Ψηφιακά Συστήματα HW-1

ΕΡΓΑΣΤΗΡΙΑΚΕΣ ΑΣΚΗΣΕΙΣ

ΑΡΙΣΤΟΤΕΛΗΣ ΤΣΕΚΟΥΡΑΣ – ΔΙΠΛ. ΗΛ. ΜΗΧ. ΚΑΙ ΜΗΧ. ΥΠΟΛ.
ΒΑΣΙΛΗΣ ΠΑΥΛΙΔΗΣ – ΑΝ. ΚΑΘΗΓΗΤΗΣ, ΤΗΜΜΥ
ΑΠΘ | ΕΡΓΑΣΤΉΡΙΟ ΗΛΕΚΤΡΟΝΙΚΗΣ

Contents

Exercise1

An Arithmetic Logic Unit (ALU) is an important component of any RISC-V processor system. This unit is responsible for performing arithmetic operations, such as addition and subtraction, as well as logical operations, such as AND/OR. In this exercise, you will create an ALU that you will use for the RISC-V processor in a later exercise. You will also use the ALU to implement a simple "calculator."

Your ALU will be designed to implement the following operations: signed addition, signed subtraction, logical AND, logical OR, logical XOR, "Less than" comparison, and three different shift operations.

Start the exercise by creating a new Verilog file named**alu.v**. Define a new module named “alu” and add the following ports (the port names MUST be exact as defined in the table):

Port name	**Direction**	**Width** [bit number]	**Purpose**
op1	Entry	32	1's complement operator with respect to 2's
op2	Entry	32	2's complement operator
alu_op	Entry	4	Indicates which operation should be performed
zero	Exit	1	Indicates when the ALU result is zero
result	Exit	32	Result

The ALU you are going to create is a "32-bit" ALU which means that the inputs are 32-bits wide, and it produces a 32-bits wide output. The ALU will perform one of the different operations between the operands 'op1' and 'op2' based on the value of the input signal 'alu_op'.

This circuit is a "combination" circuit which means that the result of the circuit's output depends only on the inputs ('op1', 'op2' and 'alu_op') and there is no memory to store the previous state (there are no registers and therefore no need for a clock or reset signal input). The operation performed by the ALU is based on the four-bit input signal 'alu_op', as defined in the following table:

alu_op	**Act**	**Result**
0000	AND logic	op1 & op2
0001	Logic OR	op1 | op2
0010	Addition	op1 + op2
0110	Removal	op1 - op2

0100	Smaller than	op1 < op2
1000	Logical shift right by op2 bits	op1 >> op2[4:0]
1001	Logical shift left by op2 bits	op1 << op2[4:0]
1010	Arithmetic shift right by op2 bits	op1 >>> op2[4:0]
0101	XOR logic	op1 ^ op2

Attention: The “Less than” operation must be performed onpre-signed numbers Also for numerical sliding, op1 should beconverted to a signed number and the result then converted back to an unsigned number .

ALU multiplexer

The ALU circuit must include a multiplexer to select which of the functions to perform. You can specify the constants using the parameter directive in Verilog as follows:

```
parameter[3:0] ALUOP_SUB = 4'b0110;
```

You will need to create a constant for each of these nine ALU operation instructions. Add all the parameters into the file**alu.v**.

Exercise2

For this exercise, you will design a calculator circuit that uses the ALU you created in the previous exercise. This circuit will maintain a current value in a 16-bit accumulator register and allow the user to update the value by implementing any of the arithmetic and logical functions provided by your ALU.

Start by creating a new Verilog file named**calc.v**. Define a new module named calc and add the following ports (the port names MUST be exact to match the testbench):

Port name	**Direction**	**Width [number of bits]**	**Purpose**
clk	Entry	1	Clock
btnc	Entry	1	Central button
btnl	Entry	1	Left button
btnu	Entry	1	Up button
btnr	Entry	1	Right button
btnd	Entry	1	Down key

sw	Entry	16	Switches for data entry
led	Exit	16	LED for battery output

The two basic components of the circuit are a 16-bit accumulator to hold the current value of your calculator and the ALU that you created in the previous exercise. The relationship between the ALU and the accumulator should be drawn as shown in the following figure.

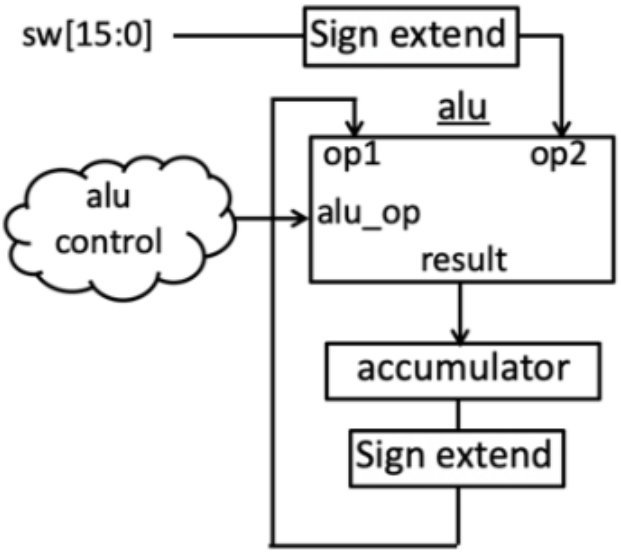

Fig. 1: Flowchart of the calculator.

