EDA322
Digital Design

Lecture 4:

The ChAcc Processor & VHDL for regular structures

Ioannis Sourdis

Outline of Lecture 4

- Lab processor specifications
  - Datapath and control
  - Instruction Set
- Structural VHDL and tips for regular structures

The ChAcc Processor specifications

Angelos Arelakis

Ioannis Sourdis

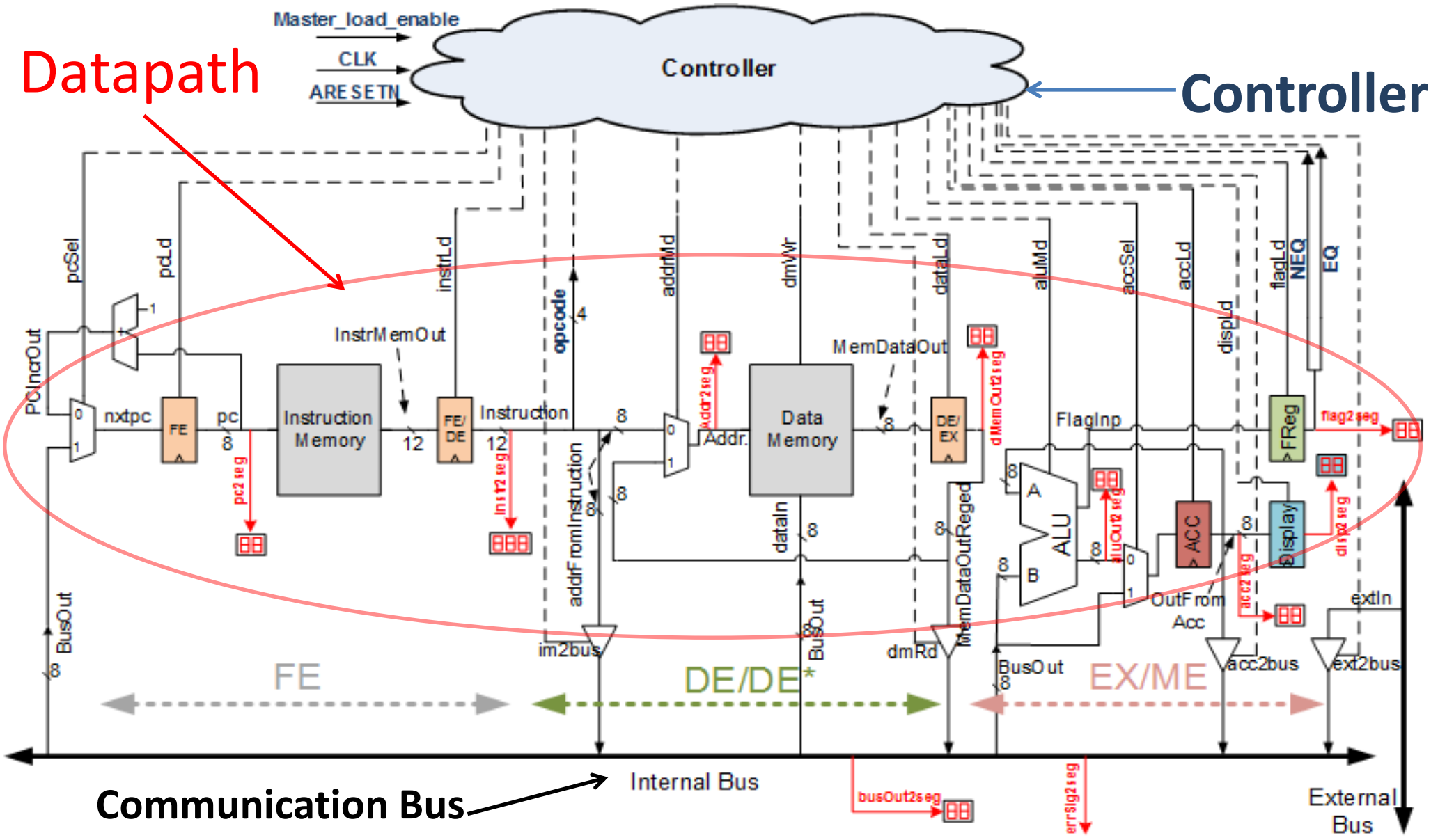
EDA322 lab processor

- Chalmers Accumulator processor (ChAcc processor)
- In EDA322 lab, ChAcc will be:
  - Implemented
  - Verified using real programs
  - Downloaded on FPGA (optional)
  - Evaluated in terms of performance, area and power dissipation (optional)
- Processor specifications are very important for the implementation of ChAcc.
  - Study the processor specification document (processor.pdf): contains all the details
  - Ask the lab assistants
- ChAcc is based on the lab processor of HY-120 course, Institute of Computer Science, FORTH, Greece

ChAcc processor - Overview

- ChAcc: Chalmers Accumulator processor
  - Only one register to save data, as opposed to modern processors
  - Keeps the result of the most recent operation
  - Consecutive operations are performed → result is accumulated in the accumulator
- ChAcc:
  - Simple to implement, but slow (about 2-4 clock cycles per executed instruction)
  - Runs a variety of programs
- ChAcc is an 8-bit processor:
  - operates on 8-bit data

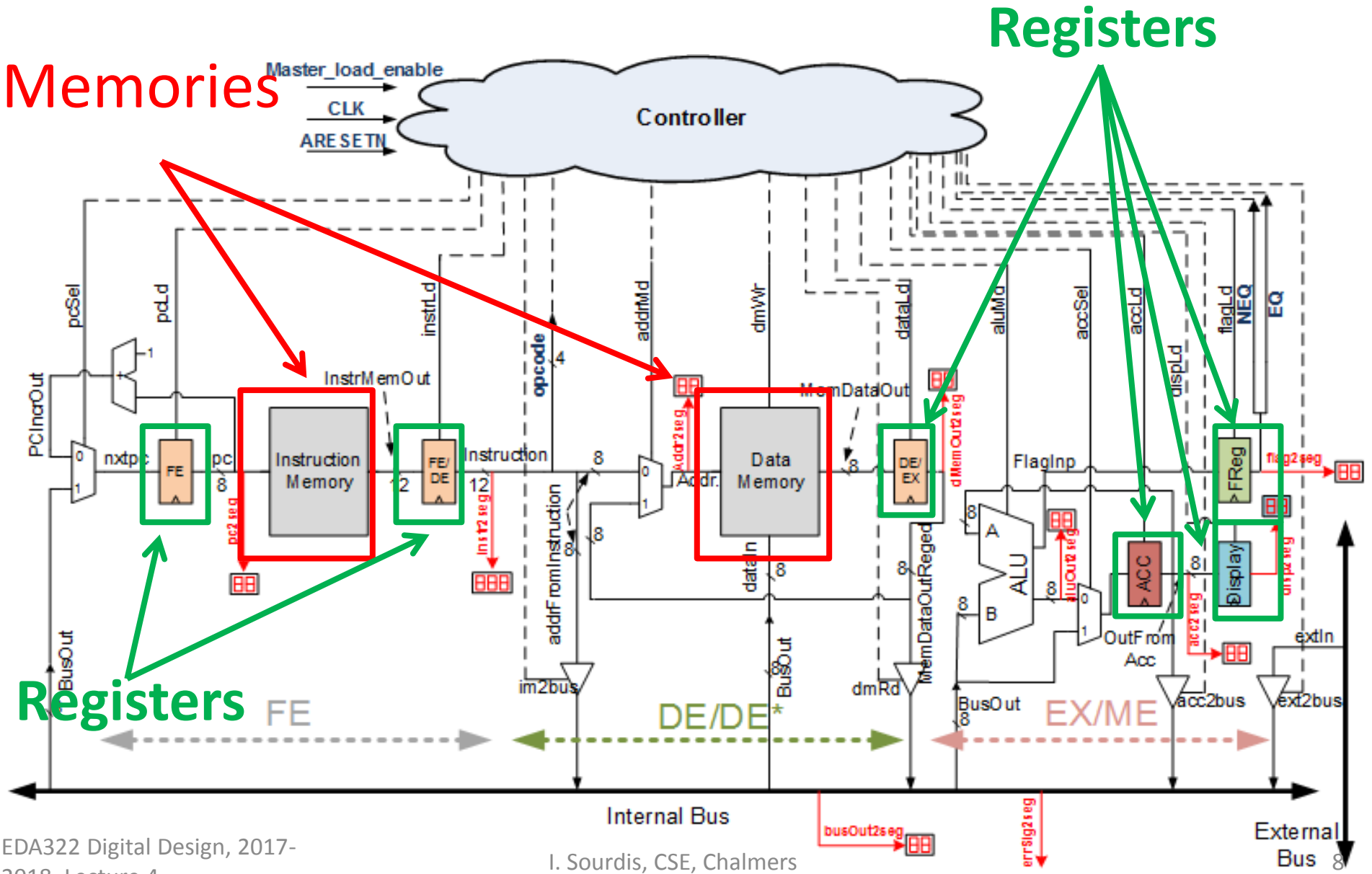
ChAcc processor - Overview



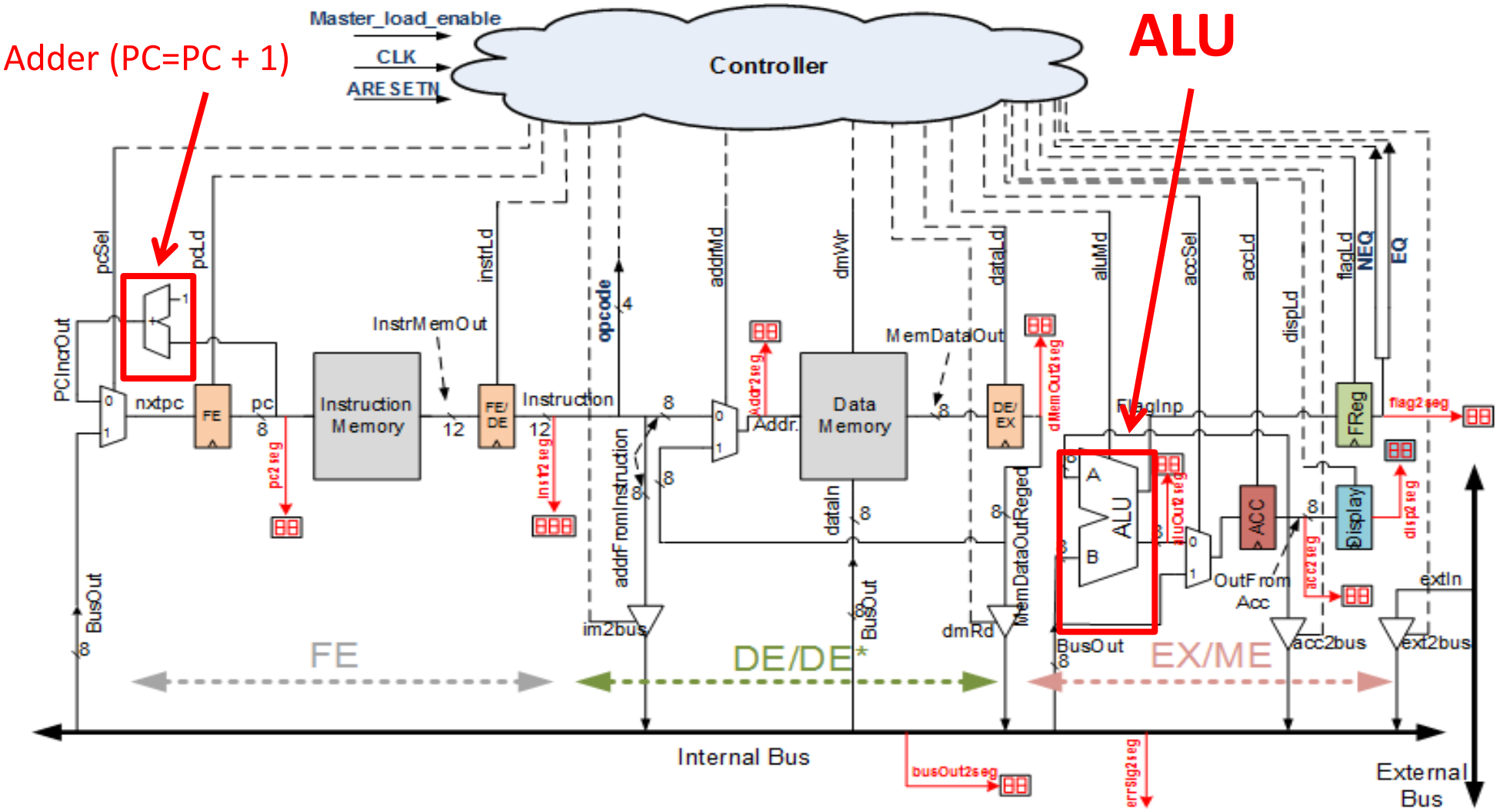
ChAcc processor - Overview

- **Datapath** is where data flow:
  - Are used or modified
  - Are read or written to memory
  - Are displayed in 7-segments
- Datapath consists of many components:
  - Memory, registers
  - ALU
  - 7-segments
  - …
- **Controller**: synchronizes processor's components and orchestrates operations
- **Communications bus** (or just bus): transfers data between different components (e.g., between ALU and memory) of the datapath.

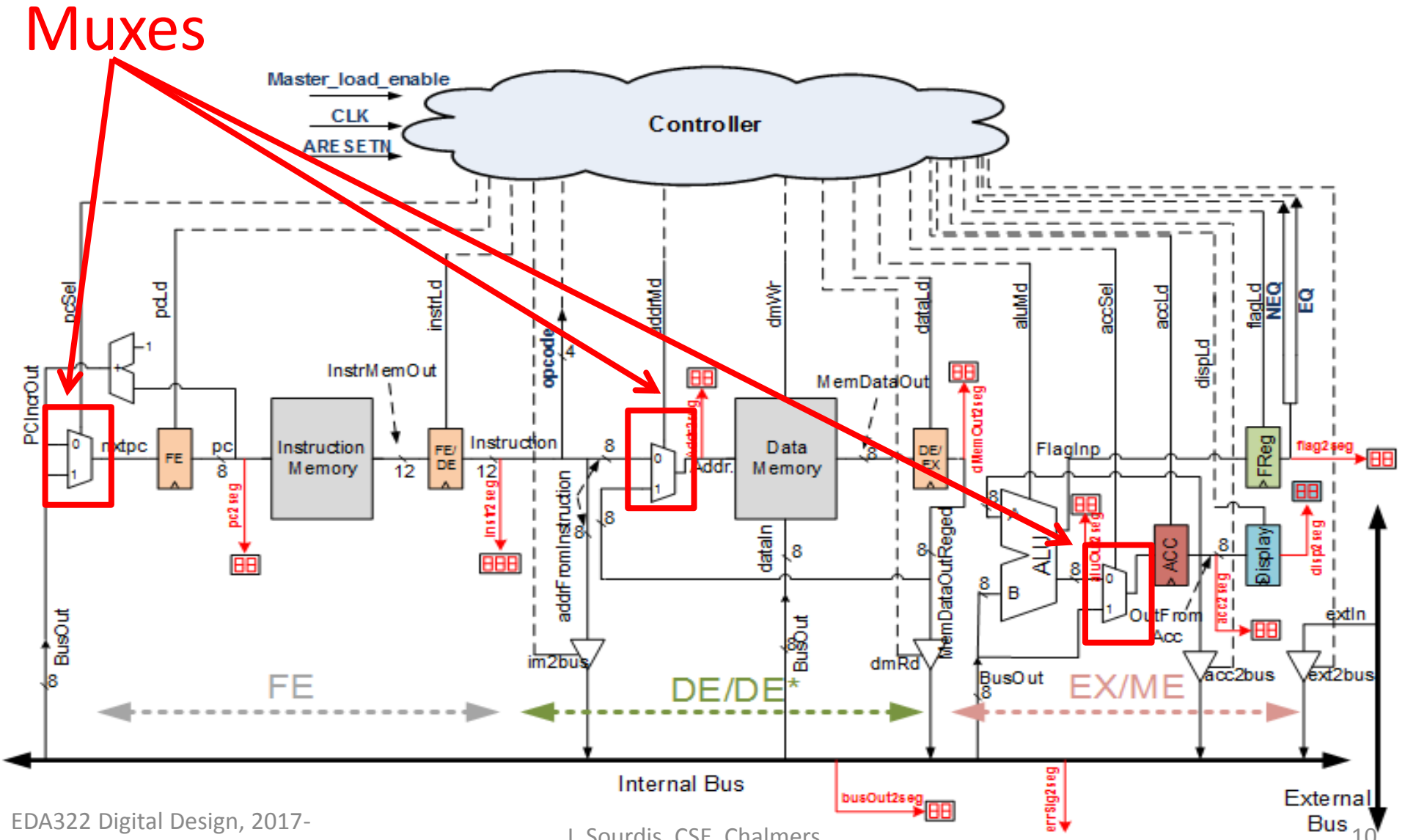
ChAcc processor – Datapath Storage elements



ChAcc processor – Datapath Arithmetic and Logic Units

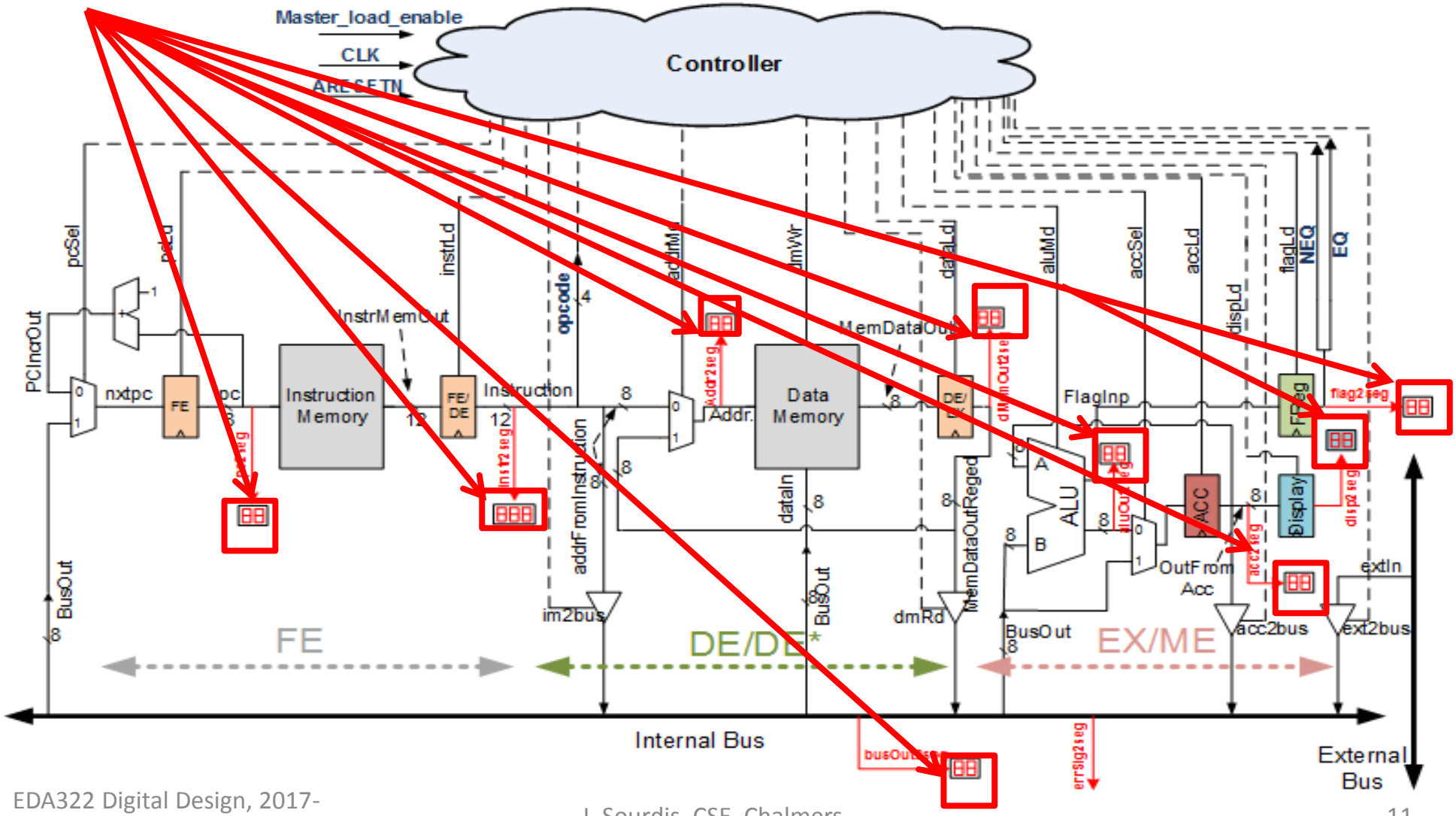


ChAcc processor – Datapath Muxes



ChAcc processor – Datapath Displays

Displays



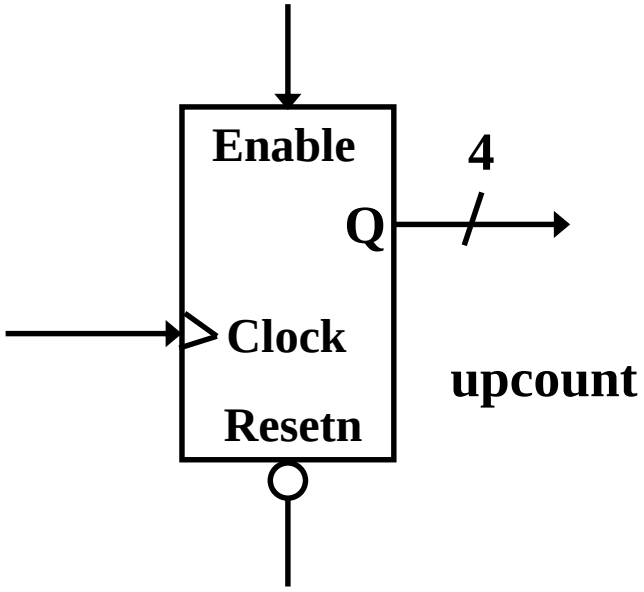
ChAcc processor – Datapath Storage elements (1)

- Memory and Registers

- Memory stores either data or program instructions
- **Harvard architecture**: memory is organized in two separate memories:
  - Instruction memory (IM): saves program instructions (12-bit); cannot be written at runtime (ROM)
  - Data memory (DM): saves data (8-bit); can be written at runtime
- Both memories are accessed using an 8-bit address → each has 256 entries (memory locations)

ChAcc processor – Datapath Storage elements (2)

- Registers: store the content when:
  - Clk is in the rising edge
  - When load enable is set



ChAcc processor – Datapath Storage elements (3)

Many types of registers in ChAcc:

- **ACC:** The main accumulator register
- **FReg:** Flag register keeps 4 1-bit flags (MSB to LSB order):
  - Ovf: overflow is set when an ALU operation results in overflow
  - NEQ: is set when the ALU operands are not equal
  - EQ: is set when the ALU operands are equal
  - Zero: is set when the ALU output is zero
- **Datapath registers (FE, FE/DE, DE/EX)**:
  - Keep information needed in the next stage
- **Display register**:
  - Keeps the data to be displayed in a particular display

