Semester Project

Theme: CPU Design in Verilog

Aim : To design a 8 bit processor connected to instruction and data memory.

In this assignment, you will design and implement a simple microprocessor with a custom instruction set. This project can be done in **groups (up to 4 members per team)**, but the final project report must give a clear breakup of portions of the design implemented by each team member.

For the purpose of this assignment, a CPU consists of the following components:

Instruction memory (read only)
Data memory (read / write)
Register file (two ports read, one port write simultaneously)
Arithmetic and Logic Unit (ALU)

Instruction format :

The CPU uses 16-bit instructions that are stored in the IMEM. There are 3 types of instructions:

Type	*15-12*	*11-9*	*8-6*	*5-3*	*2-0*
R (register)	*Opcode*	*Rd*	*Ra*	*Rb*	*Func*
I (immediate)	*Opcode*	*Imm[5:3]*	*Ra*	*Rb*	*Imm[2:0]*
J (jump)	*Opcode*	*[11:7] dont care*			*Addr [6:0]*

Assume that the register *r0* always stores the numeric value 0 (to make comparisons easy), and *Ra, Rb* can refer to any register.

The CPU you have to design will have the following instructions:

Instruction	*Opcode/Func*	*Type*	*Operation*
add rd, ra, rb	*0000 / 000*	*R*	*rd <- ra + rb*
sub rd, ra, rb	*0000 / 010*	*R*	*rd <- ra + rb*
and rd, ra, rb	*0000 / 100*	*R*	*rd <- ra AND rb*
or rd, ra, rb	*0000 / 101*	*R*	*rd <- ra OR rb*
addi rd, ra, imm	*0100*	*I*	*rd <- ra + imm*
lw rd, imm(ra)	*1011*	*I*	*rd <- DMEM[ra + imm]*
sw rd, imm(ra)	*1111*	*I*	*DMEM[ra + imm] <- rd*
beq rd, ra, imm	*1000*	*I*	*if (ra == rb) pc <- pc + imm*
j addr	*0010*	*J*	*pc <- addr X 2*

Notes:

1. The *imm* operand is split into two parts to make the instruction easier to decode - this is just for simplicity.

2. For the **J instruction**, the address is 7 bits which is then extended to 8 bits. This means you can only jump to even numbered addresses. Why? Simplicity.

Circuit elements :

Instruction memory

This can be a combinational unit - it takes just the *address bus (8 bit value)* as input, and gives out a 16-bit value that is the instruction to be decoded. The address is provided by the *Program Counter* (PC). You can use the following module to represent instruction memory

```
module imem (input [7:0] Addr
output [15:0] rd);
// Addr is the address of instruction to fetch
// for our purpose can be taken from ProgramCounter[7:0]
reg [15:0] RAM[256:0];
initial
$readmemh ("memfile.dat",RAM);
assign rd <= RAM[Addr]; // word aligned
endmodule
```

The program is assumed to start from **0X00**

Program Counter

It is a 8 bit register. Under normal operations, will always increment by 1 on every clock cycle, to access the next instruction. In case of a *JUMP or BRANCH* type of instruction, will go to the value specified by the output of the ALU or the immediate operand in the instruction.

Register file

This is a set of 8 registers, each storing an 8 bit value. There are 2 output values ra and rb, whose values are selected based on the instruction word, as in the instruction definitions given above. There is one port by which data can be written into one of the registers, but make sure to give an enable signal to control whether or not to update the value.

Data Memory

This needs to be a sequential / clocked unit. At any given cycle, you are either reading from it or writing to it, with the address given by the address bus. In case of a *lw (load-word) or sw (store-word)* instruction, the address is either immediate or the output of the ALU. The data output of the *DMEM* will always go into the register file, where it gets stored into a register selected by rd.

Arithmetic and Logic Unit (ALU)

The core of the processor - all the actual computations are performed here. As shown in the instruction set, operations such as addition, subtraction and logical operations are all done in this unit. Also, the output of the ALU is used as the address for certain memory related operations.

Control unit

The unit that actually makes the entire processor work as expected.

The input to this is the instruction word, and the output is a set of control signals that decide, for example, whether the register file is to updated, what operation is to be done by the ALU, whether memory read or write is required etc.

One way to implement this is to make it a combinational block that will generate the following signals:

MemToReg: does data being written into the register file come from memory or from output of the ALU
MemWrite: is the present operation going to write into memory?
RegWrite: is an entry in the register file (pointed to by rd) supposed to get an updated value in this instruction?
ALUSrc: is this an immediate operation, or a register operation?
Branch: If the current instruction is a branch, then use this signal as select for a multiplexer to feed the PC.
Jump: Control the PC input MUX and also the immediate operand for input to PC.

Implementation :

Before you start implementing the blocks in verilog, it is expected that you plan the design - the data path and the control. **You may consult the faculty and the TAs for the same during the Tutorial classes.**

Each of the blocks described above should be implemented as a separate verilog module with suitable input and output port definitions.

Testing :

a. Write a program to multiply two numbers - assume the numbers are stored in memory locations *DMEM[0] and DMEM[1]* and the output (16 bits) should be stored in two registers.

Evaluation :

a. Planning and Presentation : Each group would be asked to plan the data path and control and have to present their work during the tutorial classes. The groups will be evaluated on the basis of their presentation and planning in each of the modules - the data path design and the control

b. Report : Your final report should consist of the following:

i. The architecture of the CPU on paper with complete details of the design and the instruction set.

ii. Verilog code for your design

c. Result of the design tested on different input instances (Bonus points for Fastest processor)