CENG 3010
REPORT

Mustafa ÇATALTAŞ
Sevcan DOĞRAMACI

Contents

1. MIPS Simulator

At the beginning of our project, we first developed a simulator program that simulates how a MIPS works to get a deep understanding about how a processor works, how an instruction executed, etc.

We developed the simulator on Java and built our graphical user interface with JavaFX library.

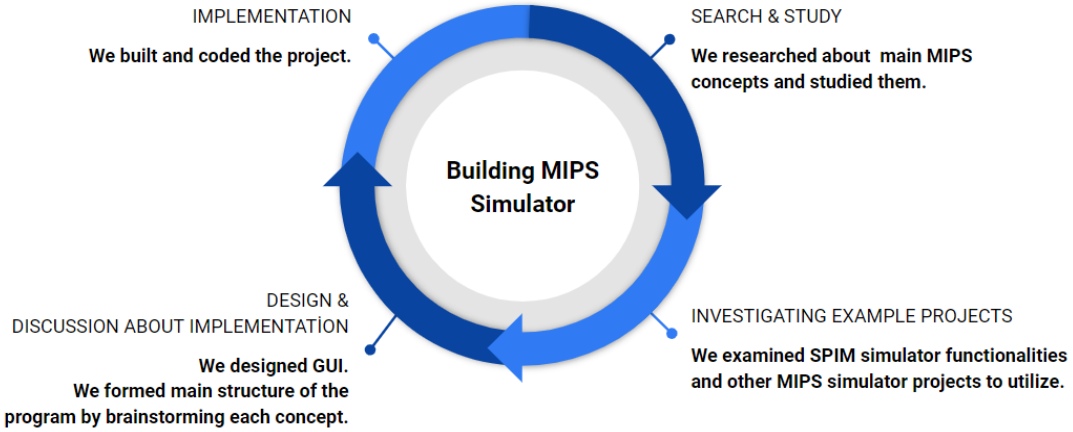

Fig. 1: Phases in Development of MIPS Simulator

Before starting to develop the simulator, we did research about MIPS instruction formats, datapath, memory organization, control signals, alu operation etc. It helped us to develop the simulator and also to have an understanding of what we are dealing with.

Then, we started searching related projects on the Internet to have an idea about what approaches we could take in this simulator. As a result of our research, we decided to develop a simulator that fully follows MIPS Datapath and memory organization.

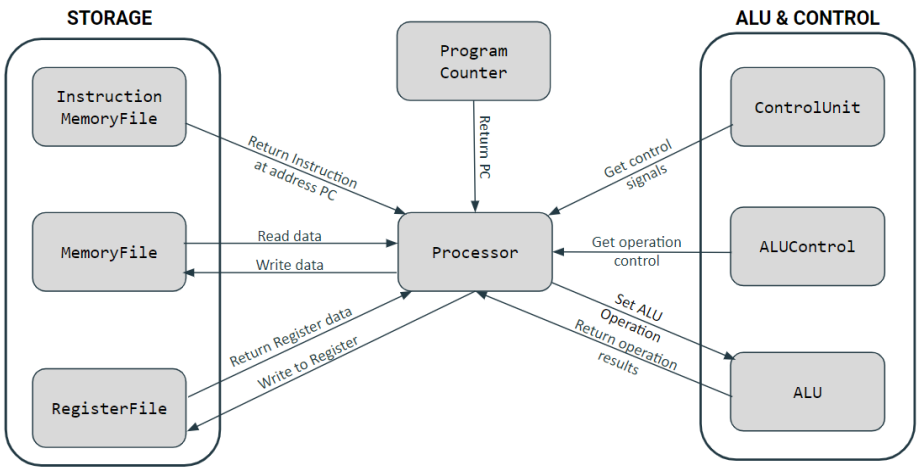

Fig. 2: General flow of the processor

Fig. 2 shows the general flow of our simulator where Processor class acts as a connective class for all other classes like MemoryFile and RegisterFile.

For memory organization, we tried to resemble MIPS' 32-bit aligned memory organization. The only thing we left out in memory is unaligned memory accesses, because in our instruction set it was not necessary to implement it. Fig. 3 shows a representation of our Memory organization.

