Computer Architecture Project MS2 Report

Due:25th April 2020

Name: Omar Mahdy

ID: 900160493

Name: Mohamed Ashraf Taha

ID: 900172754

- **Project Goals & Design:**

In this submission we try and implement single cycle RISCV processor supporting All RV32I Base Instruction with CSR, Fence, and FenceI implemented as nops and Ecall implemented as a termination instruction ,Set as well as eight instructions from the RV32M Standard Extension (Multiplication & Division). Our main goal is for our implementation to work correctly on all our implemented instructions and for it to work on all combinations of inputs. This is still not achievable for this millstone as we are only required to test basic cases, but we plan in future milestones to test all possible combinations for all instruction to achieve this goal. Also, we aim to make our code readable as much as possible. We achieve this by simply trying to label all wires and reg appropriately especially in the top module titled Full_Datapath. You can see that most of the wires are named exactly the same like that one in the lectures slides and our block diagram or with similar naming making it easier for you to know which wire corresponds to that in the diagram and all Multiplexers are labeled as MUX's with a name to signify what they are for. Also, you will find that all outputs in the top module of any module will have the word "out" in them

to show that they are outputs for a certain module, except if they are used for a selection line for something then they will have the word “sel” in them. Moreover, for human readability we try and used the defined bits given in the define file. We do this to make our code less prone to human errors, as well as make it as much readable as possible. Lastly, we try and use more case statements than if and else statements to make the code easier to read and compile. Another thing we try to avoid doing is using the shorthand if’s instead of MUXS to make the code clearer. In addition, we try and implement we as little hardware cost as possible in our capabilities. For instance, you will find that if a single bit is not needed for a case or if statement, we will not take, this can be seen clearly in the ALU,the Data memory and ALU control modules.

- **Implementation**

In our implementation we added some extra hardware and logic to that which we took in the lecture slides and the lab to accommodate the new additional instructions. For instance, we add function 3 (instruction bits [14-12]) to the ALU module to accommodate the branch instructions and Multiplication & division instructions as well. The way we chose to implement the branch instructions is as follows, we created 4 different wires 1 bit called beq, bne, blt,bge, bltu and bgeu each corresponding to appropriate instruction. If function 3 that is inputted if equivalent to one of the instructions, we set the chosen instruction’s bit equal to 1 in the ALU.

Then if this bit is 1 and its corresponding flag is set appropriately then Flag_input_ANDgate is set to 1 which is then ANDed in the top module to with branch control sign to produce AND result which will be 1 only when branch is taken and 0 when its not. For the multiplication & division instruction will explain them in report of MS4 as that's when they are required. For the immediates instructions we added their opcode in the control unit and has all the control signals the same like that of the normal register arithmetic operations except the ALUsrc was set to 1 to take the second input for the ALU from the immediate generator instead of RD2. All arithmetic operations were given already in skeleton on the ALU except xor and few others which we added for their corresponding function 3 in ALU control and appropriate output in the ALU. For the shift instructions we created a shifter module and used the shift operators given to us in Verilog. And we made shift arithmetic signed since it has a signed bit. For the data memory we made out data memory little endian byte addressable by making each memory in our memory slot equal 8 bits. We Also sent function 3 to the data memory to distinguish what type of addressing we need with the appropriate function 3. For load instructions w set a register to either 8 bits lb, 15 bits lh, and we sign extend lbu by 24 bits and lhu by 16 and lw to 32 bits to be able to read the last bit as 1 in case it's negative. Same goes for store instructions we set the appropriate memory location to either sb or sw or s. We set our memory to little endian in the memory by storing the contents of the memory ascendingly and we store in 3 different memory locations the values 5,10 and -4 in that order. The CSR opcodes and Fence and FenceI opcodes were added to the control signal module with all

its control signal equal. Finally, we had to edit the original data path to support ecall,jal,jalr and AUIPC. For ecall we added wire called ecall_sel to be 1 if the opcode of the given instruction is equal to that of ecall and if function 3 is equal to 000 which is func3 of ecall. We then added mux which is 4 when ecall_sel is 0 and when it is 1 equal to 0. And PCplus4 is added to the output of this mux. For jal created a new control signal called jalr to be set to 1 when the opcode is that of jal or jalr. This input is added to a 4x1 mux as a selection line concatenated with a Auipc control signal when jal or jalr is the one chosen we store the address we are at in a register then we take the value of the immediate generator and shift left and jump to it. With jal r we have another mux that chooses if we return to the stored address or not if instruction is jalr. For auicpc we created also a custom control signal and has the same implementation as jal but with control sign auipc instead. The imm generator all the sign extensions were done except for load instructions which we added. We also, made aluop 3 bits instead of 2 because we ran out of combinations to use when adding the opcodes in the control unit.

