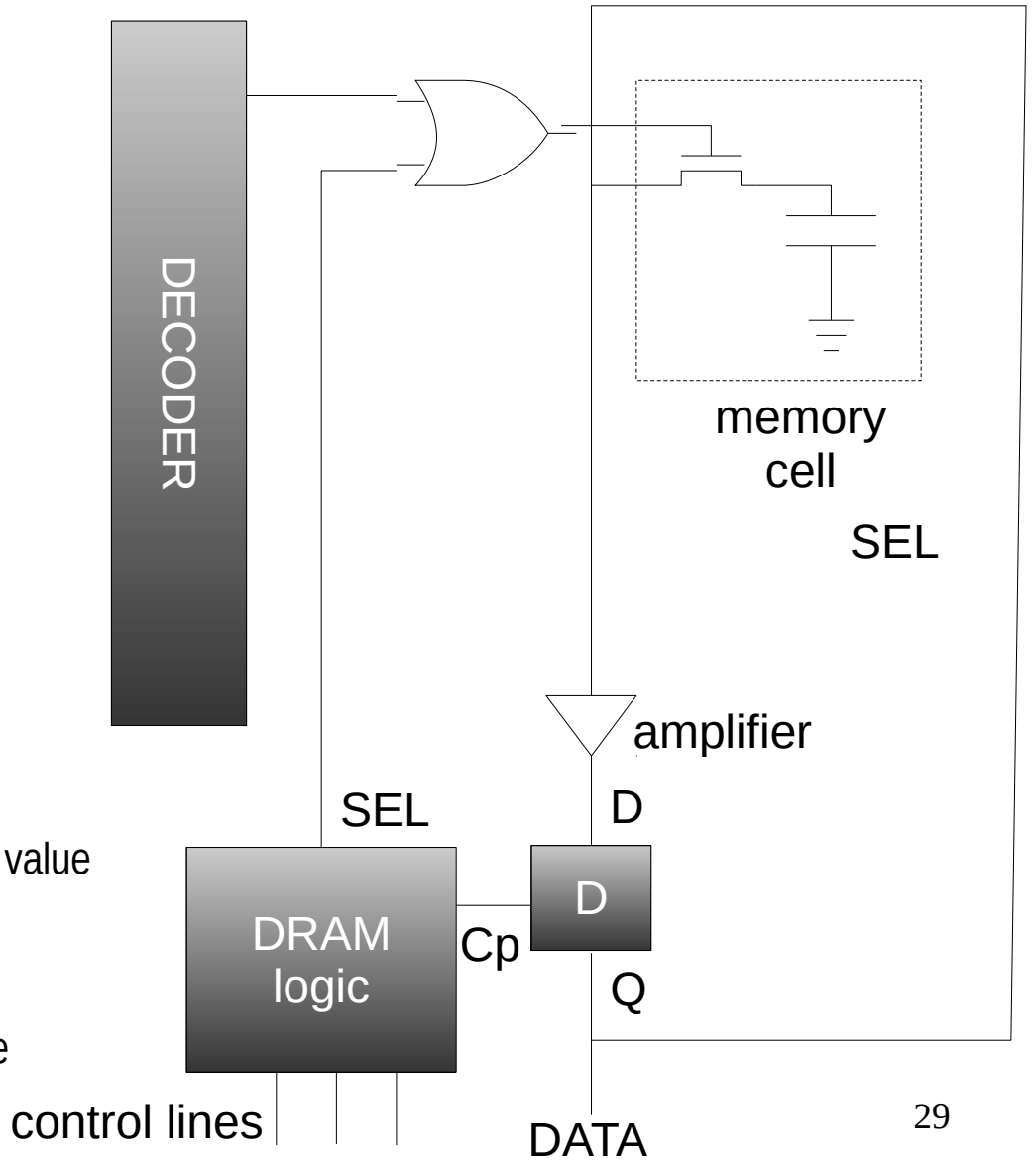


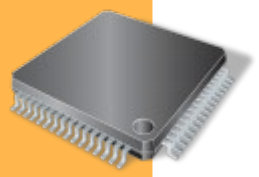
Memory types

- memory cell = D flip-flop (volatile)
  - SRAM (Static Random Access Memory)
  - very fast but requires 8 transistor / cell
- memory cell = 1 transistor + 1 capacitor (volatile)
  - DRAM (Dynamic Random Access Memory)
  - slow and cheap (often called main memory)
  - different technology as micoprocessor => different chip
- memory cell = 1 fuse (non-volatile)
  - PROM (Programmable Read-Only Memory)
  - burnt = 0, non-burnt = 1
- memory cell = flash memory cell (non-volatile)
  - EEPROM (Electrically Erasable Programmable ROM)
  - write very slow and by blocks
  - same technology as SSD or USB key

