

MANAGEMENT AND ANALYSIS OF PHYSICS DATASET (MOD. A)

Fundamentals of Boolean Algebra
Combinatorial Functions

Postulates of the Boolean algebra

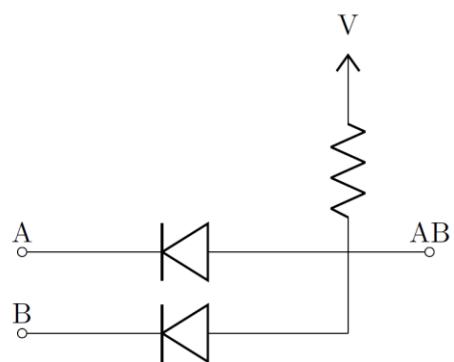
- Digital circuits or logic circuits deal with quantized signals (lack or presence of a signal)
- Fundamental elements of the digital computing
- “Logic” because we can apply analysis and synthesis methods derived from Boolean algebra
- Class of elements M and two operators \cdot and $+$

P1:	$A + B = B + A$	$AB = BA$	Commutative
P2:	$A + 0 = A$	$A \cdot 1 = A$	Identity
P3:	$A + \bar{A} = 1$	$A\bar{A} = 0$	Complement
P4:	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$	Distributive

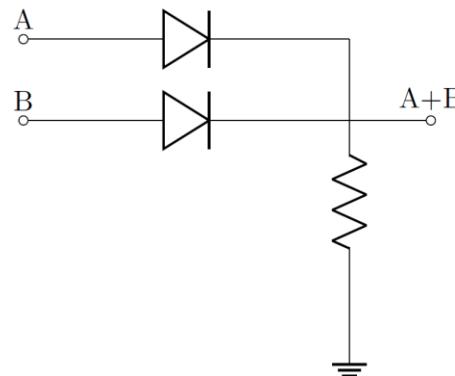
- Duality of the operators

Postulates of the Boolean algebra

- Proof of non contradiction: M are connections, · is the circuital operator AND and + is the OR



$$\begin{array}{c} \text{V} \\ \uparrow = 1 \\ \text{---} \\ \text{AB} = 0 \end{array}$$

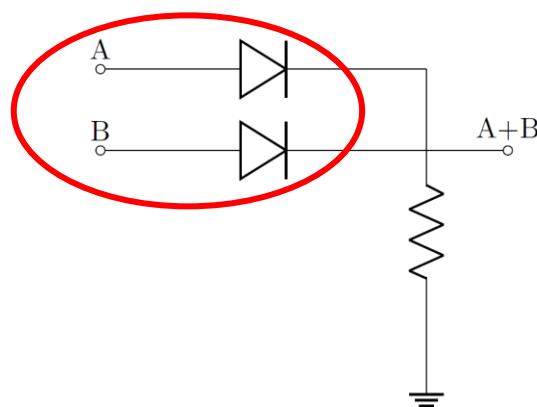
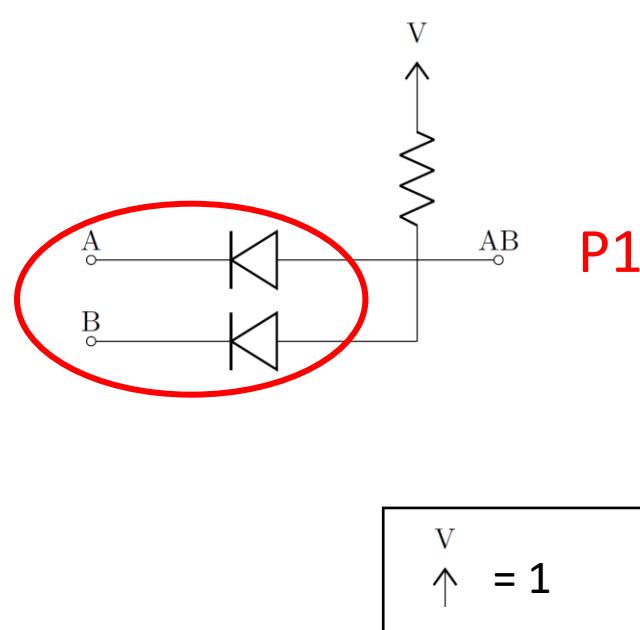


A	B	AB
0	0	0
1	0	0
0	1	0
1	1	1

A	B	A+B
0	0	0
1	0	1
0	1	1
1	1	1

Postulates of the Boolean algebra

- Proof of non contradiction: M are connections, · is the circuital operator AND and + is the OR

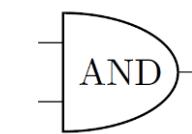
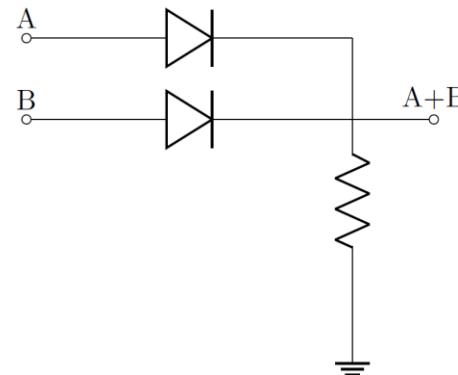
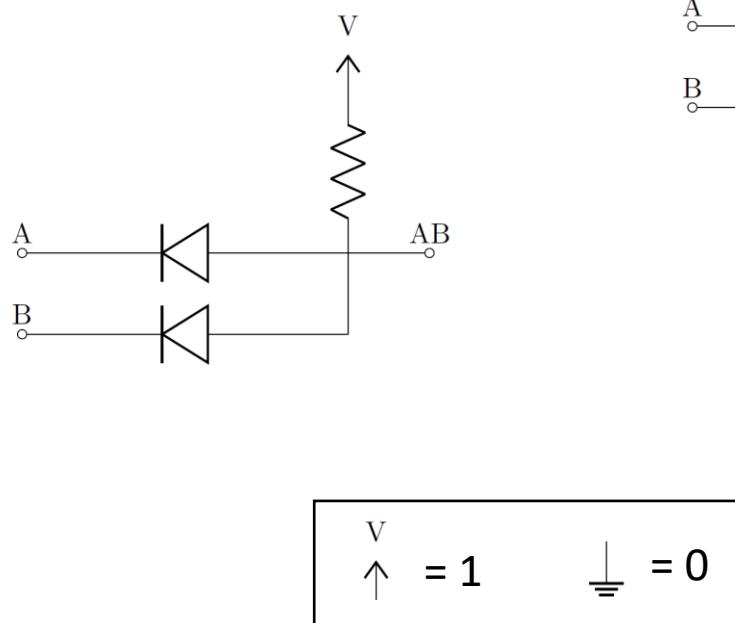


A	B	AB
0	0	0
1	0	0
0	1	0
1	1	1

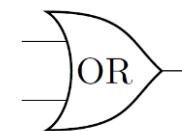
A	B	$A+B$
0	0	0
1	0	1
0	1	1
1	1	1

Postulates of the Boolean algebra

- Proof of non contradiction: M are connections, · is the circuital operator AND and + is the OR



A	B	AB
0	0	0
1	0	0
0	1	0
1	1	1

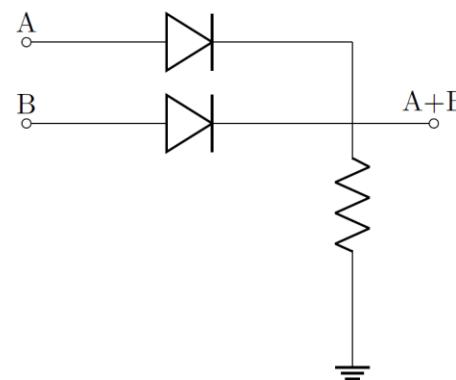
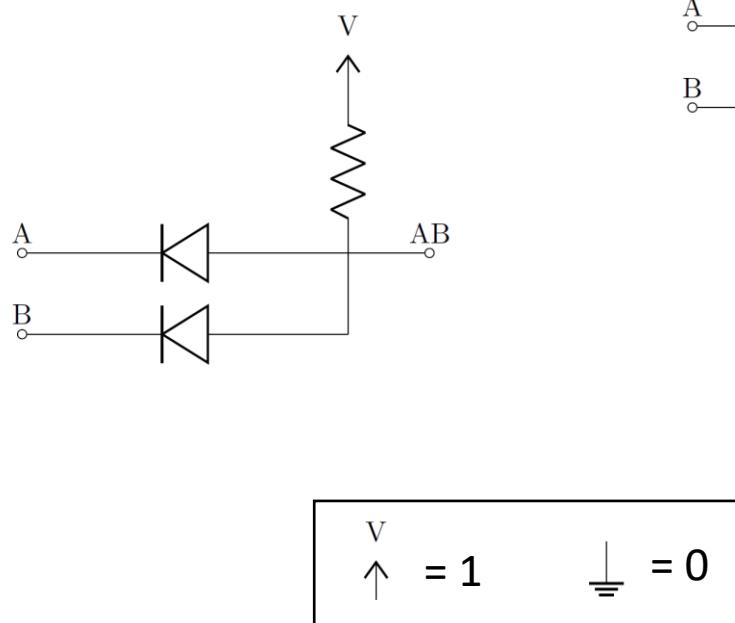


A	B	$A+B$
0	0	0
1	0	1
0	1	1
1	1	1

P2

Postulates of the Boolean algebra

- Proof of non contradiction: M are connections, · is the circuital operator AND and + is the OR



AND			OR		
A	B	AB	A	B	A+B
0	0	0	0	0	0
1	0	0	1	0	1
0	1	0	0	1	1
1	1	1	1	1	1

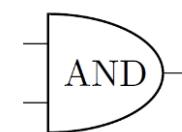
P3

Postulates of the Boolean algebra

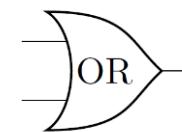
- Proof of non contradiction: M are connections, · is the circuital operator AND and + is the OR

A	B	C	$A+BC$	$(A+B)(A+C)$
0	0	0	0	0
1	0	0	1	1
0	1	0	0	0
1	1	0	1	1
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	1	1	1

P4



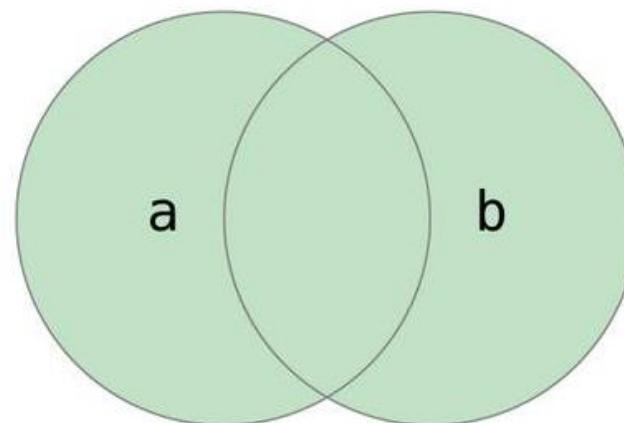
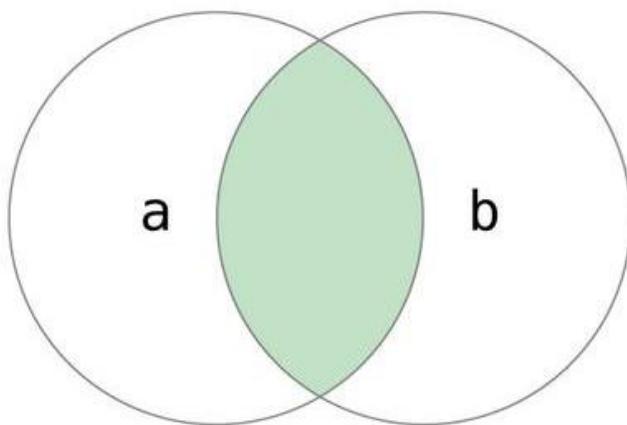
A	B	AB
0	0	0
1	0	0
0	1	0
1	1	1



A	B	$A+B$
0	0	0
1	0	1
0	1	1
1	1	1

Postulates of the Boolean algebra

- The postulates are valid also for other classes
- For instance: M is the class of all subsets of the set M , \cdot is the intersection and $+$ is the union
- Graphical representation can be used to quickly proof theorems



Simple theorems

T1: $A + A = A$ $AA = A$

Proof: $A + A = (A + A) \cdot 1 = (A + A)(A + \bar{A}) = A + A\bar{A} = A + 0 = A$

T2: $A + 1 = 1$ $A \cdot 0 = 0$

Proof: $A + 1 = (A + 1) \cdot 1 = (A + 1)(A + \bar{A}) = A + \bar{A} \cdot 1 = A + \bar{A} = 1$

T3: $A + AB = A$ $A(A + B) = A$

Proof: $A + AB = (A \cdot 1) + AB = A(1 + B) = A \cdot 1 = A$

De Morgan laws

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

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

Proof:

$$(A + B) + \overline{A} \cdot \overline{B} = (A + B + \overline{A})(A + B + \overline{B}) = (B + 1)(A + 1) = 1 \cdot 1 = 1$$

$$(A + B) \cdot \overline{A} \cdot \overline{B} = A \cdot \overline{A} \cdot \overline{B} + B \cdot \overline{A} \cdot \overline{B} = \overline{B} \cdot 0 + \overline{A} \cdot 0 = 0 + 0 = 0$$

From P3 we can conclude that $A+B$ is the complementary of $\overline{A} \cdot \overline{B}$

Generalization: the complementary of a Boolean function results from the exchange of the dual operators and from the exchange of the operands with their complement

Exercises

1. Demonstrate that $(A + B)(\overline{A} + C) = \overline{A}B + AC$

2. Demonstrate that $f(A, B) = \overline{A}\overline{B} + AB = \overline{f}(\overline{A}, B)$

Combinatorial functions

Combinatorial functions

- A simple way of describing a Boolean function is the truth table

A	B	$f(A, B) = \overline{A} + \overline{B}$
0	0	1
1	0	1
0	1	1
1	1	0

- It describes f as function of its input -> combinatory function
- Many functions can be built from the same truth table

$$\begin{aligned}f(A, B) &= \overline{A \cdot B} \\&= \overline{A} \overline{B} + \overline{A} B + A \overline{B} \\&= \overline{B} + \overline{A} B\end{aligned}$$

Which is the best?

Canonical form

- Minterm (m) of n variables is the Boolean product of all the variables where they appears only once (either in the complemented or uncomplemented form)
- They are 2^n . For instance, for $n = 2$ they are $\overline{A} \overline{B}$, $A \overline{B}$, $\overline{A} B$ and $A B$
- Similarly, Maxterm (M) can be defined for the sum

A	B	Minterm	Maxterm
0	0	$m_0 = \overline{A} \overline{B}$	$M_0 = \overline{A} + \overline{B}$
1	0	$m_1 = A \overline{B}$	$M_1 = A + \overline{B}$
0	1	$m_2 = \overline{A} B$	$M_2 = \overline{A} + B$
1	1	$m_3 = A B$	$M_3 = A + B$

Canonical form

- m_i is always equal to 0 except in one case (variables in line i)
- M_i is always equal to 1 except in one case (variables in line $2^n - 1 - i$)
- The product of two distinct minterms is always 0 and the sum of two distinct maxterm is always 1

$$\overline{m_i} = M_{2^n-1-i}$$

$$\overline{M_i} = m_{2^n-1-i}$$

$$\sum_{i=0}^{2^n-1} m_i = 1$$

$$\prod_{i=0}^{2^n-1} M_i = 0$$

Canonical form

- Combinatorial functions expressed as a sum of minterms or product of maxterms are said to be in canonical form

A	B	$f(A, B) = \bar{A} + \bar{B}$
0	0	1
1	0	1
0	1	1
1	1	0

A	B	F_k
0	0	f_0
1	0	f_1
0	1	f_2
1	1	f_3

$$f(A, B) = m_0 + m_1 + m_2$$
$$f(A, B) = M_0$$

$$F_k = \sum_{i=0}^{2^n-1} f_i m_i$$
$$F_k = \prod_{i=0}^{2^n-1} (f_i + M_{2^n-1-i})$$

Canonical form

- All the possible Boolean function of n variables are 2^{2^n}

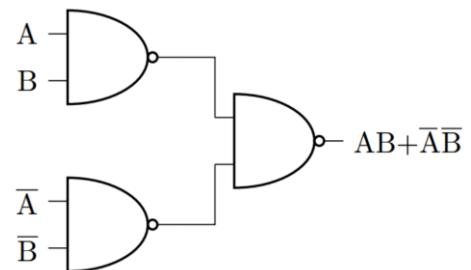
f_0	f_1	f_2	f_3	F_k
0	0	0	0	0
1	0	0	0	NOR
0	1	0	0	$A\bar{B}$
1	1	0	0	\bar{B}
0	0	1	0	$\bar{A}B$
1	0	1	0	\bar{A}
0	1	1	0	XOR
1	1	1	0	NAND
0	0	0	1	AND
1	0	0	1	\overline{XOR}
0	1	0	1	A
1	1	0	1	$A + \bar{B}$
0	0	1	1	B
1	0	1	1	$\bar{A} + B$
0	1	1	1	OR
1	1	1	1	1

From function to circuit

- Canonical form is the most generic, but it is not the simplest
- For building a circuit it is better to simplify
- Passive circuits cannot be used because they degrades the signal after few logic levels -> active components at least in the output stage
- The simplest amplifier is the common emitter transistor -> signal inversion -> NAND or NOR
- Any combinatorial Boolean function can be written with them
 - The complement of a variable is done wiring together the two inputs or applying a 1 for NAND and 0 for NOR
 - By De Morgan NAND is the OR of complements and NOR is the AND of complements

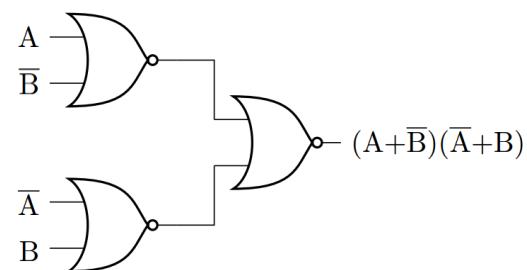
From function to circuit

$$f = A B + \overline{A} \overline{B}$$

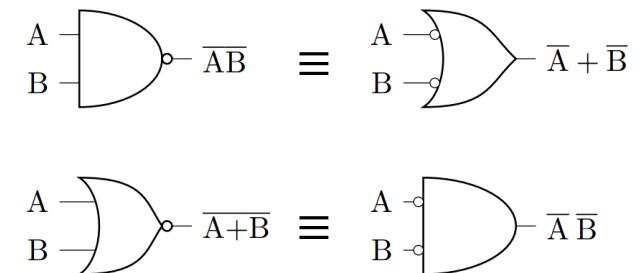


Sum of products

$$f = (A + \overline{B})(\overline{A} + B)$$



Product of sums



De Morgan duality

XOR

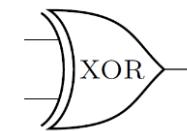
$$\bullet A \oplus B = \overline{A} \oplus \overline{B} = A \overline{B} + \overline{A} B$$

$$\bullet \overline{A \oplus B} = \overline{A} \oplus B = \overline{A} \overline{B} + A B$$

$$\bullet \overline{A} = 1 \oplus A$$

$$\bullet A + B = \overline{\overline{A} \overline{B}} = 1 \oplus [(1 \oplus A)(1 \oplus B)]$$

$$\bullet A B = \overline{\overline{A} + \overline{B}} = 1 \oplus [(1 \oplus A) + (1 \oplus B)]$$



A	B	$A \oplus B$
0	0	0
1	0	1
0	1	1
1	1	0

Graphical simplification

- A function is represented as union of intersections of subsets
- Veitch or Karnaugh maps

A	B
m_3	m_2
m_1	m_0

-----A-----			
m_3	m_7	m_6	m_2
m_1	m_5	m_4	m_0

-----C-----

-----A-----			
m_3	m_7	m_6	m_2
m_{11}	m_{15}	m_{14}	m_{10}
m_9	m_{13}	m_{12}	m_8
m_1	m_5	m_4	m_0

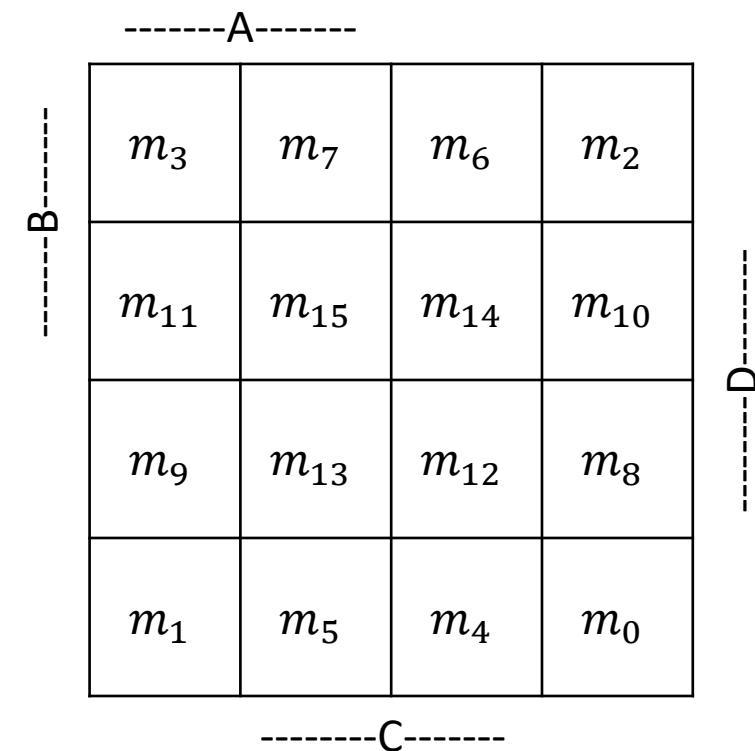
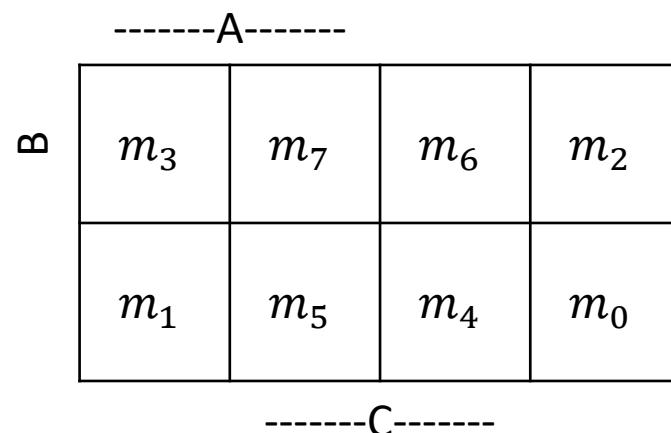
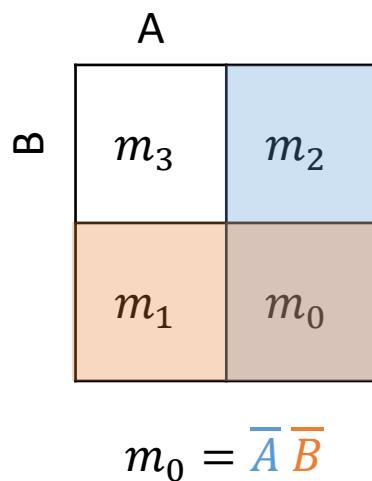
-----B-----

-----D-----

-----C-----

Graphical simplification

- A function is represented as union of intersections of subsets
- Veitch or Karnaugh maps



Graphical simplification

- A function is represented as union of intersections of subsets
- Veitch or Karnaugh maps

A	
B	m_3 m_2
	m_1 m_0

$$m_1 = A \bar{B}$$

	-----A-----
B	m_3 m_7 m_6 m_2
	m_1 m_5 m_4 m_0

-----C-----

	-----A-----
B	m_3 m_7 m_6 m_2
	m_{11} m_{15} m_{14} m_{10}
D	m_9 m_{13} m_{12} m_8
	m_1 m_5 m_4 m_0

-----C-----

Graphical simplification

- A function is represented as union of intersections of subsets
- Veitch or Karnaugh maps

A	
B	m_3 m_2
	m_1 m_0

$$m_2 = \overline{A} B$$

	-----A-----
B	m_3 m_7 m_6 m_2
	m_1 m_5 m_4 m_0

-----C-----

	-----A-----
B	m_3 m_7 m_6 m_2
	m_{11} m_{15} m_{14} m_{10}
D	m_9 m_{13} m_{12} m_8
	m_1 m_5 m_4 m_0

-----C-----

Graphical simplification

- A function is represented as union of intersections of subsets
- Veitch or Karnaugh maps

	A
B	m_3 m_2
	m_1 m_0

$$m_3 = \textcolor{blue}{A} \textcolor{orange}{B}$$

	-----A-----
B	m_3 m_7 m_6 m_2
	m_1 m_5 m_4 m_0

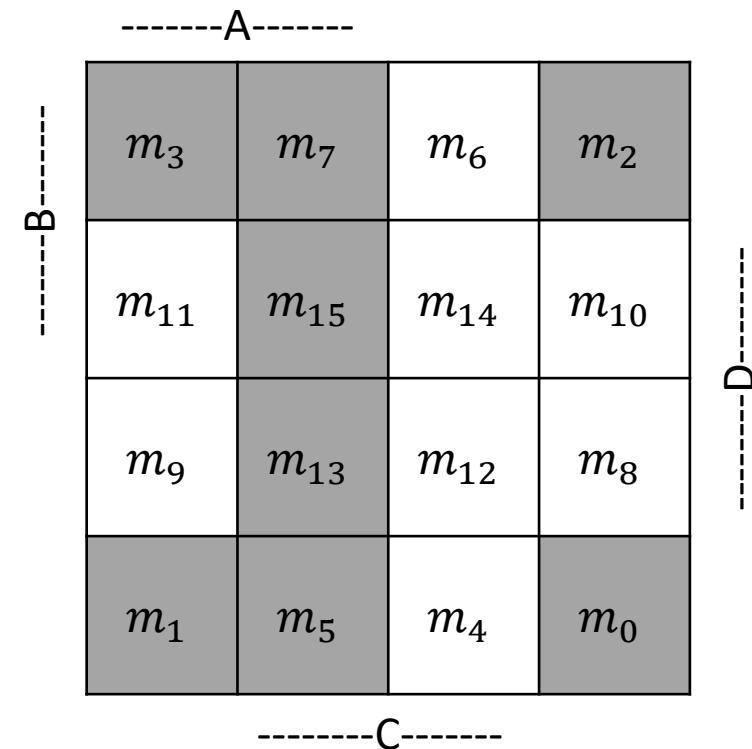
-----C-----

	-----A-----
B	m_3 m_7 m_6 m_2
	m_{11} m_{15} m_{14} m_{10}
D	m_9 m_{13} m_{12} m_8
	m_1 m_5 m_4 m_0

-----C-----

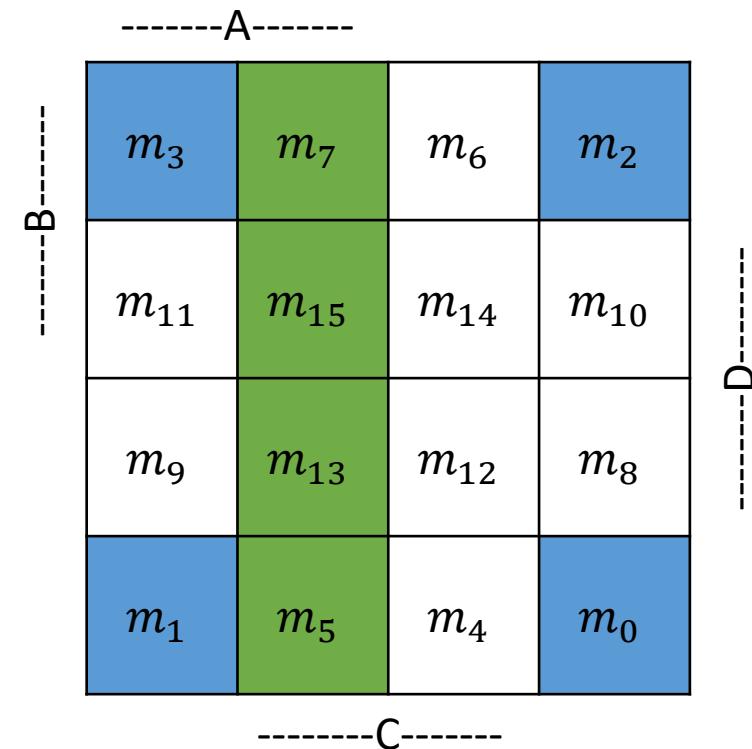
Graphical simplification

$$\begin{aligned} \bullet f = & \overline{A} \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + \overline{A} B \overline{C} \overline{D} + A B \overline{C} \overline{D} + A \overline{B} C \overline{D} + \\ & A B C \overline{D} + A \overline{B} C D + A B C D \end{aligned}$$



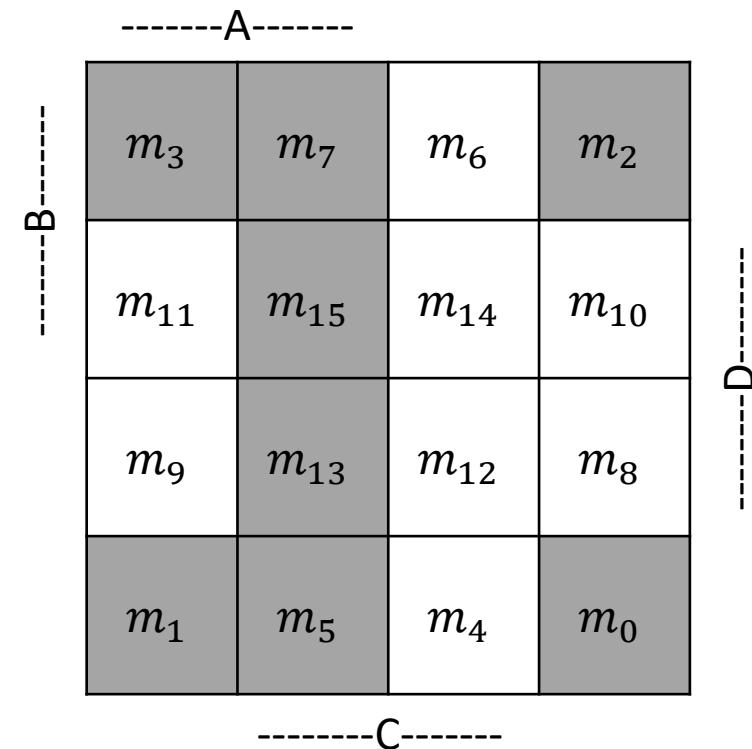
Graphical simplification

- $f = \overline{A} \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + \overline{A} B \overline{C} \overline{D} + A B \overline{C} \overline{D} + A \overline{B} C \overline{D} + A B C \overline{D} + A \overline{B} C D + A B C D$
- $f = A C + \overline{C} \overline{D}$ -> NAND



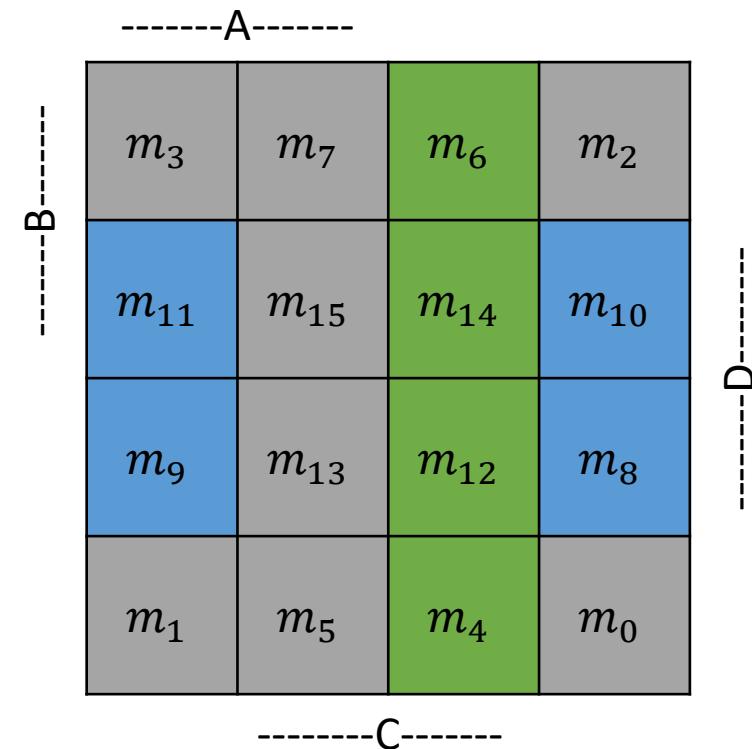
Graphical simplification

- $f = \overline{A} \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + \overline{A} B \overline{C} \overline{D} + A B \overline{C} \overline{D} + A \overline{B} C \overline{D} + A B C \overline{D} + A \overline{B} C D + A \overline{B} C D + A B C D$
- $f = A C + \overline{C} \overline{D}$ -> NAND



Graphical simplification

- $f = \overline{A} \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + \overline{A} B \overline{C} \overline{D} + A B \overline{C} \overline{D} + A \overline{B} C \overline{D} + A B C \overline{D} + A \overline{B} C D + A \overline{B} C D + A B C D$
- $f = A C + \overline{C} \overline{D}$ -> NAND
- $\overline{f} = \overline{\overline{A} C} + \overline{\overline{C} D}$
- $f = (A + \overline{C})(C + \overline{D})$ -> NOR



