CSE331 / BIL503

COMPUTER ORGANIZATION

FINAL PROJECT: 32-BITS SINGLE CYCLE MIPS PROCESSOR

DUE DATE: 19 JAN 2024, 23:59

In this assignment you will be designing a 32-bit MIPS processor on Verilog. The following instructions should be implemented. The implementation of **MOVE** instruction is a must.

Instruction	Example	Meaning	Type	OPCODE	FUNC. CODE
add	add $1,$2,$3	$1 = $2 + $3	R	000000	000010
subtract	sub $1,$2,$3	$1 = $2 - $3	R	000000	000011
add immediate	addi $1,$2,100	$1 = $2 + 100	I	000010	NA
subtract immediate	subi $1,$2,100	$1 = $2 - 100	I	000011	NA
and	and $1,$2,$3	$1 = $2 & $3	R	000000	000100
or	or $1,$2,$3	$1 = $2 | $3	R	000000	000101
and immediate	andi $1,$2,100	$1 = $2 & 100	I	000100	NA
or immediate	or $1,$2,100	$1 = $2 | 100	I	000101	NA
load word	lw $1,100($2)	$1 = Memory [$2 + 100]	I	001000	NA
store word	sw $1,100($2)	Memory [$2 + 100] = $1	I	010000	NA
load byte	lb $1,100($s2)	$1={24'b0,M[R[rs]+SignExtImm](7:0)}	I	001001	NA
store byte	sb $1,100($2)	M[R[rs]+SignExtImm](7:0)=R[rt](7:0)	I	010001	NA
set on less than	slt $1,$2,$3	if ($2 < $3) $1 = 1; else $1 = 0	R	000000	000111
set on less than immediate	slti $1,$2,100	if ($2 < 100) $1 = 1; else $1 = 0	I	000111	NA
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC + 4 + 100	I	100011	NA
branch on not equal	bne $1,$2,100	if ($1! = $2) go to PC + 4 + 100	I	100111	NA
jump	j 1000	go to address 1000	J	111000	NA
jump and link	jal 1000	$ra = PC + 4; go to address 1000	J	111001	NA
jump register	jr $1	go to address stored in $1	R	000000	001000

Move Instruction:

Instruction	Example	Meaning	Machine Code			
move	**move $rt,$rs**	**$rt = $rs**	100000 (6 bits)	Rs (5 bits)	Rt (5 bits)	0000000000000000 (16 bit)
			PS: 100000 is the opcode of move instruction.			

IMPORTANT! If your design doesn't support MOVE instruction, you get 0.

The machine code of representation of the other instructions are defined below.

R-type:

Opcode (6 bits)	Rs (5 bits)	Rt (5 bits)	Rd (5 bits)	Shamt (5 bits)	Func. Code (6 bits)

I-type:

Opcode (6 bits)	Rs (5 bits)	Rt (5 bits)	Immediate (16 bits)

J-type:

Opcode (6 bits)	Address (26 bits)

- The memory addresses will be 18-bits. Which means your data memory is 256 KB.
- Instruction memory should hold up to 1024 instructions where each instruction is 32-bits. Thus, the least significant 10 bits of the jump addresses will be enough. But you should fill the rest of the address block with zeros. For instance, jump instruction's address block (in machine code level) should be 26 bits even though you only use the least significant 10 bits.
- The information (the names, inputs, outputs etc.) about the some of the verilog modules will be announced. **You should stick into that format. If you don't, you get 0, even though your implementation works.**
- **If you don't implement MOVE instruction, you get 0.**
- **If your design is not working, you will get at most 30 pts.**
- You can use the ALU you designed for Project 2 (but you will need to upgrade it).
- Don't forget byte-access (for lb and sb) and sign extend (for immediate instructions).
- Despite there is a "Shamt" block in the machine code representation of R-type instructions, you will not implement shift instructions. Therefore, Shamt will always be "00000".
- This document doesn't describe the project fully. The details will be shared in the Problem Session on 25th December, 14:00.
- You will be using structural verilog only, except a few modules (which will be shared).
- You will be reading instructions from instruction memory (which is a file in this project) and then write the results to registers or memory (which are also files). More information about file operations will be shared in the Problem Session.
- You will also write a testbench, the details will be shared.
- Since this is a detailed project, new requirements could be shared. You are responsible of following the announcements made on Teams page.
- You should only use structural verilog except the modules **mips**, **register_block**, **instruction_block**, **memory_block**. In these modules, you should use structural verilog if you can (for instance, calculation of PC + 4 in mips can be done by using structural verilog).
- The names of the files should be "instructions.mem", "registers.mem", "memory.mem"

