CSCE 3301 Project

Milestone 3 Report

Department of Computer Science and Engineering, AUC

Done by:

Kareem Kassab	900182771
Mohammed Zaieda	900181473
Omar El Sayed	900183884

1. Introduction.

In this milestone, we transformed the single cycle implementation to a pipelined implementation. In addition, we also supported various things like the forwarding unit to handle data hazards and have started in the bonus implementation.

2. Development & Functionality

Load and Store: The load and store instructions were handled by adding 2 outputs in the ALU control unit. We used them as inputs in the data memory to check if it's a load byte, half-word, or a word. To support them we altered the control unit to allow for more incoming flags along more case statements to handle them in the ALU module. All 8 load and store instructions were supported and produced correct outputs as shown below in the screenshots.

JAL, JALR, AUIPC, LUI: For these instructions, we added a 4x1 mux that takes the following inputs: pc + 4, branch outcome, jalr, and jal. The selection line is a 2 bit register that is set in the control unit to handle the previous instruction with regards the pc output. There is another 4x1 mux before the register file with the following inputs: auipc, jal/r, final_out, and lui. Its selection line was also set in the control unit to handle which result it will write in the register file.

Ecall, Ebreak, Fence: Ecall instruction was supported by adding a one-bit wire that replaced the MemWrite bit in the PCs. This bit was passed through the pipeline and the decision was made in the execute stage to prevent not taking the results of previous instructions and taking results from proceeding instructions of ecall. This prevented the PC to move any forward and halted the simulation. For the fence and ebreak instructions, a module was created that takes the opcode of the intruction and spits out a flag bit that indicates whether this instruction is fence or ebreak or not. If the instruction was any of them, then this flag is set to 1. This flag bit is then used to be a selection line for a 2x1 mux that chooses between the normal instruction getting out from the memory and the nop instruction. If the instruction was ebreak or fence, then a nop instruction is passed to the IF_ID register instead of the normal instruction.

I-Format: The I- format instructions were heavily depending on the immediate generator given, and by changing to the second source register to an immediate. These were supported by adding cases in the control unit and handling them in the given ALU module with no extra changes needed. Note that most of the I-format instructions were initially R-format instruction with slight changes of source registers and handling them using the immediate generator and ALU.

R-Format: For the R-format instruction there were minimal modifications made to support the missing instructions that were not done in the Lab, so what we did is adding more case statements in the control unit to further accept more instructions. These instructions were also tested with provided screenshots as a proof of validity.

Branch: we made a module that takes flags, and [14:12] bits of the instruction. Using if statements we detect the type of the branch instruction. We assign the branch output = 1, when these conditions were satisfied. We also started implementing a branch outcome unit in the decode stage, which required another forwarding unit as explained in the forwarding unit section to handle any data hazards for moving it in the decode stage.

Shifter: The shifters were handled by following the same inputs and output sequence given in the ALU module. The logical operations were simply handled using the shifts operators in Verilog and assigning these to the output register when their conditions are satisfied.