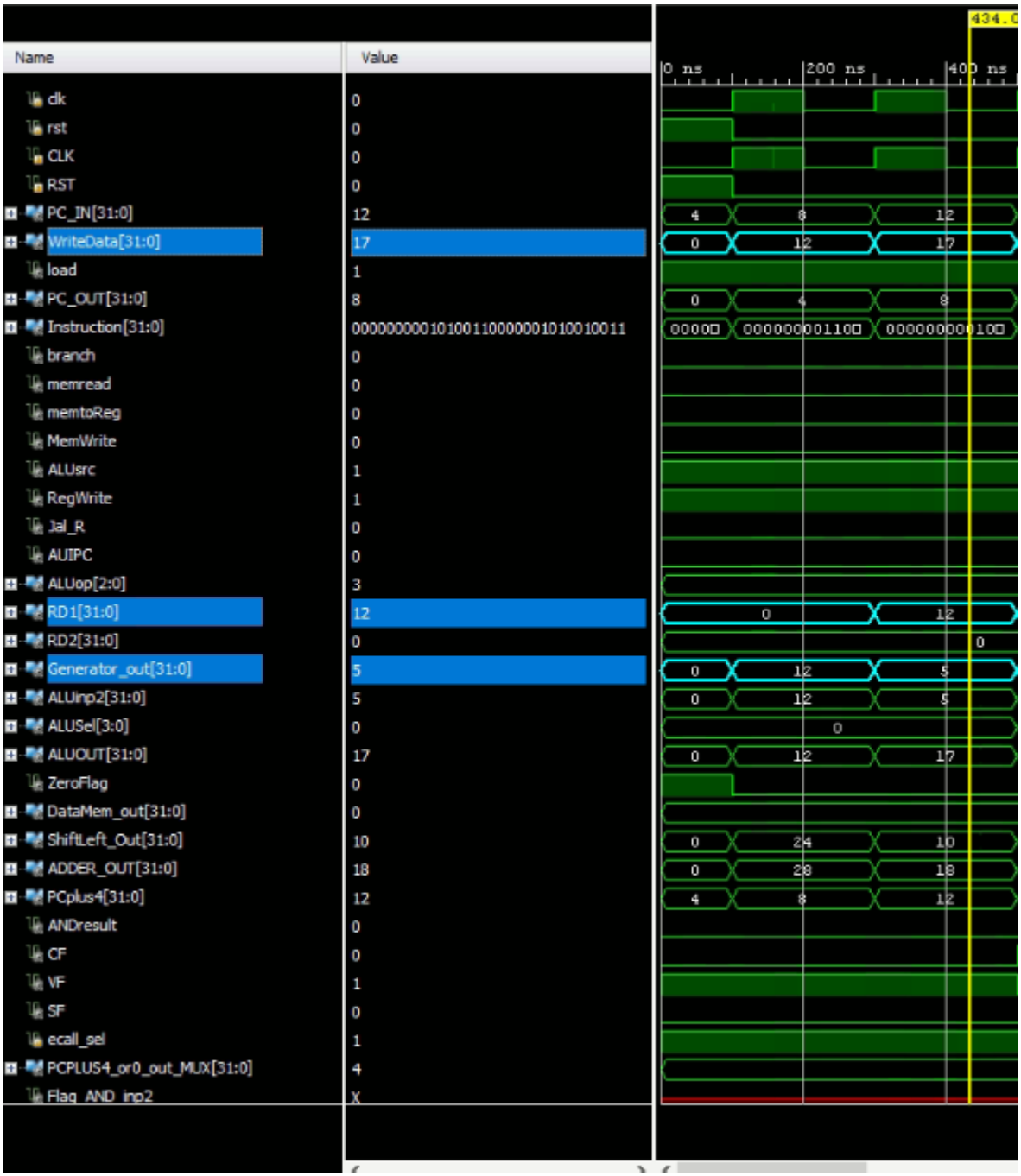
- **Testing Procedure and Results**

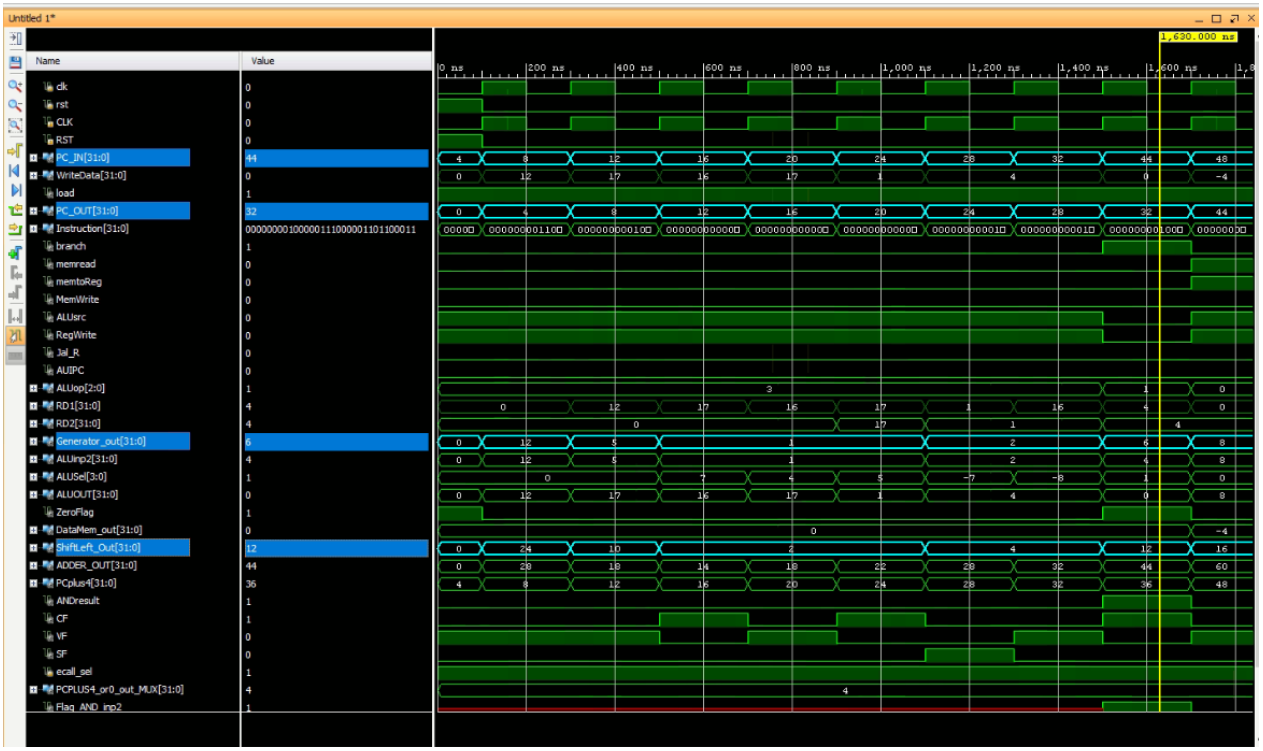
We tested by adding the machine code, the 32 bit instruction in the instruction memory and commented the equivalent assembly code next to it. We got the machine code from the RARS emulator we wrote the instruction we needed and rars gives us the instruction in hexa we then use an online converter to convert the hexa instruction to the equivalent 32-bit

instruction, as well as the expected output for a certain instruction. We tested and checked all 47 instructions at least once and checked their outputs and they we found to be correct for the cases we choose. Down below you will see the results of some instructions that we used we written the assembly code and the screen shot shows everything else.

Results.