Create a 16-bit register (called an accumulator) to hold the contents of the calculator's current value. Design your register as follows:

- Theaccumulator should be connected to the clock input.
- Theaccumulator should be reset to zeromodern with the press of a button**btnu.**
- The entrance ofaccumulator should be the 16 lower bits of the 32-bit result output of the ALU.
- Theaccumulator should be updatedmodern every time the "down" button is pressed (**btnd**).
- Connect the value ofaccumulator with the LED outputs.

Connect the inputs to the ALU as follows:

- Create a signal32-bit which is a sign-extended version of the 16-bit accumulator. Connect this signal to the ' input**op1**' of ALU.
- Create a signal32-bit which is a sign-extended version of the 16-bit switch inputs. Connect this signal to the '**op2**' of ALU.
- Connect the lower16 bits of output "**result**" at the input of the accumulator (as mentioned above).

- Create a new signal for '**alu_op**' as well as the logic for this signal as discussed below (Figures 2 – 5).

You can create a 32-bit sign-extended signal using the Verilog concatenation operator (i.e., repeat the leading bit of the sign-extended signal as many times as needed).

The signal '**alu_op**' determines which ALU operation will be performed. You will determine which operation to perform based on the value of the three keys:**btnl, btnc**and**btnr**You should create the combinational circuit of Figures 2 – 5 that produces the appropriate signal '**alu_op**' based on the value of these three keys. Implement the circuit**calc_enc.v**in **structural verilog** and connect it to the calculator circuit**calc.v**.

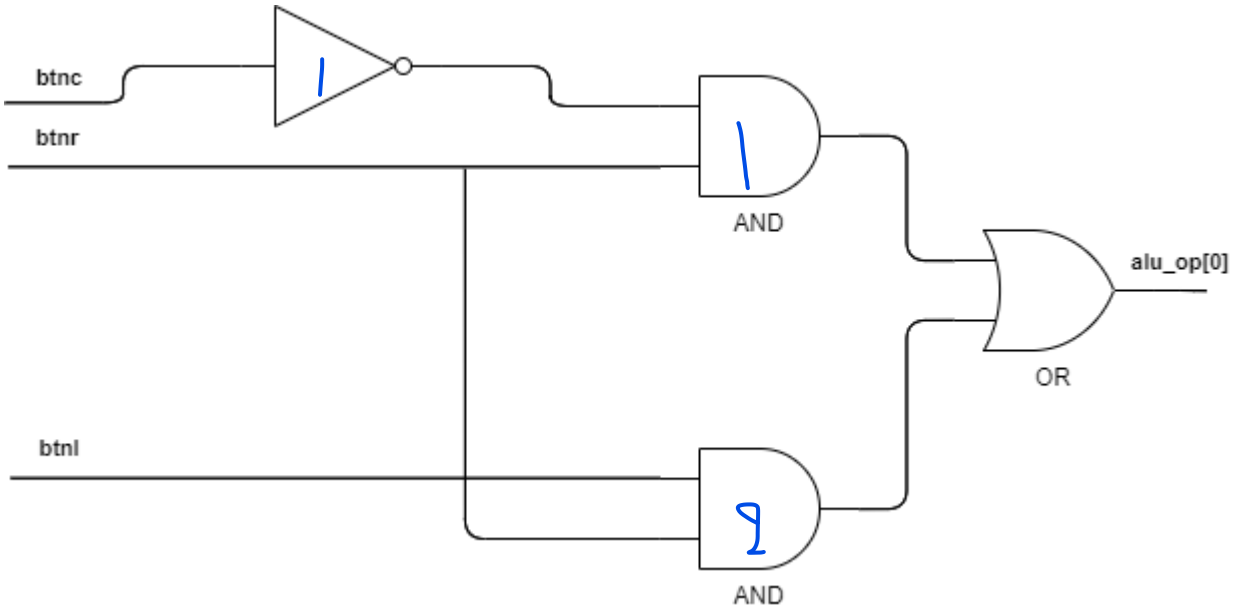

Fig. 2: Production ofalu_op[0] via btnr, btnl, btnc.