ChAcc processor – Datapath
ALU (1)

- ALU = Arithmetic and Logic Unit
- ALU performs all the arithmetic and logic operations.
  - Add 2 operands
  - Sub 2 operands
  - AND of two operands
  - NOT of one operand
  - Compare two operands and determine if they are equal or not
- It doesn't support:
  - Multiplication
  - Division
  - Floating point operations
  - Square root, etc…

ChAcc processor – Datapath
ALU (2)

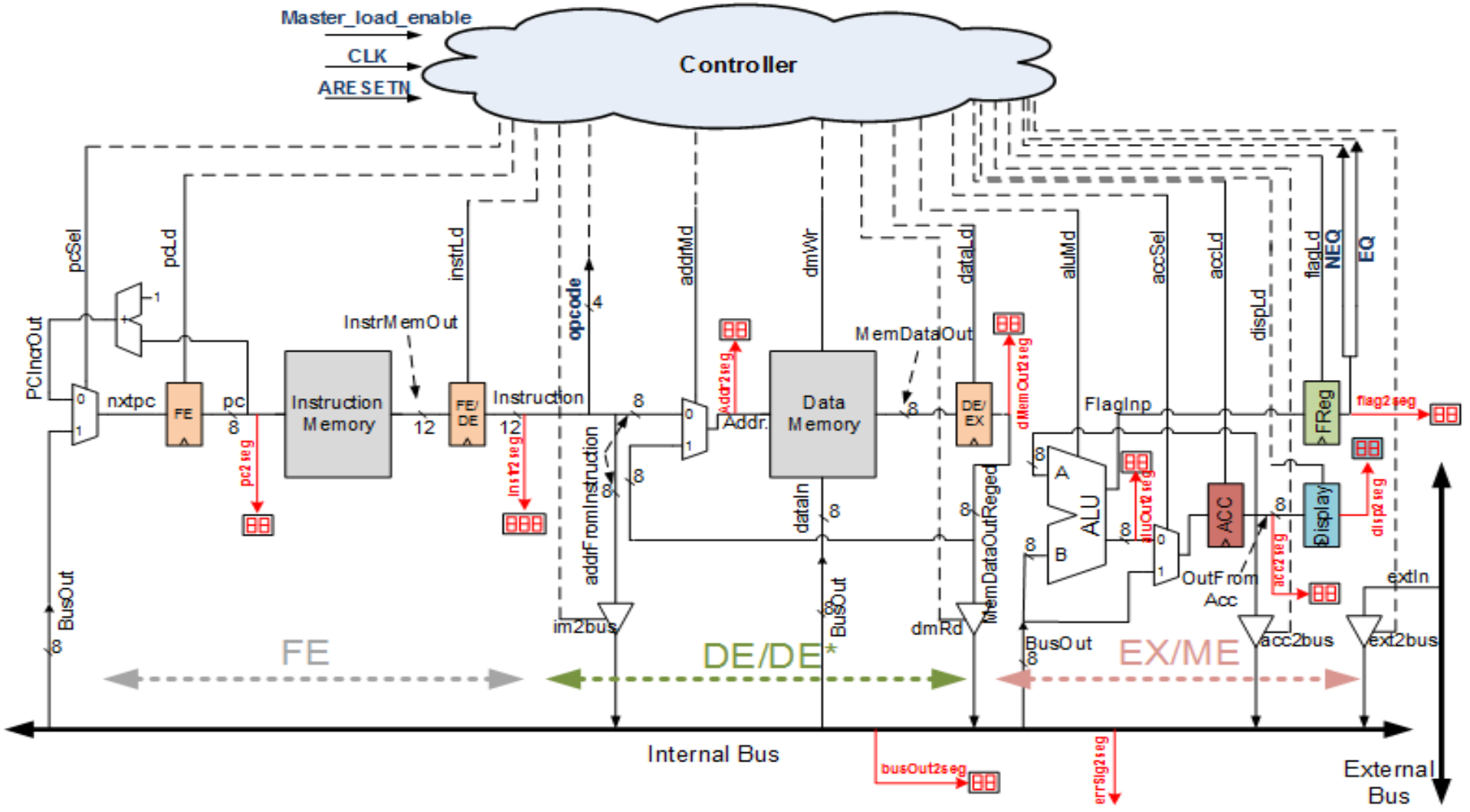
- ALU inputs:
  1. ALU_inA (8 bits): connected to ACC output
  2. ALU_inB (8 bits): given by the bus (normally the data memory output)
  3. Operation(2 bits): given by the controller and determines the current ALU operation

- ALU outputs:
  1. ALU_out (8 bits)
  2. Carry (1 bit): is set if the carry out of the adder is 1
  3. NotEq (1 bit): is set if ALU_inA and ALU_inB are not equal
  4. Eq (1 bit): is set if ALU_inA and ALU_inB are equal
  5. isOutZero (1 bit): set if ALU_out is zero

ChAcc processor – Datapath
Bus

- Bus: communicates data between different components
- Implemented with tri-states buffer
  - Need to be careful:
    - if more than one tri-state buffers are ON→ bus takes an undefined value
  - Alternative implementation:
    - Using multiplexers + extra logic you will find out

ChAcc processor – Datapath



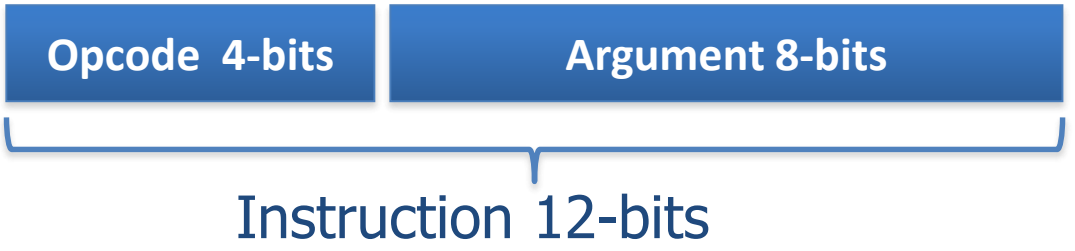
Keep the entity & signal names consistent!

- Signal names and their widths are given on the datapath.
  - If some names are not provided, set your own ones.
- In the lab assignments, for each unit, we provide:
  - The entity name (e.g. EDA322_processor is the name of the top-level design)
  - The names of the entity inputs/outputs
  - The widths of the entity inputs/outputs
- In the 6th lab assignment, we provide a wrapper that connects your implemented processor to the FPGA interface (pins, clk, switches, displays, etc)
- DON'T change any names/widths in the above specifications, otherwise the provided testbenches and the wrapper won't work!

ChAcc ISA and instruction format

ISA = Instruction Set Architecture: A set of the instructions that this processor can “recognize” and execute

- Instruction consists of two parts:
  - Opcode (4 bit): Unique ID of the instruction
  - Argument (8 bits): Can be used in different ways



Machine code	Instruction Name	Assembly language	Comment	Extra info
000000000000	No operation	NOOP	Do nothing	—
0010aaaaaaaa	Subtract	SU ACC, DM[Addr]	ACC = ACC - DataMem[Addr]	may set "Ovf", "Zero"
0001aaaaaaaa	Add	AD ACC, DM[Addr]	ACC = ACC + DataMem[Addr]	may set "Ovf", "Zero"
010000000000	Not	NT ACC	ACC = ACC'	may set "Zero"
0011aaaaaaaa	And	NA ACC, DM[Addr]	ACC = (ACC & DataMem[Addr])'	may set "Zero"
0101aaaaaaaa	Compare	CMP ACC, DM[Addr]	Compare ACC vs. DataMem[Addr]	set EQ, NEQ
0110aaaaaaaa	Load Byte	LB ACC, DM[Addr]	Load 8 byte value from location DataMem[Addr] into ACC	—
0111aaaaaaaa	Store Byte	SB DM[Addr], ACC	Store contents of ACC into location DataMem[Addr]	—
1000aaaaaaaa	Add Index	ADX ACC, DM[DM[Addr]]	ACC = ACC + DataMem[DataMem[Addr]]	may set "Ovf", "Zero"
1001aaaaaaaa	Load Byte Index	LBX ACC, DM[DM[Addr]]	ACC = DataMem[DataMem[Addr]]	—
1010aaaaaaaa	Store Byte Index	SBX DM[DM[Addr]], ACC	DataMem[DataMem[Addr]] = ACC	—
1011aaaaaaaa	Input	IN DM[Addr], IO_BUS	DataMem[Addr] = value at IO_BUS	—
1100aaaaaaaa	Jump	J addr	Execute next instruction @ PC = Addr	—
1101aaaaaaaa	Jump Not Equal	JNE addr	Jump if the corresponding flag NEQ is set	—
1110aaaaaaaa	Jump Equal	JEQ addr	Jump if the corresponding flag EQ is set	—
1111000000000	Display	DS	Move ACC to Diplay reg. (used for debugging)	—

Arithmetic and logic instructions

Memory instructions

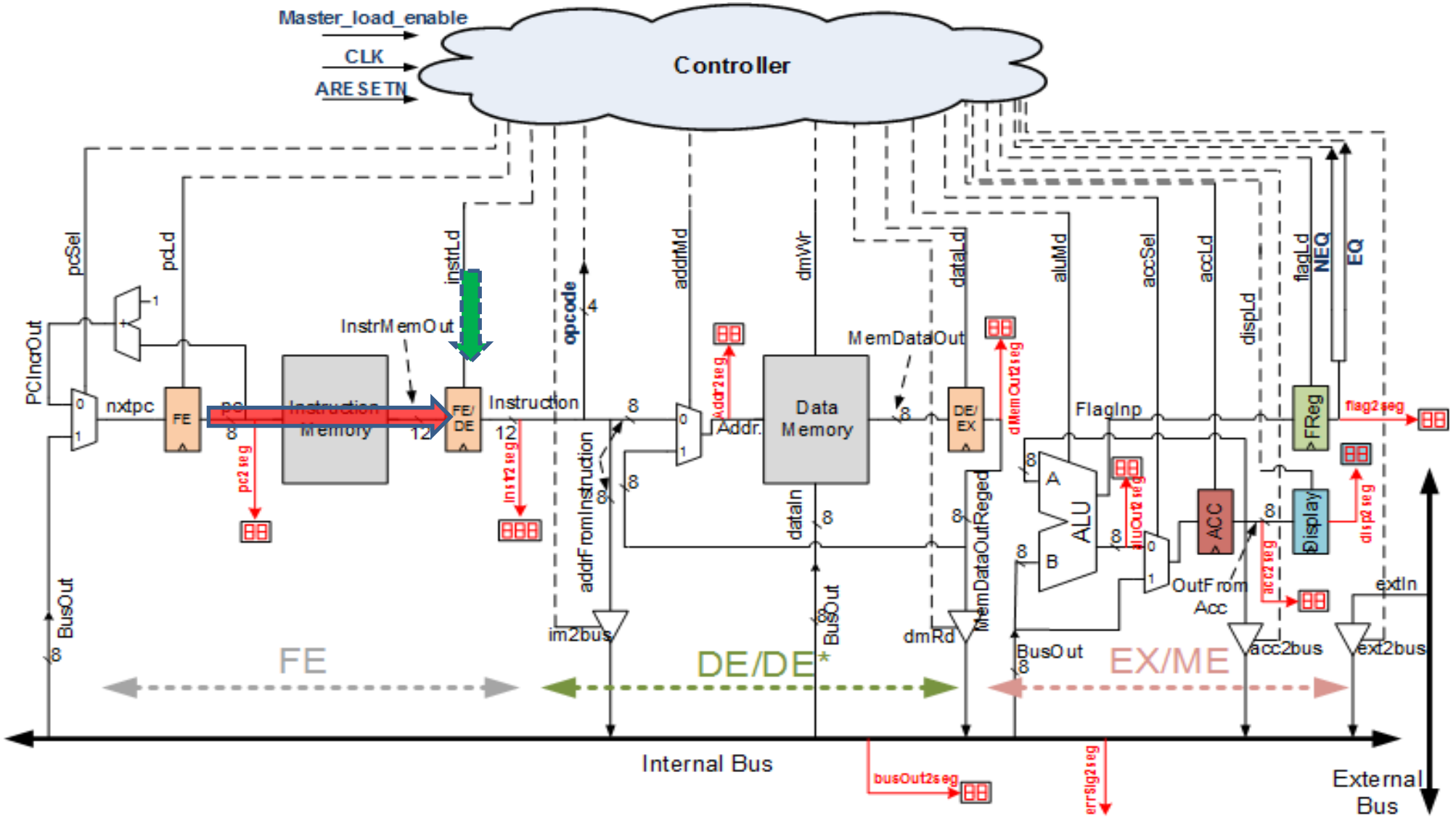
Branch instructions

*

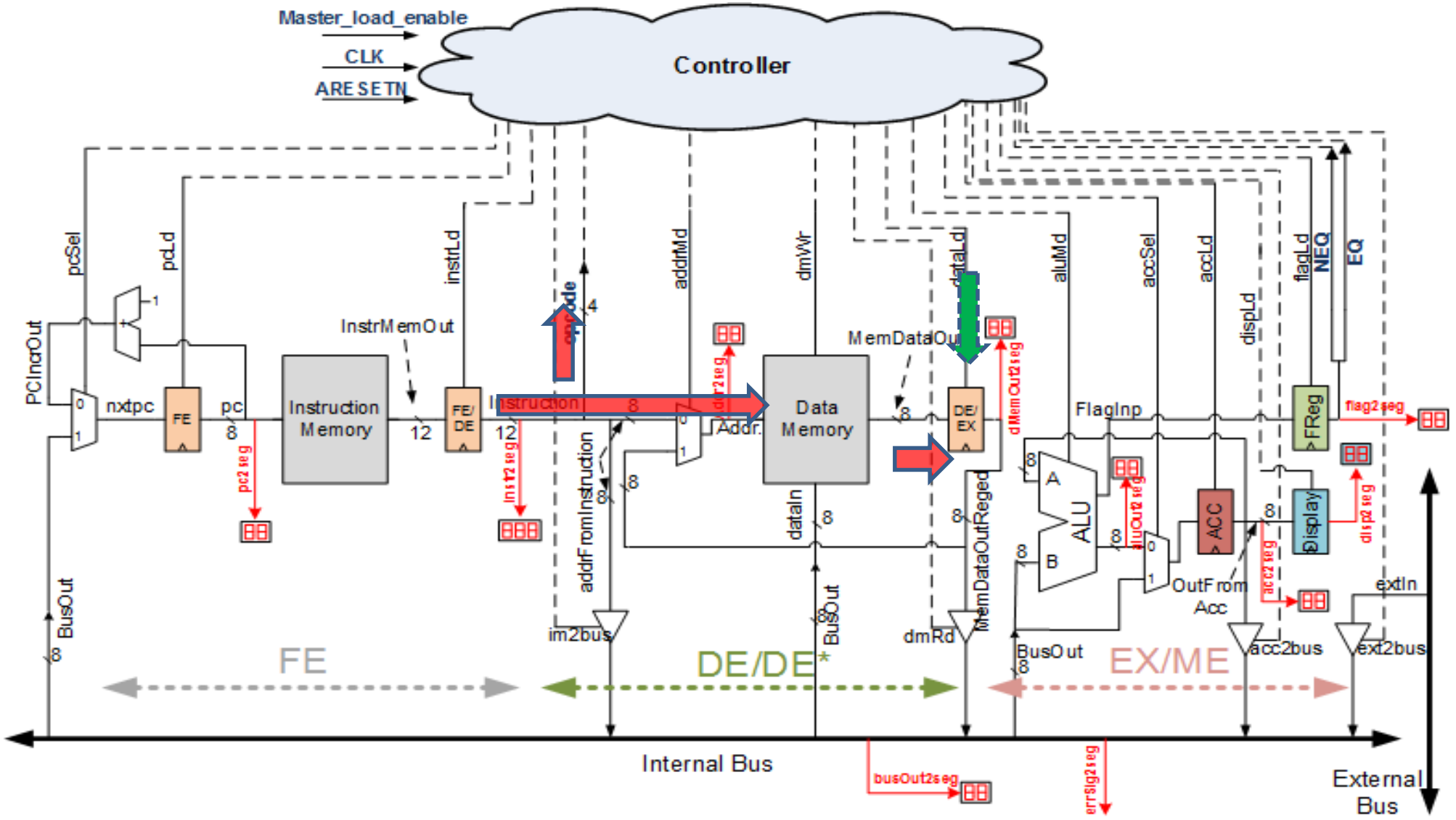
Arithmetic and logic instructions

Machine code	Instruction Name	Assembly language	Comment	Extra info
000000000000	No operation	NOOP	Do nothing	—
0010aaaaaaaa	Subtract	SU ACC, DM[Addr]	ACC = ACC - DataMem[Addr]	may set ”Ovf”, ”Zero”
0001aaaaaaaa	Add	AD ACC, DM[Addr]	ACC = ACC + DataMem[Addr]	may set ”Ovf”, ”Zero”
010000000000	NOT	NT ACC	ACC = ACC'	may set ”Zero”
0011aaaaaaaa	AND	NA ACC, DM[Addr]	ACC = ACC & DataMem[Addr]	may set ”Zero”
0101aaaaaaaa	Compare	CMP ACC, DM[Addr]	Compare ACC vs. DataMem[Addr]	set EQ, NEQ
1000aaaaaaaa	Add Index	ADX ACC, DM[DM[Addr]]	ACC = ACC + DataMem[DataMem[Addr]]	may set ”Ovf”, ”Zero”

