Assignment 04
MIPS building blocks

1 Objectives

To design and test the basic Microprocessor without Interlocked Pipeline Stage (MIPS) building blocks using SystemVerilog.

1. To design and test an Arithmetic and Logic Unit (ALU) using SystemVerilog.
2. To design and test a sign extender using SystemVerilog.
3. To design and test a Register File (RF) using SystemVerilog.
4. To design and test an Instruction Memory (IM) using SystemVerilog.
5. To design and test a Data Memory (DM) using SystemVerilog.

2 Deadline

23:59 hours on Wednesday September 30th 2020.

3 Teamwork policy

This is an individual assignment.

4 Pre-requisites

It is assumed that you are familiar with working with ModelSim and Quartus. If you require assistance, you can refer to the first assignment tutorial.

5 Background

The following sections provide some background to the designs for this assignment.

5.1 ALU

An Arithmetic and Logic Unit (ALU) is the component of a Microprocessor (μP) that performs the most basic arithmetic and logical operations. The schematic symbol of an ALU is shown in Figure 1.

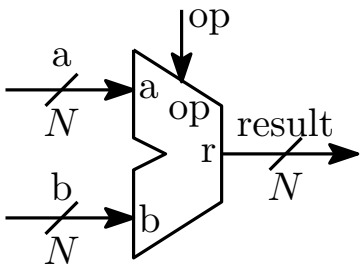

Figure 1: ALU schematic symbol.

In Figure 1, the ALU receives two N-bit operands, `a` and `b`, as well as a signal `op`, which indicates the type of operation the ALU must perform. More specifically, the value of `op` specifies whether the ALU should perform an addition, a subtraction or any bitwise logical operation such as `AND`, `OR`, `XOR`, *etc*, between the two input operands. The ALU delivers an N-bit output, labelled as `result` in Figure 1, which corresponds to the result of the arithmetic or logical operation.

5.2 Sign extender

A sign extender is a basic building block of a μP that takes an N-bit input `data_in` and generates an M-bit output `data_out`, as shown in Figure 2.

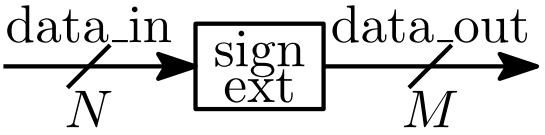

Figure 2: Sign extender schematic symbol.

Here, it is assumed that $N < M$. A sign extender increases the number of bits of a signed number while preserving the number's sign. This is achieved by replicating the sign bit, the Most Significant Bit (MSB) in 2's complement, of `data_in` $(M - N)$ times and then appending these replicated bits as its MSBs. For example, assume that $N = 5$ and $M = 8$. Here, the sign extender must generate an 8-bit output `data_out` from a 5-bit input `data_in`. In order to achieve this, this blocks replicates the sign bit of `data_in` $(M - N) = (8 - 5) = 3$ times. Furthermore, if `data_in = 5'b11000`, representing the decimal number -8 in 2's complement, its 8-bit sign-extended version corresponds to `data_out = 8'b11110000`, which also corresponds to the decimal value -8.

One important aspect of this block is that it must preserve the sign of the input data and not only increase the number of bits by doing zero-padding. This may lead to an erroneous result. Consider the same scenario as above, with $N = 5$ and $M = 8$ and `data_in = 5'b11000`, representing decimal number -8. If we carelessly increase the number of bits from 5 to 8 by adding zeros, then we will generate `data_in = 8'b00011000`, which corresponds to decimal value $+24$, rather than the expected value of -8.

5.3 Register File

One of the main characteristics of a Reduced Instruction Set Computer (RISC) Instruction Set Architecture (ISA) is the load-store approach. This implies that all operands used in the ALU must be retrieved from a register source instead of main memory. A Register File (RF) is simply a collection of registers, or memory elements, in which any register may be written to

or read from at any clock cycle. There are 32 registers in Microprocessor without Interlocked Pipeline Stage (MIPS) ISA, some of which are used to store the status of the μP and some other to store temporary data used by the user program. Figure 3 shows the schematic symbol of a simple RF that may be used in a MIPS μP.

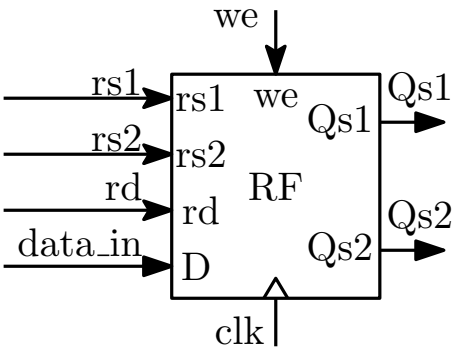

Figure 3: RF schematic symbol.

