Project Report

Shivam Singhal (200939) **Shrey Wadhawan (200948)**

Abstract

This report contains the documentation of the **CS220 (Computer Organization) Course Project**. The objective is to implement a single-cycle processor called **CSE-BUBBLE**. The following subsections detail the week-wise process of implementing the processor.

1 PDS1 - Registers and Usage Protocol

Our architecture uses **32** registers with each register being **32** bits wide and closely resembles the actual MIPS architecture. The names, register numbers, and usage protocol is given below.

Register Number	Register names	Usage
0	`$zero`	Stores and always returns 0
1-3	`$v0-$v2`	Return Values
4-7	`$a0-$a3`	Arguments for functions
8-15	`$t0-$t7`	Temporary Values
16-23	`$s0-$s7`	Saved Variables
24-30	`$t8-$t14`	More Temporary Values
31	`$ra`	Return Address

2 PDS2 - Size of Instruction and Data Memory

The memory element of our processor is called **VEDA** containing components, **Instruction Memory** and **Data Memory**. We use byte addressing in instruction memory, and each row of the instruction memory stores **8** bits (**1** byte). Each instruction is **32** bits (4 bytes) long which means an instruction occupies **4** consecutive rows in the Instruction Memory. Both instruction memory and data memory can constitute a variable number of rows. For our purposes of implementing a small program, we used only **72** rows.

Memory Name	Size of a row	Number of Rows
Instruction memory	8 bits	72
Data memory	8 bits	1024

3 PDS3 - Instruction Layout and Encoding Schema

3.1 R-type Instruction

The general layout for an R-type instruction is as follows:

Opcode	rs	rt	rd	Shamt	Funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

The encoding schemes for each R type instruction is as follows:

Instruction	Opcode	rs	rt	rd	Shamt	Funct
`add r0, r1, r2`	0	r1	r2	r0	0	0
`sub r0, r1, r2`	0	r1	r2	r0	0	1
`addu r0, r1, r2`	0	r1	r2	r0	0	2
`subu r0, r1, r2`	0	r1	r2	r0	0	3
`and r0, r1, r2`	0	r1	r2	r0	0	4
`or r0, r1, r2`	0	r1	r2	r0	0	5
`sll r0, r1, shift`	0	$zero	r1	r0	shift	6
`srl r0, r1, shift`	0	$zero	r1	r0	shift	7
`slt r0, r1, r2`	0	r1	r2	r0	0	8

Note: Here **r0,r1,r2** denote registers and **shift** denotes the shift amount.

3.2 I-type instruction

The general layout for an I-type instruction is as follows:

Opcode	rs	rt	Immediate
6 bits	5 bits	5 bits	16 bits

The encoding scheme for an individual I-type instruction is as follows:

Instruction	Opcode	rs	rt	Immediate
`addi r0,r1,c`	16	r1	r0	c
`addiu r0,r1,c`	15	r1	r0	c
`andi r0,r1,c`	14	r1	r0	c
`ori r0,r1,c`	13	r1	r0	c
`lw r0,c1(r1)`	12	r1	r0	c1
`sw r0,c1(r1)`	11	r1	r0	c1
`beq r0,r1,c2`	10	r0	r1	c2
`bne r0,r1,c2`	9	r0	r1	c2
`bgt r0,r1,c2`	8	r0	r1	c2
`bgte r0,r1,c2`	7	r0	r1	c2
`ble r0,r1,c2`	6	r0	r1	c2
`bleq r0,r1,c2`	5	r0	r1	c2
`slti r0,r1,c`	4	r1	r0	c

Note: Here **r0** and **r1** denote registers. **c**, **c1**, **c2** denote integers with **c1** must always be divisible by **4** (due to byte addressing) and **c2** denotes the relative branch address from the current position of the program counter.

3.3 J-type Instruction

The general layout for a J-type instruction is as follows:

Opcode	Address
6 bits	26 bits

The encoding scheme for an individual J-type instruction is as follows:

Instruction	Opcode	Address
`j c`	1	c
`jr r0`	2	r0
`jal c`	3	c

Note: Here **r0** denotes a register and **c** denotes absolute jump address.

4 PDS4 - Instruction fetch

The program counter (PC) points to the address of the next instruction, which is retrieved from the instruction memory and executed.

5 PDS5 - Instruction Decoder

This module outputs the type of instruction on the basis of the opcode.

Opcode	Instruction Type
0	R type
1-3	J type
4-16	I type

6 PDS6 - Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit has four inputs: Two of them (**A,B**) are operands on which the arithmetic operation is performed, third is the control signal (**ALUCtrl**) which determines the type of arithmetic operation and the last input is the shift amount value (**shamt**).

