CSE 306 : Offline 3

8-bit MIPS Design and Simulation

June 20, 2021

Lab Section : B2
Group 03

1705093
1705098
1705103
1705110
1705119

1 Introduction

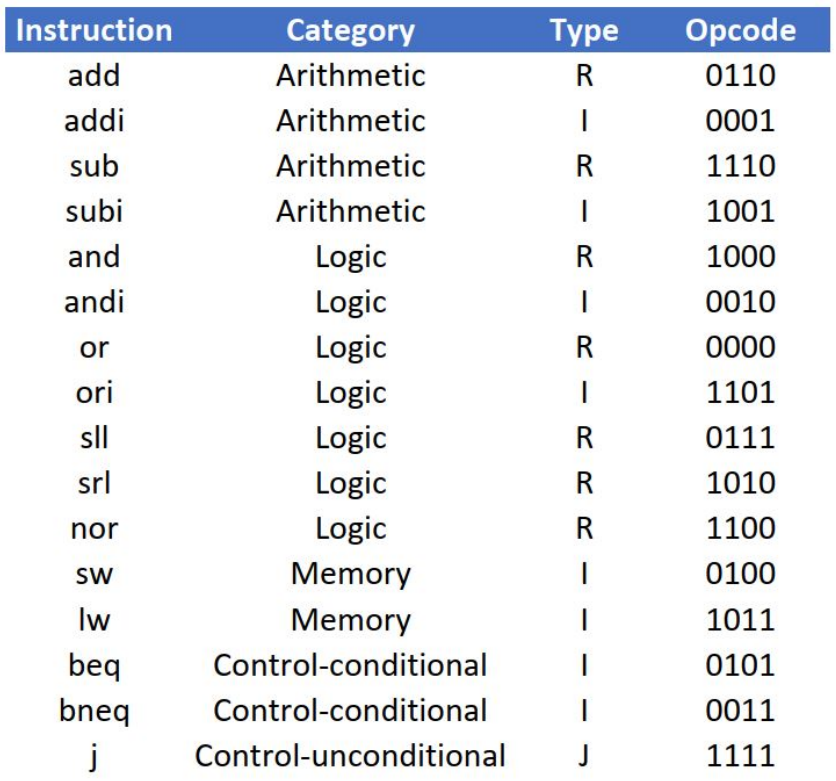
In this assignment we implemented an 8-bit Single-Cycle processor for a subset of the MIPS instruction set. The implementation is called 8-bit because both the data bus and the address bus are 8-bit wide. And it is single cycle because instructions are not overlapped in execution.

Definition of some key terms:

- **Instruction Set Architecture:** An abstract view of the processor as seen by the assembly programming language. It defines the instructions that can be directly carried out by the processor.
- **MIPS (Microprocessor without Interlocked Pipelined Stages):** A reduced instruction set computer (RISC) instruction set architecture.
- **Reduced instruction set computer, or RISC:** A computer with a small, highly optimized set of instructions.

2 Instruction Set

The table below summaries the Instruction Set that we used along with the binary values of their respective Opcodes



Instruction	Category	Type	Opcode
add	Arithmetic	R	0110
addi	Arithmetic	I	0001
sub	Arithmetic	R	1110
subi	Arithmetic	I	1001
and	Logic	R	1000
andi	Logic	I	0010
or	Logic	R	0000
ori	Logic	I	1101
sll	Logic	R	0111
srl	Logic	R	1010
nor	Logic	R	1100
sw	Memory	I	0100
lw	Memory	I	1011
beq	Control-conditional	I	0101
bneq	Control-conditional	I	0011
j	Control-unconditional	J	1111

Figure 1: Instruction Set

2.1 Instruction Format

All our instructions were 20-bits long with the following three formats.