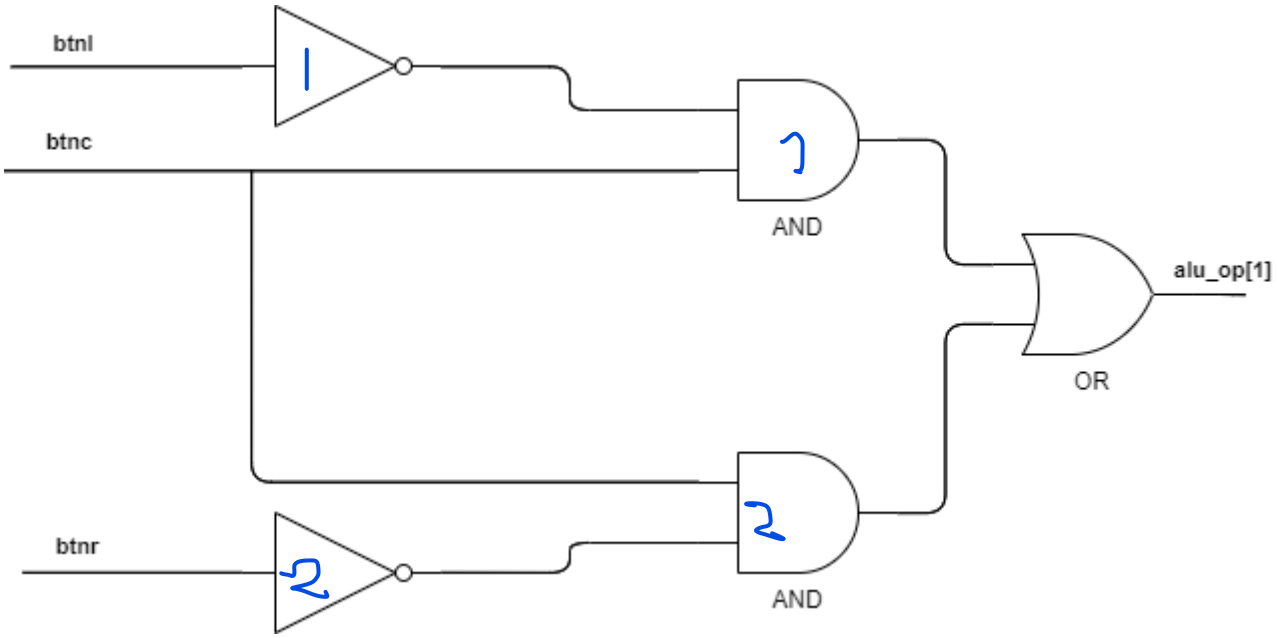

Fig. 3: Production ofalu_op[1] via btnr, btnl, btnc.

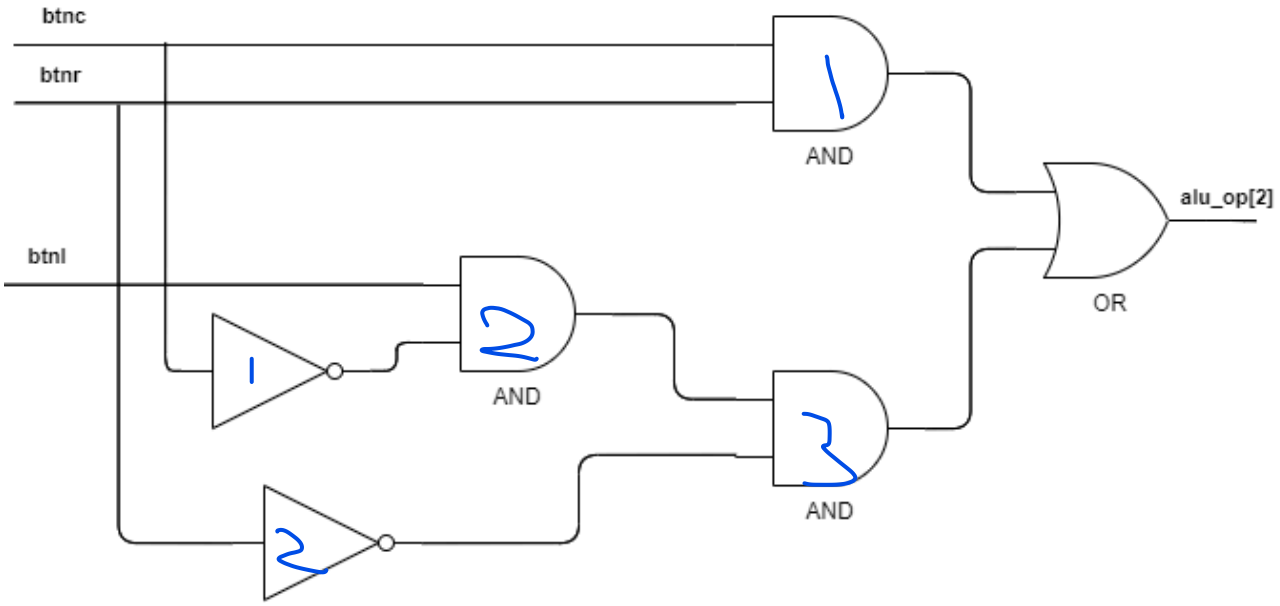

Fig. 4: Production ofalu_op[2] via btnr, btnl, btnc.