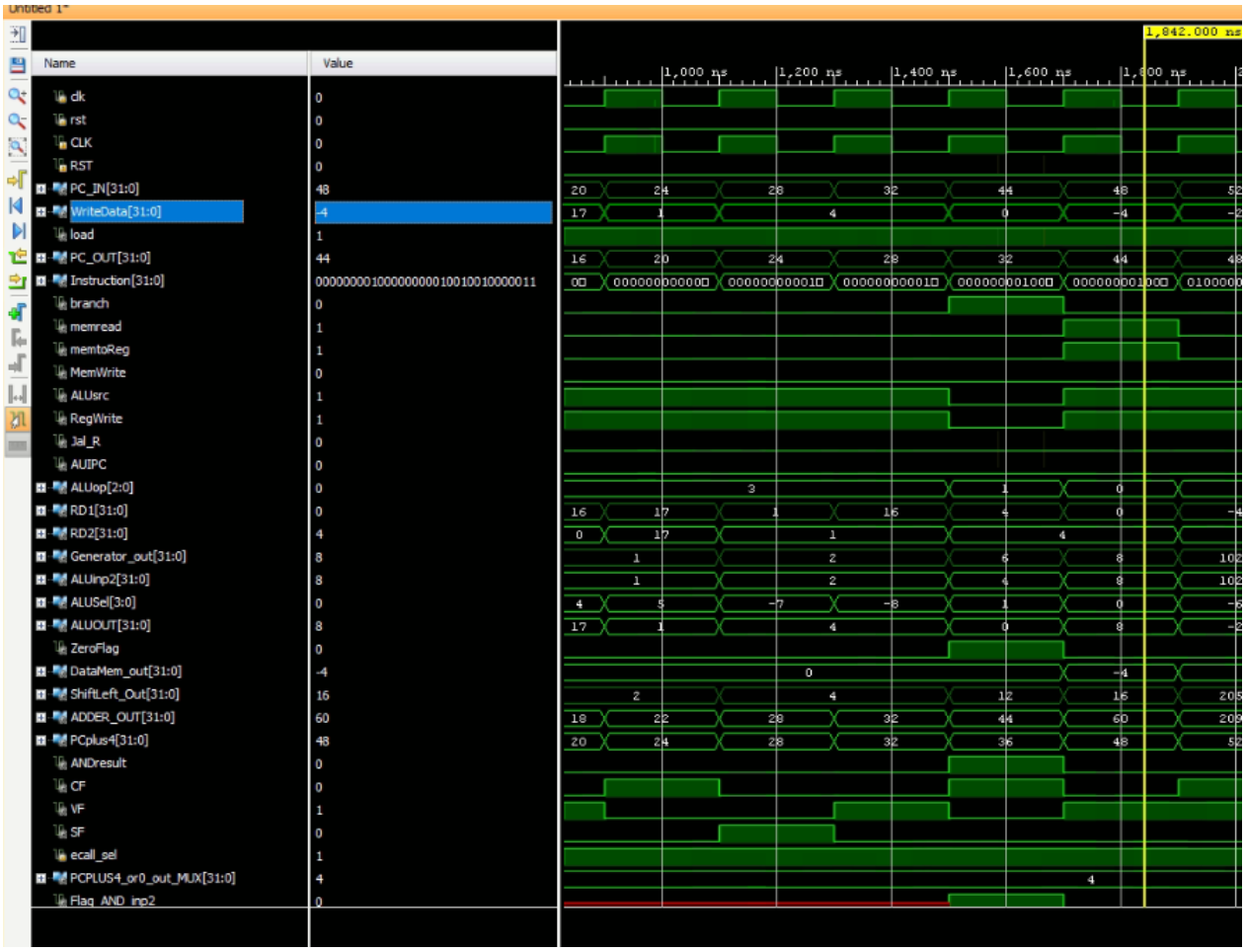
addi x5,x6,5



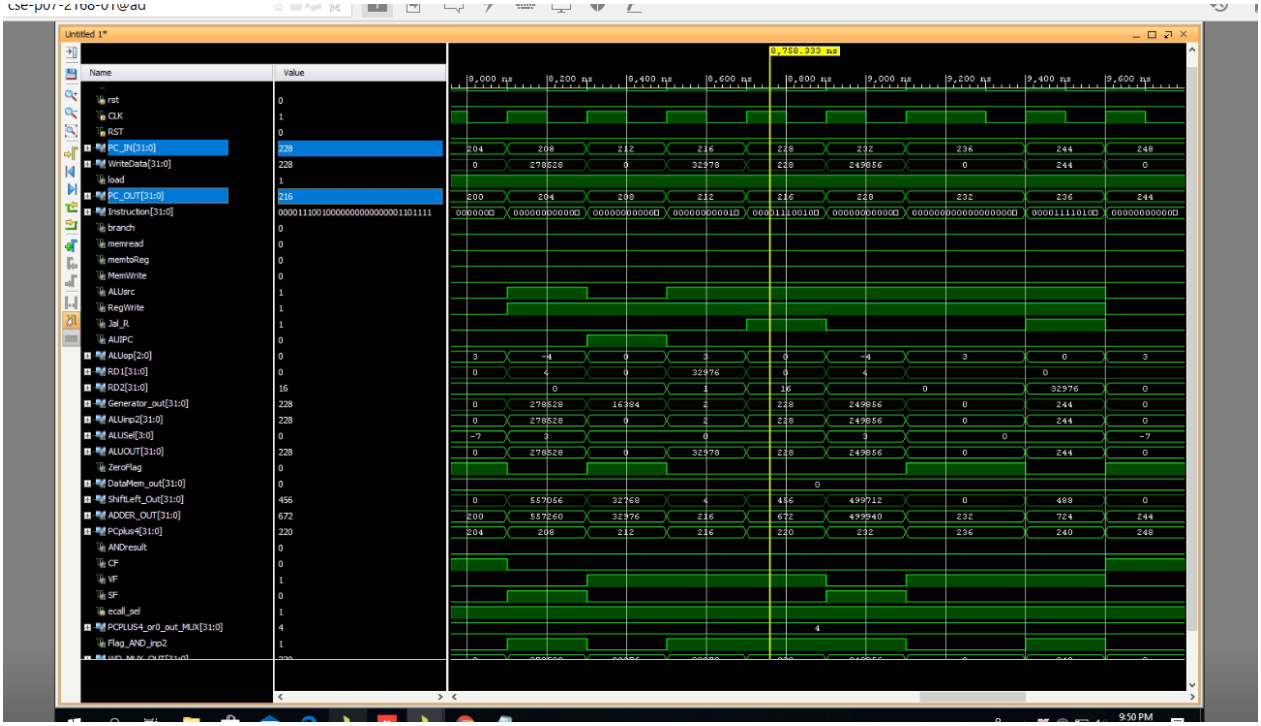

Untitled 1*
Name
Value
clk
rst
CLK
RST
PC_IN[31:0]
WriteData[31:0]
load
PC_OUT[31:0]
Instruction[31:0]
branch
memread
memtoReg
MemWrite
ALUsrc
RegWrite
Jal_R
AUIPC
ALUop[2:0]
RD1[31:0]
RD2[31:0]
Generator_out[31:0]
ALUinp2[31:0]
ALUSel[3:0]
ALUOUT[31:0]
ZeroFlag
DataMem_out[31:0]
ShiftLeft_Out[31:0]
ADDER_OUT[31:0]
PCplus4[31:0]
ANDresult
CF
VF
SF
ecall_sel
PCPLUS4_or0_out_MUX[31:0]
Flag_AND_inp2
1,630.000 ns