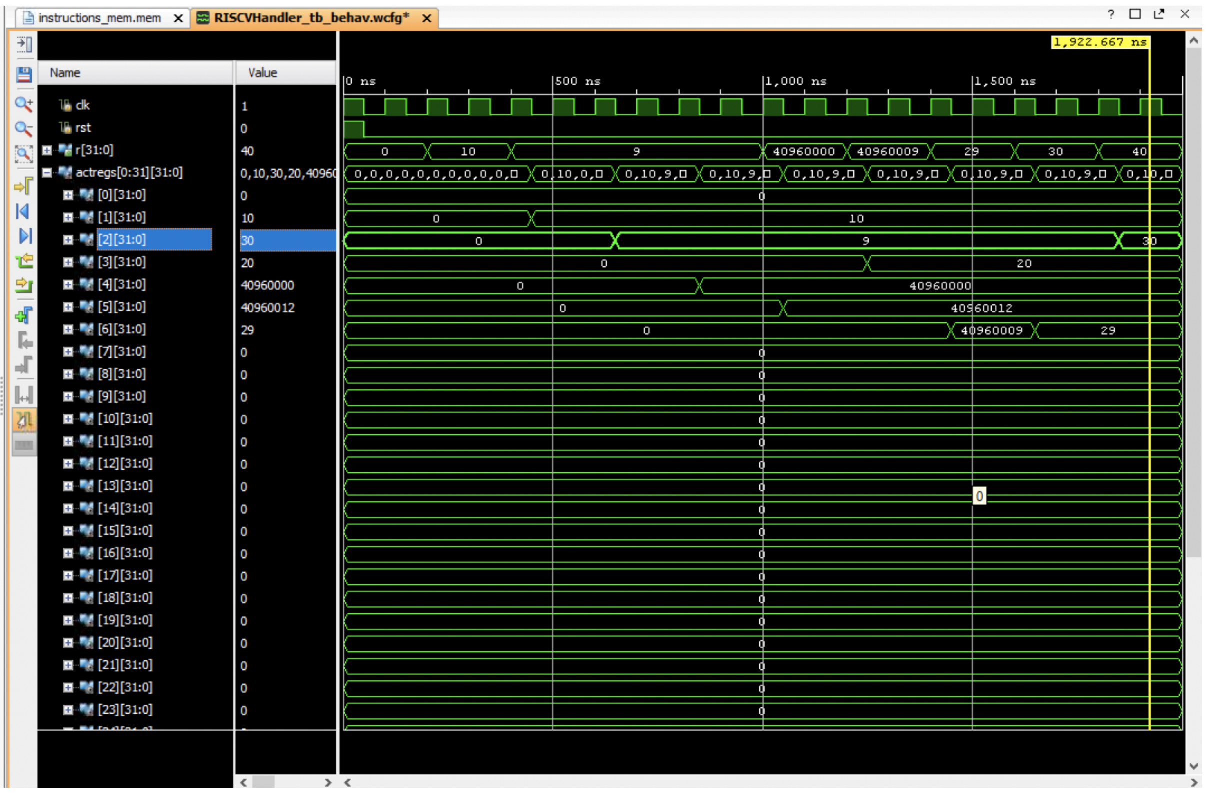
Forwarding Unit: The forwarding unit had zero change from what we have done in the lab course. But in a nutshell, the forwarding unit takes inputs from the Execute/Memory register and the Memory/Writeback register with source registers to check on the specific conditions of whether or not the instruction incoming requires forwarding from a the execute stage or from the memory stage. We added a 4x1 mux before the ALU for both inputs to satisfy the selection lines exiting from the forwarding unit.

Pipeline implementation: The pipeline implementation was very simple in theory; we used the same module of N-Bit register that takes N-bits which were altered whenever we added registers into the pipeline. These 5 registers helped us to executed instructions simultaneously and access register values whenever we wanted in whichever stage. This pipeline implementation was also done in the lab course.

Single Memory: The single memory was basically a combination of the previous instruction memory and data memory that we have implemented in the lab course and in the previous milestones. It basically read from a file of hexadecimal representations of instructions separated by spaces for each two digits. We had a base register which is basically the separation between the instruction part and data part of the memory.