DRAM memory cell

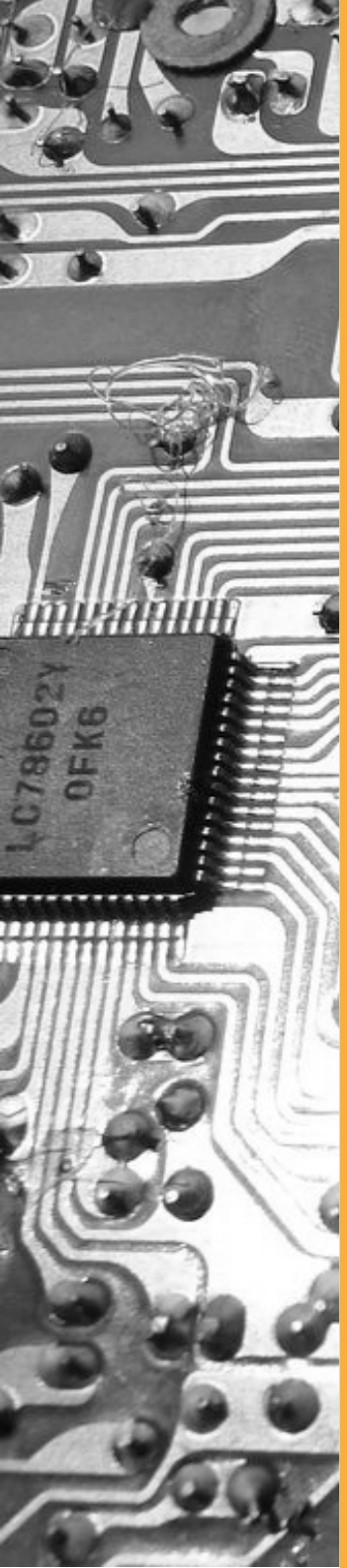
- capacitor
  - empty → 0
  - loaded → 1
- read
  - charge moved to a buffer (D flip-flop)
  - capacitor emptied
    → need for reload
- write
  - capacitor is emptied
  - new value is stored
- refresh
  - capacitor charge is leaking → loss of value
  - from time to time → word is read and rewritten
  - operations suspended during this time





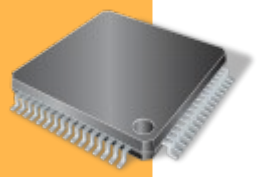
Memory performances

- t_{access} – time between start operation and end operation
- t_{cycle} – time between 2 operations
- SRAM, PROM, flash
  - $t_{access} = t_{cycle}$
- DRAM
  - $t_{cycle} = t_{access} + t_{rewrite}$
  - $t^{normal}_{access} = t_{access}$ – most of the time
  - $t^{refresh}_{access} = t_{access} + t_{refresh}$ – during a refresh cycle
  - $t_{refresh}$ – depends on the refresh policy of the memory (one line, whole memory, several lines, etc)

