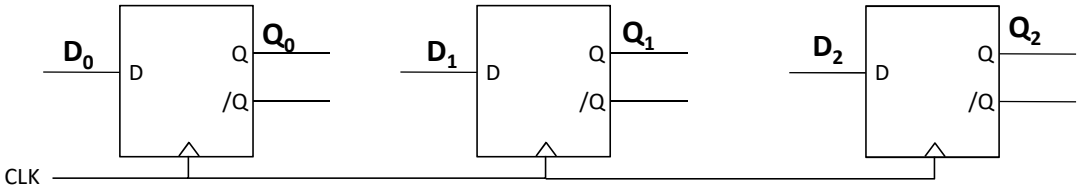
Digital systems: from bits to microcontrollers

- Introduction to digital systems fundamentals
- Combinatorial digital systems
- Sequential digital systems
- Introduction to microcontrollers
- Introduction to advanced implementation platforms

Revisiting registers

Sometimes is necessary that one register performs different functions along the time.

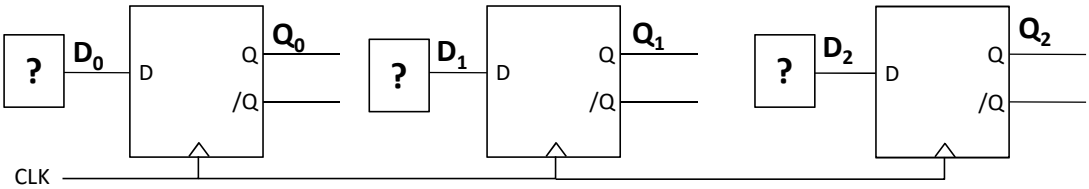
As the flip-flop structure is the same for all functions, the different functions are implemented through different flip-flop input functions.



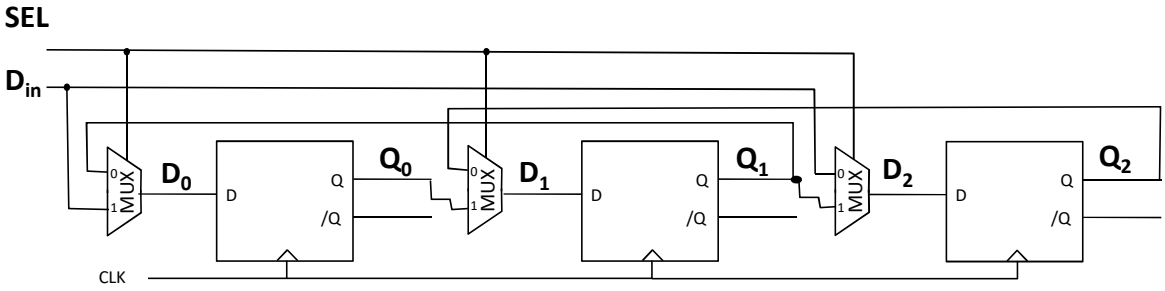
Revisiting registers

Sometimes is necessary that one register performs different functions along the time.

As the flip-flop structure is the same for all functions, the different functions are implemented through different flip-flop input functions.

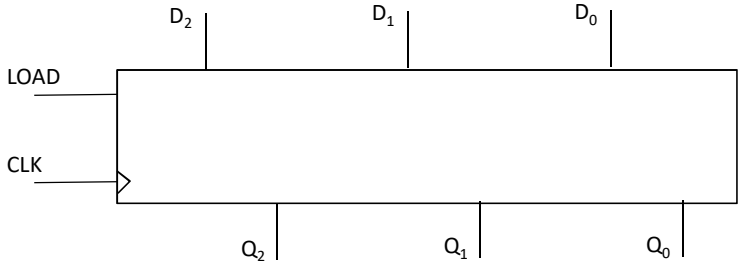


Left-right shift-register



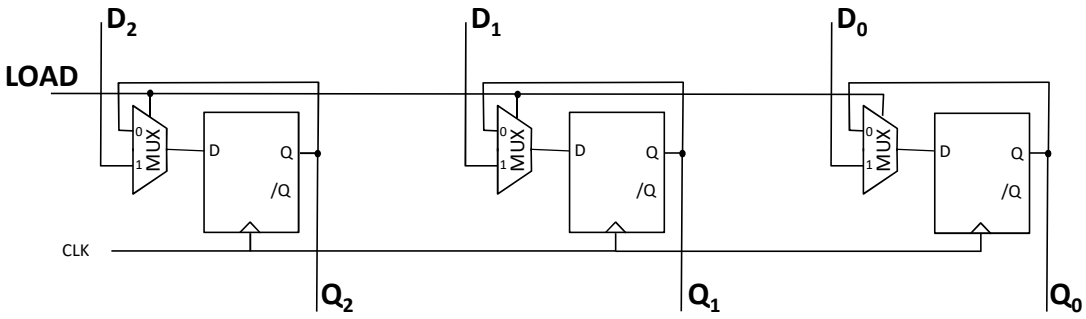
If (SEL = 0) then shift left
else shift right

Register with parallel load control



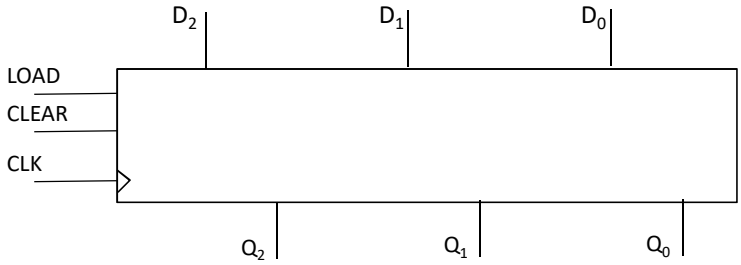
If (LOAD = 1) then load data inputs
else freeze outputs

Register with parallel load control



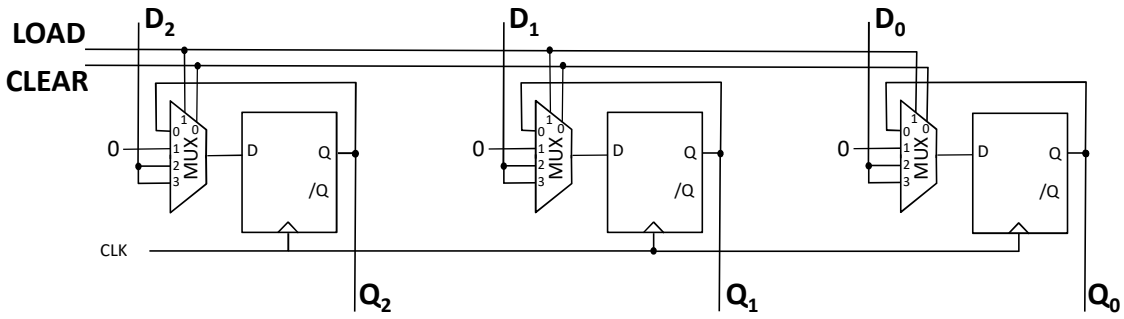
If (LOAD = 1) then load data inputs
else freeze outputs

Register with parallel load control and clear



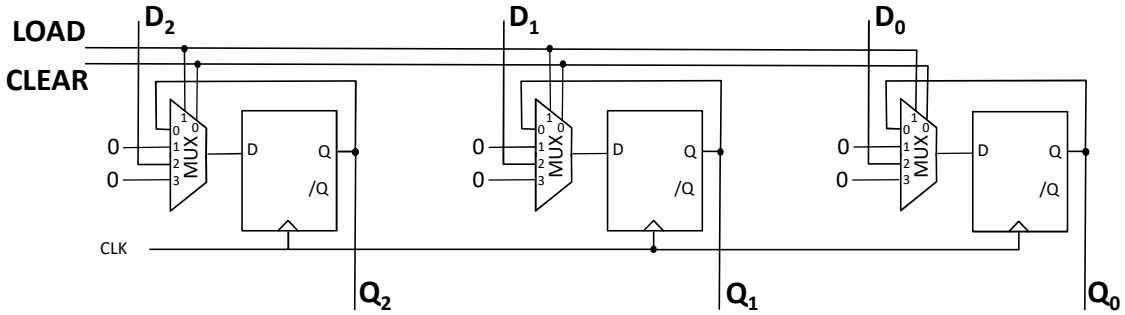
If (LOAD = 1) then load data inputs
else if (CLEAR = 1) then clear outputs
else freeze outputs

Register with parallel load control and clear



If (LOAD = 1) then load data inputs
else if (CLEAR = 1) then clear outputs
else freeze outputs

Register with parallel load control and clear



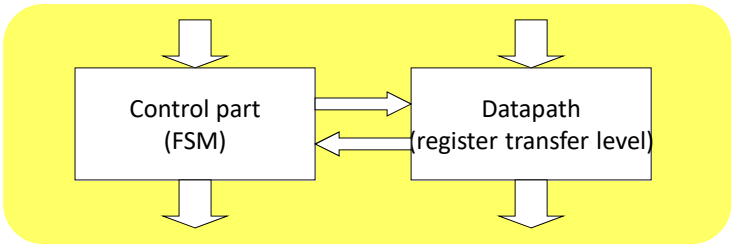
If (CLEAR = 1) then clear outputs
else if (LOAD = 1) then load data inputs
else freeze outputs

Coping with data processing dominated systems → FSM with Datapaths

Finite State Machine ← "Control-dominated"

Finite State Machine cooperative with data processing architecture ← "Data-processing-dominated"

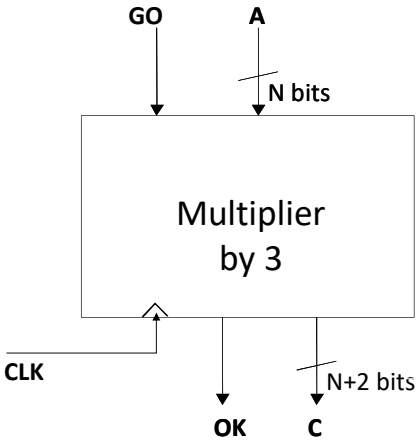
→ Partitioning the system into control part and datapath



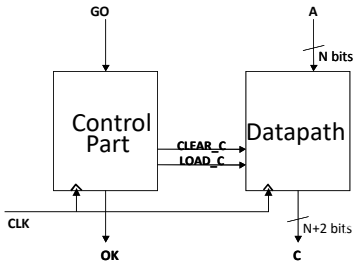
A simple multiplier

Goal: to implement one multiplier of one number (with N bits) by 3.

