Middle East Technical University - Department of Computer Engineering

CENG 232

Logic Design

Spring '2021-2022

Lab 4

Part 1 Due Date: 5 June 2022, Sunday, 23:55
Part 2 Due Date: 5 June 2022, Sunday, 23:55
No late submissions

1 Part 1: Polynomial Memory (50 pts)

In this part, you are expected to implement basic memories as 2 Verilog modules. The modules together take a binary number and an operation type (read/write) and then according to the operation type, either evaluate and store the result of the evaluations to the given memory index (write mode) or return the previous data from the given index of the memory (read mode).

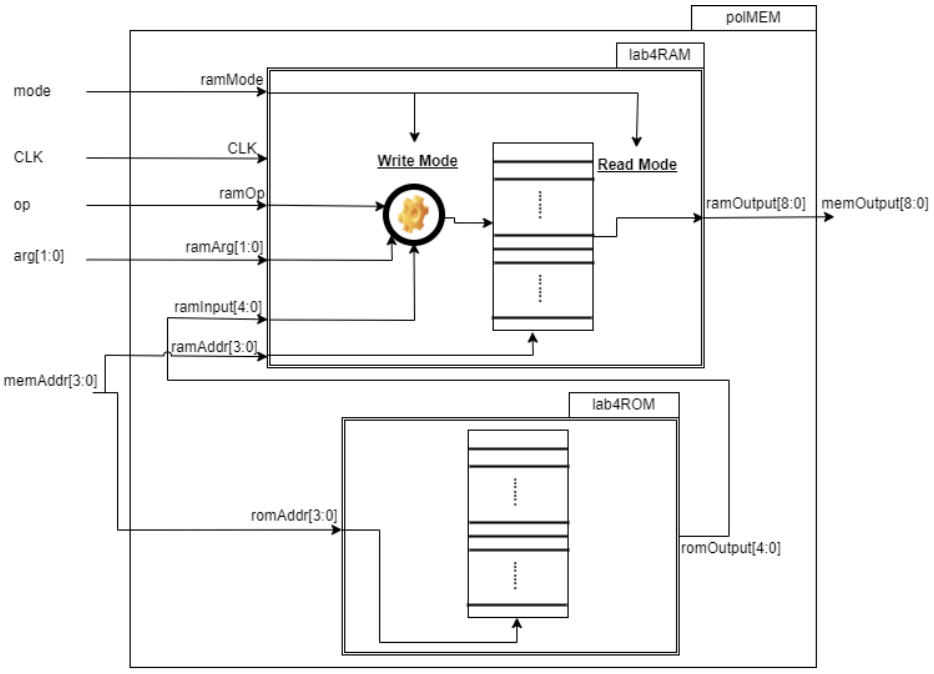

Figure 1: Illustration of the modules.

1.1 lab4ROM Module

The module **lab4ROM** will contain 16 registers. Each register holds a 5-bit binary number which represents the coefficients of the polynomial starting from a_4 through a_0. Each polynomial function will be in the form: $a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0$. The values of a_4 through a_0 can either get +1 or -1: These values are represented in a register as 0 and 1, respectively.

You should take into consideration the specifications below:

1. The lab4ROM works asynchronously, it is not triggered by a clock pulse.
2. It returns the cell value with the given index (**romAddr**) as output (**romOutput**) immediately.
3. Examples of 3 registers and the polynomials they represent are given in the below table:

Register Value	Function
11111	$-x^4 - x^3 - x^3 - x^2 - x$
00000	$x^4 + x^3 + x^3 + x^2 + x$
11000	$-x^4 - x^3 + x^3 + x^2 + x$

4. The values of ROM should be as provided in the following table:

Index	Register Value
0	00000
1	00001
2	00110
3	00111
4	01011
5	01100
6	01101
7	01110
8	11101
9	11110
10	11111
11	10000
12	10111
13	11000
14	11001
15	11010

Use the following lines as the port definition of the lab4ROM Module:

```
module lab4ROM (
input [3:0] romAddr,
output reg[4:0] romOutput);
```

1.2 lab4RAM Module

The module lab4RAM will contain 16 registers. Each register contains 9-bit binary number. You should take into consideration the specifications below:

1. Initially, the values of all RAM registers will be 0.
2. There are 2 modes in this module (**ramMode**):
   - 0 - read mode
   - 1 - write mode
3. Write mode is synchronous. It is triggered by the positive edge of the clock (CLK).
4. Read mode is asynchronous. That means, it is not triggered by a clock pulse.