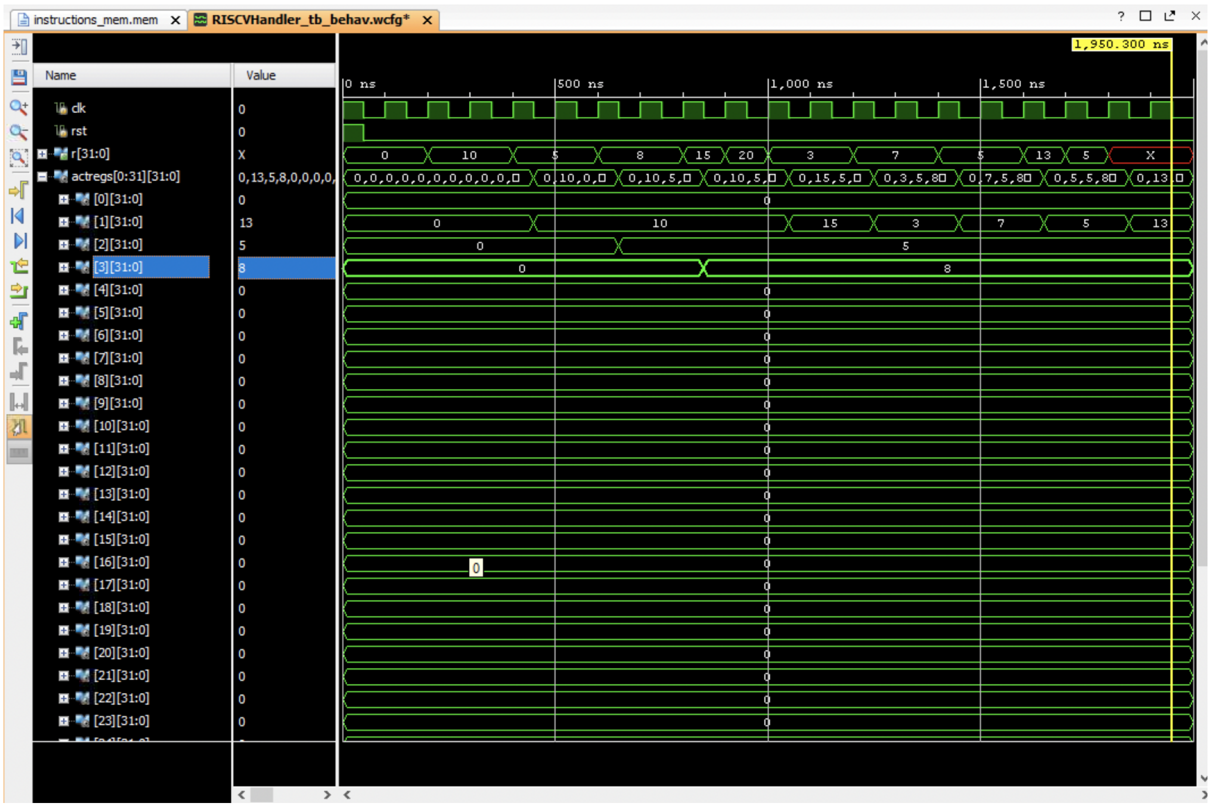
3. Testing:

This is lui-jal/r-auipc.txt file test case.



