University of California, Irvine

Team Superheroes in Training

EECS 31L Final Project

Design Report

Date: November 28, 2016. 11:00PM

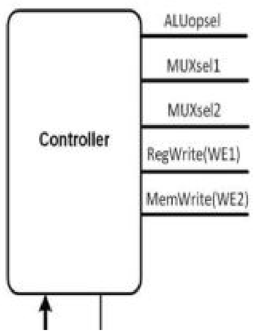
Gerry Su	16325043
Joshua Lazaro	57177939
JiaRui Zhu	75042047
Christopher Kevin Bravo	23748994

Tasks Done by Each Member:

- Gerry Su
    - Working on Design Report and providing explanations for how each module works both independently and in conjunction with others.
    - Come up with ingenious ideas for the code
    - Developed the Controller
    - Helped with the development of testbenches, testing and analysis of waveforms
- Joshua Lazaro
    - Taking descriptions of each module and translating them into Systemverilog code
    - Created Source files and testbenches to test each module
    - Developed the Instruction Block
    - Created processor.sv and connected all of the blocks together
- JiaRui Zhu
    - Developed the Register Block
    - Helped with the development of testbenches, testing and analysis of waveforms
    - Helping with understanding and briefing of project goals and design.
    - Helped with compiling the final draft of the report.
- Christopher Kevin Bravo
    - Help brainstorm the design project
    - Co-operate in the development of sources for the final code
    - Format the final draft of the report
    - Make sure data was accurately reported in the final draft of the report
    - Developed the Data memory and its testbenches

1 Input/Output Port Description

Controller:

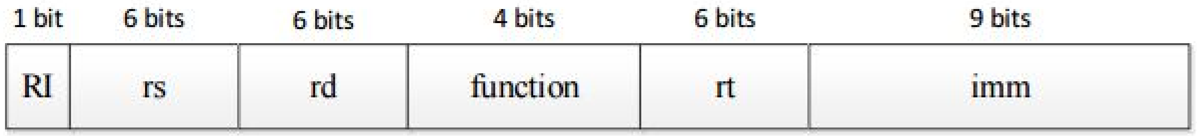


A controller/decoder which takes a 32 bit instruction as input (from Instruction Memory) and then set the function types, register indexes, and all the other controlling signals in your design for choosing appropriate selector lines of the multiplexer.

Inputs:

[31:0]	in32	/*32 bit instruction*/

Outputs:



1 bit	6 bits	6 bits	4 bits	6 bits	9 bits
RI	rs	rd	function	rt	imm

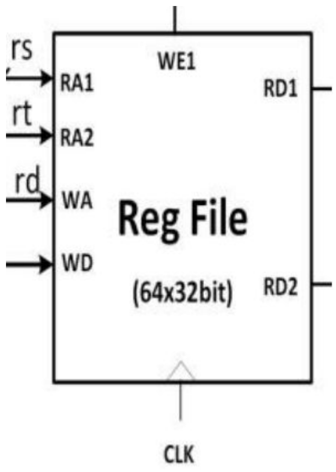
- Neglect imm from [8:0] if RI = 0 and output imm as[14:0] if RI = 1.

[3:0]	ALUOpsel	/*Outputs the corresponding ALU function opsel from the */ /*processor instruction set.*/

Flags

- MUXsel1: 1 if RI = 1 , else 0
- MUXsel2: 1 if function is LOAD (0100) or STORE (0110), else 0
- RegWrite(WE1): 1 if LOAD, else 0
- MemWrite(WE2): 1 if STORE, else 0

Register File (64x32bit):



Inputs:

[5:0]	RA1	/*read address 1*/
[5:0]	RA2	/*read address 2*/
[5:0]	WA	/*write register address */
[31:0]	WD	/*write data*/
[1 bit]	WE	/*write enabled*/
[1 bit]	CLK	/*clock signal*/

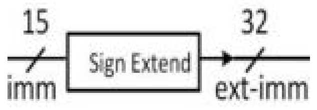
Outputs:

[31:0]	RD1	/*read data output 1*/
[31:0]	RD2	/*read data output 2*/

The register file acts as temporary storage for the processor's operations. It contains a memory consisting of 64 rows of 32 bits. The register file always reads data in the memory at some address (RA1 and RA2). The data is outputted only for the corresponding port. (Ex: RA1 corresponds to RD1) If the write enable is on, then the register file will also write some data (WD) at some address (WA).