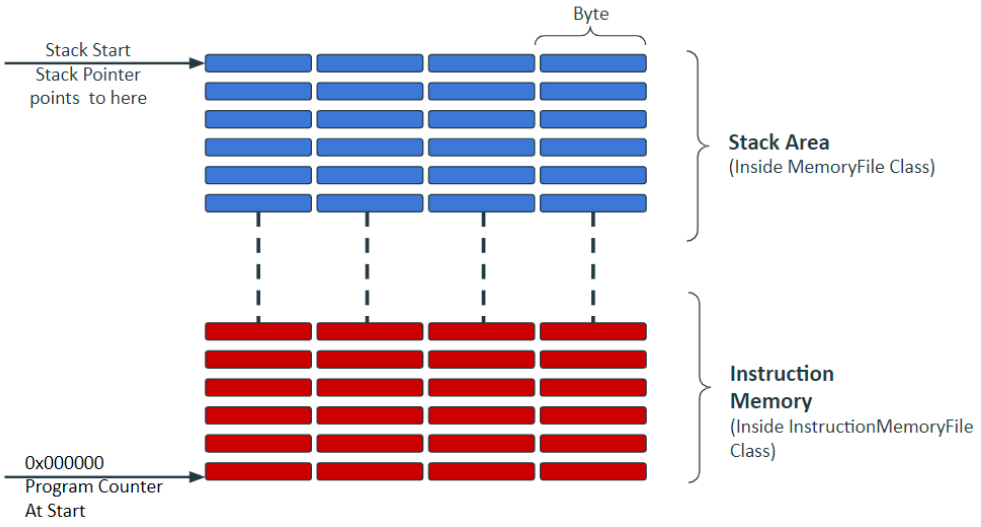

Fig. 3: Memory organization in MIPS Simulator

To test our simulator, we have written several tests that cover all instructions in our instruction set, register reads and writes, memory reads and writes, branches, jumps. At the end, we had a fully working MIPS simulator that covers all instructions in our instruction set without any error.

Besides unaligned memory accesses, we also excluded some other MIPS features too. These are namely coprocessor and coprocessor instructions which are regarding floating point numbers, multiplication and division operations in ALU, syscall and break instruction.

2. Design

In the second phase of the project, our mission was to design our own 16-bit RISC-based processor. We were assigned to execute a set of MIPS instructions in our design.

• add	• slt	• mul	• j
• sub	• slti	• muli	• jal
• and	• sll	• lw	• beq
• or	• srl	• sw	• bne
• jr	• lui		

Fig. 4: The set of MIPS instructions to be implemented in our design

We were expected to perform the instructions stated in Fig.4 with procedures successfully. In addition to this, there were several other limitations that we needed to take into consideration :

1. Making a single-cycle design,
2. Using eight registers each of which was in 16-bit length,
3. Supporting 8 x 8-bit multiplication in the multiplication instructions (mul and muli),
4. Using the lower half of the register with immediate value in muli instruction,
5. Using 256 bytes of instruction memory,
6. Using 256 bytes of data memory,
7. Supporting loops, branches, procedures and procedural calls.

Steps

We followed certain steps to accomplish the task.

a. Searching about Instructions

Before diving into depth, we made a search about the compulsory instructions to see how they are executed with registers in MIPS to remember the background and find out the essential registers.

b. Deciding on Registers

Since we would have eight registers, 3 bits would be required to access all of them. We assigned a purpose for each of them to conceptualize their particular use cases. We considered that we needed a stack pointer to keep track of the location of the top of the stack for procedures.

Also, there was an apparent need for storing the return address to support jumps and procedure calls. Apart from the stack pointer and return address registers, we made the remaining registers as general purpose registers. However, we gave argument and return value purposes for a0 and v0 registers consecutively for enabling a register content assignment practice among programmers.

Table 1: Registers in Register File

Registers		
Register Number	**Name**	**Purpose**
0	r0	general
1	r1	general
2	r2	general
3	r3	general
4	a0	argument
5	v0	return value
6	sp	stack pointer
7	ra	return address

Program counter (PC) register is excluded in this register file and thought separately. Since we would have 256 bytes of instruction memory, instead of using a 16-bit register, we designed the PC as a 8-bit register because we could reach every memory address with this number of bits.

c. Deciding on Instruction Format

As in MIPS architecture, we decided to have instructions with three formats: R, I and J.

