CS220 Assignment 7

Nikhil Gupta and Gautam Bansal

Department of Electrical Engineering, IIT Kanpur, 208016, UP, India.

Contributing authors: nikhilg20@iitk.ac.in; gbansal20@iitk.ac.in;

Problem Statement

We want to create a new processor CSE-BUBBLE that has the instruction set architecture as shown in Tab.1 below. We assume that the processor word size and instruction size are 32 bits and we use the VEDA memory as implemented in Assignment 4. The VEDA memory has two parts: a) Instruction Memory, b) Data Memory. We also assume that RISC-BUBBLE has total 32 registers of which certain registers follow the same roles as in MIPS-32 ISA. Finally we shall build a single-cycle instruction execution unit for CSE-BUBBLE.

1 PDS-1 (Registers and their usage protocol)

For implementing this RISC-BUBBLE, we used 32 registers of 32 bits in our architecture. The register naming along with their usage is given below.

Name	Number	Usage
$zero	0	Constant 0
$at	1	Assembler temporary
$v0-$v1	2-3	Function return values
$a0-$a3	4-7	Function arguments
$t0-$t9	8-15	Temporary data
$s0-$s7	16-23	Saved registers
$k0-$k1	26-27	Reserved for kernel
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Function return address

2 PDS-2 Data and Instruction Memory Size

The memory element we implemented is similar to **VEDA MEMORY** which we implemented in one of previous assignments. It is divided into two parts one for storing *Instruction Memory* and the latter for *Data Memory*. We have implemented byte addressing, meaning we can address each byte (8 bits). So, for storing 32 bit instruction or 32 bit data we need to combine 4 byte together and then store it in their respective memory locations.

- **Instruction Memory:** It has 1024 rows each of 8 bits.
- **Data Memory:** It has 256 rows each of 8 bits.

3 PDS-3 Instruction Layout & Encoding Methodology

There are three types of instructions in our processor architecture.

3.1 R-Type Instructions

Instruction layout and encoding methodology for the R-Type instruction.

Opcode	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Instruction	**Opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**
add	0	rs	rt	rd	0	1
sub	0	rs	rt	rd	0	2
addu	0	rs	rt	rd	0	3
subu	0	rs	rt	rd	0	4
and	0	rs	rt	rd	0	5
or	0	rs	rt	rd	0	6
sll	0	rs	rt	rd	0	7
srl	0	rs	rt	rd	0	8
slt	0	rs	rt	rd	0	9

3.2 I-Type Instruction

Instruction layout and encoding methodology of a I-type instruction.

Opcode	**rs**	**rt**	**Constant or Immediate**
6 bits	5 bits	5 bits	16 bits

Instruction	Opcode	rs	rt	Constant or Address
addi	1	rs	rt	immediate
addiu	2	rs	rt	immediate
andi	3	rs	rt	immediate
ori	4	rs	rt	immediate
slti	5	rs	rt	immediate
lw	6	rs	rt	offset
sw	7	rs	rt	offset
beq	8	rs	rt	offset
bne	9	rs	rt	offset
bgt	10	rs	rt	offset
bgte	11	rs	rt	offset
ble	12	rs	rt	offset
bleq	13	rs	rt	offset

3.3 J-Type Instruction

Instruction layout and encoding methodology of a J-type instruction.

Opcode	address
6 bits	26 bits

Instruction	Opcode	Address
j k	14	k
jr ra	15	ra
jal k	16	k

4 PDS-4 Instruction Fetch

This part is implemented in code in file name ***instruction_fetch.v***

5 PDS-5 Instruction Decode

This part is implemented in ***instruction_decode.v***. PC is a input to this module which then uses ***instruction_fetch.v*** module to fetch the instruction from instruction memory. Then the type of instruction is returned according to the opcode of instruction.

6 PDS-6 Arithmetic Logic Unit (ALU)

This part is implemented in ***alu.v*** The arithmetic logic unit takes three inputs.

1. **input1:** This is first operand.
2. **input2:** This is second operand.
3. **input3:** This is a control signal corresponding to control signal value the action is performed on input1 and input2.

There are two outputs in of ALU.

1. **output1 (zero flag):** This output is set when the result is zero.
2. **output2:** This output is the result of the arithmetic operation performed on input1 and input2.

7 PDS-7 Branching Operations

This operation depends on the flag *branch* which comes out from *control.v* and *zero flag* which is an output of *alu.v*. The *zero flag* will be set when the branching condition is true and the branch flag will be set when instruction is of type branch.

8 PDS-8 Final Processor

Final processor is the main program which first uses *instruction_fetch.v* module to fetch instructions and data as per requirement. And then sends instruction to *control.v* to decode instructions. And then decoded flags goes to corresponding modules and further actions takes place such as arithmetic, data storage, load or jump.

9 PDS-9 Bubble Sort

The MIPS code and their binary instruction for *Bubble Sort* is implemented in the code.

10 PDS-10 Final Run

The machine code (binary instructions) is written in *instruction memory* and array of integers is stored in *data memory* and then final *testbench.v* gets executed and the final sorted array gets printed.