Exercises

1. Demonstrate that $(A \oplus B) \oplus C = A \oplus (B \oplus C) = B \oplus (A \oplus C)$
2. Simplify $AB\overline{C} + ABC + \overline{A}BC + \overline{A}B\overline{C} + A\overline{B}C + \overline{A}\overline{B}C$

Examples of combinatorial logic

Adder

- Elementary cell for adding two bits (binary digit)
- Extension to two n-bits numbers

A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

$$S_i = (A \bar{B} \bar{C} + \bar{A} B \bar{C} + \bar{A} \bar{B} C + A B C)_i$$

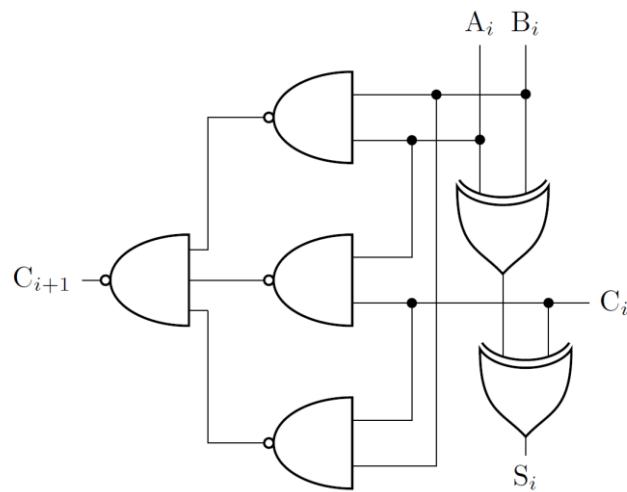
$$C_{i+1} = (A B \bar{C} + A \bar{B} C + \bar{A} B C + A B C)_i$$

$$S_i = (A \bar{B} + \bar{A} B) \bar{C}_i + (\bar{A} \bar{B} + A B) C_i = (A \oplus B)_i \oplus C_i$$

$$C_{i+1} = (A B + A C + B C)_i$$

Adder

- Elementary cell for adding two bits (binary digit)
- Extension to two n-bits numbers



$$S_i = (A \bar{B} \bar{C} + \bar{A} B \bar{C} + \bar{A} \bar{B} C + A B C)_i$$

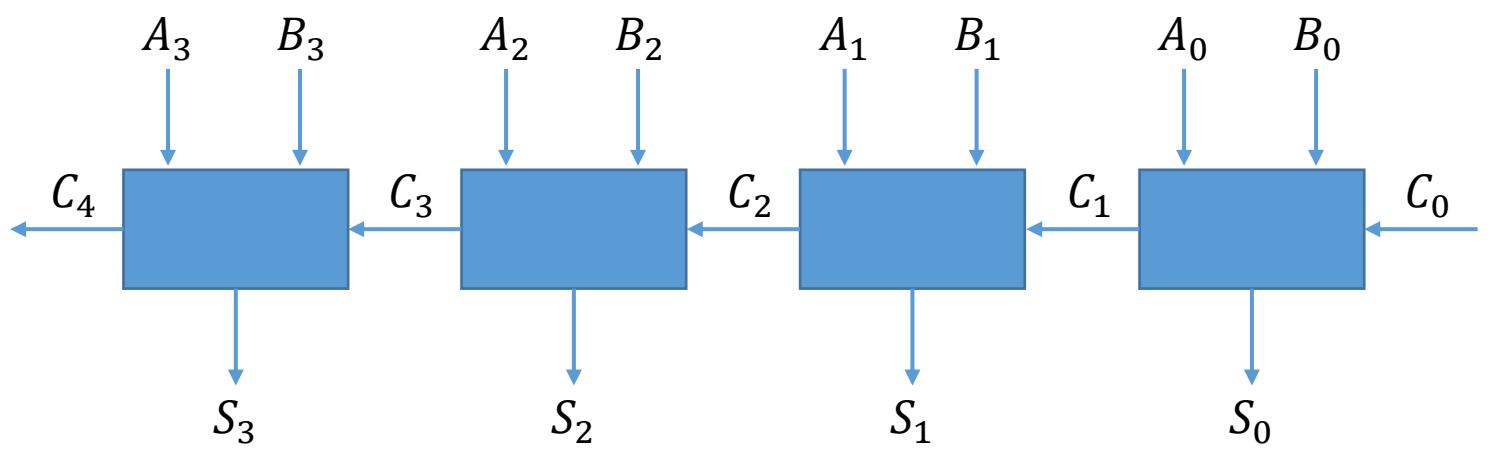
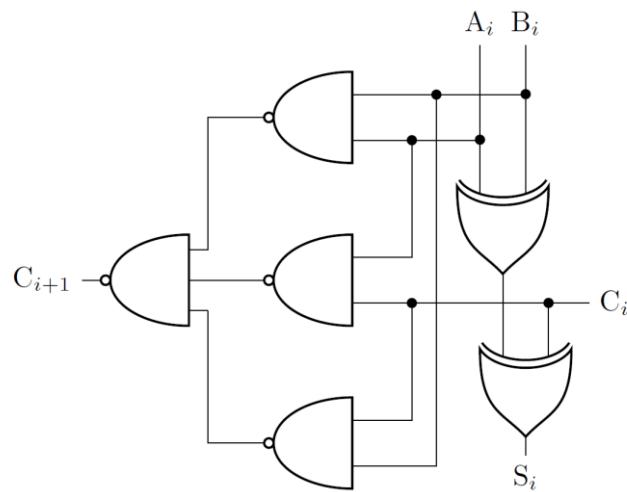
$$C_{i+1} = (A B \bar{C} + A \bar{B} C + \bar{A} B C + A B C)_i$$

$$S_i = (A \bar{B} + \bar{A} B) \bar{C}_i + (\bar{A} \bar{B} + A B) C_i = (A \oplus B)_i \oplus C_i$$

$$C_{i+1} = (A B + A C + B C)_i$$

Adder

- Elementary cell for adding two bits (binary digit)
- Extension to two n-bits numbers



Subtractor

- Exercise
 - Draw the circuit
 - Can be easily obtained from an adder?

Subtractor

A_i	B_i	R_i	D_i	R_{i+1}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	1
1	1	0	0	0
0	0	1	1	1
1	0	1	0	0
0	1	1	0	1
1	1	1	1	1

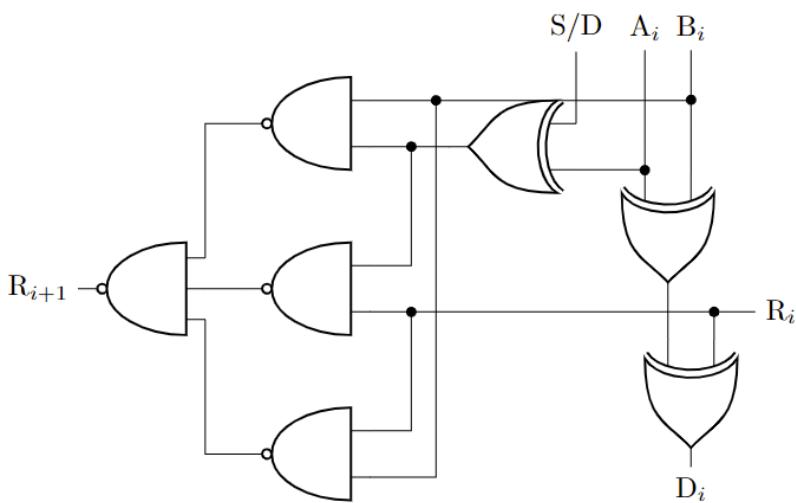
$$D_i = (A \bar{B} \bar{R} + \bar{A} B \bar{R} + \bar{A} \bar{B} R + A B R)_i$$

$$R_{i+1} = (\bar{A} B \bar{R} + \bar{A} \bar{B} R + \bar{A} B R + A B R)_i$$

$$D_i = (A \bar{B} + \bar{A} B) \bar{R}_i + (\bar{A} \bar{B} + A B) R_i = (A \oplus B)_i \oplus R_i$$

$$R_{i+1} = (\bar{A} B + \bar{A} R + B R)_i$$

Subtractor



$$D_i = (A \bar{B} \bar{R} + \bar{A} B \bar{R} + \bar{A} \bar{B} R + A B R)_i$$

$$R_{i+1} = (\bar{A} B \bar{R} + \bar{A} \bar{B} R + \bar{A} B R + A B R)_i$$

$$D_i = (A \bar{B} + \bar{A} B)_i \bar{R}_i + (\bar{A} \bar{B} + A B)_i R_i = (A \oplus B)_i \oplus R_i$$

$$R_{i+1} = (\bar{A} B + \bar{A} R + B R)_i$$

Adder

- Modular adder has a propagation delay that depends on the number of bits (n)
- Carry propagation delay: $2T_P \cdot n$
- Carry can be calculated in parallel
- Carry-lookahead
 - Generate a carry if A and B are 1: $G_i = A_i B_i$
 - Propagate a carry if A or B are 1: $P_i = A_i + B_i$ but also if $P_i = A_i \oplus B_i$
- $C_{i+1} = G_i + C_i P_i$
- $C_4 = A_3B_3 + A_2B_2(A_3 \oplus B_3) + A_1B_1(A_2 \oplus B_2)(A_3 \oplus B_3) + A_0B_0(A_1 \oplus B_1)(A_2 \oplus B_2)(A_3 \oplus B_3) + C_0(A_0 \oplus B_0)(A_1 \oplus B_1)(A_2 \oplus B_2)(A_3 \oplus B_3)$
- Parallel carry propagation delay: $3T_P$

Parity generator

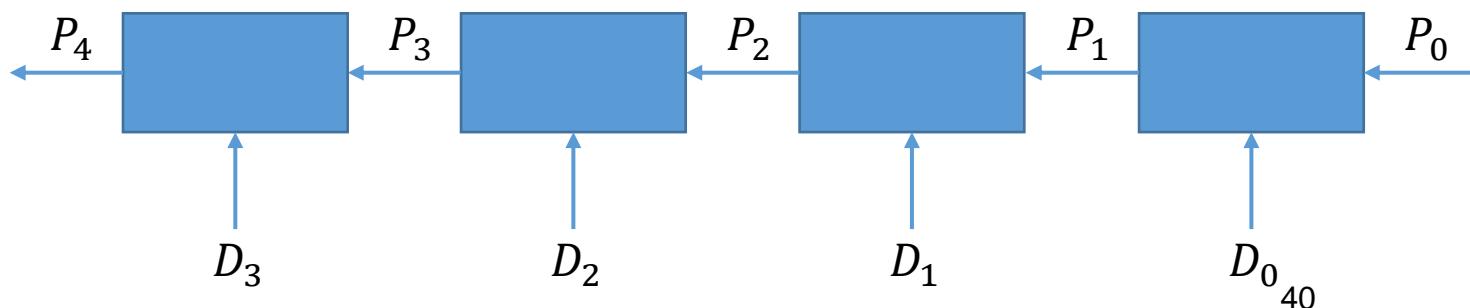
- The parity bit is a bit that added to the data makes the total number of ones (1s) even (even parity) or odd (odd parity)
- Parity is used for error detection during data transmission
- For instance, 1001010 has $P_{odd} = 0$ and $P_{even} = 1$
- Elementary cell for odd parity generation
- Extension to two n-bits numbers

D_i	P_i	P_{i+1}
0	0	0
1	0	1
0	1	1
1	1	0

$$P_{i+1} = (D \bar{P} + \bar{D} P)_i = (D \oplus P)_i$$
$$P_0 = 1$$

Parity generator

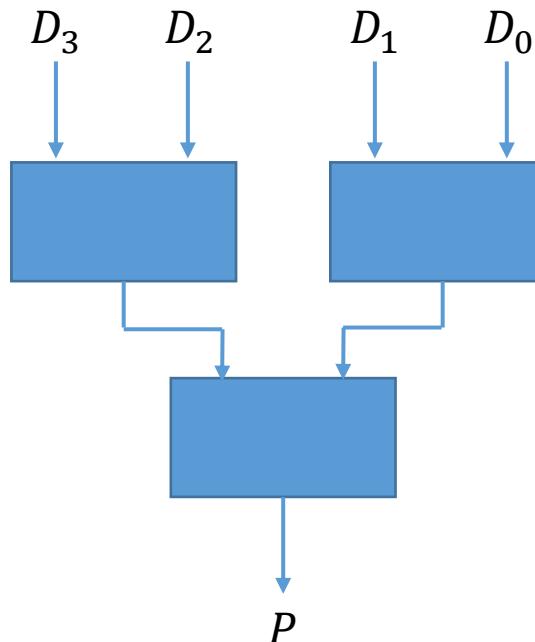
- The parity bit is a bit that added to the data makes the total number of ones (1s) even (even parity) or odd (odd parity)
- Parity is used for error detection during data transmission
- For instance, 1001010 has $P_{odd} = 0$ and $P_{even} = 1$
- Elementary cell for odd parity generation
- Extension to two n-bits numbers



$$P_{i+1} = (D \bar{P} + \bar{D} P)_i = (D \oplus P)_i$$
$$P_0 = 1$$

Parity generator

- And for even parity generator?
 - Just change P_0
- As for the adder we can limit the propagation time with a module that works in parallel
- It is possible to connect the single module in a tree-like structure

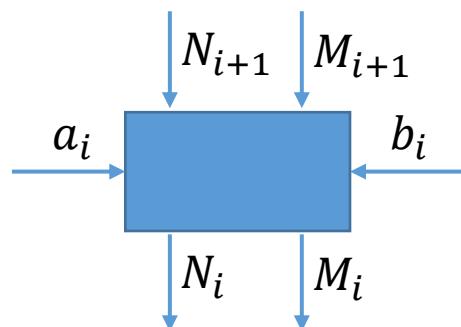


$$P = (D_0 \oplus D_1) \oplus (D_2 \oplus D_3)$$

Comparison

- For comparing two binary numbers (a, b), three output values are needed -> two output bits (N, M)
- Modular approach for n-bits extension

	N	M
$a > b$	0	1
$a < b$	1	0
$a = b$	0	0

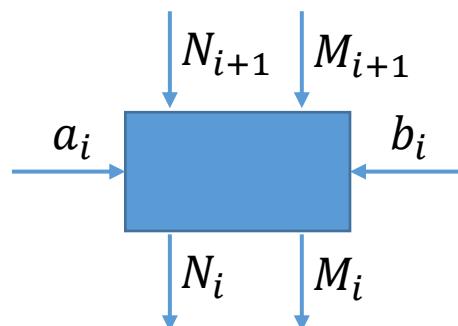


a_i	b_i	N_{i+1}	M_{i+1}	N_i	M_i
0	0	0	0	0	0
1	0	0	0	0	1
0	1	0	0	1	0
1	1	0	0	0	0
0	0	1	0	1	0
1	0	1	0	1	0
0	1	1	0	1	0
1	1	1	0	1	0
0	0	0	1	0	1
1	0	0	1	0	1
0	1	0	1	0	1
1	1	0	1	0	1
0	0	1	1	X	X
1	0	1	1	X	X
0	1	1	1	X	X
1	1	1	1	X	X

Comparison

- For comparing two binary numbers (a, b), three output values are needed -> two output bits (N, M)
- Modular approach for n-bits extension

	N	M
$a > b$	0	1
$a < b$	1	0
$a = b$	0	0



Redundant terms

a_i	b_i	N_{i+1}	M_{i+1}	N_i	M_i
0	0	0	0	0	0
1	0	0	0	0	1
0	1	0	0	1	0
1	1	0	0	0	0
0	0	1	0	1	0
1	0	1	0	1	0
0	1	1	0	1	0
1	1	1	0	1	0
0	0	0	1	0	1
1	0	0	1	0	1
0	1	0	1	0	1
1	1	0	1	0	1
0	0	1	1	X	X
1	0	1	1	X	X
0	1	1	1	X	X
1	1	1	1	X	X

Comparison

- For comparing two binary numbers (a, b), three output values are needed -> two output bits (N, M)
- Modular approach for n-bits extension

$$N_i = N_{i+1} + \overline{a_i} b_i \overline{M_{i+1}}$$

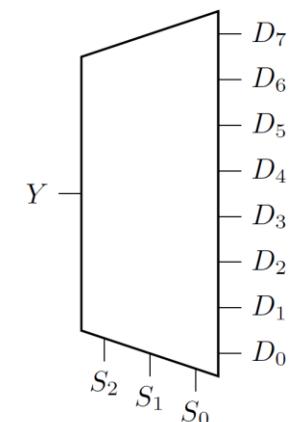
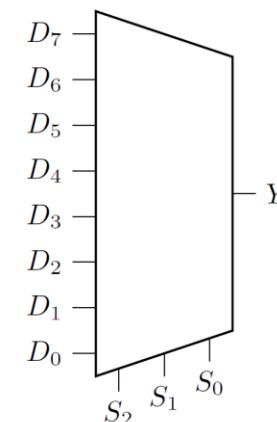
$$M_i = M_{i+1} + a_i \overline{b_i} \overline{N_{i+1}}$$

Redundant terms

a_i	b_i	N_{i+1}	M_{i+1}	N_i	M_i
0	0	0	0	0	0
1	0	0	0	0	1
0	1	0	0	1	0
1	1	0	0	0	0
0	0	1	0	1	0
1	0	1	0	1	0
0	1	1	0	1	0
1	1	1	0	1	0
0	0	0	1	0	1
1	0	0	1	0	1
0	1	0	1	0	1
1	1	0	1	0	1
0	0	1	1	X	X
1	0	1	1	X	X
0	1	1	1	X	X
1	1	1	1	X	X

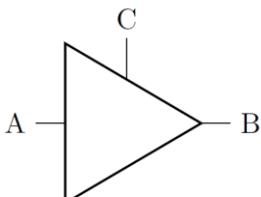
Multiplexer

- A Mux selects one signal among 2^n (D_i) thanks to n address lines (S_j)
- For $n = 1$, two levels of NAND ports can easily implement it
- Any combinatorial function can be generated connecting the address lines to the variables and connecting 1 to D_i if the minterm m_i is present, 0 otherwise
- A Demux applies the inverse operation
- It can be used as decoder because it selects the output line D_i corresponding to the minterm m_i

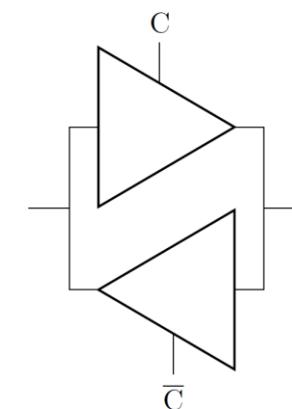


Tristate

- It is used for developing bidirectional connections
- Many data buses are usually tristate because they are used to link devices that can be both source and sink of information
- A control line is used for putting the output in high-impedance

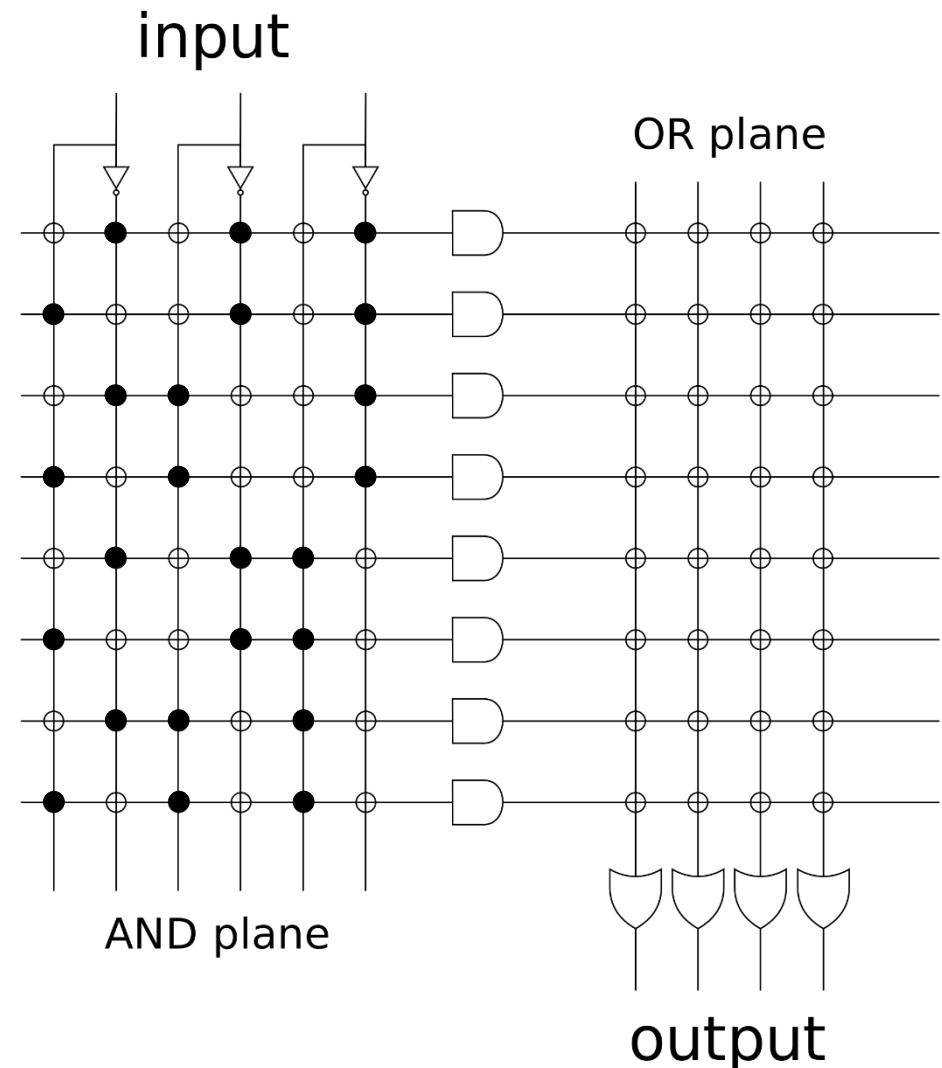


<i>A</i>	<i>C</i>	<i>B</i>
0	0	Z
1	0	Z
0	1	0
1	1	1



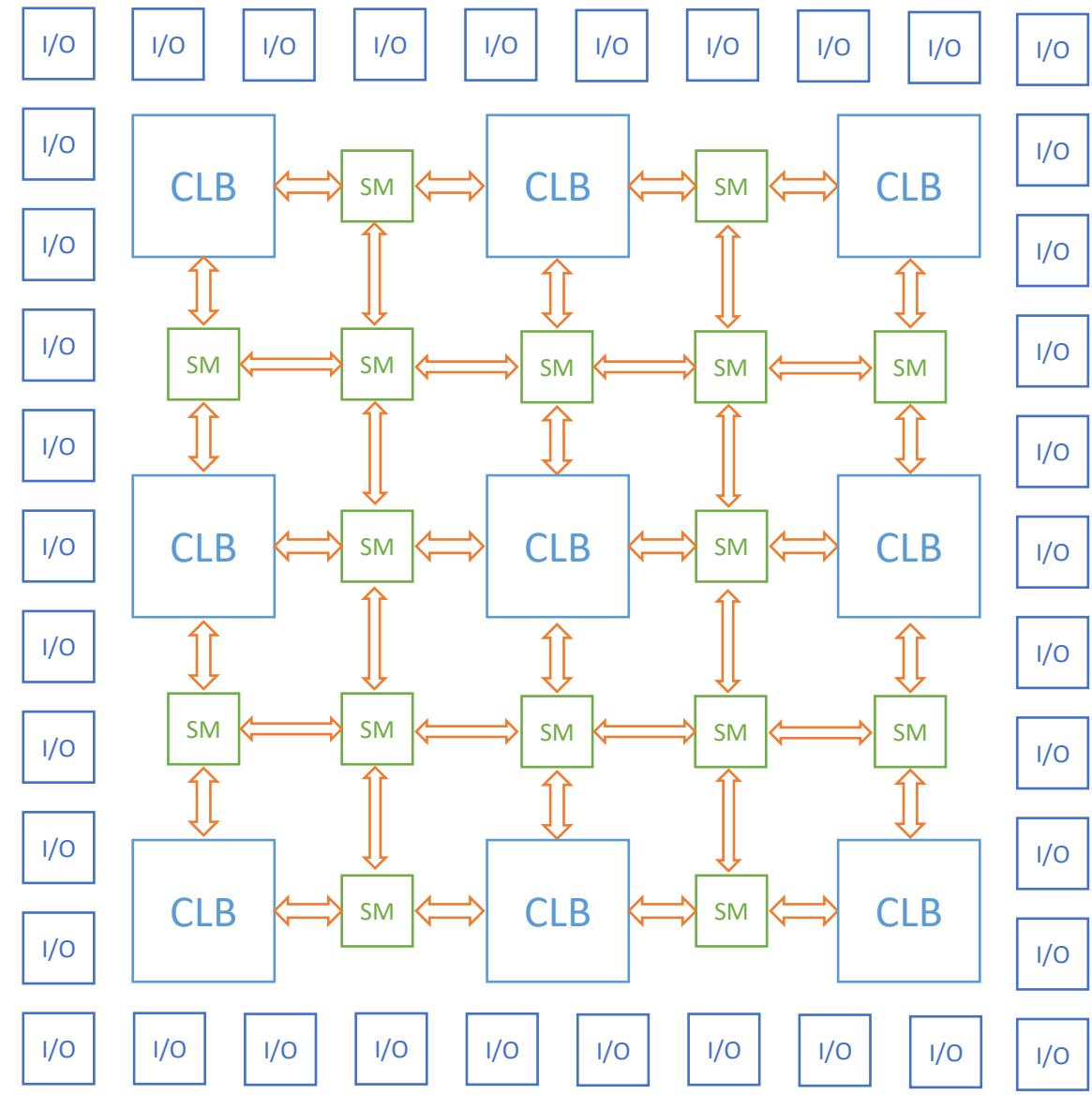
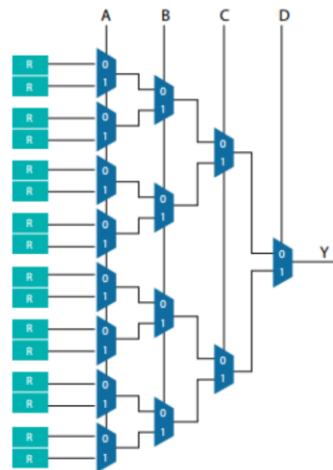
Programmable Logic Array

- PLA consists of an AND array connected to an OR array
- Constant propagation delay
- It can build any sum of products
- The AND array can be fixed while the OR array is programmable
- Programmable logic-planes grow too quickly in size as the number of inputs is increased



Field-Programmable Gate Array

- Programmable logic blocks linked by programmable interconnections
- Programmable Input and Output
- Configurable Logic Block can be either a combinatorial or a sequential circuit
- Look up tables are used to implement the truth table of a combinatorial function



Exercise

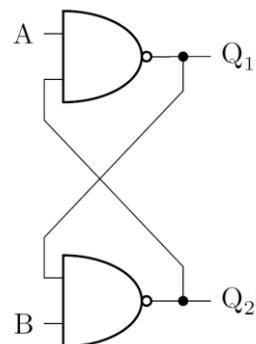
- Design a circuit able to compare two 2-bit numbers at a time

MANAGEMENT AND ANALYSIS OF PHYSICS DATASET (MOD. A)

Memory Elements
Sequential Circuits

Memory elements

- If both inputs are at 1, the circuit keep memory of the previous state

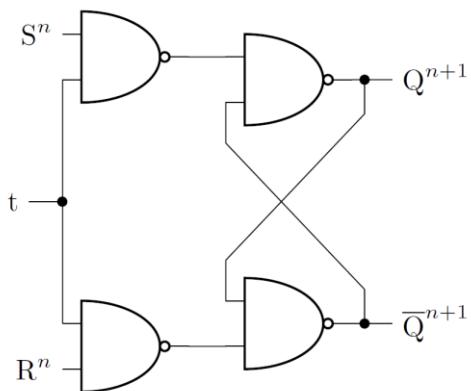


A	B	Q_2	Q_1
0	0	1	1
1	0	1	0
0	1	0	1
1	1	X	\overline{X}

- Latch

Memory elements

- Flip-flop set-reset (S-R)
 - A synchronization input t is added
 - If t is not present the output doesn't change



S^n	R^n	Q^{n+1}	\bar{Q}^{n+1}
0	0	Q^n	\bar{Q}^n
1	0	1	0
0	1	0	1
1	1	-	-

$$Q^{n+1} = (Q \bar{S} \bar{R} + S \bar{R})^n = (S + Q \bar{R})^n$$
$$S R = 0$$

Memory elements

- Flip-flop J-K
 - It removes the forbidden state
- Flip-flop D
 - Only one input
- How to build J-K and D?