5. In write mode, your task is to evaluate the polynomial/derivative of the polynomial (**ramInput**) with the given argument (**ramArg**) and store the result in the **ramAddr** location of lab4RAM.
   - The input **ramOp** is used to determine the type of the evaluation.
     - 0 - the polynomial in its original form will be evaluated
     - 1 - the derivative of the polynomial will be evaluated
   - The **ramArg** corresponds to the argument of the polynomial:
     - 00 $\Rightarrow$ +1
     - 01 $\Rightarrow$ +2
     - 10 $\Rightarrow$ -1
     - 11 $\Rightarrow$ -2
   - The evaluation result will be a 9-bit number where the most significant bit denotes the sign (0 if positive and 0, 1 if negative).
   - The examples are given in the below table :

ramInput	Corresponing Polynomial Func.	ramOp	ramArg	Evaluation	9-bit Result
00001	$P(X) = x^4 + x^3 + x^2 + x^1 - 1$	0 (Polynomial)	00	$P(1) = 1^4 + 1^3 + 1^2 + 1^1 - 1 = 3$	000000011
00001	$P(X) = x^4 + x^3 + x^2 + x^1 - 1$	1 (Derivative)	00	$P'(X) = 4x^3 + 3x^2 + 2x + 1$ $P'(1) = 4 + 3 + 2 + 1 = 10$	000001010
11111	$P(X) = -x^4 - x^3 - x^2 - x^1 - 1$	1 (Derivative)	01	$P'(X) = -4x^3 - 3x^2 - 2x - 1$ $P'(2) = -32 - 12 - 4 - 1 = -49$	100110001
01011	$P(X) = x^4 - x^3 + x^2 - x^1 - 1$	1 (Derivative)	10	$P'(X) = 4x^3 - 3x^2 + 2x - 1$ $P'(-1) = -4 - 3 - 2 - 1 = -10$	100001010

6. In read mode, no write operation to the memory is conducted. The value stored in the **ramAddr** location of the RAM will be returned in ramOutput.
7. The initial value of **ramOutput** should be 0. It can only be changed in read mode. It should retain the last read value when the module is in write mode.

Use the following lines as the port definition of the lab4RAM Module:

```
module lab4RAM (
input ramMode,
input [3:0] ramAddr,
input [4:0] ramInput,
input  ramOp,
input [1:0] ramArg,
input CLK,
output reg [8:0] ramOutput);
```

1.3 polMEM Module

This is the upper module, in which inputs and outputs of other modules are defined. The inputs and outputs of this module; mode, memAddr, op, arg, CLK and memOutput are distributed to lab4ROM and lab4RAM modules.

You should not edit this module.

Illustration of the overall Polynomial Memory system is given in Figure 1.

```
module polMEM(
input mode,
input [3:0] memAddr,
input op,
input [1:0] arg,
input CLK,
output wire [8:0] memOutput);
```

1.4 Deliverables

- Implement both modules in a single Verilog file. Upload only **lab4_1.v** file to ODTUClass system. Do **NOT** submit your testbenches. You can share your testbenches on ODTUClass discussion page.
- Submit the file through the ODTUClass system before the deadline given at the top.
- This is an individual work, any kind of cheating is not allowed.

2 Part 2: CENG Accumulator (50 pts)

You are given an assignment by your boss to implement a continuous accumulator chip for generic usage. Due to your superior knowledge of circuit design, you design this chip using verliog and validate your design using an FPGA Board.

Accumulator is required to execute loaded instructions on an infinite loop. In order to achieve this, accumulator has two modes, *Load Mode* and *Calculate Mode*. In load mode, user will give "instruction code - value pairs" that will be loaded to the memory of the chip. In calculate mode, these series of instruction value pairs will be calculated one by one. When all of the instructions are exhausted, chip will start from the beginning. Instruction code will be refereed as op (operation) code from now on.

