CS220: Assignment 7

CSE-BUBBLE Processor

Chitwan Goel (210295) & Siddharth Kalra (211032)

PDS1: Decide the registers and their usage protocol.

We will use 32 registers in the processor. Details are as follows:

- Register number 0 stores the constant 1 to efficiently add and subtract one for the loops.
- Register number 10 is used to store 0.
- Register number 1 to 9 and 11 to 15 are used to store temporary values, they have the same role as temporary registers in the standard register file of processor.
- Register number 16 to 31 are used to store variables. They have the same role as 's' registers.

PDS2: Decide upon the size of instruction and data memory in VEDA.

- We will use an instruction memory of 2^{16} words, where each word (instruction) is of 32 bits. Therefore, the size of instruction memory will be $2^{16} \times 32$.
- The data memory will be separate with the instruction memory and its size will be same as former, i.e., $2^{16} \times 32$.

PDS3: Design the instruction layout for I-type, R-type and J-type instructions and their respective coding methodologies.

We will use a layout somewhat different from MIPS ISA for our instruction set. Below is the description of the layout for the three types of instructions:

1. R-type instructions: Instructions which involve arithmetic or logical operations on two registers and then storing the result in a third register are included in this. Here is the division of 32 bits into different parts for R-type instructions.

op (6 bits)	r_dest (5 bits)	r_source1 (5 bits)	r_source2 (5 bits)	shamt (5 bits)	funct (6 bits)

- *op (opcode)* and *funct (function)* together determine the type of instruction to be executed.
- *shamt (shift amount)* represents the shift amount in case of shifting operations and remains 0 in other types of instructions.
- *r_*source1 and *r_*source2 represent the source registers for two operands.
- *r_dest* represents the destination register where the result after the operation will be stored.

2. I-type instructions: Instructions which involve two registers and an immediate (constant), data transfer instructions and conditional branch type instructions are included in this type of instructions. Here is the division of 32 bits into different parts for I-type instructions.

op (6 bits)	r_dest (5 bits)	r_source (5bits)	Constant or address (16 bits)

- *op (opcode)* determines the type of instruction.

- r_dest represents destination register and *r_source* represents the source register involved in executing the instruction.
- The last 16 bit represents the *constant* in case of arithmetic, logical or data transfer type instructions and *address* in case of conditional branch type instructions.

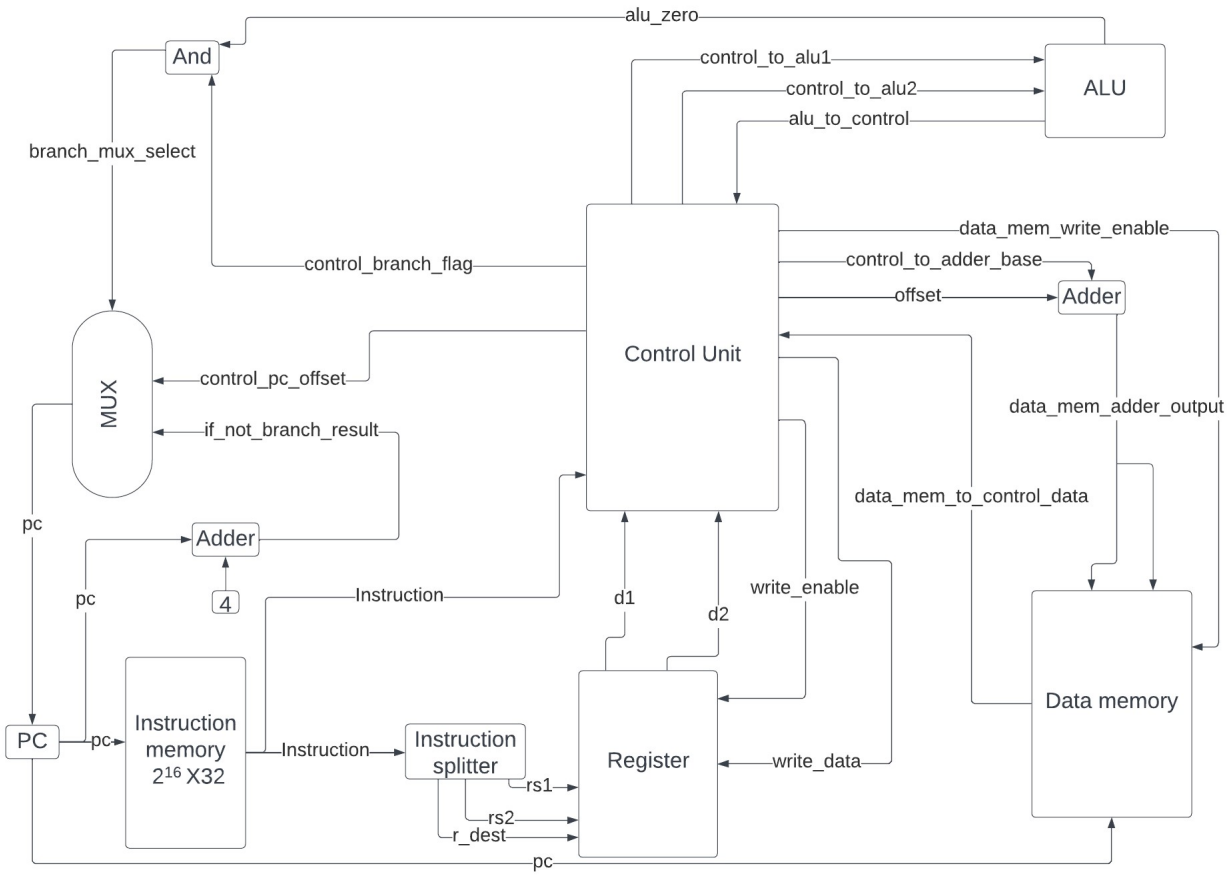
3. J-type instructions: Instructions which involve unconditional jump are included in this type of instructions. Here is the division of 32 bits into different parts for J-type instructions:

