BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Course No: CSE306
Course Name: Computer Architecture Sessional

Name of the Experiment:
Assignment on 8-bit MIPS Design and Simulation

Level/Term: 3-1
Section: B2

Group No: 02
Student ID: 1705092
1705111
1705116
1705118
1705121

Date of Submission: 20-06-2021

Introduction

In this assignment, we have designed a single clock cycle 8-bit processor that implements the MIPS instruction set.

The processor has an 8-bit ALU, hence it is an 8-bit MIPS. Each instruction takes 1 clock cycle to be executed. So, this is a single clock cycle processor.

The main components of the processor are as follows: instruction memory, data memory, register file, ALU, and a control unit.

Instruction Set

The ALU has 7 operations. Operation is decided by ALUOp.

ALUOp	Operation
000	ADD
001	SUB
010	AND
011	OR
100	NOR
101	Shift Logic Left
110	Shift Logic Right
111	x

The second input of ALU is decided by ALUSrc.

ALUSrc	Source
00	Read data 2
01	Shift amount
10	Immediate
11	x

The 16-bit control sequence,

-			Reg Write	ALU Src		Mem Write	ALUOp			Memto Reg	Mem Read	Branc hNeq	Branc hEq	Jump	Reg Dst

MSB

For group 2 in section B2, the instruction set sequence:

AKEHPDFOBCNLGMJI

The Instruction Set,

Instruction ID	Instruction	Type	Opcode	Control Sequence
A	add	R	0x0	0b 0001 0000 0000 0001 = 0x1001
K	nor	R	0x1	0b 0001 0001 0000 0001 = 0x1101
E	and	R	0x2	0b 0001 0000 1000 0001 = 0x1081
H	ori	I	0x3	0b 0001 1000 1100 0000 = 0x18C0
P	j	J	0x4	0b 0000 0000 0000 0010 = 0x0002
D	subi	I	0x5	0b 0001 1000 0100 0000 = 0x1840
F	andi	I	0x6	0b 0001 1000 1000 0000 = 0x1880
O	bneq	I	0x7	0b 0000 0000 0100 1000 = 0x0048
B	addi	I	0x8	0b 0001 1000 0000 0000 = 0x1800
C	sub	R	0x9	0b 0001 0000 0100 0001 = 0x1041
N	beq	I	0xA	0b 0000 0000 0100 0100 = 0x0044
L	sw	I	0xB	0b 0000 1010 0000 0000 = 0x0A00
G	or	R	0xC	0b 0001 0000 1100 0001 = 0x10C1
M	lw	I	0xD	0b 0001 1000 0011 0000 = 0x1830
J	srl	R	0xE	0b 0001 0101 1000 0001 = 0x1581
I	sll	R	0xF	0b 0001 0101 0100 0001 = 0x1541

Every instruction is 20-bit long with following 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

The 20-bit instructions are generated from the assembly language with an assembler written in C++ language in the file “***assembler.cpp***”.