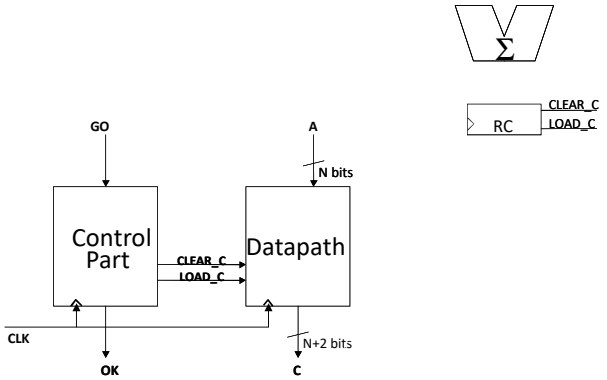
Strategy: Result = 0 + A + A + A



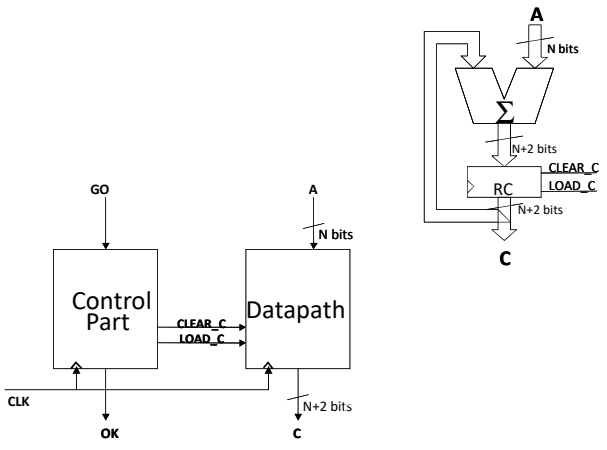
A 3*A multiplier (C = 0+A+A+A)



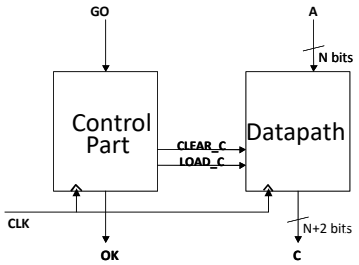
A 3*A multiplier (C = 0+A+A+A)

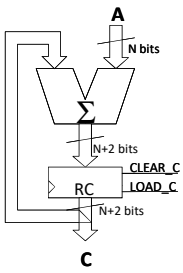


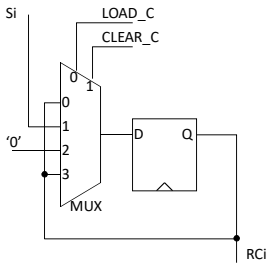
A 3*A multiplier (C = 0+A+A+A)



A 3*A multiplier (C = 0+A+A+A)

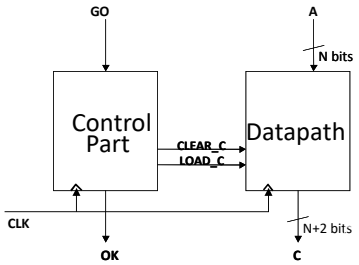


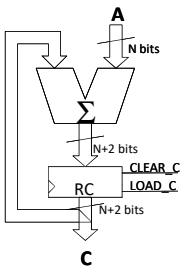


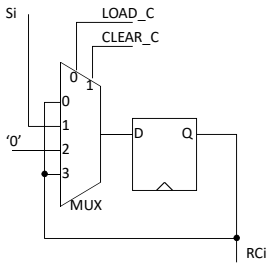


A 3*A multiplier (C = 0+A+A+A)

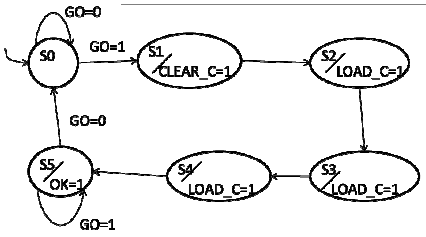
- wait for GO=1
- Clear RC
- Add A
- Add A
- Add A
- Wait for GO=0



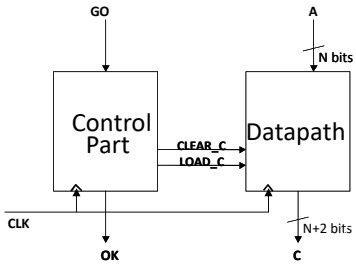


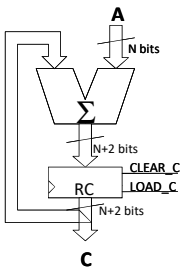


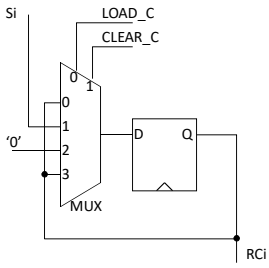
A 3*A multiplier (C = 0+A+A+A)



- wait for GO=1
- Clear RC
- Add A
- Add A
- Add A
- Wait for GO=0





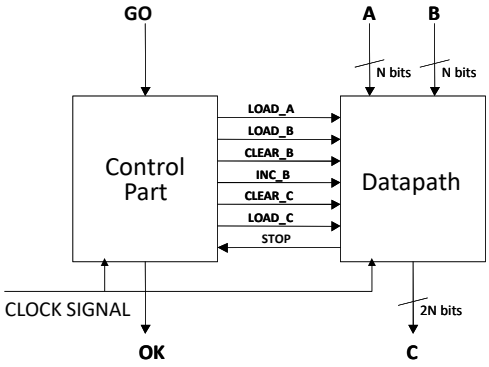


Exercise: multiplier of two numbers

Goal: to implement one multiplier of two numbers (with N bits each).

One multiplier (I)

Goal: to implement one multiplier of two numbers (with N bits each), using successive additions algorithm.



$C = A \times B$

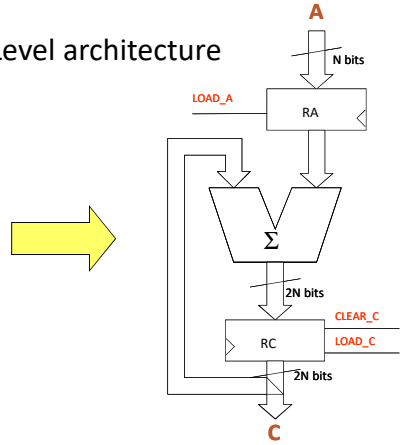
$C = 0 + A + A + \ldots + A$

B times

One multiplier (II)

Datapath based on a Register Transfer Level architecture

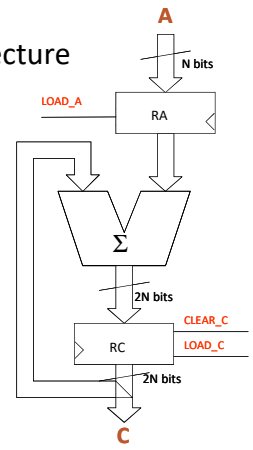
→ Successive additions, using add-and-accummulate register to store partial additions as well as final result



One multiplier (II)

Datapath based on a Register Transfer Level architecture

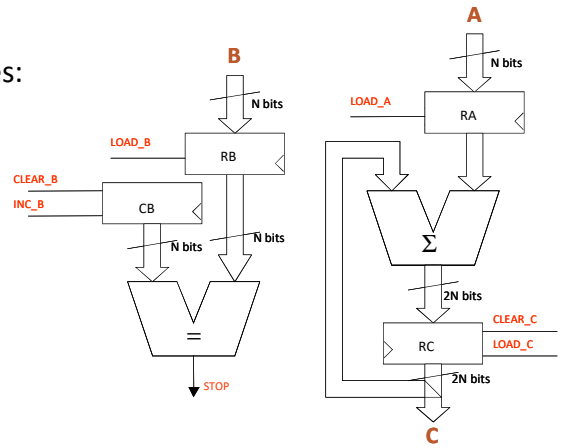
- Successive additions, using add-and-accummulate register to store partial additions as well as final result
- What about B times counter?