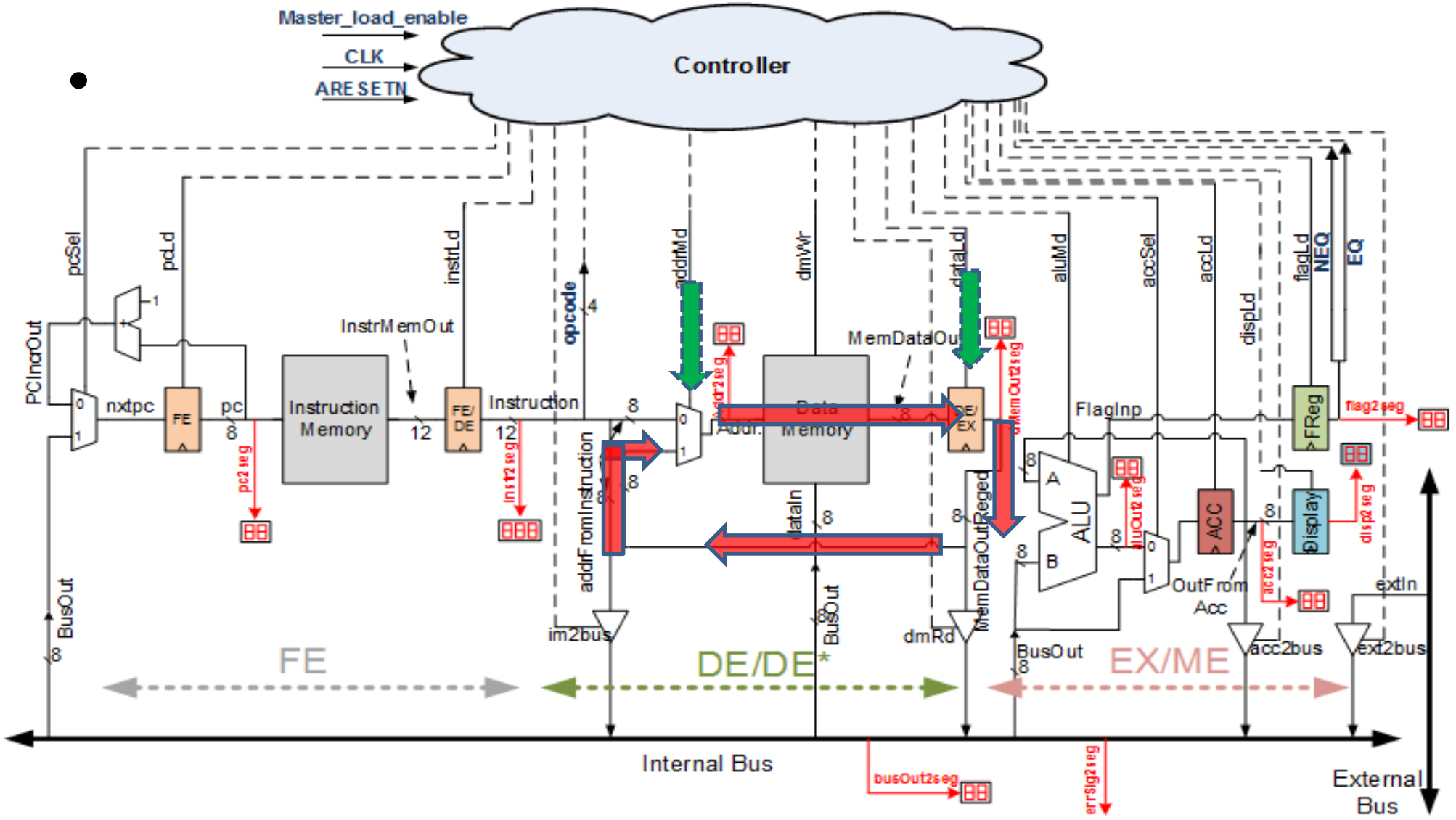
ADX ACC, DM[DM[Addr]]
Fetch



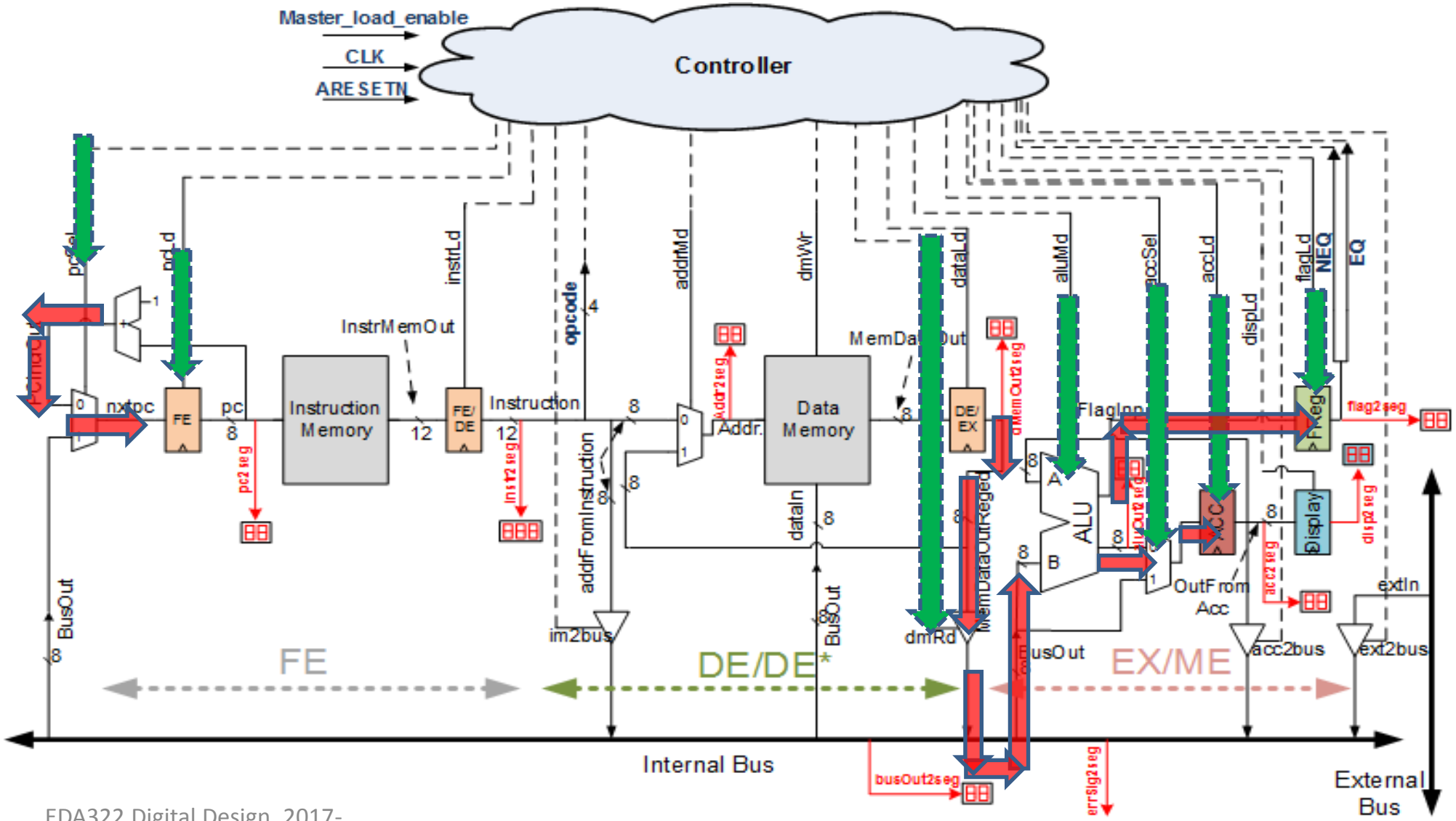
ADX ACC, DM[DM[Addr]] Decode



ADX ACC, DM[DM[Addr]] Decode*

- 

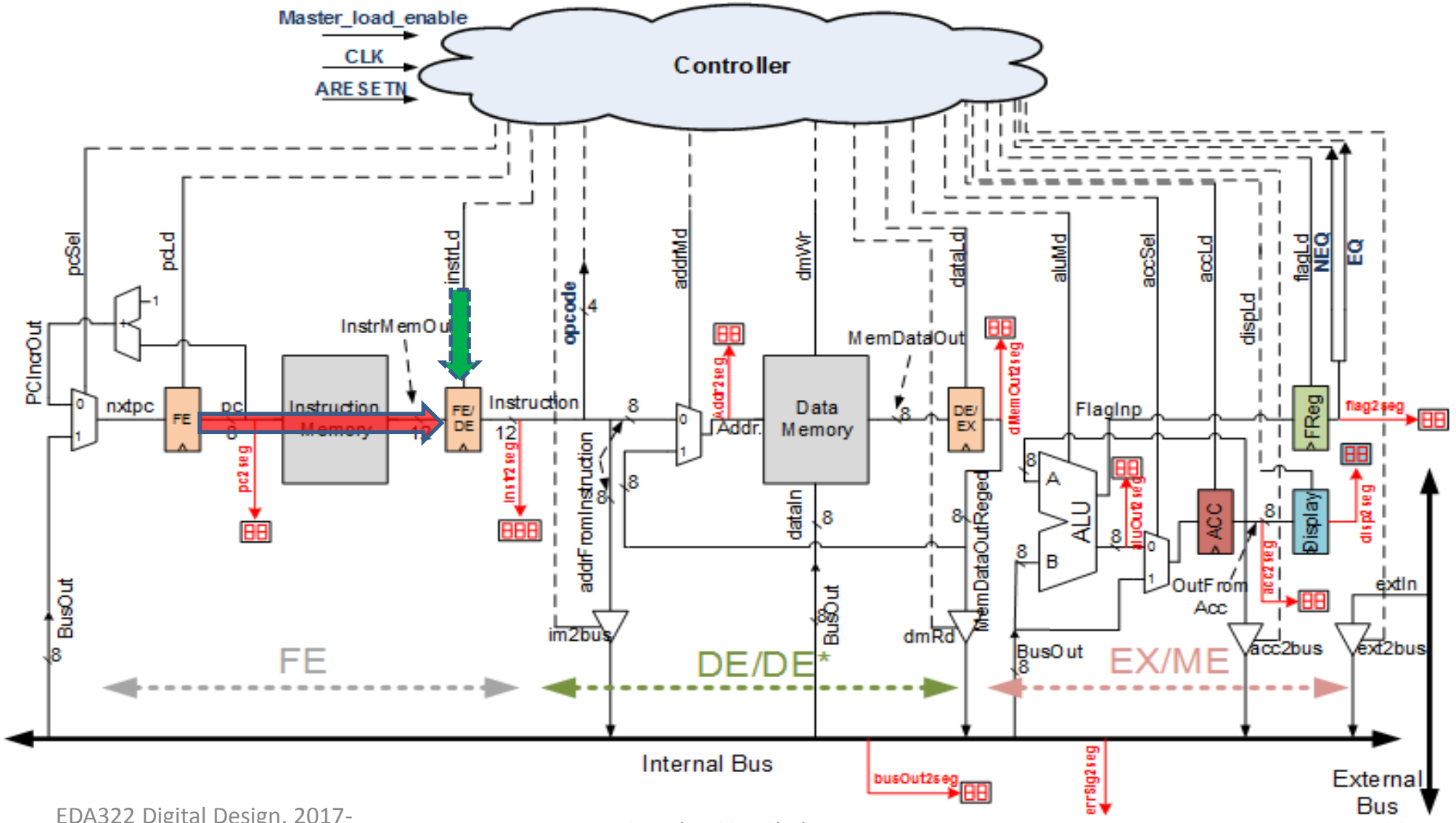
ADX ACC, DM[DM[Addr]]
Execute



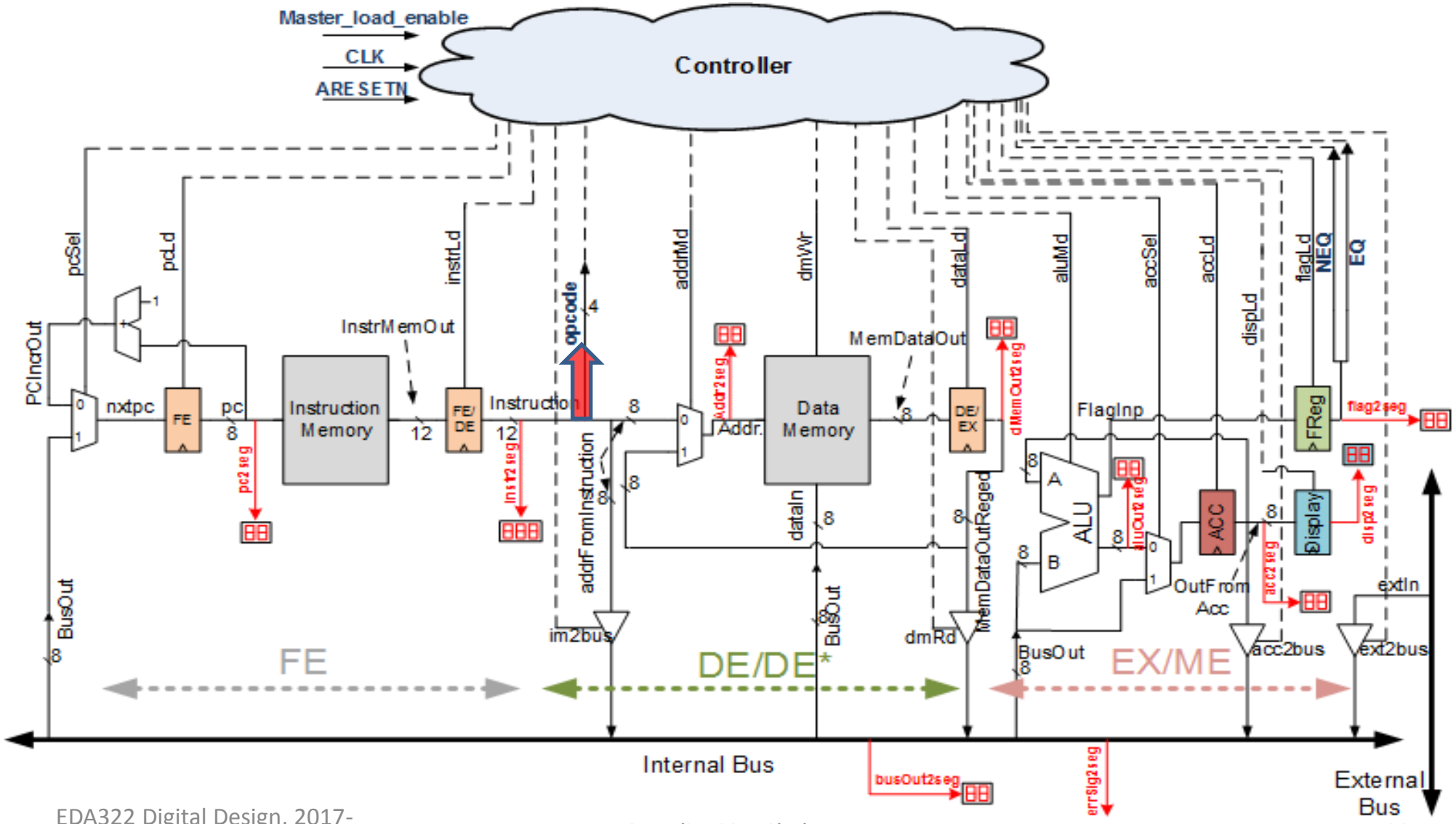
Memory instructions

Machine code	Instruction Name	Assembly language	Comment	Extra info
0110aaaaaaaa	Load Byte	LB ACC, DM[Addr]	Load 1 byte value from location DataMem[Addr] into ACC	—
0111aaaaaaaa	Store Byte	SB DM[Addr], ACC	Store contents of ACC into location DataMem[Addr]	—
1001aaaaaaaa	Load Byte Index	LBX ACC, DM[DM[Addr]]	ACC = DataMem[DataMem[Addr]]	—
1010aaaaaaaa	Store Byte Index	SBX DM[DM[Addr]], ACC	DataMem[DataMem[Addr]] = ACC	—
1011aaaaaaaa	Input	IN DM[Addr], IO_BUS	DataMem[Addr] = value at IO_BUS	—

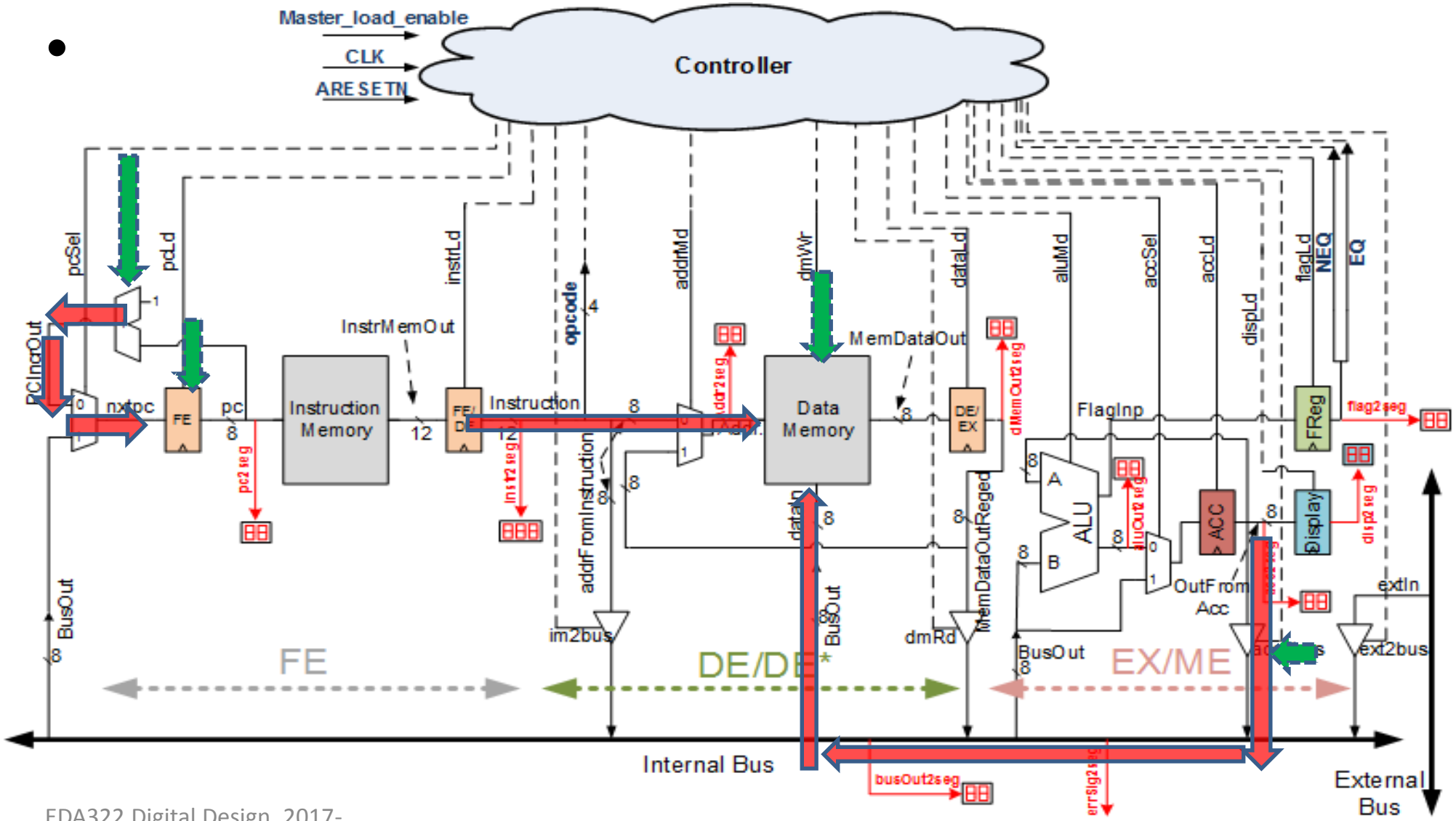
SB DM[Addr], ACC Fetch



SB DM[Addr], ACC Decode



SB DM[Addr], ACC Memory

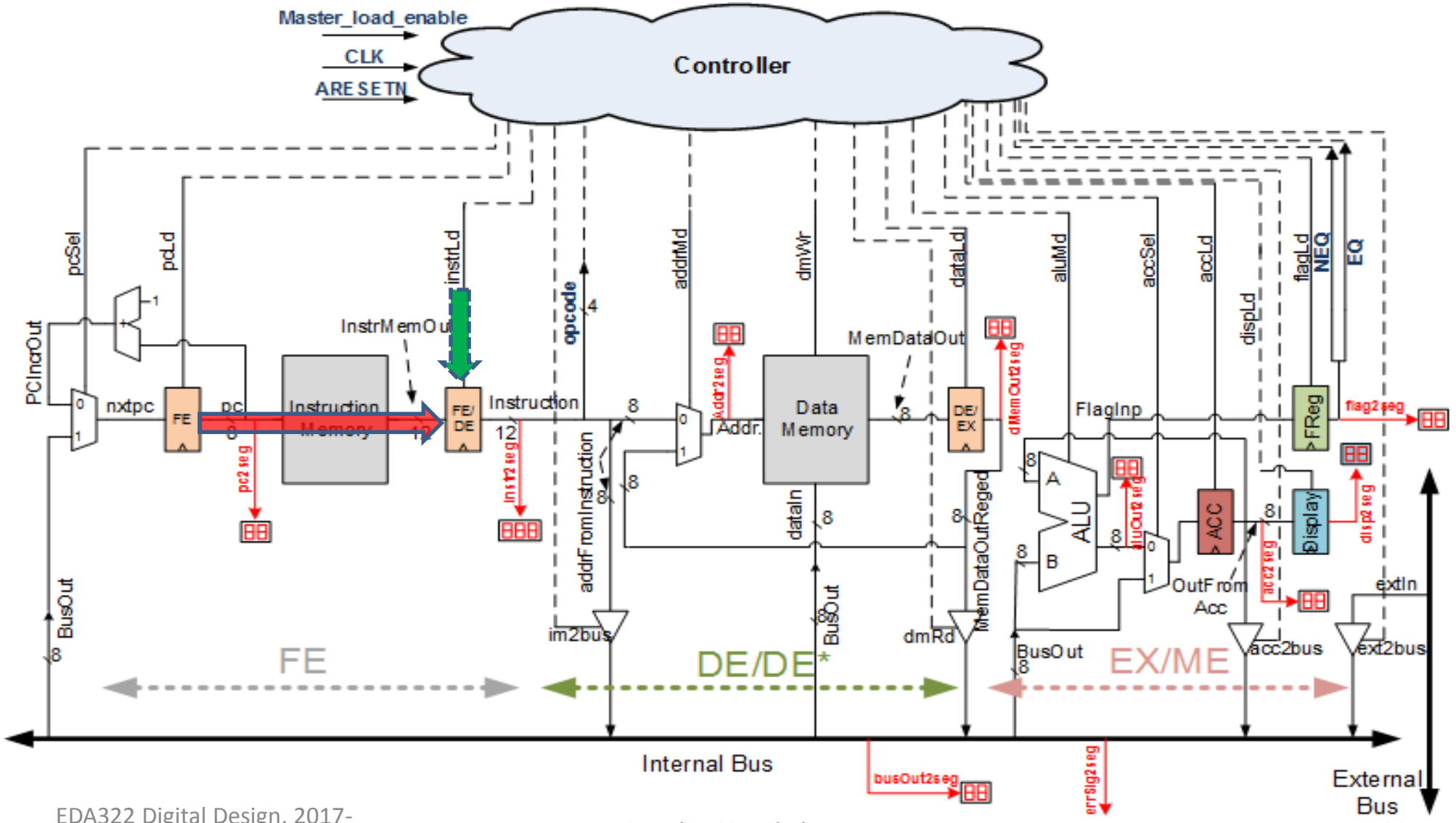
- 

Branch instructions ++

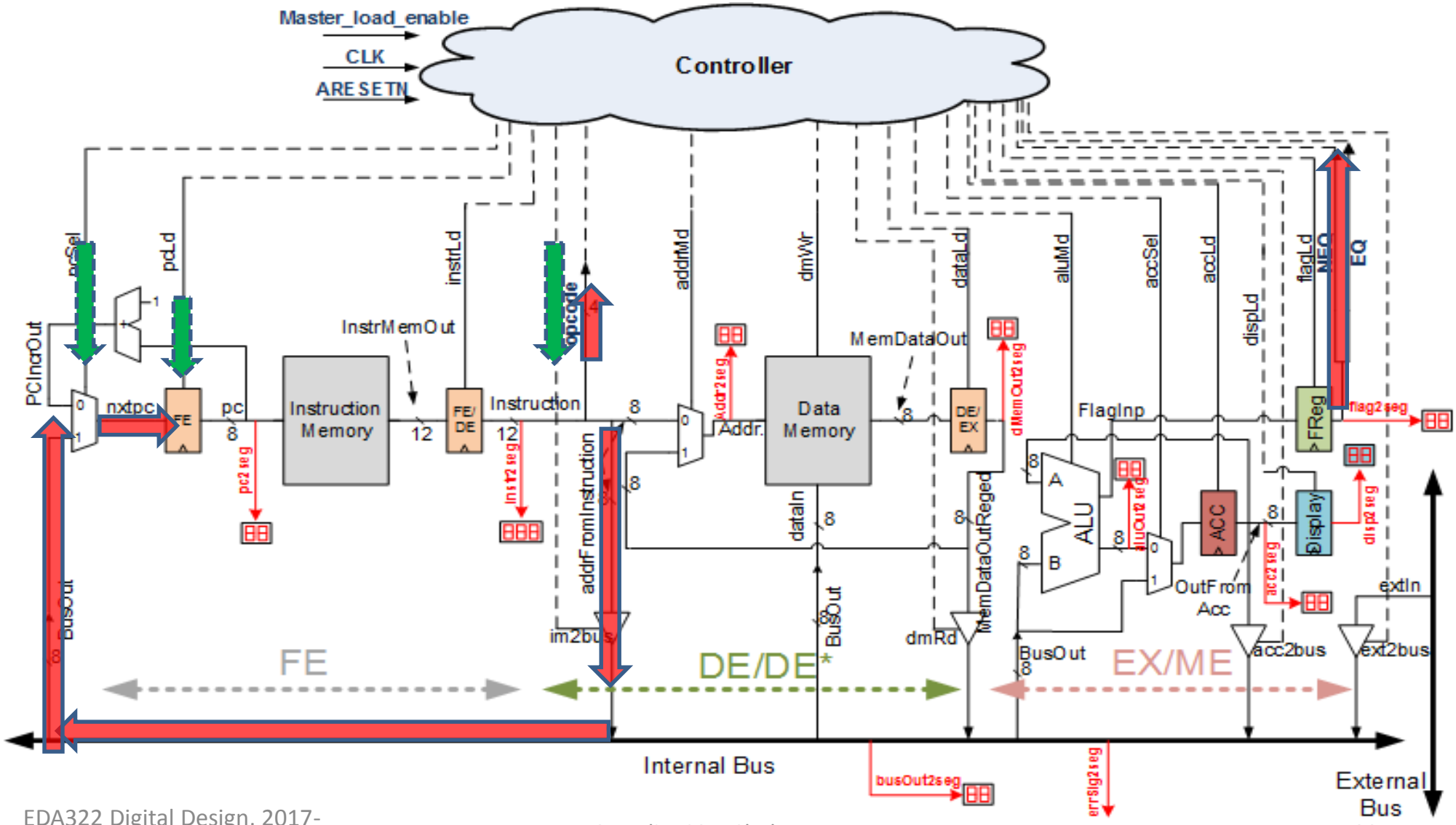
Machine code	Instruction Name	Assembly language	Comment	Extra info
1100aaaaaaaa	Jump	J address	Execute next instruction @ PC = address	—
1101aaaaaaaa	Jump Not Equal	JNE address	Jump if the corresponding flag NEQ is set	—
1110aaaaaaaa	Jump Equal	JEQ address	Jump if the corresponding flag EQ is set	—
111100000000	Display	DS	Move ACC to Diplay reg. (used for debugging)	—