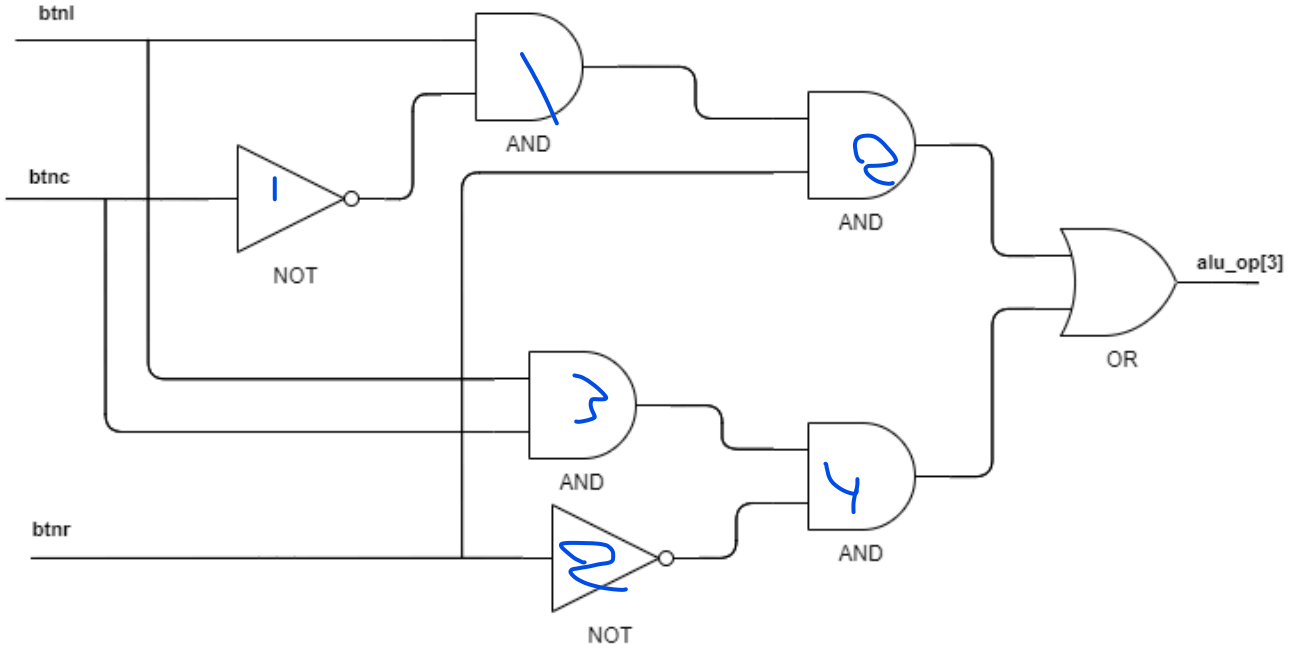

Fig. 5: Production ofalu_op[3] via btnr, btnl, btnc.

Testbench

Create a testbench that will check the correct operation of the calculator, as well as the correct operation of the ALU. Check the calculator in the following order for the values:

btnl, btnc, btnr (input)	**Previous value (adj.)**	**Switches (input)**	**Function in ALU**	**Expected Result**
(btnu for reset)	xxxx	xxxx	Reset	0x0

0,1,0	0x0	0x354a	ADD	0x354a
0,1,1	0x354a	0x1234	SUB	0x2316
0,0,1	0x2316	0x1001	OR	0x3317
0,0,0	0x3317	0xf0f0	AND	0x3010
1,1,1	0x3010	0x1fa2	XOR	0x2fb2
0,1,0	0x2fb2	0x6aa2	ADD	0x9a54
1.0.1	0x9a54	0x0004	Logical Shift Left	0xa540
1,1,0	0xa540	0x0001	Shift Right Arithmetic	0xd2a0
1.0.0	0xd2a0	0x46ff	Less Than	0x0001

Attention: The calculator will not support logical right shift.

Exercise3

The register file is another important component of any processor system. The register file stores the values of the registers used by the RISC-V processor. Almost all instructions read and/or write to the register file. In this exercise, you will create a register file that you will include in your processor. Start this exercise by creating a new Verilog file named **regfile.v**and add the following ports to this unit:

Port name	**Direction**	**Width [number of bits]**	**Purpose**
clk	Entry	1	Clock
readReg1	Entry	5	Address for read port 1
readReg2	Entry	5	Address for read port 2
writeReg	Entry	5	Address for registration port
writeData	Entry	DATAWIDTH	Data to be recorded
write	Entry	1	Control signal indicating recording
readData1	Exit	DATAWIDTH	Reading data from port 1
readData2	Exit	DATAWIDTH	Reading data from port 2

Where as "*DATAWIDTH*" is defined as a parameter of the "regfile" module, which should be equal to32 bits by default . Implement a 32× register file*DATAWIDTH*-bit. Initialize the 32 registers with zeros using an initial block and a for. Then use an always block to read the values of the registers from each output (readData1,

readData2) and depending on the write signal, write the data from writeData to the corresponding address you have as input. In case of writing a signal, you should pay attention to the case where the write address is the same as one of the read addresses. In this case,give priority to writing “writeData” .

Exercise4

A high-level diagram of the RISC-V processor you will design is shown in Figure 6. This diagram contains the following four elements:

- ☐ Command memory: An addressable memory that contains the instructions that implement the program being executed (the .data file you are given, as well as rom.v). Data is only read from this memory.
- ☐ Data memory: An addressable memory that contains user data (the ram.v file). It contains the stack and all user data used by the program. Data is read from and written to this memory.
- ☐ Data path: The internal operating units, registers, and multiplexers used to implement individual instructions.
- ☐ Control unit: It checks the operation of the data path.