Table 2: Instruction formats

INSTRUCTION FORMATS									
R	Field	opcode	is_jump	is_imm	rs	rt	rd	unused	Total
	Bits	3	1	1	3	3	3	2	16
I	Field	opcode	is_jump	is_imm	rs	rt	immediate		Total
	Bits	3	1	1	3	3	5		16
J	Field	opcode	is_jump	is_imm	label				Total
	Bits	3	1	1	11				16

We designed our all instruction types with a number of common fields:

Table 3: Explanation of fields used

Field	Explanation
opcode	3 bits to identify a instruction
is_jump	1 bit to determine that the instruction has jump
is_imm	1 bit to determine that the instruction has immediate
Rd	Destination register address
Rs	Source register address
Rt	Target register address

While deciding on the fields in the instruction formats, we aimed to utilize the 3-bit opcode field as ALU operation code in order to reduce decoding complexity of an instruction. The idea was that we would first look at is_jump and is_imm fields and decode the instruction accordingly. In the decode phase, we tried to choose 3-bit opcodes the same as or closer to ALU operation codes to decrease time and hardware complexity to resolve which operation to select in ALU.

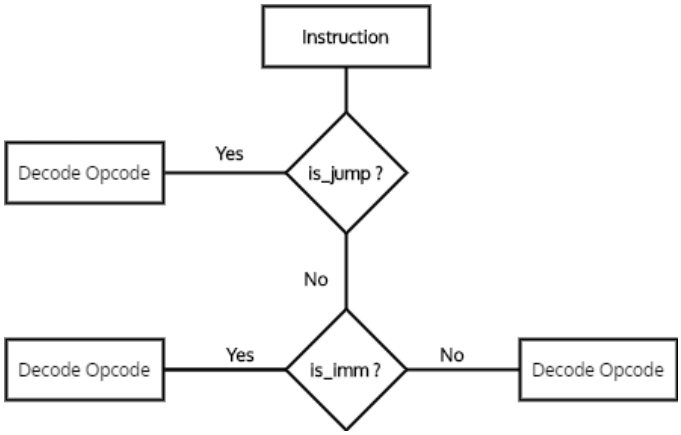

Fig. 5: Decoding logic for control unit

R-Format Instructions :

R-format instructions are the instructions with three register values. Instructions in this format include add, sub, or, and, mul, sll, srl, jr, syscall and slt. We added an instruction called *syscall* which was not mandatory, for enabling programmers to display the content of a register when it was called. Jr and syscall instructions are exceptions that do not utilize target and destination registers, but they are categorized as R-Format due to their use of source registers. The operations performed in each R-format instruction is demonstrated in Table 4.

Table 4: R-Format instructions

R-Format Instructions					
Instruction	**Explanation**	**Format**	**Opcode**	**is_jump**	**is_imm**
add	$d = $s + $t pc = pc + 4	add $d, $s, $t	000	0	0
sub	$d = $s - $t pc = pc + 4	sub $d, $s, $t	001	0	0
mul	$d = ($s[8]) x ($t[8]) pc = pc + 4	mul $d, $s, $t	010	0	0
and	$d = $s & $t pc = pc + 4	and $d, $s, $t	011	0	0
or	$d = $s | $t pc = pc + 4	or $d, $s, $t	100	0	0
sll	$d = $t << $s pc = pc + 4	sll $d, $s, $t	101	0	0
srl	$d = $t >> $s pc = pc + 4	srl $d, $s, $t	110	0	0
slt	if $s <$t $d = 1 else $d = 0 pc = pc + 4	slt $d, $s, $t	010	1	0
jr	pc = $s	jr $s	101	1	0
syscall	display $s	syscall $s	111	0	0

I-Format Instructions :

I-format instructions are the instructions utilizing an immediate value. Instructions in this format include lui, slti, muli, beq, bne, sw and lw. This format instructions does not have the Rd field. Yet, they include a 5-bit immediate field for storing the immediate value. Because they use immediates, their is_imm field is always set to 1. The operations performed in each I-format instruction is demonstrated in Table 5.