One multiplier (III)

Datapath: counting B times:

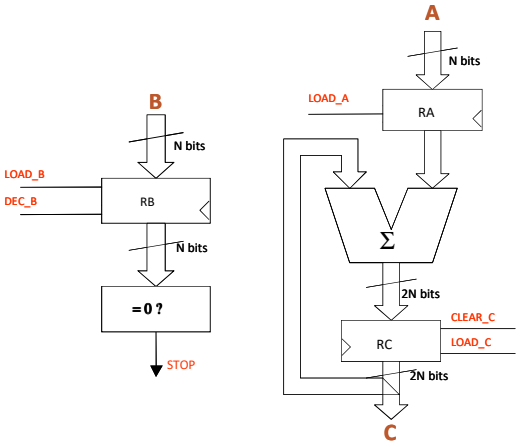
Option 1 - using an up-counter



One multiplier (IV)

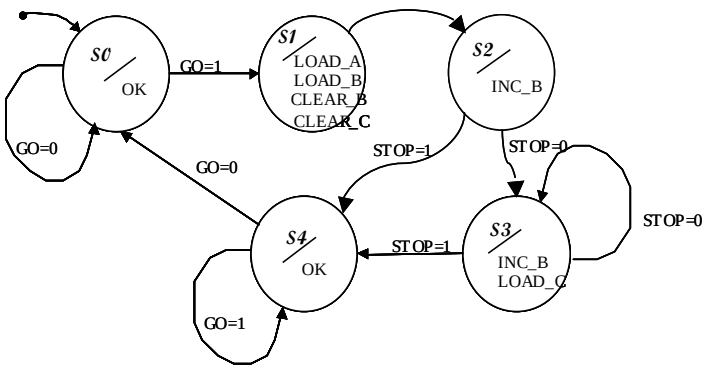
Datapath: counting B times

Option 2 - using a down-counter

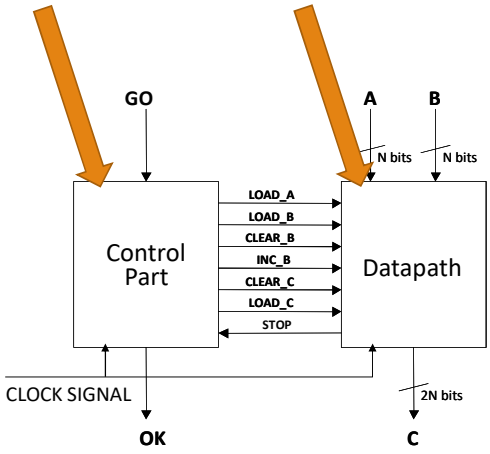


One multiplier (V)

Control part



FSM with Datapath

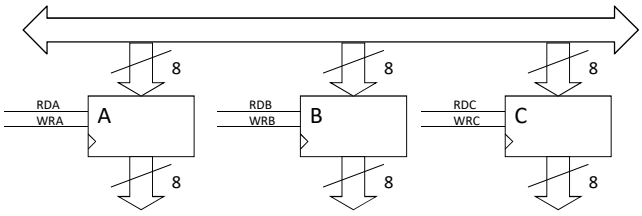


$C = A \times B$

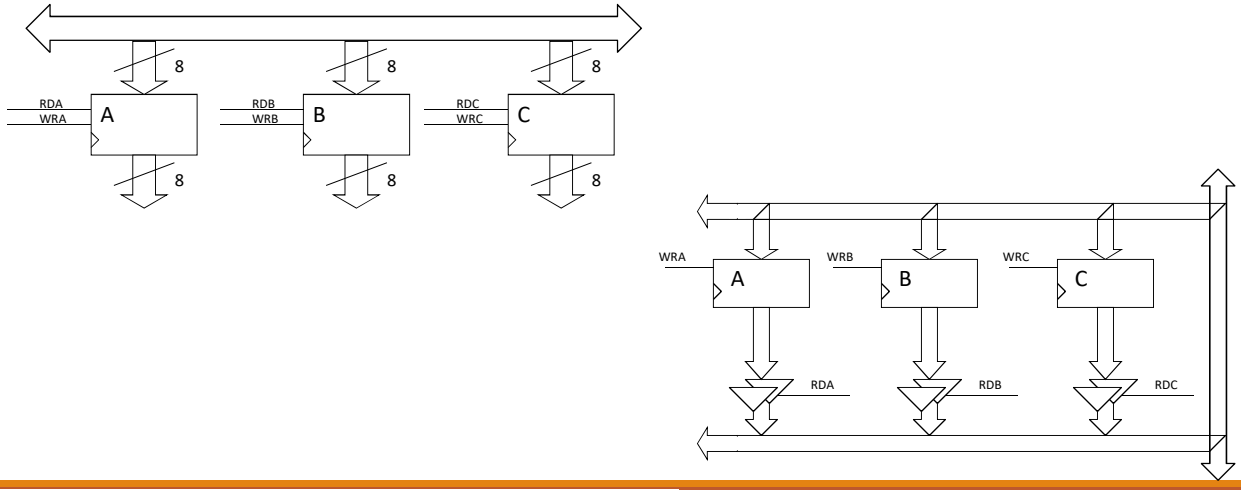
$C = 0 + A + A + \ldots + A$

B times

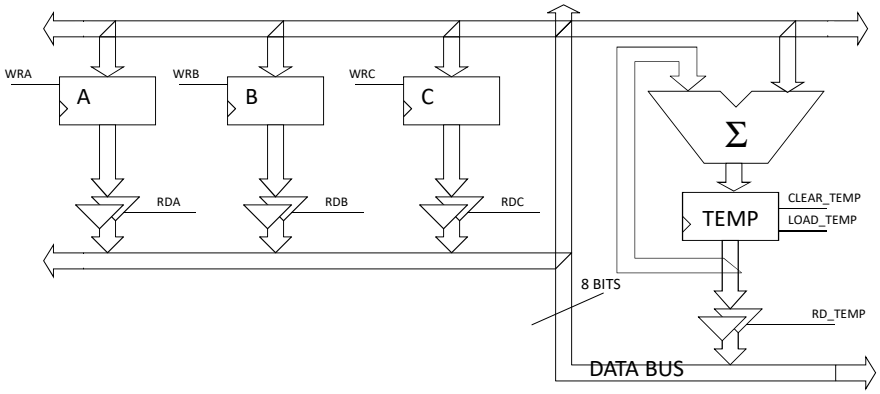
Introducing tri-state buffers



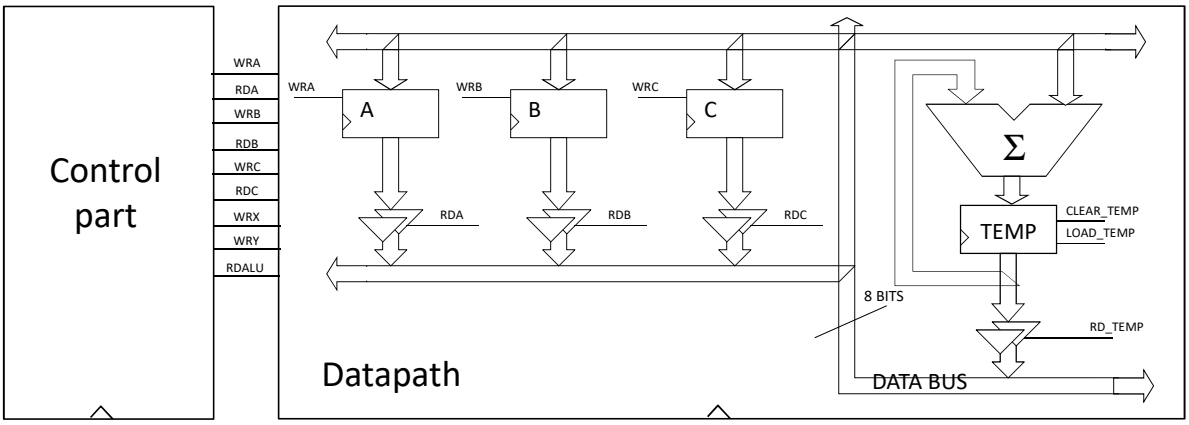
Introducing tri-state buffers



Introducing buses

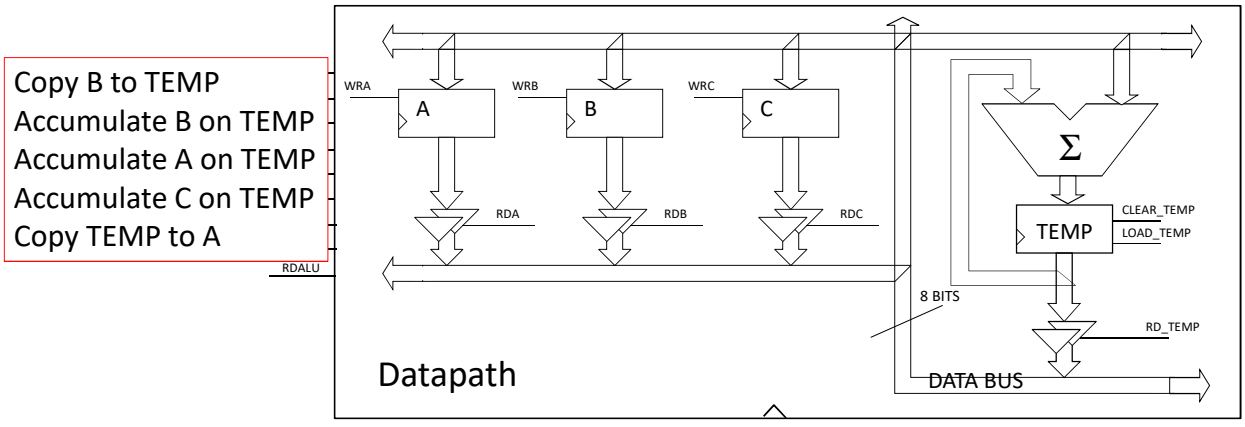


Control sequence to compute A ← 2*B+A+C



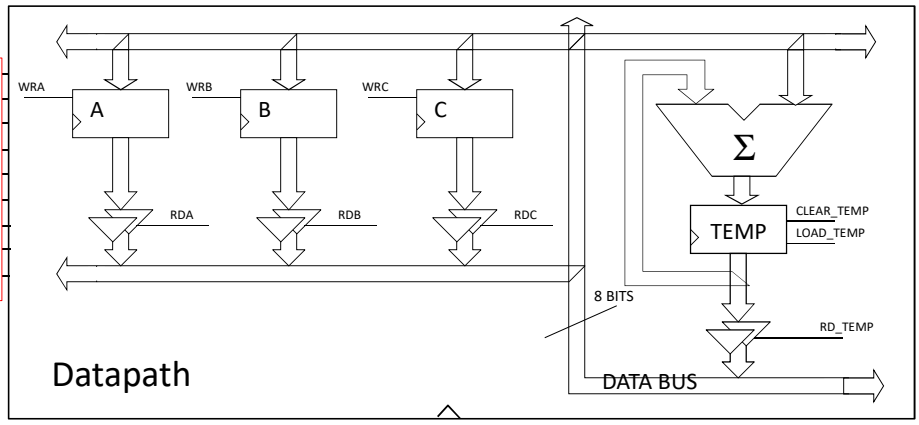
Control sequence to compute A ← 2*B+A+C (A ←B+B+A+C)