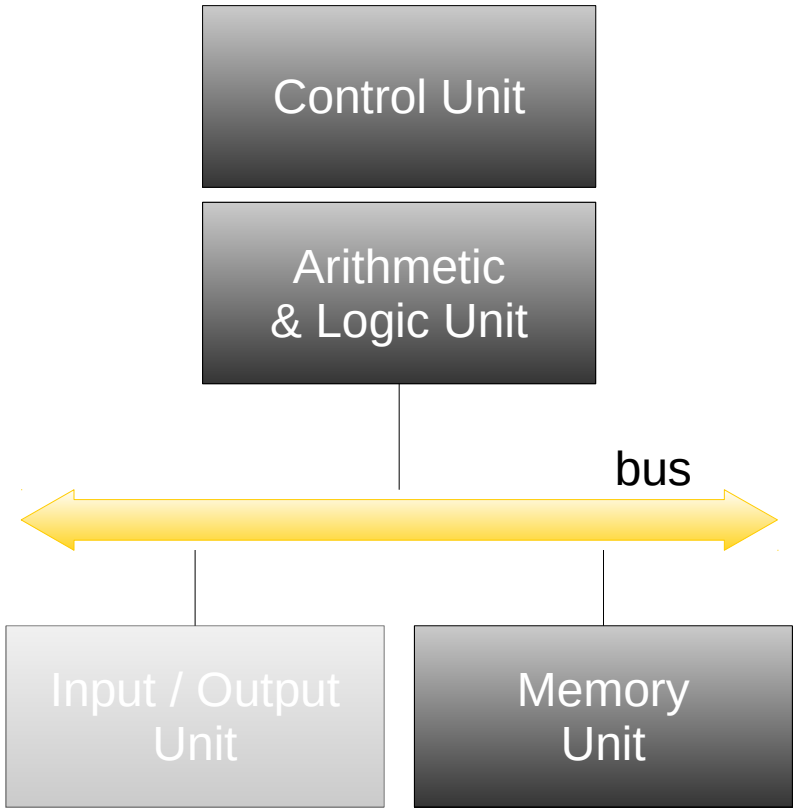


Overview

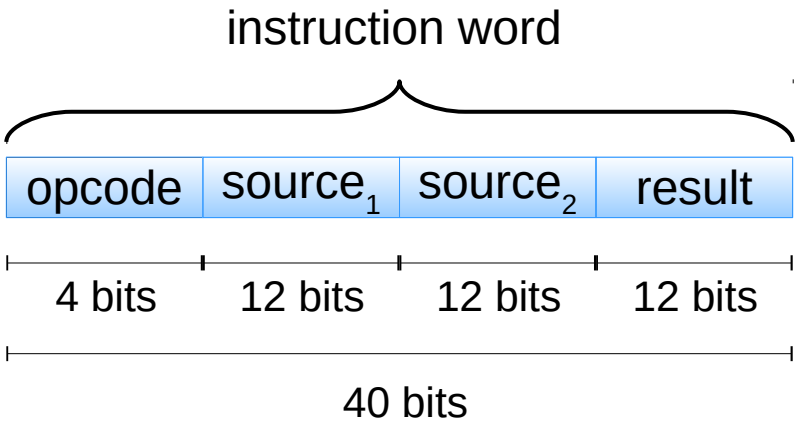
- Introduction
- Making basic circuits
- Making the memory
- **Making the processor**
- Conclusion