Table 5: I-Format instructions

I-Format Instructions					
Instruction	**Explanation**	**Format**	**Opcode**	**is_jump**	**is_imm**
lui	$t = immediate << 8 pc = pc + 4	lui $t immediate	101	0	1
slti	if $s < imm $t = 1 else $t = 0 pc = pc + 4	slti $t, $s, immediate	111	0	1

I-Format Instructions					
Instruction	**Explanation**	**Format**	**Opcode**	**is_jump**	**is_imm**
muli	$t = ($s[8]) x (immediate[8]) pc = pc + 4	muli $t, $s, immediate	100	0	1
beq	if $s == $t pc = (offset << 2) else pc = pc + 4	beq $s, $t, offset	001	0	1
bne	if $s != $t pc = (offset << 2) else pc = pc + 4	bne $s, $t, offset	011	0	1
sw	MEMORY[$s + offset] = $t pc = pc + 4	sw $t, offset($s)	000	0	1
lw	$t = MEMORY[$s + offset] pc = pc + 4	lw $t, offset($s)	010	0	1

J-Format Instructions :

J-format instructions are the instructions targeting a jump. Instructions in this format include jal and j. This format instructions have no Rd, Rs and Rt fields. Instead, they include a 11-bit label field for storing the target address. Because they are intended for jump, their is_jump field is always set to 1. The operations performed in each J-format instruction is demonstrated in Table 6.

Table 6: J-Format instructions

J-Format Instructions					
Instruction	**Explanation**	**Format**	**Opcode**	**is_jump**	**is_imm**
jal	ra = pc + 4 pc = label	jal label	000	1	0
j	pc = label	j label	001	1	0

d. Deciding on ALU Operations

When we considered our instruction set architecture (ISA), we identified arithmetic and logic operations that would be executed. Furthermore, since we had a 3-bit opcode field that we utilized as ALU operation code, we put eight operations which were addition, subtraction, and, or, multiplication, logical shifts (left and right) and set less than to our ALU implementation.

Table 7: ALU operations

ALU				
No	**Operation**	**Operation Control**		
1	add	0	0	0
2	sub	0	0	1
3	and	0	1	0
4	or	0	1	1
5	mul	1	0	0
6	sll	1	0	1
7	srl	1	1	0
8	slt	1	1	1

e. Multiplication in ALU

Since an iterative approach was not feasible for single-cycle datapath, we designed a multiplication unit in ALU, that was composed of eight shifters and eight 8-bit adders. We utilized the add and shift algorithm which is demonstrated in Fig. 6. in this unit.