- R-type

Opcode	Src Reg 1	Src Reg 2	Dst Reg	Shft Amnt
4-bits	4-bits	4-bits	4-bits	4-bits

- I-type

Opcode	Src Reg	Dst Reg	Address / Immediate
4-bits	4-bits	4-bits	8-bits

- J-type

Opcode	Target Jump Address	0	0
4-bits	8-bits	4-bits	4-bits

Figure 2: Instruction Format

3 Block diagram of 8-bit MIPS processor

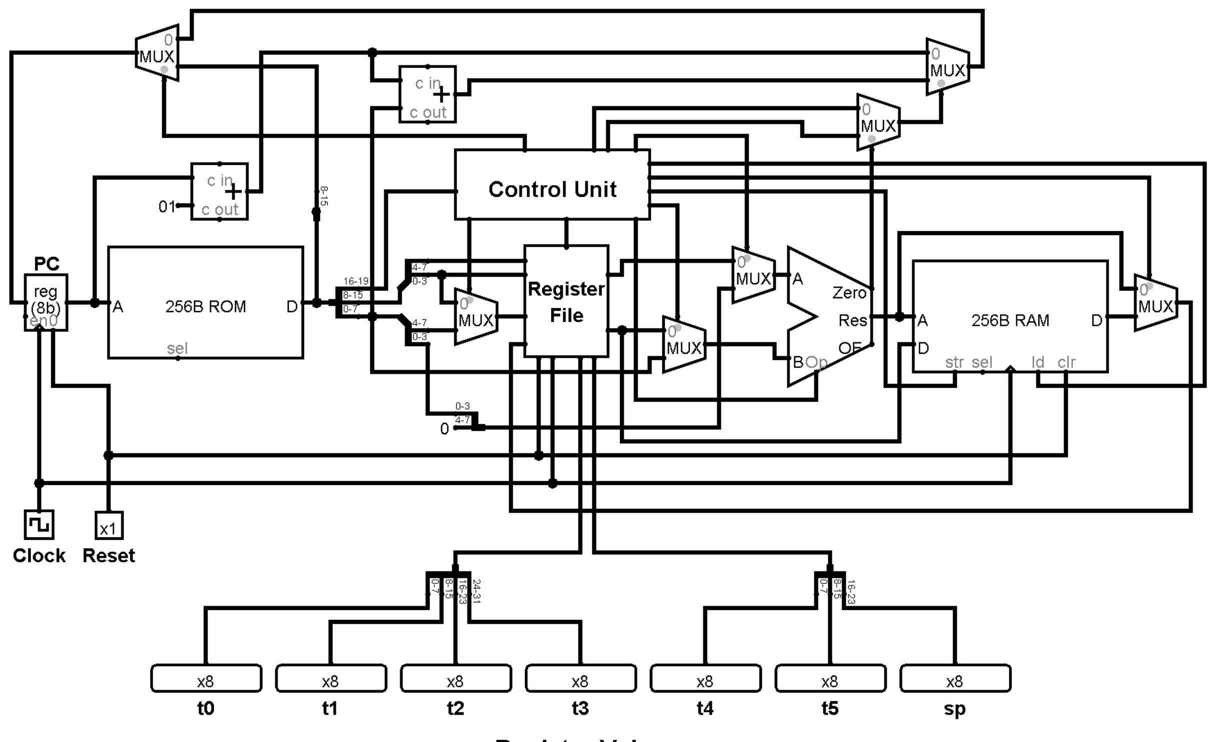


Figure 3: 8-bit MIPS processor

- A 20 bit instruction read from the Instruction Memory is divided into specific chunks as required using a splitter. The OpCode is sent to the control unit. The Register File receives the Read/Write register number.
- Depending on the instruction type, the correct Write number for the is sent to the Register File using a MUX.

- Input A of ALU is Read Register 1 or the Shift Amount. This is controlled by a MUX.
- Input B of ALU is Read Register 2 or an immediate value from the 20 bit instruction. This is controlled by a MUX.
- The Write Data input to the register file is either from the Data Memory or from the ALU output. This is controlled by MUX.
- PC value is incremented by 1 and an offset is added to it if conditional branching is required.
- For unconditional branching the exact value of the jump address is provided by the 20 bit instruction.
- The register values at the bottom of the diagram above are not essential to the operation of the circuit. They are added for convenience - to see how the values of registers change as an instruction is executed.

4 Main Components

4.1 Instruction memory with PC

- The PC register holds the address of the instruction to be executed in the next clock pulse.
- The 256B ROM is used as the Instruction Memory

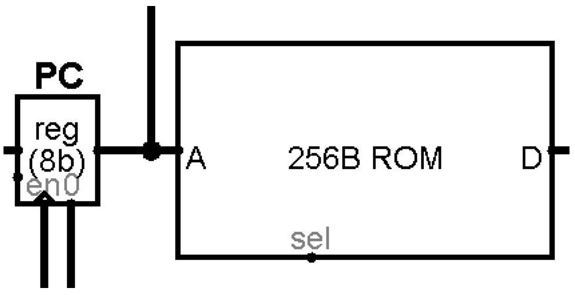


Figure 4: Instruction Memory

4.2 Register file

- The decoder selects the register to which data will be written using the Write Data input in the next clock pulse.
- The value of the $zero register is never changed from 0
- Two 8 to 1 MUXes are used to select the output for the two Read Data outputs. These MUXes are 8bit wide.