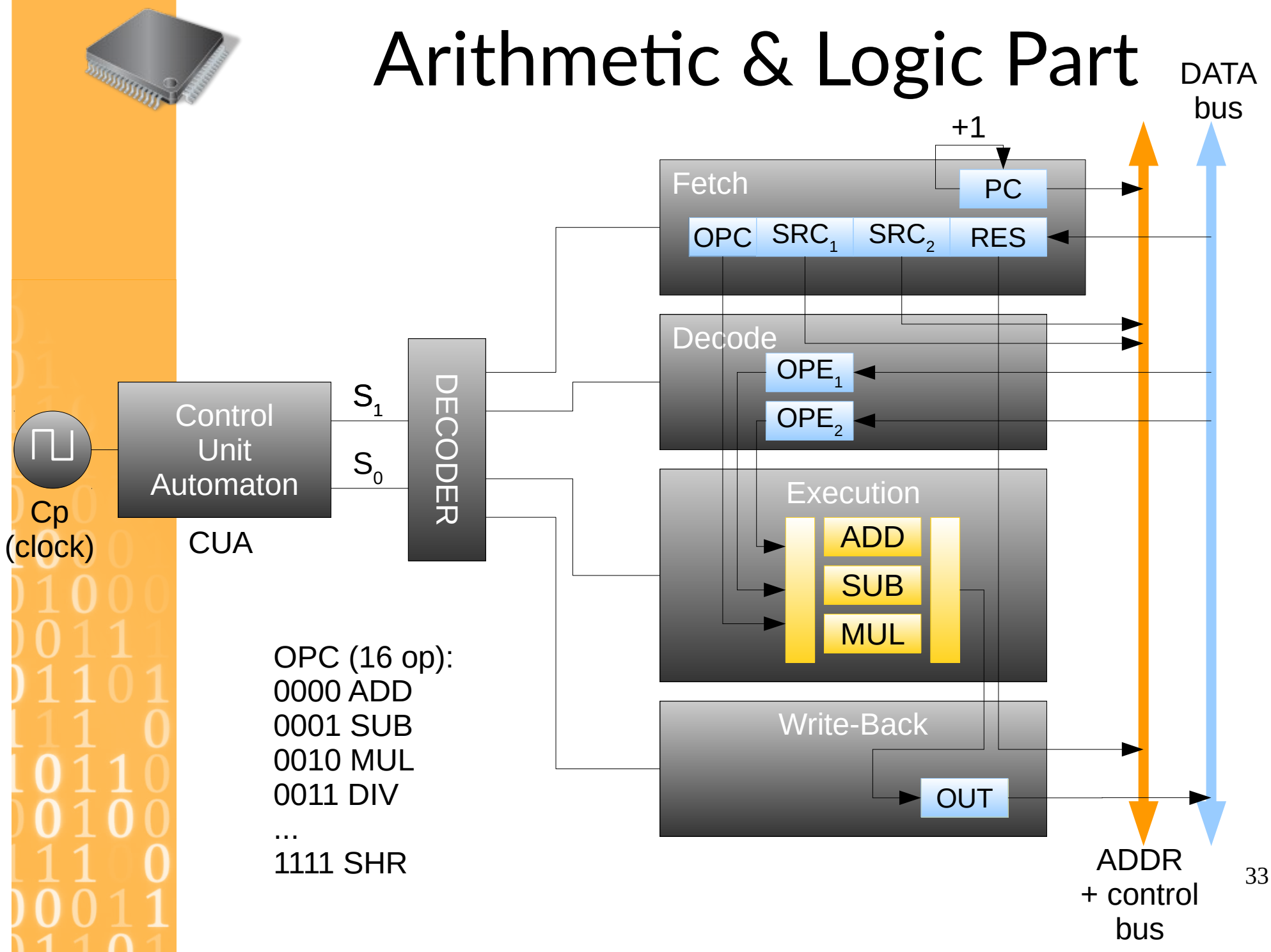
Implementing the Von Neuman machine



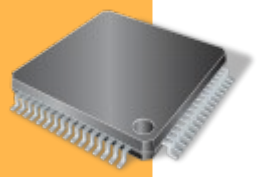
- instruction
  - operation (add, subtract, etc)
  - first operand address
  - second operand address
  - result address



Arithmetic & Logic Part



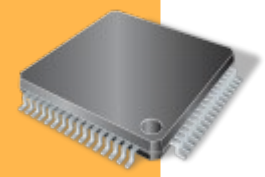
OPC (16 op):
0000 ADD
0001 SUB
0010 MUL
0011 DIV
...
1111 SHR