The RF of Figure 3 outputs `Qs1` and `Qs2`, which are the contents of the two read register `rs1` and `rs2`, respectively. These reads are performed at any given time. However, writes of input `data_in` in write address `rd` are controlled by the write enable control signal `we`, which must be asserted for a write to occur at the clock edge determined by `clk`.

5.4 Instruction Memory

As its name suggests, an Instruction Memory (IM) stores the instructions to be performed by the MIPS. This memory is a Read Only Memory (ROM), whose values are initialized by the programmer prior the execution of the very first instruction by the MIPS. Figure 4 shows the schematic symbol of an IM.

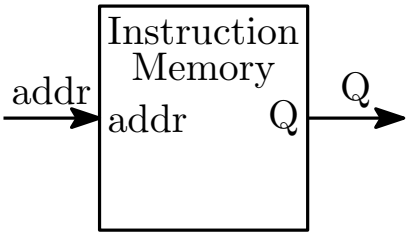

Figure 4: IM schematic symbol.

Here, output `Q` outputs the data stored in memory address `addr`.

5.5 Data Memory

As its name suggests, a Data Memory (DM) stores temporary data used by the user program. DM is basically a Random Access Memory (RAM) whose values are read/written according to the user program. Figure 5 shows the schematic symbol of an DM.

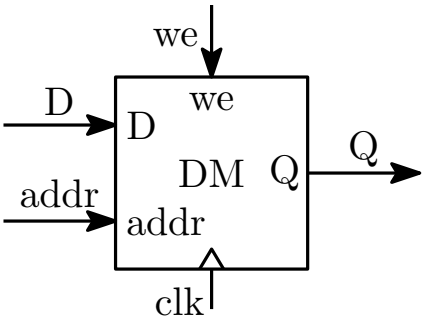

Figure 5: DM schematic symbol.

Here, `we` is a control signal that enables writes into the memory. More specifically, if `we` is asserted to logic 1, the input data `D` will be stored in address `addr` at the rising edge of `clk`. If `we` has the value of 0, input data `D` will be ignored. Output `Q` outputs the data stored in memory address `addr` and it is not dependant on `we` or `clk`.

6 Design specifications

The following sections describe the specifications for all the required MIPS building blocks. Please make sure you use the names provided in the specifications for both the file names and the ports of each module.

6.1 ALU

Your ALU design must be done in a SystemVerilog file named `alu.sv`, which must contain a `module` named `alu` and the port definition of Table 1. For this assignment, assume a value of `DATA_WIDTH=16`. However, bear in mind that your ALU design **must** be able to correctly perform all operations regardless of the value assign to the `parameter` `DATA_WIDTH`. This `parameter` must be declared in a SystemVerilog `package` named `mips_pkg.sv`.

Table 1: ALU ports.

Port name	Direction	Data type	Width	Description
`a`	`input`	logic signed	`DATA_WIDTH`	Operand a.
`b`	`input`	logic signed	`DATA_WIDTH`	Operand b.
`op`	`input`	`alu_op_t`	-*	Operation to be performed.
`result`	`output`	logic signed	`DATA_WIDTH`	Result from the arithmetic or logical operation.
`zero`	`output`	logic	1	Zero flag. Active high when result is exactly zero.
`neg`	`output`	logic	1	Negative flag. Active high when result is negative.
`grt`	`output`	logic	1	Greater-than flag. Active high when `a` is greater than `b`.
`eq`	`output`	logic	1	Equal flag. Active high when `a` and `b` have the same value.

* There's no need to specify the width of this port. This data type is specified in `mips_pkg.sv`.

The operations that the ALU must perform are described in Table 2. These operations must

be defined in a user-defined `enum` data type named `alu_op_t` and declared in a SystemVerilog `package` named `mips_pkg.sv`. Moreover, Table 2 specifies the name of the `enum` you must use for the different ALU operations.

Table 2: ALU operations.