The practical details of the system are described below:

1. As we have discussed earlier, there will be two modes, which are all synchronous and triggered by the **positive edge** of the clock:
   - Load Mode (0): User will **load** an opcode-value pair to the system.
   - Calculate Mode (1): Accumulator **execute** the current instruction and display the result.
2. Accumulator can hold a limited amount of opcode-value pairs. This limit is **32**.
3. If user tries to submit more than 32 operations, system should warn the user. (*cacheFull*)
4. If user tries to submit an invalid opcode-value pair, system should also give a warning. (*invalidOp*)
5. Accumulator mode can be changed at anytime (by setting *mode* input).
6. Accumulator executes the given instructions in an **infinite loop** manner.
7. Accumulator will take opcode-value pairs, this value will undergo an operation defined by the opcode with the result of previous instruction(s). Initial instructions may be operated with the value and **zero**.

   To clarify further:
   - Lets assume "**ADD-5**" is the first instruction and "**ADD-8**" is the second instruction. Currently, there are total of two instructions available in the accumulator.
   - After the first clock cycle, 5 will be added with zero and the result is saved (which is 5). Subsequent clock cycle will add 8 to the previous result.
   - Since; in calculate mode, the accumulator executes its instructions indefinitely, the result will be added with the 5 again in the next clock cycle.
8. Some instructions may require subsequent previously accumulated values and some instructions may not need a value at all. Please check the upcoming instruction section (Section 2.1).
9. Internally, the accumulator should do its operations over a 10-bit buffer. If the result exceeds the capacity of 10-bits, the accumulator should warn the user (*overflow*). However, the accumulator should continue working by truncating the result to the 10-bit range.

10. User can reset the system anytime by pressing a reset button (*reset*). Reset operation should clear all of the internal state of the accumulator. This operation works **synchronously** with respect to the clock, and this operation should be triggered with respect to the **positive edge** of the reset signal.

2.1 Instruction Set

Instructions consists of two parts; *opcode* and *value*. Opcode consists of **three bits** and value bit consists of **four bits**. By definition, there can be total of eight instructions however only six of them are used.

Instructions may operate on the given value, previous result and the second last result. In order to clarify the explanation of the instructions, these variables are notated as; value (v), last result (p_0), second last result (p_1). At most last two results may be required by an instruction. In all instructions, if (p_0) or (p_1) cannot be expressed logically, then inputs should be considered as **zero**. For example; if MAD instruction (please check below) is the very first instruction, both p_0 and p_1 are considered zero. If it is the second instruction, only p_1 is considered zero.

Accumulation buffers; namely p_0 and p_1, should be 10-bits long. Instructions can be seen on Table 1.

Instruction Name	**Op Code**	**Definition**	**Math Expression**
ADD	000	Adds the given 4-bit value with the p_0.	$p_0 + v$
ADD2	001	Adds the given 4-bit value with p_0 and p_1.	$p_0 + p_1 + v$
FMA	010	Multiplies the previous outputs p_0 and p_1 then adds the given 4-bit input to the result. FMA stands for "Fused multiply and add".	$p_0 \times p_1 + v$
-	011	**This instruction is not available.**	
POPC	100	This instruction counts the high bits (bits that are set to one) in p_0 and returns the result. This instruction **ignores** the 4-bit value. POPC stands for "population count".	
BREV	101	This instruction reverses the bits of the p_0 and returns the result (0^{th}-bit will be 9^{th} bit, 1^{st} bit will be 8^{th} bit and so on). For example; if p_0 is 0110010100 then result will be 0010100110. BREV stands for "bit reverse".	
SETR	110	This instruction is a special instruction that sets the loop start offset. Initially, when accumulator finishes all of the instructions it will return to the very first instruction (instruction zero). However; this instruction changes it to be the v^{th} instruction in the list. This instructions does not modify accumulated results. SETR stands for "set return".	
-	111	**This instruction is not available.**	