Copy B to TEMP
Accumulate B on TEMP
Accumulate A on TEMP
Accumulate C on TEMP
Copy TEMP to A

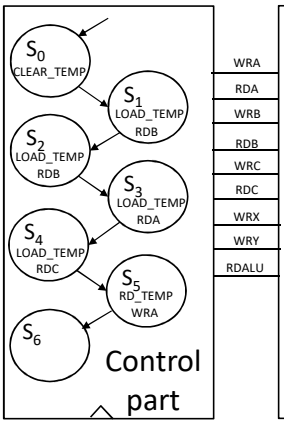


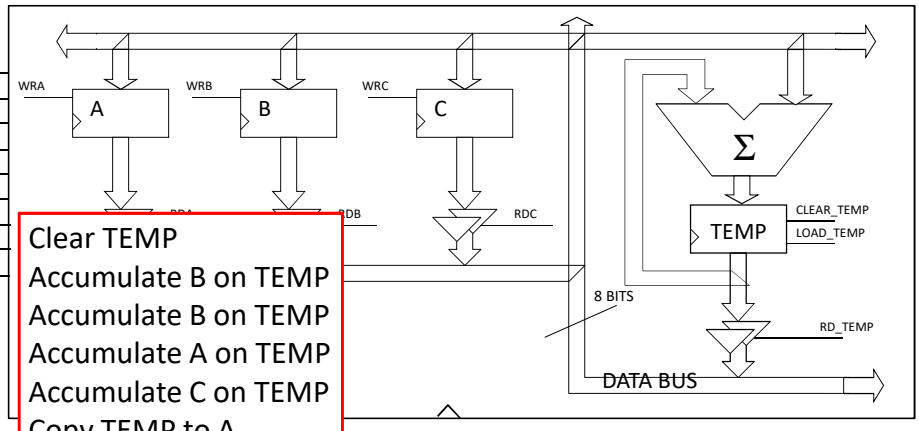
Control sequence to compute A ← 2*B+A+C (A ←B+B+A+C)

Clear TEMP
Accumulate B on TEMP
Accumulate B on TEMP
Accumulate A on TEMP
Accumulate C on TEMP
Copy TEMP to A



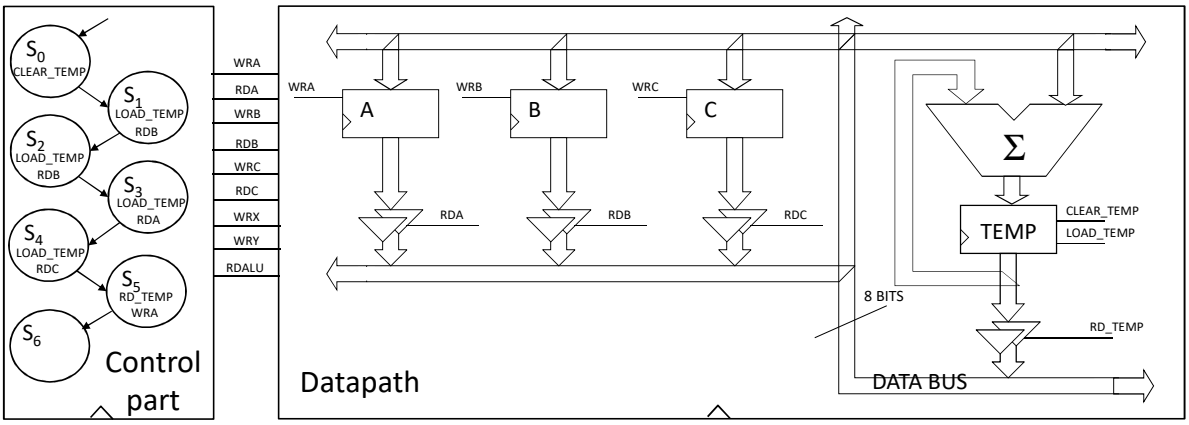
Control sequence to compute A ← 2*B+A+C (A ←B+B+A+C)



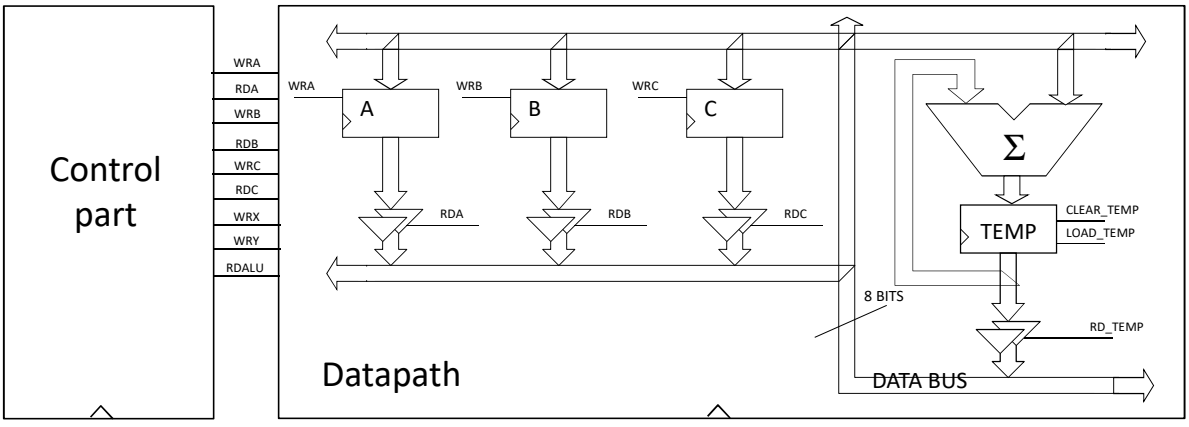


Clear TEMP
Accumulate B on TEMP
Accumulate B on TEMP
Accumulate A on TEMP
Accumulate C on TEMP
Copy TEMP to A

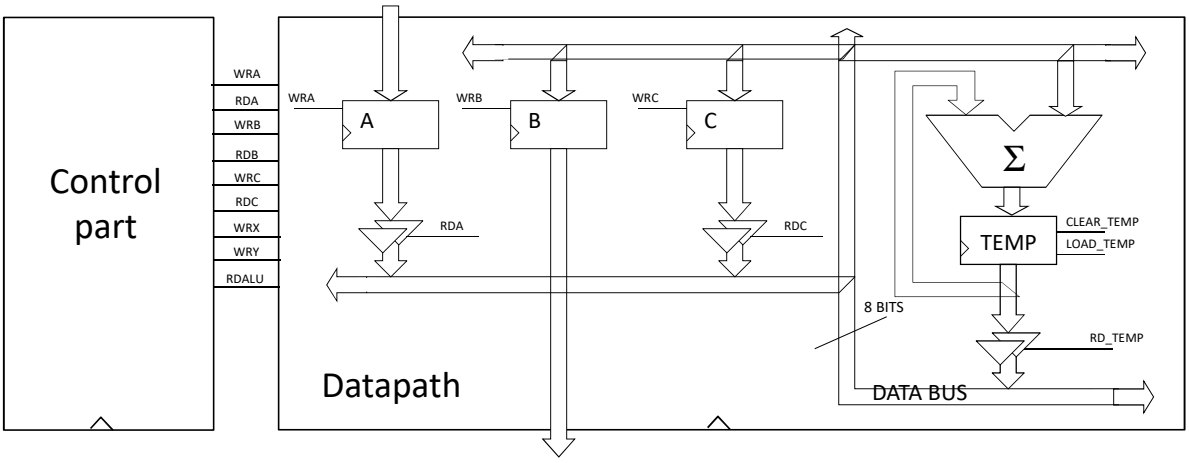
Control sequence to compute A ← 2*B+A+C (A ←B+B+A+C)



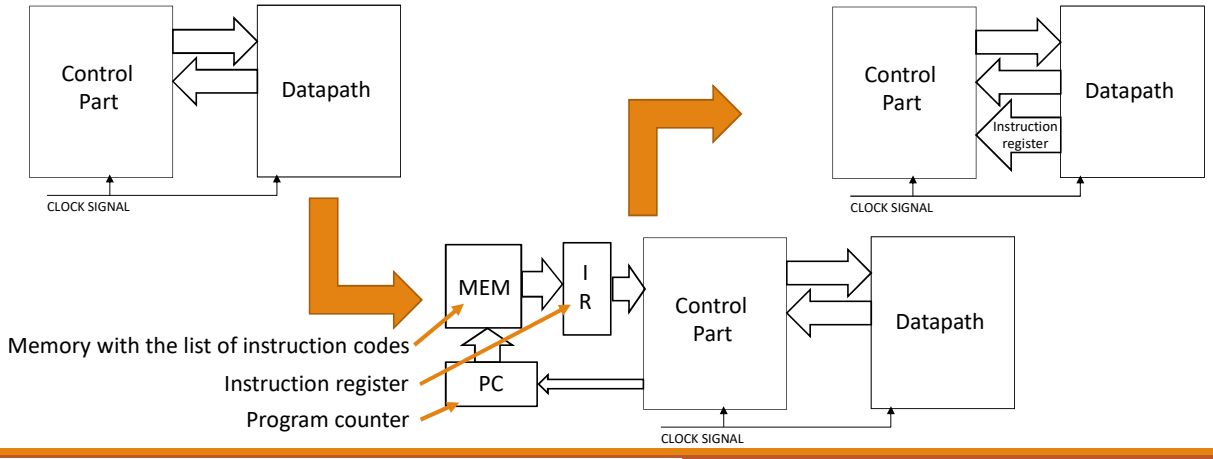
Control sequence to compute A ← 2*(B+A)+3*C