J^n	K^n	Q^{n+1}	\bar{Q}^{n+1}
0	0	Q^n	\bar{Q}^n
1	0	1	0
0	1	0	1
1	1	\bar{Q}^n	Q^n

$$Q^{n+1} = (Q \bar{J} \bar{K} + J \bar{K} + \bar{Q} J K)^n = (Q \bar{K} + \bar{Q} J)^n$$

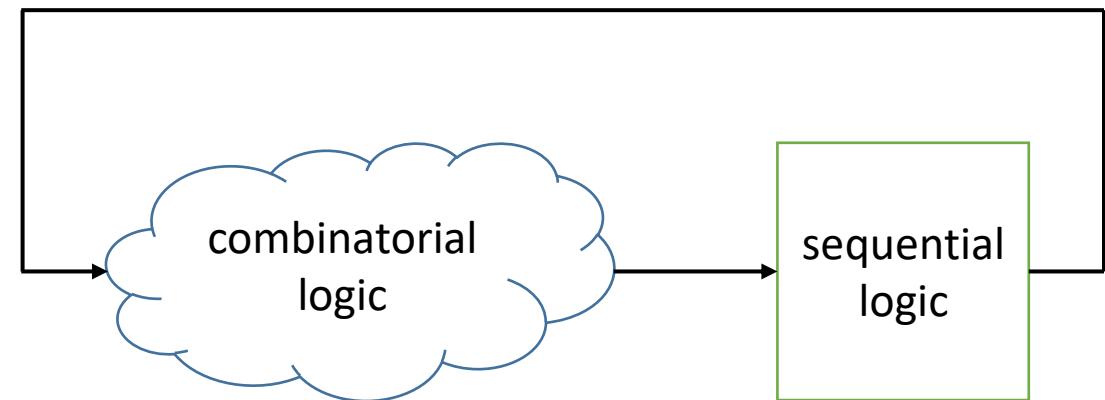
D^n	Q^{n+1}	\bar{Q}^{n+1}
0	0	1
1	1	0

$$Q^{n+1} = D^n$$

Memory elements

- Usually, memory elements are controlled by logic functions

a_0^n	a_1^n	Q^n	Q^{n+1}	S^n	R^n	J^n	K^n	D^n
0	0	0	0	0	X	0	X	0
1	0	0	0	0	X	0	X	0
0	1	0	1	1	0	1	X	1
1	1	0	1	1	0	1	X	1
0	0	1	0	0	1	X	1	0
1	0	1	1	X	0	X	0	1
0	1	1	0	0	1	X	1	0
1	1	1	1	X	0	X	0	1



$$Q^{n+1} = (a_0 Q + a_1 \bar{Q})^n$$

$$S = a_1 \bar{Q}$$

$$R = \overline{a_0} Q$$

$$J = a_1$$

$$K = \overline{a_0}$$

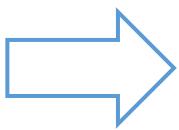
Memory elements

$$S = a_1 \bar{Q}$$

$$R = \bar{a}_0 Q$$

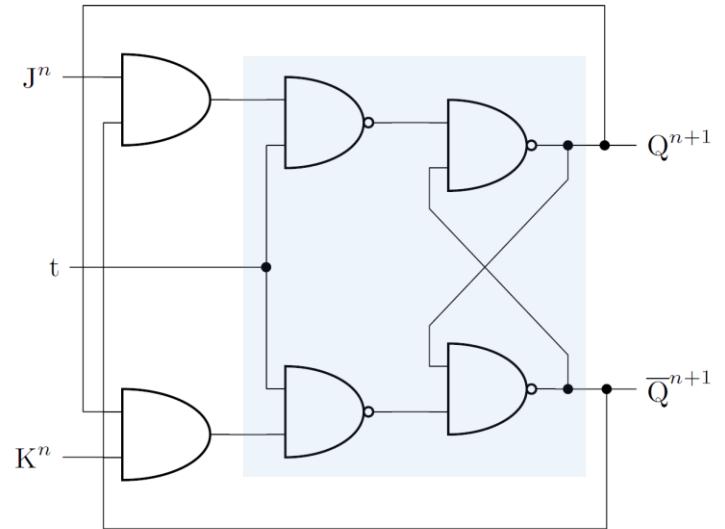
$$J = a_1$$

$$K = \bar{a}_0$$



$$S = J \bar{Q}$$

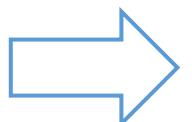
$$R = K Q$$



$$J = a_1$$

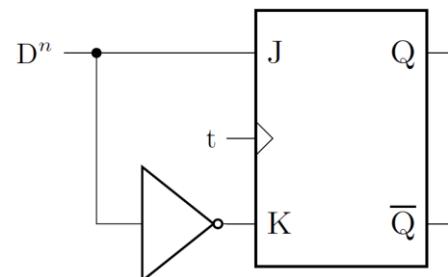
$$K = \bar{a}_0$$

$$D = a_0 Q + a_1 \bar{Q}$$



$$J = D$$

$$K = \bar{D}$$



Exercises

- Make a J-K flip-flop from a D flip-flop
- Find the truth table of the flip-flop with the following equation
$$Q^{n+1} = [Q \oplus (M \oplus N)]^n$$
- Make the above flip-flop from a J-K flip-flop

Counters and registers

Decimal counter

- Design a 0-9 counter
- $A^{n+1} = [\bar{A}]^n$
- $B^{n+1} = [B(\bar{A}) + \bar{B}(A \bar{D})]^n$
- $C^{n+1} = [C(\bar{A} + \bar{B}) + \bar{C}(A B)]^n$
- $D^{n+1} = [D(\bar{A}) + \bar{D}(A B C)]^n$
- 4 FF J-K and 3 logic ports
(assuming the complement of each variable is available)

D^n	C^n	B^n	A^n	D^{n+1}	C^{n+1}	B^{n+1}	A^{n+1}
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0

Johnson counter

- It is a counter without switching incertitude because it changes only one bit at a time
- m -bit Johnson counter \rightarrow sequence of $2m$ numbers
- If we use the redundant terms, we get

$$A^{n+1} = \overline{C^n}$$

$$B^{n+1} = A^n$$

$$C^{n+1} = B^n$$

- It nicely corresponds to a D flip-flop chain, but if the system starts by chance from a redundant term, it never gets out!
- We need to avoid that making the schematics a little bit more complex

$$A^{n+1} = \overline{C^n}$$

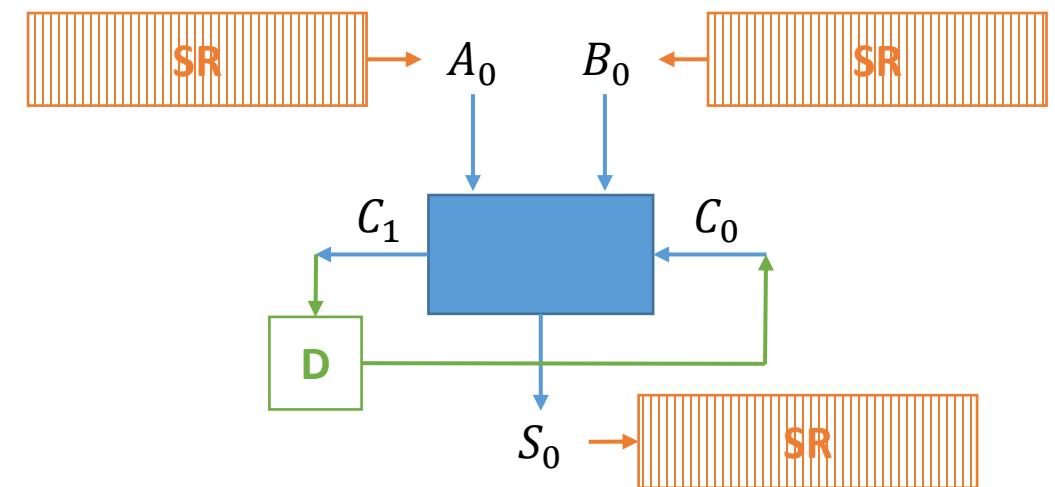
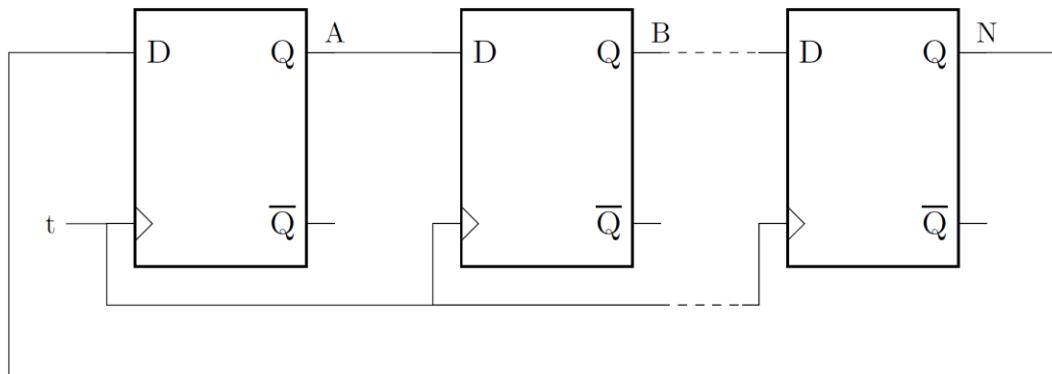
$$B^{n+1} = [A(B + \overline{C})]^n$$

$$C^{n+1} = B^n$$

A^n	B^n	C^n	A^{n+1}	B^{n+1}	C^{n+1}
0	0	0	1	0	0
1	0	0	1	1	0
1	1	0	1	1	1
1	1	1	0	1	1
0	1	1	0	0	1
0	0	1	0	0	0

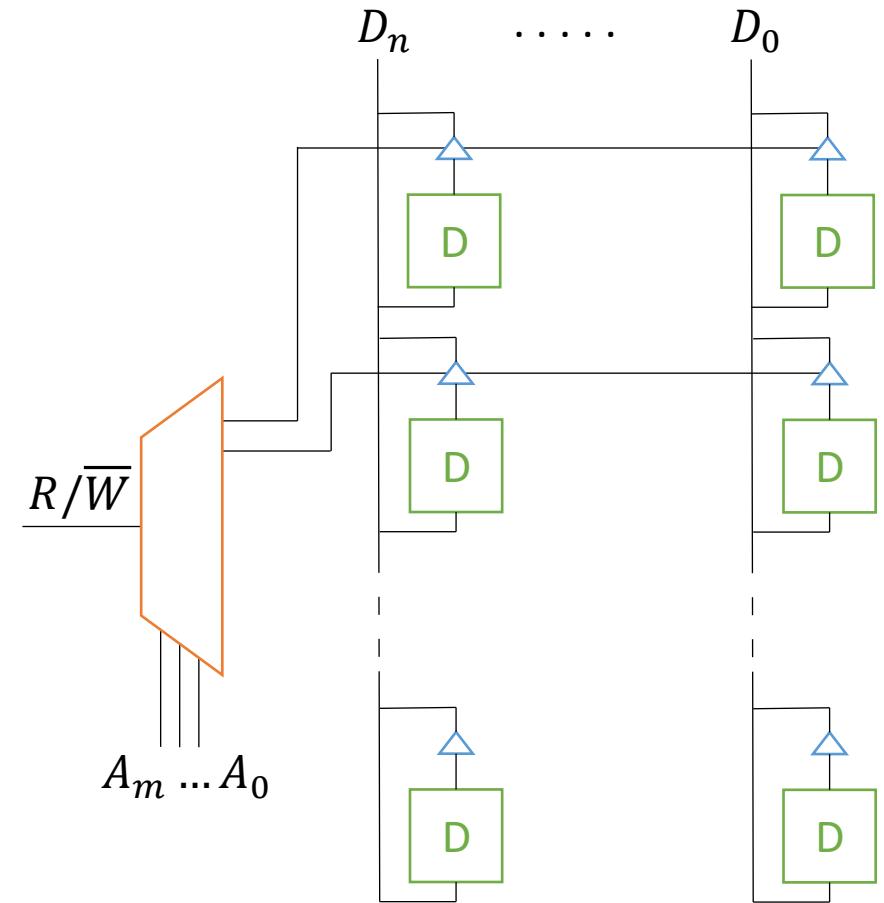
Shift register

- It is used to shift a signal
- Its design can be obtained by the sequential table
- Without feedback can be used to access in parallel to a serial data or to serialize a parallel data
- Each cell can have a multiplexer to choose if shifting the bit or loading a new bit



Memory

- It can be seen as an array of FF
- A decoder is used for addressing the data row
- A tristate allows read/write operation
- Memory size = data width $\times 2^{\text{address width}}$
- Random Access Memory (RAM)
 - If Read Only is called ROM
 - It can contain a truth table -> It implements any combinatorial function (one for each data bit)



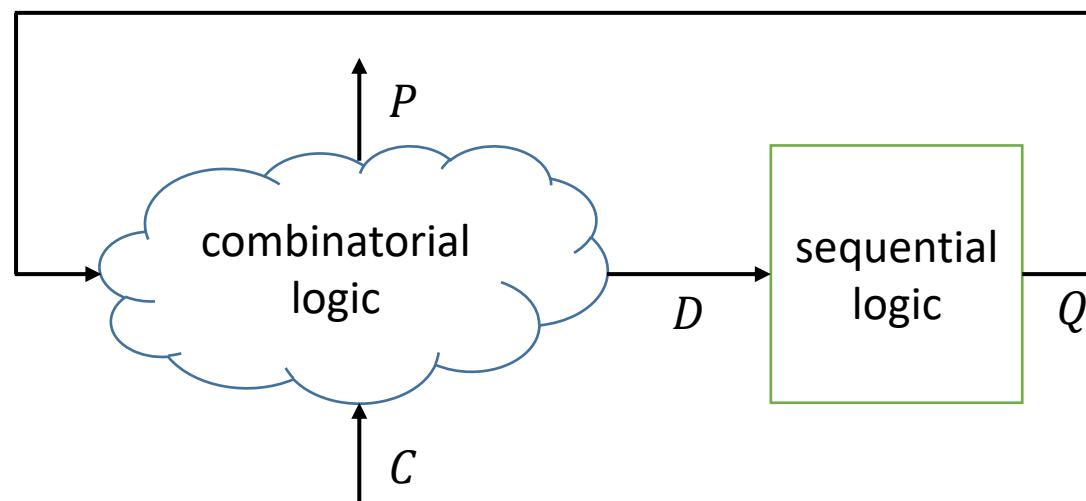
Exercises

- Design a 4-bit down counter
- Design an n -bit shift register that starts from a state where all the bits are equal to zero: $00\dots0 \rightarrow 10\dots0 \rightarrow 01\dots0$, etc.

State machines

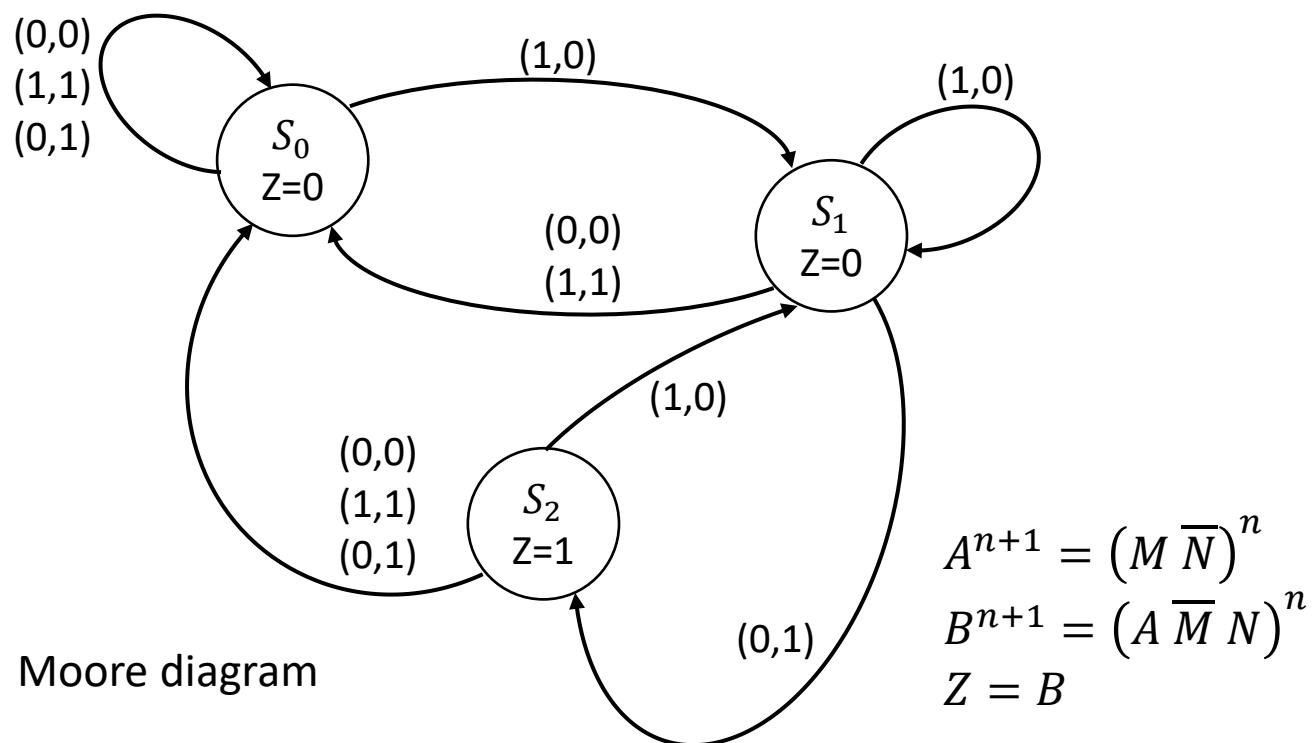
State machine

- A general schema for functions that control sequential logic take into account external signals C and produce control signals P
 - Mealey state machine: P is function of C and Q
 - Moore state machine: P is only function of Q
- Moore outputs are synchronous



State machine

- Find the transition $(1,0) \rightarrow (0,1)$ of two bits (M,N)

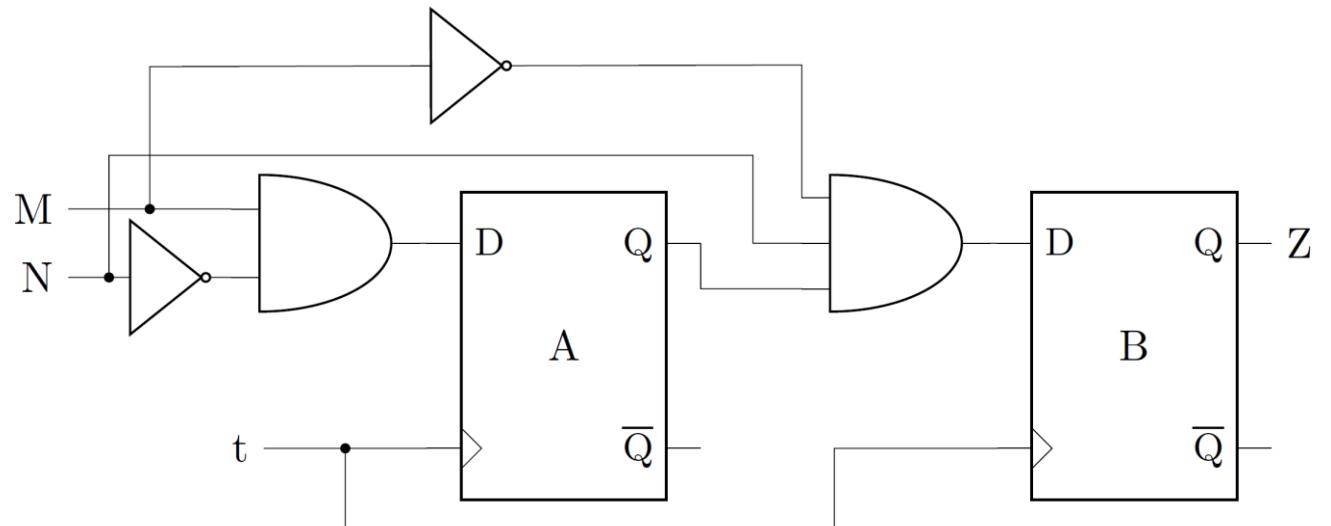


M	N	A^n	B^n	A^{n+1}	B^{n+1}	Z
0	0	0	0	0	0	0
1	0	0	0	1	0	0
0	1	0	0	0	0	0
1	1	0	0	0	0	0
0	0	1	0	0	0	0
1	0	1	0	1	0	0
0	1	1	0	0	1	1
1	1	1	0	0	0	0
0	0	0	1	0	0	0
1	0	0	1	1	0	0
0	1	0	1	0	0	0
1	1	0	1	0	0	0

State machine

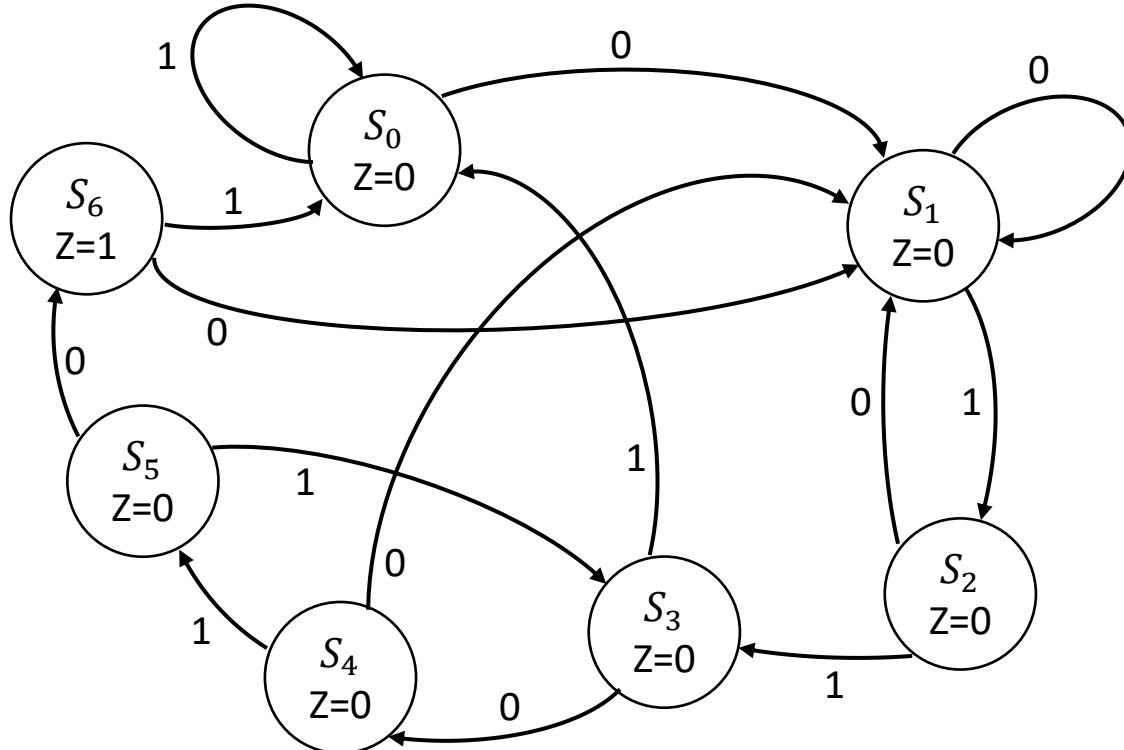
- Find the transition $(1,0) \rightarrow (0,1)$ of two bits (M,N)

$$\begin{aligned}A^{n+1} &= (M \bar{N})^n \\B^{n+1} &= (A \bar{M} N)^n \\Z &= B\end{aligned}$$



State machine

- Find the sequence ...011010... on a data line M



M	A^n	B^n	C^n	A^{n+1}	B^{n+1}	C^{n+1}
0	0	0	0	1	0	0
1	0	0	0	0	0	0
0	1	0	0	1	0	0
1	1	0	0	0	1	0
0	0	1	0	1	0	0
1	0	1	0	1	1	0
0	1	1	0	0	0	1
1	1	1	0	0	0	0
0	0	0	1	1	0	0
1	0	0	1	1	0	1
0	1	0	1	0	1	1
1	1	0	1	1	1	0
0	0	1	1	1	0	0
1	0	1	1	0	0	0

Exercises

- Complete the previous example designing the state machine circuit with the use of J-K flip-flops
- Design a controller for an elevator
 - The elevator can be at one of the three floors: Ground, First and Second
 - There is a button that controls the elevator, and it has two possible values: up and down
 - The elevator goes up one floor every time you set the button to “up”, and goes down one floor every time you set the button to “down”
 - There are two lights in a row in the elevator that indicate the current floor: both lights off (00) indicates the ground floor; the left light off and right light on (01) indicates the first floor; the right light off and left right on (10) indicates the second floor

MANAGEMENT AND ANALYSIS OF PHYSICS DATASET (MOD. A)

Number Systems
Arithmetic Operations

Binary numbers

- Computers doesn't work with decimal numbers
 - Binary numbers are more reliable -> two-states discrimination
- Decimal number system uses ten *digits* (from 0 to 9)
 - Reset-and-carry
 - The units reset to zero and carry to the tens...
- Binary number system uses two *digits* (0 and 1)
 - Still Reset-and-carry



Binary numbers

- Computers doesn't work with decimal numbers
 - Binary numbers are more reliable -> two-states discrimination
- Decimal number system uses ten *digits* (from 0 to 9)
 - Reset-and-carry
 - The units reset to zero and carry to the tens...
- Binary number system uses two *digits* (0 and 1)
 - Still Reset-and-carry
- Base or radix of a number system is the number of digits
- *Bit* stands for binary digit
- Binary is less compact than decimal

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

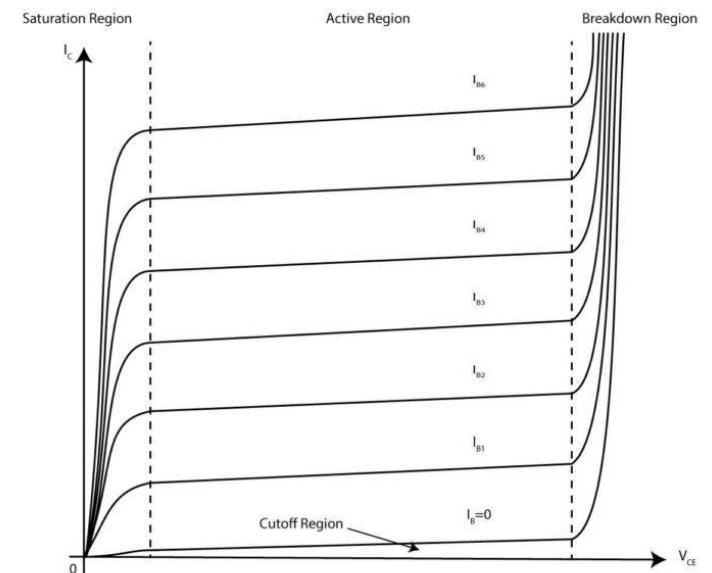
Binary numbers

- Computers doesn't work with decimal numbers
 - Binary numbers are more reliable -> two-states discrimination
- Decimal number system uses ten *digits* (from 0 to 9)
 - Reset-and-carry
 - The units reset to zero and carry to the tens...
- Binary number system uses two *digits* (0 and 1)
 - Still Reset-and-carry
- Base or radix of a number system is the number of digits
- *Bit* stands for binary digit
- Binary is less compact than decimal
- They are equivalent (ten pebbles are 10_{10} pebbles or 1010_2 pebbles)



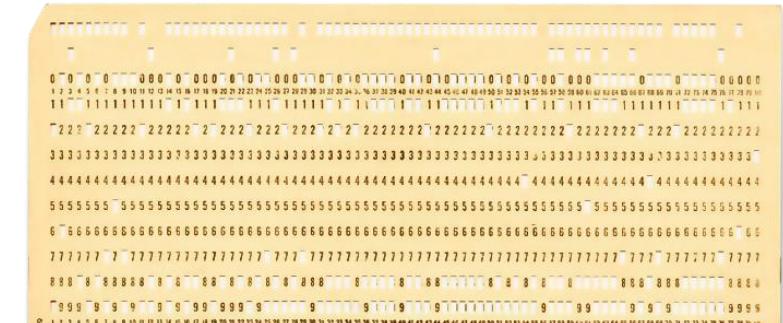
Binary numbers

- Computer can store Program and Data only in a binary form
- Transistors have parameters that can vary of more than 50% with temperature and among them for manufacturing differences
- Two-state design uses only two working points
 - For instance, cutoff and saturation



Binary numbers

- Computer can store Program and Data only in a binary form
- Transistors have parameters that can vary of more than 50% with temperature and among them for manufacturing differences
- Two-state design uses only two working points
 - For instance, cutoff and saturation
- Punched cards are another example of Program and Data storing



Binary numbers

- 4 bit are a *nibble*
- 8 bit are a *byte*
- 16 bit are a *halfword*
- 32 bit are a (*single*)*word*
- 64 bit are a *doubleword*
- 128 bit are a *quadword*
- A *string* is a group of character (either letters or digits) written one after the other
- *Word* can also refer to a string of bits of a specific length (i.e., a 24-bit word)
- The abbreviation *K* stands for 1024 (powers of two!)
 - 64K is 65536
- *b* is the abbreviation of bit, *B* of byte
 - 1 MB = 1048576 B = 8388608 b

Binary-to/from-decimal

- Both systems use positional notation
 - Decimal -> power of ten
 - Binary -> power of two
- $38572 = 3 \cdot 10^4 + 8 \cdot 10^3 + 5 \cdot 10^2 + 7 \cdot 10^1 + 2 \cdot 10^0$
- $10110100 = 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$
- Double-dabble algorithm

19	
9	1
4	1
2	0
1	0
0	1

-> 10011

Hexadecimal numbers

- Number system with base 16
 - Digits are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Shorter representation than binary
- Easy to convert
 - Nibble by nibble
 - $A3C = 101000111100$
- It uses positional notation as well
 - $A3C = A \cdot 16^2 + 3 \cdot 16^1 + C \cdot 16^0 = 2620_{10}$
- Hex-dabble algorithm

2620
163 C
10 3
0 A -> A3C

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

Exercises

- Convert 82_{10} in binary
- Convert $10A4_{16}$ in binary
- Convert 650_{10} in hex

Arithmetic Operations

Binary addition

- Simple additions
 - $0 + 0 = 0$
 - $0 + 1 = 1$
 - $1 + 0 = 1$
 - $1 + 1 = 10$ (zero, carry one)
 - $1 + 1 + 1 = 11$ (one, carry one)
- Larger numbers

$$\begin{array}{r} 11100 \\ + 11010 \\ \hline 110110 \end{array}$$

Binary subtraction

- Simple subtractions

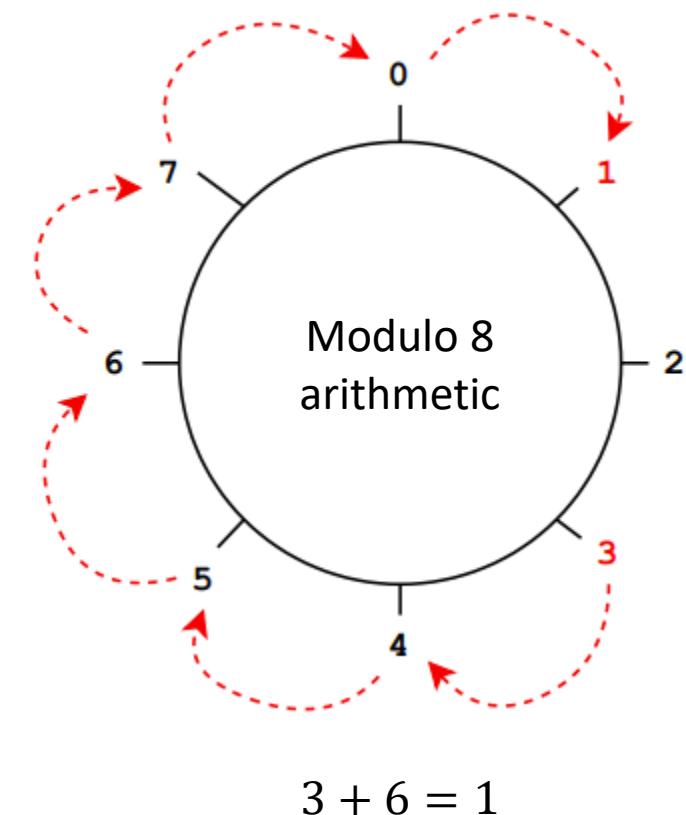
- $0 - 0 = 0$
- $1 - 0 = 1$
- $1 - 1 = 0$
- $10 - 1 = 1$

- Larger numbers

$$\begin{array}{r} 1101 - \\ 1010 = \\ 0011 \end{array}$$

Modular arithmetic

- When the set of number is finite, we use modular arithmetic
- Numbers “wrap around”
- $a \equiv b \pmod{n} \rightarrow a = kn + b$
 - $25 \pmod{8} = 1$
 - $24 \pmod{8} = 0$
 - $17 \pmod{3} = 2$
- Examples of operations on 3 bit ($\pmod{2^3}$)
 - $3 + 2 = 5$
 - $3 + 6 = 1$
 - $7 - 5 = 2$
 - $2 - 5 = 5$



Signed binary numbers

- Simply setting one as MSB (MSB as sign)
 - For instance, in an 8-bit representation: $9 \rightarrow 00001001$ $-9 \rightarrow 10001001$
 - +0 and -0
 - $9 + (-9) \neq +0$ $9 + (-9) \neq -0$
- Inverting all the bits (one's complement)
 - For instance, in an 8-bit representation: $9 \rightarrow 00001001$ $-9 \rightarrow 11110110$
 - +0 and -0
 - $9 + (-9) = -0$
- Inverting all the bits and adding one (two's complement)
 - For instance, in an 8-bit representation: $9 \rightarrow 00001001$ $-9 \rightarrow 11110111$
 - Only one 0
 - $9 + (-9) = 0$ (plus a carry)
 - In general, all the sums work: $9 + (-3) = 6$ $00001001 + 11111101 = 100000110$

2's complement operations

- Given k bit, we can represent the numbers in the interval $[-2^{k-1}, 2^{k-1} - 1] \subset \mathbb{Z}$
- The most significant bit tells us if the number is positive or negative
- For instance, 8 bit \rightarrow 256 numbers from -128 to 127
 - 2+5 \rightarrow 11111110+
00000101=
100000011
 - 5-3 \rightarrow 00000101+
11111100=
100000010

10000000	-128
...	...
11111110	-2
11111111	-1
00000000	0
00000001	1
00000010	2
...	...
01111111	127

Binary multiplication

- Same as decimal multiplication
- For instance

1101 ·

1011 =

1101

1101

0000

1101

10001111

- If M has n -bit and Q has m -bit, $M \cdot Q$ has $n+m$ bit

Binary division

- Same as decimal division
- For instance

$$\begin{array}{r} 100010010 \quad / \quad 1101 \\ 1101 \qquad \qquad 10101 \\ 10000 \\ 1101 \\ 1110 \\ 1101 \\ 1 \end{array}$$

- If M has n -bit and P has m -bit, M/P has $n-m$ bit

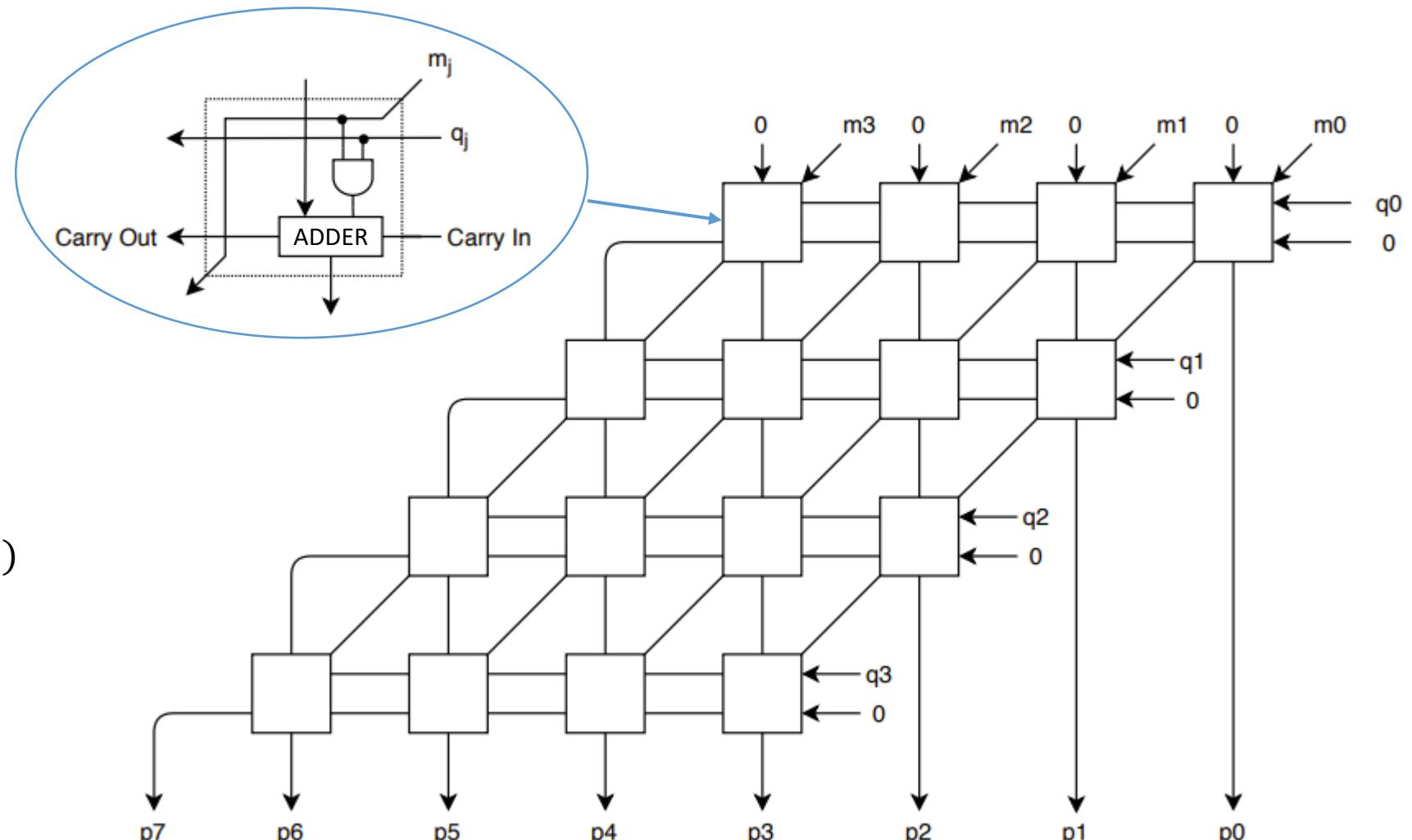
Parallel multiplier

$$P = M \cdot Q$$

$$M = (m_3, m_2, m_1, m_0)$$

$$Q = (q_3, q_2, q_1, q_0)$$

$$P = (p_7, p_6, p_5, p_4, p_3, p_2, p_1, p_0)$$



Exercises

- Calculate the binary sum $11100+10011$
- Calculate the binary subtraction $100101-11011$
- Calculate the binary multiplication $1011 \cdot 1001$
- Calculate the binary division $1111 \div 110$
- Calculate $10011-11100$ using 2's complement method

Bit manipulation

Bitwise operations

- Operators that work on a bit array at the level of its individual bit
 - NOT (complement)
 - NOT 10010101 = 01101010
 - AND
 - 0111 AND 1010 = 0010
 - OR
 - 0011 OR 0100 = 0111
 - XOR
 - 0010 XOR 1010 = 1000
- Quite often a single register has several bits with different functionalities
- To manipulate the single bit, we do bit-masking

Bitwise operations

Register 135. Reset/Freeze/Memory Control

Bit	D7	D6	D5	D4	D3	D2	D1	D0
Name	RST_REG	NewFreq	Freeze M	Freeze VCADC	N/A		RECALL	
Type	R/W	R/W	R/W	R/W	R/W		R/W	

Bit	Name	Function
7	RST_REG	Internal Reset. 0 = Normal operation. 1 = Reset of all internal logic. Output tristated during reset. Upon completion of internal logic reset, RST_REG is internally reset to zero. Note: Asserting RST_REG will interrupt the I ² C state machine. It is not the recommended approach for starting from initial conditions.
6	NewFreq	New Frequency Applied. Alerts the DSPLL that a new frequency configuration has been applied. This bit will clear itself when the new frequency is applied.
5	Freeze M	Freezes the M Control Word. Prevents interim frequency changes when writing RFREQ registers.
4	Freeze VCADC	Freezes the VC ADC Output Word. May be used to hold the nominal output frequency of an Si571.
3:1	N/A	Always Zero.
0	RECALL	Recall NVM into RAM. 0 = No operation. 1 = Write NVM bits into RAM. Bit is internally reset following completion of operation. Note: Asserting RECALL reloads the NVM contents into the operating registers without interrupting the I ² C state machine. It is the recommended approach for starting from initial conditions.