For Debug

JEQ Addr Fetch



JEQ Addr Decode



ChAcc processor – ISA

- **Arithmetic and logic instructions:**
  - perform arithmetic and logic operations between operands on the ALU.
  - Data memory is accessed using instruction's address-field to retrieve the second ALU operand (one is ACC).
- **Memory instructions:**
  - The memory instructions access the data memory using the address-field of the instruction. Accesses:
    - read something from the data memory and save it to the ACC (*Load Byte*, *Load Byte Index*)
    - write the content of the ACC into the data memory (*Store Byte*, *Store Byte Index*)
    - write the data that come from the I/O bus into the data memory (*Input*)
- **Branch instructions:**
  - change the program flow by modifying the program counter (PC) based on a condition (*JE*, *JNE*) or unconditionally (*J*), by jumping to a particular address

ChAcc processor – ISA

- Other instructions:
  - NOOP: keeps processor idle by doing nothing: adding DM[0] (is always 0) to the ACC
  - Display: displays the content of ACC register → useful for debugging

- ADX, LBX and SBX are special instructions:
  - they access the data memory twice. Mentioned as Index instructions

Index instructions

- **A** is an array of 10 1-byte numbers
- The data memory locations 100 to 109 have been allocated to save the array elements)

Pseudocode:

```
...
x=1;
for (i=0; i!=10; i++)
{
  A[i] += x;
}
...
```

Assembly:

```
# Initialize some memory locations
1. IN DM[1], IO_BUS      # Write 1 from IO_BUS to DM[1];  x= 1 the value to add to the A[i]
2. IN DM[2], IO_BUS      # Write 10 from IO_BUS to DM[2]; 10 --> Max loop iteration
3. IN DM[3], IO_BUS      # Write 1 from IO_BUS to DM[3]; 1--> loop iteration step
4. IN DM[5], IO_BUS      # Write 100 from IO_BUS to DM[5]; 100 → the location of A[0] in DM
5. IN DM[6], IO_BUS      # Write 0 from IO_BUS to DM[6]; Initialize the current loop iteration to 0
# Calculate new A[i]
6. LB ACC, DM[1]                    # Load x to ACC
7. ADX ACC, DM[DM[5]] # ACC = x + A[i]
8. SBX DM[DM[5]], ACC               # A[i] = ACC
# Calculate next address of A[i]
9. LB ACC, DM[3]                    # Load iteration step to ACC
10. AD ACC, DM[5]                   # Calculate next address of A[i]
11. SB DM[5], ACC        # Save ACC (new calculated address) back to DM[5]
# Increment the iteration counter (i) and check if done
12. LB ACC, DM[3]        # Load iteration step to ACC
13. AD ACC, DM[6]        # Calculate new iteration
14. SB DM[6], ACC        # Save iteration counter back to DM[6]
15. CMP ACC, DM[2]       # Compare current iteration (@ACC) to max iteration (10)
16. JNE 6     # Evaluate NEQ flag and if it was set PC=6 (instr #6)
17. ...
```

The array is stored in the DM locations 100-109

Initially DM[100-109] = 0,1,2,3,4,5,6,7,8,9

1st iteration (i=0) DM[5] = 100
2nd iteration (i=1) DM[5] = 101
3rd iteration (i=2) DM[5] = 102

At the end of the loop iteration the memory locations DM 100-109 will have 1,2,3,4,5,6,7,8,9,10 respectively

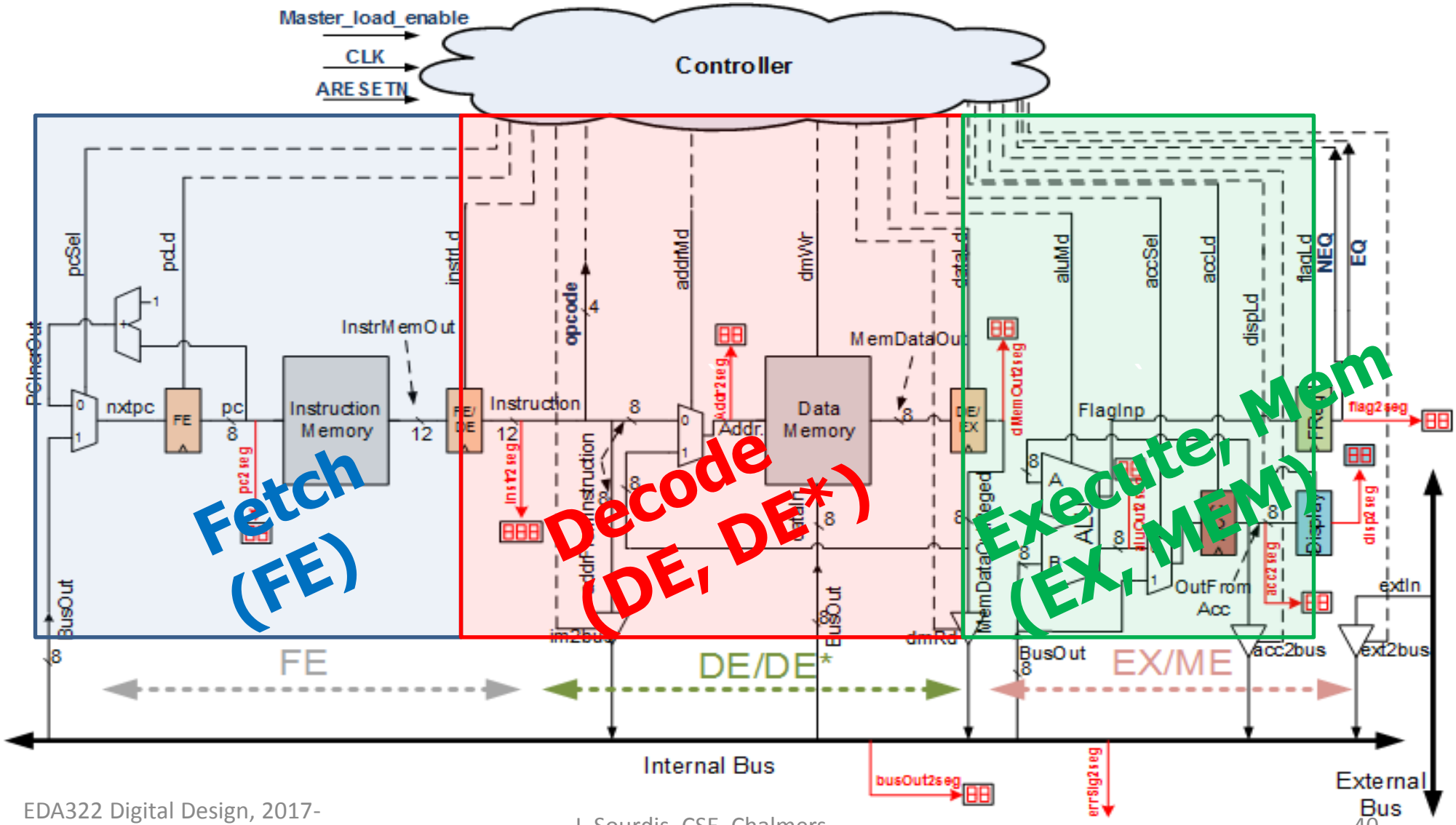
ChAcc processor – Global control signals

- Every sequential circuit is synchronized by a common signal → the global clock (CLK)
- Many components are initialized to zero when reset is on (ARESETn)
- Reset is asynchronous:
  - Effective when active and not dependent on the rising/falling edge of clock
  - Active when ‘0’

ChAcc processor – Controller

- Controller:
  - synchronizes the rest of the processor's units and orchestrates their operations
- Controller:
  - ➢ breaks the datapath into five stages
    - **Fetch, Decode, Decode*, Execute, Memory**
    - Instructions DON'T pass through all stages
      - All instructions pass through Fetch and Decode
  - Generates the control signals of the datapath units in these stages based on the executed instruction's opcode (part of the instruction)

ChAcc processor – Controller Datapath stages



opcode	detailed instr	FE	DE	DE*	EX	MEM	#stages
0000	AD ACC, DM[0]	y	y	n	y	n	3
0010	SU ACC, DM[Addr]	y	y	n	y	n	3
0001	AD ACC, DM[Addr]	y	y	n	y	n	3
0100	NT ACC, DM[Addr]	y	y	n	y	n	3
0011	AND ACC, DM[Addr]	y	y	n	y	n	3
0101	CMP ACC, DM[Addr]	y	y	n	y	n	3
0110	LB ACC, DM[Addr]	y	y	n	y	n	3
0111	SB DM[Addr], ACC	y	y	n	n	y	3
1000	ADX ACC, DM[DM[Addr]]	y	y	y	y	n	4
1001	LBX ACC, DM[DM[Addr]]	y	y	y	y	n	4
1010	SBX DM[DM[Addr]], ACC	y	y	y	n	y	4
1011	IN DM[Addr], IO_BUS	y	y	n	n	n	2
1100	J address	y	y	n	n	n	2
1101	JNE address, NEQ	y	y	n	n	n	2
1110	JEQ address, EQ	y	y	n	n	n	2
1111	DS	y	y	n	y	n	3

- EX: uses the ALU
- MEM: writes ACC into data memory

Extra DE stage for the Index instructions → they access the data memory 2x

All enter FE and DE

opcode	detailed instr	FE	DE	DE*	EX	MEM	#stages
0000	AD ACC, DM[0]	y	y	n	y	n	3
0010	SU ACC, DM[Addr]	y	y	n	y	n	3
0001	AD ACC, DM[Addr]	y	y	n	y	n	3
0100	NT ACC, DM[Addr]	y	y	n	y	n	3
0011	AND ACC, DM[Addr]	y	y	n	y	n	3
0101	CMP ACC, DM[Addr]	y	y	n	y	n	3
0110	LB ACC, DM[Addr]	y	y	n	y	n	3
0111	SB DM[Addr], ACC	y	y	n	n	y	3
1000	ADX ACC, DM[DM[Addr]]	y	y	y	y	n	4
1001	LBX ACC, DM[DM[Addr]]	y	y	y	y	n	4
1010	SBX DM[DM[Addr]], ACC	y	y	y	n	y	4
1011	IN DM[Addr], IO_BUS	y	y	n	n	n	2
1100	J P Offset	y	y	n	n	n	2
1101	JNE P Offset, NEQ	y	y	n	n	n	2
1110	JEQ P Offset, EQ	y	y	n	n	n	2
1111	DS	y	y	n	y	n	3

Variable number of stages per instruction

ChAcc processor – Controller Control signals

- Controller sets/resets the signals that control the datapath units
- Certain signals must be set/reset in every stage

X_Y: where **X** is the value of the control signal, **Y** is the stage it gets that value.
Stages: Fetch: **FE**, Decode: **DE**, Decode*: **DE***, Execute: **EX**, Memory: **ME**

opcode	pcSel	pcLd	instrLd	addrMd	dmWr	dataLd	flagLd	accSel	accLd	im2bus	dmRd	acc2bus	ext2bus	dispLd	aluMd
0000	0	1_EX	1	0	0	1_DE	1_EX	0	1_EX	0	1_EX	0	0	0	00
0010	0	1_EX	1	0	0	1_DE	1_EX	0	1_EX	0	1_EX	0	0	0	01
0001	0	1_ EX	1	0	0	1_DE	1_EX	0	1_EX	0	1_EX	0	0	0	00
0100	0	1_ EX	1	0	0	1_DE	1_EX	0	1_EX	0	1_EX	0	0	0	11
0011	0	1_ EX	1	0	0	1_DE	1_EX	0	1_EX	0	x	0	0	0	10
0101	0	1_ EX	1	0	0	1_DE	1_EX	0	0	0	1_EX	0	0	0	xx
0110	0	1_ EX	1	0	0	1_DE	0	1_EX	1_EX	0	1_EX	0	0	0	xx
0111	0	1_ME	1	0	1_ME	0	0	0	0	0	0	1	0	0	xx
1000	0	1_EX	1	1_ DE *	0	1_DE/ 1_DE*	1_EX	0	1_EX	0	1_EX	0	0	0	00
1001	0	1_EX	1	1_ DE *	0	1_DE/ 1_DE*	0	1_EX	1_EX	0	1_EX	0	0	0	xx
1010	0	1_ME	1	1_ ME	1_ME	1_DE	0	0	0	0	0	1	0	0	xx
1011	0	1_DE	1	0	1_DE	0	0	0	0	0	0	0	1	0	xx
1100	1_DE	1_DE	1	0	0	0	0	0	0	1	0	0	0	0	xx
1101	1_DE	1_DE	1	0	0	0	0	0	0	1	0	0	0	0	xx
1110	1_DE	1_DE	1	0	0	0	0	0	0	1	0	0	0	0	xx
1111	0	1_EX	1	0	0	0	0	0	0	0	0	0	0	1_EX	00

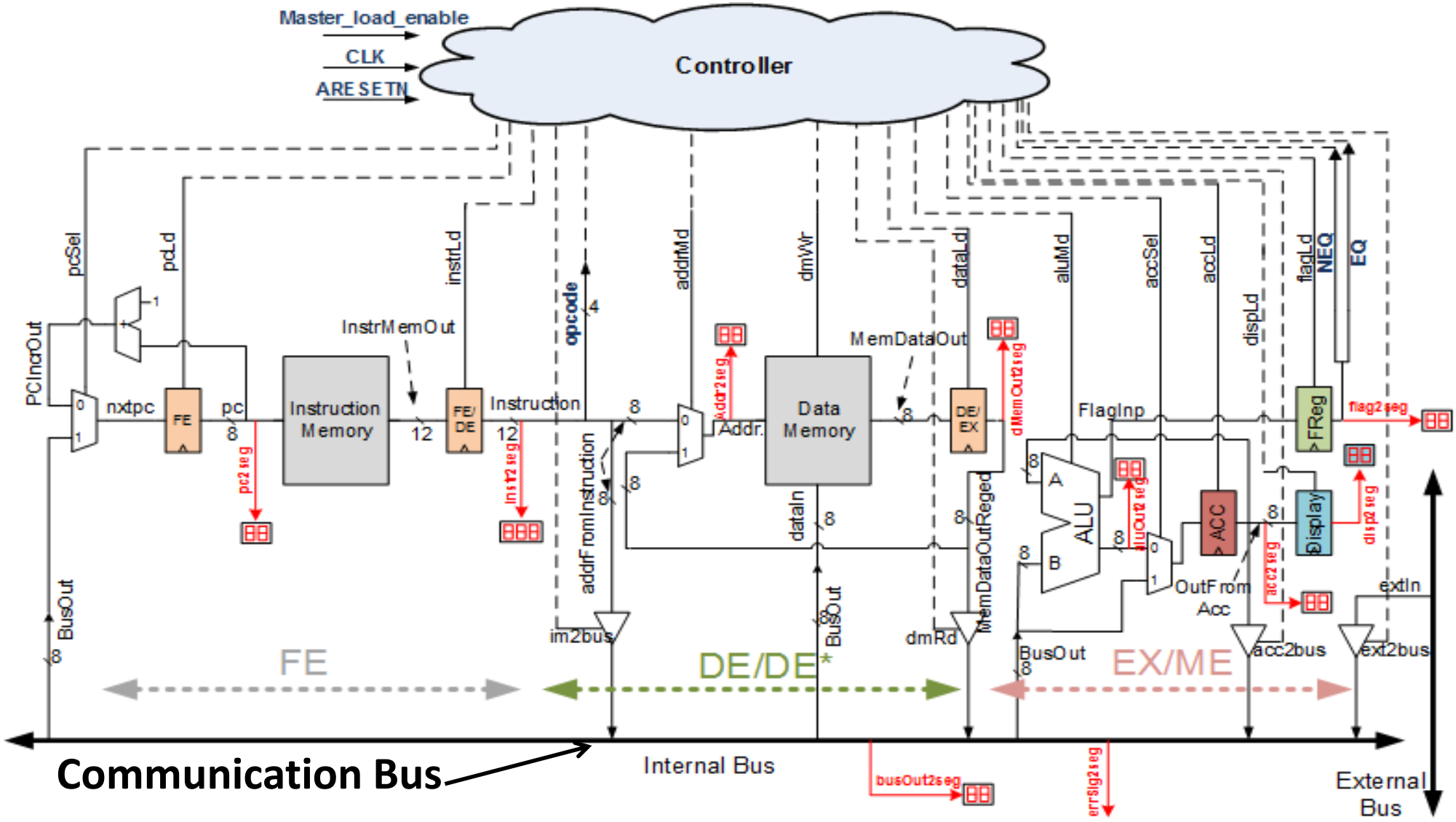
ChAcc processor – Controller Control signals

- **Opcode** (input): the opcode of the read instruction
- **EQ, NEQ** (input): used by the controller when determining the pcSel and pcLd
- **pcSel and pcLd:** control the logic that is relevant to the program counter (PC)
- **instrLd**: load enable of the register that keeps the read instruction
- **addrMd:** control the address input of the data memory. Can be:
  - The address field of the instruction (if normal instruction)
  - The data memory output (if Index instruction)

ChAcc processor – Controller Control signals

- **dmWr:** enables the write operation in the data memory
- **dmRd:** enables the read operation in the data memory
- **dataLd:** enables "DE/EX" to store the output of the data memory
- **flagLd:** enables "FReg" to store *Flags*
- **accSel:** controls the source of "ACC" register's input:
  - From ALU
  - From bus: If it is a load instruction
- **accLd:** enables the ACC register to save the input
- **dispLd:** load enable of the Display register
- **dmRd, acc2bus and ext2bus:** control signals of the bus
- **aluMd:** determines the operation of the ALU

ChAcc processor - Overview



ChAcc processor – Controller Control signals

- Registers have an additional enable signal (except for one of the previous load enable signals) → *master_load_enable*
- *master_load_enable*:
  - Controls the clock toggling manually
  - Controlled by the user through one FPGA switch
  - Affects FSMs and registers
- A register, in particular, is enabled if and only if:
  - Reset is disabled (ARESETN='1')
  - Positive edge of the clock
  - master_load_enable is set (master_load_enable='1')
  - load enable signal (given by the controller) is set

ChAcc processor – Controller

Controller:

- Is implemented as a Finite State Machine (FSM)
  - Instructions pass through a different number of stages and thus states
  - Different signals are set/reset based on the current stage and the executed instruction (opcode)
  - State transitions are enabled when *master_load_enable='1'*

Summary of the ChAcc processor

- Processor specifications are important for the correct implementation of ChAcc
- Always consult the processor.pdf document and this presentation, during the labs
- Ask the instructors, if you are unsure about something or if something is not clear
- Keep consistent with the processor specs:
  - entities, signal names or widths
  - files (memory initialization files, testbenches, wrapper, etc)

Quiz 6-1

http://b.socrative.com/student/#joinRoom

room number: 713113

- Q1: What does an "Add index" instruction do?
  - ADDs Acc register and a memory location and put the result to Acc
  - ADDsAcc register and a memory location and put the result to Mem
  - Uses the contains of a memory location as an address to access the memory again and ADD the contains of the second memory location to Acc. The result is put to Acc
  - Uses the contains of a memory location as an address to access the memory again and ADD the contains of the second memory location to Acc. The result is put to Memory
- Q2: In the ChAcc instruction set which are the stages that all instructions have in common and will pass through?
  - Fetch and Decode
  - Fetch and Execute
  - Fetch, Decode and Execute
  - All
- Q3: What are the 3 types of instructions in ChAcc?
- Q4: the flag register contains..
  - Exceptions
  - Results of comparison in the ALU
  - Both
  - None of the above

Lecture 6, part 2:
Modeling of Circuits with a Regular Structure

Generate scheme for equations

Data-flow VHDL

Major instructions

Concurrent statements

- concurrent signal assignment (⇐)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- **generate scheme for equations (for-generate)**

Conditional concurrent signal assignment

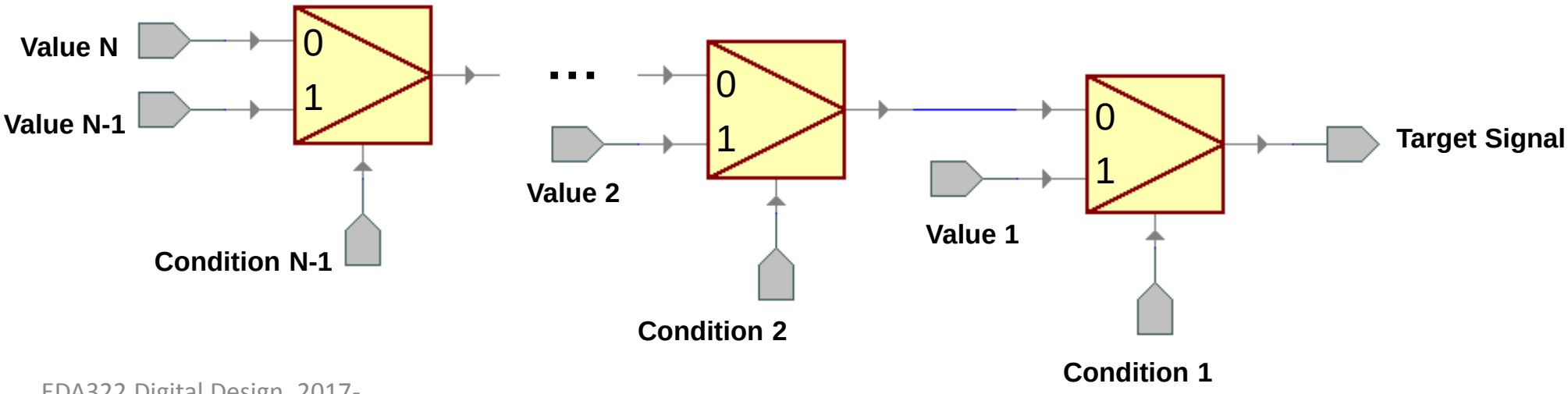
When - Else

```
target_signal <= value1 when condition1 else
                 value2 when condition2 else
                   . . .
                 valueN-1 when conditionN-1 else
                 valueN;
```

Most often implied structure

When - Else

```
target_signal <= value1 when condition1 else
                 value2 when condition2 else
                    . . .
                 valueN-1 when conditionN-1 else
                 valueN;
```



Data-flow VHDL

Major instructions

Concurrent statements

- concurrent signal assignment (⇐)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- **generate scheme for equations (for-generate)**