Connecting to the external world



Moving from dedicated architectures to general purpose microprocessor architectures



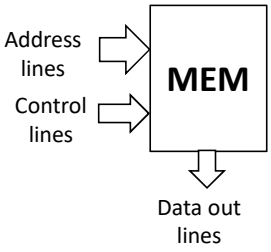
Introducing memories

Unit: BIT (BInary digiT)

- 1 Byte = 8 Bits
- 1 K (Kilobyte) = 2^{10} = 1024
- 1 M (Megabyte) = 2^{20} = 1 048 576
- 1 G (Gigabyte) = 2^{30} = 1 073 741 824
- 1 T (Terabyte) = 2^{40} = 1 099 511 627 776
- 1 P (Petabyte) = 2^{50} = 1 125 899 906 842 624

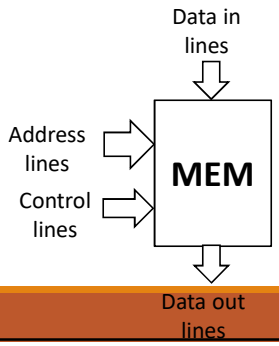
Different technologies available for solid state memories

ROM	Read-Only Memory
PROM	Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory

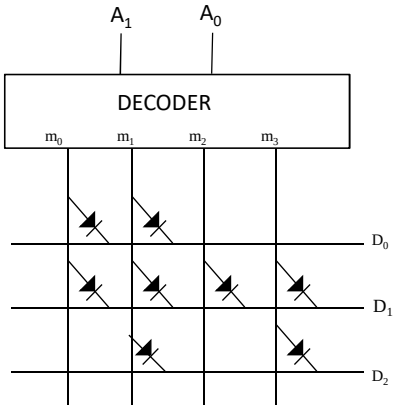


RAM	Random Access Memory
SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory

…



ROM structure



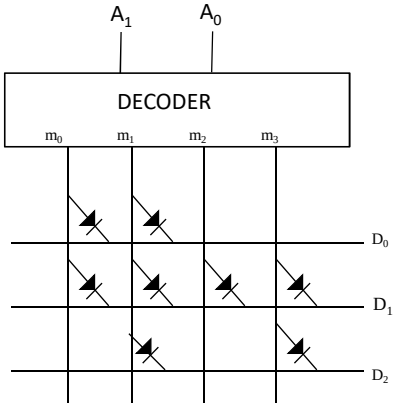
$D_0=m_0 + m_1$

$D_1=m_0 + m_1 + m_2 + m_3$

$D_2=m_1 + m_3$

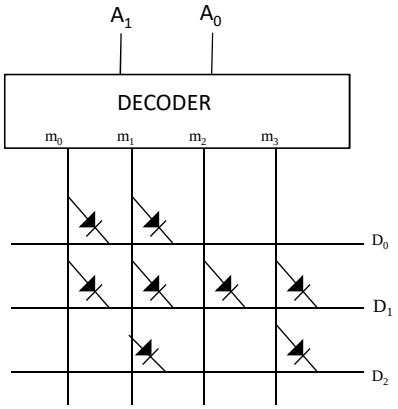
Address	Data
0	011
1	111
2	010
3	110

Using a ROM to implement a function



Address	A_1A_0	Data
0	00	011
1	01	111
2	10	010
3	11	110

Using a ROM to implement a function

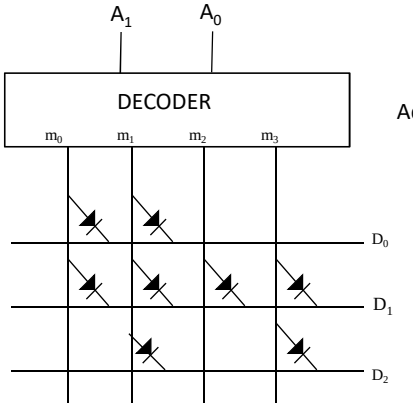


F2(A,B)=Σ(1,3)
F1(A,B)=Σ(0,1,2,3)
F0(A,B)=Σ(0,1)

A B	$F_2F_1F_0$
00	0 1 1
01	1 1 1
10	0 1 0
11	1 1 0

A_1A_0	$D_2D_1D_0$
00	0 1 1
01	1 1 1
10	0 1 0
11	1 1 0

Using a ROM to implement a function



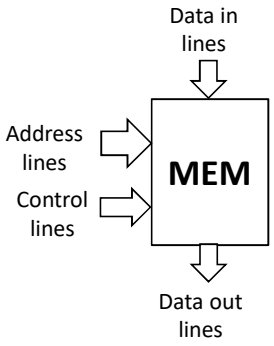
Address lines
A_1A_0

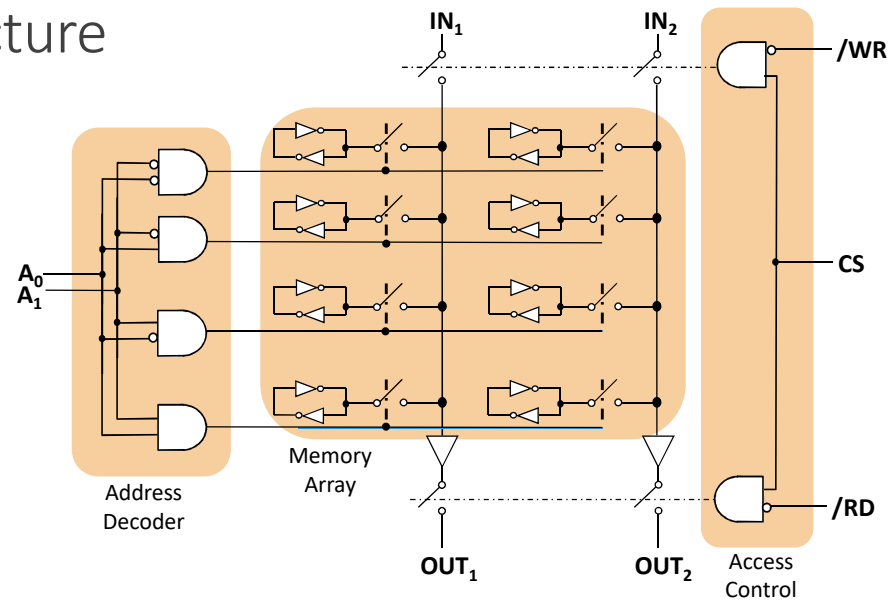
MEM 4x3

$D_2D_1D_0$
Data out lines

A_1A_0	$D_2D_1D_0$
00	0 1 1
01	1 1 1
10	0 1 0
11	1 1 0

RAM structure

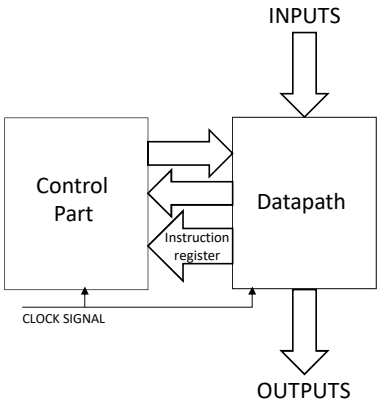




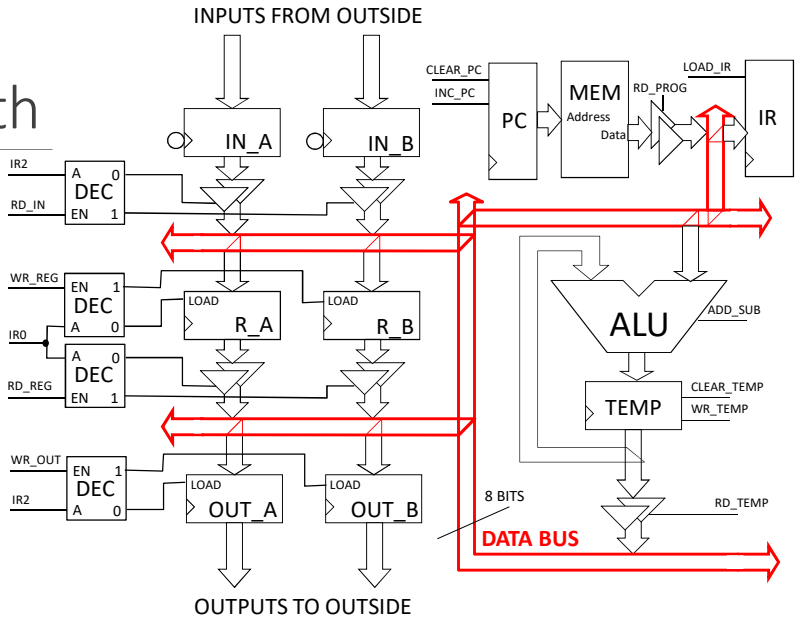
Revisiting how to implement combinatorial functions?