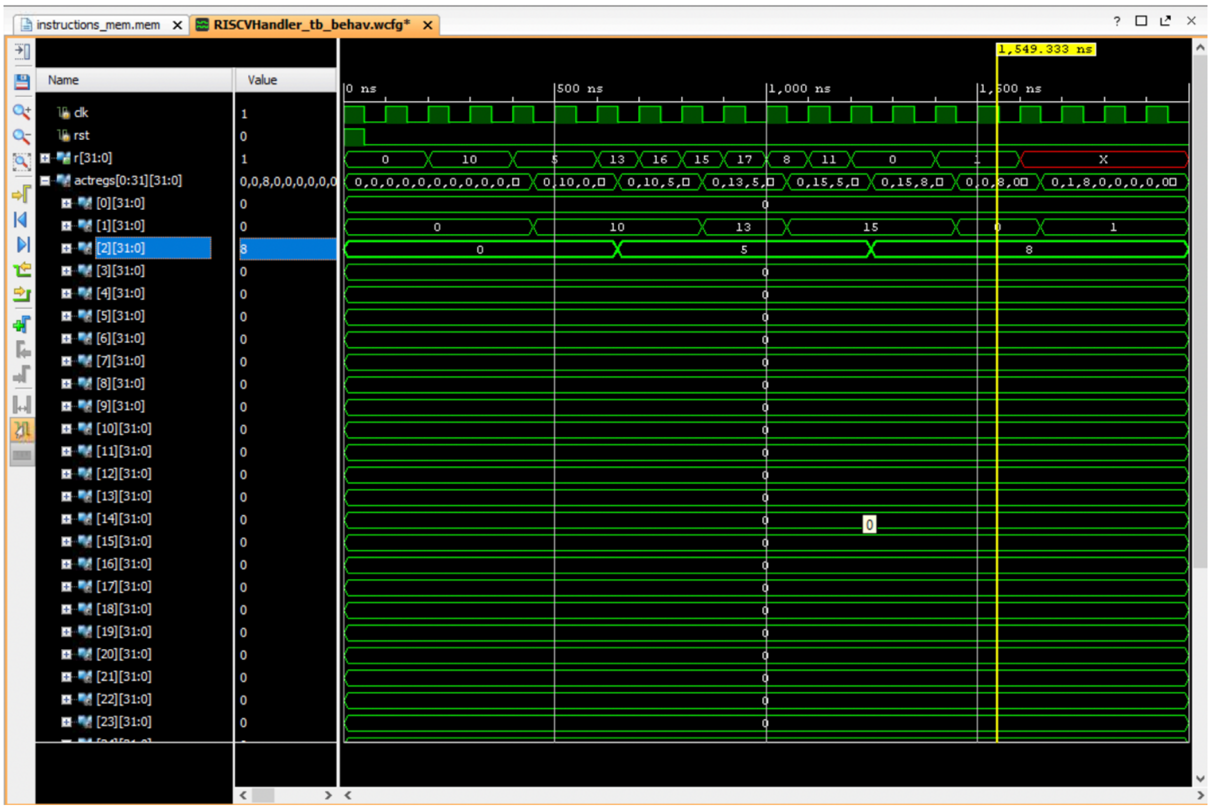
Name	Value
clk	1
rst	0
r[31:0]	40
actregs[0:31][31:0]	0,10,30,20,40960
[0][31:0]	0
[1][31:0]	10
[2][31:0]	30
[3][31:0]	20
[4][31:0]	40960000
[5][31:0]	40960012
[6][31:0]	29
[7][31:0]	0
[8][31:0]	0
[9][31:0]	0
[10][31:0]	0
[11][31:0]	0
[12][31:0]	0
[13][31:0]	0
[14][31:0]	0
[15][31:0]	0
[16][31:0]	0
[17][31:0]	0
[18][31:0]	0
[19][31:0]	0
[20][31:0]	0
[21][31:0]	0
[22][31:0]	0
[23][31:0]	0

This is R.txt file test case.