Selected concurrent signal assignment

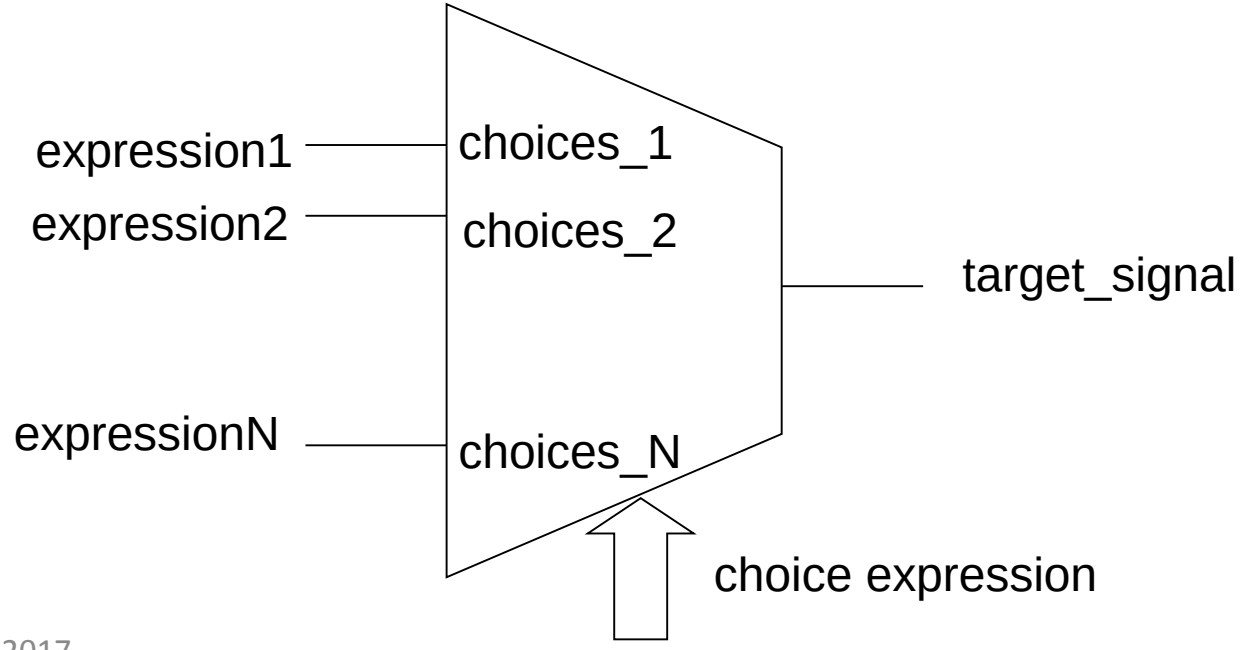
With –Select-When

```
with choice_expression select
   target_signal <= expression1 when choices_1,
                    expression2 when choices_2,
                        . . .
                    expressionN when choices_N;
```

Most Often Implied Structure

With –Select-When

```
with choice_expression select
   target_signal <= expression1 when choices_1,
                    expression2 when choices_2,
                        . . .
                    expressionN when choices_N;
```



Allowed formats of *choices_k*

WHEN value

WHEN value_1 | value_2 | .... | value N

WHEN OTHERS

Allowed formats of choice_k - example

```
WITH sel SELECT
     y <= a WHEN "000",
           c WHEN "001" | "111",
           d WHEN OTHERS;
```

Data-flow VHDL

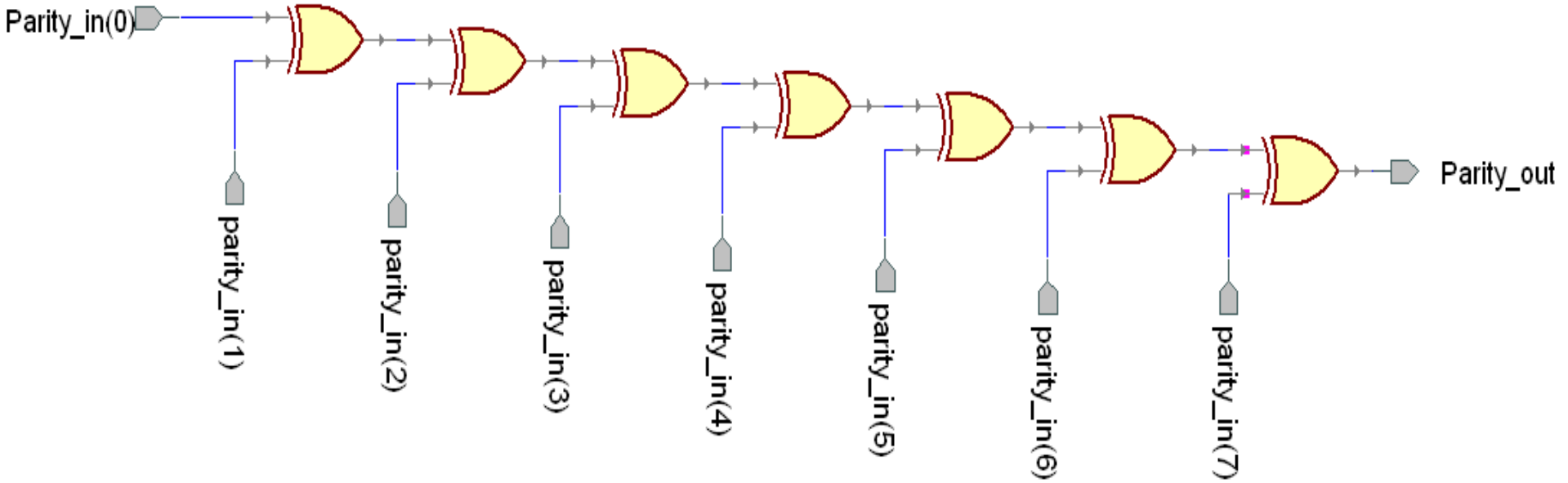
Major instructions

Concurrent statements

- concurrent signal assignment (⇐)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- **generate scheme for equations (for-generate)**

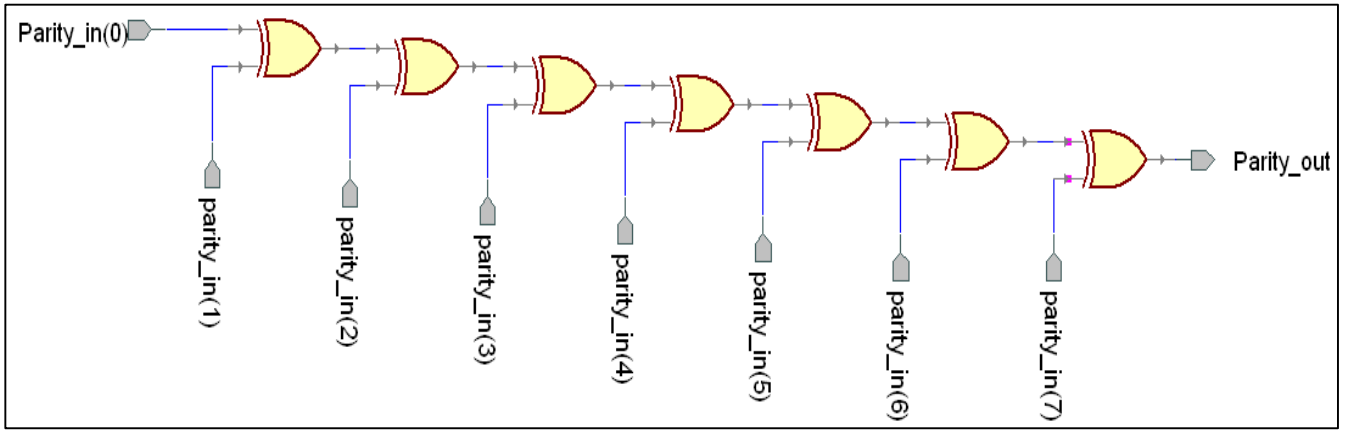
PARITY Example

PARITY: Block Diagram

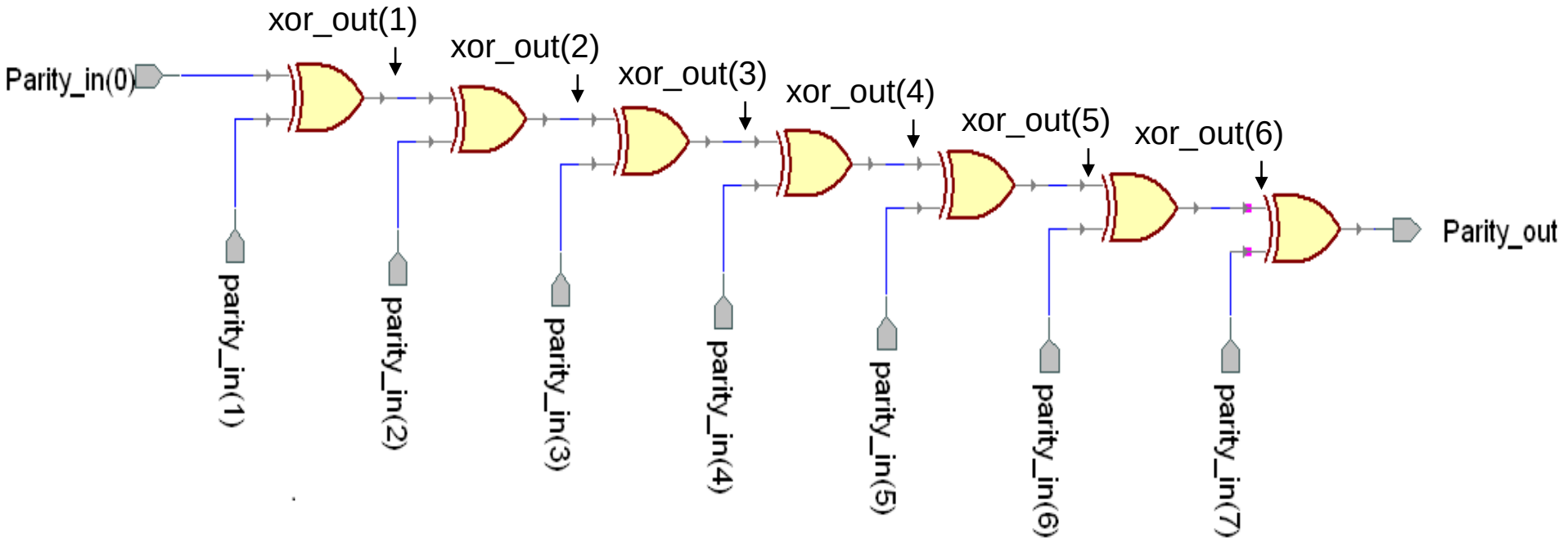


PARITY: Entity Declaration

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY parity IS
      PORT(
            parity_in    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            parity_out  : OUT STD_LOGIC
          );
END parity;
```

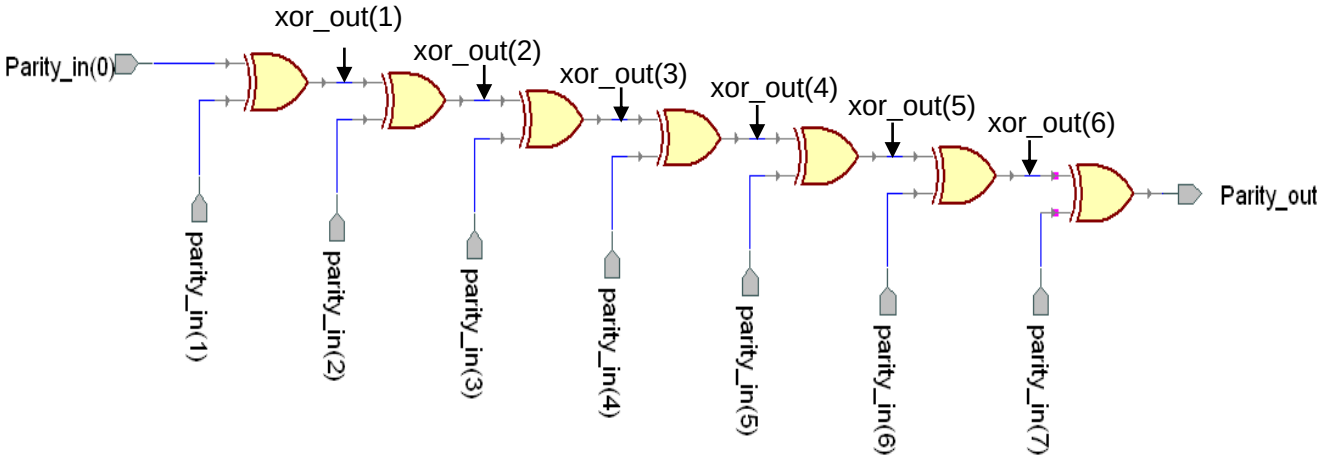


PARITY: Block Diagram

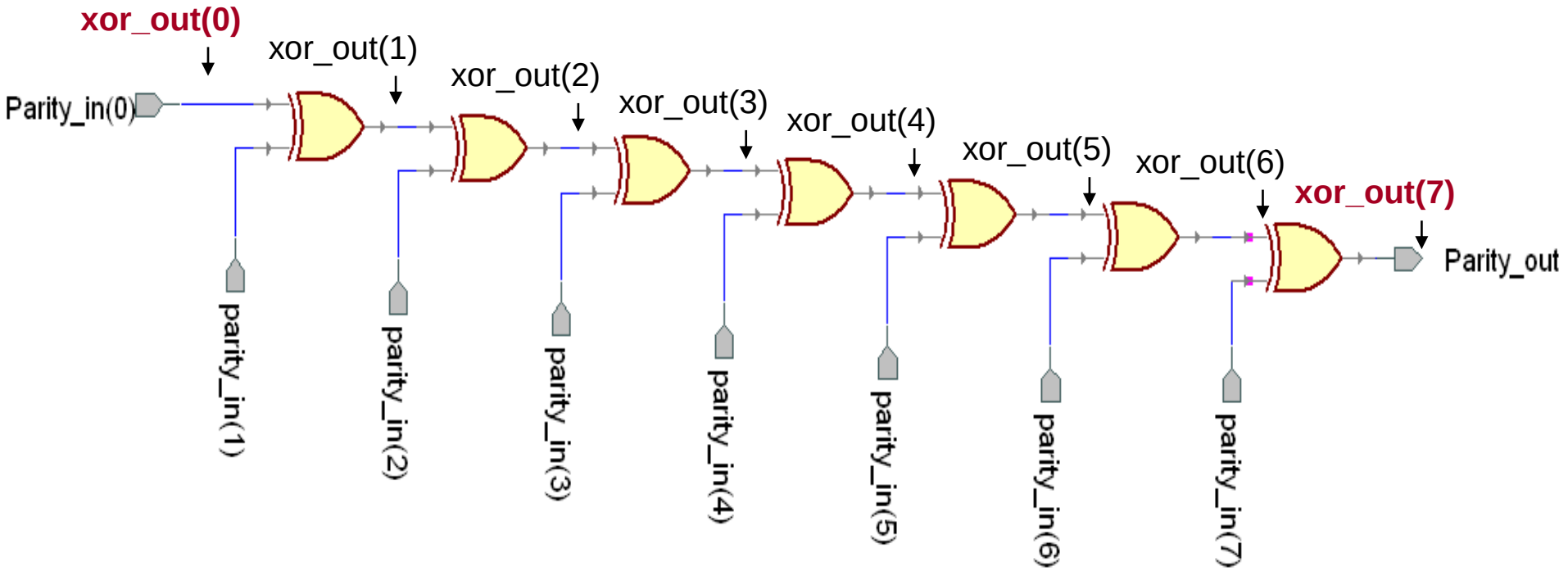


PARITY: Architecture

```
ARCHITECTURE parity_dataflow OF parity IS

SIGNAL xor_out: std_logic_vector (6 downto 1);

BEGIN

    xor_out(1) <= parity_in(0) XOR parity_in(1);
    xor_out(2) <= xor_out(1) XOR parity_in(2);
    xor_out(3) <= xor_out(2) XOR parity_in(3);
    xor_out(4) <= xor_out(3) XOR parity_in(4);
    xor_out(5) <= xor_out(4) XOR parity_in(5);
    xor_out(6) <= xor_out(5) XOR parity_in(6);
    parity_out <= xor_out(6) XOR parity_in(7);

END parity_dataflow;
```

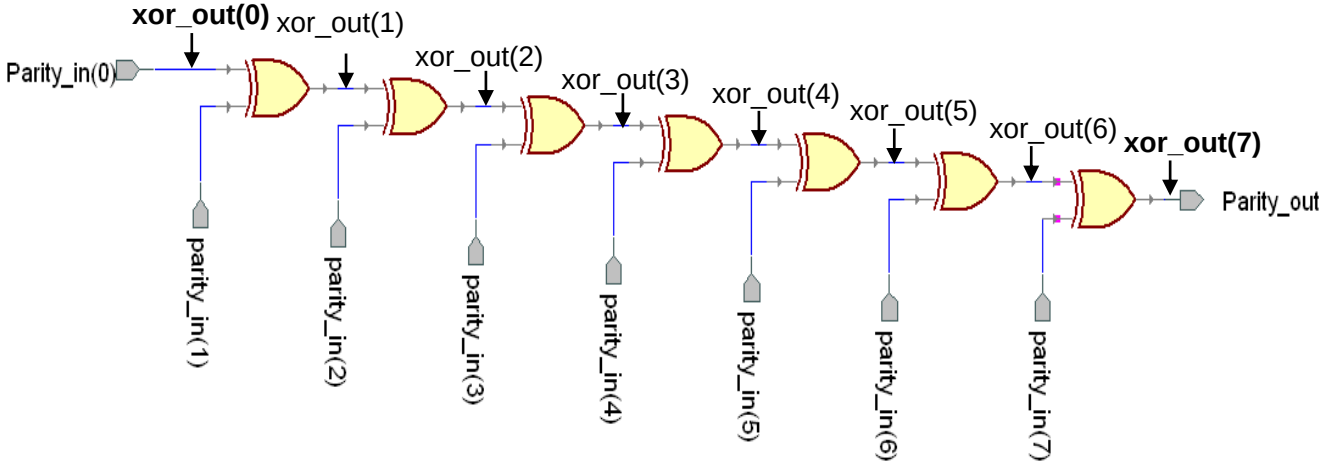


PARITY: Block Diagram (2)



PARITY: Architecture

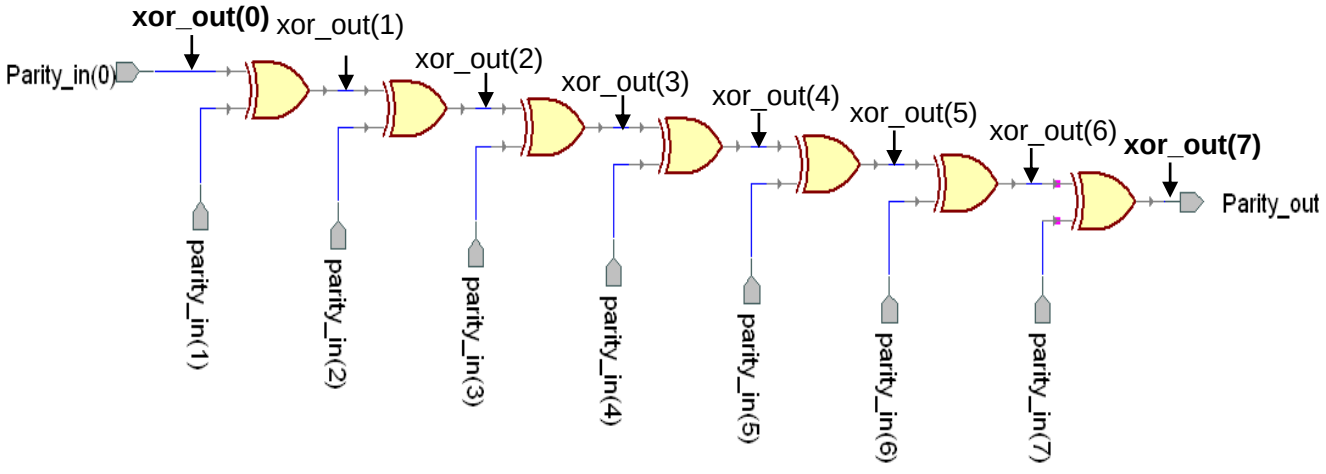
```
ARCHITECTURE parity_dataflow OF parity IS

SIGNAL xor_out: STD_LOGIC_VECTOR (7 downto 0);

BEGIN

    xor_out(0) <= parity_in(0);
    xor_out(1) <= xor_out(0) XOR parity_in(1);
    xor_out(2) <= xor_out(1) XOR parity_in(2);
    xor_out(3) <= xor_out(2) XOR parity_in(3);
    xor_out(4) <= xor_out(3) XOR parity_in(4);
    xor_out(5) <= xor_out(4) XOR parity_in(5);
    xor_out(6) <= xor_out(5) XOR parity_in(6);
    xor_out(7) <= xor_out(6) XOR parity_in(7);
    parity_out <= xor_out(7);

END parity_dataflow;
```



PARITY: Architecture

```
ARCHITECTURE parity_dataflow OF parity IS

SIGNAL xor_out: STD_LOGIC_VECTOR (7 downto 0);

BEGIN

    xor_out(0) <= parity_in(0);
    xor_out(1) <= xor_out(0) XOR parity_in(1);
    xor_out(2) <= xor_out(1) XOR parity_in(2);
    xor_out(3) <= xor_out(2) XOR parity_in(3);
    xor_out(4) <= xor_out(3) XOR parity_in(4);
    xor_out(5) <= xor_out(4) XOR parity_in(5);
    xor_out(6) <= xor_out(5) XOR parity_in(6);
    xor_out(7) <= xor_out(6) XOR parity_in(7);
    parity_out <= xor_out(7);