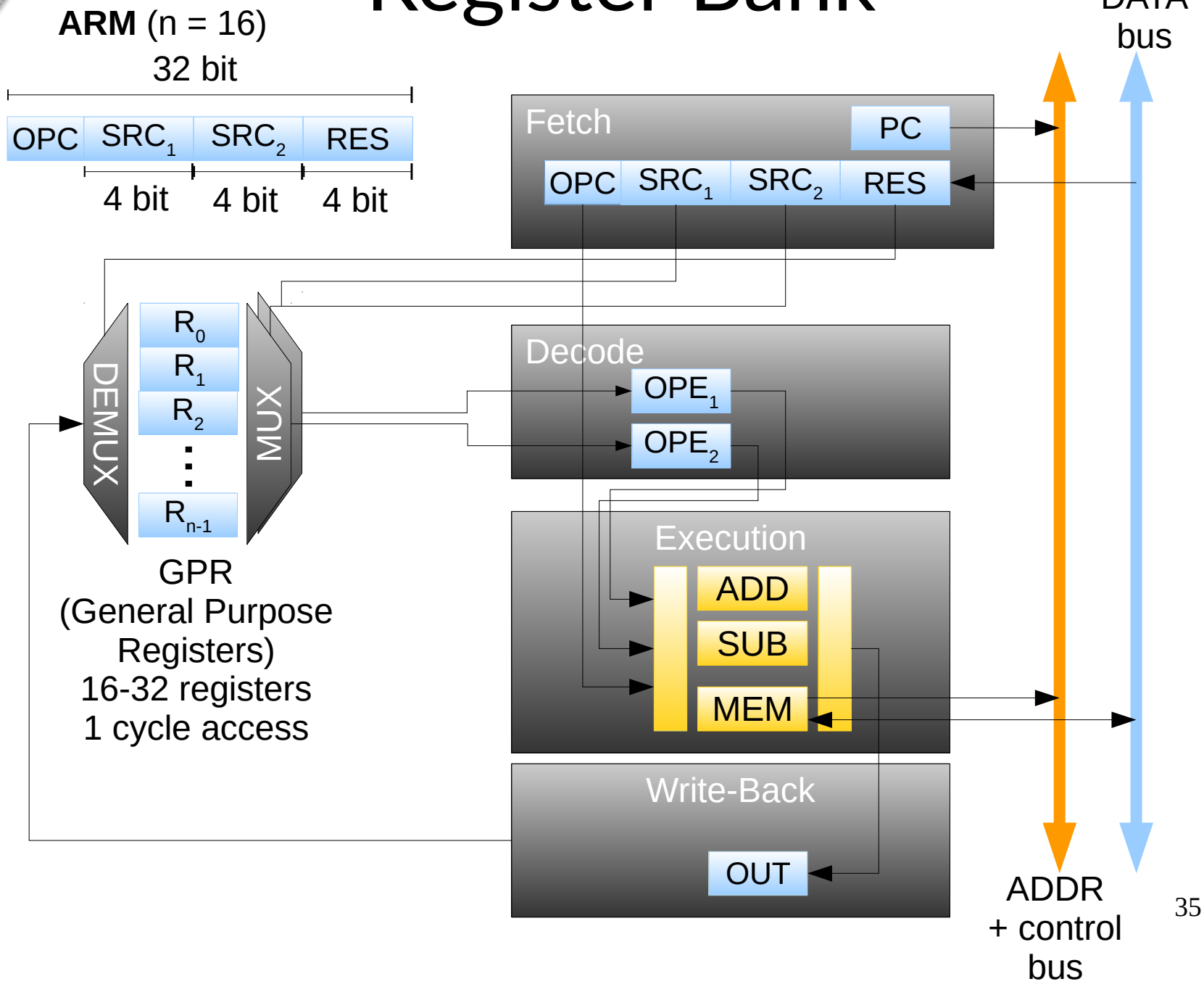


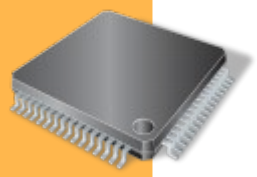
Issues with Von Neuman machine

- memory is limited by operand address encoding
  - 12 bits → 4,096 addressable word
  - bigger memory → bigger instruction word
- machine is slow
  - bottleneck = access to memory
  - 4 accesses / instruction: instruction word + operand 1 + operand 2 + result
  - several cycles required
    - setting the address (+ data)
    - reading data + waiting acknowledge