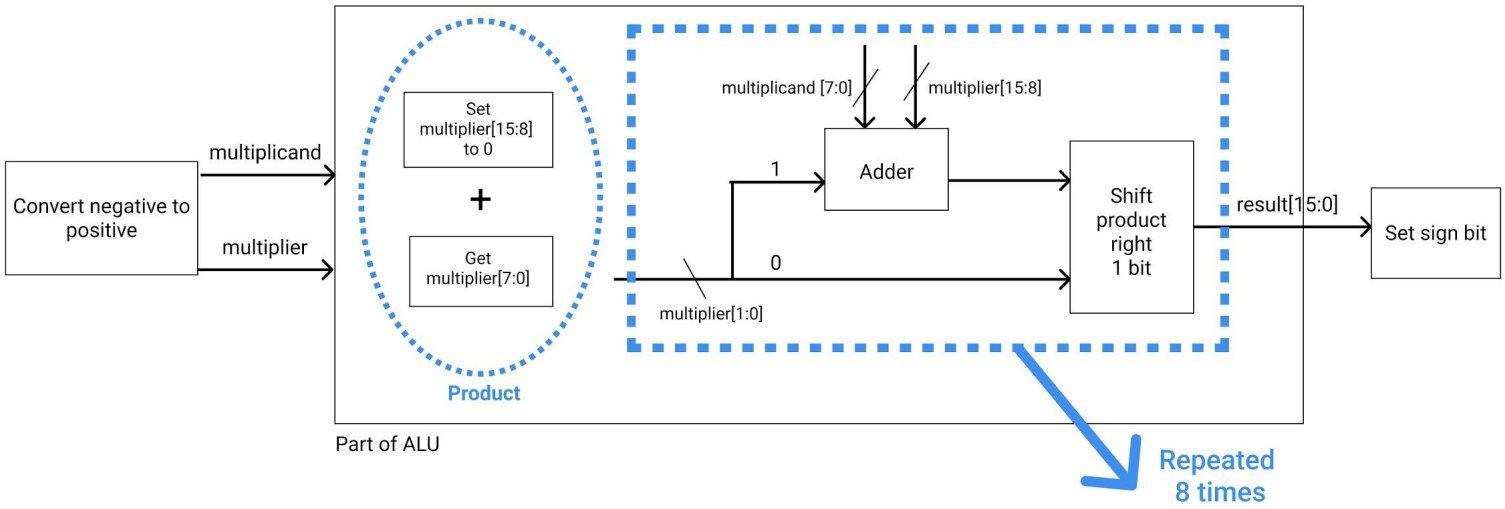

Fig. 6: Multiplication implementation in ALU

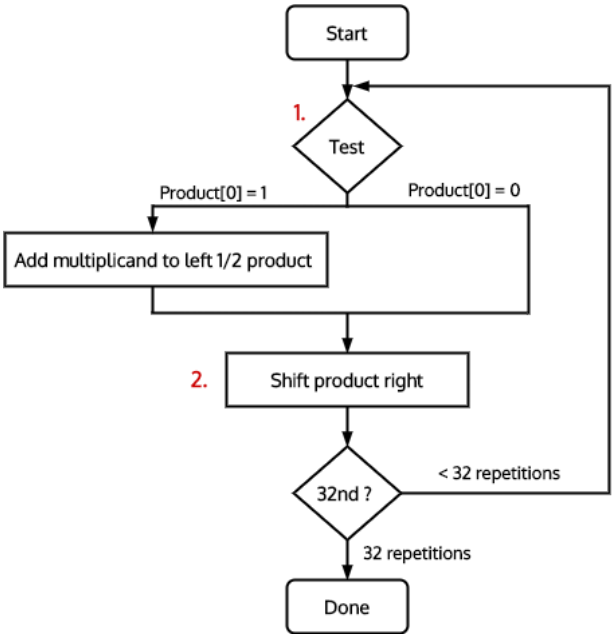

Fig. 7: Multiplication algorithm

f. Deciding on Control Signals

As in MIPS architecture, there is a need for control signals in a processor in order to decode an instruction. They are used in the hardware datapath to direct the signals and make proper data selections. We used 11 control signals in total to achieve a full-working datapath. They are specified in Table X. with their use cases.

Table 8: Control signals

Control Signal	Explanation
RegDst	to select between $rt and $rd to branch on not equal
ALUSrc	to select between immediate and register data 2
RegWrite	to enable writing into register
MemRead	to enable reading from memory
MemWrite	to enable writing into memory
Branch	to go to branch address
ALUOp	to select ALU operation - 3 bits
Jump	to go to jump address
JumpReg	to select register data 1 for jr instruction
ShiftReg	to select shift amount between register data 1 and 8
SysCall	to select register data 1 for LCD display

Table 9: Control signals for each instruction

Format	Instruction	RegDst	ALUSrc	RegWrite	MemRead	MemWrite	Branch	ALUOp2	ALUOp1	ALUOp0	Jump	JumpReg	ShiftReg	Syscall
R	add	1	0	1	0	0	0	0	0	0	0	0	0	0
	sub	1	0	1	0	0	0	0	0	1	0	0	0	0
	mul	1	0	1	0	0	0	1	0	0	0	0	0	0
	and	1	0	1	0	0	0	0	1	0	0	0	0	0
	or	1	0	1	0	0	0	0	1	1	0	0	0	0
	sll	1	0	1	0	0	0	1	0	1	0	0	0	0
	srl	1	0	1	0	0	0	1	1	0	0	0	0	0
	jr	1	0	1	0	0	0	x	x	x	x	1	1	0
	syscall	0	0	0	0	0	0	x	x	x	x	0	0	1
	slt	1	0	1	0	0	0	1	0	0	0	0	0	0
I	lui	0	1	1	0	0	0	1	0	1	0	0	1	0
	slti	0	1	1	0	0	0	1	0	0	0	0	0	0
	muli	0	1	1	0	0	0	1	0	0	0	0	0	0
	beq	0	0	0	0	0	1	0	0	1	0	0	0	0
	bne	1	0	0	0	0	1	0	0	1	0	0	0	0
	sw	0	1	0	0	1	0	0	0	0	0	0	0	0
	lw	0	1	1	1	0	0	0	0	0	0	0	0	0
J	jal	0	0	1	0	0	0	x	x	x	1	0	0	0
	j	0	0	0	0	0	0	x	x	x	1	0	0	0