Table 1: CENG Accumulator Instruction Set

2.2 Example Usage: Fibonacci Series

				Current State					Next State				
Line No	mode	opCode	value	cacheFull	invalidOp	overflow	result	clk	cacheFull	invalidOp	overflow	result	Explanation
1									0	0	0	X	Initial state
2	0	000	0001	0	0	0	X	↑	0	0	0	X	User issued ADD instruction with a value of 1 Instruction is valid and saved on accumulators internal buffer. There are total of one instruction inside the accumulator.
3	0	011	XXXX	0	0	0	X	↑	0	1	0	X	Mistakenly user tried to issue an invalid instruction. *invalidOp* is set. There are still one instruction inside the accumulator.
4	0	001	0000	0	1	0	X	↑	0	0	0	X	User issued ADD2 instruction with value zero. *invalidOp* is cleared since this instruction is valid. There are two instruction inside the accumulator.
4	0	110	0011	0	0	0	X	↑	0	0	0	X	User issued SETR instruction with value three. Accumulators next iteration will start from the 3^{rd} (zero indexed) instruction. There are three instructions in the accumulator. Please note that; if user start executing now, this is an undefined behaviour. (There is no 3^{rd} instruction in the accumulator).
5	0	001	0000	0	0	0	X	↑	0	0	0	X	User issued ADD2 instruction with value zero. Undefined behaviour is resolved.
6	1	XXX	XXXX	0	0	0	X	↑	0	0	0	$(1)_{10}$	User changed the mode to execute mode. Very first instruction is immediately executed. There is no ‘‘previous’’ result thus, one is added with zero. *result* is one.
7	1	XXX	XXXX	0	0	0	$(1)_{10}$	↑	0	0	0	$(1)_{10}$	Second instruction (ADD2-0) is executed. ‘‘previous’’ is one but there is no second previous result. $0+0+1$ is executed, *result* is one.
8	1	XXX	XXXX	0	0	0	$(1)_{10}$	↑	0	0	0	$(1)_{10}$	Third instruction (SETR) is executed, internally loop return index is set to 3. *result* is still displayed and it is two. Please note that; this operation does not change the ‘‘previous’’ or ‘‘second previous’’ results.
9	1	XXX	XXXX	0	0	0	$(1)_{10}$	↑	0	0	0	$(2)_{10}$	Fourth instruction (ADD2-0) is executed, Since SETR instruction does not change the previous results, $0+1+1$ is executed. *result* is 2.
10	1	XXX	XXXX	0	0	0	$(2)_{10}$	↑	0	0	0	$(3)_{10}$	Fourth instruction (ADD2-0) is executed again. (SETR did set the loop start index to 3, which is this instruction) Since SETR instruction does not change the previous results, $0+1+2$ is executed. *result* is three.
11	1	XXX	XXXX	0	0	0	$(3)_{10}$	↑	0	0	0	$(5)_{10}$	(ADD2-0) is executed again (3^{rd}*instruction*) $(0+2+3)$. *result* is eight.
12							...						As you can see, accumulator is outputting Fibonacci series. Lets skip little further in order to demonstrate overflow situation. Assume that the user monitoring inputs every clock cycle and does not change the mode.
13	1	XXX	XXXX	0	0	0	$(610)_{10}$	↑	0	0	0	$(987)_{10}$	(ADD2-0) is executed again $(0+377+610)$. *result* is nine hundred eighty seven.
14	1	XXX	XXXX	0	0	0	$(987)_{10}$	↑	0	0	1	$(573)_{10}$	(ADD2-0) is executed again $(0+610+987)$. Result; which is 1597, exceeds the 10-bit buffer. It is truncated and become 573.
15	0	100	XXXX	0	0	1	$(573)_{10}$	↑	0	0	1	X	User changed the mode and adds (POPC) instruction. Internal instruction buffer has 5 instructions. Last instruction was overflowed therefore, overflow bit is still one.
16	1	XXX	XXXX	0	0	1	X	↑	0	0	1	$(536)_{10}$	User changed the mode again and the ADD2-0 (4^{th} instruction) is executed $(0+987+573)$. Result still overflows, overflow bit is still set. *result* is 536. Overflow bit is still one.
17	1	XXX	XXXX	0	0	1	$(536)_{10}$	↑	0	0	0	$(3)_{10}$	POPC instruction is executed. It operates on previous value ($(536)_{10}$). Total number of bits that is set on that number is ($(536)_{10} = (1000011000)_2$) 3. *result* is 3. This operation did not overflow, so overflow bit is lowered.
18	1	101	XXXX	0	0	0	$(3)_{10}$	↑	0	0	0	X	User adds BREV operation. (There are 6 instructions).
19							...						Lets show the full instruction buffer situation. Assume user continuously adding BREV operation.
20	1	101	XXXX	0	0	0	X	↑	0	0	0	X	User adds BREV operation. (There are 31 instructions).
21	1	101	XXXX	0	0	0	X	↑	0	0	0	X	User adds BREV operation. (There are 32 instructions).
22	1	101	XXXX	0	0	0	X	↑	1	0	0	X	User tries to add BREV operation. Since the instruction cache is full. *cacheFull* bit is set to one. (There are still 32 instructions in the cache).