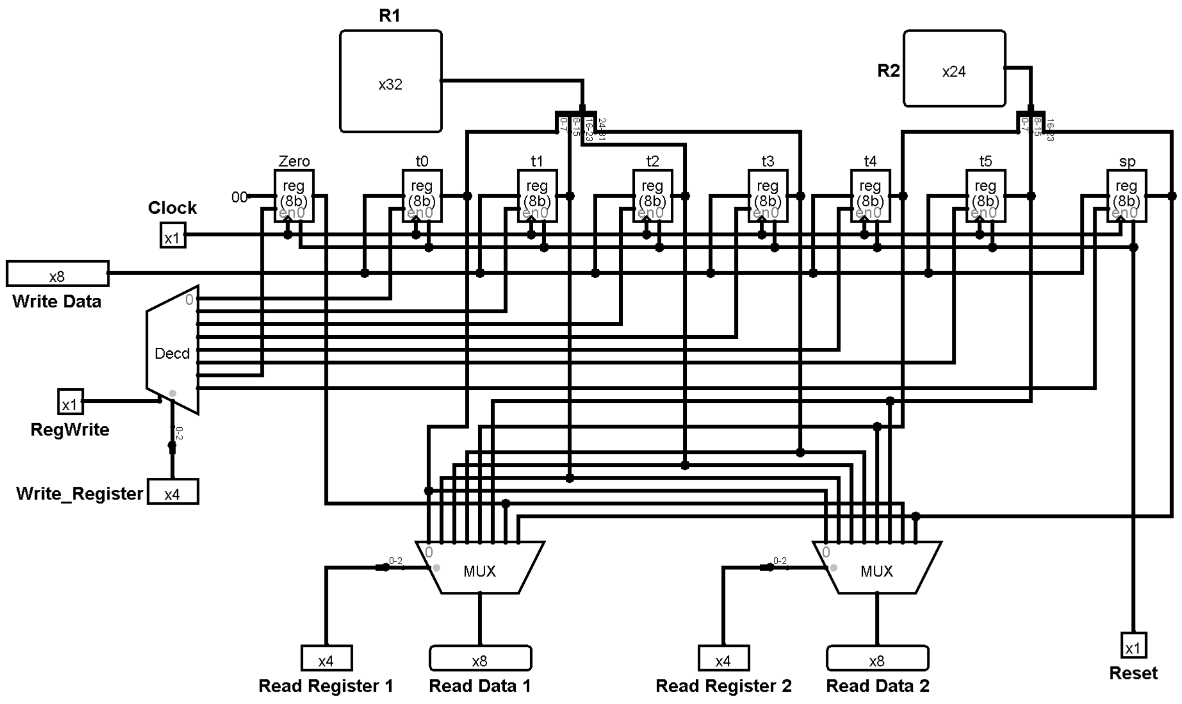


Figure 5: Register File

- The R1 and R2 outputs are not significant for the operation of the circuit. They are used for convenience - to allow us to see the values of the register from outside the Register File.

4.3 Data memory with the Stack

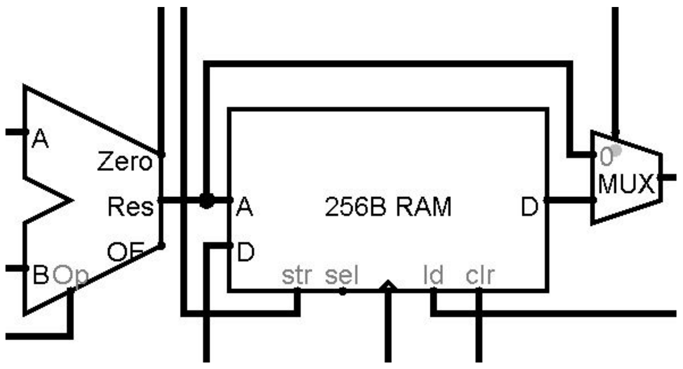


Figure 6: Data Memory

- The output of the ALU is fed into the Data Memory as the address.
- Two outputs from the Control Unit are used to enable Read or Write on the Data Memory
- The data to be written comes from the register file

- The output of the Data Memory is passed to the Register File through a MUX which uses receives its selection input from the Control Unit
- To handle the stack, we have a stack pointer register in the register file which is set to 0xFF by the first instruction in the instruction memory

4.4 Control unit

- The control unit produces 11 signals based on the OpCode input. 10 of these control signals are 1 bit wide but the ALUOp is 3 bit wide. So in total, 13 bits are needed to produce the output given an OpCode.
- For implementation we used a ROM with 4-bit addressing and 13-bit wide data unit