Register Bank



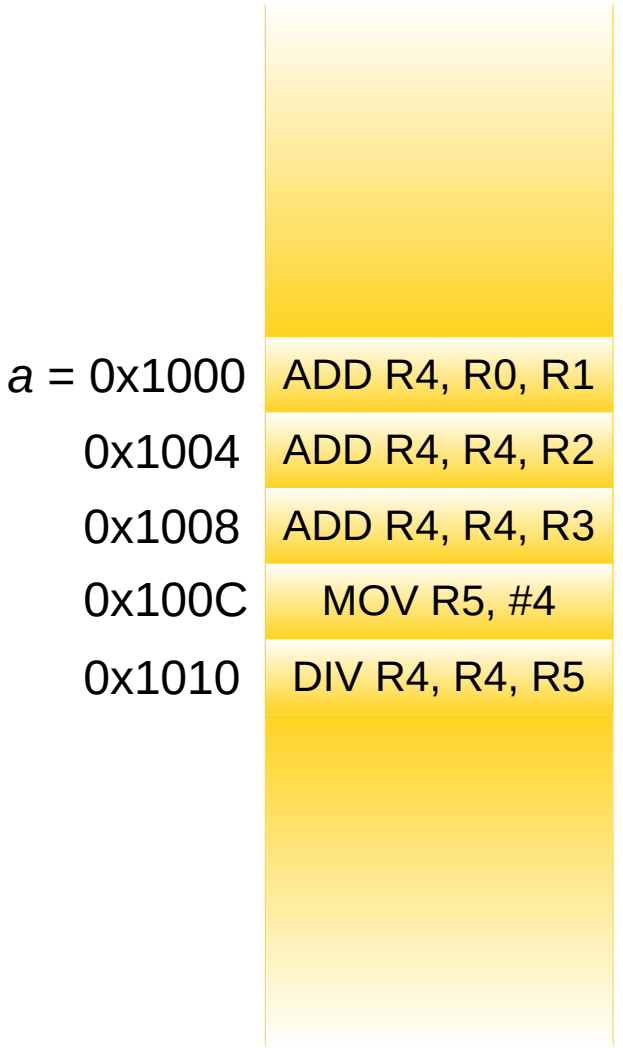




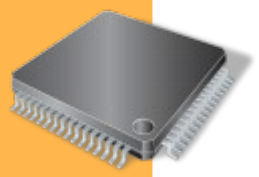
Programming this machine (ARM example)

- ARM case
  - 1 instruction = 32-bit = 4 bytes
  - PC incremented of 4
- sequential behaviour
  - read instruction at PC
  - PC ← PC + 4
  - perform it in the execution unit
- program =
  - sequence of instructions
  - at address a in memory
  - execution = PC ← a

compute (R0 + R1 + R2 + R3)/4



main memory

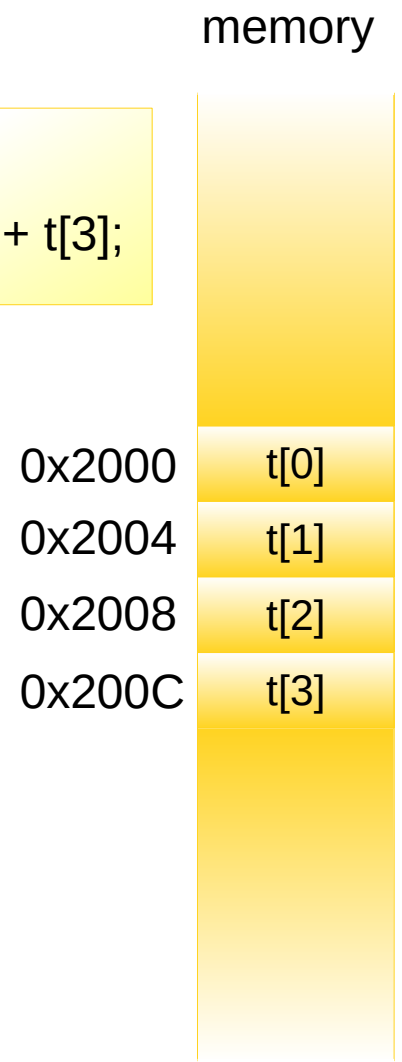


Storing variables/array in memory

- select place to store the data
- build the address in GPR
- perform the memory access
- use the read data