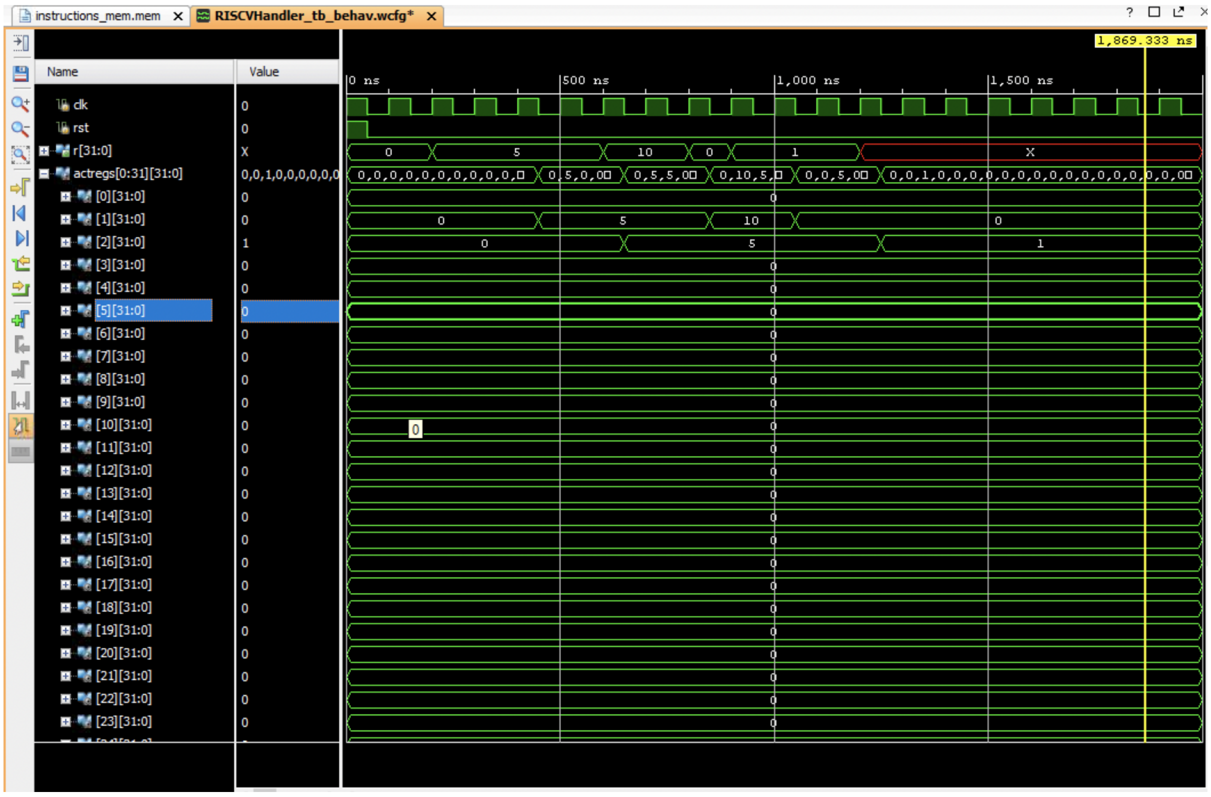


Name	Value
clk	0
rst	0
r[31:0]	X
actregs[0:31][31:0]	0,13,5,8,0,0,0,0,
[0][31:0]	0
[1][31:0]	13
[2][31:0]	5
[3][31:0]	8
[4][31:0]	0
[5][31:0]	0
[6][31:0]	0
[7][31:0]	0
[8][31:0]	0
[9][31:0]	0
[10][31:0]	0
[11][31:0]	0
[12][31:0]	0
[13][31:0]	0
[14][31:0]	0
[15][31:0]	0
[16][31:0]	0
[17][31:0]	0
[18][31:0]	0
[19][31:0]	0
[20][31:0]	0
[21][31:0]	0
[22][31:0]	0
[23][31:0]	0

