CSCE3301 – Computer Architecture

Summer 2022

Project 1: RISC-V processor

Final Project Report

Project Overview:
RISC-V processor simulated using Xilinx Vivado or any other Verilog simulator

Implementation language:
Verilog

A Brief Description of Code Implementation

The RISC-V processor is implemented using Verilog and simulated using Vivado. In milestone 2 , the processor was implemented as a single-cycle processor. Firstly, we started creating all the modules included in our processor datapath. We created modules for ALUcontrol , ControlUnit , Data Memory , DFilpFlop , 32-bit Multiplexer 2x1 , 32-bit Multiplexer 4x1 , Branching Unit , Shifting Unit,Instruction Memory , Program Counter, Register File ,Full Adder and Ripple Carry adder. The Immediate generator ,ALU and DEFINES were provided module code as support project codes. Then finally we joined all these modules in our top module "FullDataPath". In this milestones 3 , 4 we implemented the RISC-V processor as a pipelined processor using the same modules above and we added memory, ForwadingUnit modules. Then Our new top module is "pipeline_Datapath" where we connect all our modules in and implement our IF_ID , ID_EX, EX_MEM, MEM_WB Registers. While Implemting the piplined processor we used a single memory module and deleted the 2 separate modules of instruction memory and data memory , instead we merged them both in the new module memory. The following sections will discuss and describe each module's implementation and function.

ALUcontrol Module

The ALUcontrol module consists of 3 inputs and 1 output. For the inputs we have 32-bit instruction , 2-bits ALUOP, and 1-bit immtype . We implemented a wire named "inst1" to take the func3 of the instruction, which is 3 bits from bit 12 to bit 14, and another wire named "inst2" which is assigned to be the 30th bit of the instruction to be used for instructions that have the same opcode and funct3 but differ in their 30th bit , for instance the instructions ADD and SUB . Then we created an always block that consists of if statements basing the conditions on the 2-bits ALUOp input , "inst1" , "inst2" and immtype . In the always block, the output is assigned to be the instruction defined in the"DEFINES" file provided according to each instruction's opcode , funct3(inst1), 30th bit (inst2) its immtype .

ControlUnit Module

In this module,we implemented the control unit by having the 32-bit instruction as an input . For the output , we created 10 signal outputs: memRead,memtoReg,memWrite , ALUSrc , Regwrite ,jal,immtype , 2-bits each, ALUOp, MUXSel, and Branch. In the module we created an always block that consists of a case statement on a wire called temp that is assigned to 5 bits from bit 2 to bit 6 of the 32-bit instruction input . Then we assign our 40 instruction opcodes of 9 bits to be the signals mentioned above . For instance (OPCODE_Arith_R : outputss = 9'b100010001;) here we assigned the opcode of all the R-format instructions to be 9-bits (100010001). Then finally, we assigned all the output signals to be equal to each bit accordingly to "outputss."

DFilpFlop

The DFilpFlop module is created to be used in the module of the program counter, it is simply implemented by having an always block that assigns the output based on the input reset.

32-bit Multiplexer 2x1

The multiplexer 2x1 module is implemented by having 2 inputs , 1 select line also as an input and an output. The module consists of a generate block that generates 32 multiplexer 2x1 that are 1 bit each .

32-bit Multiplexer 4x1

This module takes 4 inputs 32-bit each , 1 select line also as an input 2-bits and has one 32-bit output , we created an always block that has case statement based on the select line input to assign the output.For instance 2'b00:out=a; so here is the select line input 00 then the output is assigned to be equal to input a .

Branching Unit

We created this module to handle the branching. The module is called ”BU” in has input signals cf,zf,vf,sf , 3-bits input funct3 2-bit branchsignal input and 2-bits output.The module contains an always block that has a case statement based of the branchsignal input so if the branchsignal is equal to 2’b00 then the processor branches and then it goes to the second case statement that is based on funct3 that classifies the branching if it is BEQ , BNE , BLT,BGE,BLTU or BGEU. Also in this module we handle the JALr instruction by assign the it to be the branchsignal 2’b01

Shifting Unit

We created this module to handle the shifting .The module is called “Shifter”.It has input 32-bit a , 5-bits input “shamt” (provided in the ALU code) , 2-bits iput type and output r .This module consists of an always block that contains a for loop that loops “shamt” of case statements based on type input to assign the output.