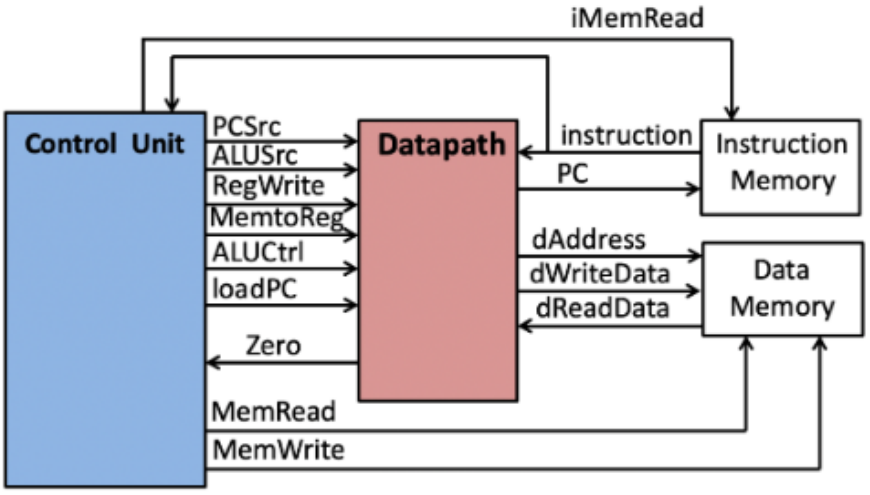

Fig. 6: Processor diagram. In the context of the Exercise4 the datapath will be designed.

You will design the data path unit in this exercise and the control unit in the next. You will not design the instruction memory and data memory (they are given to you).

Your data path will be able to support the following commands:

Register-Register: ADD, SUB, AND, OR, XOR, SLT, SLL, SRL, SRA

ALU Immediate: ADDI, ANDI, ORI, XORI, SLTI, SLLI, SRLI, SRAI

Memory: LW, SW

Branch: BEQ

Attention: Because your circuit will decode commandsRISC-V, it is important to understand how to properly decode and interpret these binary instructions. You will find the RISC-V instruction manual useful as you work through this and the next exercise. The instruction set table is on page 130 (pdf page 148). Pay attention to the R, I, S, B types.It is also recommended to use a RISC-V Instruction Encoder/Decoder to decode the instructions in ROM. You can find such decoders online.

Your implementation should include the ALU you created in Exercise 1 and the register file you created in Exercise 3. Start your exercise by creating a Verilog file named **datapath.v**with the doors summarized in table below along with a table showing the parameter you need to include in your Verilog file.

Parameter	**Width [number of bits]**	**Default value**
INITIAL_PC	32	0x00400000

Port Name	**Direction**	**Width** [bit number]	**Purpose**
clk	Entry	1	Clock
first	Entry	1	Synchronous Reset
instr	Entry	32	Command data from command memory
PCSrc	Entry	1	PC source
ALUSrc	Entry	1	Source of the 2nd operand of the ALU
RegWrite	Entry	1	Writing data to registers
MemToReg	Entry	1	Input multiplexer to registers
ALUCtrl	Entry	4	Indicates which operation the ALU should perform
loadPC	Entry	1	Updating the PC with a new price
PC	Exit	32	Program Counter
Zero	Exit	1	ALU reset indication
dAddress	Exit	32	Address for memory data
dWriteData	Exit	32	Data to be written to data memory
dReadData	Entry	32	Data to be read from data memory

WriteBackData	Exit	32	WriteBack data returning to registers

The purpose of the data path is to receive an instruction (instr), decode the instruction (recognizing the instruction type and instruction fields) to appropriately configure the ALU operands (for example, the ALU operand signals will be different for an ADD instruction and an SW instruction), and then feed the control signals (which are given to the input ports) where necessary.

The rest of this exercise will guide you through the process of designing the data path in Verilog. Consider the following five sections along with the Figure 7 provided.