Complete Block diagram of the 8-bit MIPS processor

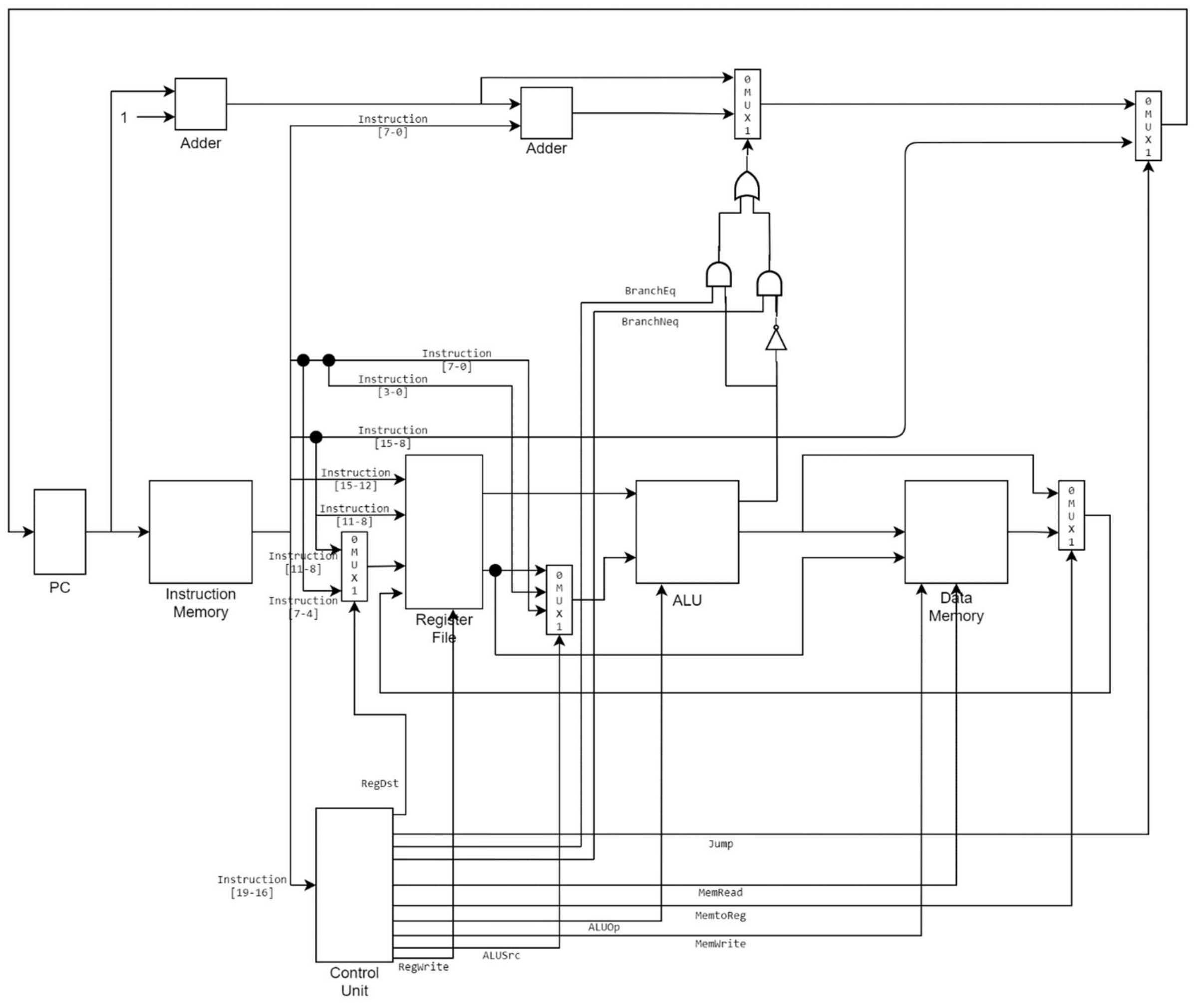


Figure: Complete Block diagram of an 8-bit MIPS processor

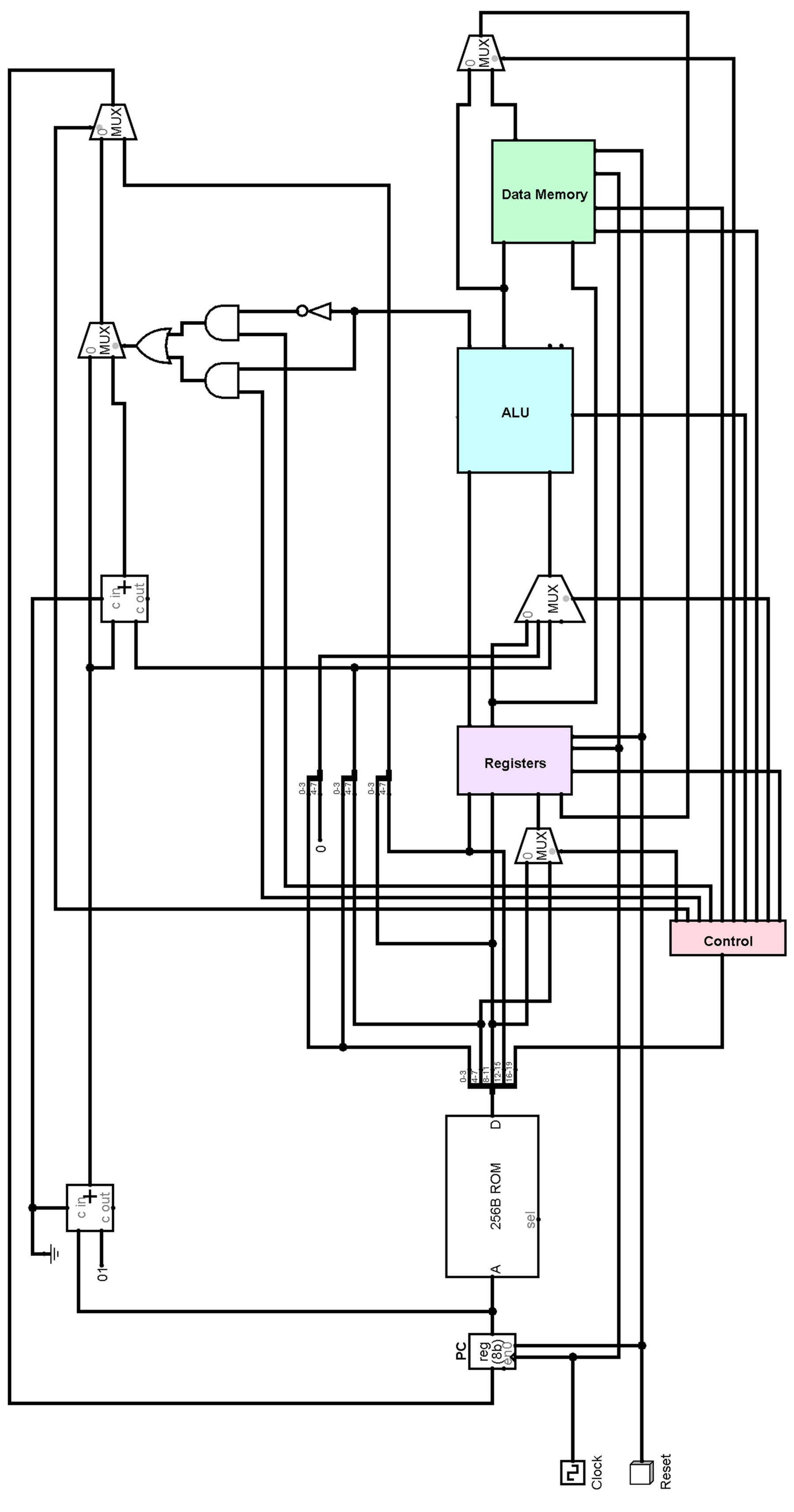


Figure: Circuit diagram of an 8-bit MIPS processor

The PC holds the address of the instruction memory. When an instruction is read, the most significant 4-bits [19-16] are opcode. The input to the control unit is the opcode. The control unit controls the multiplexers and enables of different registers of the processor.

The next 4-bits of the instruction [15-12] is the first input of the register file. The second input of the register file is instruction [11-8]. The third input, write register, is the result of a multiplexer which chooses between instruction [11-8] and instruction [7-4].