Program Counter

The module of the program counter is named by “regg_32”. This module simply consists of a generate block that generates 32 DFilpFlops and 32 Multiplexers 2x1. It consists of inputs clk,load,reset 32-bit ,D and output 32-bit Q.

Register File

This module has inputs , clk,rst ,5-bits readreg1 , 5-bits readreg2,writereg,writedata ,regwrite and 32-bits outputs readdata1,readata2.It aslo contains a 32-bit wires named “outt” and “load”. There is an always block that always assigns the load to be 0 and assigns load of writereg to be 1 if regwrite signal is 1 and writereg is not 0 . The module aslo contains a generate block that generates 32 , 32-bit registers from the module”regg_32”.Then it assigns the outputs to be the out of registers with specific address for instance , assign readdata1 =

outt[readreg1]; here it assigns the output readdadat1 to be the output “outt” of address readreg1 .

Full Adder

The module consists of 3 inputs A,B and cin and 2 outputs carry and sum . Then we created 3 temp wires “T1,T2,T3” and assign each to a specific function . for instance (assign T2=A&B;) T2 is assigned the be the anding of inputs A ,B .For the carry it is assigned to be the oring of temp wires T2,T3 and for the sum it is assigned to be the XORing of the cin and temp wire T1

Ripple Carry adder

The ripple carry adder module consists of a generate loop that creates 32 fulladders from the module created “FullAdder” and it takes 32-bits A ,B and an inputs and Cin. For the outputs, it has output carry “C” and a 32-bit bit output Sum .

Memory Module

In this module, we created a parameter called “MemSize “ that is equal to 512 and also another parameter called “InstSize” which is equal to half of MemSize, and that is because we wanted to use the single memory and divide it into 2 sections so the first section is for the data memory and the second section is for instruction memory with size 256 for both sections,