Operation type	Enum name	Meaning
Arithmetic	`ALU_ADD`	Addition
	`ALU_SUB`	Subtraction
Logical	`ALU_NAND`	Logical NAND
	`ALU_NOR`	Logical NOR
	`ALU_XNOR`	Logical XNOR
	`ALU_AND`	Logical AND
	`ALU_OR`	Logical OR
	`ALU_XOR`	Logical XOR
Shift	`ALU_SLL`	Shift left logical
	`ALU_SRL`	Shift right logical
	`ALU_SLA`	Shift left arithmetic
	`ALU_SRA`	Shift right arithmetic
Load with immediate	`ALU_LUI`	Load upper with immediate
	`ALU_LLI`	Load lower with immediate
Comparison	`ALU_CMP`	Compares `a` and `b`

It is important to note that `zero`, `neg`, `grt` and `eq` flags of Table 1 must be updated for every operation of Table 2. More specifically, these outputs must be calculated for each of the 15 different ALU operations. The following sections provide some information about each type of operation of Table 2

Arithmetic operations

Arithmetic operations must be performed over signed operands.

Logical operations

All logical operations are bitwise operations.

Shift operations

Operand `a` indicates the shift amount for `b`. For example, assume the following scenarios.

Table 3: ALU shift operations example.

op	a	b	result	zero	neg	grt	eq
ALU_SLL	16'h0004	16'hFEDC	16'hEDC0	0	1	1	0
ALU_SRA	16'h0004	16'hFEDC	16'hFFED	0	1	1	0

Load with immediate operations

The lower half of operand `b`, *i.e.* bits `[DATA_WIDTH/2-1:0]`, are used to load the upper or lower part of the output `result`. The other half of `result` must be padded with zeros. For example, assume `DATA_WIDTH=16`. In this scenario, `result` is calculated as follows.

```
ALU_LUI: result = {b[7:0],8'b0}

ALU_LLI: result = {8'b0,b[7:0]}
```

Table 4 provides a more thorough example of load with immediate operations considering all ALU inputs and outputs. Here, note that the value of operand `a` is only relevant for the flags.

Table 4: ALU load operations.

op	a	b	result	zero	neg	grt	eq
ALU_LUI	16'h0000	16'hFEDC	16'hDC00	0	1	1	0
ALU_LLI	16'h8888	16'hFEDC	16'h00DC	0	0	0	0

Comparison

Output `result` must be set to 0 and all the flags must be updated accordingly. For example, assume the following scenarios.

Table 5: ALU comparison operation example.

op	a	b	result	zero	neg	grt	eq
ALU_CMP	4	-3	0	1	0	1	0
ALU_CMP	-1	-1	0	1	0	0	1
ALU_CMP	-7	0	0	1	0	0	0

6.2 Sign extender

Your sign extender design must be done in a SystemVerilog file named `sgn_ext.sv`, which must contain a `module` named `sgn_ext` and the port definition of Table 6.

Table 6: Sign extender ports.

Port name	Direction	Data type	Width	Description
`sign`	`input`	logic	`1`	Determines whether `data_in` should be sign- or zero-extended.
`data_in`	`input`	logic	`DATA_WIDTH_IN`	Input data.
`data_out`	`output`	logic	`DATA_WIDTH_OUT`	Sign-extended output data.

Parameters `DATA_WIDTH_IN` and `DATA_WIDTH_OUT` must be included as part of the `module` declaration and not in `mips_pkg.sv`.

Note that in contrast to the sign extender described in Section 5.2, you are required to include the option to perform zero-extension. This will be achieved by the input signal `sign`. More specifically, if input `sign = 1`, output `data_out` should be sign-extended. By contrast, if input `sign = 0`, `output data_out` should be zero-extended. For example, assume the following scenarios for the case when `DATA_WIDTH_IN = 4` and `DATA_WIDTH_OUT = 8`.

Table 7: Sign extender operation example.

sign	data_in	data_out
0	4'h8	8'h08
0	4'h7	8'h07
1	4'h8	8'hF8
1	4'h7	8'h07

6.3 Register File

Your RF design must be described in a SystemVerilog file named `rf.sv`, which must contain a `module` named `rf` and the port definition of Table 8. For this assignment, assume a value of `DATA_WIDTH=16` and `RF_N_WORDS=32`. However, bear in mind that your design **must** be able read/write data from/to the RF correctly regardless of the number of addresses or the width of the input data. These `parameter` must be declared in the `mips_pkg.sv`.

Table 8: Register File ports.