The read data output of the register file is then connected to the ALU and used to perform some operation dictated by the operation selection code.

Reading occurs immediately despite the clock, but writing only occurs at the positive edge of the clock.



Sign Extender (15 to 32bit):

Inputs:

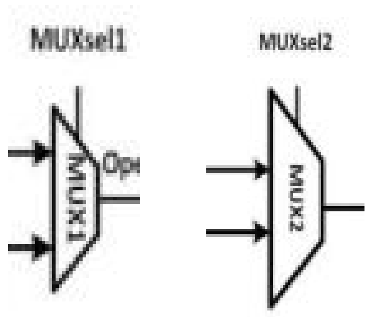
[14:0] in	/*input value*/

Outputs:

[31:0] out	/*sign extended value*/

Extends the sign of an input value by repeatedly concatenating the most significant bit of the input value.

Selection Multiplexer (32 bit, 2 inputs):



Inputs:

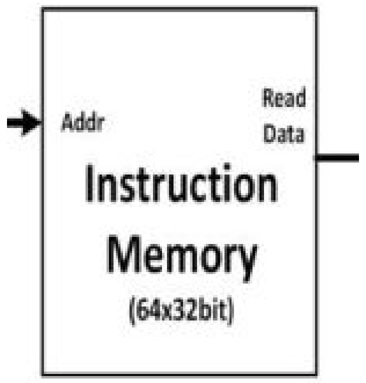
[31:0] a	/*input 1*/
[31:0] b	/*input 2*/
muxsel	/*selection signal*/

Outputs:

[31:0] Y	/*selected output*/

A multiplexer that selects the input to be passed through the output based on the multiplexer selection signal.

Instruction Memory (64x32bit):



Inputs:

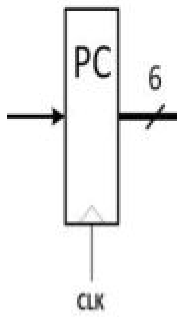
[5:0] addr	/*Current Instruction's Address*/

Outputs:

[31:0] read_data	/*Data of Current Instruction*/

The Instruction Memory contains a 2^6 size memory which contains 32-bit instructions. It receives a 6-bit input address to the current instruction from the PC. It outputs a 32-bit instruction set from the specified address to the Controller.

PC:



Inputs:

[5:0] addr	/*Address of Instruction to be executed on positive edge of clock*/
Clk	/*Clock*/
Rst	/*Reset*/

Outputs:

[5:0] addr_out	/*Current Instruction's Address*/

The PC outputs the next Instruction's address to the Instruction Memory on every positive clock edge. It contains a Reset that forces PC to output the address of the very first instruction (index 0) inside Instruction Memory.

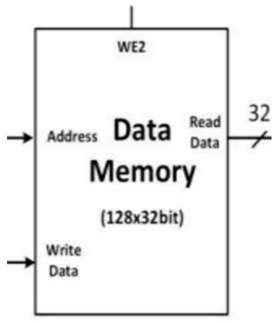
Adder:

Inputs:
[5:0] inc /*Value to increment each Instruction address by (default inc = 1)*/
[5:0] addr /*Address of the previous Instruction, to be incremented by inc*/

Outputs:
[5:0] addr_out /*Current Instruction's Address (After incrementing the previous address by inc)*/
The Adder serves the purpose of incrementing each instruction address by inc, outputting the next Instruction's address to the PC.

Data Memory:
Inputs:
[31:0]Address /*ALU result, selects the section of the Data Memory*/
[31:0]Write Data /*Read data output 2, give the possible writing data*/
[0] Writing Enable 2 /*Allows to write in the memory, works when load and store*/