Forwarding Unit module

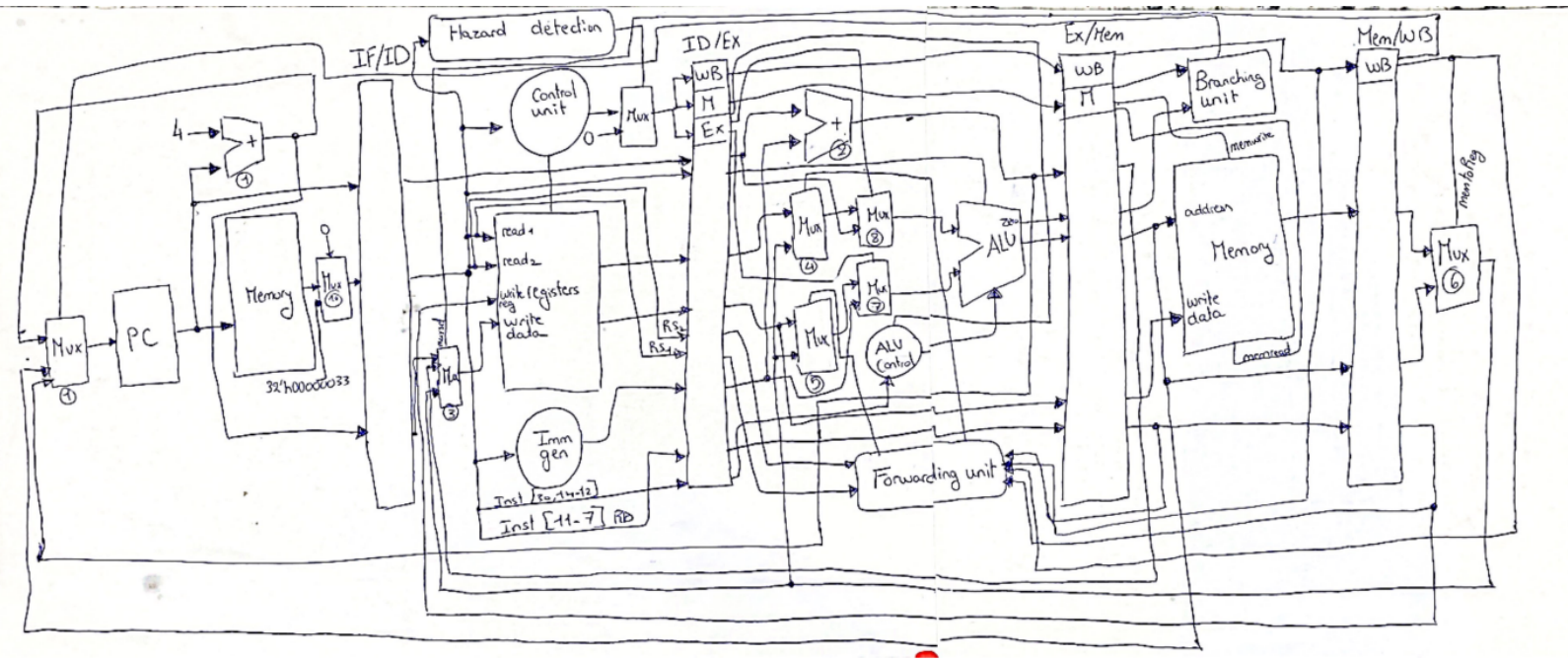
In this module, we created if statements for the forwarding process to happen.The module has outputs out, out2 and take input regwrite and 5-bits inputs rd , rs1 ,rs2 . In an Always block we assigned out and out2 to be zero and we created and if statement for each output to be 1 if the if statements conditions are true, For example , if (regwrite && (rd != 0) && (rd == rs1)) out =1;.

Pipeline_Datapath Top Module

This is the top Module that we created to join all our other modules together to create the Pipelined RISV-C processor. This module takes input clk and reset only. It contains an instantiation of all modules listed and discussed above in addition of IF_ID , ID_EX, EX_MEM, MEM_WB registers. Also it contains wires to connect all these module's inputs and outputs together, connection is done by according to our datapath that is shown below

Note : we also handled the hazards in the top module “pipeline_Datapath” by creating an if statements and conditions , that checks the branching unit outputs , if there is data dependency then stages need to stop so signals are produced to get to muxes responsible for data_memory. A hazard singal is produced to be given as an input for other modules .

Full Pipelined Datapath Schematic



In our DataPath sketch, we included two separate memories for simplification to separate the instruction and the data memory, However in our code implementation its only one module “memory" that includes both.

Design Decisions and Assumptions

Our Pipelined processor implementation uses a single memory for both data and instructions. The memory is single-ported and byte-addressable.