Bit-masking

- For setting a bit to 1, we do an OR with a mask done by all 0s except the bit we want to set
 - Set bit 3 of register R0

R0	B7	B6	B5	B4	B3	B2	B1	B0	OR
Mask	0	0	0	0	1	0	0	0	=
R0	B7	B6	B5	B4	1	B2	B1	B0	

- R0 OR 0x08

Bit-masking

- For clearing a bit to 0, we do an AND with a mask done by all 1s except the bit we want to clear
 - Clear bit 3 of register R0

R0	B7	B6	B5	B4	B3	B2	B1	B0	AND
Mask	1	1	1	1	0	1	1	1	=
R0	B7	B6	B5	B4	0	B2	B1	B0	

- R0 AND 0xF7

Bit-masking

- For inverting a bit, we do a XOR with a mask done by all 0s except the bit we want to invert
 - Invert bit 3 of register R0

R0	B7	B6	B5	B4	B3	B2	B1	B0	XOR
Mask	0	0	0	0	1	0	0	0	=
R0	B7	B6	B5	B4	$\overline{B3}$	B2	B1	B0	

- R0 XOR 0x08

Bit-masking

- For testing a bit, we do an AND with a mask done by all 0s except the bit we want to test and then we check if the result is equal to 0
 - Test bit 3 of register R0

R0	B7	B6	B5	B4	B3	B2	B1	B0	AND
Mask	0	0	0	0	1	0	0	0	=
R0	0	0	0	0	B3	0	0	0	

- $R0 \text{ AND } 0x08 = ?$ $0x00$

Exercises

- Check if the MSB of a 32-bit register is set
- Get the third bit (starting from the LSB) of an 8-bit register

Real numbers

Real numbers representation

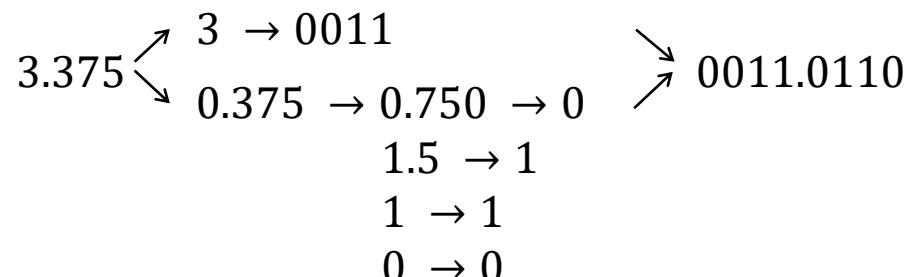
- As for \mathbb{Z} , with a finite number of bit we can represent only a subset of \mathbb{R}
- Any number belonging to \mathbb{R} must be approximated
- Two main techniques
 - Fixed point
 - Floating point

Fixed point

- Given k bit, we set a number of bit $r < k$ for the fractional part
- The weight of each bit of the fractional part is given by $2^{-1}, 2^{-2}, \dots$
- The integer part will have $k - r$ bit
- For instance, if $k = 8$ and $r = 4$ the number 00110110 means:

$$\begin{aligned}00110110 &= 0011.0110 \\&= 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} \\&= 2 + 1 + 0.25 + 0.125 = 3.375\end{aligned}$$

- Real to binary conversion algorithm
 - Usual algorithm for the integer part
 - Fractional part multiplied by two, extraction of the integer part and so on
 - For instance, if $k = 8$ and $r = 4$ the number 3.375 becomes:



Floating point

- Fixed point represents a too small subset of \mathbb{R}
- For being more flexible floating point uses a *significand* or *mantissa* and an *exponent*
- A typical example is IEEE 754 the float of C/C++
 - 1 bit for sign S
 - 8 bit for exponent E
 - 23 bit for mantissa M
 - $(-1)^S \cdot 1.M \cdot 2^{E-127}$
- For instance
 - $BFA00000_{16} = 10111111010000000000000000000000$
 - $= (-1)^1 \cdot 1.01_2 \cdot 2^{127-127} = -1.25$

Sum of floating points

- $M_1 \cdot 2^{E_1} + M_2 \cdot 2^{E_2}$
- Let's assume $E_1 < E_2$, we get M'_1 as a right shift of M_1 of $E_2 - E_1$ positions
- The mantissa of the sum will be $M_s = M'_1 + M_2$
- So, the sum will be $M_s \cdot 2^{E_2}$
- Finally, the result is normalized to the form $1.M \cdot 2^E$

$$1.0001 \cdot 2^2 + 1.1101 \cdot 2^4 =$$

$$0.010001 \cdot 2^4 + 1.1101 \cdot 2^4 =$$

$$10.000101 \cdot 2^4 =$$

$$1.0000101 \cdot 2^5$$

Multiplication of floating points

- $M_1 \cdot 2^{E_1} \cdot M_2 \cdot 2^{E_2} = (M_1 \cdot M_2) \cdot 2^{E_1+E_2}$
- The result is then normalized to the form $1.M \cdot 2^E$

- $M_1 \cdot 2^{E_1} / M_2 \cdot 2^{E_2} = (M_1/M_2) \cdot 2^{E_1-E_2}$
- The result is then normalized to the form $1.M \cdot 2^E$

$$\begin{aligned}1.0001 \cdot 2^2 \cdot 1.1101 \cdot 2^4 &= \\(1.0001 \cdot 1.1101) \cdot 2^{(2+4)} &= \\1.11101101 \cdot 2^6\end{aligned}$$

ASCII code

- To input and output data to/from a computer we need numbers, letters and other symbols
- *American Standard Code for Information Interchange* is a standard input/output code
- 7-bit code
- 0x48 0x65 0x6C 0x6C 0x6F -> Hello
- 0x48 0x65 0x6C 0x6C 0x77 0x08 0x6F
-> Hello

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1100000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1100001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1100000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1100001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1100010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1100011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1101000	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1101001	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1101100	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1101111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111000	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111011	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111100	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	^					
46	2E	101110	56	.	94	5E	1011110	136	_					
47	2F	101111	57	/	95	5F	1011111	137	-					

Exercises

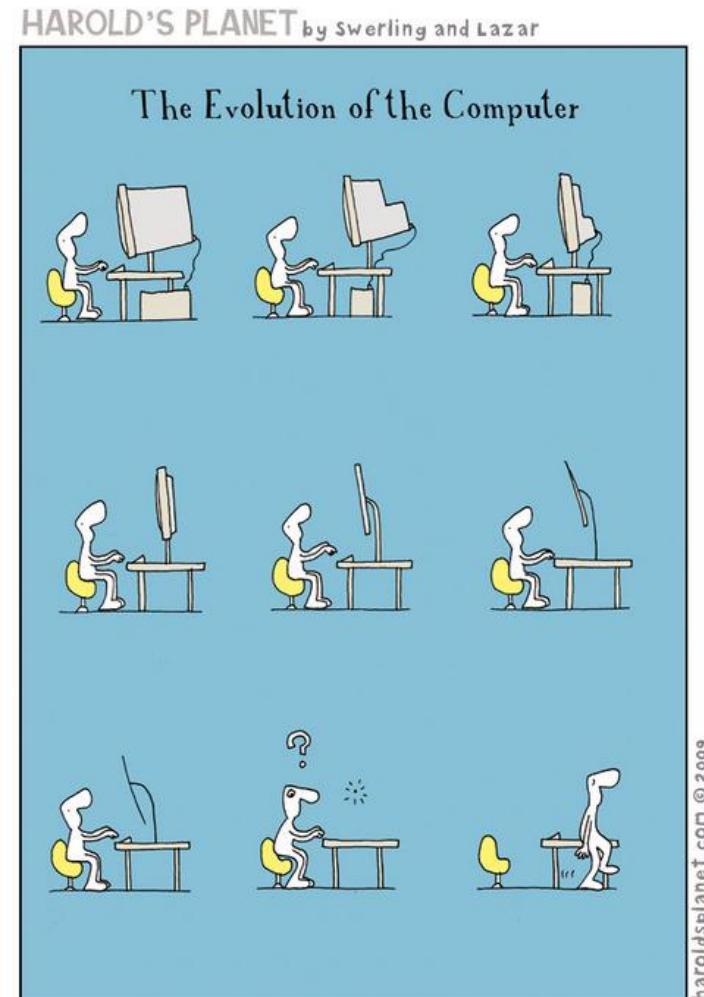
- Find the decimal value of the binary number 11101.011
- Calculate $23.75_{10} + 4.5_{10}$ using floating point

MANAGEMENT AND ANALYSIS OF PHYSICS DATASET (MOD. A)

SAP-1 computer

Simple As Possible computer

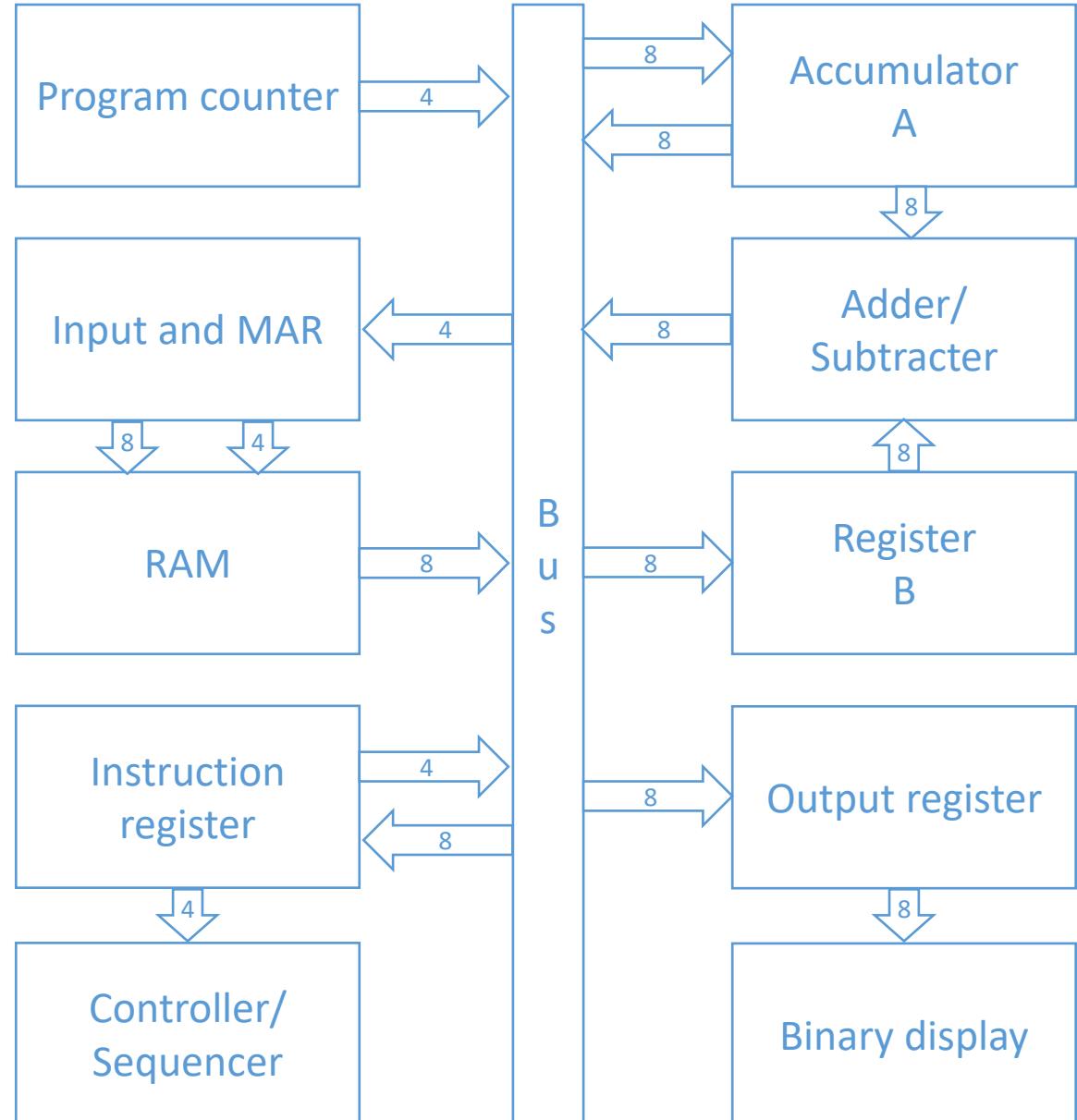
- A computer designed to show all the main ideas behind computer operation without unnecessary detail
- As simple as possible, but no simpler
- Three different generation will be presented SAP-1-2-3
- It is the evolution towards modern computer



SAP-1

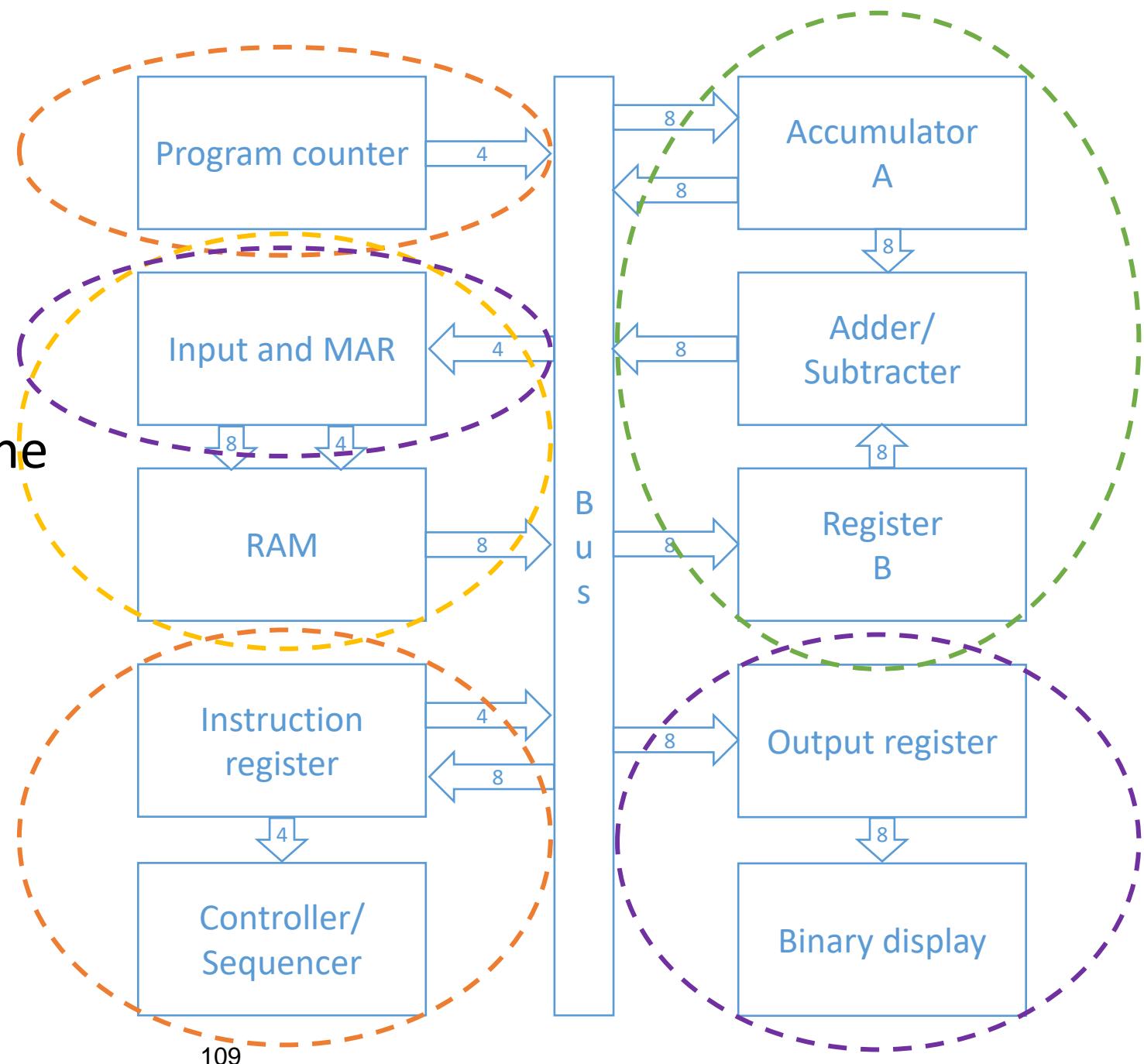
Architecture

- Simplified schema without clock, reset and controls
- All the register outputs to the bus are three-state



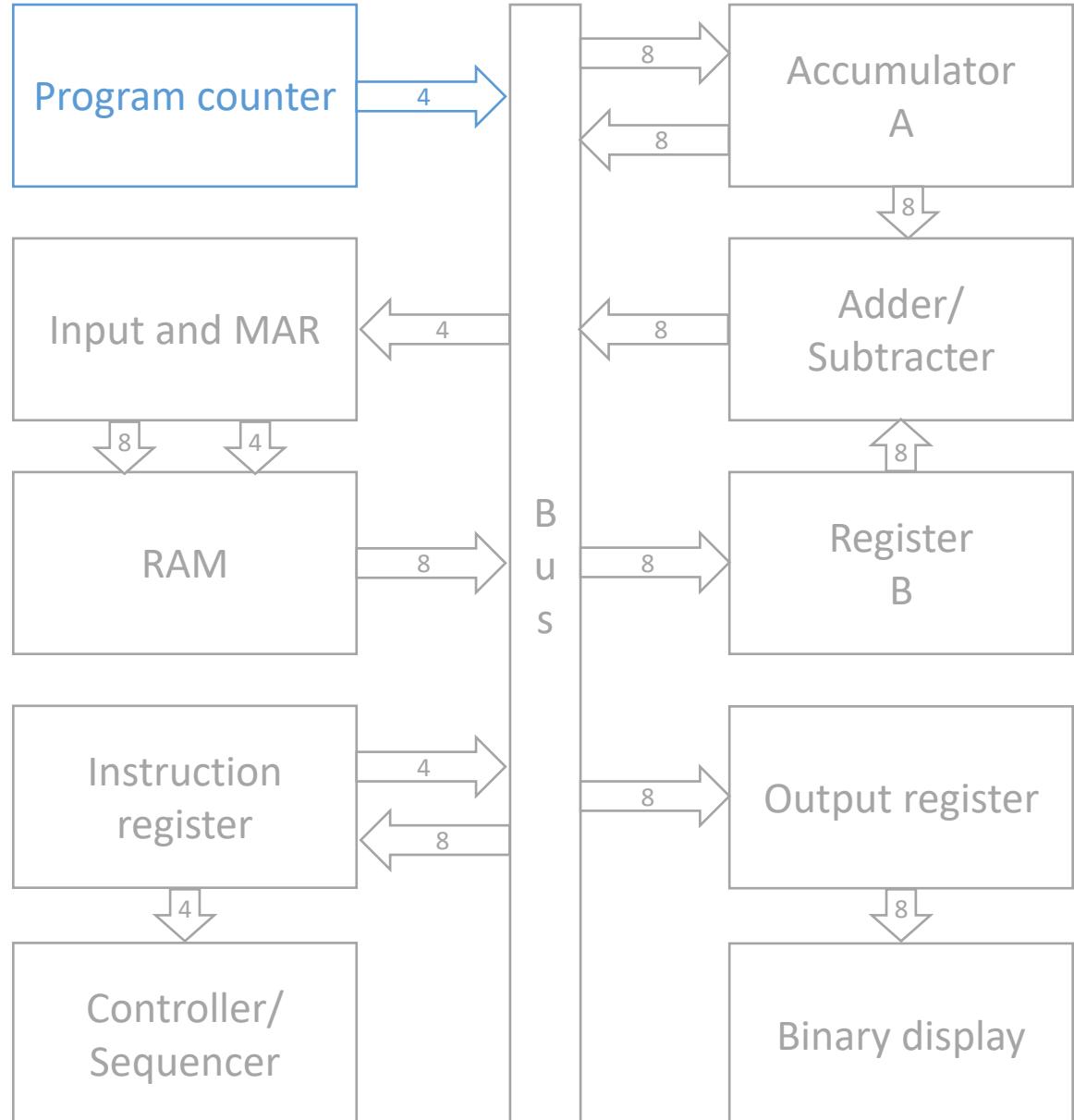
Architecture

- Simplified schema without clock, reset and controls
- All the register outputs to the bus are three-state
- It is built by a **Control Unit**, an **ALU**, a **Memory** and an **I/O Unit**



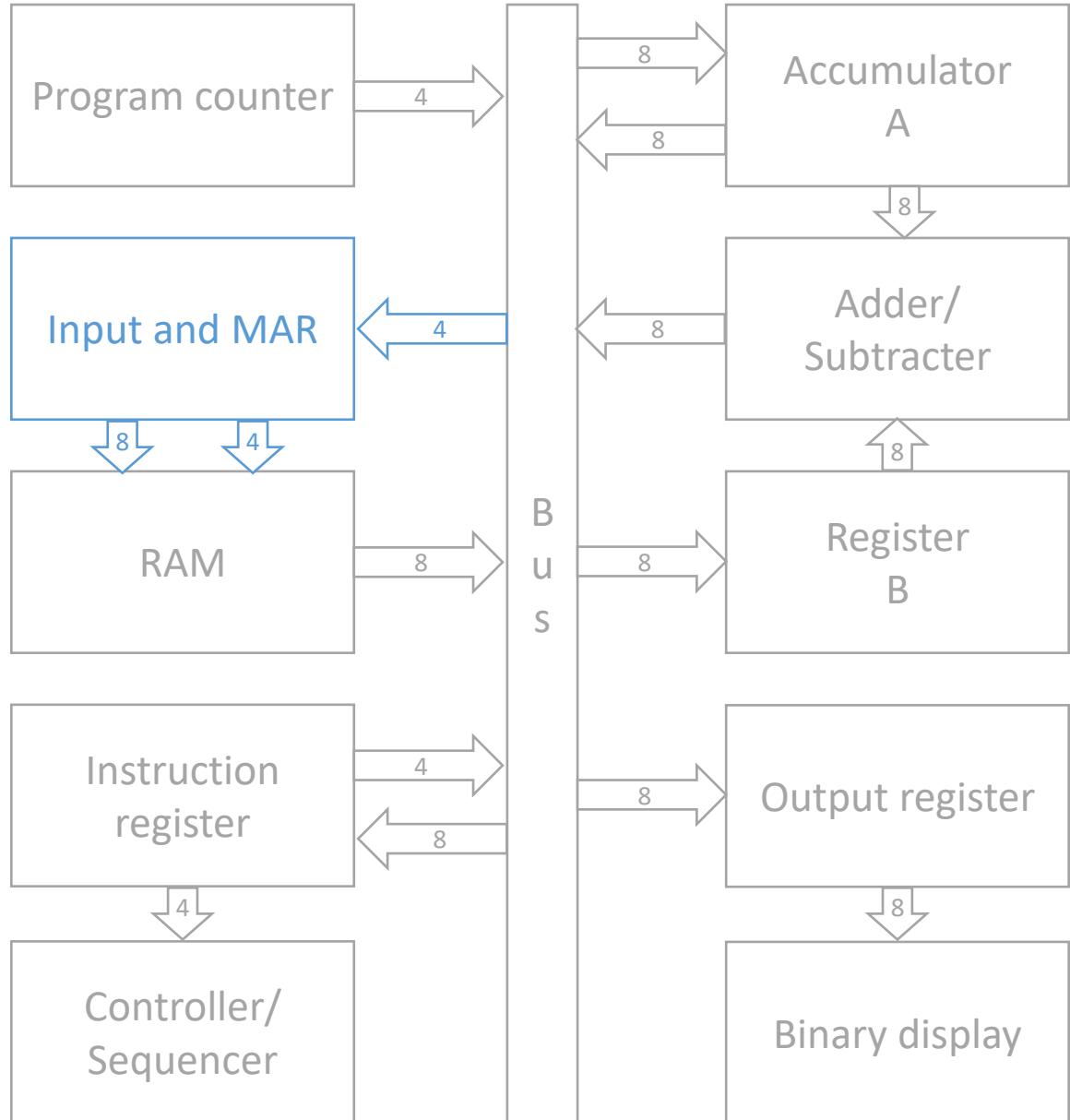
Architecture

- Program counter counts from 0000 to 1111
- It sends to the memory the address of the next instruction to be fetch and executed
- Before a run it is reset to 0000
- It is incremented each fetch and execution cycle
- It is also called pointer because it points to the address where the next instruction is stored



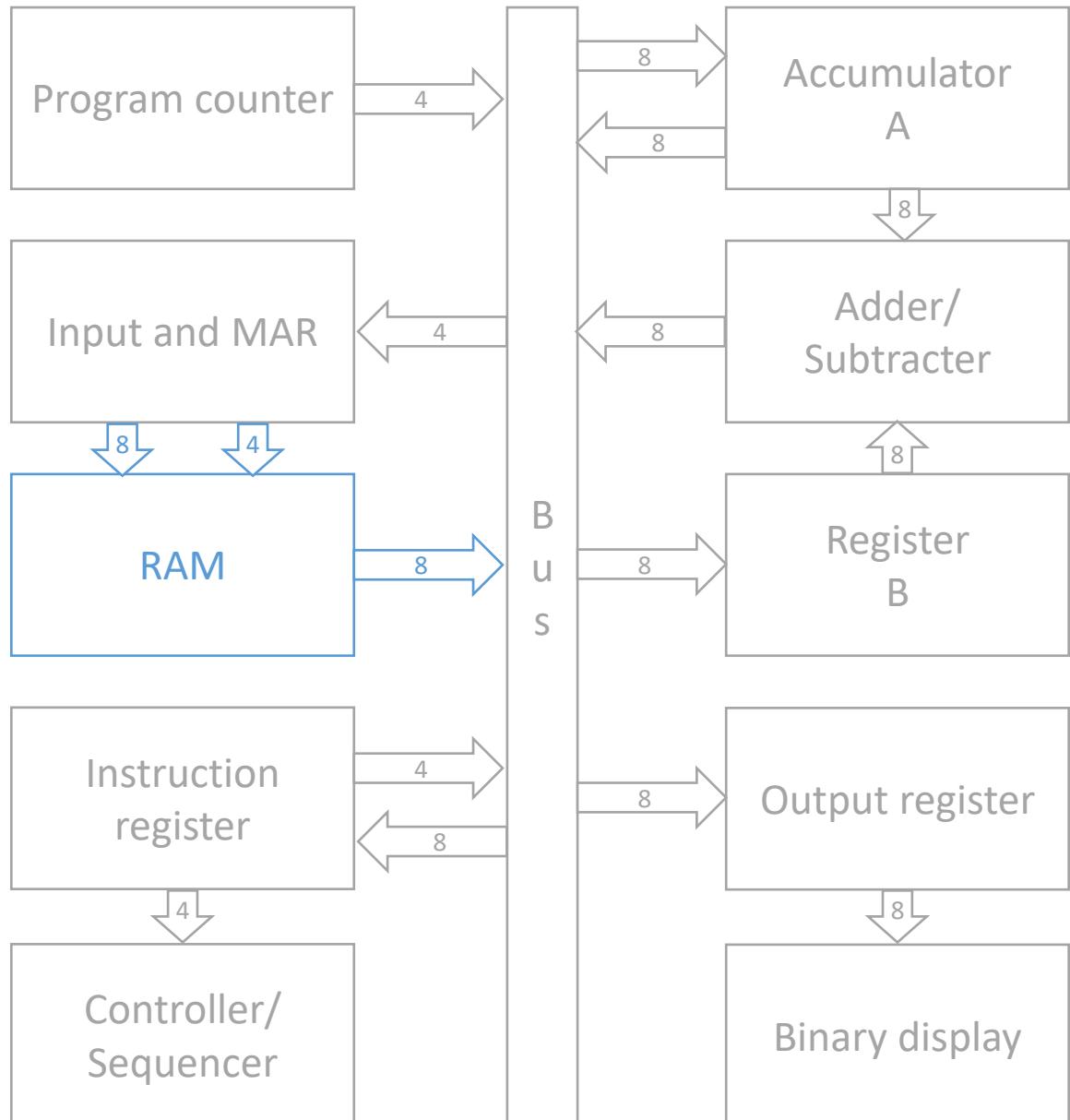
Architecture

- Input and MAR includes
 - Two switch registers (input registers) for address and data
 - A Memory Address Register (MAR)
- The input switch registers are used to write into the RAM the instruction and data words before a run
- MAR is used to latch the program counter address and to apply this address to the RAM



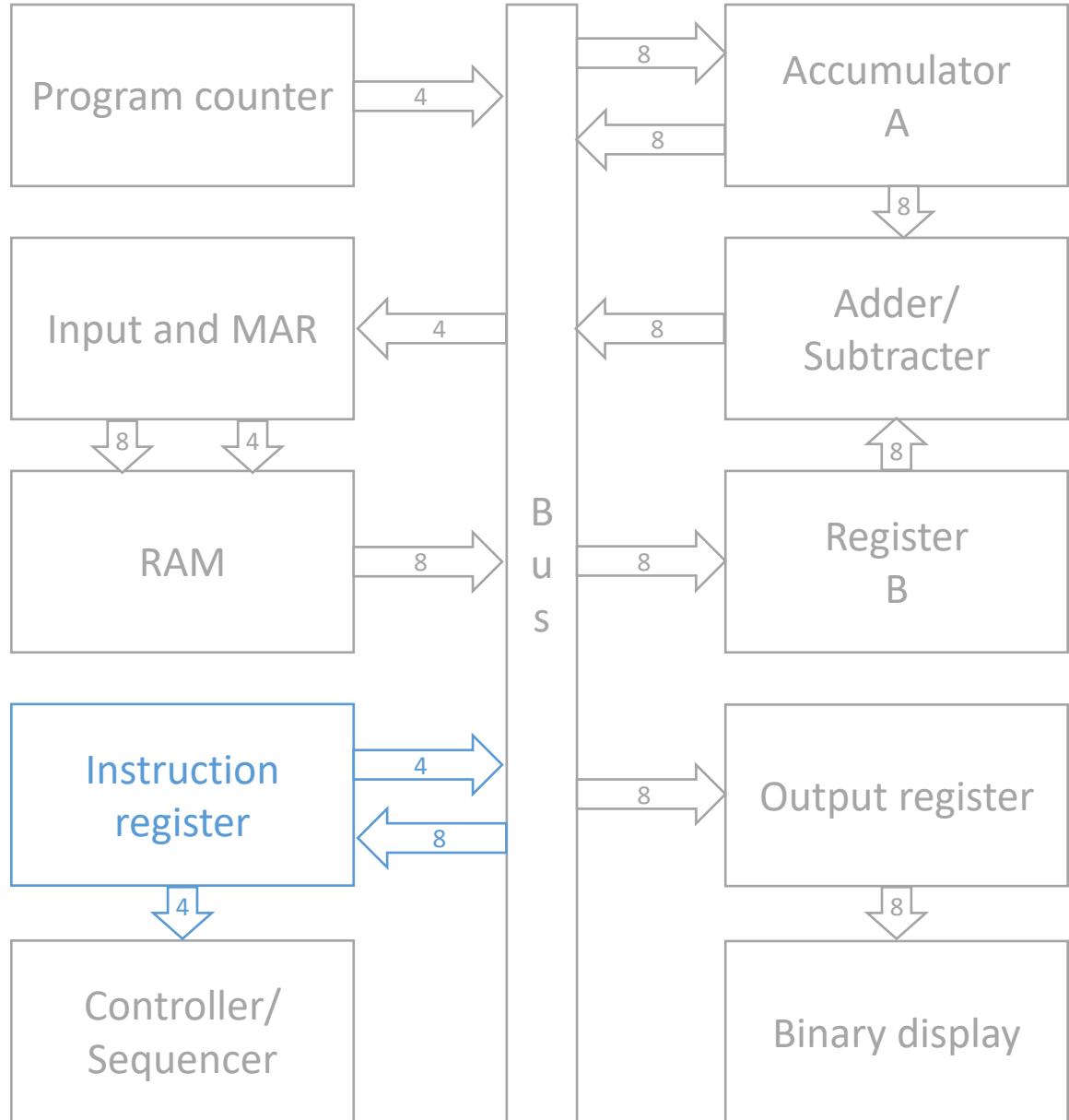
Architecture

- The memory block is a Random Access Memory (RAM) organized in 16x8-bit
- During a run RAM receive addresses from MAR and perform a read operation
- The instruction or data stored in RAM is placed in the bus to be used by some other block