g. Choosing Basic Outputs

In order to monitor the workflow in our processor design, we decided to use led diodes for showing active control signals, memory address value, immediate value as well as PC value. In addition to this, we preferred to use a 2x16 LCD display to show opcode, is_jump and is_imm fields' values when syscall is not called and to show the register content specified in the syscall instruction when it is called. Due to LCD display configurations and communication issues, we would benefit from an Arduino UNO to make the process easier.

h. Datapath

Due to our opcode-ALU operation code relationship, we eliminated ALU Control unit in MIPS. Our resulting datapath is as follows in Fig. 8.

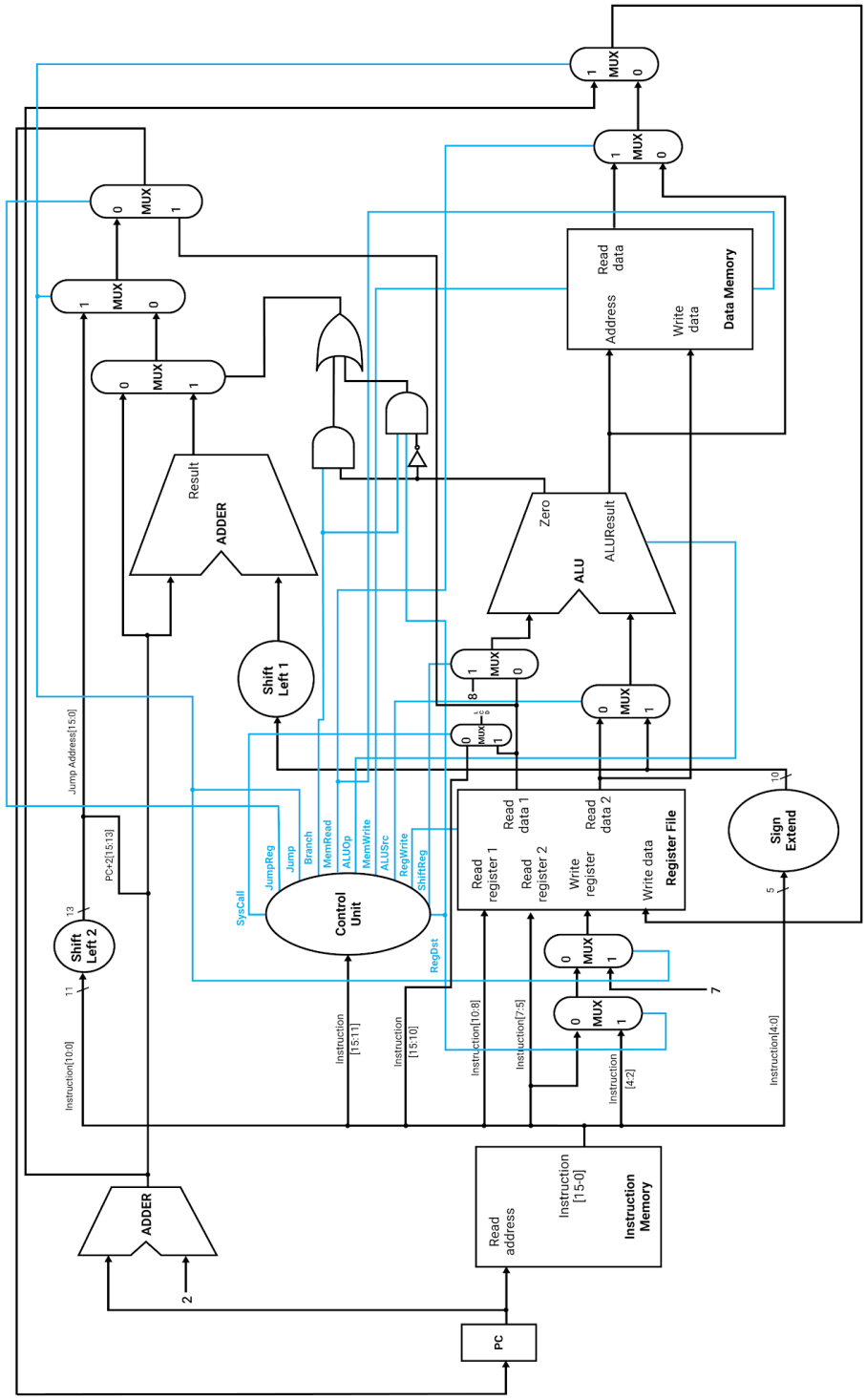

Fig. 8: Datapath

3. Design's Simulator

After designing our 16-bit processor with our instruction set, we modified our MIPS simulator, which we developed in the first phase of the project, to implement our