In **clk** column of table above, "↑" represents the rising edge of the clock.

2.3 FAQ

1. **Q: What will happen when the very last instruction is SETR instruction? From where the next iteration will start?**

   A: Next instruction should be the value given by the SETR instruction (v^{th} instruction).

2. **Q: What if user set the loop start offset to a value that is larger than the current instruction count?**

   A: You can ignore this case. You can assume user is a perfect programmer and never do such mistake.

3. **Q: During execution mode, can user change the mode to load and add new instructions?**

   A: Yes, user can add additional instructions. After instructions are added, user will continue executing instructions where it was left off.

4. **Q: What happens to the overflow output when user is in load mode? Should it still be lit, or cleared during load mode and re-lit during execution mode? (Same goes for the cacheFull signal)**

   A: *cacheFull* and *overflow* signals should always be signalled when their cases are satisfied. However; invalidOp **should be cleared when the next instruction is valid, or user is changed to the execution mode and executed an instruction.**

5. **Q: I did not understand what you mean by "truncating" when an overflow occurs? Can you clarify it further?**

   A: Imagine p_0 is currently 1020, and next instruction is **ADD-15** instruction. After clock is triggered, $1020 + 15 = 1035$ is calculated. Since 10-bit buffer can hold 1023 as a maximum, *result* should be 11 $(1035 - 1024 = 11)$. Additionally, *overflow* bit should be set as well.

6. **Q: What would happen if both *reset* and *clk* is triggered exactly at the same time? Which one would have the precedence?**

   A: This case is physically impractical, since we assign the clock and reset signals to buttons. However during a test case simulation, you can ignore this case. Thus, you can implement it either way and it won't be checked.

7. **Q: Can you clarify the reset signal?**

   A: Reset signal should clear all of the warning bits, clears the inner instruction cache, and accumulation buffers. Inner loop offset should also be set to zero as well. Basically, accumulator should be on its initial state.

8. **Q: What should happen when user immediately tries to execute instructions without setting any instruction?**

   A: *result* should be zero. At every clock cycle, the accumulator should continue outputting zero.

2.4 Input/Output Specifications

Name	Type	Size
mode	Input	1
opCode	Input	3
value	Input	4
reset	Input	1
clk	Input	1
result	Output	10
cacheFull	Output	1
invalidOp	Output	1
overflow	Output	1

- **lab** is used for the selection of the mode.
  - $mode = 0 \Rightarrow$ Load Mode

- $mode = 1 \Rightarrow$ Execute (Calculate) Mode

- **opCode** represents the 3-bit opcode.
- **reset** is the reset signal for the accumulator.
- **clk** is the clock input for the module.
- **result** is the 10-bit result of the last instruction.
- **cacheFull** Warning about the opcode-value cache being full or not.
  - $cacheFull = 0 \Rightarrow$ Cache is **not** full, user can add more instructions.
  - $cacheFull = 1 \Rightarrow$ Cache is full, user cannot add any more instructions.
- **invalidOp** Warning about the validity of the opcode that is tried to be issued.
  - $invalidOp = 0 \Rightarrow$ Last instruction that was tried to be issued is valid.
  - $invalidOp = 1 \Rightarrow$ Last instruction that was being issued is an invalid instruction. (Section 2.1, opCode was either 011 or 111)
- **overflow** Warning about the result is overflowed or not.
  - $invalidOp = 0 \Rightarrow$ Last instruction result is **not** overflowed.
  - $invalidOp = 1 \Rightarrow$ Last instruction result is overflowed.