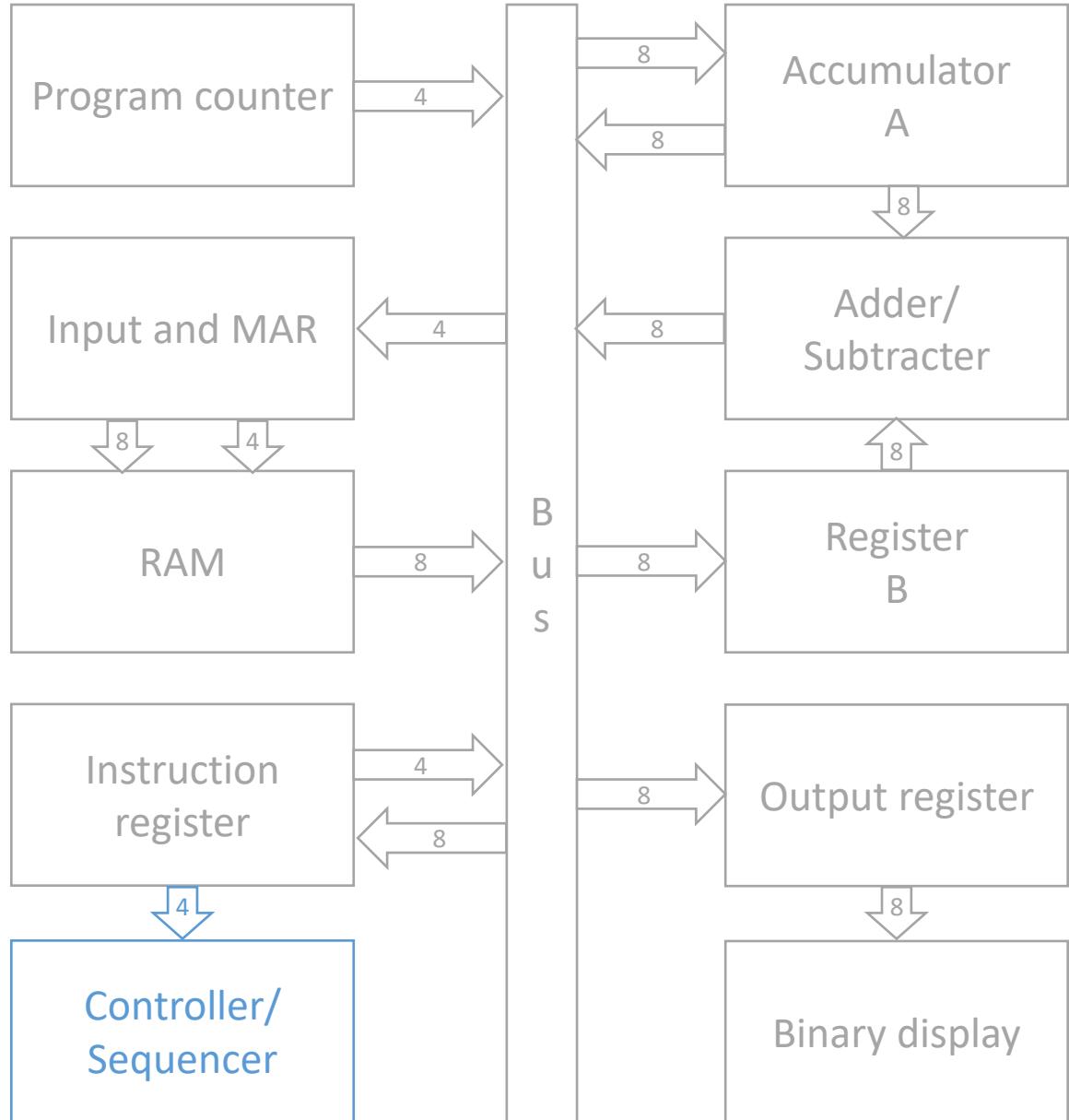
Architecture

- After reading an instruction from the memory, its content is placed on a register
- The content is split into two parts, one for the controller/sequencer and the other it is available on the bus



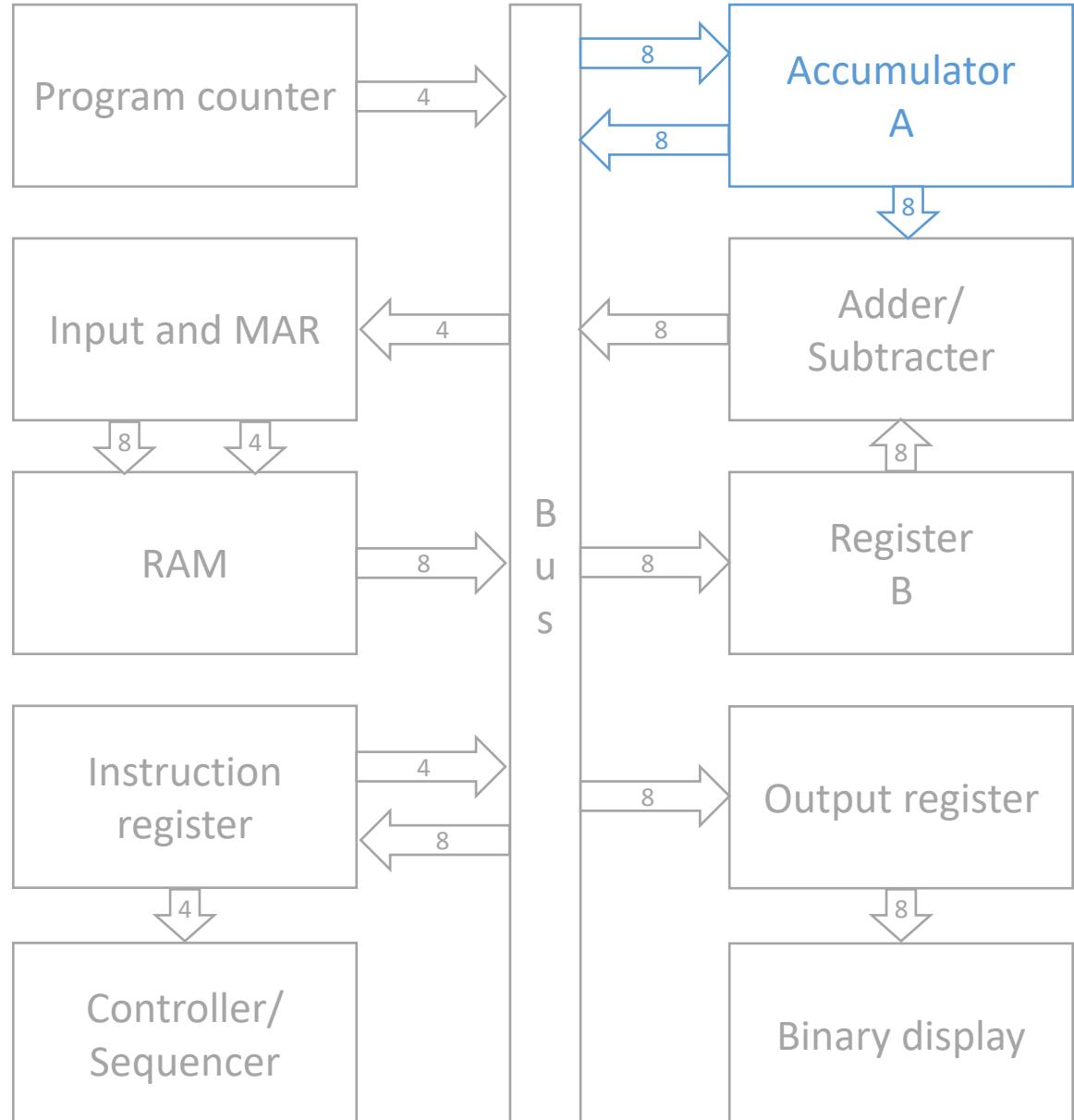
Architecture

- Controller/sequencer distribute the reset signal to the program counter and the instruction register at the beginning of a run
- It synchronize all the blocks providing the clock signal
- It provides a 12-bit word that is used for controlling all the blocks



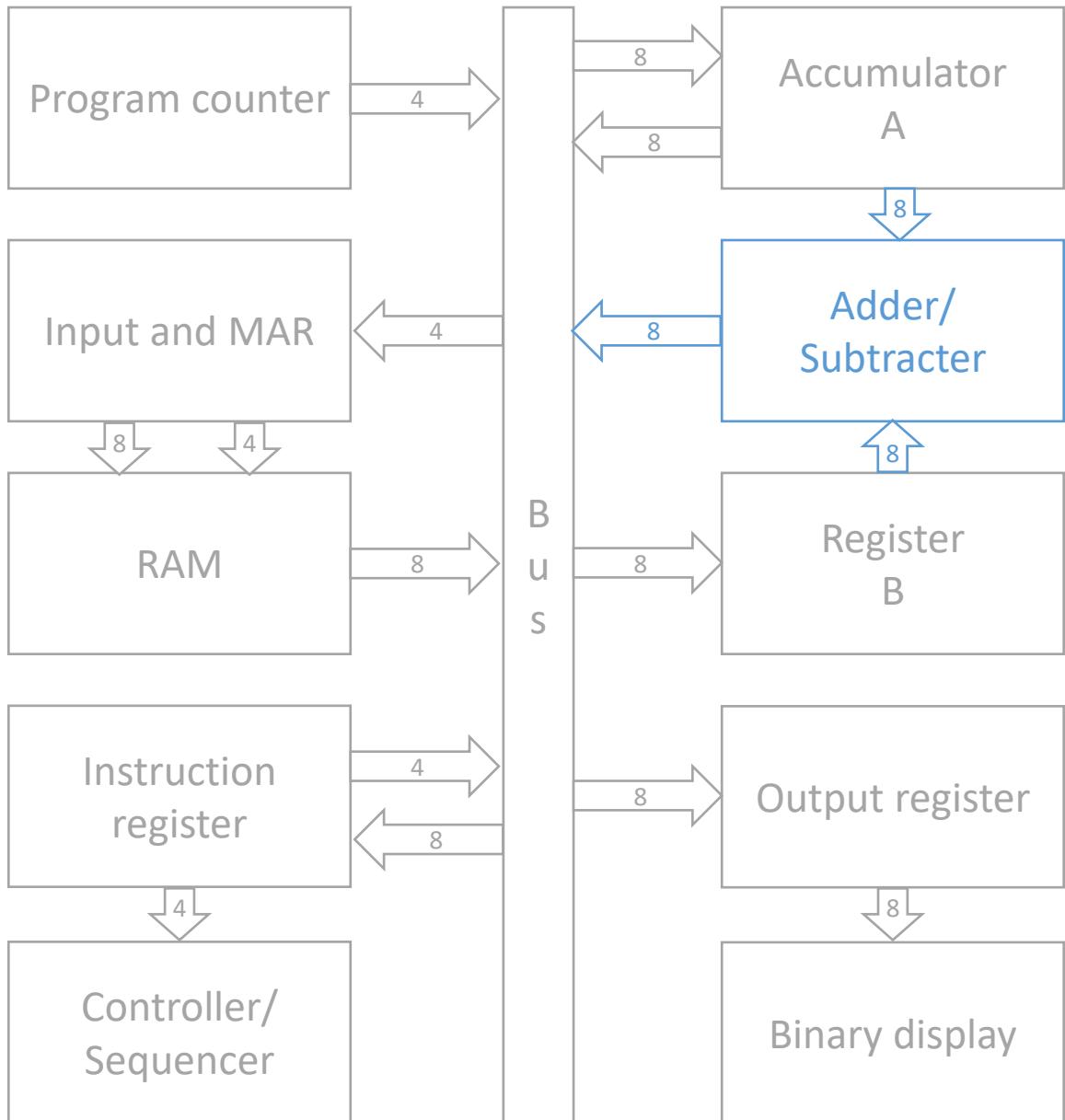
Architecture

- During a run intermediate answers are stored in the Accumulator
- It is a register that is placed on the bus and that have also a direct access to the adder/subtractor



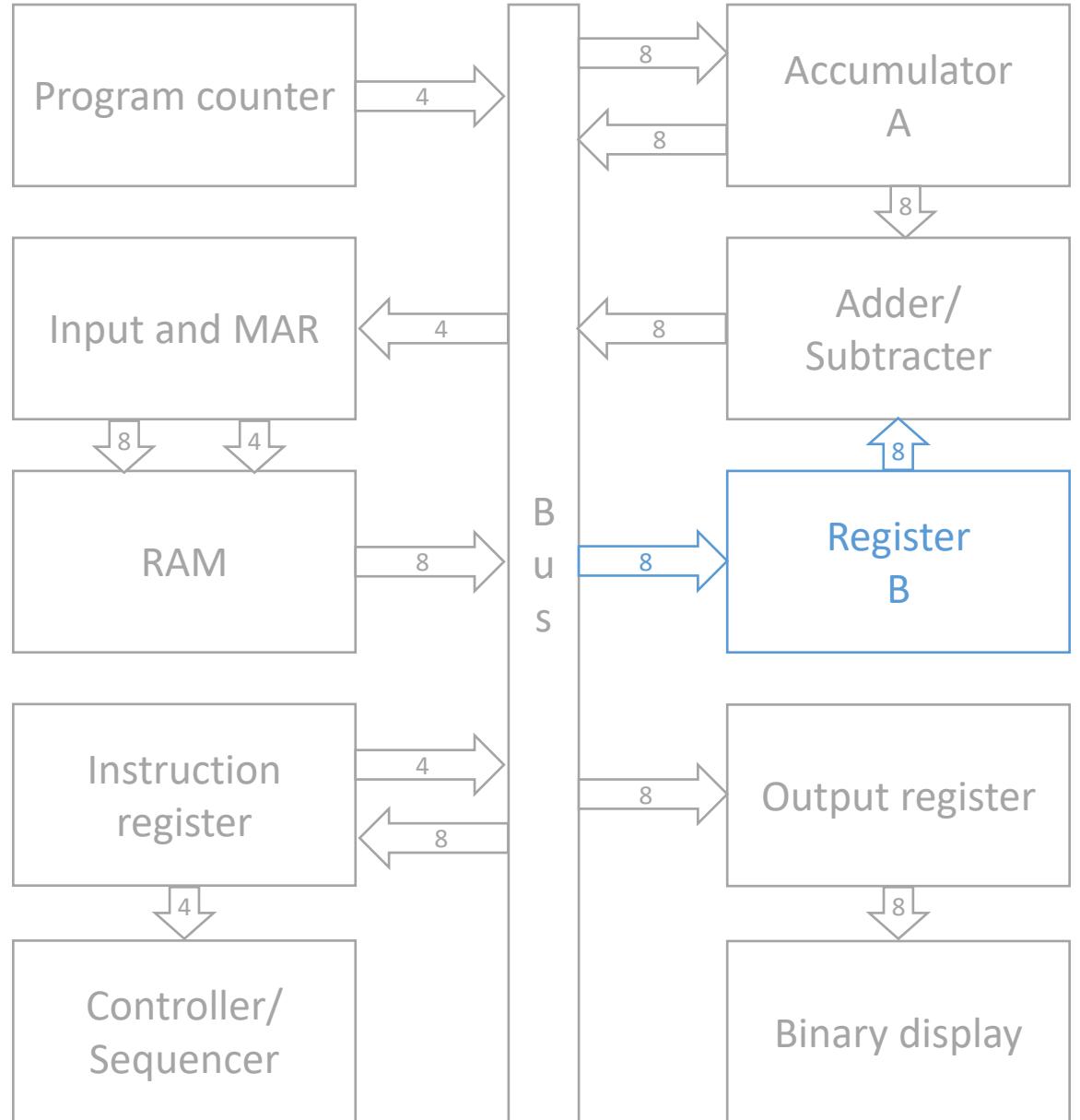
Architecture

- 2's complement adder/subtractor
- $S = A + B$ or $S = A + B'$
- It is asynchronous



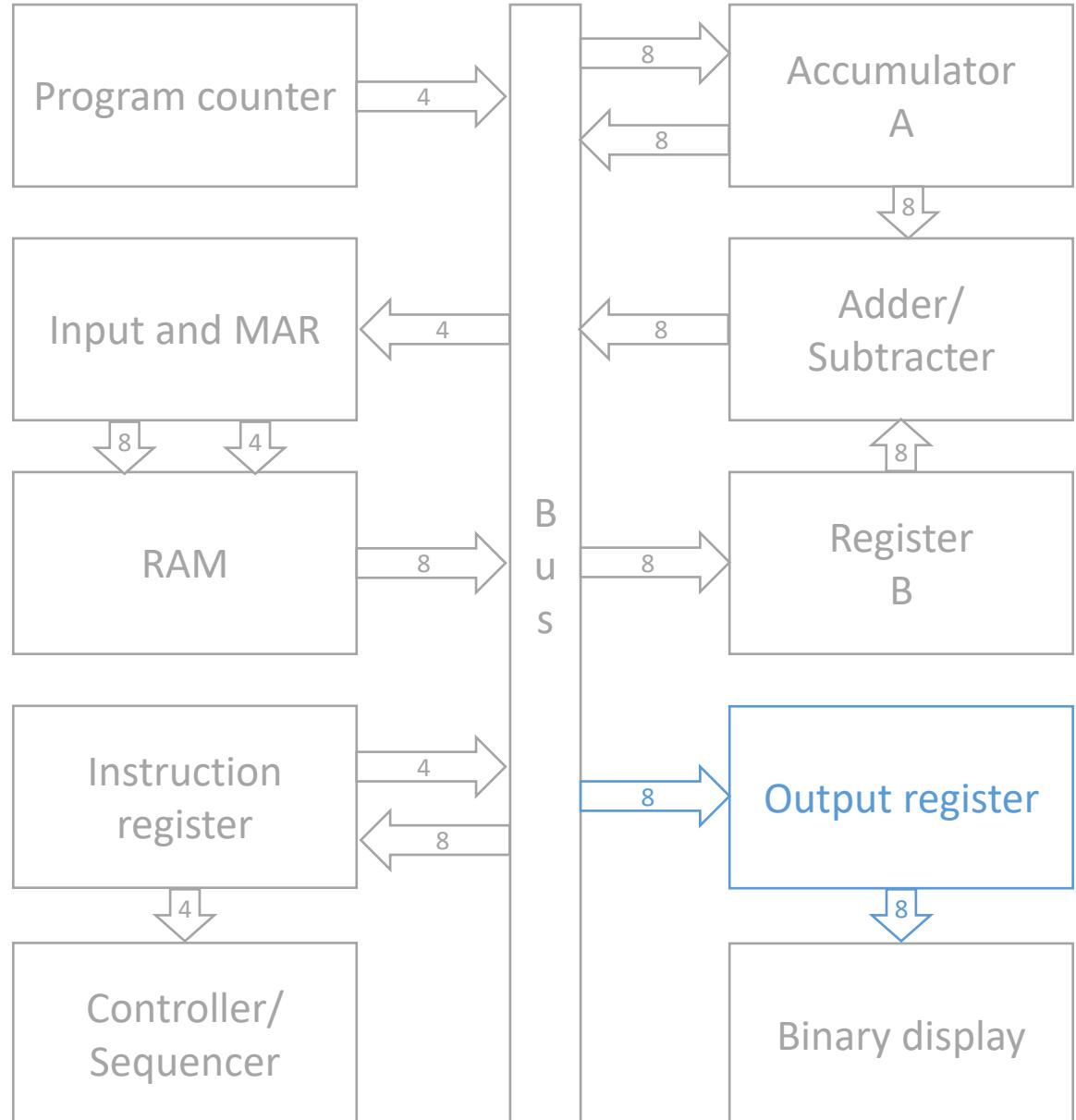
Architecture

- It is a register like the accumulator
- But it cannot be read from the bus



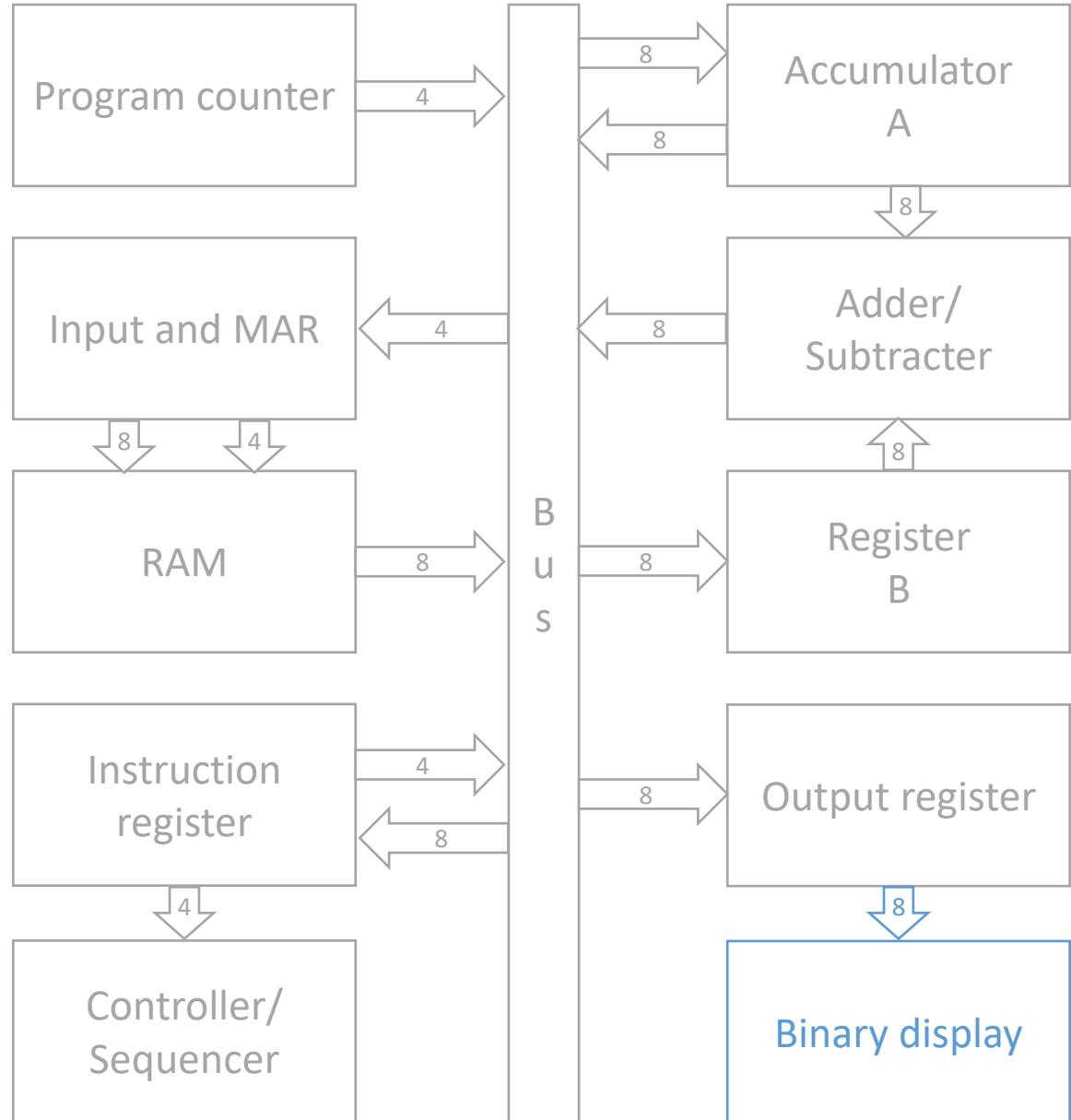
Architecture

- It is a register like B
- At the end of a run the accumulator contents is transferred to the output register
- It is used as a port to the outside word



Architecture

- The outcome of a run is provided to an interface that drives peripheral devices
- In SAP-1 a binary display is used to show the content of the output register
- It is simply built with 8 LEDs



Instruction set

- A computer needs to be programmed (damn it!)
- A program is a series of instructions that have to be loaded into memory before running
- For programming we need to know the instruction set
- SAP-1 has 5 instructions: LDA, ADD, SUB, OUT and HLT
- Abbreviated instructions are called mnemonics

LDA

- LoaD the Accumulator
- It loads the accumulator with the content of a memory address
- It has to include the memory address to be loaded to the accumulator
- For instance, “LDA 0x8” means to copy the content of memory at address 0x8 to the accumulator register

$$M_8 = 00000011 \xrightarrow{\text{LDA } 0x8} A = 00000011$$

ADD

- It adds the content of a memory address to the accumulator content
- It has to include the memory address to be added to the accumulator
- For instance, “ADD 0x9” means to copy the content of memory at address 0x9 to the register B and to load the output of the adder to the accumulator register

$$M_9 = 00000011 \quad A = 00000100 \xrightarrow{\text{ADD } 0x9} A = 00000111$$

SUB

- It subtract the content of the of a memory address from the accumulator content
- It has to include the memory address to be subtracted from the accumulator
- For instance, “SUB 0xC” means to copy the content of memory at address 0xC to the register B and to load the output of the subtracter to the accumulator register

$$M_C = 00000011 \quad A = 00000111 \xrightarrow{\text{SUB } 0xC} A = 00000100$$

OUT

- It copy the accumulator content to the output port
- It does not include any memory address

$$A = 00000100 \xrightarrow{\text{OUT}} OUT = 00000100$$

HLT

- It stands for HaLT
- It marks the end of the run
- It does not include any memory address
- It is mandatory to include it at the end of any program otherwise there would be meaningless answers caused by runaway processing

Operation code

- To encode the instructions in binary format we need the so called op code
- The op code is mapped in the upper nibble of the instruction while the lower is the operand
- An 8-bit instruction is split into two nibbles: op code and operand

Mnemonics	Op Code
LDA	0000
ADD	0001
SUB	0010
OUT	1110
HLT	1111

- A program written with mnemonics is expressed in Assembly language (source program)
- A program written with op code and operand is expressed in Machine language (object program)
- In SAP-1 the operator translates the source into object program when programming the switch register

Source program example

Address	Data
0x0	LDA 0x8
0x1	ADD 0x9
0x2	ADD 0xA
0x3	SUB 0xB
0x4	OUT
0x5	HLT
0x6	0xFF
0x7	0xFF
0x8	0x4
0x9	0x6
0xA	0x1
0xB	0x2
0xC	0xFF
0xD	0xFF
0xE	0xFF
0xF	0xFF

$$M_8 = 00000100 \xrightarrow{\text{LDA } 0x8} A = 00000100$$

$$M_9 = 00000110 \quad A = 00000100 \xrightarrow{\text{ADD } 0x9} A = 00001010$$

$$M_{10} = 00000001 \quad A = 00001010 \xrightarrow{\text{ADD } 0xA} A = 00001011$$

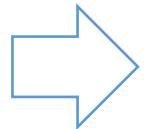
$$M_{11} = 00000010 \quad A = 00001011 \xrightarrow{\text{SUB } 0xB} A = 00001001$$

$$A = 00001001 \xrightarrow{\text{OUT}} OUT = 00001001$$

HLT

Object program example

Address	Data
0x0	LDA 0x8
0x1	ADD 0x9
0x2	ADD 0xA
0x3	SUB 0xB
0x4	OUT
0x5	HLT
0x6	0xFF
0x7	0xFF
0x8	0x4
0x9	0x6
0xA	0x1
0xB	0x2
0xC	0xFF
0xD	0xFF
0xE	0xFF
0xF	0xFF



Address	Data
0x0	0x08
0x1	0x19
0x2	0x1A
0x3	0x2B
0x4	0xE8
0x5	0xF8
0x6	0XX
0x7	0XX
0x8	0x04
0x9	0x06
0xA	0x01
0xB	0x02
0xC	0XX
0xD	0XX
0xE	0XX
0xF	0XX

Timing states

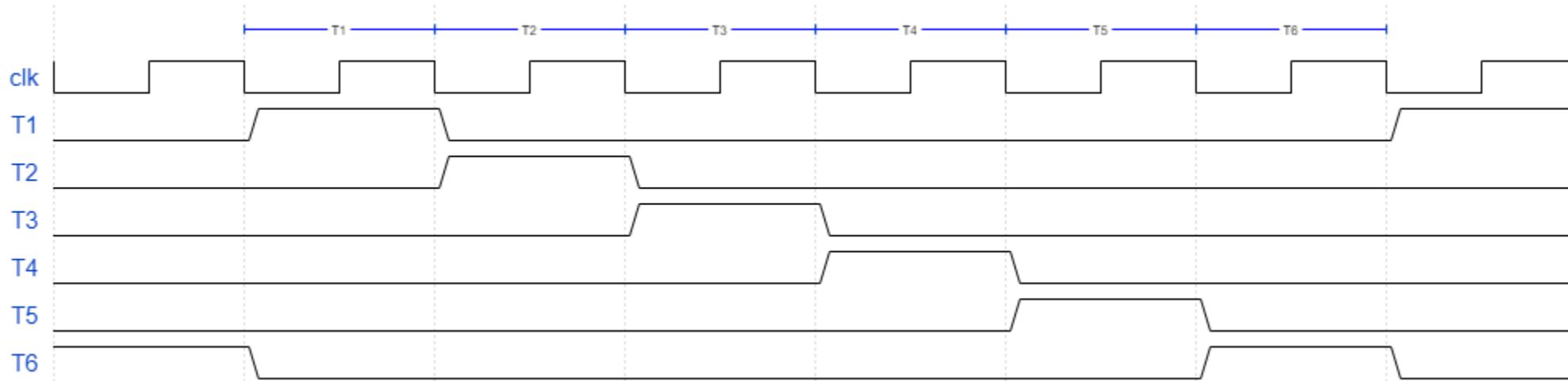
- The control unit generates the control words that fetch and execute each instruction
- The computer pass through different timing states during fetch and execute
- SAP-1 uses a 6-bit ring counter (circular shift register) to generate the timing states

$$T = T_6 T_5 T_4 T_3 T_2 T_1$$

- The ring counter start from

$$T = 000001$$

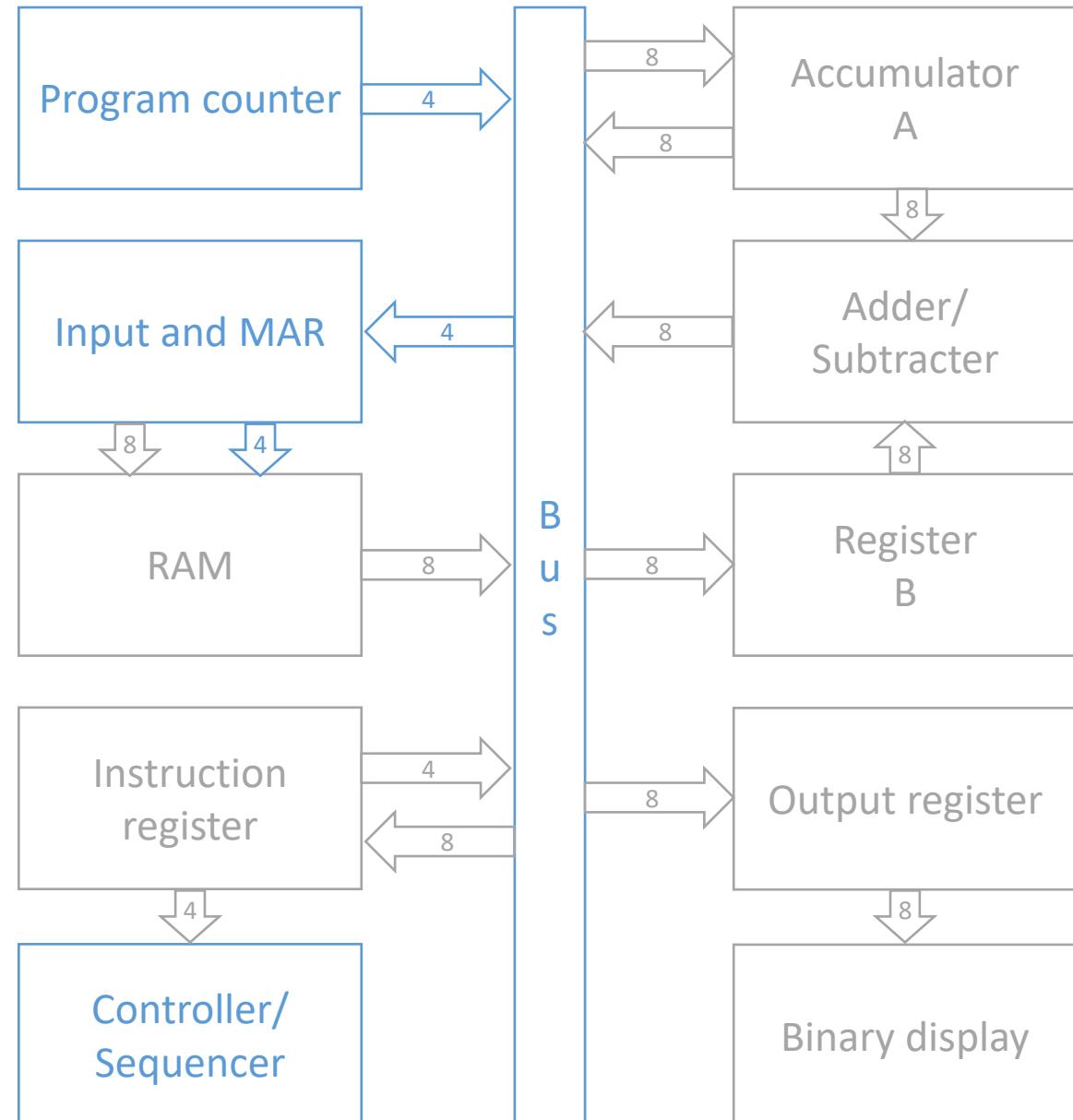
Timing states



- Each timing state starts and ends between two negative clock edges
- During the first timing state the first bit remains high, during the second the second bit remains high and so on
- The positive clock edge occurs in the middle of each timing states