own 16-bit processor design. Thanks to this, we were able to see if there were any flaws in the design and made sure our processor design is going to work.

Our simulator simulates execution of instructions. While simulating, it shows current register values, data and instruction memory, program counter, LCD display value along with control signals of current instruction.

As can be seen in Fig. 9, we take an aligned approach in data memory where only aligned 16-bit word accesses are allowed.

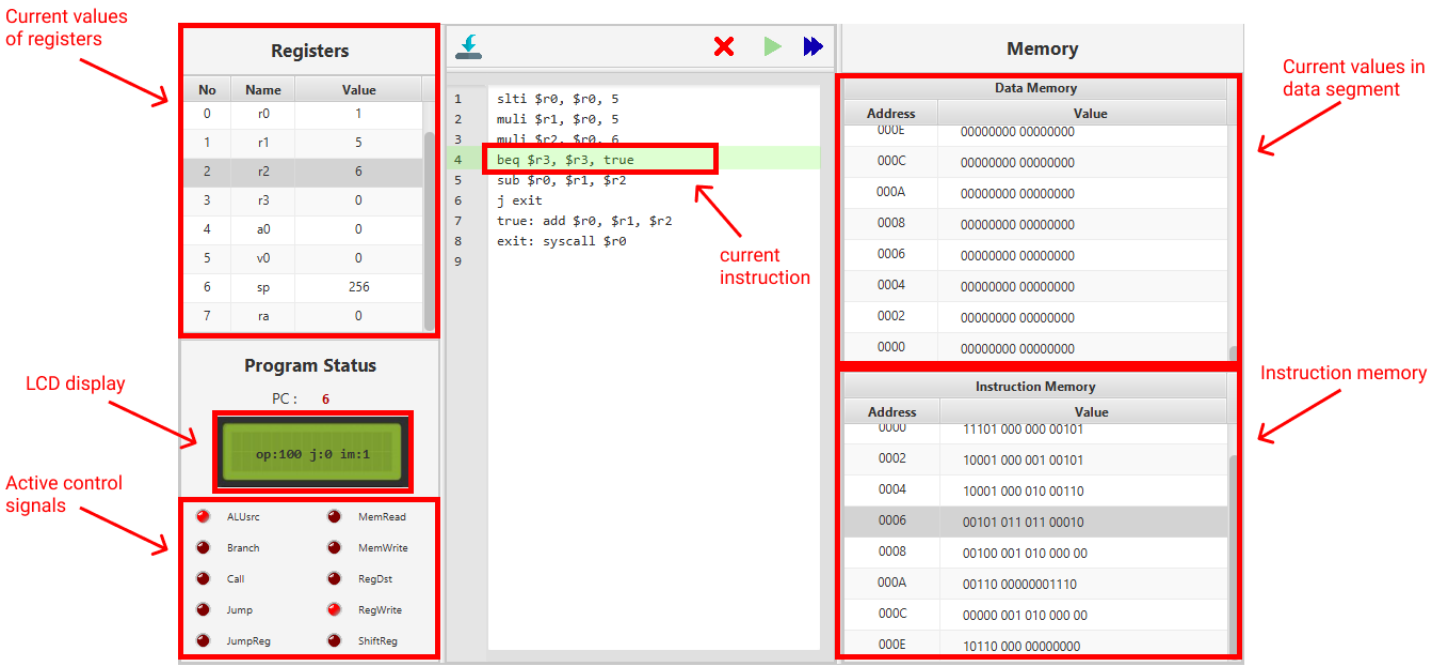

Fig. 9: GUI of processor simulator

4. Verilog Simulation

After simulating our design in a Java application, the next mission was writing a simulation of our own design using a hardware description language. The goal of this task was simulating the design we made at hardware level to verify the design model. We programmed our simulation in Verilog. We selected Xilinx ISE Design Suite as a development environment.