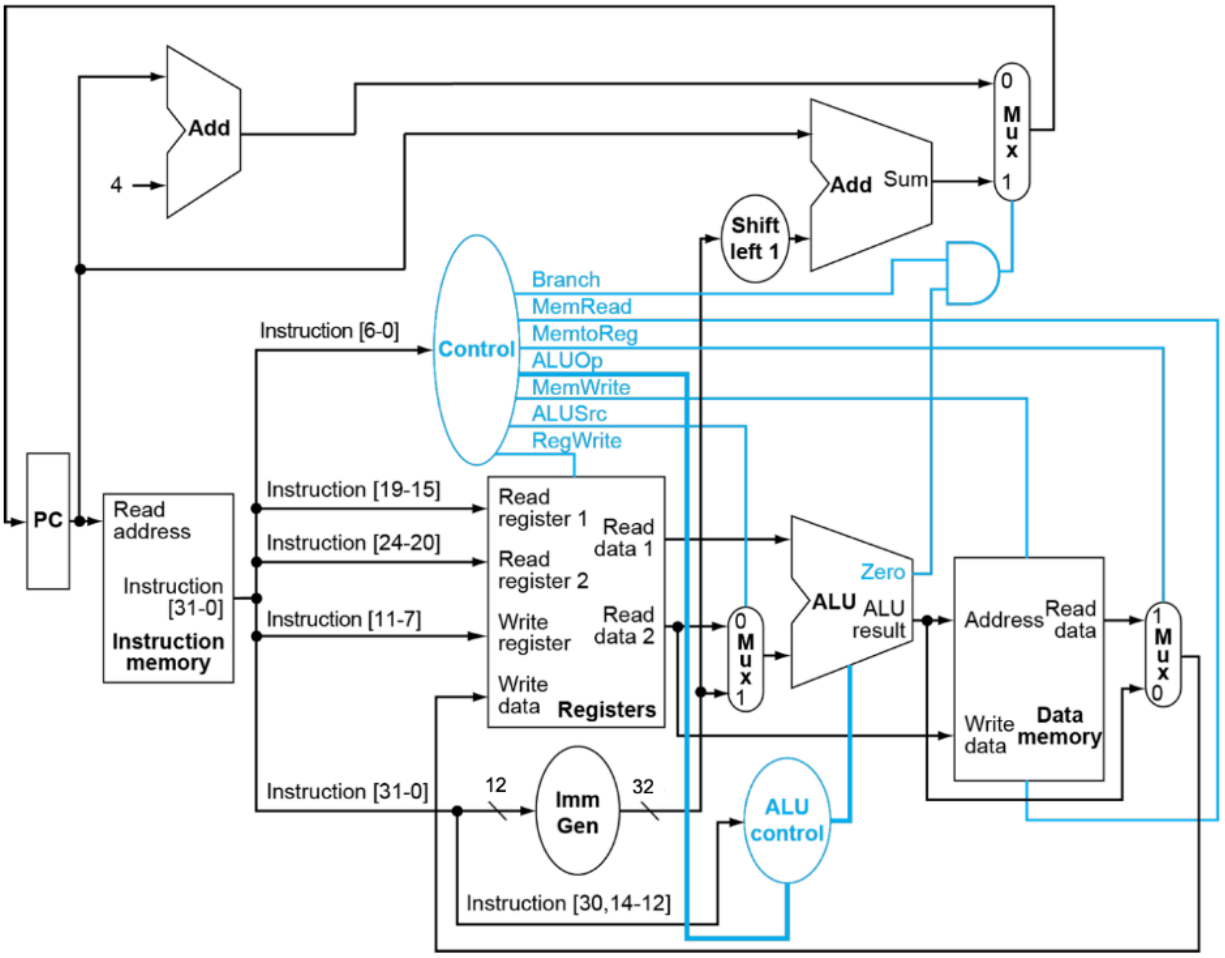

Fig. 7: Implementation diagram fordatapath (some signals are not shown in the diagram). The blue parts of the diagram are part of the next exercise,

therefore you don't need to implement them for her.

Program Counter (PC)

The first element of the data path is the Program Counter or PC. The PC is a register that contains the instruction memory address value. It indicates the location in instruction memory where the current instruction being executed is located.

Note that your Verilog file must support the "**INITIAL_PC**". This parameter is used to specify the initial value of the PC when the data path is reset with the synchronous rst signal. The default value is "0x00400000".

The input control signal**loadPC**is used to indicate that thePC should be updated on the next clock edge. When this signal is high, the PC should be updated with one of two different values:

- PC + 4 (when the program advances to the next instruction in memory), and
- PC + branch_offset (when the program branches).

The decision as to which of these two options is loaded on the PC is dictated by the control signal**PCSrc**The details of the branch shift will be described below.

Register File

Within the data path is the register file that contains the 32 processor registers. Add the register file from Exercise 3 to your data path. You will need to create decoding logic that determines the values at the address ports of the register file (i.e.,**readReg1, readReg2,** and **writeReg**) from the command that is ready from the command memory (i.e., the input**instr**). The register write signal should be connected to the control input**RegWrite**of the superior level (will be checked by the control unit completed in the next workshop).

Immediate Generation

For type I (direct) instructions, the data source entering the second read port of the ALU is the 12 MSBs of the instruction word (**instr[31:20]**), which has been sign-extended to a 32-bit value. Create a new signal in your data path that creates the sign-extended 32-bit value for the immediate data to be used by the ALU.

You will also need to create an immediate value for the S-type instructions (store) from the instruction read from the instruction memory. The immediate value for S-type instructions is different from the immediate value for I-type instructions (see pdf page 148) and therefore a separate signal will be needed. The same must be followed for B-type instructions (BEQ).

ALU

The next important component of the data path is the arithmetic logic unit (ALU). You will use the ALU from Exercise 1 in your data path circuit. The input to the first operand of the ALU (**op1**) is the output from the port **readData1**from the register file. The second operand in theALU can come from one of two sources: from the second read data port of the register file

(**readData2**) or from the extended immediate sign data generated by the command as described above in the “Immediate Generation” section.

The control signal**ALUSrc**dictates which input will be used. Create a multiplexer that selects between these two signals and drives the input**op2**herYour ALU. Connect the signal**ALUCtrl**of the top level at the entrance**alu_op**As described in Exercise 1, the signal **Zero**is used to indicate when the result ofALU is zero. This signal is used to decide when to take branches, as shown in Figure 7.

Branch Target

The data path logic must specify the branch target address used when a branch (BEQ) is made.

The target of a branch is calculated by adding the current PC value to the "branch offset" calculated by decoding the instr instruction (instruction type B). Implement the Verilog code required to calculate the branch offset. Also, generate the branch signal by adding this branch offset to the current PC. This branch target is loaded into the PC as described in the “Program Counter” section.