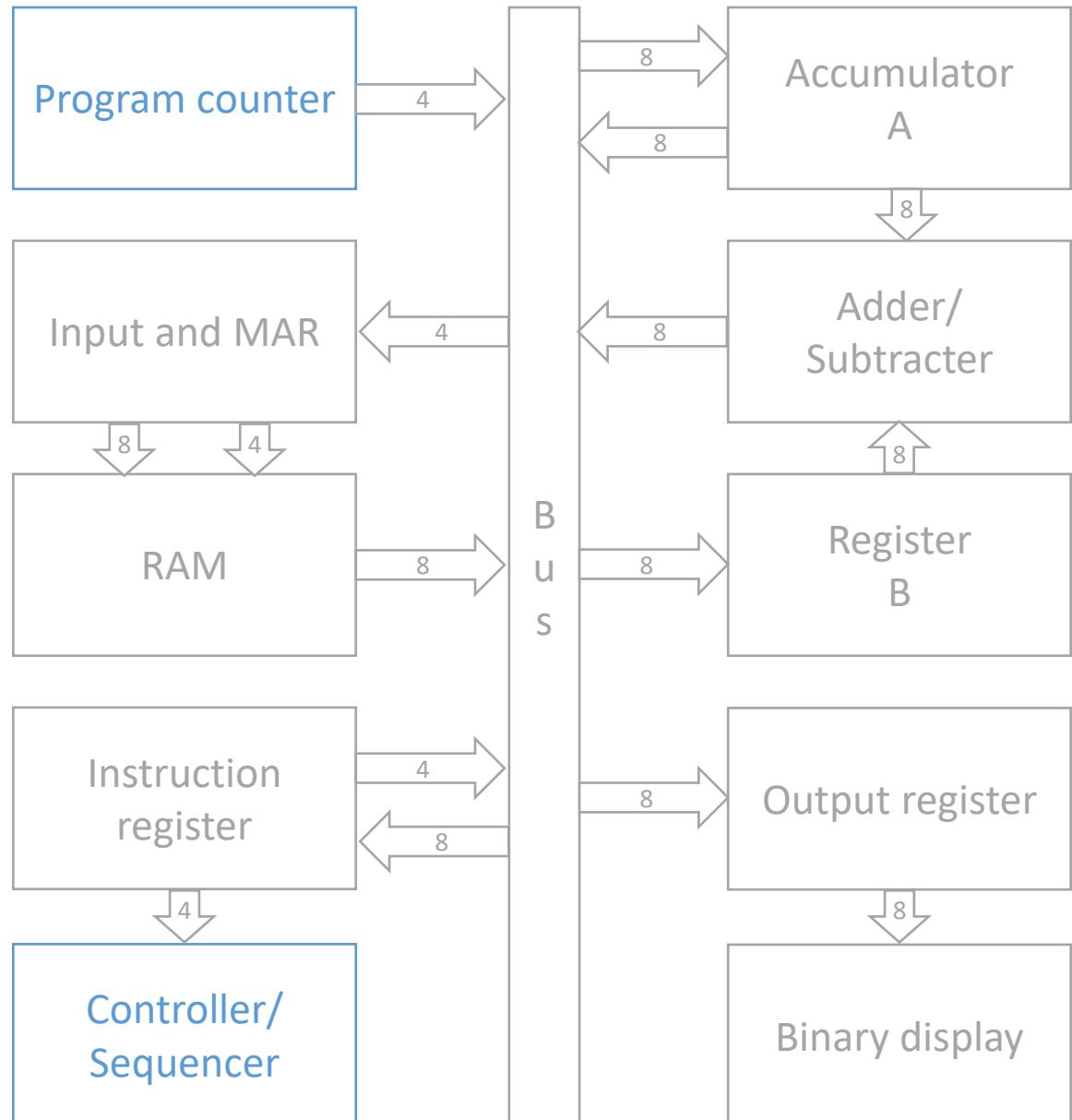
T_1 : Address state

- The address in the program counter is transferred to the MAR



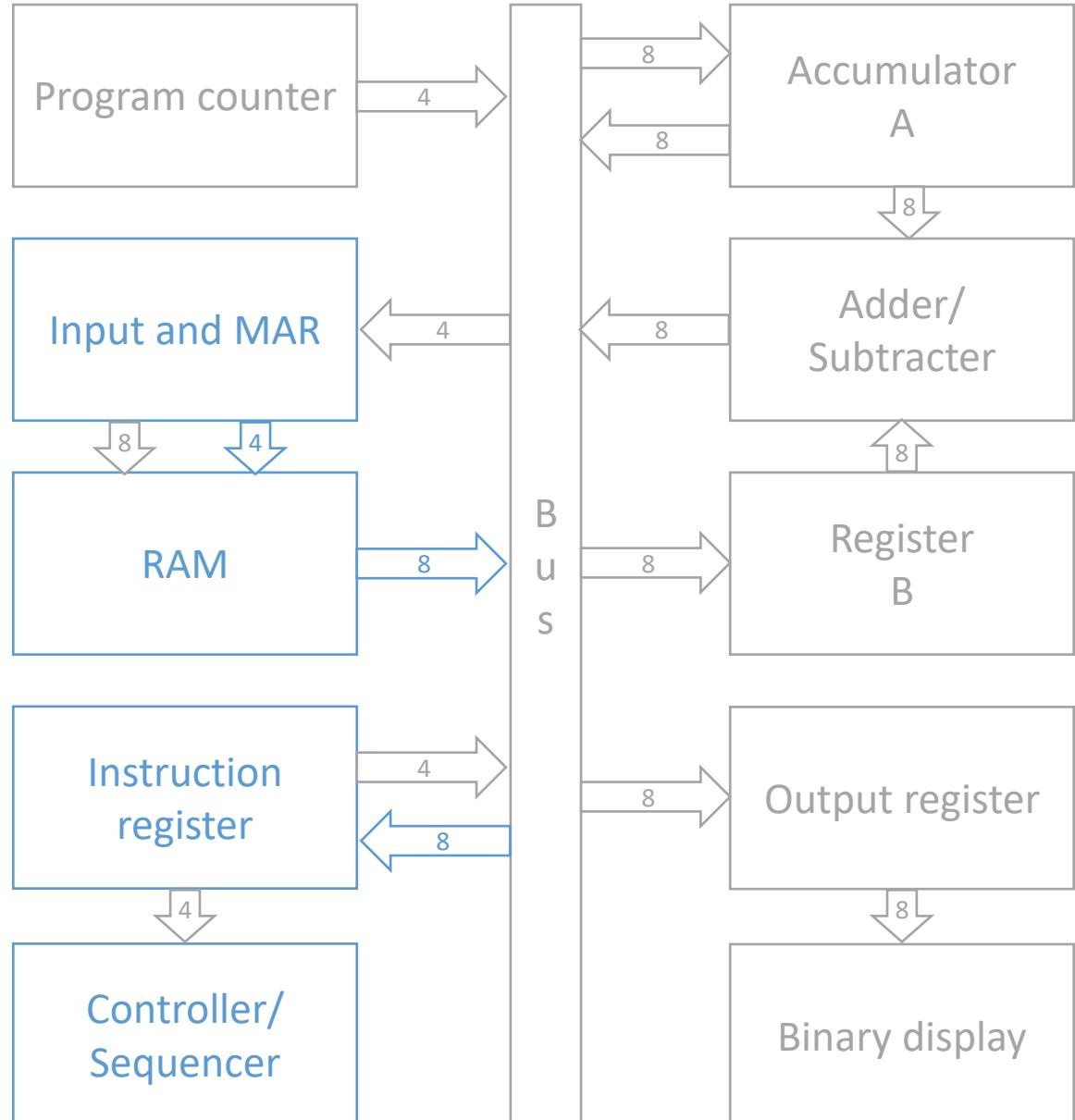
T_2 : Increment state

- The program counter is incremented



T_3 : Memory state

- The addressed RAM instruction is transferred from the memory to instruction register

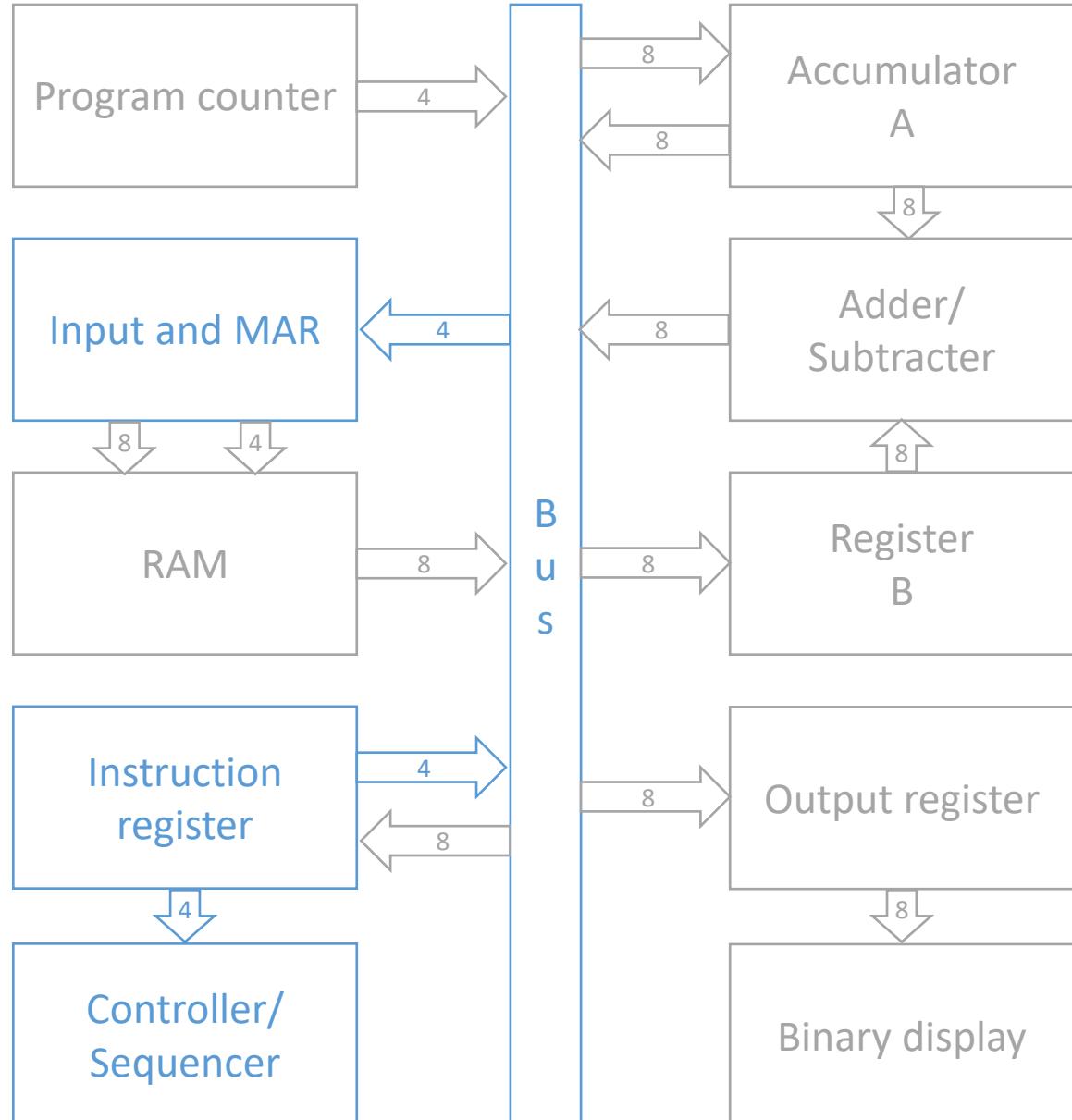


Timing states

- The first three timing states are called fetch cycle
- The second three timing states are called execution cycle
- The registers transfer during execution cycle are instruction dependent
- There is a different control routine for each instruction

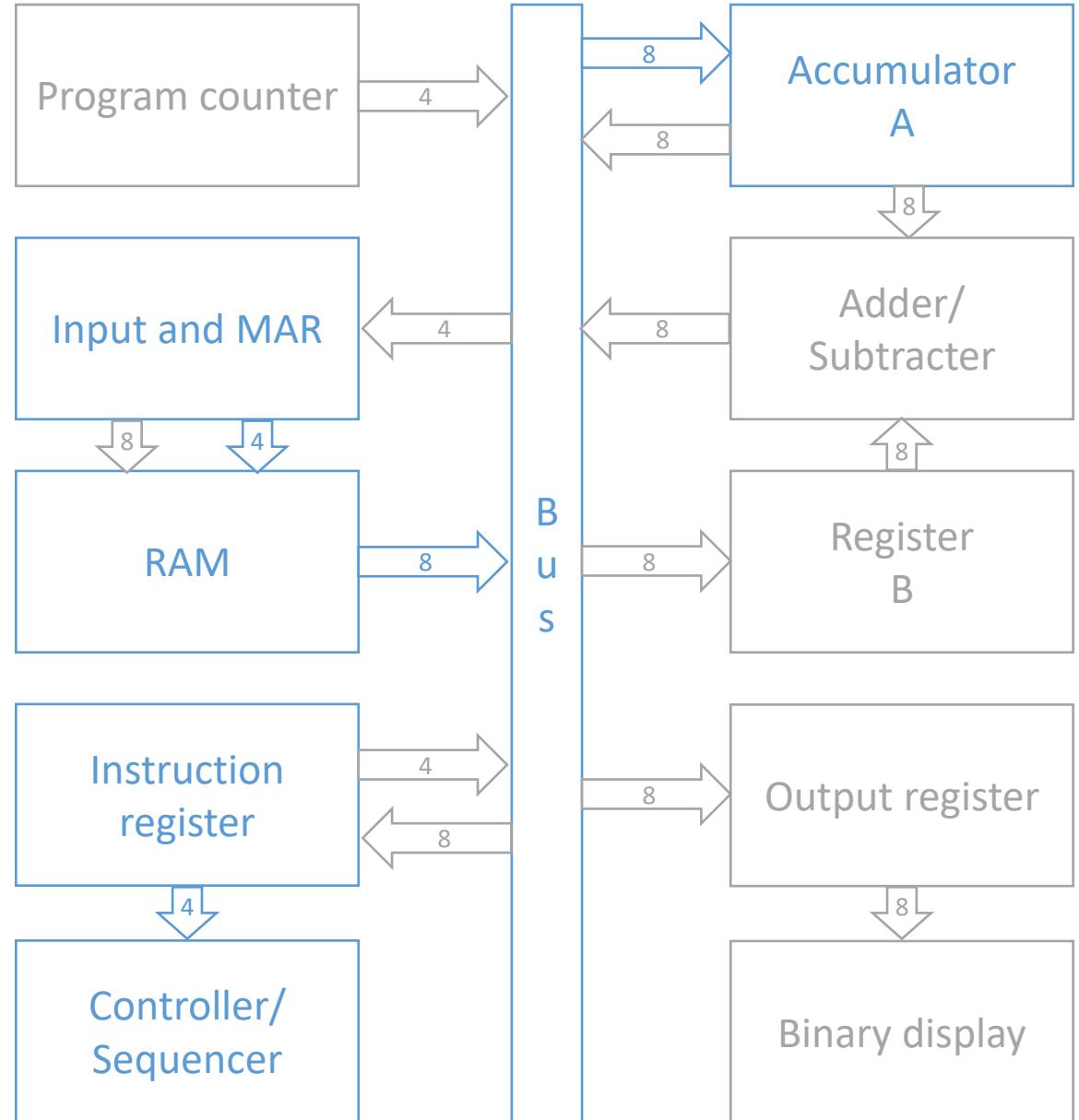
T_4 for LDA routine

- The first nibble of the instruction register (the instruction field) goes to the Controller/Sequencer
- The second nibble (the address field) is loaded into the MAR



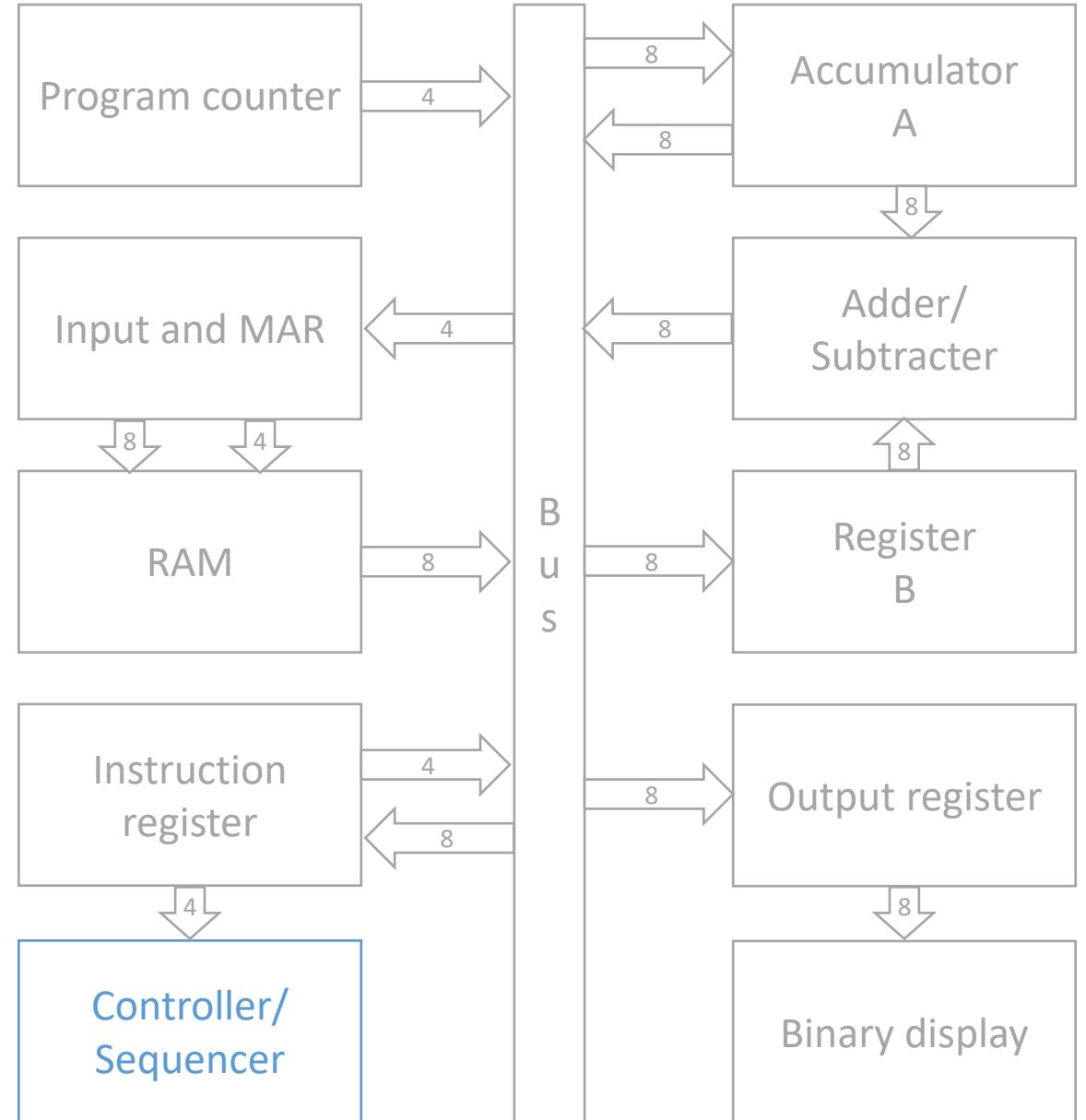
T_5 for LDA routine

- The addressed data word in RAM is copied to the accumulator



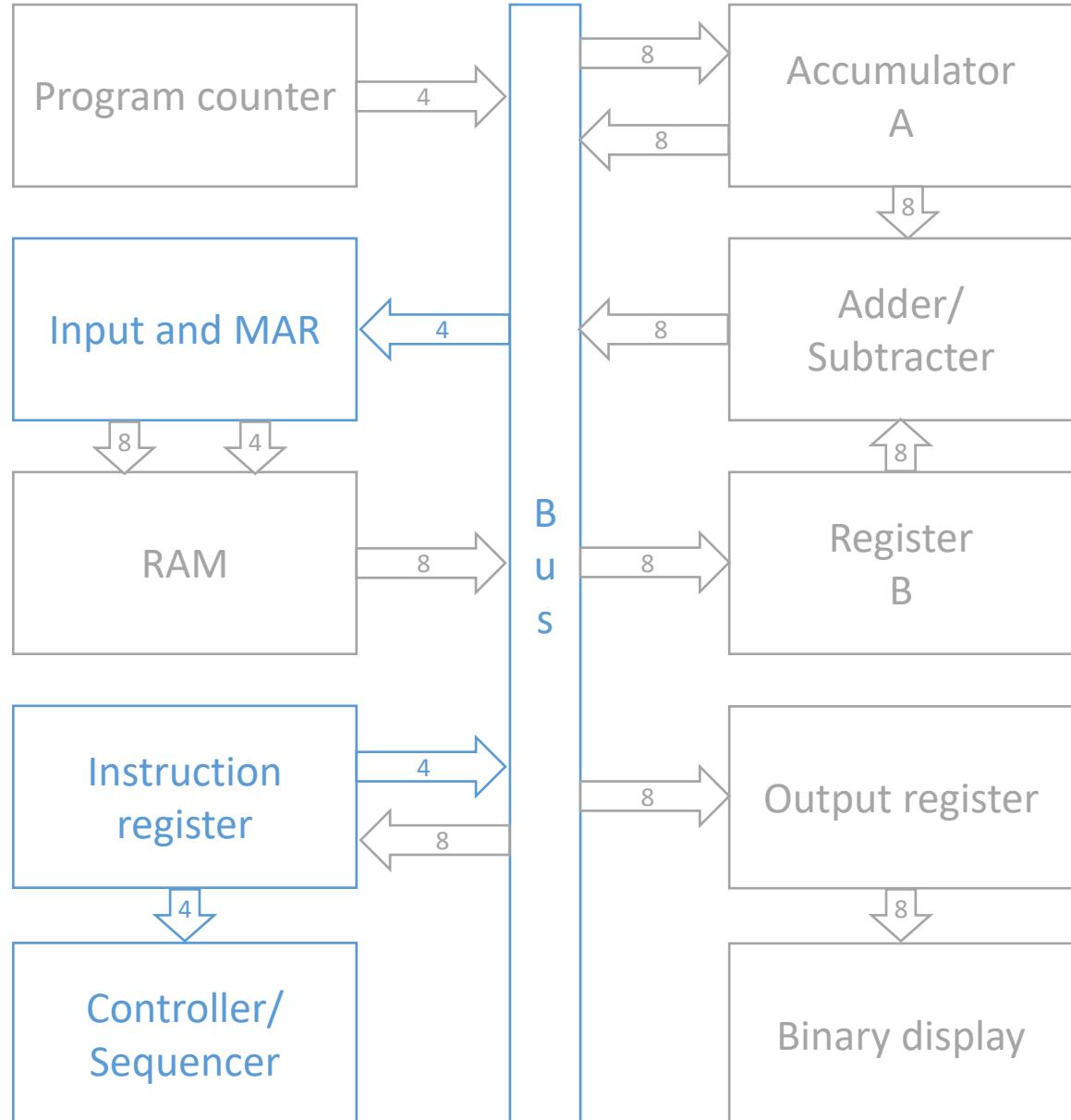
T_6 for LDA routine

- It is a nop (no-operation) state
- All registers are inactive



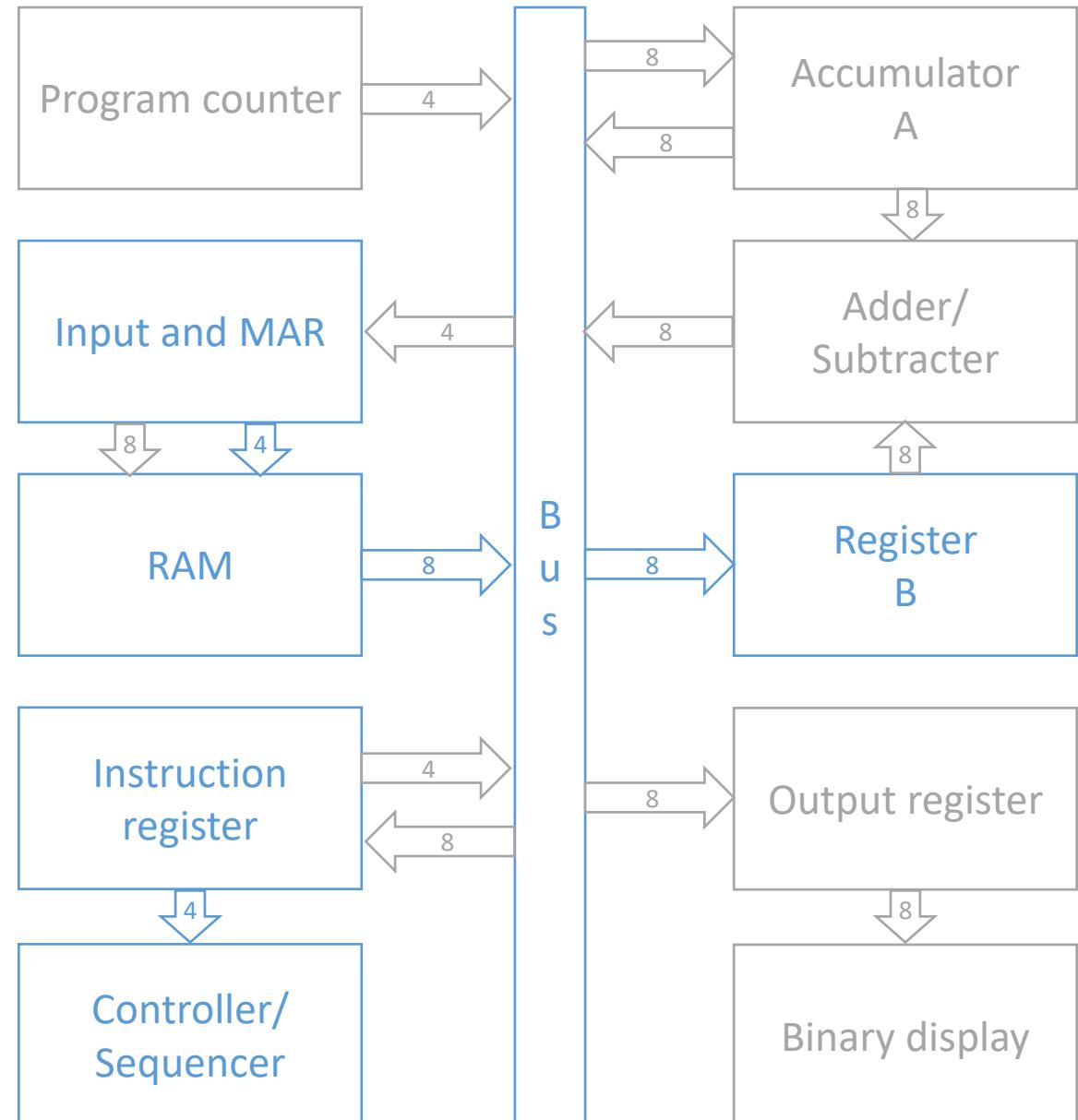
T_4 for ADD routine

- The first nibble of the instruction register (the instruction field) goes to the Controller/Sequencer
- The second nibble (the address field) is loaded into the MAR



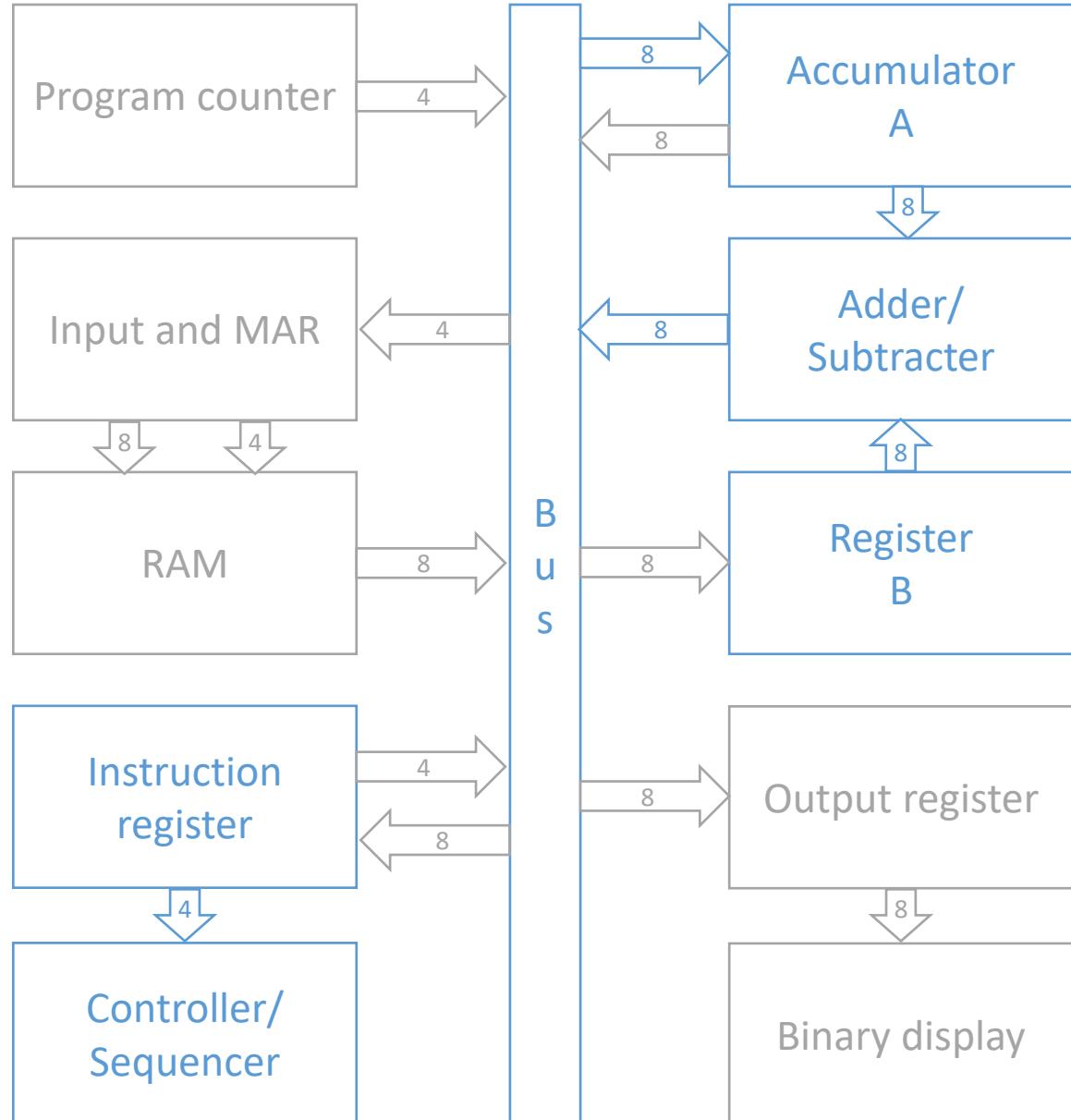
T_5 for ADD routine

- The addressed data word in RAM is copied to register B



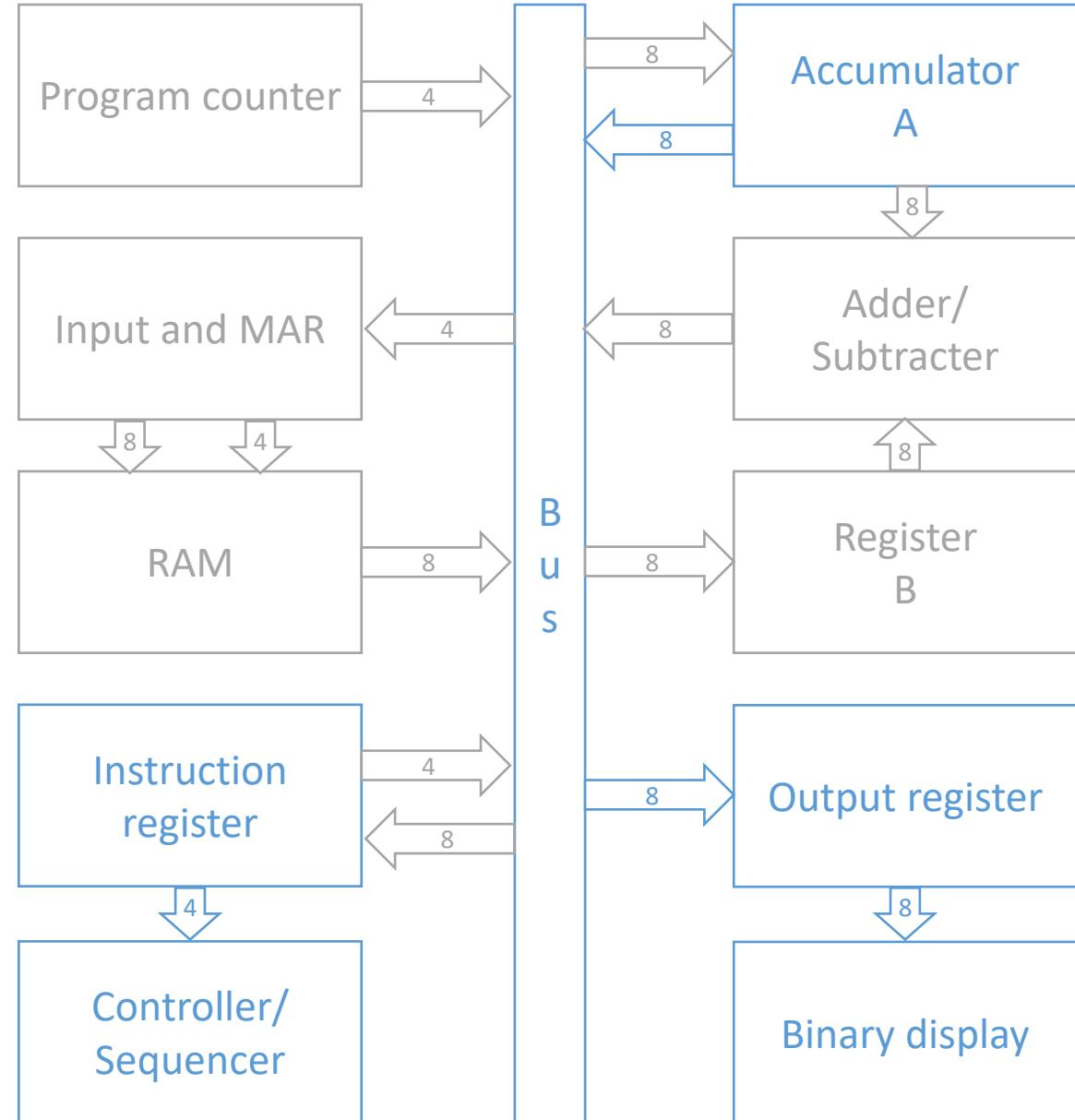
T_6 for ADD routine

- The output of the Adder/Subtractor is loaded to the accumulator
- Set up time and propagation delay prevent racing of the accumulator
- SUB control routine is identical except for the configuration of the Adder/Subtractor