op (6 bits)	Address (26 bits)

- *op (opcode)* determines the type of instruction.
- *Address* represents the address where the program counter needs to be jumped.

Processor design:

The design is as follows. The names of wires is as per "main.v" file. Our Verilog code is exact implementation of this design.



How to run the code:

1. How to input the array to be sorted

We store data, i.e., number of elements in the array to be sorted in "data_mem.v", where mem[1] will store the number of elements in the array to be sorted and mem[2] onwards stores the array. Currently, we have stored a random array of 25 integers from 1 to 25 and set the number of elements as 25.

2. How to sort the array using the code

Main.v is the complete processor verilog file, that is, the overall/top module. Thus, to sort the array, we need to run main_tb.v file using the commands:

iverilog -o main_tb.vvp main_tb.v

vvp main_tb.vvp

This will print the sorted array on the console.

Instruction Layout for each instruction:

The opcode of varous instructions are as follows:

0: add

1: sub

2: addu

3: subu

4: addi

5: addiu

6: and

7: or

8: andi

9: ori

10: sll

11: srl

12: lw

13: sw

14: beq

15: bne

16: bgt

17: bgte

18: ble

19: bleq

20: j 10

21: jr r0

22: jal 10

23: slt

24: slti

1. For add, addi, addu, sub, subi, subu, and, andi, or, ori, normal R type instructions follow but in the way described above.

2. For sll,srl, R type instruction is followed, where shamt stores the shift amount, and rs2 is not used

3. For lw, eg lw $r0, 10($r1), I type instruction followed. r0 is stored in r_dest in machine code above. r1 stored in r_source1, offset (10 here, for ex) stored in the remaining 16 bits. Offset is stored as it is and not multiplied by 4, since we are working with array indices in verilog.

4. For sw, eg sw $r0,10($r1), R type followed. r0 stored in r_source1, r1 stored in r_source2, and offest in [r_dest+shamt+funct]. Note that 16 bits are not continuous. Offset, again, is not multiplied by 4 before storing.

5. For any branch operation: x $r0, $r1, c0, R type followed, r0 stored in rs1, r1 stored in rs2, const c0 in the 16 bits [r_dest+shamt+funct].

6. For jump operations, J type used, where the last 26 bits after opcode representing the instruction memory (not offset in our processor) where we need to jump.

7. For slt $r0, $r1, $r2 normal R type follows. For slti, normal I type instruction follows.