Write Back

The last piece of the data path is the “write-back” logic that determines which value is written to the register file. The value written to the register file is one of two values: the result of the ALU for conventional arithmetic and logic instructions, or the result of a memory read (**dReadData**). The control signal**MemtoReg**dictates which of these two signals is used by the file registers. Create a multiplexer that selects between these two signals and connect the output of the multiplexer to the input**writeData**of the registrar file. The data written to the registers must also be connected to the output **WriteBackData**of the circuit.

Exercise5

For this exercise, you will create a multi-cycle controller that executes each instruction in five clock cycles. Begin your exercise by creating a Verilog file named**top_proc.v**The ports that the module must have are the following:

Parameter	**Width [number of bits]**	**Default value**
INITIAL_PC	32	0x00400000

Port Name	**Direction**	**Width** [bit number]	**Purpose**
clk	Entry	1	Clock

first	Entry	1	Synchronous Reset
instr	Entry	32	Command data from command memory
dReadData	Entry	32	Reading data from data memory
PC	Exit	32	Program Counter
dAddress	Exit	32	Address for memory data
dWriteData	Exit	32	Data to be written to data memory
MemRead	Exit	1	Control signal indicating memory read
MemWrite	Exit	1	Control signal indicating memory write
WriteBackData	Exit	32	Data written to registers (for debugging)

As in the previous exercise, the verilog file top_proc.v should have a parameter for INITIAL_PC. The following steps will guide you through the process of creating the control unit.

Datapath

The first step is to initialize the datapath circuit that you created in the previous exercise. When you initialize your datapath, you will need to pass the parameter **INITIAL_PC**of the unittop_proc in your datapath module. This parameter indicates the initial value of the command to be executed.

Connect the following datapath ports to the corresponding port of the top_proc module:

- PC
- instr
- dAddress
- dReadData
- dWriteData

The remaining logic within this module will generate the control signals that control the datapath module and will be described in more detail below.

FSM

The next step for the control unit is to create the five-state machine that runs through the following five steps of executing an instruction in sequence:

Situation	**Purpose**
IF	Instruction Fetch: Providing the PC with instructions in memory

ID	Instruction Decode: Decode the received instruction and start accessing the registers
EX	Execute: Execute the operation in the ALU
MEM	Memory: Perform memory access (for lw/sw)
WB	Write Back: Writing new data to the registers

The primary purpose of this exercise is to implement the state machine and create the control signals for the data path module (the blue section in Figure 7). Figure 8 and the following sections will help you understand the logic you need to create the control signals.