T_4 for OUT routine

- The first nibble of the instruction register (the instruction field) goes to the Controller/Sequencer
- The accumulator content is copied to the output register
- The binary display shows its content
- T_5 and T_6 for the OUT control routine are nop

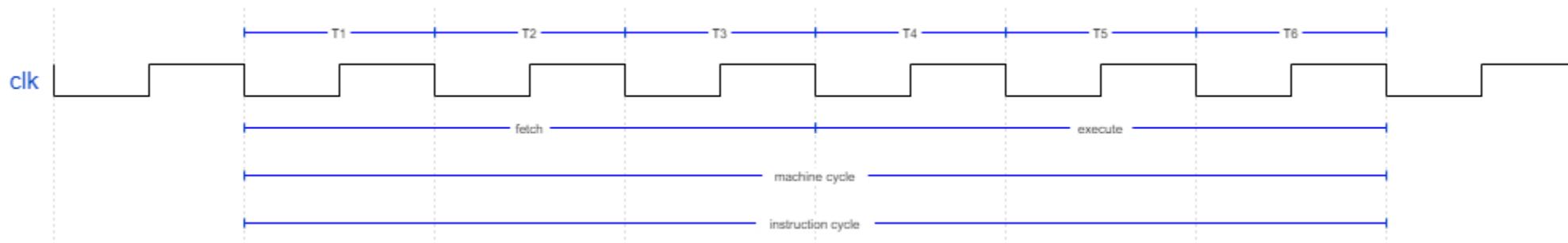


HLT routine

- HLT does not require a control routine because no registers are involved in its execution
- The Controller/Sequencer stops the computer by turning off the clock

Machine/Instruction cycle

- SAP-1 has six timing states (three fetch and three execute) that are called a machine cycle
- In SAP-1 all the instruction can be executed within a machine cycle -> the instruction cycle correspond to the machine cycle
- That is not always the case: the instruction cycle can use more than one machine cycle



Microprogram

- Controller/Sequencer send out one control word every state T , called microinstructions because it is part of an instruction
- With this nomenclature instructions are called macroinstructions
- In SAP-1 each macroinstruction is made by three microinstructions
- The fetch routine is also made by three microinstructions

Exercises

- Write an SAP-1 program that perform and display the result of the arithmetic operation $4+5-3$
- If the clock frequency of SAP-1 is 4 MHz, how long does it take to run the previous program?

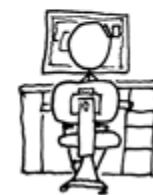
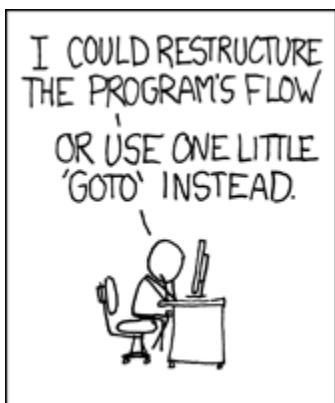
MANAGEMENT AND ANALYSIS OF PHYSICS DATASET (MOD. A)

SAP-2 and SAP-3 computer

SAP-2

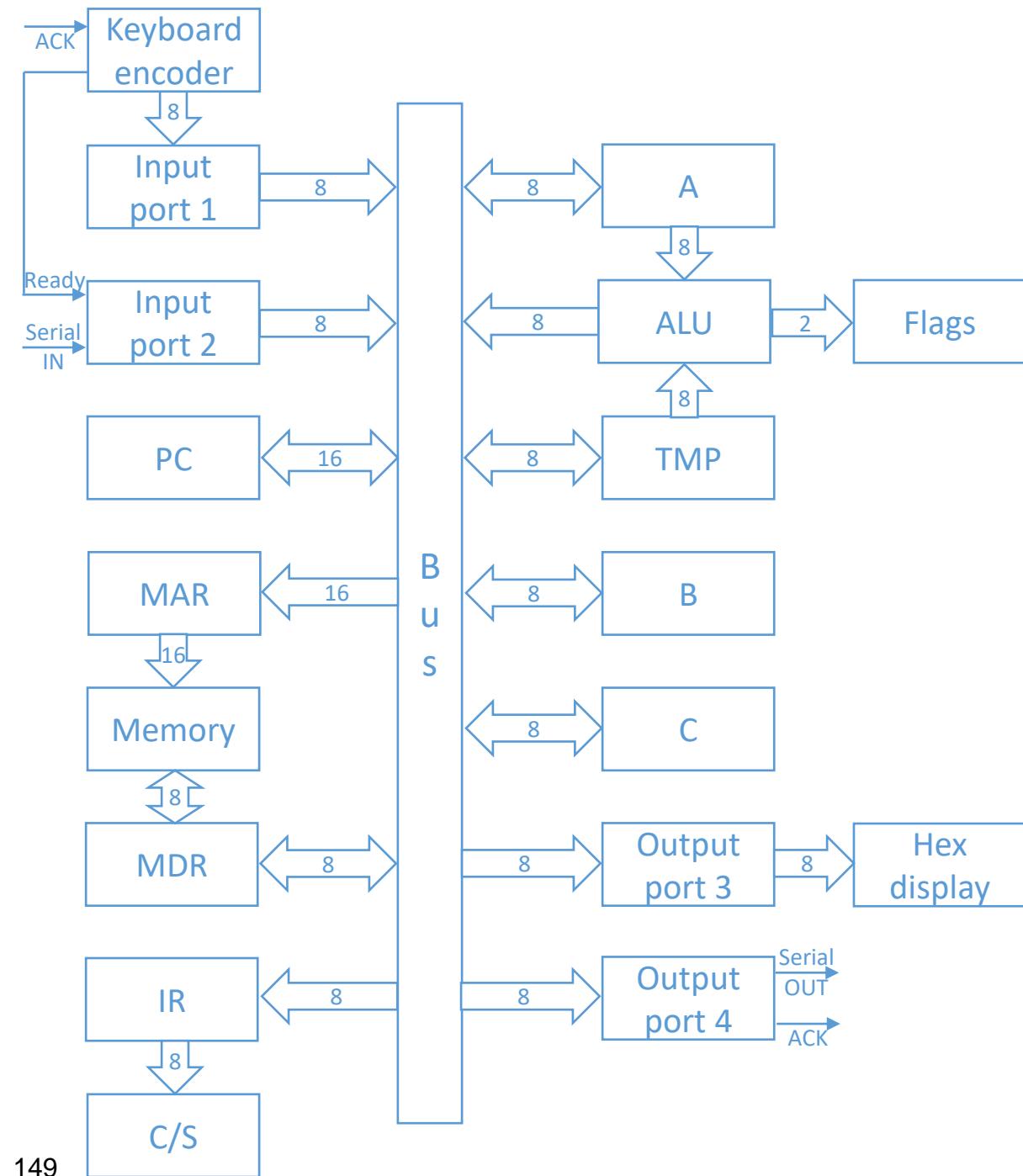
SAP-2

- SAP-1 is a computer
 - Store program and data before calculation
 - Automatically carry out program instructions
- SAP-2 is an evolution towards modern computer
 - Jump instructions give much more computing power



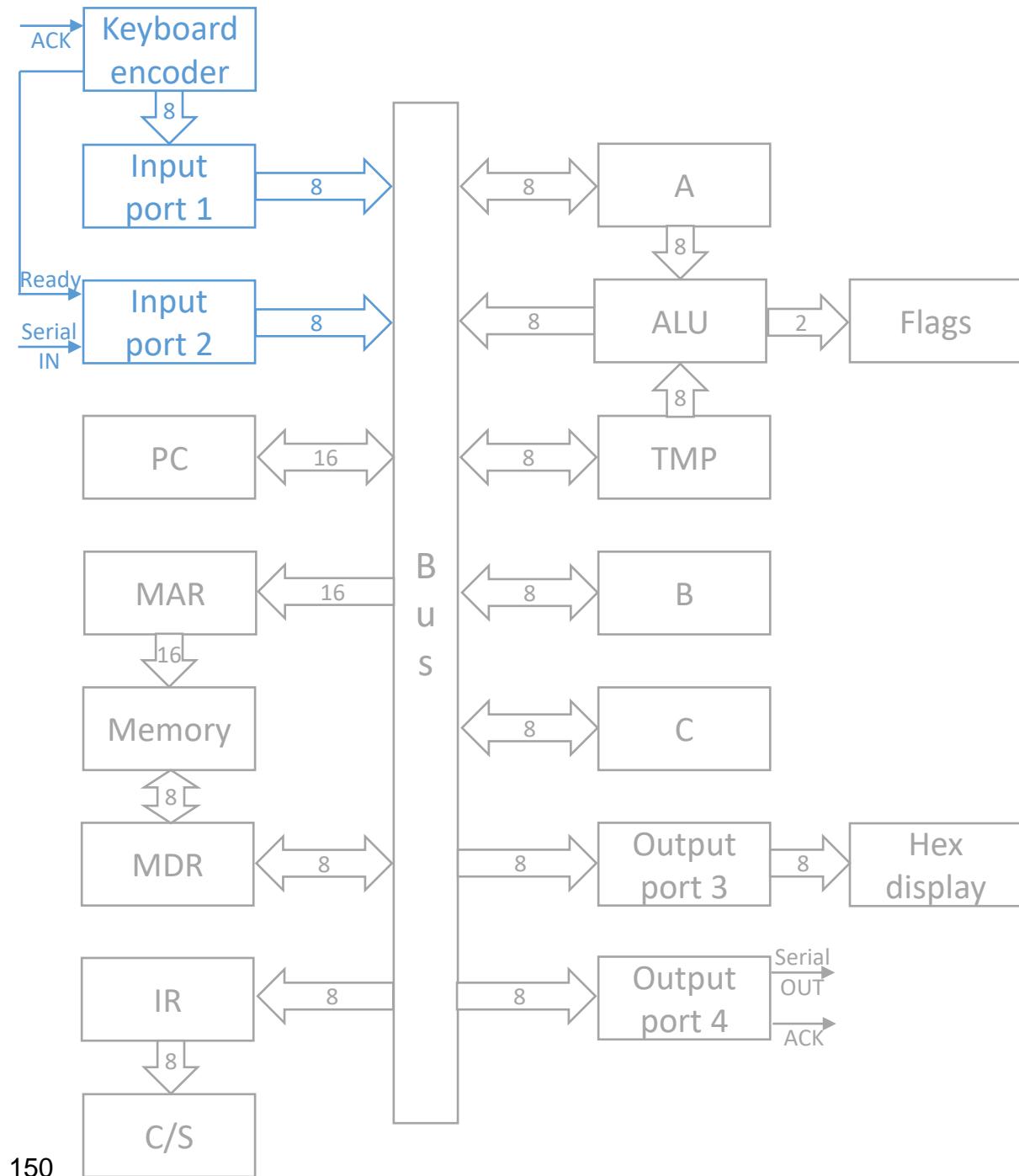
Architecture

- Simplified schema without clock, reset and controls
- All the register outputs to the bus are three-state
- Adopt bidirectional registers



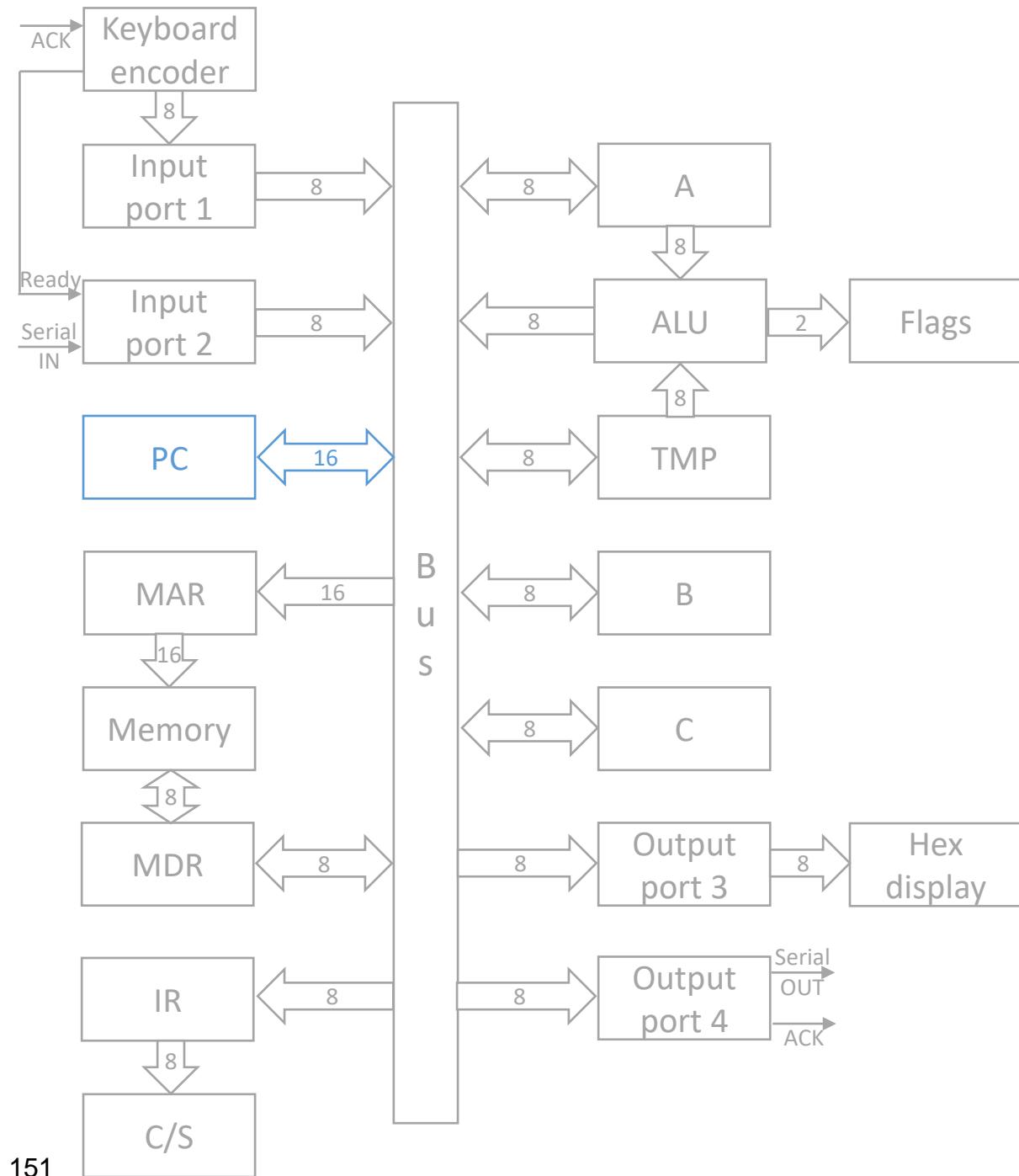
Architecture

- There are two input ports
- A hexadecimal keyboard encoder allows to enter hex instructions and data through port 1
- A serial input is connected to port 2



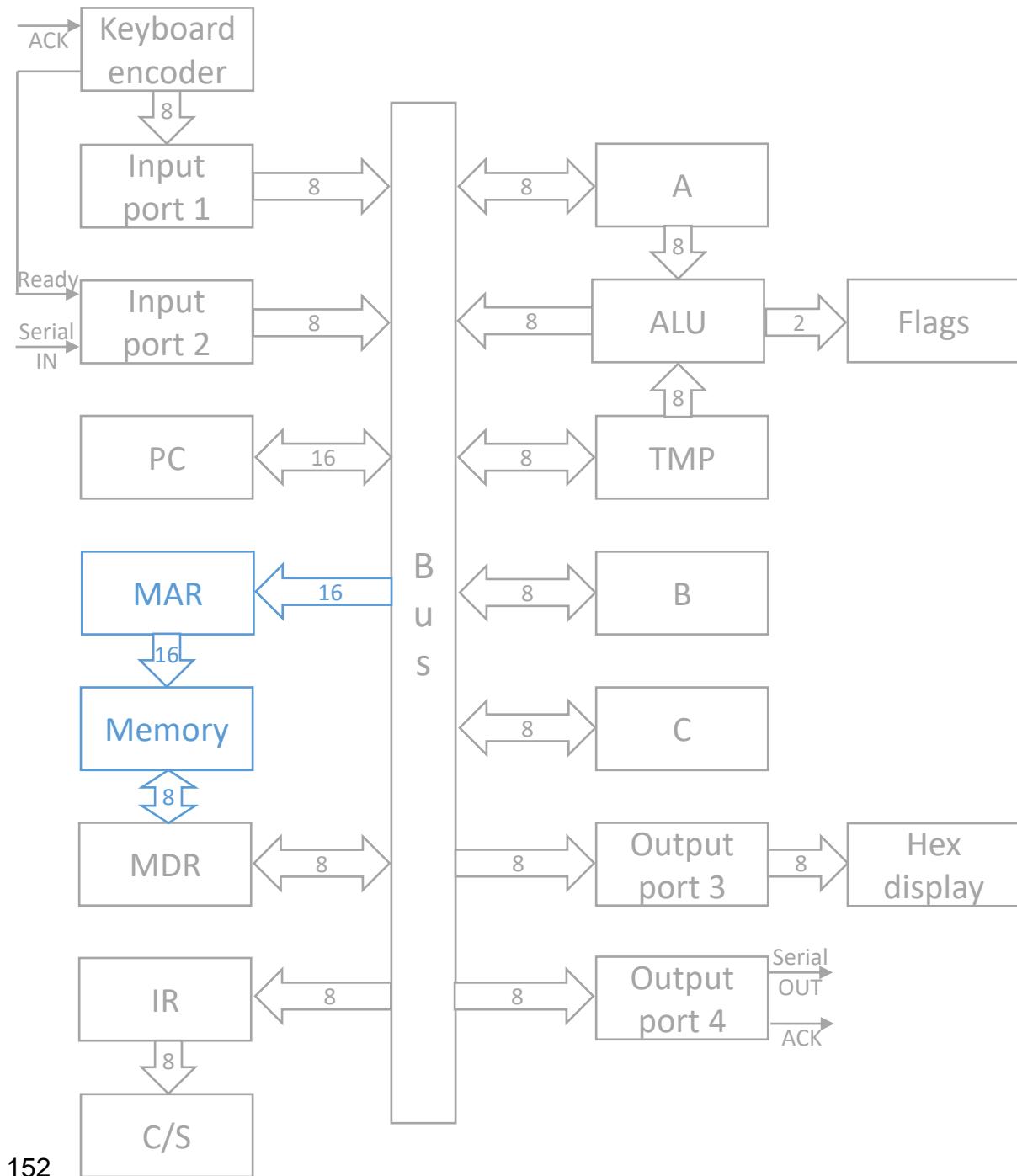
Architecture

- Program counter has the same role has for SAP-1 but it is 16-bit wide
- It counts from 0x0000 to 0xFFFF
- Before a run it is reset to 0x0000
- The connection to the bus is bidirectional



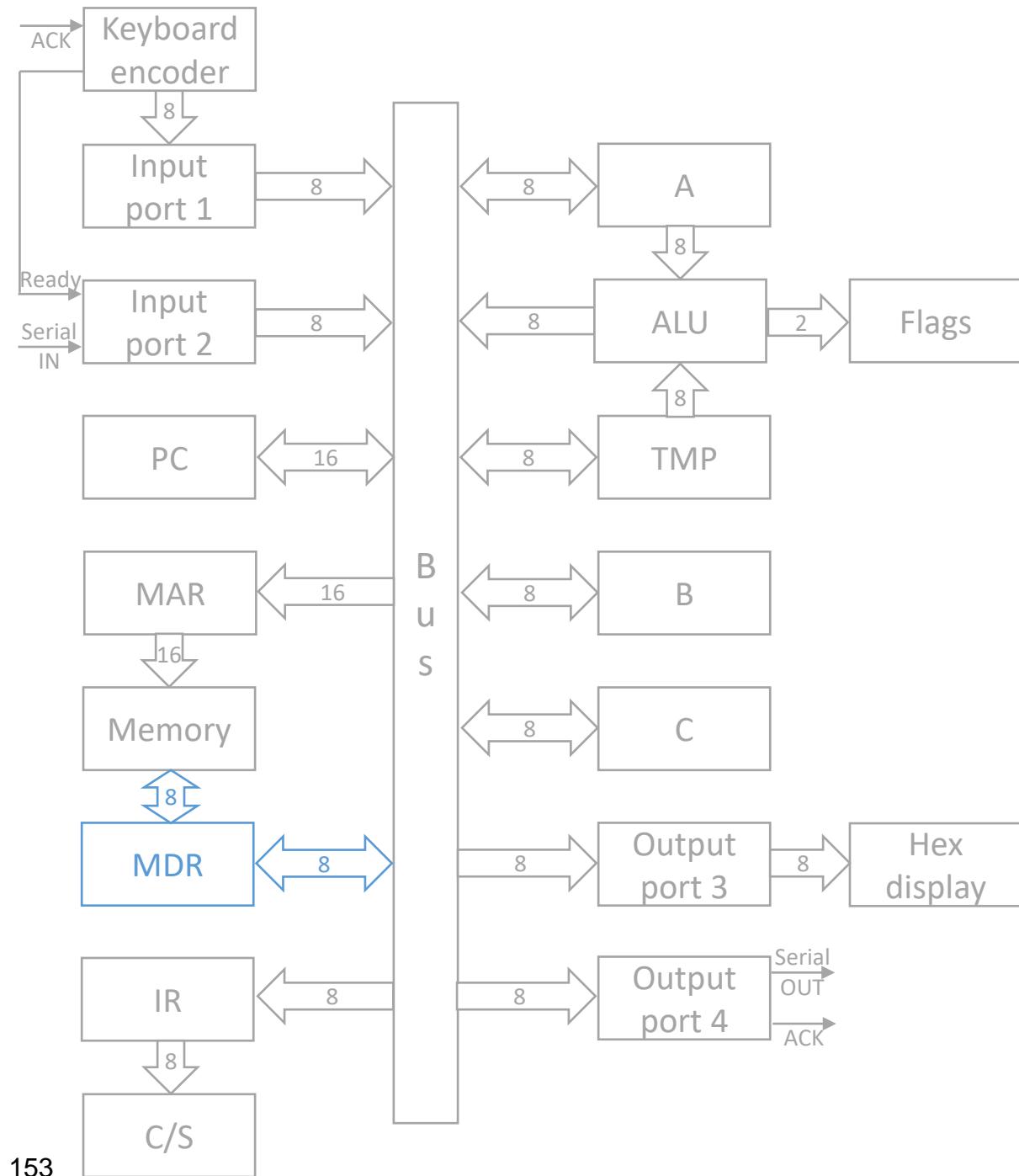
Architecture

- During fetch cycle MAR receive an address from PC and addresses a 64KB memory
- The memory has an initial 2KB ROM (0x0 to 0x7FF) followed by 62KB RAM (0x800 to 0xFFFF)
- ROM contains a program that initializes SAP-2 at power-up and interpret the keyboard inputs



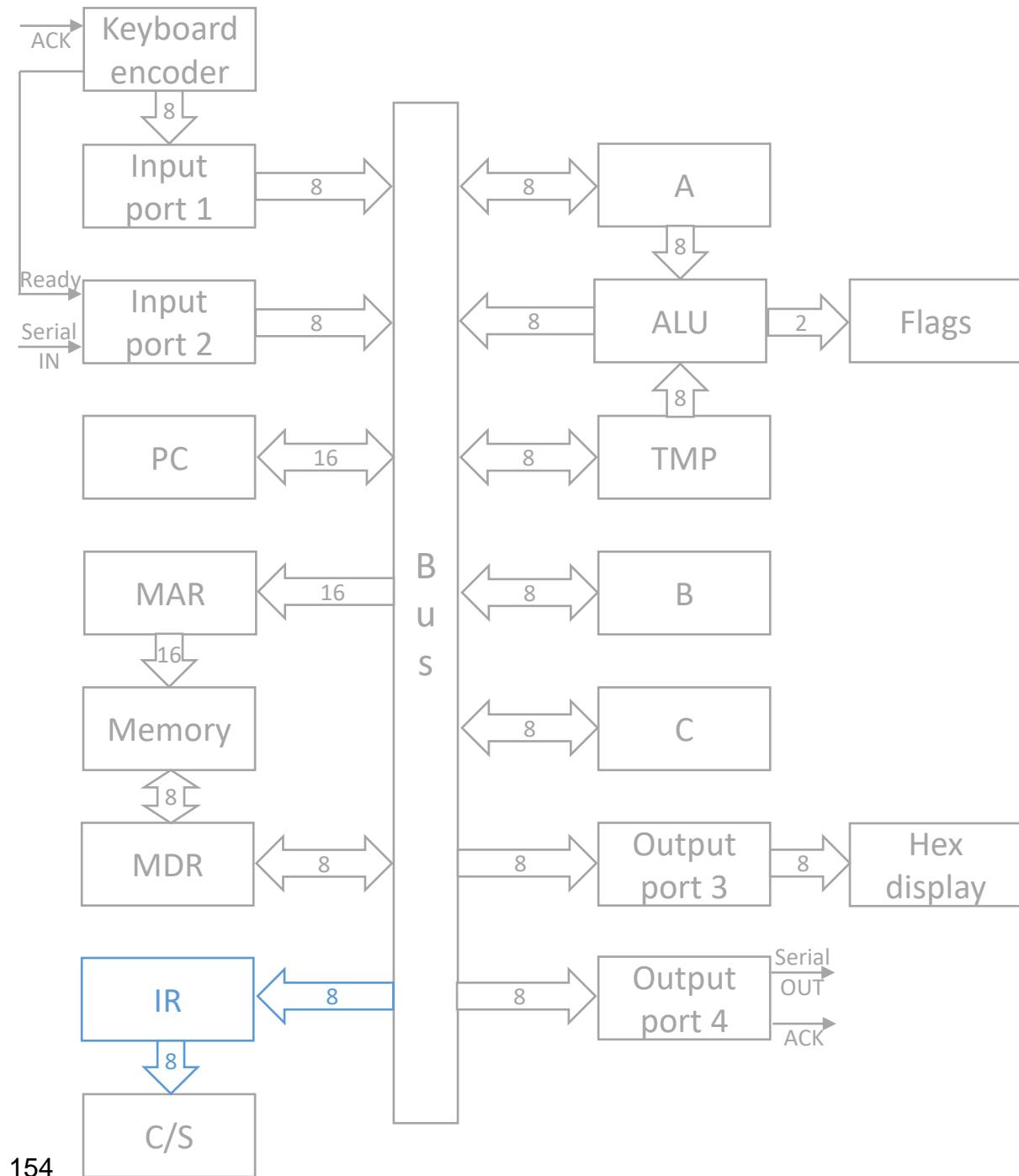
Architecture

- The Memory Data Register (MDR) is a 8-bit buffer register
- It receives data from the bus for writing to the RAM and it sends data to the bus for reading



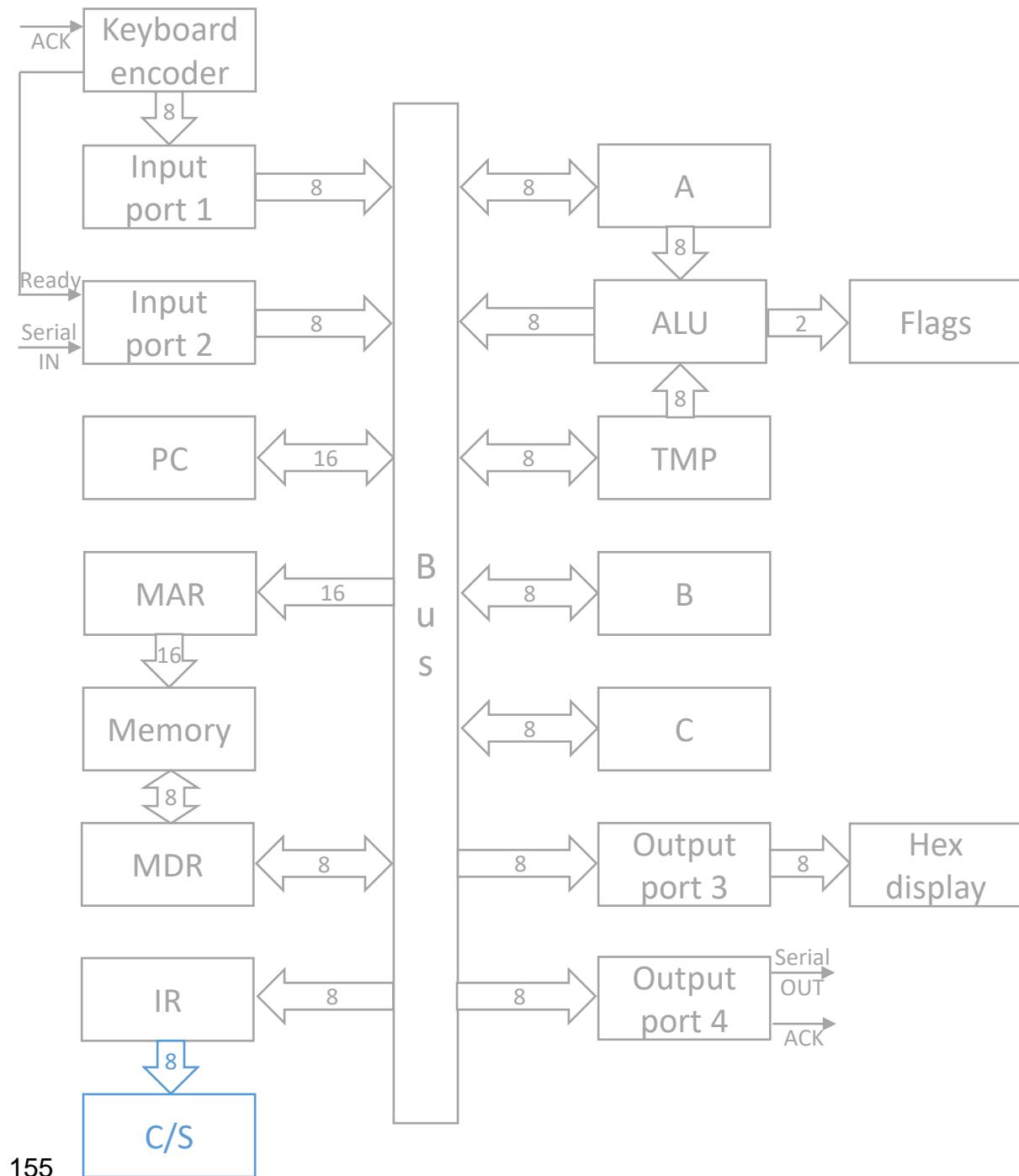
Architecture

- The instruction register use an 8-bit op code because the number of instructions is grater than in SAP-1