ALUCtrl	Arithmetic Operation
0	`AND`
1	`OR`
2	`ADD`
3	`Non-Equality Comparison` ($\neq$)
4	`Strictly Less than (<)`
5	`Less than or equal to` ($\leq$)
6	`SUB`
7	`Greater than or equal to` ($\geq$)
8	`Greater than (>)`
10	`Shift Left Logical`
11	`Shift Right Logical`
12	`NOR`

The ALU has two outputs: The first (**out**) stores the output value after performing the operation on the operands. The other output is a **Zero** wire which is asserted when **ALUResult** equals **0**. The value of **Zero** is used to check for branching instructions while updating program counter.

7 PDS7 - Branching Operations

7.1 Unconditional Branching:

For the unconditional branching operations like `j` and `jal`, we directly jump to the corresponding address stored in the lower **26** bits in the instruction. For `jal`, we also store **PC+4** value in the return address register **$ra**. For `jr`, we jump to the address stored in the register specified in the instruction.

7.2 Conditional Branching:

For conditional branching, we use the **Zero** wire of the ALU. Based on whatever comparison is specified in the instruction, we use the corresponding ALU function to assert the **Zero** wire which updates the program counter to the corresponding branch address.

Instruction	ALUCtrl Used
`bne`	3
`ble`	4
`bleq`	5
`beq`	6
`bgte`	7
`bgt`	8

8 PDS8 - Control Unit

The processor implements two control units for sending appropriate commands to the other modules.

8.1 Main Control Unit

This module takes the **opcode** of an instruction as input and outputs various control signals that decide the flow of data and the execution of the instruction properly. The meaning of various control signals is listed below:

- **ALUOp**: Control Signal for the other control unit ALUCtrl and decides on the type of the instruction i.e. I-type, R-type or J-type
- **Jump**: Determines whether a jump has to occur or not
- **RegDst** : Chooses destination register for writing (whether it is `rd` or `rt` or `$ra` in case of `jal` instruction)
- **RegWrite**: Determines whether the register on the Write register input has to be written with the value of the Write Data line
- **ALUSrc**: Decides whether the second operand of the ALU comes from the second register file output or is a sign-extended value
- **MemtoReg**: Determines whether the value written on the Write Data lines comes from ALU or from Data Memory or is it **PC+4** (in case of `jal` instruction)
- **MemRead**: Decides whether data from the data memory is to be accessed or not
- **MemWrite**: Decides whether data is to be written to the data memory or not
- **Branch**: Determines whether a branch has to occur or not
- **jr_Control**: Control signal for detecting the `jr` instruction

The values of the control signals for various opcodes are as follows:

Opcode	**ALUOp**	**RDst**	**RWrite**	**MRead**	**MWrite**	**MtoR**	**ALUSrc**	**jr_ctrl**	**Branch**	**Jump**
0	0	1	1	0	0	0	0	0	0	0
1	2	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	1	0	0
3	2	2	1	0	0	2	0	0	0	1
4	1	0	1	0	0	0	1	0	0	0
5-10	1	0	0	0	0	0	0	0	1	0
11	1	0	0	0	1	0	1	0	0	0
12	1	0	1	1	0	1	1	0	1	0
13-16	1	0	1	0	0	0	1	0	0	0

8.2 ALU Control Unit

The ALU Control Unit takes **ALUOp**, **Opcode** and **Funct** as inputs and returns the appropriate ALU operation to be performed.

ALUOp	Opcode	Funct	ALUCtrl
0	0	0	2
0	0	1	6
0	0	2	2
0	0	3	6
0	0	4	0
0	0	5	1
0	0	6	10
0	0	7	11
0	0	8	7
1	4	NA	7
1	5	NA	5
1	6	NA	4
1	7	NA	7
1	8	NA	8
1	9	NA	3
1	10	NA	6
1	11	NA	2
1	12	NA	2
1	13	NA	1
1	14	NA	0
1	15	NA	2
1	16	NA	2

9 PDS9 - Bubble Sort

The MIPS code, as well as the machine code for *Bubble Sort*, are included in the zip file.

10 PDS10 - Execution

The execution of the processor for *Bubble Sort* was shown in the lab. To run the code, compile the `CSE_Bubble_tb.v` file and execute its output file. A sample `Execution.txt` file shows the output on running the code.

Initial Unsorted Array:
A[0] = 5, A[1] = 7, A[2] = 3, A[3] = 9, A[4] = 1, A[5] = 7, A[6] = 3, A[7] = 9
Final Output:
A[0] = 1, A[1] = 3, A[2] = 3, A[3] = 5, A[4] = 7, A[5] = 7, A[6] = 9, A[7] = 9

Note: All the relevant codes are included in the zip file.