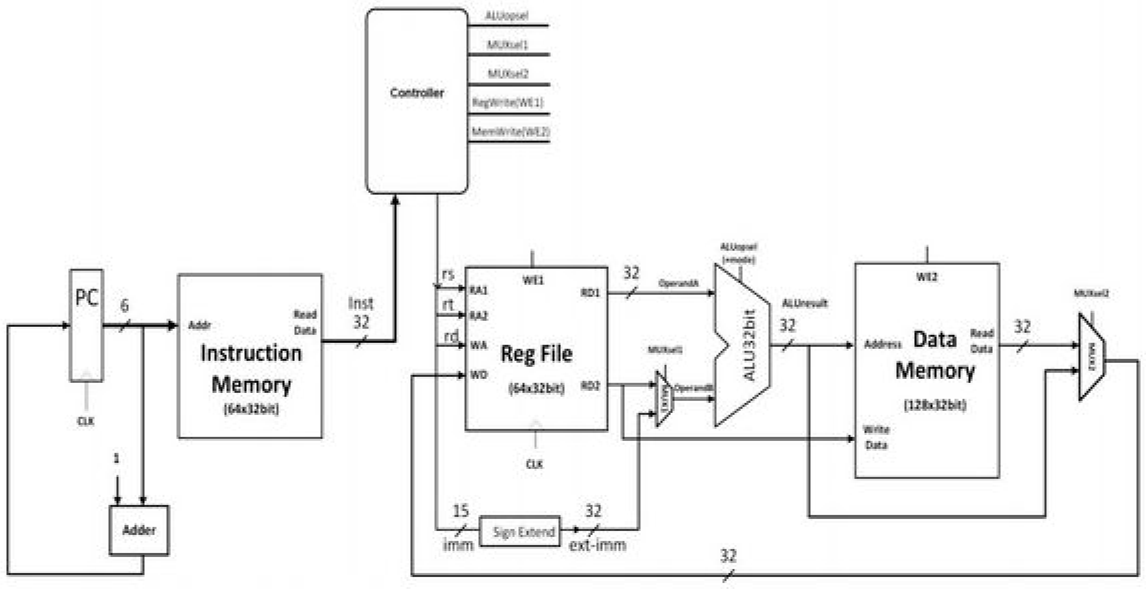
Outputs:
[31:0]Read Data /*Gives the answer produced by the Data Memory process

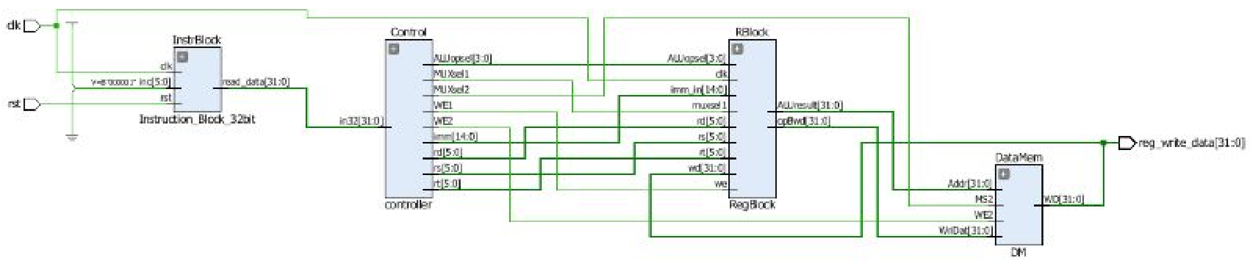


Processor **instruction** set:

Instruction	**Description**	**Function Code**	**Comments**
NOP	nothing	1111	
ADD / ADDi	rd <- rs + rt | rd <- rs + imm	0000	
SUB / SUBi	rd <- rs - rt | rd <- rs - imm	0011	
OR / ORi	rd <- rs OR rt | rd rs OR imm	1001	
NOT	rd <- NOT rs	1011	
XOR / XORi	rd <- rs XOR rt | rd <- rs XOR imm	1010	
SLL / SLLi	rd <- shift left (rs) | rd <- shift left (rs)	1101	rt/ imm value decides how many bits to shift
MOV / MOVi	rd <- rs | rd <- shift left (rs)	0010	
LOAD	rd <- Data Memory(rs)	0100	
STORE	Data Memory(rs) <- rt	0110	

2 Design Schematic





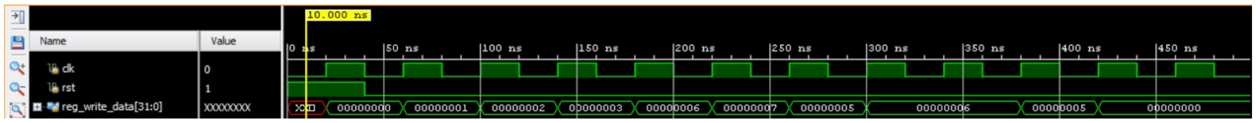
3 Simulation Waveform

Instructions hard-coded in instruction memory.