BEQ x7,x8,6

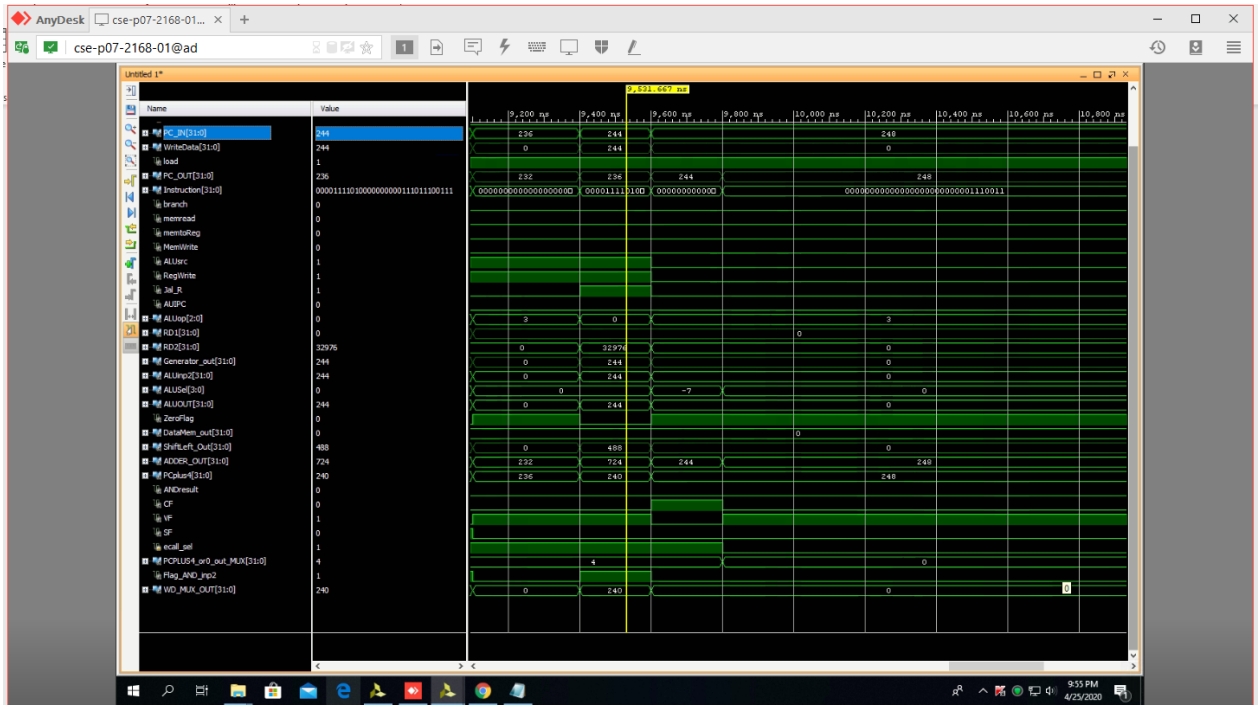
lw x9,8(x0))


Untitled 1*
Name
Value
clk 0
rst 0
CLK 0
RST 0
PC_IN[31:0] 48
WriteData[31:0] -4
load 1
PC_OUT[31:0] 44
Instruction[31:0] 00000000100000000010010010000011
branch 0
memread 1
memtoReg 1
MemWrite 0
ALUsrc 1
RegWrite 1
Jal_R 0
AUIPC 0
ALUop[2:0] 0
RD1[31:0] 0
RD2[31:0] 4
Generator_out[31:0] 8
ALUinp2[31:0] 8
ALUSel[3:0] 0
ALUOUT[31:0] 8
ZeroFlag 0
DataMem_out[31:0] -4
ShiftLeft_Out[31:0] 16
ADDER_OUT[31:0] 60
PCplus4[31:0] 48
ANDresult 0
CF 0
VF 1
SF 0
ecall_sel 1
PCPLUS4_or0_out_MUX[31:0] 4
Flag_AND_inp2 0
1,000 ns
1,200 ns
1,400 ns
1,600 ns
1,842.000 ns

jal x0,228


Untitled 1*
Name
Value
rst
CLK
RST
PC_IN[31:0]
WriteData[31:0]
load
PC_OUT[31:0]
Instruction[31:0]
branch
memread
memtoReg
MemWrite
ALUsrc
RegWrite
Jal_R
AUIPC
ALUop[2:0]
RD1[31:0]
RD2[31:0]
Generator_out[31:0]
ALUinp2[31:0]
ALUSel[3:0]
ALUOUT[31:0]
ZeroFlag
DataMem_out[31:0]
ShiftLeft_Out[31:0]
ADDER_OUT[31:0]
PCplus4[31:0]
ANDresult
CF
VF
SF
ecall_sel
PCPLUS4_or0_out_MUX[31:0]
Flag_AND_inp2

ecall


AnyDesk
cse-p07-2168-01...
cse-p07-2168-01@ad
Untitled 1*
Name
Value
PC_IN[31:0]
WriteData[31:0]
load
PC_OUT[31:0]
Instruction[31:0]
branch
memread
memtoReg
MemWrite
ALUsrc
RegWrite
Jal_R
AUIPC
ALUop[2:0]
RD1[31:0]
RD2[31:0]
Generator_out[31:0]
ALUinp2[31:0]
ALUSel[3:0]
ALUOUT[31:0]
ZeroFlag
DataMem_out[31:0]
ShiftLeft_Out[31:0]
ADDER_OUT[31:0]
PCplus4[31:0]
ANDresult
CF
VF
SF
ecall_sel
PCPLUS4_or0_out_MUX[31:0]
Flag_AND_inp2
WD_MUX_OUT[31:0]
9:55 PM
4/25/2020