We created the following modules in Verilog for making abstractions.

- LCD
- ALU
- register_file
- instruction_memory
- data_memory
- control_unit
- processor

We created a module for each processor unit. Then, we put them together in a separate processor module. We coded a testbench program called processor_tb to test the processor design with certain test codes generated in the Java simulator in the Section 3.

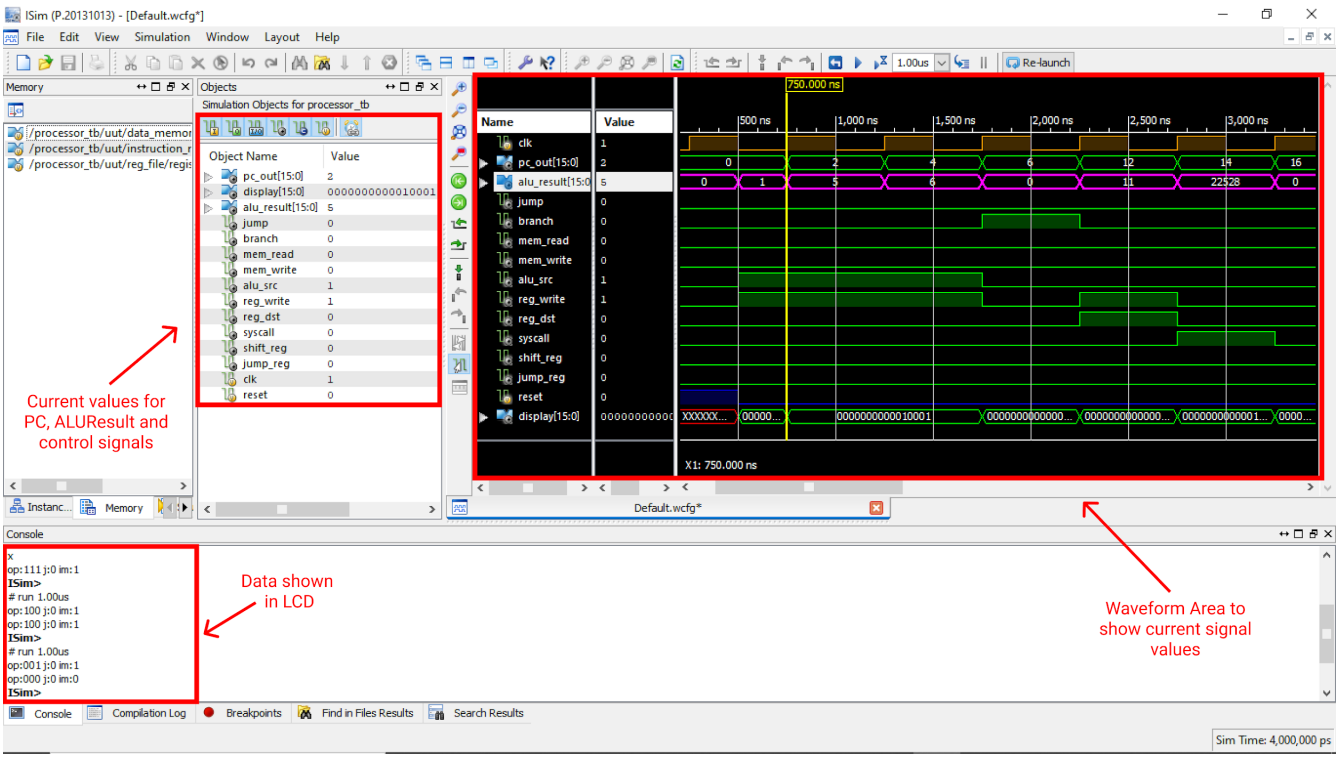

Fig. 10: Verilog simulation