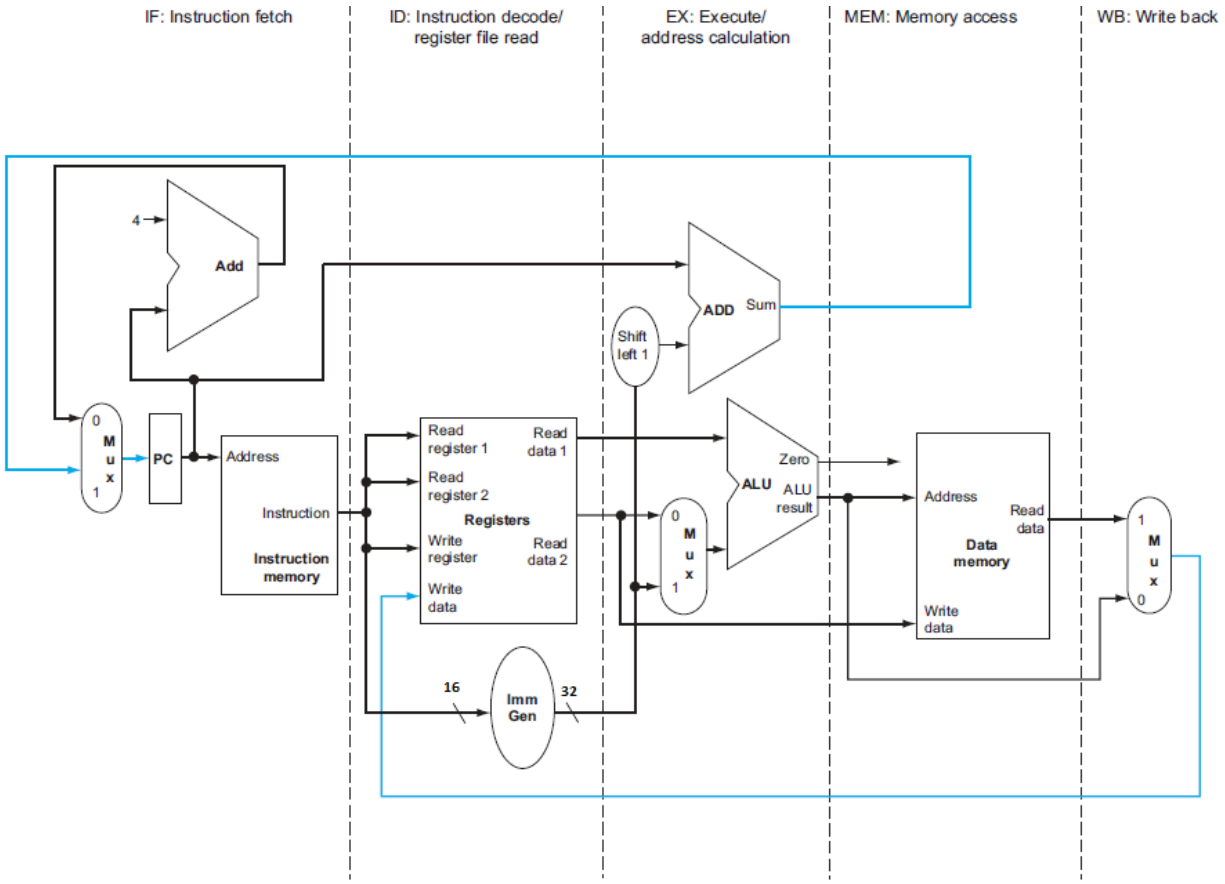

Fig. 8: Separation of the diagramdatapath in each of the 5 states.

ALUCtrl

The '**ALUCtrl**' is a four-bit control signal that dictates the operation of the ALU (ALU Control of Figure 7). The generation of the ' signal**ALUCtrl**' requires multi-level decoding that includes the ' field**opcode**' of a command (**instr**) as well as the bits '**function3**' and '**function7**'. The decoding should define the operation of the ALU based on the instruction as follows:

LW/SW: addition

BEQ: removal

Other ALU functions: operation based on funct3 and funct7 (including shift commands)

Produce the signal '**ALUCtrl**' based on the data above as well as on page 148 of the pdf. Also take into account the table in Exercise 1.

ALUSrc

When**ALUSrc**'== 0, the output of read register 2 is used as input to operand 2 of the ALU. When '**ALUSrc**'== 1, immediate data is used for input to operand 2 of the ALU. Immediate data is only required for the 'load', 'store' and 'ALU Immediate' instructions. You can specify the value of '**ALUSrc**' directly from the opcode bits of the instruction. As before use the data on page 148. This signal does not depend on the state of the FSM.

MemRead and MemWrite

Two signals control the data memory interface: '**MemRead**' and '**MemWrite**'. The signal '**MemRead**' is set for the 'load' commands and the 'signal'**MemWrite**' for 'store' commands. Unlike the previous signals, these signals must only be set during the ' **MEM**' of the FSM machine.

MemtoReg and RegWrite

The signal '**RegWrite**' is used to control the writing of values to the register file. For this five-cycle execution sequence, '**RegWrite**' should be set to '1' only during the ' state**WB**' of the instruction state machine. Some instructions do not write back to the register file, so this signal will remain low for the entire five clock cycle instruction sequence.

The signal '**MemtoReg**' is based on the current instruction and is set to '1' only when a 'load' instruction is executed. This signal does not depend on the current state (it does not matter what this signal is during all states except the '**WB**').

loadPC and PCSrc

The signal '**loadPC**' is used to update the PC with a new value. The ' signal**loadPC**' must be set to '1' during the ' state**WriteBack**' of each instruction (thus loading a new PC value before entering the next state**IF**).

The signal '**PCSrc**' is used to indicate which value should be loaded into the PC: either PC+4 for normal instructions, or PC+branch_offset for operations taken by branching. This signal is based on both the current instruction and the value of the ' signal.**Zero**' (see Figure 7). This signal must be set to '1' when the following conditions exist: (1) the current command is a "function"**BEQ**" and (2) the**Zero**is equal to '1'.

Testbench

Create a testbench "**top_proc_tb.v**" which will check all the supported commands, as presented in the previous exercise and as shown below:

Register-Register: ADD, SUB, AND, OR, XOR, SLT, SLL, SRL, SRA

ALU Immediate: ADDI, ANDI, ORI, XORI, SLTI, SLLI, SRLI, SRAI

Memory: LW, SW

Branch: BEQ

Use the memory files provided for the instruction memory and the data memory, as well as the file with the instruction initialization (.data) and run simulations.

Exercise Deliverables1-5

For the above exercises you must submit a report containing the following in the following order:

1) The file**alu.v**of the exercise1.
2) The files**calc.v, calc_enc.v, calc_tb.v**of the exercise2.
3) The file**regfile.v**of the exercise3.
4) The file**datapath.v**of the exercise4.
5) The files**top_proc.v**and**top_proc_tb.v**of the exercise5.

In your report please include**schematic diagram fromFSM** of the Exercise5 as well as **simulation waveforms** from the Exercises2 and 5. For Exercise 5 you will need to write the instructions that the processor runs from ROM (using online RISC-V decoders will help you with this). Also, add brief references to the process you followed to design and implement each module from each exercise.

The report must be submitted as1 pdf file where your name and your AEM will be mentioned. Otherwise, no grade will be given to the assignment.

Rating: The assignment is individual and mandatory, is graded with a maximum of 30 and is equivalent to 30% of the total course grade (i.e. up to 3 points). Failure to submit or late submission of the assignment is equivalent to a loss of three points in the total grade. Also, the grading is based on the overall picture of the report.

The deadline for submitting the work is:**January 21, 2025 at 23:59**, only through the elearning platform.

Reports

ECEN 323: Computer Organization Home • ECEn 323: Computer Organization. Available at: https://ecen323wiki.groups.et.byu.net/ (Accessed: 12 November 2023).