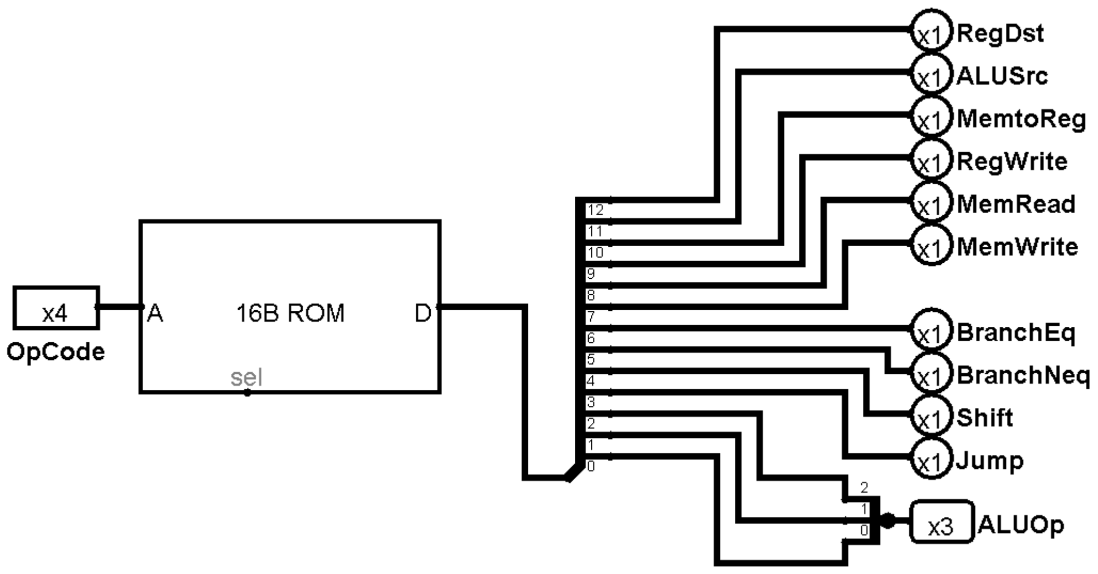


Figure 7: Control Unit

- The table below shows the values of the control signals generated by the Control Unit for each OpCode input.
- The values of the ALUOp are generated depend on the ALU being used. We used the ALU provided in the news forum of CSE 306 course in Moodle.

OpCode	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	BranchEq	BranchNeq	Shift	Jump	ALUOp
0000	1	0	0	1	0	0	0	0	0	0	010
0001	0	1	0	1	0	0	0	0	0	0	000
0010	0	1	0	1	0	0	0	0	0	0	011
0011	0	0	0	0	0	0	0	1	0	0	001
0100	0	1	0	0	0	1	0	0	0	0	000
0101	0	0	0	0	0	0	1	0	0	0	001
0110	1	0	0	1	0	0	0	0	0	0	000
0111	1	0	0	1	0	0	0	0	1	0	101
1000	1	0	0	1	0	0	0	0	0	0	011
1001	0	1	0	1	0	0	0	0	0	0	001
1010	1	0	0	1	0	0	0	0	1	0	110
1011	0	1	1	1	1	0	0	0	0	0	000
1100	1	0	0	1	0	0	0	0	0	0	100
1101	0	1	0	1	0	0	0	0	0	0	010
1110	1	0	0	1	0	0	0	0	0	0	001
1111	0	0	0	0	0	0	0	0	0	1	000

Figure 8: Control Unit Data

- The table below gives the operation performed by the ALU for each ALU Opcode input.

ALU Opcode	Operation
000	A+B
001	A-B
010	A or B
011	A and B
100	A nor B
101	B << A
110	B >>> A
111	B >> A

Figure 9: ALU Opcode

5 Implementing Push and Pop Instructions

- Push and Pop are not in the instruction set. However, they are part of the assembly language used. So to handle Push and Pop the assembler replaces Push and Pop with their implementation using instructions from the instruction set only.

- In our implementation stack pointer $sp always points to the top of the stack - the first **unoccupied** position in the stack in memory.

5.1 Push

push $t3 is implemented as follows:

- sw $t3, 0($sp)
- subi $sp, $sp,1

push 7($t3) is implemented as follows:

- lw $t5, 7($t3)
- sw $t5, 0($sp)
- subi $sp,$sp,1

5.2 Pop

pop $t3 is implemented as follows:

- addi $sp,$sp,1
- lw $t3,0($sp)

6 IC Count

Component	**IC Number**	**Count**
Quad 2:1 MUX	IC 74157	12
4 bit Adder	IC 7483	4
8 bit Register	IC 74ACT825	9
8:1 MUX	IC 74151	16
3:8 Decoder	IC 74238	1
ALU	-	1**
ROM (256×8)	74x271	3**
ROM (32×8)	74x88	2**
RAM (256×4)	74x921	2**

*** → Please See Discussion

7 Simulation Platform

Logisim-2.7.10

8 Discussion

- No IC number WAS mentioned for ALU because we used the ALU provided by course teacher
- Our requirement was a 256×20 ROM. However, we could not find any IC which satisfied this requirement directly. But it can be implemented three 256×8 ROMs.
- Our requirement was a 16×13 ROM. However, we could not find any IC which satisfied this requirement directly. But it can be implemented two 32×8 ROMs.
- Our requirement was a 256×8 RAM. However, we could not find any IC which satisfied this requirement directly. But it can be implemented two 256×4 RAMs.