END parity_dataflow;
```

```
G2: FOR i IN 1 TO 7 GENERATE
        xor_out(i) <= xor_out(i-1) XOR
    parity_in(i);
    END GENERATE G2;
```



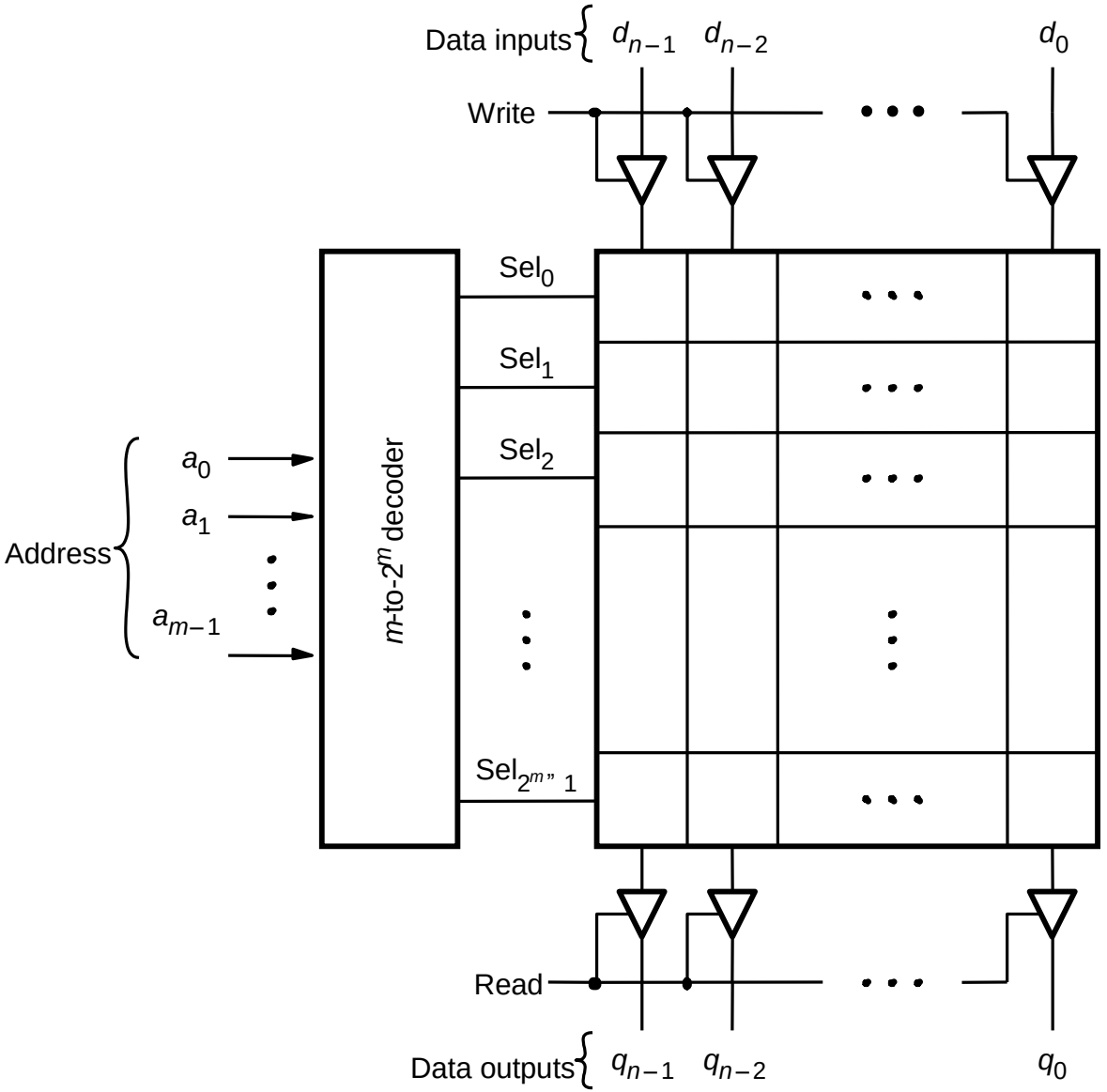
For Generate Statement

For - Generate

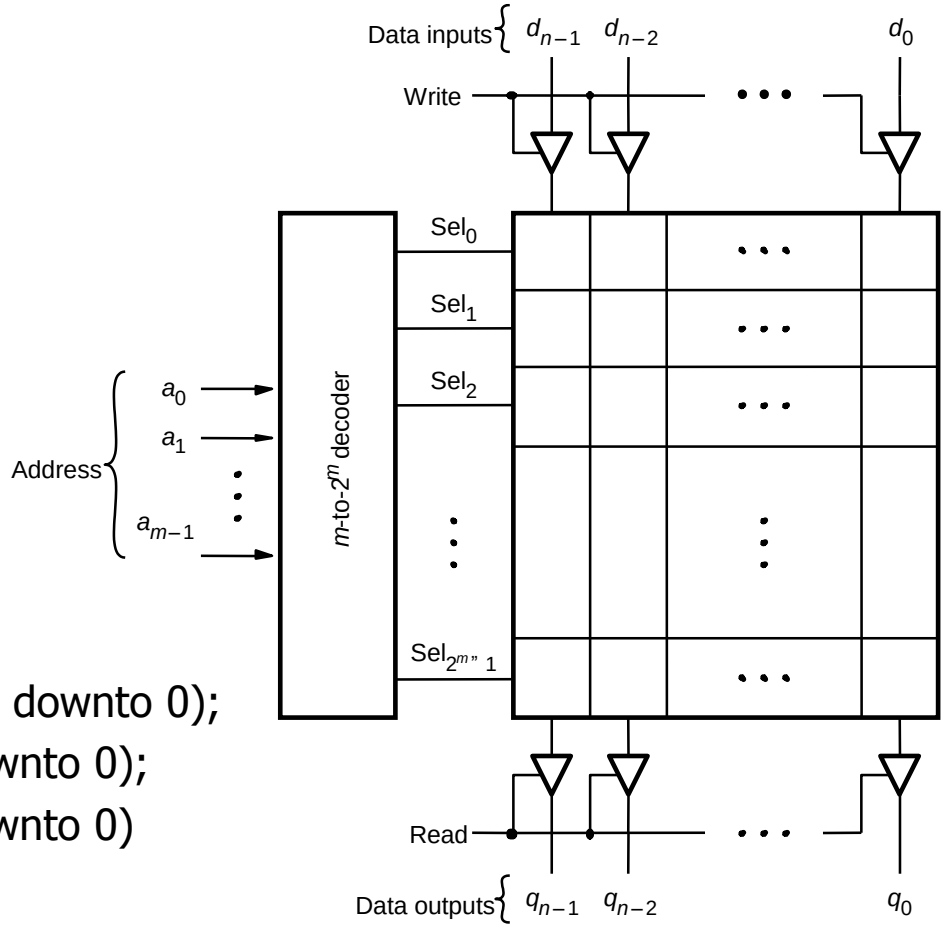
```
label: FOR identifier IN range GENERATE
          BEGIN
            {Concurrent Statements}
          END GENERATE;
```

Memory

Random Access Memory (RAM)



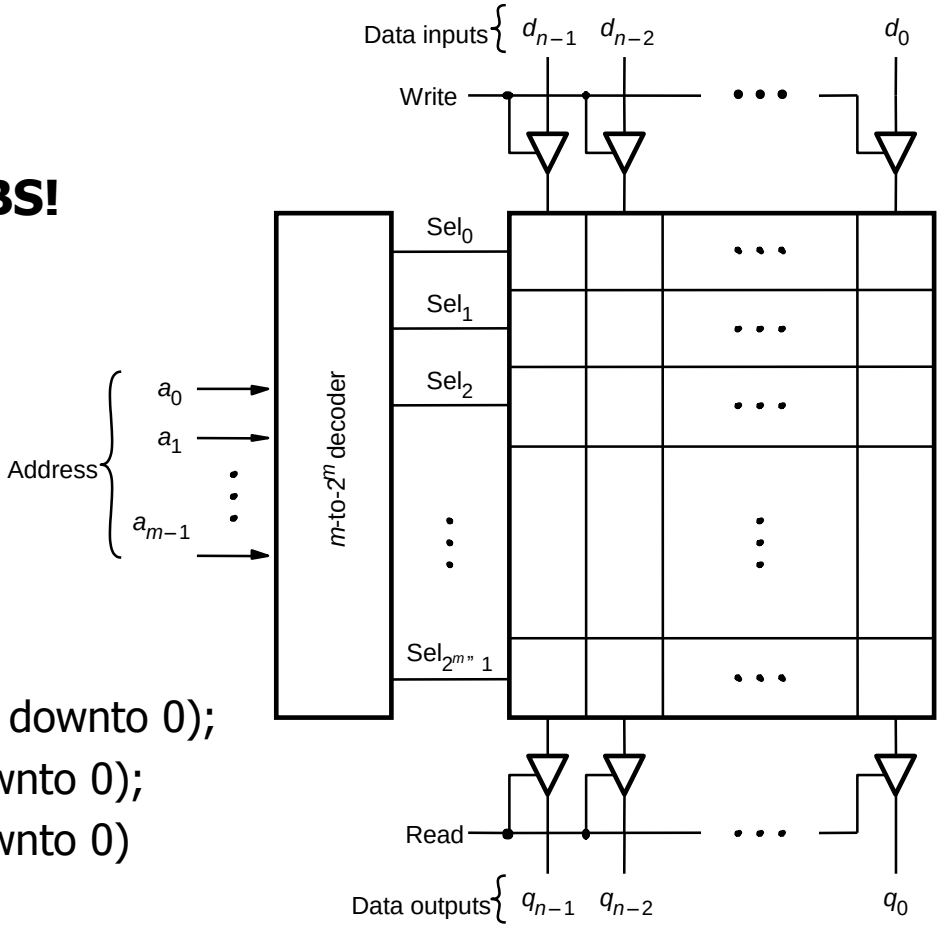
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all ;

entity SRAM is
generic(   width:          integer:=4;
           addr_bits:      integer:=2);

port(Clock:          in    std_logic;
     Read:           in    std_logic;
     Write:          in    std_logic;
     Address:        in    std_logic_vector(addr_bits-1 downto 0);
     Data_in:        in    std_logic_vector(width-1 downto 0);
     Data_out:       out   std_logic_vector(width-1 downto 0)
);
end SRAM;
```



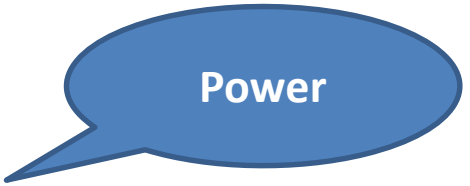
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all ;

entity SRAM is
generic( width:          integer:=4;
         addr_bits:      integer:=2);

port(Clock:          in    std_logic;
     Read:           in    std_logic;
     Write:          in    std_logic;
     Address:        in    std_logic_vector(addr_bits-1 downto 0);
     Data_in:        in    std_logic_vector(width-1 downto 0);
     Data_out:       out   std_logic_vector(width-1 downto 0)
);
end SRAM;
```

OBS!

OBS!

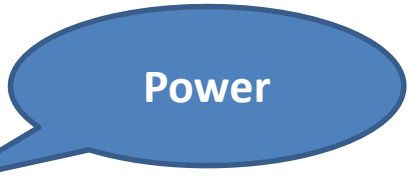


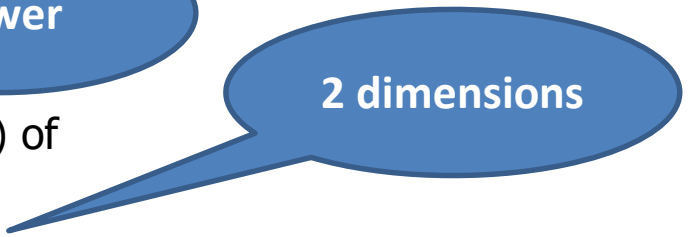
```
architecture behav of SRAM is

type ram_type is array (0 to (2**addr_bits)-1) of
      std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type := (OTHERS => (OTHERS => '0')); -- initialize memory with '0'

begin

process(Clock, Read, Write)
   begin
      if (Clock'event and Clock='1') then

            if Read='1' then
-- build-in functions to_integer(unsigned(x)) change the type from std_logic_vector to integer
               Data_out <= tmp_ram(to_integer(unsigned(Address)));
            else
               Data_out <= (OTHERS => 'Z');
            end if;
            if Write='1' then
               tmp_ram(to_integer(unsigned(Address))) <= Data_in;
            end if;

      end if;
end process;

end behav;
```




```
architecture behav of SRAM is
type ram_type is array (0 to (2**addr_bits)-1) of
      std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type := (OTHERS => (OTHERS => '0')); -- initialize memory with '0'
begin
process(Clock, Read, Write)
   begin
      if (Clock'event and Clock='1') then
            if Read='1' then
-- build-in functions to_integer(unsigned(x)) change the type from std_logic_vector to integer
               Data_out <= tmp_ram(to_integer(unsigned(Address)));
            else
               Data_out <= (OTHERS => 'Z');
            end if;
            if Write='1' then
               tmp_ram(to_integer(unsigned(Address))) <= Data_in;
            end if;
      end if;
end process;
end behav;
```

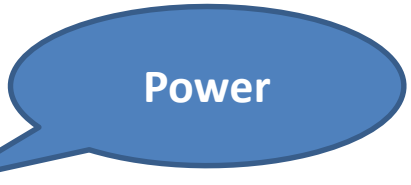

```
architecture behav of SRAM is

type ram_type is array (0 to (2**addr_bits)-1) of
        std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type := (OTHERS => (OTHERS => '0')); -- initialize memory with '0'

begin

process(Clock, Read, Write)
    begin
        if (Clock'event and Clock='1') then

                if Read='1' then
-- build-in functions to_integer(unsigned(x)) change the type from std_logic_vector to integer
                    Data_out <= tmp_ram(to_integer(unsigned(Address)));
                else
                    Data_out <= (OTHERS => 'Z');
                end if;
                if Write='1' then
                    tmp_ram(to_integer(unsigned(Address))) <= Data_in;
                end if;

        end if;
end process;

end behav;
```

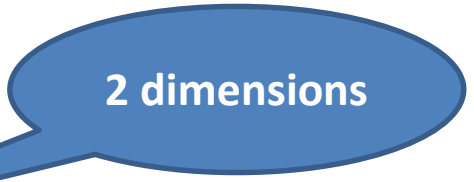





```
architecture behav of SRAM is
type ram_type is array (0 to (2**addr_bits)-1) of
        std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type := (OTHERS => (OTHERS => '0')); -- initialize memory with '0'
begin
process(Clock, Read, Write)
    begin
       if (Clock'event and Clock='1') then
            if Read='1' then
-- build-in functions to_integer(unsigned(x)) change the type from std_logic_vector to integer
               Data_out <= tmp_ram(to_integer(unsigned(Address)));
            else
               Data_out <= (OTHERS => 'Z');
            end if;
            if Write='1' then
               tmp_ram(to_integer(unsigned(Address))) <= Data_in;
            end if;
       end if;
end process;
end behav;
```

Sequential Part






```
architecture behav of SRAM is
type ram_type is array (0 to (2**addr_bits)-1) of
        std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type := (OTHERS => (OTHERS => '0')); -- initialize memory with '0'
begin
```


```
process(Clock, Read, Write)
    begin
       if (Clock'event and Clock='1') then
            if Read='1' then
-- build-in functions to_integer(unsigned(x)) change the type from std_logic_vector to integer
                Data_out <= tmp_ram(to_integer(unsigned(Address)));
            else
                Data_out <= (OTHERS => 'Z');
            end if;
            if Write='1' then
                tmp_ram(to_integer(unsigned(Address))) <= Data_in;
            end if;
       end if;
end process;
```

Sequential Part

```
end behav;
```

```
architecture behav of SRAM is
type ram_type is array (0 to (2**addr_bits)-1) of
        std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type := (OTHERS => (OTHERS => '0')); -- initialize memory with '0'
begin
process(Clock, Read, Write)
    begin
       if (Clock'event and Clock='1') then
            if Read='1' then
-- build-in functions to_integer(unsigned(x)) change the type from std_logic_vector to integer
                Data_out <= tmp_ram(to_integer(unsigned(Address)));
            else
                Data_out <= (OTHERS => 'Z');
            end if;
            if Write='1' then
                tmp_ram(to_integer(unsigned(Address))) <= Data_in;
            end if;
       end if;
end process;
end behav;
```

Power

2 dimensions

Sensitivity List

Better to use: rising_edge(clk)

Sequential Part

Behavioral SRAM Memory

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all ;

entity SRAM is
generic(    width:      integer:=4;
            addr_bits:  integer:=2);

port(Clock:         in      std_logic;
     Read:          in      std_logic;
     Write:         in      std_logic;
     Address:               in      std_logic_vector(addr_bits-1 downto 0);
     Data_in:               in      std_logic_vector(width-1 downto 0);
     Data_out:      out     std_logic_vector(width-1 downto 0)
);
end SRAM;
```

```
architecture behav of SRAM is
type ram_type is array (0 to (2**addr_bits)-1) of
        std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type := (OTHERS => (OTHERS => '0'));
begin
process(Clock, Read, Write)
   begin
      if (Clock'event and Clock='1') then
            if Read='1' then

               Data_out <= tmp_ram(to_integer(unsigned(Address)));
            else
               Data_out <= (OTHERS => 'Z');
            end if;
            if Write='1' then
               tmp_ram(to_integer(unsigned(Address))) <= Data_in;
            end if;
      end if;
end process;
end behav;
```

Generate scheme for components

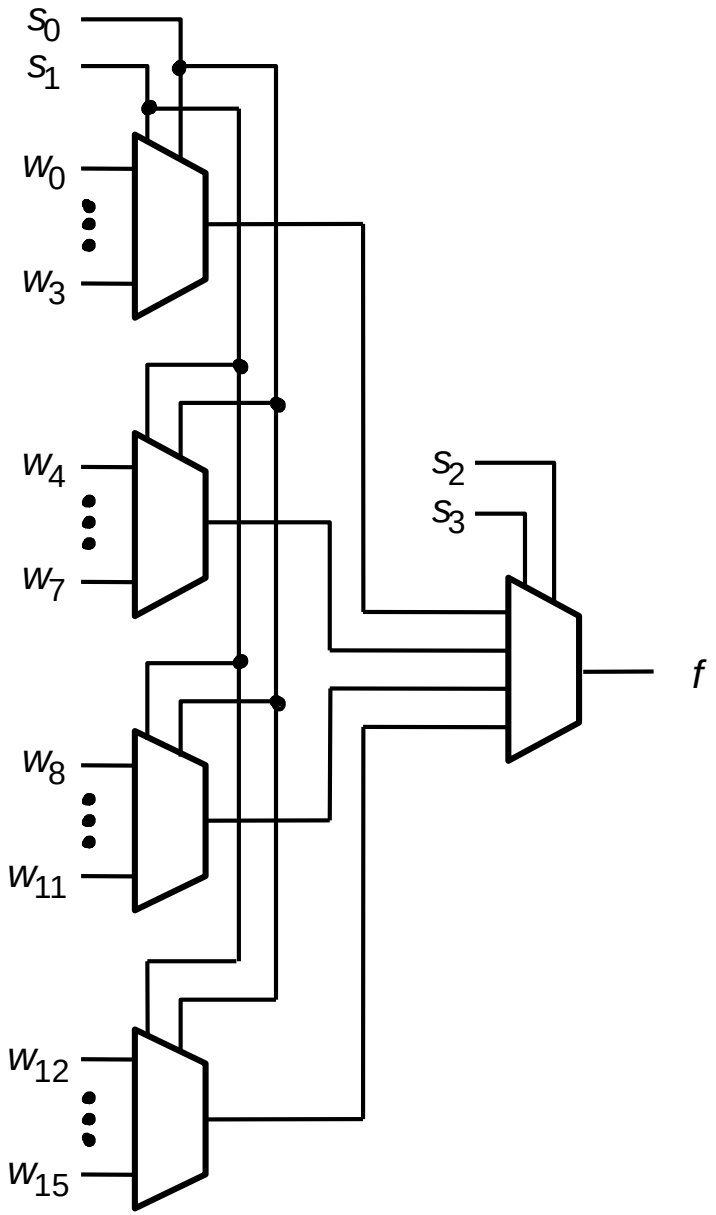
Structural VHDL

Major instructions

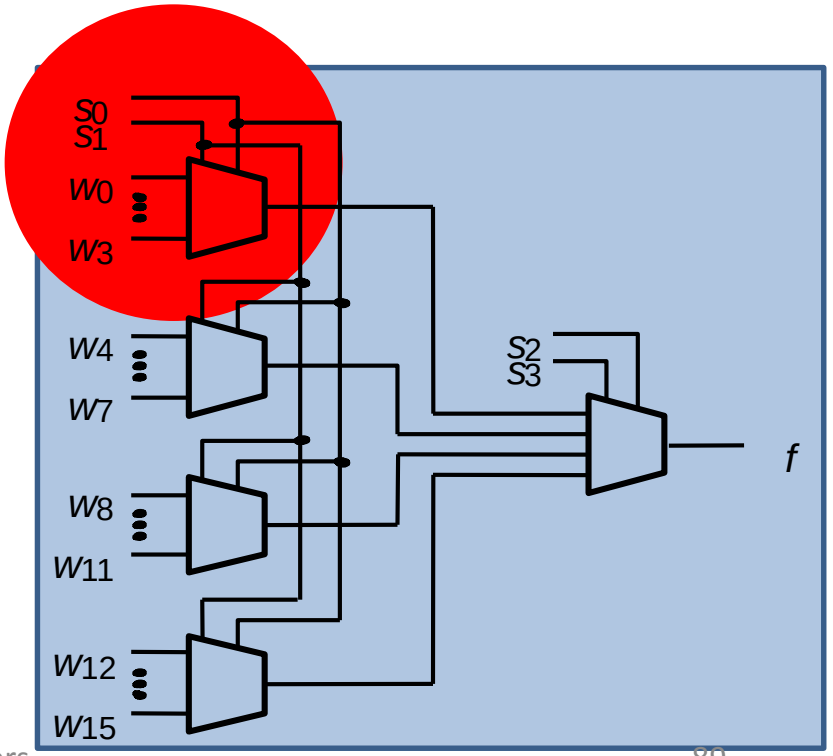
- component instantiation (port map)
- component instantiation with generic (generic map, port map)
- **generate scheme for component instantiations (for-generate)**

Example 1

Example 1

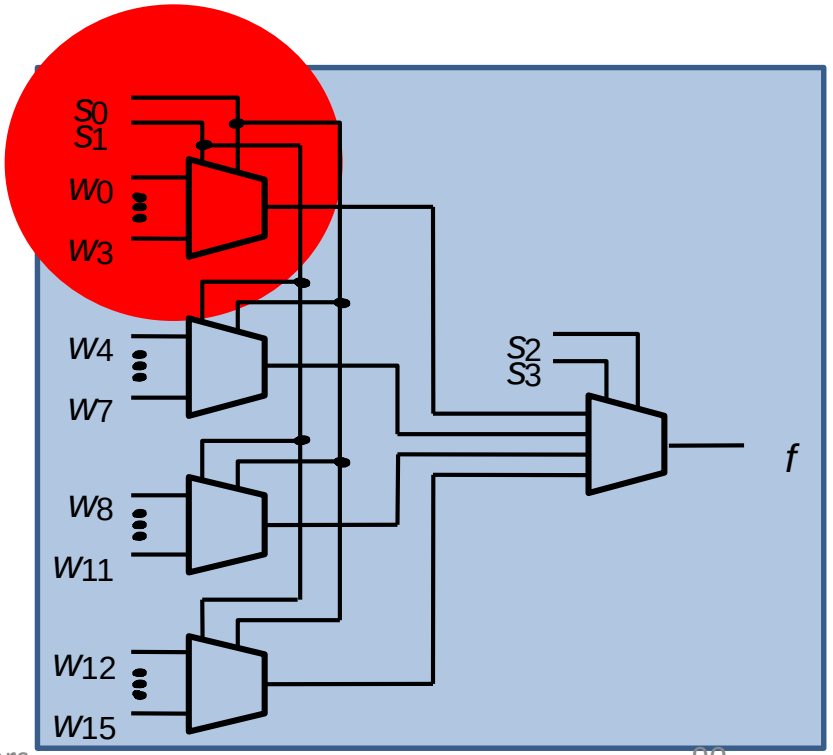


A 4-to-1 Multiplexer



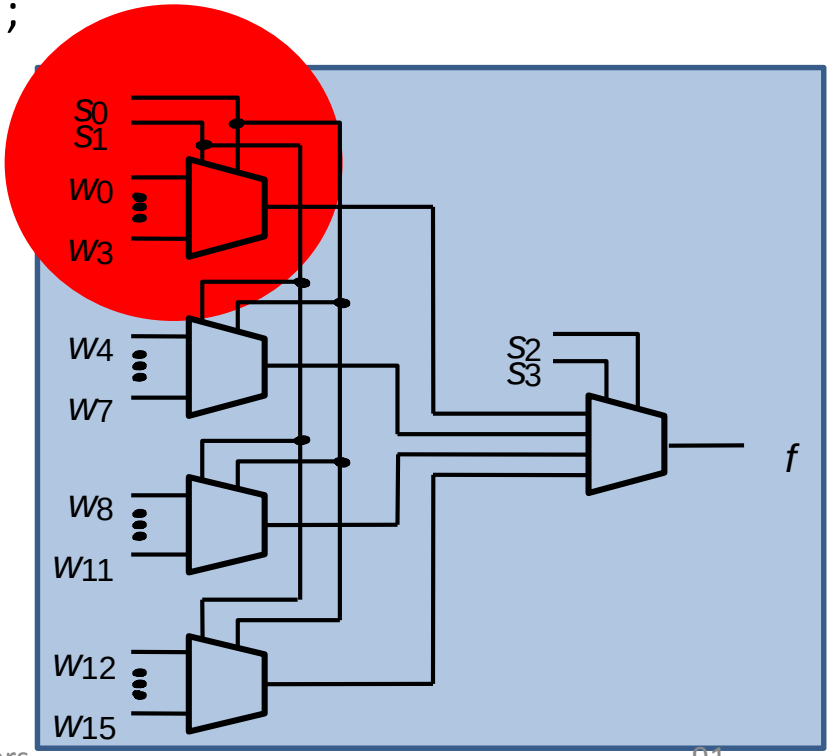
A 4-to-1 Multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT (