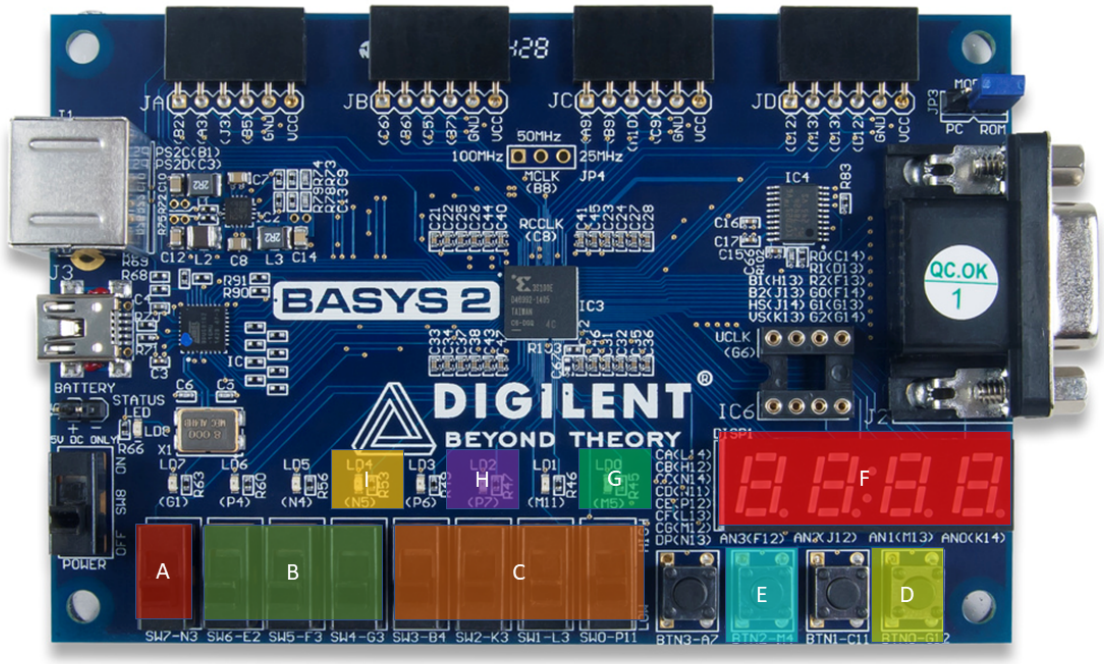

Figure 2: Board figure with labels

2.5 FPGA Implementation

You will be provided with a Board232.v file (and a ready-to-use Xilinx project), which will bind inputs and outputs of the FPGA board with your Verilog module. You are required to test your Verilog module on the FPGA boards.

Name	FPGA Board	Description	
mode	SW[7]	(A)	5 left-most switch
opCode	SW[6..4]	(B)	Most significant bit is SW6
value	SW[3..0]	(C)	Right-most switches, most significant bit is SW3
Clock(cl)	BTN0	(D)	Right-most button
reset	BTN2	(E)	Third right-most button
result	AN[3..0]	(F)	All four 7-seg displays
cacheFull	LD[0]	(G)	Leds
invlidOp	LD[2]	(H)	
overflow	LD[4]	(I)	

Table 2: Module I-O to FPGA Board I-O Mappings

2.6 Deliverables

- Implement your module in a single Verilog file. Upload only **lab4_2.v** file to ODTUClass system. Do **NOT** submit your testbenches, bit files or other project files. You can share your testbenches on ODTUClass discussion page.
- Submit the file through the ODTUClass system before the deadline given at the top.
- Use the ODTUClass discussion for any questions regarding the homework.
- This is an individual work, any kind of cheating is not allowed.