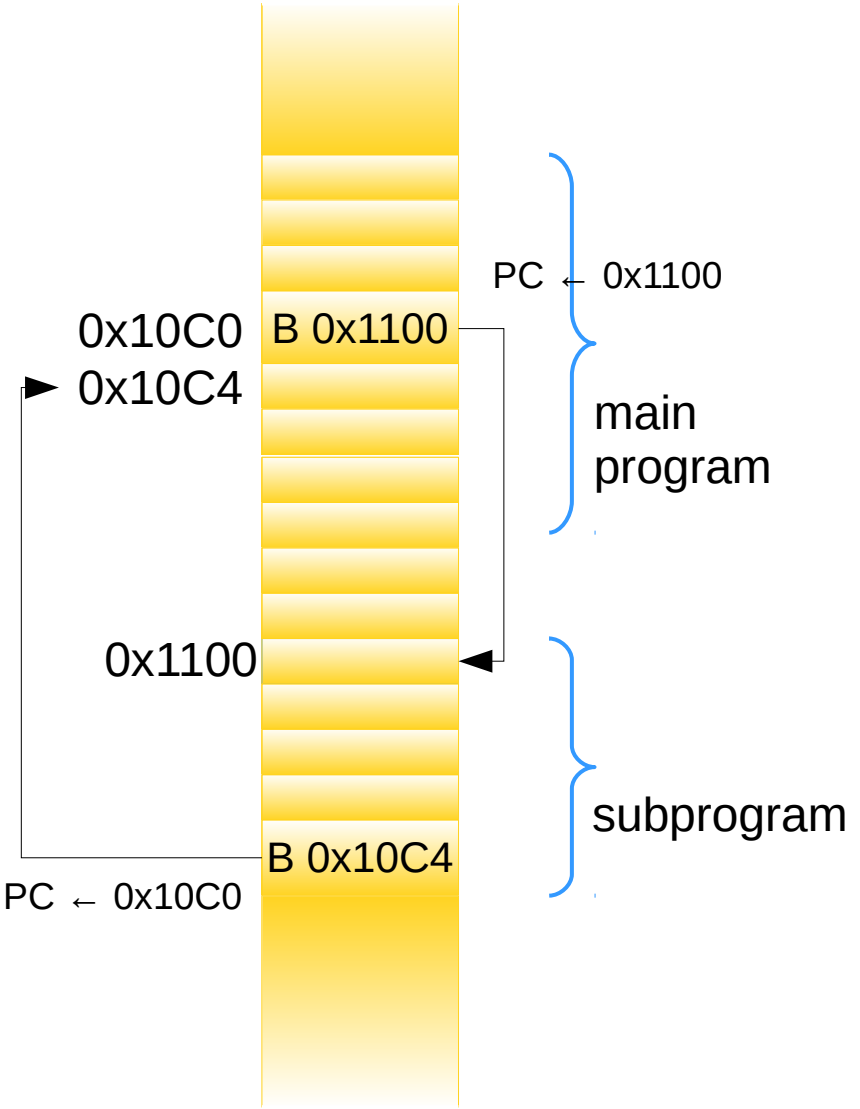
```
int t[4];
int s = t[0] + t[1] + t[2] + t[3];
```

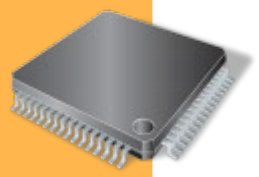
```
MOV R0, #0              @ R0←0 (= s)
MOV R1, #0x2000         @ R1 ← 0x2000 (@t[0])
LDR R2, [R1]
ADD R0, R0, R2          @ R0 ←R0 + t[0]
ADD R1, R1, #4          @ R1 ← R1 + 4 (@t[1])
LDR R2, [R1]
ADD R0, R0, R2          @ R0 ←R0 + t[0]
ADD R1, R1, #4          @ R1 ← R1 + 4 (@t[2])
...
```



Managing the control

- PC = address of instruction to execute
  - PC ← PC + 4 = sequential execution
  - PC ← a = select a different sequence to execute
- implementing a sub-progrram





Conditional execution

- while / if requires condition
- CMP R_i, R_j
  - compare Ri and Rj
  - store result in status register SR
- conditional branch
  - condition true → perform the branch
  - condition false → go on in sequence

```
int abs(int x) {
        if(x < 0)
                x = -x;
        return x;
}
```

```
0x1000  CMP R0, #0
0x1004  BGE 0x1010
0x1008  MOV R1, #0
0x100C  SUB R0, R1, R0
0x1010  ...
```