                ) ;
END mux4to1 ;

ARCHITECTURE Dataflow OF mux4to1 IS
BEGIN




END Dataflow ;
```



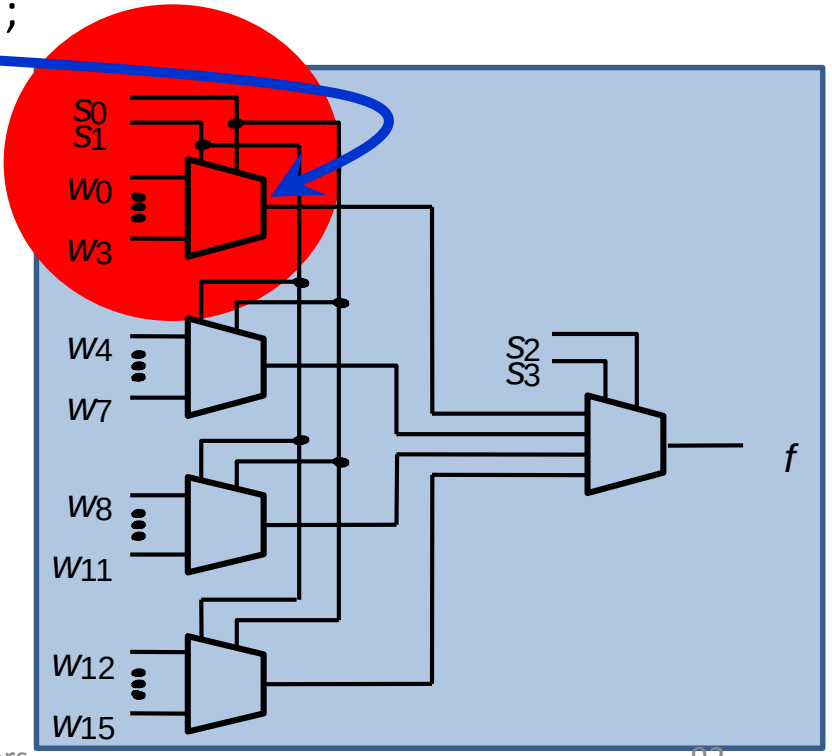
A 4-to-1 Multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT (      w0, w1, w2, w3    : IN      STD_LOGIC ;
                s          : IN      STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                f          : OUT     STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Dataflow OF mux4to1 IS
BEGIN




END Dataflow ;
```



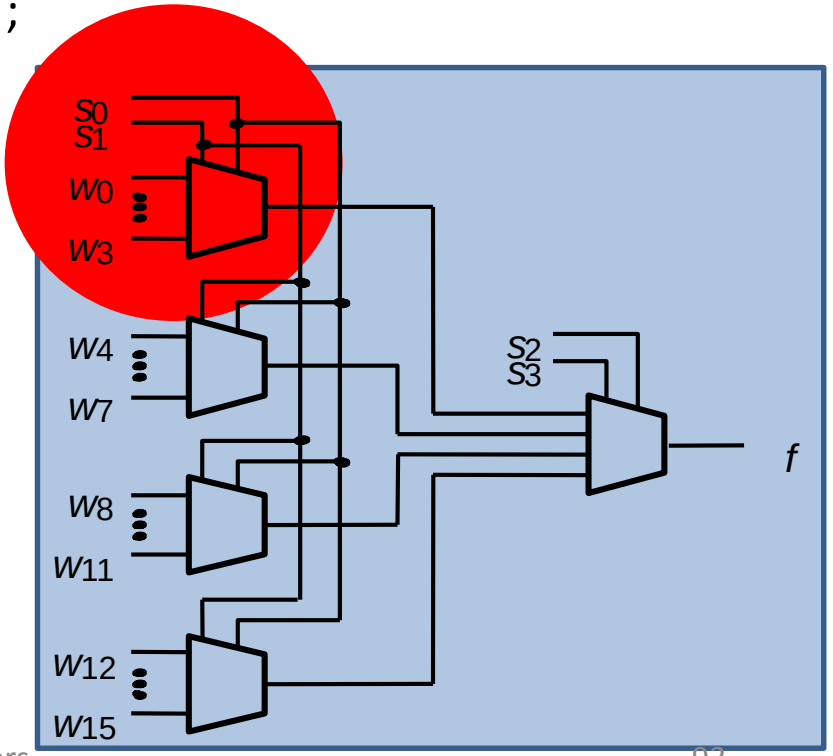
A 4-to-1 Multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT (      w0, w1, w2, w3    : IN      STD_LOGIC ;
                s                 : IN      STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                f                 : OUT     STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Dataflow OF mux4to1 IS
BEGIN




END Dataflow ;
```



A 4-to-1 Multiplexer

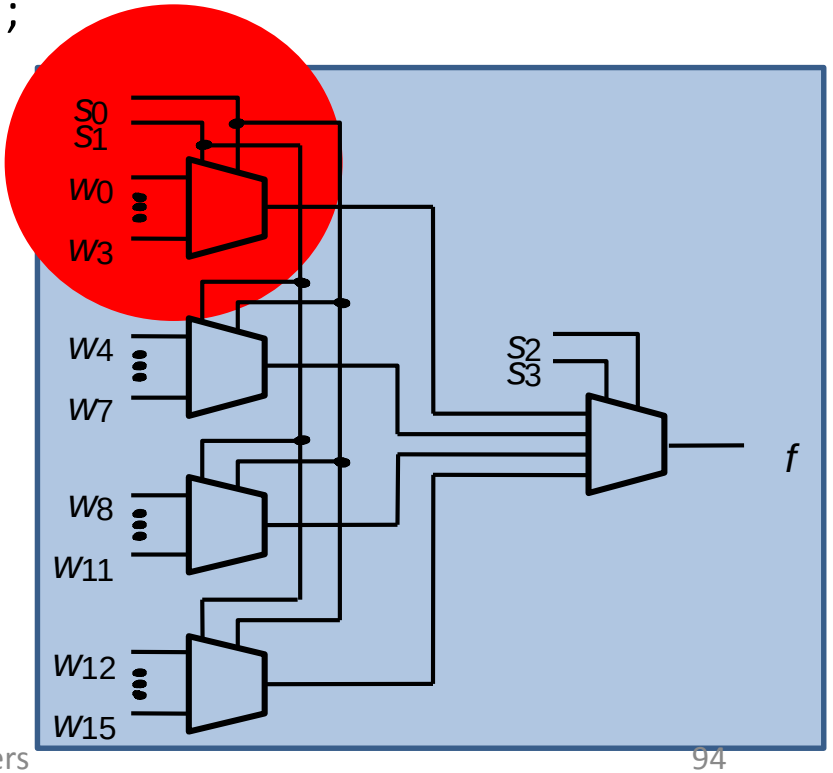
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT (          w0, w1, w2, w3    : IN       STD_LOGIC ;
                    s                 : IN       STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                    f                 : OUT      STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Dataflow OF mux4to1 IS
BEGIN
    WITH s SELECT
          f <= w0 WHEN "00",
               w1 WHEN "01",
               w2 WHEN "10",
               w3 WHEN OTHERS ;
END Dataflow ;
```



A 4-to-1 Multiplexer

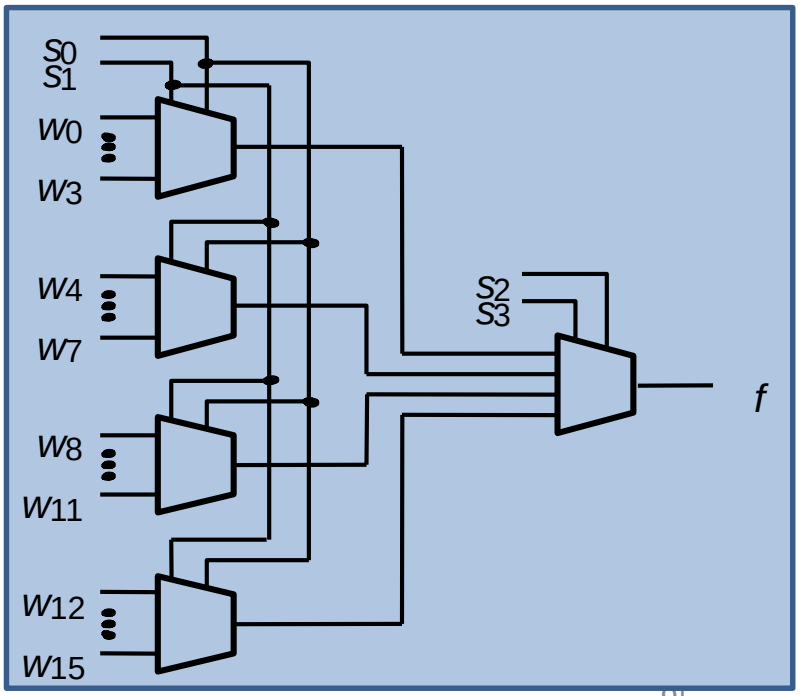
Save it as mux4to1.vhd file.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT (      w0, w1, w2, w3    : IN      STD_LOGIC ;
                s                 : IN      STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                f                 : OUT     STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Dataflow OF mux4to1 IS
BEGIN
    WITH s SELECT
        f <= w0 WHEN "00",
             w1 WHEN "01",
             w2 WHEN "10",
             w3 WHEN OTHERS ;
END Dataflow ;
```



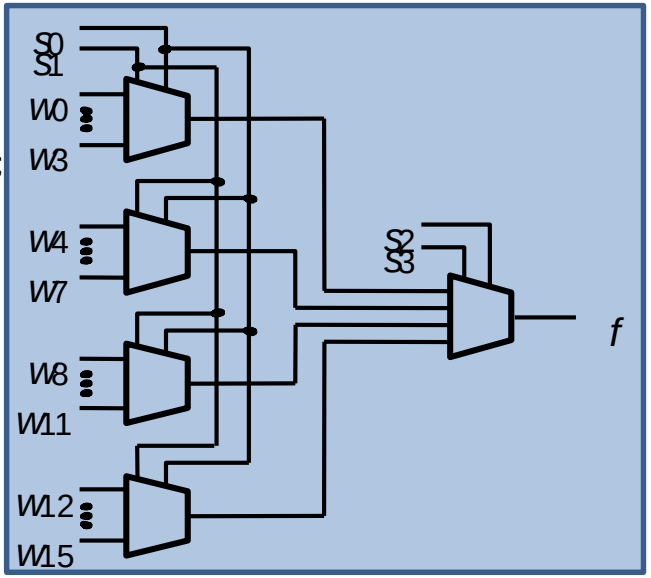
Straightforward code for Example 1

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Example1 IS
    PORT ( w     : IN     STD_LOGIC_VECTOR(0 TO 15) ;
           s     : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           f     : OUT    STD_LOGIC ) ;
END Example1 ;
```



Straightforward code for Example 1

```
ARCHITECTURE Structure OF Example1 IS

    COMPONENT mux4to1
        PORT (    w0, w1, w2, w3      : IN     STD_LOGIC ;
                  s                   : IN     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                  f                   : OUT    STD_LOGIC ) ;
    END COMPONENT ;

    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;

BEGIN
    Mux1: mux4to1 PORT MAP ( w(0),   w(1),   w(2),   w(3),    s(1 DOWNTO 0), m(0) ) ;
    Mux2: mux4to1 PORT MAP ( w(4),   w(5),   w(6),   w(7),    s(1 DOWNTO 0), m(1) ) ;
    Mux3: mux4to1 PORT MAP ( w(8),   w(9),   w(10),  w(11),   s(1 DOWNTO 0), m(2) ) ;
    Mux4: mux4to1 PORT MAP ( w(12),  w(13),  w(14),  w(15),   s(1 DOWNTO 0), m(3) ) ;
    Mux5: mux4to1 PORT MAP ( m(0),   m(1),   m(2),   m(3),    s(3 DOWNTO 2),  f  ) ;
END Structure ;
```

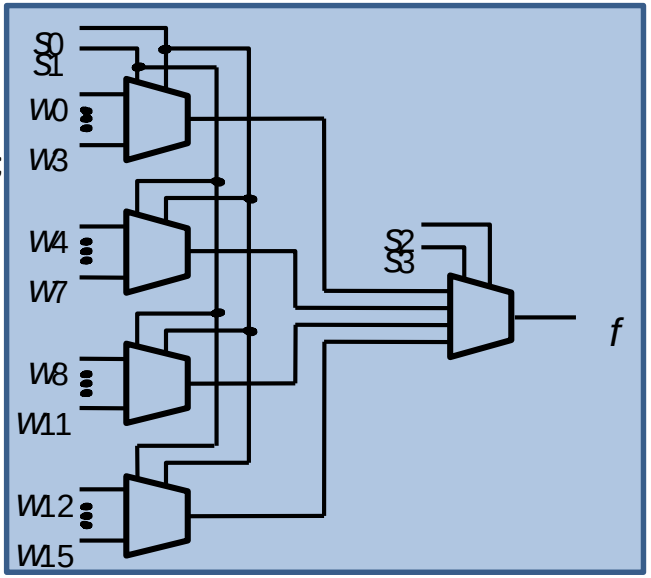


Straightforward code for Example 1

```
ARCHITECTURE Structure OF Example1 IS
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;

BEGIN
    Mux1: mux4to1 PORT MAP ( w(0),  w(1),  w(2),  w(3),   s(1 DOWNTO 0), m(0) ) ;
    Mux2: mux4to1 PORT MAP ( w(4),  w(5),  w(6),  w(7),   s(1 DOWNTO 0), m(1) ) ;
    Mux3: mux4to1 PORT MAP ( w(8),  w(9),  w(10), w(11),  s(1 DOWNTO 0), m(2) ) ;
    Mux4: mux4to1 PORT MAP ( w(12), w(13), w(14), w(15),  s(1 DOWNTO 0), m(3) ) ;
    Mux5: mux4to1 PORT MAP ( m(0),  m(1),  m(2),  m(3),   s(3 DOWNTO 2),  f  ) ;
END Structure ;
```

You can remove component declaration but then you have to use this format for calling a component:

Mux1: entity work.mux4to1 PORT MAP (…

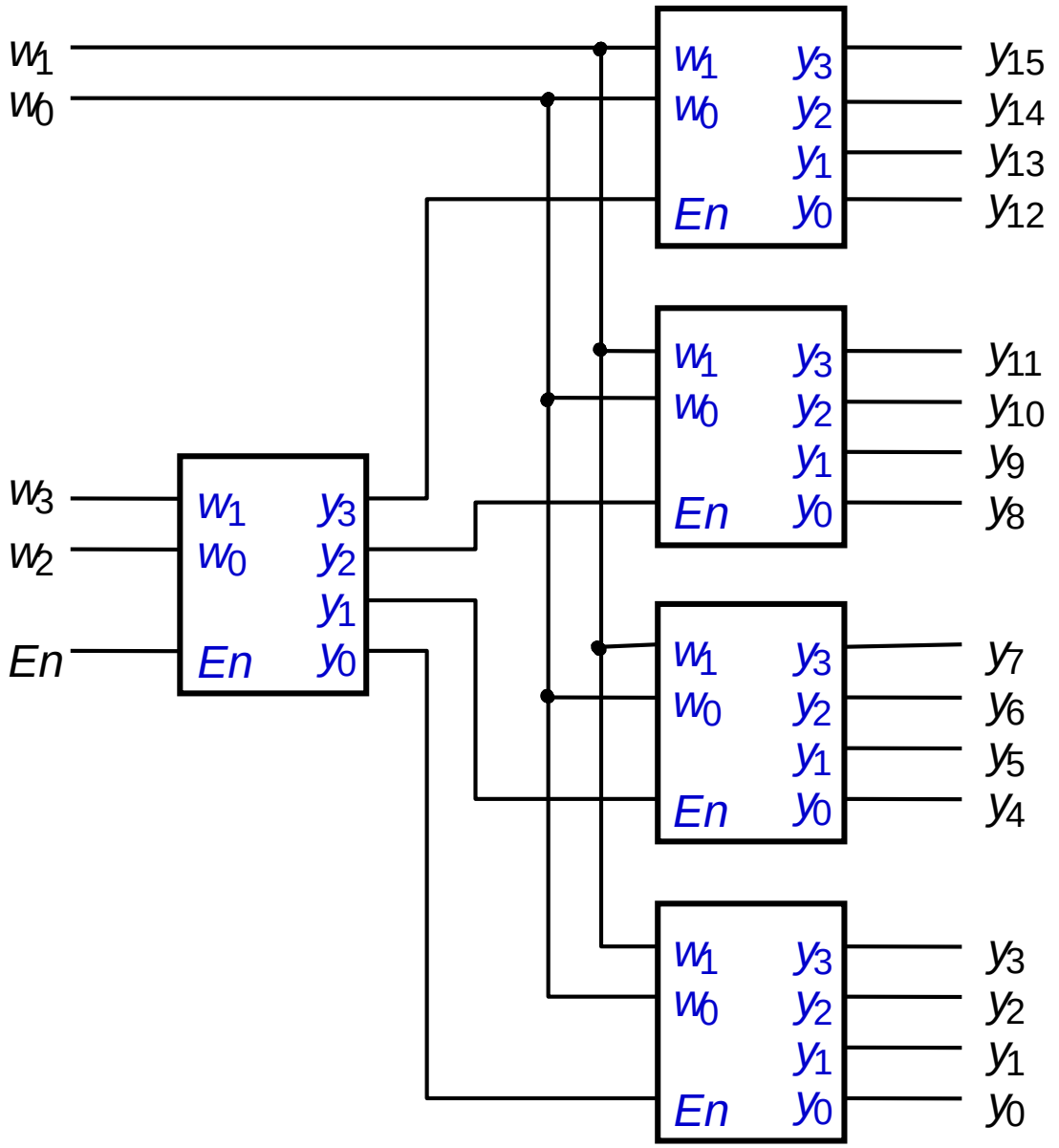


Modified code for Example 1

```
ARCHITECTURE Structure OF Example1 IS

        COMPONENT mux4to1
            PORT (      w0, w1, w2, w3      : IN        STD_LOGIC ;
                        s                   : IN        STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                        f                   : OUT       STD_LOGIC ) ;
        END COMPONENT ;

        SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;

BEGIN
        G1: FOR i IN 0 TO 3 GENERATE
            Muxes: mux4to1 PORT MAP (
                        w(4*i), w(4*i+1), w(4*i+2), w(4*i+3), s(1 DOWNTO 0), m(i) ) ;
        END GENERATE ;
        Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
END Structure ;
```

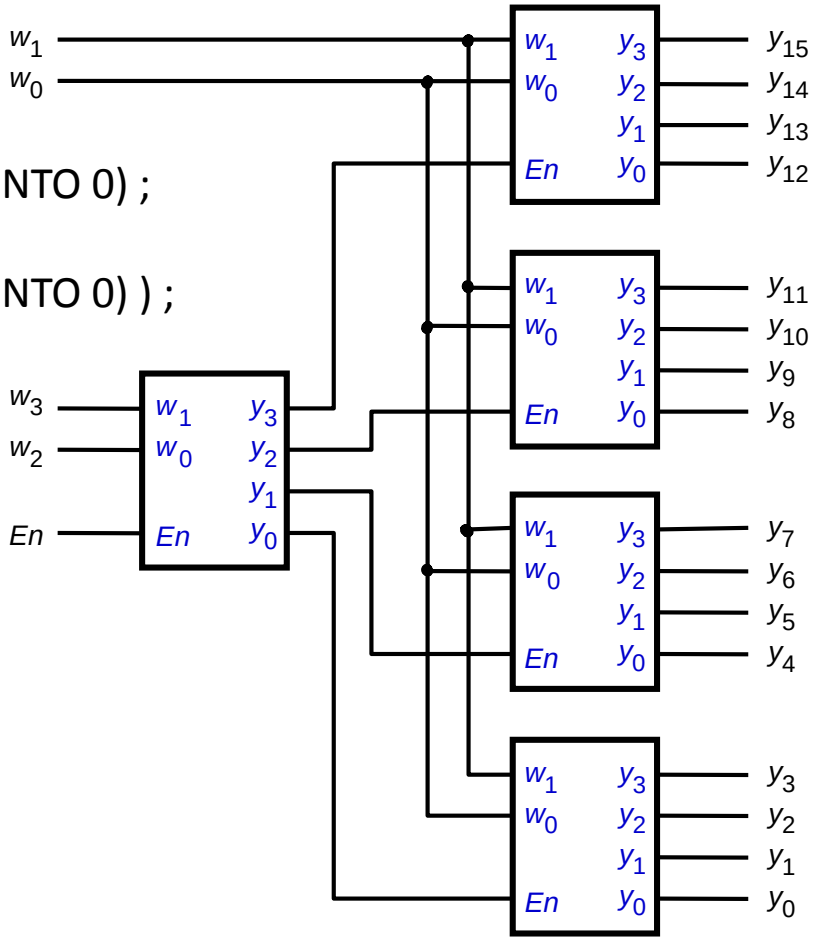
Example 2

Example 2


w_1
w_0
w_3
w_2
En
y_3
y_2
y_1
y_0
y_{15}
y_{14}
y_{13}
y_{12}
y_{11}
y_{10}
y_9
y_8
y_7
y_6
y_5
y_4
y_3
y_2
y_1
y_0