- Simplification using Karnaugh maps or Quine-McCluskey methods
- Direct implementation:
  - Using decoders + additional logic
  - Using multiplexers
  - Using memories

A simple microcontroller



The datapath

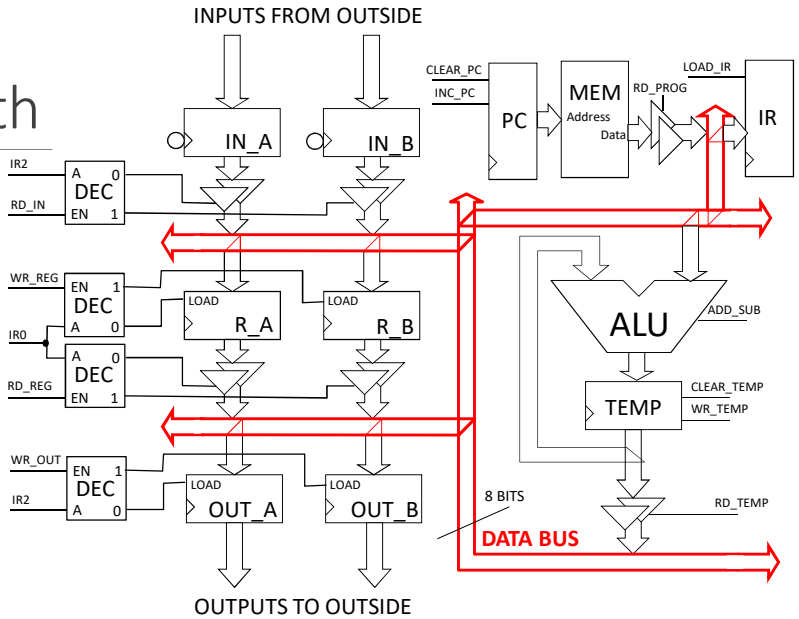


Instruction set

IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0	Instructions
0	0	0	0	-	-	-	-	**HALT**
0	0	0	1	-	-	-	-	**CLEAR** TEMP
0	0	1	0	-	n	-	m	**IN** FROM n TO m
0	0	1	1	-	-	-	m	**STORE** TEMP TO m
0	1	0	0	-	-	-	m	**ADD** m TO TEMP
0	1	0	1	-	-	-	m	**SUB** m TO TEMP
0	1	1	0	-	-	-	m	**LOAD** TEMP WITH m
0	1	1	1	-	n	-	m	**OUT** FROM M TO n

The datapath

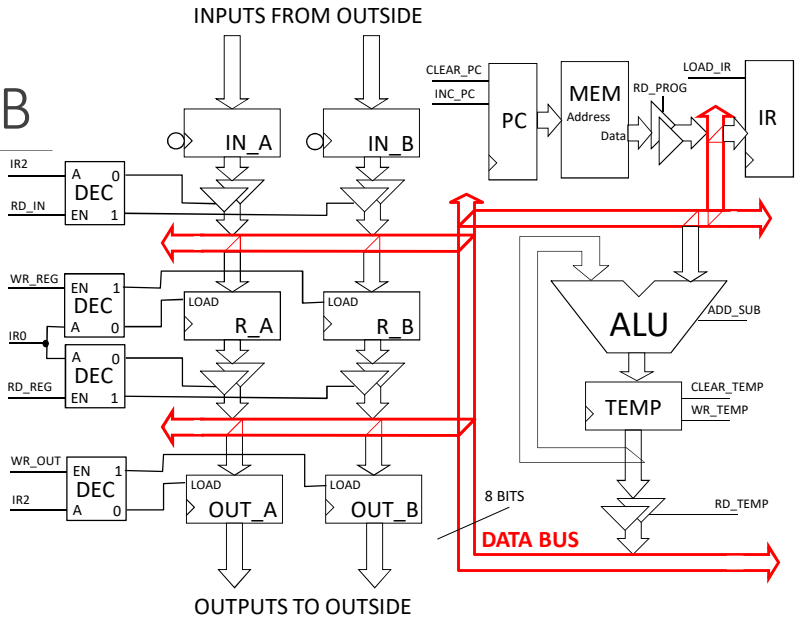
Instructions
HALT
CLEAR TEMP
IN FROM n TO m
STORE TEMP TO m
ADD m TO TEMP
SUB m TO TEMP
LOAD TEMP WITH m
OUT FROM M TO n



OUT_A = 2*IN_A-IN_B

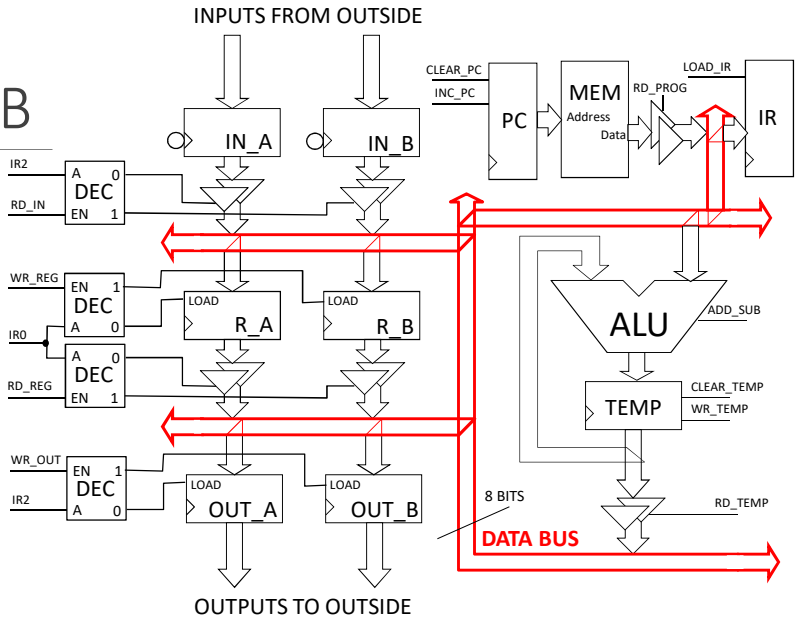
Instructions

HALT
CLEAR TEMP
IN FROM n TO m
STORE TEMP TO m
ADD m TO TEMP
SUB m TO TEMP
LOAD TEMP WITH m
OUT FROM m TO n



OUT_A = 2*IN_A-IN_B

IN FROM IN_A TO REG_A
LOAD TEMP WITH REG_A
ADD REG_A TO TEMP
IN FROM IN_B TO REG_A
SUB REG_A TO TEMP
STORE TEMP TO REG_A
OUT FROM REG_A TO OUT_A
HALT



OUT_A = 2*IN_A-IN_B

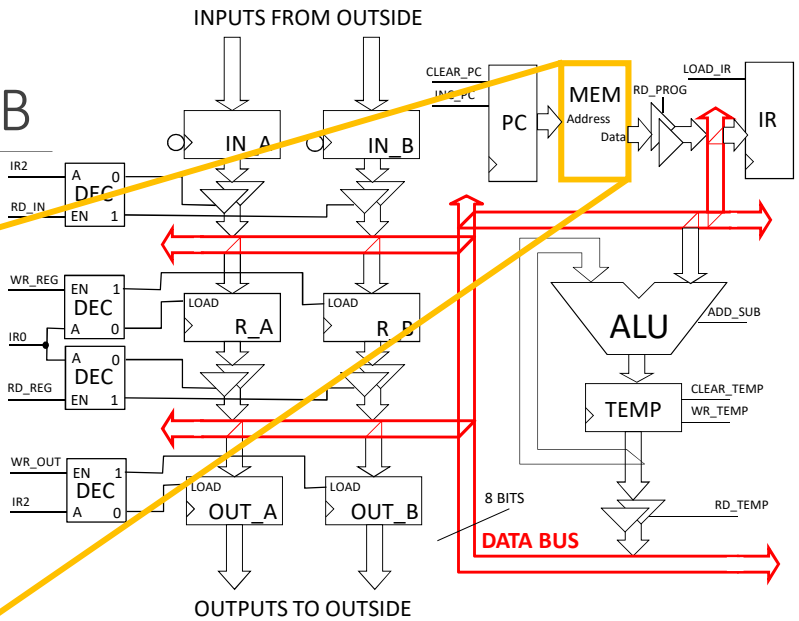
IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0	Instructions
0	0	0	0	-	-	-	-	**HALT**
0	0	0	1	-	-	-	-	**CLEAR** TEMP
0	0	1	0	-	n	-	m	**IN** FROM n TO m
0	0	1	1	-	-	-	m	**STORE** TEMP TO m
0	1	0	0	-	-	-	m	**ADD** m TO TEMP
0	1	0	1	-	-	-	m	**SUB** m TO TEMP
0	1	1	0	-	-	-	m	**LOAD** TEMP WITH m
0	1	1	1	-	n	-	m	**OUT** FROM M TO n

Instruction code

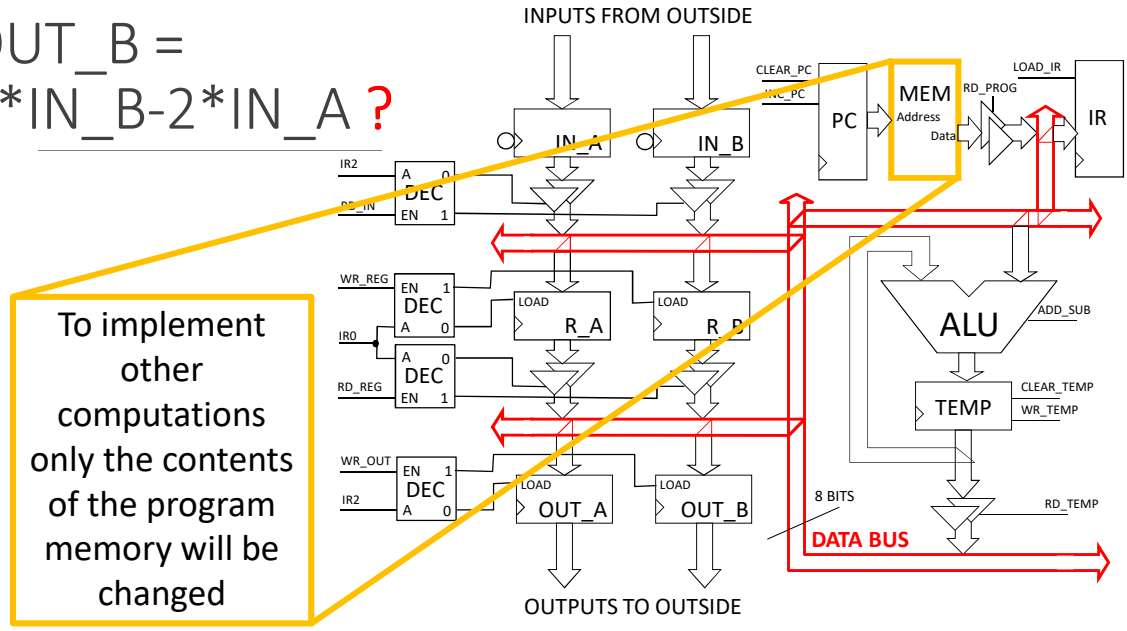
0 0 1 0 - 0 – 0	**IN** FROM IN_A TO REG_A
0 1 1 0 - - - 0	**LOAD** TEMP WITH REG_A
0 1 0 0 - - - 0	**ADD** REG_A TO TEMP
0 0 1 0 – 1 – 0	**IN** FROM IN_B TO REG_A
0 1 0 1 - - - 0	**SUB** REG_A TO TEMP
0 0 1 1 - - - 0	**STORE** TEMP TO REG_A
0 1 1 1 – 0 – 0	**OUT** FROM REG_A TO OUT_A
0 0 0 0 - - - -	**HALT**