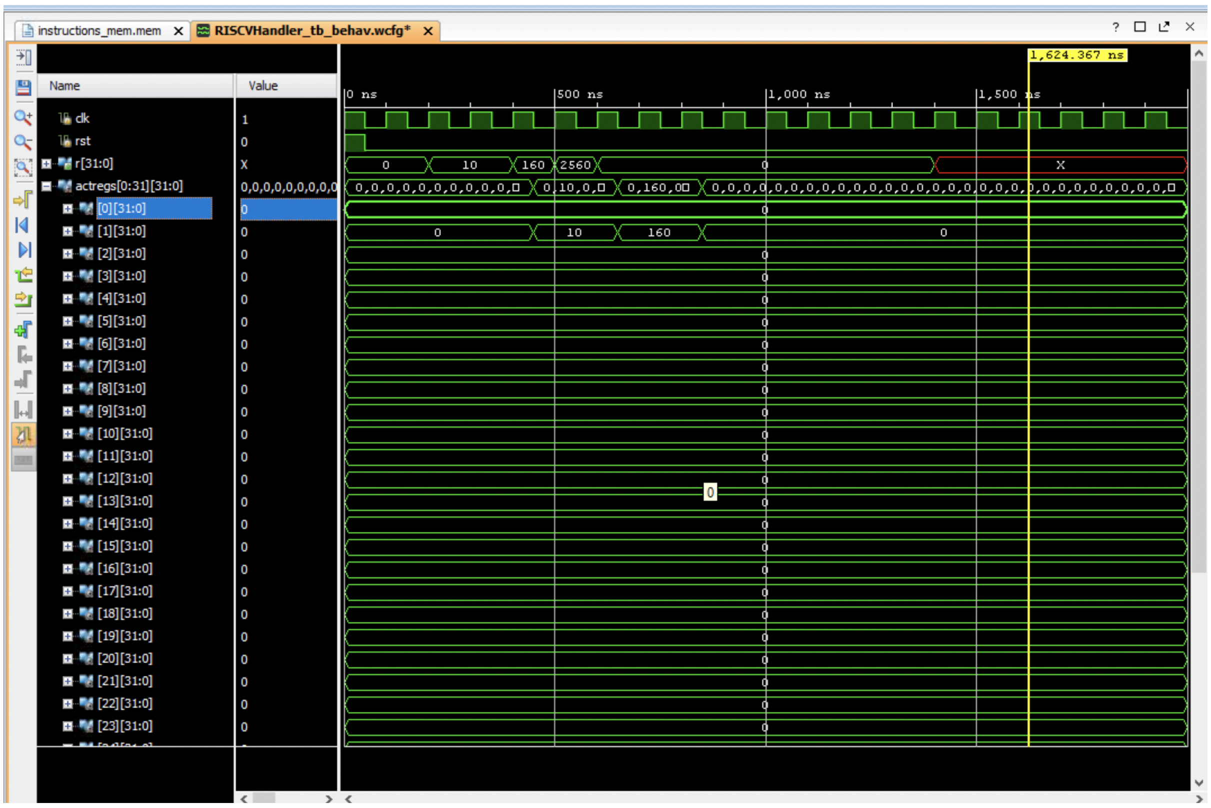
This is R-I.txt file test case.



This is sets.txt file test case.