Bonus Part: Any successful hardware implementation of the project grants you **extra 25 points**. Such an implementation requires a face-to-face demo.

PS: The hardware will be provided for the students who wish to do the bonus part. Thus, you need to contact with TA.

PS: The weight of this project is higher than the previous ones.

REQUIRED MODULES:

- **mips** (Top-level entity)
  - (input clock);
- **register_block** (
  - output reg [31:0] read_data1,
  - output reg [31:0] read_data2,
  - ***input*** *byteOperations,* *(This is an **optional** signal for lb and sb operations)*
  - input [31:0] write_data,
  - input [4:0] read_reg1,
  - input [4:0] read_reg2, *(These were assigned as [5:0] before, by mistake.*
  - input [4:0] write_reg, *They should be [4:0] since register addresses are 5 bits long, not 6)*
  - input regWrite);
- **instruction_block** (
  - output reg [31:0] instruction,
  - input [31:0] pc);
- **memory_block** (
  - output reg [31:0] read_data,
  - *input* byteOperations, (This is an **optional** signal for lb and sb operations)
  - input [17:0] address,
  - input [31:0] write_data,
  - input memRead,
  - input memWrite);
- **control_unit** (
  - output reg regDst,
  - output reg branch,
  - output memRead, (This can also be used as memToReg)
  - output memWrite,
  - output [2:0] ALUop, (This was assigned as [1:0] before, by mistake. It should be [2:0])
  - output ALUsrc,
  - output regWrite
  - output jump,
  - output byteOperations, (This is an **optional** signal for lb and sb operations)
  - output move, (This is an **optional** signal for MOVE operation)
  - input [5:0] opcode);
- **alu** (
  - output reg [31:0] alu_result,
  - output reg zero_bit,
  - input [31:0] alu_src1,
  - input [31:0] alu_src2,
  - input [2:0] alu_ctr);
- **alu_control** (
  - output reg [2:0] alu_ctr,
  - input [5:0] function_code,
  - input [2:0] ALUop);
- **sign_extend** (
  - output reg [31:0] sign_ext_imm,
  - input [15:0] imm);
- **shift_left_2** (
  - output reg [31:0] shifted_address,
  - input [31:0] address);

ALUop and ALUctr signals:

OPERATION	ALUctr
AND	000
OR	001
XOR	010
NOR	011
LESS THAN	100
ADD	101
SUB	110
MOD	111

ALUop (generated and sent by control_unit, received by alu_control)	INSTRUCTION	OPERATION (the operation that will be performed in ALU)	ALUctr (generated and sent by alu_control and received by alu)
000	andi	AND	000
001	ori	OR	001
100	slti	LESS THAN	100
101	addi, lb, sb, lw, sw	ADD	101
110	subi, beq, bne	SUB	110
111	R type inst.	Depends on the function code	Depends on the function code

- Move operation will not use ALU, that's why it is not listed in the table above. But if you want to take the output of MOVE operation from ALU, you can use any of the ALU operations.
- PS: Make sure your design supports byte-addressing to get full credit.
- PS: You don't have to fill the memory.mem file for all of the 256 KB since it takes too much time in Quartus to assemble (you can use a small file with less data in it). Same goes for the instructions.mem. **But address lengths remains the same.**
- PS: You don't have to use outputs as "reg".
  For instance:
  The following output of memory block could be
  - output reg [31:0] read_data,

  or
  - output [31:0] read_data,

  This is up to your decision.

  This also applies vice versa.
  For instance:
  The output of control unit could be
  - output memWrite

  or
  - output reg memWrite