The first data read from register file goes to the first input of ALU. The second data read goes to a multiplexer with instruction [3-0] (padded with 0 to make 8-bit) and instruction [7-0]. The output of this multiplexer goes to the second input of the ALU. Also, the second data read from register goes to the write data input of the data memory unit.

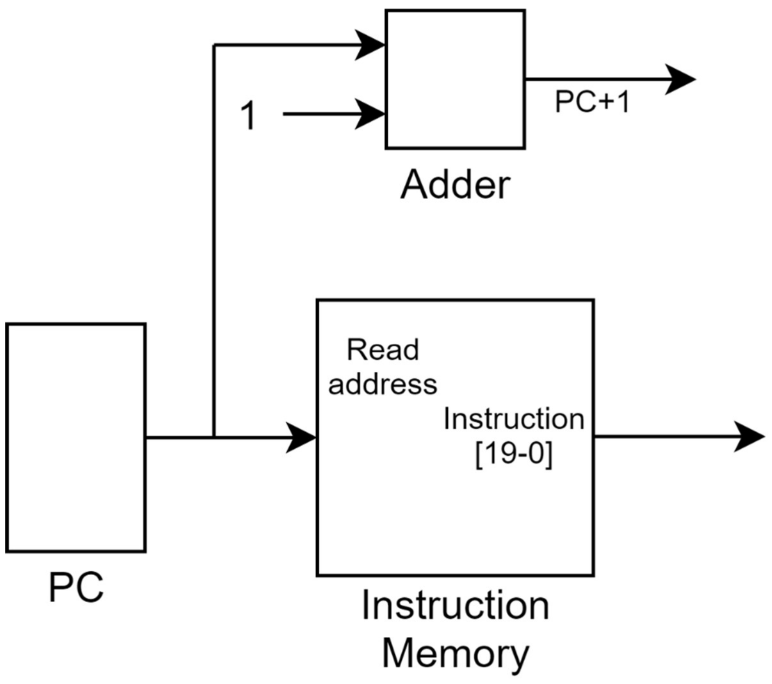
The output from ALU is the input data of the data memory unit and it also goes to a multiplexer to choose between ALU output and data read from the data memory unit. The zero flag output along with the control unit flags create the selection bits of the conditional jump.