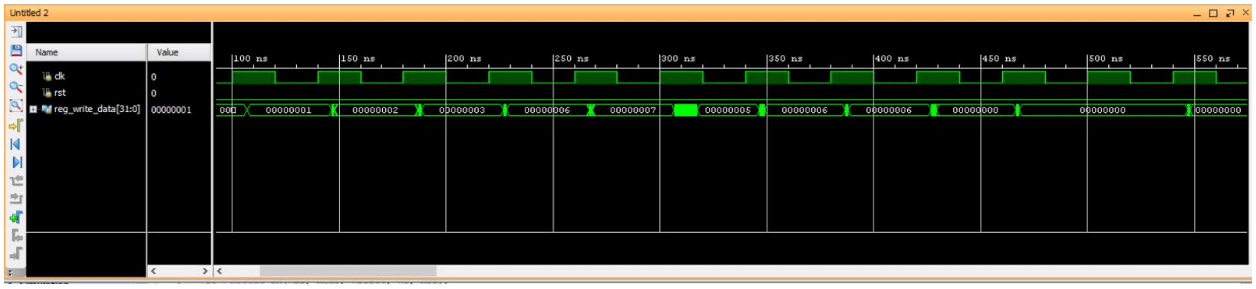
Here are the instructions passed through Controller.

Read (0) Immediate(1) (ri)	Source Register (rs)	Destination Register (rd)	Function	Target Register (6 bit) or Immediate(15 bit)	Description
0	000000	000000	1111	000000000000000	NOP
1	000000	000001	0000	000000000000001	R1 <- R0 + R1
1	000001	000010	0000	000000000000001	R2 <- R1 + 1
0	000001	000011	1010	000010	R3 <- R1 XOR R2
0	000011	000100	1101	000000000000000	R4 <- R3 SLL by 1
0	000100	000111	1001	000011	R7 <- R4 OR R3
0	000111	000101	0011	000010	R5 <- R7 - R2

1	000000	000110	0000	000000000000110	R6 <- R0 + 6
0	000110	000000	0110 (store)	000101	DataMem(R6)<- R5
0	000110	000110	0100 (load)	000000000000000	R6<- DataMem(R6)



This is our RTL analysis behavioral simulation for the processor.
We are receiving the correct outputs for each of the functions (0, 1, 2, 3, 4, 6, 7, 6, 6, 5). We successfully tested and synthesized each block design individually. Each individual testbench we simulated produced expected results; however, when we connected each block together, we observed in the post synthesis timing simulation that we are not outputting the correct values. We believe this is due to inefficient code, therefore the timing for one or more modules is slow.



This is our post-synthesis timing simulation.
The first instruction seems to be delayed. The last output before the NOP instruction is 0 instead of 5, which is incorrect.

Problems (and how we fixed them)

1. The behavioral simulation was not outputting correctly. This was because the WriteEnable flag in Controller.sv was improperly set. We had thought WriteEnable1 = 1 for only the LOAD function and WriteEnable2 = 1 for only the STORE function. We fixed this by setting WE1 = 1 for all functions excluding NOP, and STORE.
2. A small issue was in MuxSel2. In the specifications listed by the T.A, the processor needed to output the address of register 6 for the 9th instruction. Like all other instructions, it was outputting whatever was written into R6 (in this case 5). We fixed this by changing MuxSel2 = 1 for both LOAD and STORE functions to MuxSel2 =1 only for the LOAD function, so it will properly return address instead for STORE.
3. We are not getting proper outputs for both post-synthesis and post-implementation timing simulations, we suspect that it is because of an “inferred latch” warning / issue in the datamem.sv file. Additionally, we believe that some portions of our code is inefficient in terms of timing. For example, some assign statements in our code might cause additional delay.
4. When we generated the bitstream and programmed the code onto the board, we achieved expected results for all the operations except the last one, which was LOAD. We suspect this is a timing issue as mentioned before(#3).