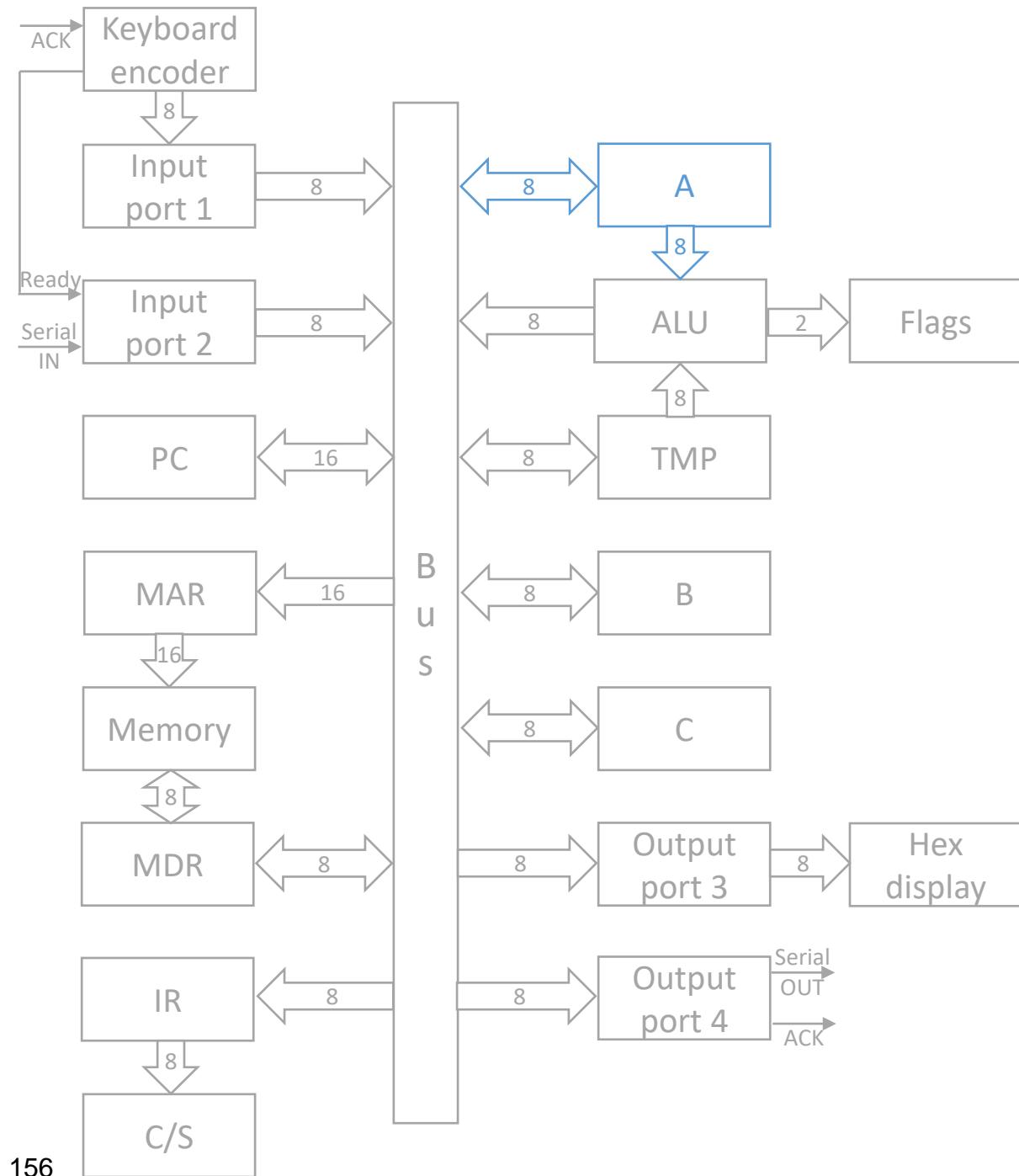
Architecture

- Except from the fact that the control word is bigger, it is conceptually identical to the one of SAP-1



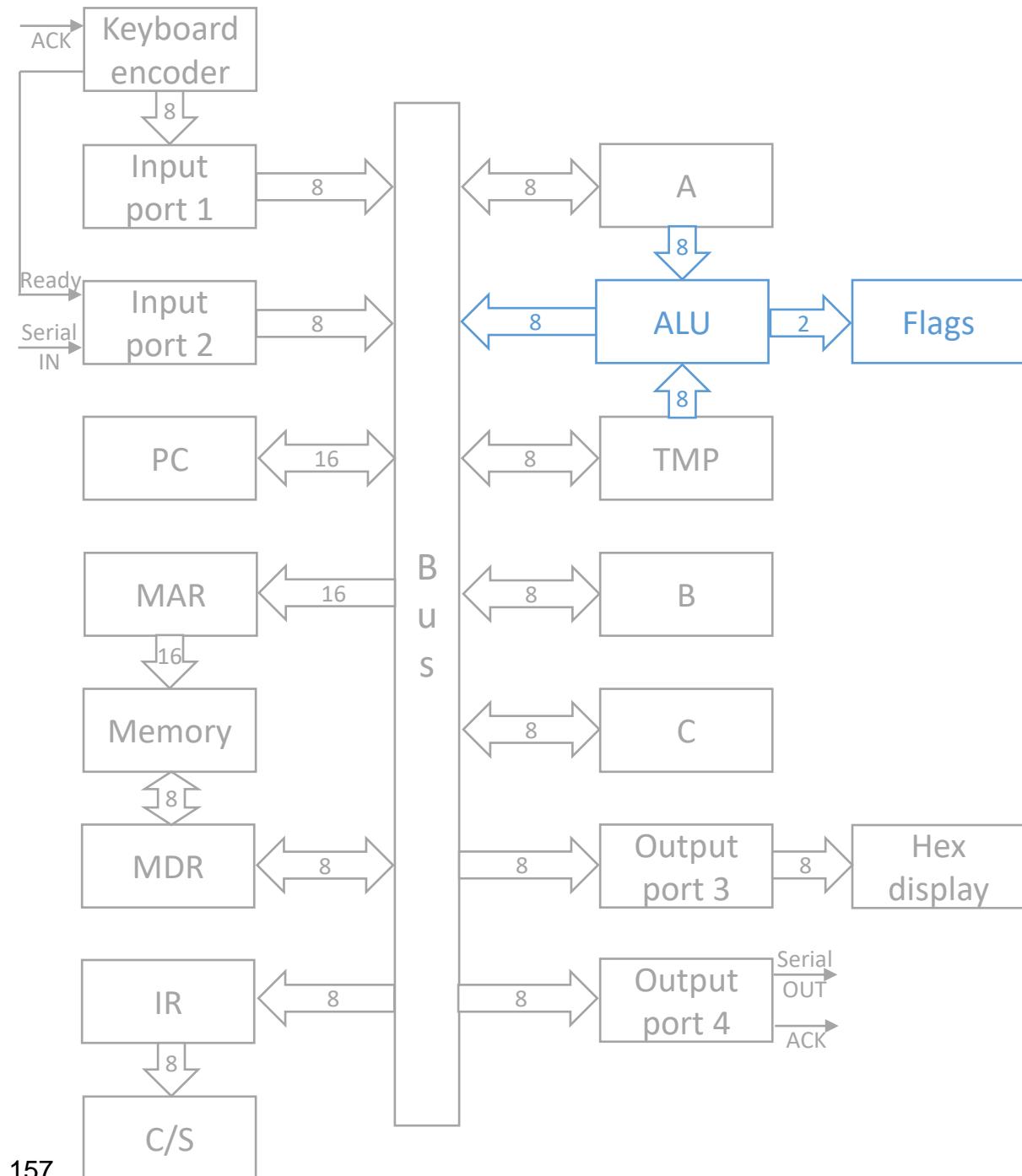
Architecture

- The accumulator is identical to the one of SAP-1



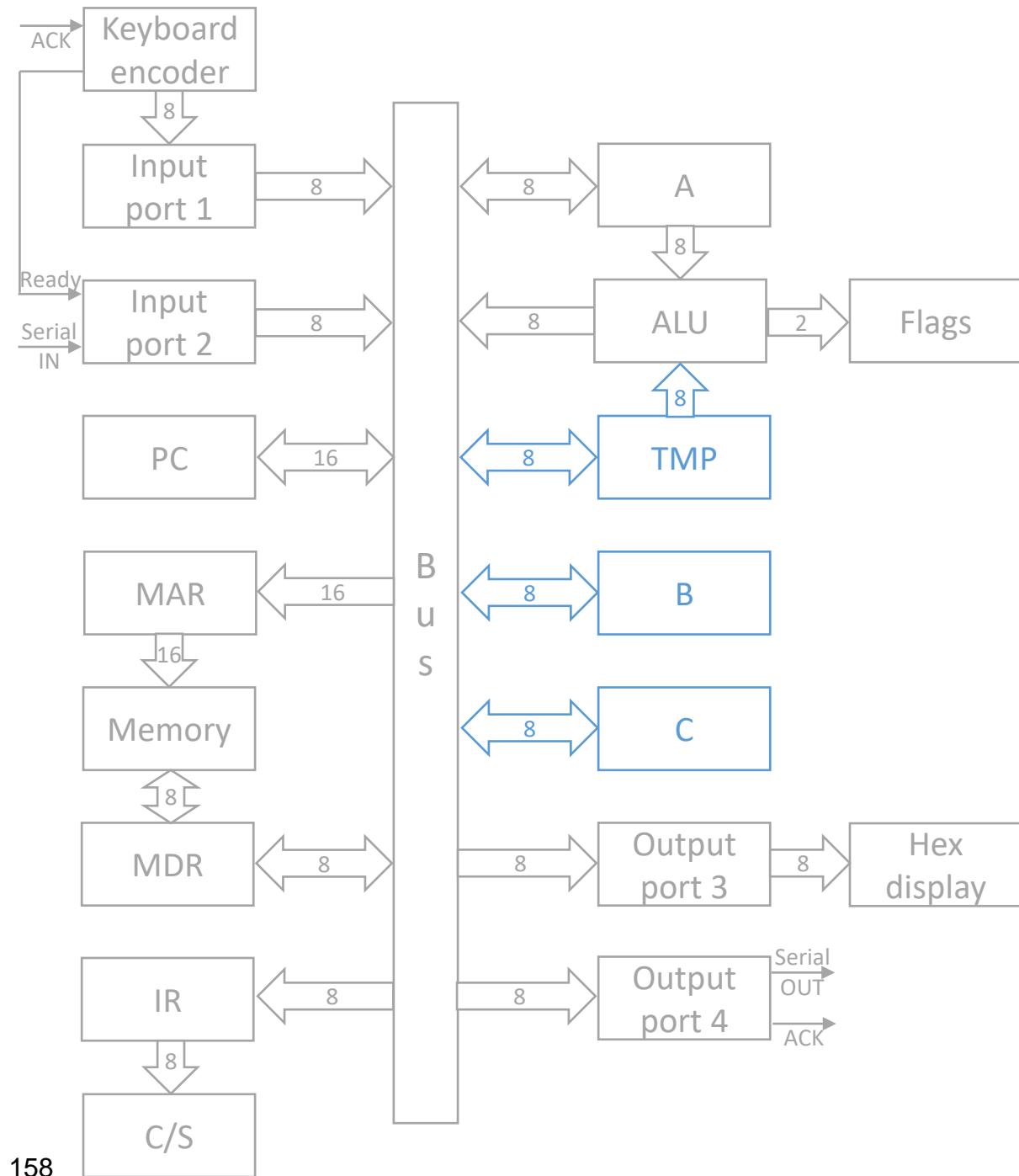
Architecture

- ALU can perform arithmetic and logic operations
- It has some control bits that determine the operation
- Flag is a flip-flop that can keep track of a changing condition
- A Sign flag is set when the accumulator become negative
- A Zero flag is set when the accumulator become zero
- All the instructions that use the ALU can affect the flags



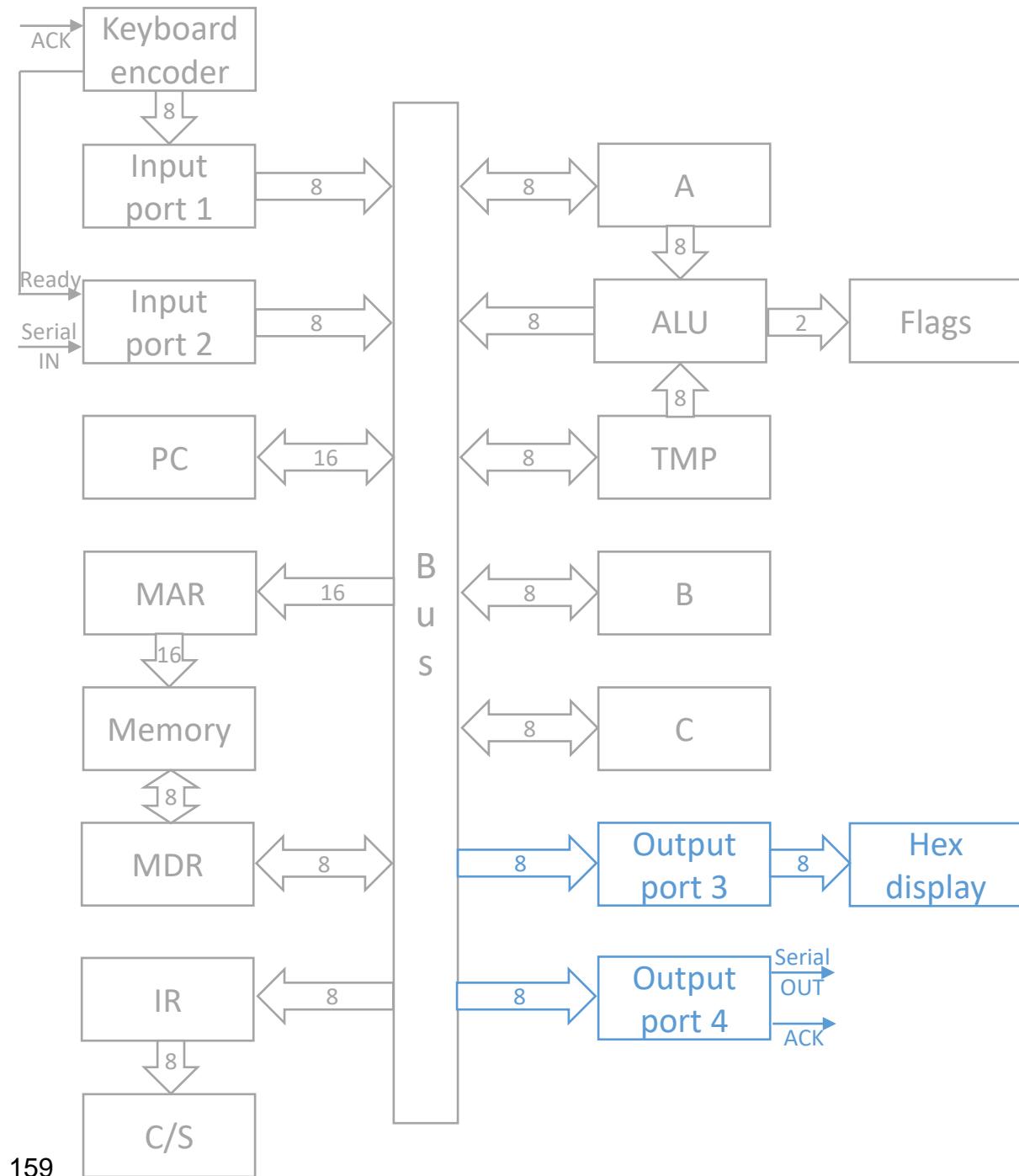
Architecture

- A temporary register TMP provide data to the ALU
- It gives more freedom in the calculation
- There is one more register (C) in SAP-2



Architecture

- There are two output ports
- A hexadecimal display is driven by port 3
- A serial link is fed by port 4
- Acknowledge (ACK) and ready are two signals used for handshaking



Timing states

- The fetch cycle is the same as the one of SAP-1
- The timing states of the execution cycle are different for each instruction and can involve the use of memory (memory-reference instruction)
- The instruction set is built by 43 instructions

LDA and STA

- Load the accumulator is the same as before but the operand is a 16-bit address
- STA means STore the Accumulator
- It needs three bytes to be stored in memory: the op code and the 16-bit operand
- Three memory addresses are used to store LDA or STA instruction
- For instance, “STA 0x00A8” means to copy the content of the accumulator to memory address 0x00A8

$$A = 00010011 \xrightarrow{\text{STA } 0x00A8} M_{168} = 00010011$$

MVI X

- MVI means MoVe Immediate
- It loads a designed register $X=\{A,B,C\}$ with some data
- It needs two bytes to be stored in memory: the op code and the 8-bit operand
- Two memory addresses are used to store a MVI X instruction
- For instance, “MVI B 0x35” means to load the content of register B with the byte 0x35

MVI B 0x35
————— $B = 00110101$

MOV X Y

- Instructions that move the data without passing through memory are called register instructions
- MOV stands for MOVe
- It copies the data from one register $Y=\{A,B,C\}$ to another register $X=\{A,B,C\}$ with $X \neq Y$
- For instance, “MOV A B” means to load the content of register B to register A

$$A = 0x35 \quad B = 0x81 \xrightarrow{\text{MOV A B}} A = 0x81 \quad B = 0x81$$

ADD X and SUB X

- It adds/subtracts the content of register X={A,B,C} to/from the accumulator content
- For instance, “ADD B” means to sum the content of register B to the accumulator register

$$A = 0x35 \quad B = 0x81 \xrightarrow{\text{ADD B}} A = 0xB6 \quad B = 0x81$$

INR X and DCR X

- INR stands for INcRement and DCR for DeCRement
- It increments/decrements by one the content of register X={A,B,C}
- For instance, “INR C” means to add one to the content of register C

$$C = 0x1D \xrightarrow{\text{INR C}} C = 0x1E$$

- These instructions use the accumulator for computing the increment/decrement and then they send back the result to the corresponding register

JMP

- Instructions that change the program sequence are called jump instructions
- JMP stands for JuMP
- It tells to get the instruction from a particular memory address
- It needs a 2-byte operand
- For instance, “JMP 0x0020” at address 0x0005 means to execute as next instruction the one stored at memory address 0x0020

$$PC = 0x0006 \xrightarrow{\text{JMP } 0x0020} PC = 0x0020$$

JM

- JM means Jump if Minus
- The jump is done if and only if the sign flag is set (conditional jump)
- Sign flag is defined as follow

$$S = \begin{cases} 0 & \text{if } A \geq 0 \\ 1 & \text{if } A < 0 \end{cases}$$

- For instance, “JM 0x0020” at address 0x0005 means to execute as next instruction the one stored at memory address 0x0020 if S=1, otherwise the next instruction to be executed is 0x0006

$$PC = 0x0006 \xrightarrow{\text{JM } 0x0020} \begin{cases} PC = 0x0020 & \text{if } S = 1 \\ PC = 0x0006 & \text{if } S = 0 \end{cases}$$

JZ

- JZ means Jump if Zero
- The jump is done if and only if the zero flag is set (conditional jump)
- Zero flag is defined as follow

$$Z = \begin{cases} 0 & \text{if } A \neq 0 \\ 1 & \text{if } A = 0 \end{cases}$$

- For instance, “JZ 0x0020” at address 0x0005 means to execute as next instruction the one stored at memory address 0x0020 if Z=1, otherwise the next instruction to be executed is 0x0006

$$PC = 0x0006 \xrightarrow{\text{JZ 0x0020}} \begin{cases} PC = 0x0020 & \text{if } Z = 1 \\ PC = 0x0006 & \text{if } Z = 0 \end{cases}$$

JNZ

- JNZ means Jump if Not Zero
- The jump is done if and only if the zero flag is not set (conditional jump)
- Zero flag is defined as follow

$$Z = \begin{cases} 0 & \text{if } A \neq 0 \\ 1 & \text{if } A = 0 \end{cases}$$

- For instance, “JNZ 0x0020” at address 0x0005 means to execute as next instruction the one stored at memory address 0x0020 if Z=0, otherwise the next instruction to be executed is 0x0006

$$PC = 0x0006 \xrightarrow{\text{JNZ 0x0020}} \begin{cases} PC = 0x0020 & \text{if } Z = 0 \\ PC = 0x0006 & \text{if } Z = 1 \end{cases}$$

CALL and RET

- CALL stands for CALL the subroutine
- RET stands for RETurn
- A subroutine is a program stored in memory for possible use in another program
- CALL must include the starting address of the desired subroutine
- RET is used at the end of the subroutine to go back to the original program
- When CALL is executed the content of the PC is stored in the last two memory addresses. The operand of CALL is then loaded in the PC
- When RET is executed, the address loaded in the last two memory addresses is loaded back into the PC

Logic instructions

- Besides arithmetic, SAP-2 can do logic
- CMA (CoMplement the Accumulator)
 - inverts each bit of the accumulator
- ANA X (ANd the Accumulator)
 - bitwise AND between the accumulator and register $X=\{B,C\}$
- ORA X (OR the Accumulator)
 - bitwise OR between the accumulator and register $X=\{B,C\}$
- XRA X (XoR the Accumulator)
 - bitwise XOR between the accumulator and register $X=\{B,C\}$

Logic instructions

- Each operator has also an immediate version of the instruction
- ANI (ANd Immediate)
 - bitwise AND between the accumulator and one byte given as operand
- ORI (OR Immediate)
 - bitwise OR between the accumulator and one byte given as operand
- XRI (XoR Immediate)
 - bitwise XOR between the accumulator and one byte given as operand

Miscellaneous instructions

- NOP (No Operation)
 - It is used for delaying data processing
 - Four timing states are needed to fetch and execute a NOP
- HLT (HaLT)
 - As for SAP-1
- IN (INput) and OUT (OUTput)
 - It transfers the data from/to a specific port to/from the accumulator
 - The port is specified in the operand
- RAL (Rotate the Accumulator Left) and RAR (Rotate the Accumulator Right)
 - Shift all accumulator bit to the left/right and move the MSB/LSB to the LSB/MSB

$$A = 0x35 \xrightarrow{\text{RAL}} A = 0x6A$$

Op Code

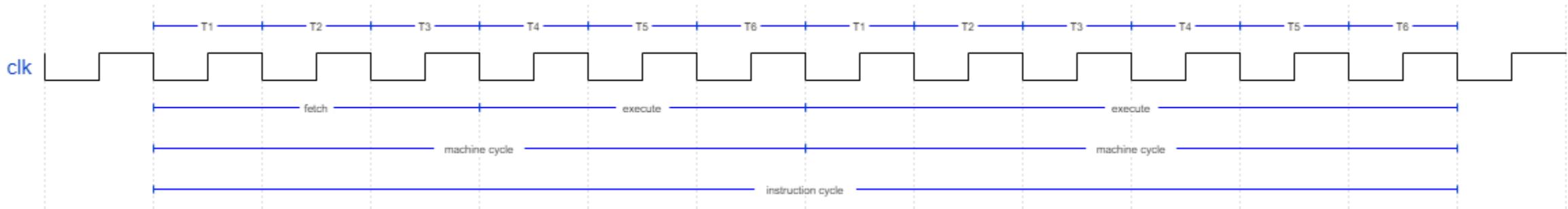
Mnemonics	Op Code
ADD B	0x80
ADD C	0x81
ANA B	0xA0
ANA C	0xA1
ANI byte	0xE6
CALL address	0xCD
CMA	0x2F
DCR A	0x3D
DCR B	0x05
DCR C	0x0D
HLT	0x76
IN byte	0xDB
INR A	0x3C
INR B	0x04
INR C	0x0C

Mnemonics	Op Code
JM address	0xFA
JMP address	0xC3
JMZ address	0xC2
JZ address	0xCA
LDA address	0x3A
MOV A,B	0x78
MOV A,C	0x79
MOV B,A	0x47
MOV B,C	0x41
MOV C,A	0x4F
MOV C,B	0x48
MVI A byte	0x3E
MVI B byte	0x06
MVI C byte	0x0E
NOP	0x00

Mnemonics	Op Code
ORA B	0xB0
ORA C	0xB1
ORI byte	0xF6
OUT byte	0xD3
RAL	0x17
RAR	0x1F
RET	0xC9
STA address	0x32
SUB B	0x90
SUB C	0x91
XRA B	0xA8
XRA C	0xA9
XRI byte	0xEE

Machine/Instruction cycle

- In SAP-2 some instructions takes more than one machine cycle to fetch and execute
- CALL is the more complex instruction that require 18 timing states that correspond to 3 machine cycle for one instruction cycle
- SAP-2 C/S has a variable machine cycle



Exercises

- How many times the DCR instruction is executed in the following program? How much space is needed to store it?

MVI C 0x03

DCR C

JZ 0x0009

JMP 0x0002

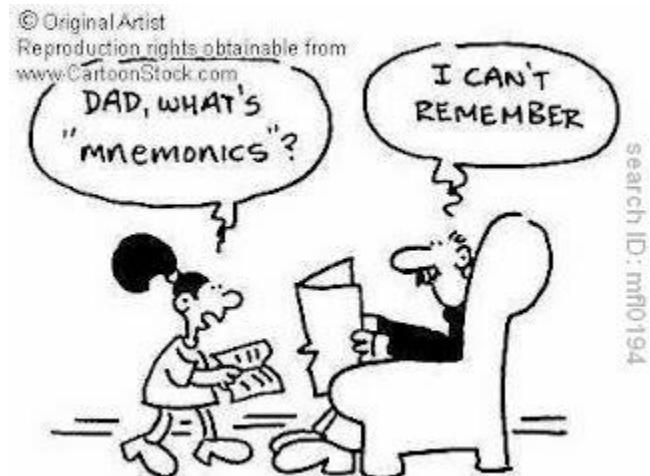
HLT

- Write a program that multiplies the two numbers 11_{10} and 5_{10}
- Write a program that read from input port 2 and if the LSB is a 1 it loads the accumulator with ASCII character Y otherwise with N. Finally, it sends the results to output port 3

SAP-3

SAP-3

- An increased instruction set
- Additional registers (D, E, H and L) equivalent to SAP-2 registers
- F register for storing flags value
- New flags
- Stack pointer: a 16-bit register that controls a dedicated portion of memory



Old instructions

- MOV and MVI
 - Are the same but with more registers
 - More registers means to be faster because register instructions use less timing states than memory reference instructions
- INC and DCR
- ANA, ORA and XRA
- ANI, ORI and XRI
- JMP, JM, JZ and JNZ

New flags

- Carry (CY) flag
 - In order to detect overflow of the accumulator a carry flag is used
 - In output of the 8-bit adder/subtractor there is a carry bit that in the previous versions was disregarded
 - SAP-3 uses this bit called carry for sum and borrow for subtraction as a flag
 - It can be considered as the ninth bit of the accumulator
- Parity (P) flag
 - P is set if the number of 1s of the result of the ALU is even

New instructions

- STC (SeT Carry)
 - It sets the carry flag if it is not already set
- CMC (CoMplement the Carry)
 - It sets the carry to its complement
 - STC followed by CMC resets the carry
- ADD X and SUB X are the same but setting the carry flag
- ADC X (Add with Carry) and SBB (SuBtract with Borrow)
 - It adds/subtracts the content of a register to/from the accumulator plus/minus the carry flag

$$A = 0x83 \quad E = 0x12 \quad CY = 1 \xrightarrow{ADC \ E} A = 0x96 \quad E = 0x12 \quad CY = 0$$

$$A = 0xFF \quad E = 0x02 \quad CY = 1 \xrightarrow{SBB \ E} A = 0xFC \quad E = 0x02 \quad CY = 0$$

New instructions

- RAL (Rotate All Left) and RAR (Rotate All Right)
 - Rotate left/right all the bit of the accumulator including the carry flag

$$CY = 1 \ A = 0x74 \xrightarrow{RAL} CY = 0 \ A = 0xE9$$

- RLC (Rotate Left with Carry) and RRC (Rotate Right with Carry)
 - Rotated left/right the accumulator bit and the MSB/LSB is saved in the carry flag

$$CY = 1 \ A = 0x74 \xrightarrow{RLC} CY = 0 \ A = 0xE8$$

New instructions

- CMP X (CoMPare)
 - Compare the content of a register with the content of the accumulator
 - The outcome can be seen from the zero flag

$$Z = \begin{cases} 0 & \text{if } A \neq X \\ 1 & \text{if } A = X \end{cases}$$

- ADI, ACI, SUI, SBI and CPI are immediate instructions
 - They need a byte as operand
 - ADI/ACI adds the operand to the accumulator without/with the carry flag
 - SUI/SBI subtracts the operand and the carry flag from the accumulator
 - CPI compares the immediate byte with the accumulator

New instructions

- JP (Jump if Positive)
 - It has the opposite effect of JM
 - It jumps also if zero
- JC and JNC (Jump if Carry and Jump if Not Carry)
 - JC (JNC) jumps if the carry flag is (is not) set
- JPE and JPO (Jump if Parity Even and Jump if Parity Odd)
 - JPE/JPO jumps in case P is set/reset

Extended register instructions

- Some instructions use pairs of registers in order to process 16-bit word
- Pairs are B-C, D-E and H-L so they are abbreviated B, D and H in the context of extended register
- LXI Y double-byte (Load the eXtended Immediate)
 - Load the operand to a specific extended register $Y=\{B,D,H\}$
- DAD Y (Double Add)
 - Add the content of a specific extended register $Y=\{B,D,H\}$ to H
- INX Y and DCX Y (Increment and DeCrement the eXtended)
 - Increment/decrement a specific extended register $Y=\{B,D,H\}$

Indirect instructions

- The extended register H is used as a data pointer
- Instead of using a direct addressing (like LDA X where X is a double-byte), they use a pointer to a memory location
- MOV X M
 - Load a specified register X={A,B,C,D,E,H,L} with the data addressed by HL

$$HL = 0x3404 \quad M_{0x3404} = 0x74 \xrightarrow{\text{MOV B M}} B = 0x74$$

- MOV M X
 - Load the memory location addressed by HL with the content of X={A,B,C,D,E,H,L}

$$HL = 0x3404 \quad C = 0x98 \xrightarrow{\text{MOV M C}} M_{0x3404} = 0x98$$

Indirect instructions

- MVI M byte
 - Load the memory location addressed by HL with a data byte

$$HL = 0x14A3 \xrightarrow{\text{MVI M } 0xB1} M_{0x14A3} = 0xB1$$

- All the following instructions are analogue to their register version, but they use instead the memory location at address HL
 - ADD M, ADC M, SUB M, SBB M, INR M, DCR M, ANA M, ORA M, XRA M and CMP M

Stack instructions

- Stack is a portion of memory used for storing the return address of subroutines
- In SAP-2 there is a small stack at the end of the memory: two memory addresses are used for storing the return address
- In SAP-3
 - The programmer can choose where to locate the stack and how big it is
 - There are special instructions for read and write into the stack
 - Stack instructions are indirect because they use a Stack Pointer (SP) similarly to HL
 - To initialize SP the immediate load instruction can be used: LXI SP double-byte

Stack instructions

- The PUSH Y instruction is used for saving the program status before calling a subroutine
 - The extended register is $Y=\{B,D,H,PSW\}$ where PSW is the program status word that is the concatenation of the registers A (Accumulator) and F (Flags)
 - When PUSH is executed
 - The stack pointer is decremented: $SP-1$
 - The high byte of Y is stored in M_{SP-1}
 - The stack pointer is decremented again: $SP-2$
 - The low byte of Y is stored in M_{SP-2}

Stack instructions

- The POP Y instruction is used for getting back the program status after a subroutine is called back
 - The extended register is $Y=\{B,D,H,PSW\}$ where PSW is the program status word that is the concatenation of the registers A (Accumulator) and F (Flags)
 - When POP is executed
 - The low byte of Y is loaded with the content of M_{SP}
 - The stack pointer is incremented: $SP+1$
 - The high byte of Y is loaded with the content of M_{SP+1}
 - The stack pointer is incremented again: $SP+2$

Stack instructions

- CALL is used for saving automatically return addresses
 - When executed the PC is pushed onto the stack and the starting address of the routine is loaded into the PC
- RET is used at the end of a subroutine for coming back to the previous address
 - When executed it pops the return address off the stack into the PC
- For instance:

0x2000	LXI SP 0x2100
0x2003	CALL 0x8050
0x2006	MVI A 0x0E
...	
0x20EF	HLT
...	
0x8050	...
...	
0x8059	RET

- LXI and CALL takes 3 bytes because of the operand
- LXI load the SP with 0x2100
- CALL makes the address 0x2006 saved in the stack
 - SP is decremented
 - 0x20 is stored in 0x20FF
 - SP is decremented
 - 0x06 is stored in 0x20FE
- CALL makes the address 0x8050 loaded in PC
- At the end of the routine RET is executed
 - 0x06 is loaded into the low byte of PC
 - SP is incremented
 - 0x20 is loaded into the high byte of PC
 - SP is incremented

Stack instructions

- Call and return instructions can be conditional
- CNZ, CZ, CNC, CC, CPO, CPE, CP and CM
- RNZ, RZ, RNC, RC, RPO, RPE, RP and RM
- They are similar to conditional jumps with the same meanings
(instead of J, C for call and R for return)

Exercises

- Write a program that adds 1200_{10} to 3400_{10} and store the result in registers H and L, with and without the use of extended register instructions
- Using pointers, write a program that moves all the data stored between memory address 0x2030 and 0x212F to the memory space between address 0x5030 and 0x512F