Conditional jump, branching, is PC relative. The address is found from adding instruction [7-0] and PC+1.

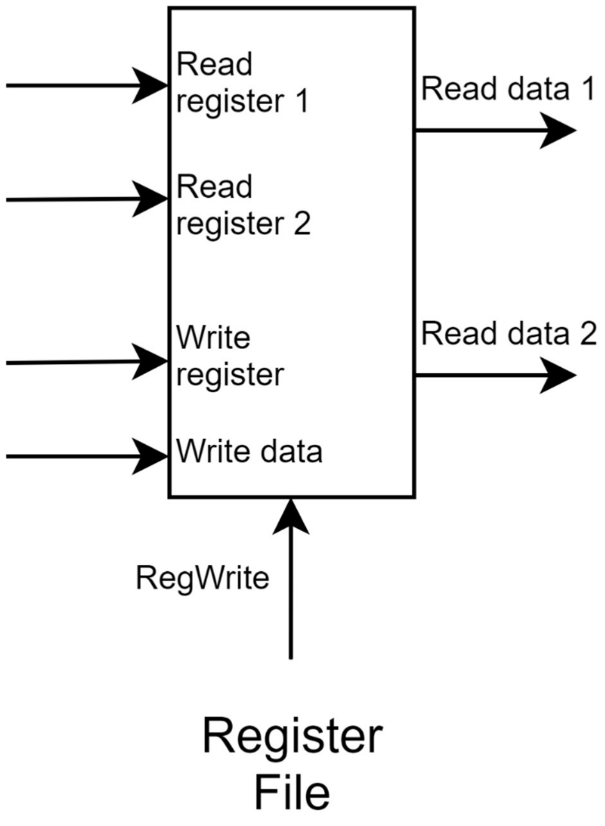
Jump address is instruction [15-8]. It goes into another multiplexer to choose from conditional jump (previously chosen with a multiplexer) or unconditional jump.

This address then goes into the input of the PC for the next instruction address.

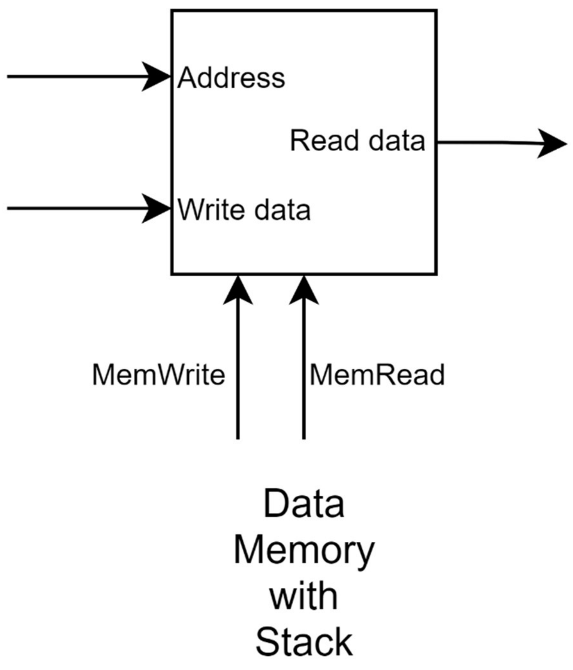
Block diagram of Instruction memory with PC