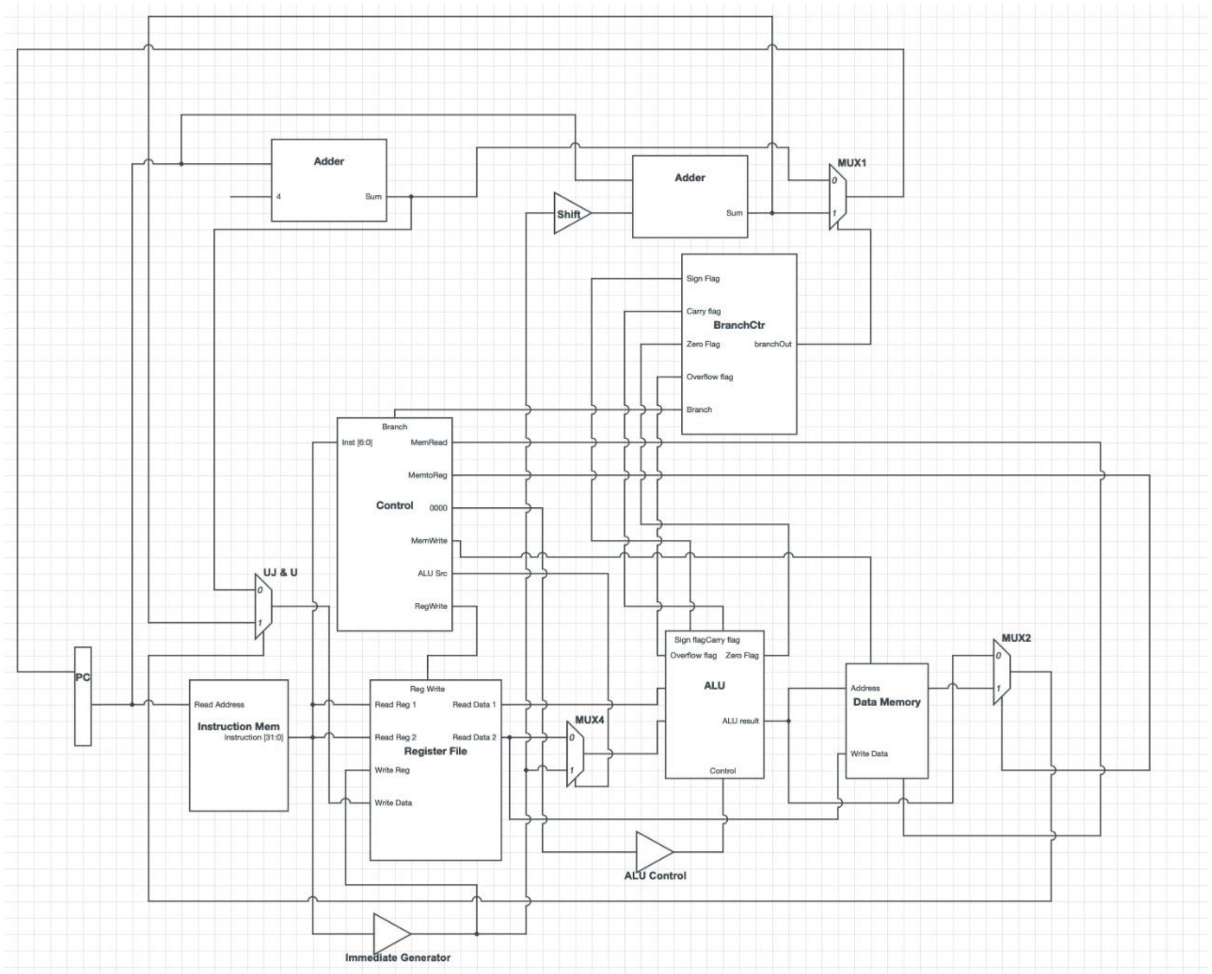
This is shift-I.txt file test case.



Name	Value
clk	1
rst	0
r[31:0]	X
actregs[0:31][31:0]	0,0,0,0,0,0,0,0,0
[0][31:0]	0
[1][31:0]	0
[2][31:0]	0
[3][31:0]	0
[4][31:0]	0
[5][31:0]	0
[6][31:0]	0
[7][31:0]	0
[8][31:0]	0
[9][31:0]	0
[10][31:0]	0
[11][31:0]	0
[12][31:0]	0
[13][31:0]	0
[14][31:0]	0
[15][31:0]	0
[16][31:0]	0
[17][31:0]	0
[18][31:0]	0
[19][31:0]	0
[20][31:0]	0
[21][31:0]	0
[22][31:0]	0
[23][31:0]	0

Other instructions and testcases will be done in MS4.

4. DataPath


Adder
4
Sum
Adder
Sum
Shift
MUX1
0
1
Sign Flag
Carry flag
BranchCtr
Zero Flag
branchOut
Overflow flag
Branch
Branch
Inst [6:0]
MemRead
MemtoReg
Control
0000
MemWrite
ALU Src
RegWrite
UJ & U
0
1
PC
Read Address
Instruction Mem
Instruction [31:0]
Reg Write
Read Reg 1
Read Data 1
Read Reg 2
Read Data 2
Register File
Write Reg
Write Data
MUX4
0
1
Sign flag
Carry flag
Overflow flag
Zero Flag
ALU
ALU result
Control
Address
Data Memory
Write Data
MUX2
0
1
ALU Control
Immediate Generator