R0 < 0

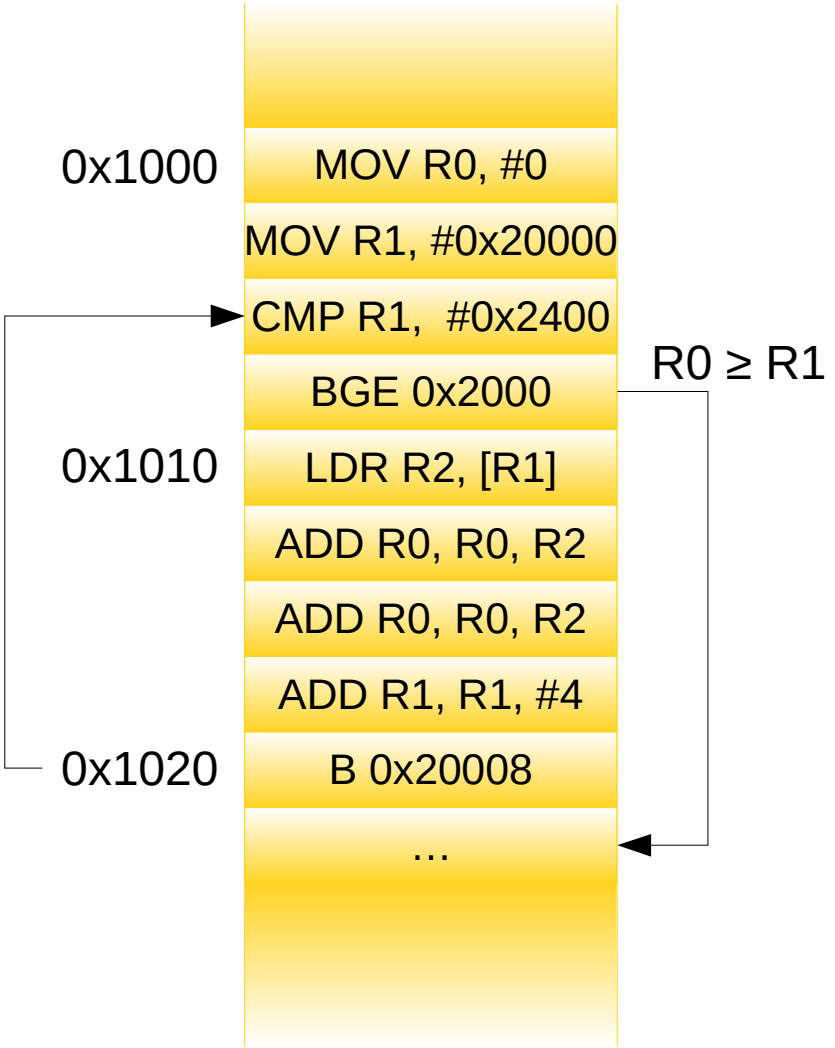
R0 ≥ 0

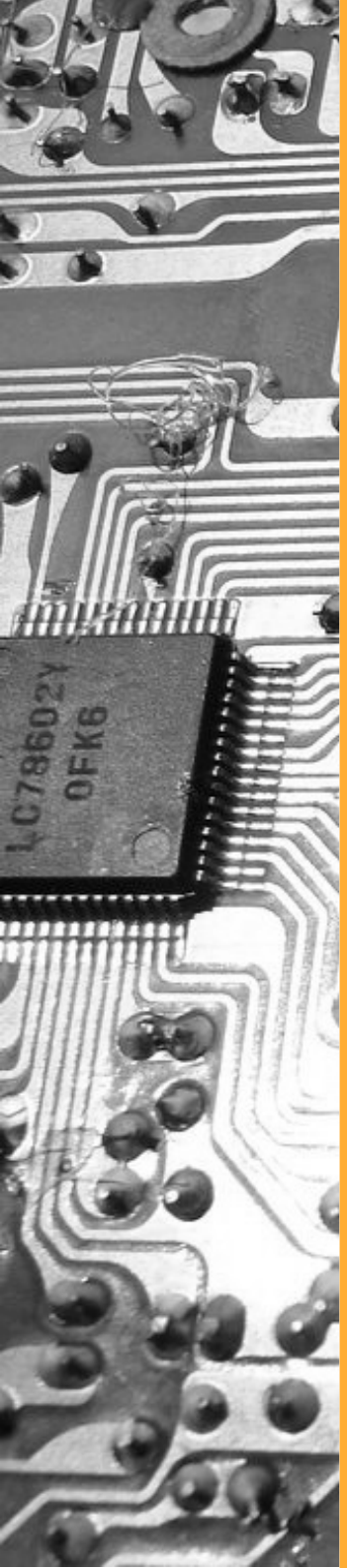
GE = Greater or Equal

Loop implementation

- looping =
  - restarting the same sequence of code = branch back
  - stopping = branching out when the condition is true

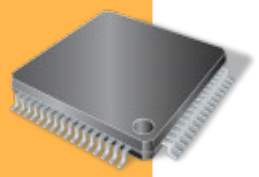
```
int t[256];
s = 0;
for(i = 0; i < 256; i++)
      s = s + t[i];
```





Overview

- Introduction
- Making basic circuits
- Making the memory
- Making the processor
- **Conclusion**



Conclusion

- how to build a microprocessor
  - transistor → Boolean algebra → logic gates → binary function
  - binary function → basic operations (addition, etc)
  - logic gates → flip-flop → register → register bank → memories
  - control unit, Arithmetic & Logic Unit → microprocessor
- now
  - how to drive input / output (outside world)
  - how to improve the performances of the microprocessor
    - computation acceleration
    - memory footprint reduction