Port name	Direction	Data type	Width	Description
clk	input	logic	1	Clock
asyn_n_rst	input	logic	1	Active-low asynchronous reset
we	input	logic	1	Write enable
rs1	input	logic	RF_ADDRESS_WIDTH*	First read address
rs2	input	logic	RF_ADDRESS_WIDTH*	Second read address
rd	input	logic	RF_ADDRESS_WIDTH*	Write address
data_in	input	logic	DATA_WIDTH	Data to be stored
Qs1	output	logic	DATA_WIDTH	Data from read from rs1
Qs2	output	logic	DATA_WIDTH	Data from read from rs2

* RF_ADDRESS_WIDTH=$\lceil \log_2$(RF_N_WORDS)$\rceil$

Note that asyn_n_rst input of Table 8 is not depicted in Figure 3 for simplification purposes only.

6.4 Instruction Memory

Your IM design must be described in a SystemVerilog file named im.sv, which must contain a module named im and the port definition of Table 9.

Table 9: Instruction Memory ports.

Port name	Direction	Data type	Width	Description
init_file	-	parameter	-	Name of IM initialization file.
addr	input	logic	IM_ADDRESS_WIDTH*	Read address
Q	output	logic	INSTRUCTION_WIDTH	Data from read from addr

* IM_ADDRESS_WIDTH=$\lceil \log_2$(IM_N_WORDS)$\rceil$

For this assignment, you may use INSTRUCTION_WIDTH=32 and IM_N_WORDS=64. However, bear in mind that your design **must** be parameterizable and able read data from the IM correctly regardless of the number of addresses or the width of data stored in the memory. These parameter must be declared in the mips_pkg.sv.

HINT. Your IM design must initialize the contents of the memory by loading a text file defined in module parameter init_file. For this purpose, you may like to investigate the SystemVerilog directives $readmemb and $readmemh.

6.5 Data Memory

Your DM design must be described in a SystemVerilog file named dm.sv, which must contain a module named dm and the port definition of Table 10.

For this assignment, you may use DATA_WIDTH=16 and DM_N_WORDS=64. However, bear in mind that your design **must** be parameterizable and able read/write data from/to the DM correctly regardless of the number of addresses or data width. These parameter must be declared in the mips_pkg.sv.

HINT. Your DM does not require a reset signal. As a result of this, you may simply initialize the DM contents directly in your testbench.

Table 10: Data Memory ports.

Port name	Direction	Data type	Width	Description
`clk`	`input`	logic	1	Clock
`we`	`input`	logic	1	Write enable
`D`	`input`	logic	`DATA_WIDTH`	Data input
`addr`	`input`	logic	`DM_ADDRESS_WIDTH`*	Read address
`Q`	`output`	logic	`DATA_WIDTH`	Data from read from `addr`

* `DM_ADDRESS_WIDTH`=$\lceil \log_2(\texttt{DM\_N\_WORDS}) \rceil$

7 Grading criteria

For this assignment, I will consider the following grading criteria.

1. **The correct functionality of your designs.** I will use my own testbenches in order to automatically stress your designs and verify that they perform the tasks according to the specifications. For example, I will try different values for the parameters in your designs and I expect them to still perform according to the specifications. This is why it is paramount that you follow the name convention specified for file names and port names. Moreover, it is important that your designs and testbenches compile in ModelSim without errors. **Your maximum grade for this assignment will automatically drop to 50/100 should ModelSim trigger a compilation or simulation error.**

2. **The quality of your testbenches.** Even though I will use my own testbenches, I expect you to consider a thorough and concious set of test scenarios. In this way, you should be able to spot any mismatches between the expected results and the actual results provided by your designs.

3. **Your designs must be synthesized in Quartus without latches and without errors.** Warnings are tolerated at this point. **Your maximum grade for this assignment will automatically drop to 50/100 should Quartus trigger a synthesis error or generate unwanted latches.** Remember that a design is not useful if it can't be synthesized.

8 Deliverables and Submission instructions

Prepare a single `zip` file containing the following files.

1. `mips_pkg.sv`.
2. `alu.sv`.
3. `sgn_ext.sv`.
4. `rf.sv`.
5. `im.sv`.
6. `dm.sv`.
7. `tb_alu.sv`.
8. `tb_sgn_ext.sv`.

9. `tb_rf.sv`.
10. `tb_im.sv`.
11. `tb_dm.sv`.
12. A `.do` script for compiling and simulating any of the building blocks listed above.

Submit your assignment through Canvas **no later** than 23:59 hours on Wednesday September 30th 2020.
Please send any questions to isaac.perez.andrade@tec.mx.