OUT_A = 2*IN_A-IN_B

Address	Data
000	0 0 1 0 - 0 – 0
001	0 1 1 0 - - - 0
010	0 1 0 0 - - - 0
011	0 0 1 0 – 1 – 0
100	0 1 0 1 - - - 0
101	0 0 1 1 - - - 0
110	0 1 1 1 – 0 – 0
111	0 0 0 0 - - - -

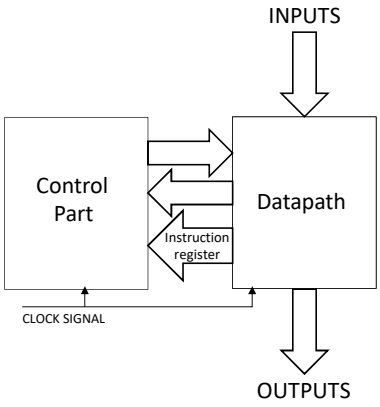


OUT_B = 3*IN_B-2*IN_A ?



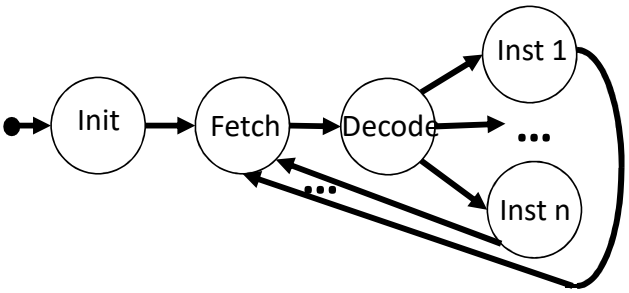
To implement other computations only the contents of the program memory will be changed

A simple microcontroller

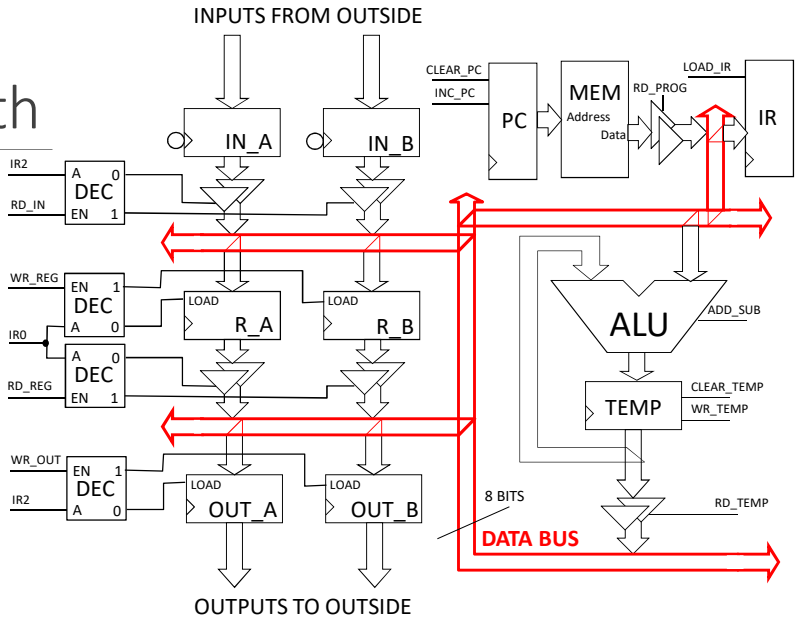


The control part

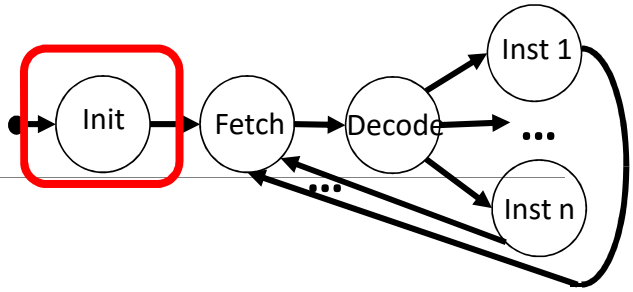
Fetch the instruction code
Decode &
Execution the instruction