Block diagram of Register file

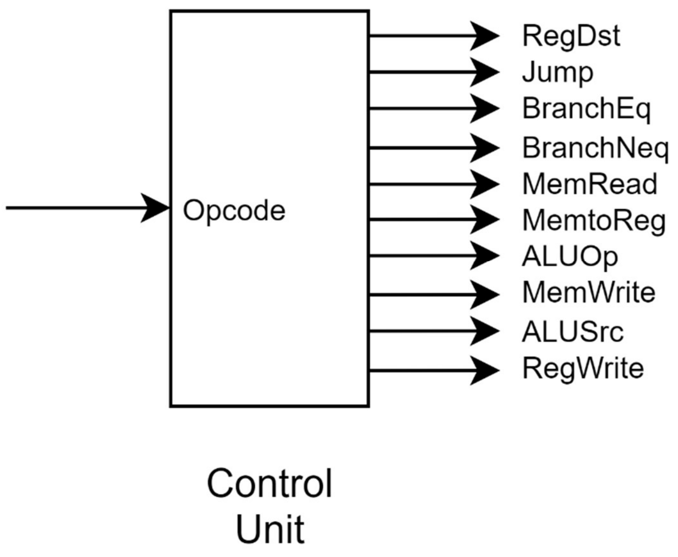


Block diagram of Data memory with the Stack



The data memory and stack memory are in a single memory unit. The stack top address is stored in a register (stack pointer) starting from the highest memory address of the memory unit and stack memory grows to the decreasing memory address.

Control Unit



Approach to implement the push and pop instructions

The push and pop instruction to be used in our assembly language is according to the following format,

Instruction	Description
push $t0	mem[**$sp**] = **$t0**
push 3($t0)	mem[**$sp**] = mem[**$t0**+3]
pop $t0	**$t0** = mem[**$sp**]

The assembler converts the push or pop operation in necessary MIPS instruction set.

Push
When a push is called, top address of stack (stored in stack pointer register, **$sp**) is decreased and the data is stored in the stack memory to that address.

So, for push operation in the format, **push $t0**, the address stored in **$sp** is decremented by 1 and the data stored in **$t0** is stored in the memory address from **$sp**.

```
push $t0
```

Generated intermediate code:

```
subi $sp, $sp, 1
sw $t0, 0($sp)
```

For push operation in the format, **push 3($t0)**, the data stored in **mem**[**$t0**+3] is first loaded in a register and then the data is stored in stack. As we did not want to use another register solely for this one operation, we used the **$t0** register to temporarily load the memory in it. To preserve the data currently stored in **$t0**, here one extra push operation is implemented to store the **$t0** in the stack and then after the main push operation it is retrieved from stack again.

```
push 3($t0)
```

Generated intermediate code:

```
subi $sp, $sp, 2
sw $t0, 0($sp)
lw $t0, 3($t0)
sw $t0, 1($sp)
lw $t0, 0($sp)
addi $sp, $sp, 1
```

Pop

When a pop is called, the data stored in the top of the stack memory is loaded in a register and the top address of stack (stored in stack pointer register, **$sp**) is increased.

So, for pop operation in the format, **pop $t0**, the data stored in the top of the stack, pointed by the address in **$sp**, is loaded into **$t0.** And the address in **$sp** is incremented by 1.

```
pop $t0
```

Generated intermediate code:

```
lw $t0, 0($sp)
addi $sp, $sp, 1
```

ICs used with their count

Component	IC Number	Count of ICs
4-bit Full Adder	7483	4
AND Gate	7408	1
OR Gate	7432	1
NOT Gate	7404	1
2x1 MUX	74157	7
4x1 MUX	74153	4
16x1 MUX	744067	16
4x16 Decoder	744514	8
8-bit ALU		1
ROM		2
RAM		1

Simulator used

The simulation software used in this 8-bit MIPS Design and Simulation experiment is ***logisim-win-2.7.1.exe.***

Discussion

- An assembler is written in C++ language to convert the given assembly code into 20-bit instructions. The assembler generates an intermediate assembly code to ignore unnecessary spacing in the beginning of a line and to convert the push-pop instructions in MIPS instruction set.
- The assembler is made according to the sample assembly codes that were provided and may not work in a different code format.
- The 20-bit instructions generated from the assembler are written in a format compatible to load into Logisim simulator.
- The control unit of our processor is micro-programmed. The control sequences for opcodes are already loaded into the control unit memory.
- This is a single clock cycle processor. All the clock signals necessary are provided from a single clock source. The registers are configured to be enabled in rising and falling edge as necessary to execute a full operation in one clock.
- The register file is designed to support up to 16 registers. 7 registers are used currently for the given assembly code instructions.
- The zero register is given a zero input data in case the code tries to write into zero register it will not throw an error but rewrite zero in it.
- For the IC counts, some of the used ICs (MUX, Decoder, Adder) are 8-bit wide whereas the documented ICs are 4-bit or 1-bit wide. IC counts are calculated accordingly so that in real life implementation these ICs can be used as banks to support 8-bit data.
- The ICs used in the ALU are not counted as the ALU is counted as a built-in module.