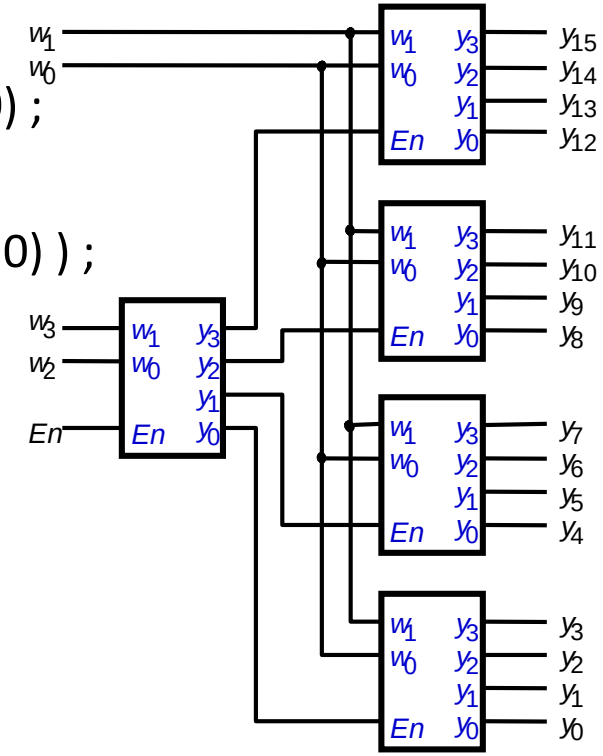
A 2-to-4 binary decoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w     : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En    : IN    STD_LOGIC ;
           y     : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END dec2to4 ;

ARCHITECTURE Dataflow OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <= "0001" WHEN "100",
             "0010" WHEN "101",
             "0100" WHEN "110",
             "1000" WHEN "111",
             "0000" WHEN OTHERS ;
END Dataflow ;
```



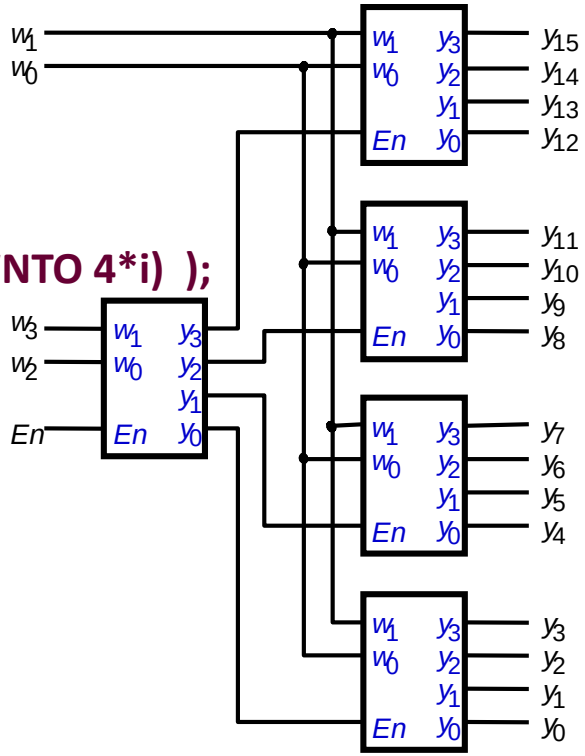
VHDL code for Example 2 (1)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec4to16 IS
    PORT (w     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
          En    : IN    STD_LOGIC ;
          y     : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END dec4to16 ;
```



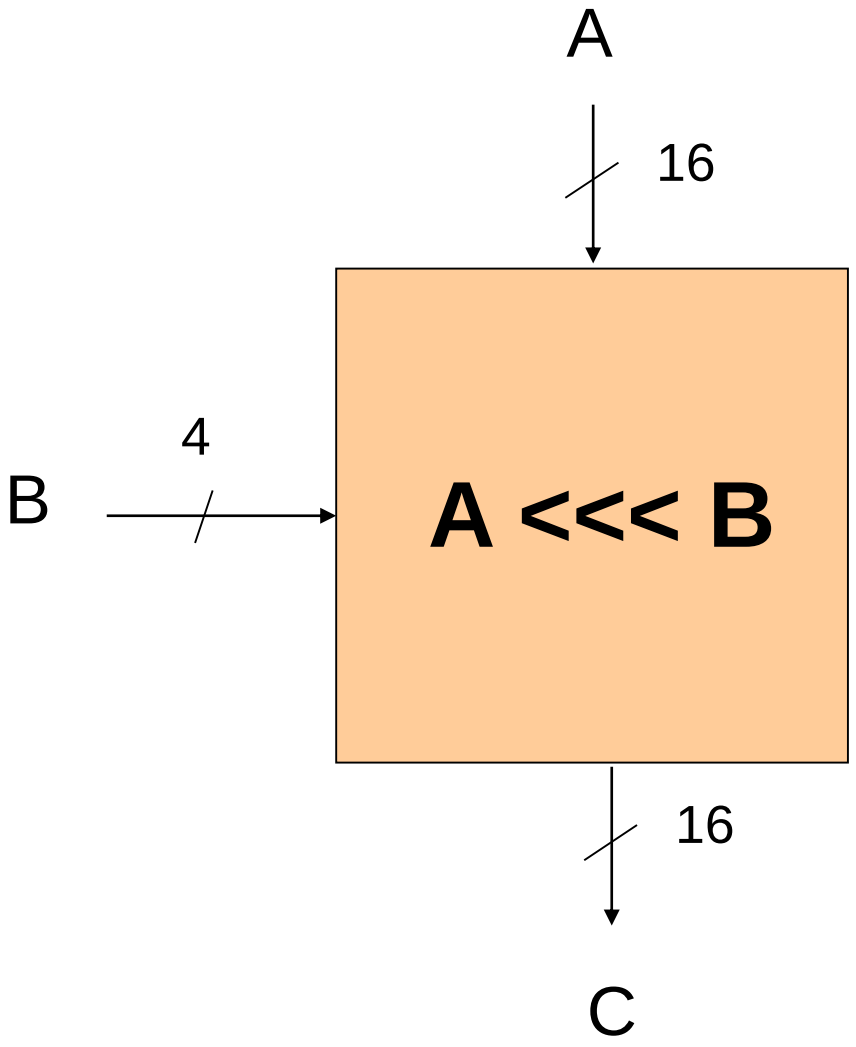
VHDL code for Example 2 (2)

```
ARCHITECTURE Structure OF dec4to16 IS

    COMPONENT dec2to4
        PORT (   w      : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                 En     : IN    STD_LOGIC ;
                 y      : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
    END COMPONENT ;

    SIGNAL m : STD_LOGIC_VECTOR(3 DOWNTO 0) ;

BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Dec_ri: dec2to4 PORT MAP ( w(1 DOWNTO 0), m(i),  y(4*i+3 DOWNTO 4*i)  );
    END GENERATE ;
    Dec_left: dec2to4 PORT MAP ( w(3 DOWNTO 2), En, m ) ;
END Structure ;
```

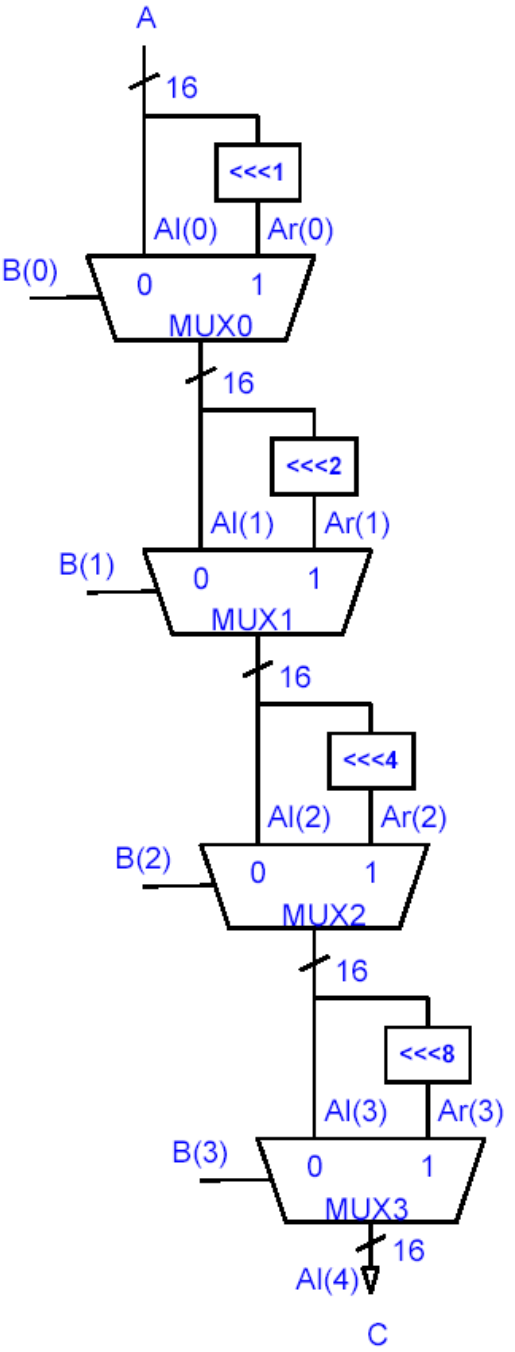


Example 3
Variable Rotator

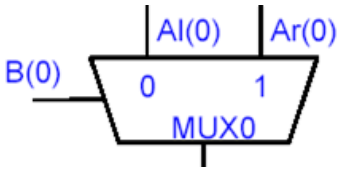
Example 3: Variable rotator - Interface



Block diagram

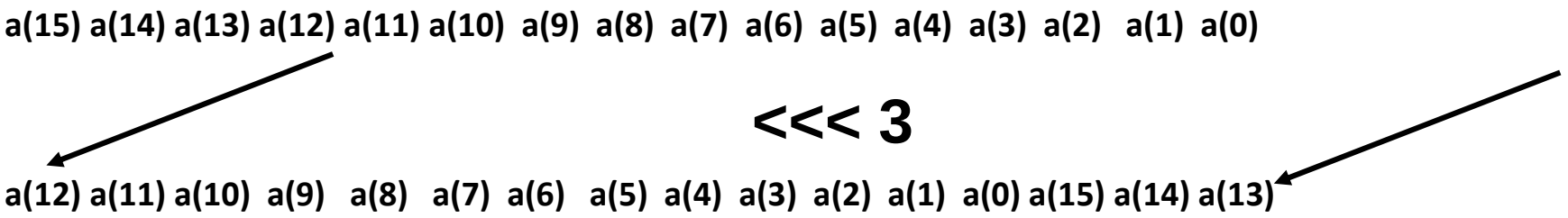


VHDL code for a 16-bit 2-to-1 Multiplexer

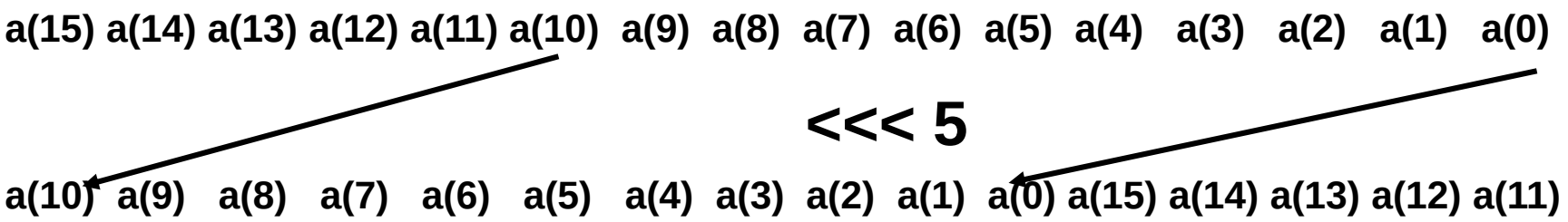



```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1_16 IS
        PORT (      w0          : IN      STD_LOGIC_VECTOR(15 DOWNTO 0);
                    w1          : IN      STD_LOGIC_VECTOR(15 DOWNTO 0);
                    s           : IN      STD_LOGIC ;
                    f           : OUT     STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END mux2to1_16 ;

ARCHITECTURE dataflow OF mux2to1_16 IS
BEGIN
        f <= w0 WHEN s = '0' ELSE w1 ;
END dataflow ;
```

Fixed rotation




```
y <= a(12 downto 0) & a(15 downto 13);
```




```
y <= a(10 downto 0) & a(15 downto 11);
```

Fixed rotation by L positions

a(15) a(14) a(13) a(12) a(11) a(10) a(9) a(8) a(7) a(6) a(5) a(4) a(3) a(2) a(1) a(0)

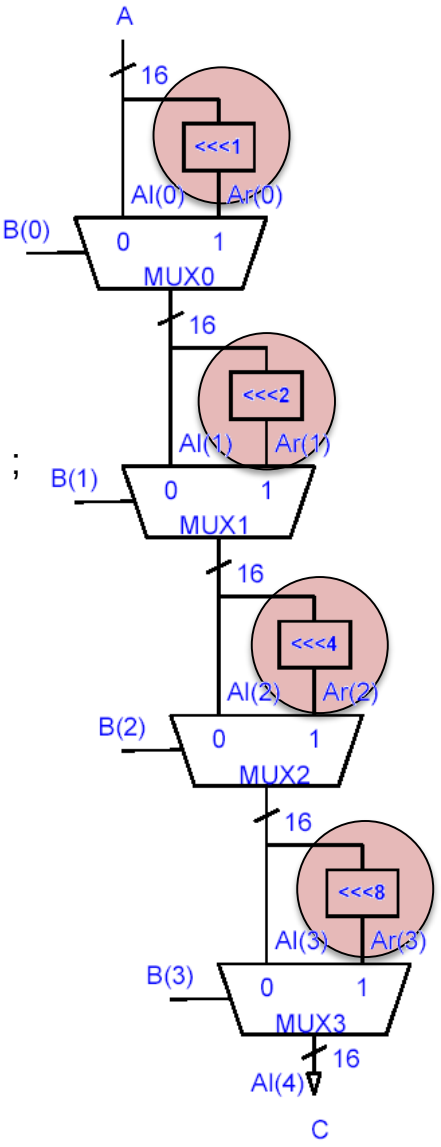
<<< L

a(15-L) a(15-L-1) . . . . . a(1) a(0) a(15) a(14) . . . . . a(15-L+2) a(15-L+1)

y <= a(15-L downto 0) & a(15 downto 15-L+1);

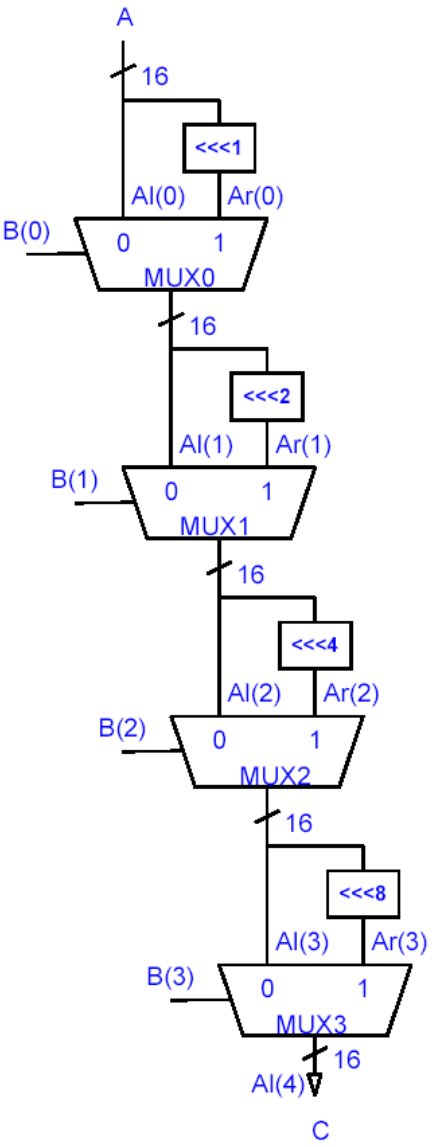
VHDL code for for a fixed 16-bit rotator

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fixed_rotator_left_16 IS
        GENERIC ( L : INTEGER := 1);
        PORT (          a                       : IN            STD_LOGIC_VECTOR(15 DOWNTO 0);
                        y                       : OUT           STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END fixed_rotator_left_16 ;

ARCHITECTURE dataflow OF fixed_rotator_left_16 IS
BEGIN
        y  <=  a(15-L downto 0) & a(15 downto 15-L+1);
END dataflow ;
```



Structural VHDL code for for a variable 16-bit rotator (1)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY variable_rotator_16 is
        PORT(
                        A : IN STD_LOGIC_VECTOR(15 downto 0);
                        B : IN STD_LOGIC_VECTOR(3 downto 0);
                        C : OUT STD_LOGIC_VECTOR(15 downto 0)
            );
END variable_rotator_16;
```

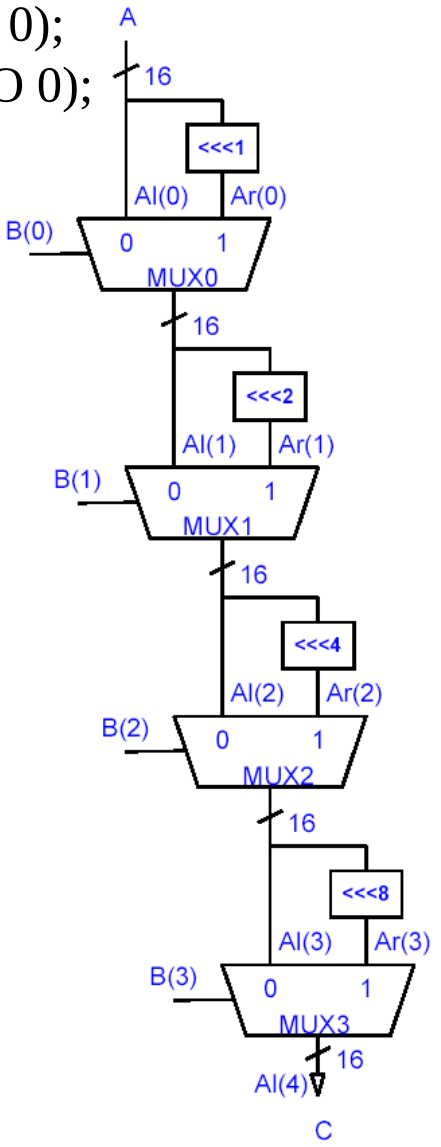


Structural VHDL code for for a variable 16-bit rotator (2)

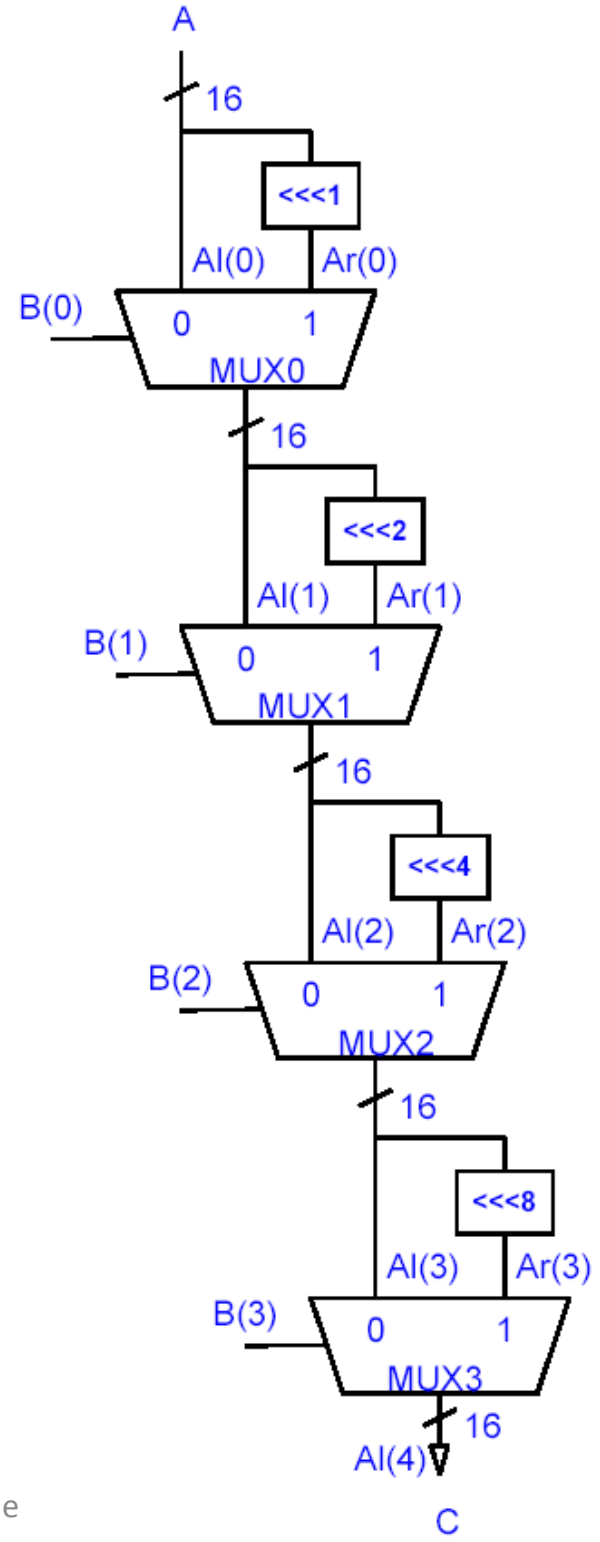
```
ARCHITECTURE structural OF variable_rotator_16 IS

  COMPONENT mux2to1_16
      PORT (     w0          : IN     STD_LOGIC_VECTOR(15 DOWNTO 0);
                 w1          : IN     STD_LOGIC_VECTOR(15 DOWNTO 0);
                 s           : IN     STD_LOGIC ;
                 f           : OUT    STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
  END COMPONENT ;

  COMPONENT fixed_rotator_left_16
      GENERIC ( L : INTEGER := 1);
      PORT (     a           : IN     STD_LOGIC_VECTOR(15 DOWNTO 0);
                 y           : OUT    STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
  END COMPONENT ;
```

Structural VHDL code for for a variable 16-bit rotator (3)

```
TYPE array1 IS ARRAY (0 to 4) OF STD_LOGIC_VECTOR(15 DOWNTO 0);
TYPE array2 IS ARRAY (0 to 3) OF STD_LOGIC_VECTORS(15 DOWNTO 0);
SIGNAL Al : array1;
SIGNAL Ar : array2;

BEGIN
      Al(0) <= A;
      G: FOR i IN 0 TO 3 GENERATE
          ROT_I: fixed_rotator_left_16
                    GENERIC MAP (L => 2** i)
                    PORT MAP ( a => Al(i) ,
                                        y => Ar(i));
           MUX_I:  mux2to1_16  PORT MAP (w0   => Al(i),
                                              w1   => Ar(i),
                                              s  => B(i),
                                              f  => Al(i+1));

      END GENERATE;
      C <= Al(4);
END variable_rotator_16;
```



Block diagram



Summary of Lecture 4

- Lab processor specifications
  - Datapath and control
  - Instruction Set
- Structural VHDL and tips for regular structures

- Additional Reading material:
  - Lab processor specifications
  - Book (complimentary to the slides):
    - 8.9

- Next Lecture 5:
  - Sequential circuits