CSE-BUBBLE

PROF. URBI CHATTERJEE

Danish | CS220A | 210297
Jayesh | CS220A | 210296

REGISTER ARCHITECTURE

The VEDA memory has to be used in CSE-BUBBLE processor, so we will use it to hold our registers. The register architecture will be such that VEDA will hold 32 registers each 32 bits or 4 words wide. The memory allocation of these 32 registers is as follows:

- Instruction Registers (IRs): 16 out of 32 registers allocated. These will be mostly used internally for execution of an instruction in a program.
- Memory Register (MRs): 16 out of 32 registers allocated. These will be used to store data from the main memory into the processor to carry out the instructions. These will be used as 'variables' of MIPS32 program code.

The IRs will have following functions:

- IR0: Similar to $zero in MIPS
- IR1: Similar to $ra in MIPS
- IR2: Similar to $sp in MIPS
- IR3: Similar to $pc in MIPS
- IR4: This register will be storing current instruction that is to be executed
- IR5-IR8: These will be used to store values during procedure calls
- IR9-IR15: Will be used to store memories during subroutines of other instructions.

The VEDA memory code is as follows:

module veda_modified (clk, reset, write_enable, address_a, data_in, address_b, mode, data_out);

```
parameter scribble = 1'b0, interpret = 1'b1;
input clk, reset, write_enable, mode;
input [4:0] address_a, address_b;
input [31:0] data_in;
```

```
    output [31:0] data_out;

    always @ (posedge clk or posedge reset) begin
    if (reset) begin
        for (i = 0; i < 32; i = i + 1) begin
            memory[i] <= 32'b0;
        end
    end
    else begin
        if (write_enable && mode == scribble) begin
            memory[address_a] <= data_in;
            intermediate_reg <= data_in;
        end
        else if (write_enable) begin
            memory[address] <= data_in;
            intermediate_data <= memory[address_b];
        end
        else if (mode == scribble) begin
            intermediate_reg <= data_in;
        end
        else if (mode == interpret) begin
            intermediate_reg <= memory[address_b];
        end
    end
    end

    always @ (posedge clk) begin
        data_out <= intermediate_reg;
    end

endmodule
```

INSTRUCTION LAYOUT FOR R, I, J TYPE INSTRUCTIONS

We know that each instruction is 32 bit long and also VEDA has a height of 32 only so registers can be accessed with 5 bit addresses.

For R type instructions, we have 3 registers as an input to the instruction. Thus we will use the following layout for R type instructions:

6 bits (opcode)	5 bits (destination register)	5 bits (source register 1)	5 bits (source register 2)	5 bits (shift amount- for shift instructions)	6 bits (function code)

Function code specifies the exact instruction that has to be executed for the given op code.

The following map is used for R-type instructions of of CSE-BUBBLE:

INSTRUCTION	OPCODE	FUNCTION CODE
add	000000	100000
sub	000000	100010
addu	000000	100001
subu	000000	100011
and	000000	100100
or	000000	100101
sll	000000	000000
srl	000000	000010
slt	000000	101010

Now, I-type instructions will only have 2 registers as input and the remaining bits can be used as a constant or an address. In case of data transfer instructions, the 16 bits can be used as the offset.

6 bits (opcode)	5 bits (destination register)	5 bits (source register 1)	16 bits (imm - a constant or address)

The following map is used for I-type instructions of CSE-BUBBLE:

INSTRUCTION	OPCODE
addi	001000
addiu	001001
andi	001100
ori	001101
lw	100011
sw	101011
beq	000100
bne	000101
bgt	000111
bgte	000001
ble	000001
bleq	000110
slti	001010

Now, instructions of the J-type will have the following structure: j and jal do not have a register input. For consistency, they will have 6 bits for the opcode and 26 remaining bits can specify the target.

6 bits (opcode)	26 bits (target)

jr instruction has a register input. So we will need to reserve some bits in the instruction for the address of the register.

opcode (000000)	register (5 bits)	0000000000000000	funct (001000)

INSTRUCTION FETCH PHASE

The instruction fetch phase works in the following manner:

1. Initially the program loads in the main memory (RAM) at a pre-determined address, say 0. So the IR3 (which is working as $pc from MIPS) <- 0.

2. Now, the address from the program counter (IR3) is stored in the current instruction storing register (IR4). This register then calls the instruction to be executed. And the PC (IR3) is incremented by 4, which gives the address to the next location.

3. Now the instruction from the IR4 is decoded and executed.

4. Move to step 2 while the end of the instructions.

Following pseudocode explains it better:

IR3 <- 0

loop

IR4 <- MEM[IR3] // MEM[] : memory, RAM

IR3 <- IR3 + 4

Decode and Execute instruction in IR4

end loop