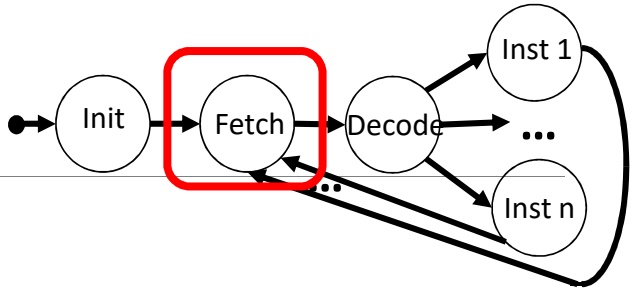
The datapath

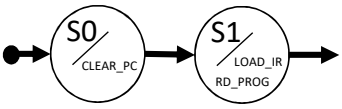


Control part - initialization

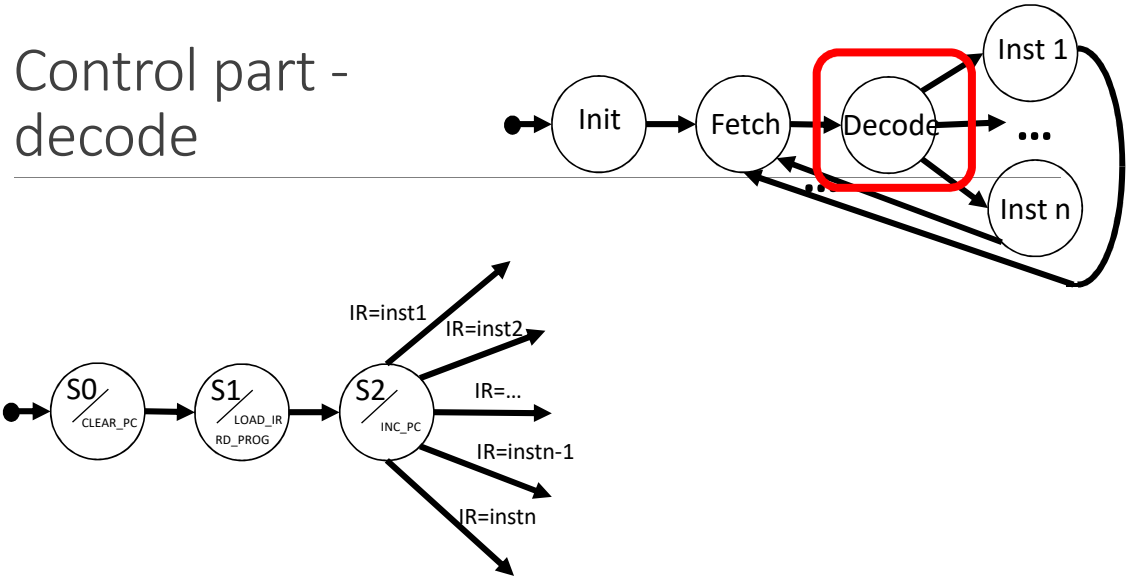


Control part - fetch

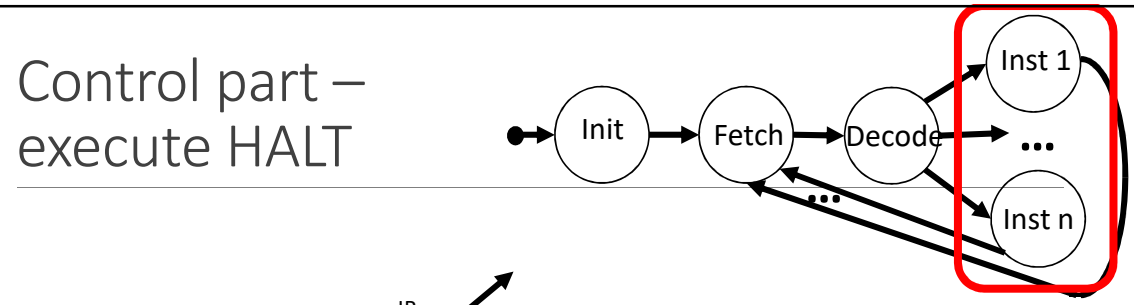


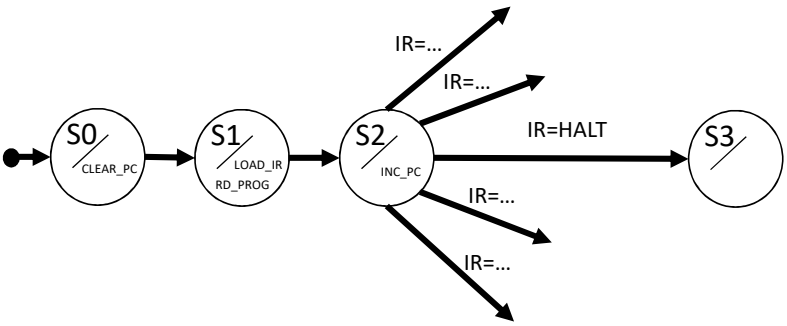


Control part - decode

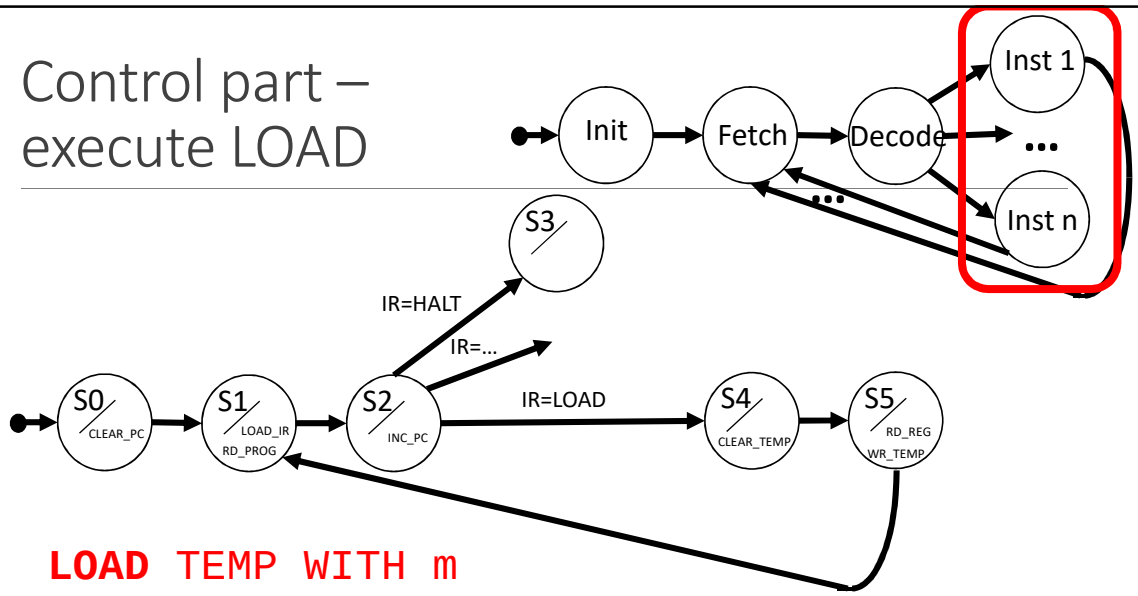


Control part – execute HALT



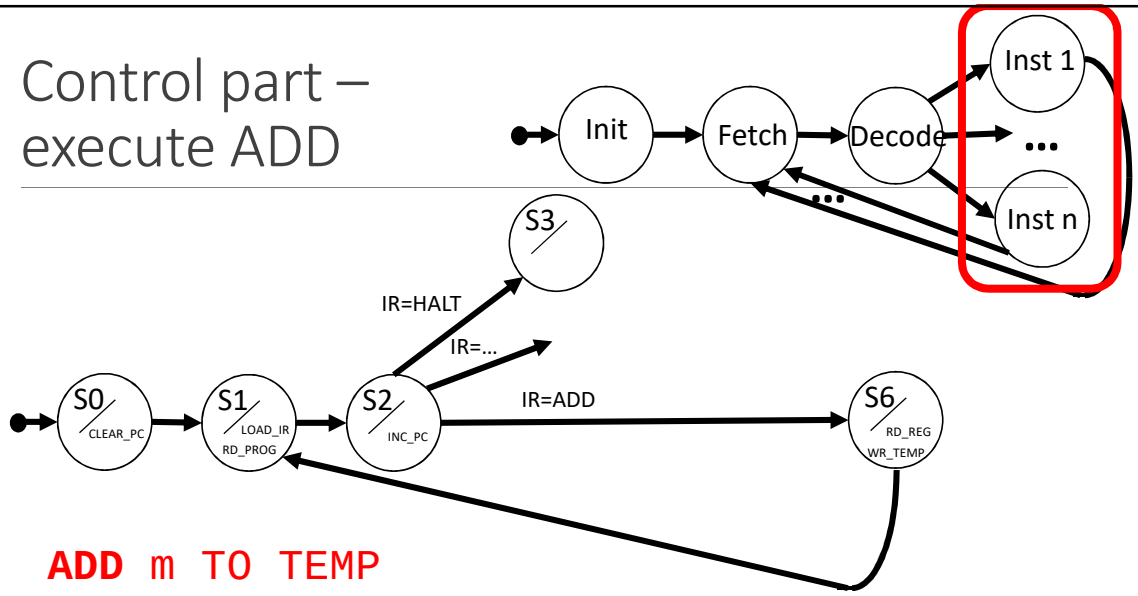


Control part – execute LOAD



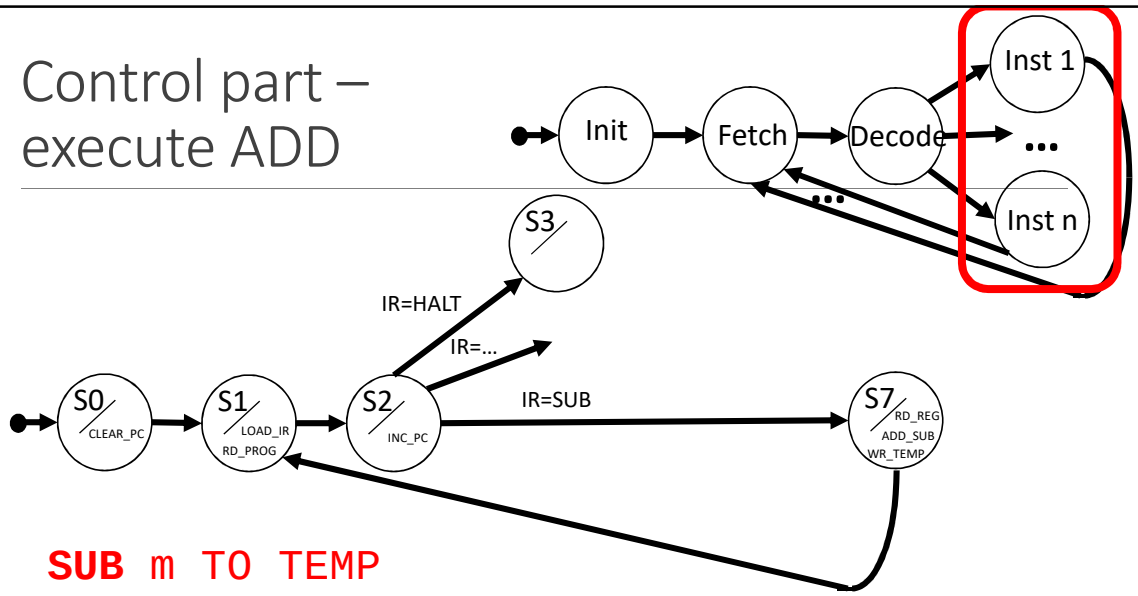
LOAD TEMP WITH m

Control part – execute ADD



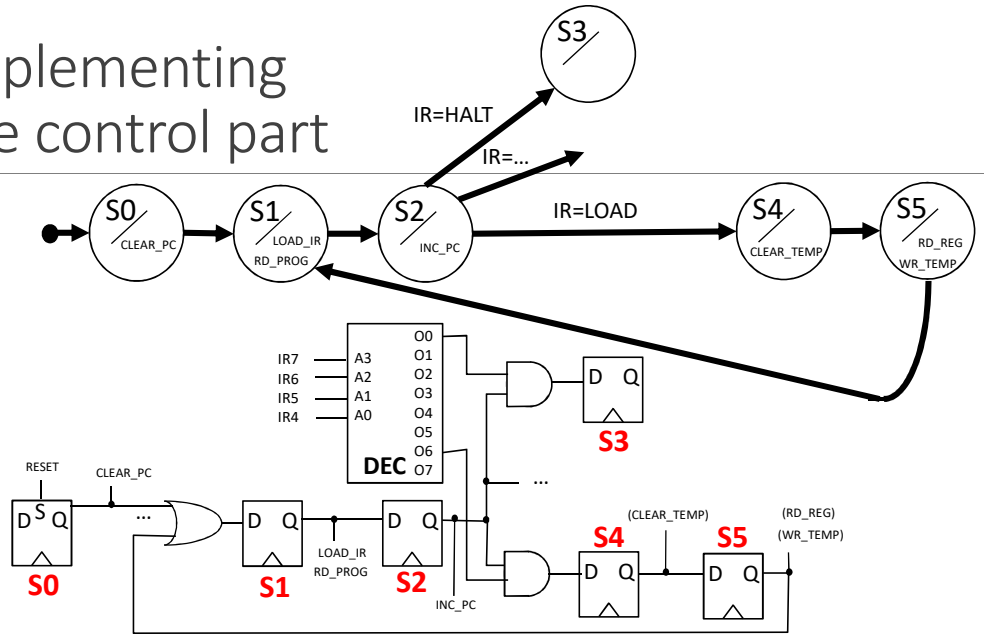
ADD m TO TEMP

Control part – execute ADD

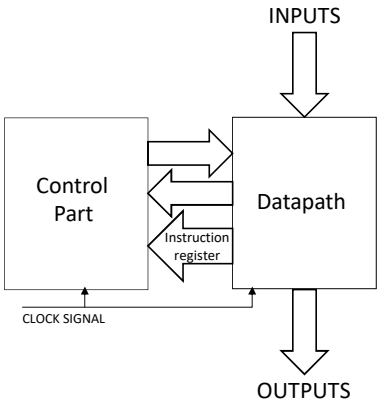


SUB m TO TEMP

Implementing the control part



A simple microcontroller



A simple microcontroller

Instructions

HALT
CLEAR TEMP
IN FROM n TO m
STORE TEMP TO m
ADD m TO TEMP
SUB m TO TEMP
LOAD TEMP WITH m
OUT FROM m TO n