Simple Machine

The goal of this project is to design a machine that can run some simple instructions on registers and return the final result.

First to generilize the idea and devide each function task, I separate Memory module (contains registers) and Excutor module (decode and excute opcodes)

these modules shares a common bus line (8 bit) to communicate (Read/Write)

because there is no CU (control unit) to send OpCodes to Excutor I also write a verilog module test named Instruction.v that reads a file of test opcodes and excute them line by line

Memory Module:

to implement a general memory module I use parameters to receive N (size of DataBus or size of each memory cell -aka register- in bits) and M (size of AddressBus so we have 2^M memory cells to address)

this module has and input/output DataBus in Read mode on Write mode, so I can read given data.

R/W wire choose how this module works. In 0 (Read mode) I write selected register value and in 1 (Write mode) I write 'Z (High-Ampdance) on DataBus so I can read from it, then I change the future value of selected cell to value readed from DataBus.

All changed happened async on Low level of ResetN or sync on possitive edge of Input Clock, In ResetN=0 all registers values sets to zero, and on posedge clock I write register value on DataBus (Read mode) or Read register value from DataBus and save it for next clock.

AddressBus (Select) is M-bit bus which is simply a binary number reperesents index of target register to Read/Write.

To design this module, I use generative ability in verilog so I can set M from parent modules and have desired Memory module. to do that, first I need a genvar variable, then in for loop I name the block which allows me later to define local registers in each itterations, next to implement R/W mechanism I assign that register to an internal wire (bus) if Address(Select)Bus Matches register index else I use 'z to separate it from assign (High-Ampedance).

Then to implement Write mode for each register (cell) I write an always block sensetive to async ResetN LowLevel or Clock posedge then in it I check if ResetN=0. if it is then I set register value to zero else check if SelectBus address matches the register index, if it's not register will hold its value else I write new value from DataBus to it.

At last I assign the internal DataBus value to real DataBus when RW in Read mode else separate them by 'Z.

Excutor Module:

It is heart of the system, it reads 20-bit OpCode from input at posedge Clock separate 4bit as Part1 or Command, next 8 bit as Part2 or Reg1 and remaining 8 bit as Part3 or Reg2.

then in parallel Par2 and Part3 wires (buses) feed to two OnHotDecoder to decode possible onehot data from these two parts (lately they are used in some Commands). these decoded OnHot choose Reg1 and Reg2 index to operate on them as Command logic wants.

also these two parts parallely feed to a FloaterModule (described later) which its 16bit output needed in float opperation.

this excutor module has an 16-bit output register bus that change on each Excution cycle to it's final result. also I use it's values to save result in excution of an operation.

this excutor has an 3-bit timer which resets at start of each opcode excution and holds current state of operation. some Commands need 2 some need 3, ... T to excute, after excution of an OpCode is done Done output wire sets to 1 to notify parent module (implementor) to use it's result or sends another OpCode to excute.

this module has an inout(Input/Output) bus shared with memory module (parent module must wire them together) and also this module controls RW line of memory module.

there are two other outputs SignFlag and ZeroFlag which I set them at the end of each Computational Commands (Add-Sub-Mul-Div-Shift-Float) to represente the true state of output.

to implement each command first I check the part1 (CMD) wire with predefined constant equalant (requested in project document) values (I use localparam to name each)

then in each case I write logic acording to Time state:

* LoadConst command:

T=0: set DataBus to part2 (constant value), AddressBus to part1 OneHot decoded value then set RW to Write mode, increse T, set output to constant value

T=1: signal DONE to parent module, set back mode to read (to avoid unwanted writes)

* LoadReg command:

T=0: set RW to read and write AddressBus to part1 OneHot decoded value

T=1: read value in DataBus, save it to output, write it to DataBus, write Reg2 OneHot decoded address (from part2), set RW to write

T=2: signal DONE to parent module, set back mode to read (to avoid unwanted writes)

to calculate computational commands I use builtin verilog language features:

A+B, A-B, A*B, A/B, A<<B, A>>B

* ...

* MulReg command:

T=0: [ReadReg2]: set RW to Read, set Address to onehot decoded part2, increase T

T=1: [SaveReg2, ReadReg1]: tempReg=DataBus, set RW to Read, set AddressBus to onehot decoded Re1, increase T

T=2: [compute, SaveRegA]: output=DataBus*tempReg, set RW to Write, set AddressBus to RegA, set DataBus to lower byte of Output, increase T

T=3: [SaveRegD]: set RW to Write, set AddressBus to RegD, set DataBus to upper byte of Output, increase T

T=4: [DONE]: set SignFlag, setZeroFlag, RW is Read, signal DONE

***: RegA=0, RegB=1, RegC=2, RegD=3, these are predefine index for known registers

* ...

all other commands use the same idea as above to implement desired action.

this module also inputs posedge sync Clock and async Low Level ResetN line, in ResetN=0, all the internal states (output, T, RW, SelectAddress, SignFlag, ZeroFlag) retsets to 0.

in all steps of these design I use separate test modules to simulate input and outputs of each module and check their result with my expectation and fix any bugs (check project github commit history for more detail)

Floater Module:

in this module I convert a fixed point decimal to 16-bit floating point version, to implement, first I calculate B or RHS(a'ashari) from it's decimal base to binary base (cascade multiply by two) after that I have an 8-bit binary mantis.

then I check A or LHS(sahih) to see if it's above zero or not.

1) if it is above zero then I find it's last 1 bit (it's index is equal to the power needed to multiply mantis part to get the number for example 110 power is two because 110=1.10*(2^**2**). then add this power by 127 to get floating-

point exponent part.

then to result mantis, first I write new A (after extract power there is a single 1 in left and (8 minus power) bits of 0/1 on right) right part to result mantis, then to fill remaining bits I ues previous calculated (coverted) mantis to fill, at last result is { 0 as sign + 8bits of exp + 7 bits of mantis }

2) if A is zero, then I search mantis for last 1 and power is (8-index). then exponent is 127-power (because its negetive) and I skip this last 1 bit (becuase 0.01101 = 1.101*(2^-2)) and write remaining mantises to result mantis, then fill the remaining result mantis bits with zeros. (so 1.101 mantis is 1010000).

after that the result is calculated the same as before.

OneHotDecoder:

the goal of this module is to convert OneHot (2^N)-bit inputs to N bit binary output.

to reach the goal I loop throw all lines and find the last (so greater line indexed has pariority to lower) 1 and write it's index to output

so this module won't handle errors and there is always a result in it (0..00 in put is same as 0..01 and so on)

test-instructions:

to test my excutor I wrote this, It first read instruction file then loop in each OpCode and set OpCode line to it and wait for result.

to simulate clock I use an always block with 10ns delay,

to show output I use another always block sensetive to changes and show it to console

and to excute each opcode first I wait for clock low level when done is high (I can excute new command) then set it.

I increse counter after this beacuse know it is sync with my each excution cycle, (now it's easier to follow in signal panel to read each state)

also I decrease simulation precison to 100ps to speed up simulation.