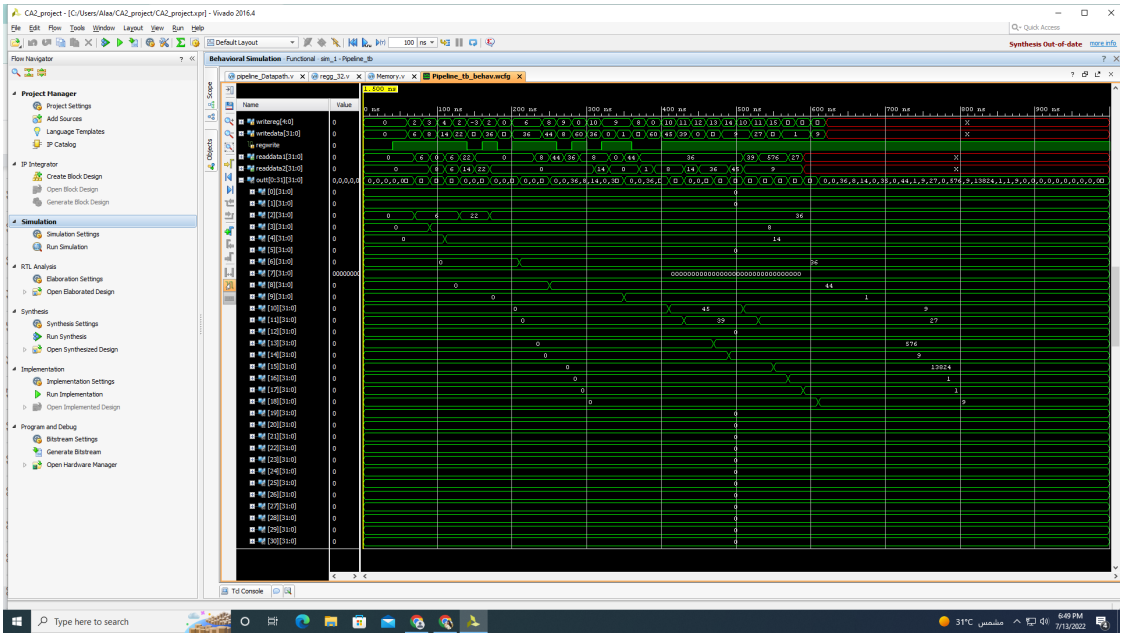
Bonus Task Requirement

Our implementation supports integer multiplication and division to effectively support the full RV32IM instruction set. Specifically supporting RV32M Standard Extension , MUL ,MULH ,MULHSU ,MULHU ,DIV , DIVU ,REM and REMU Instructions .

Test Cases and Simulation Screenshot

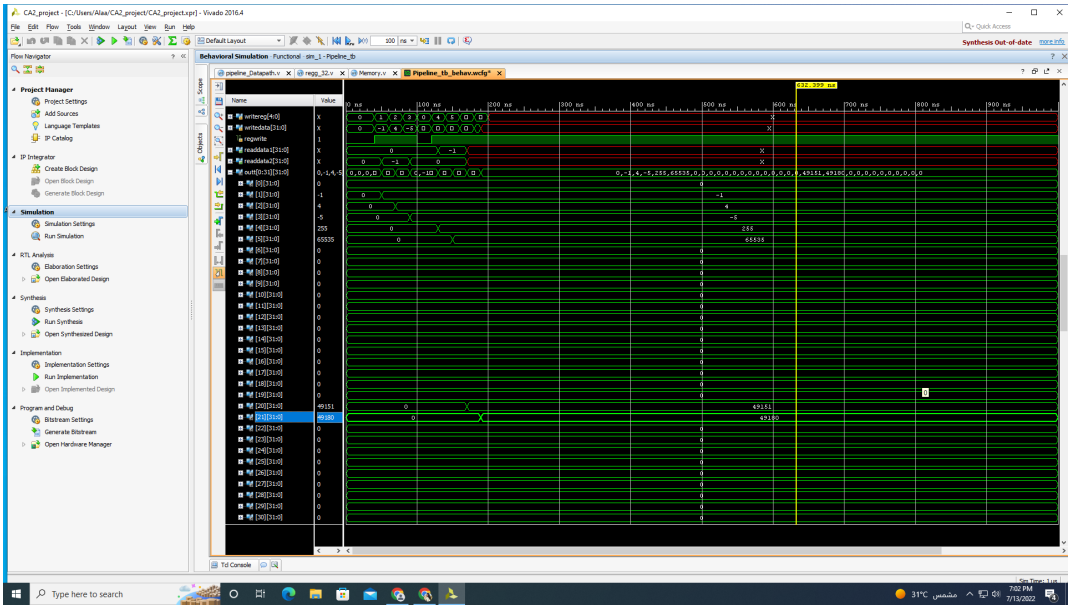
Test case 1)

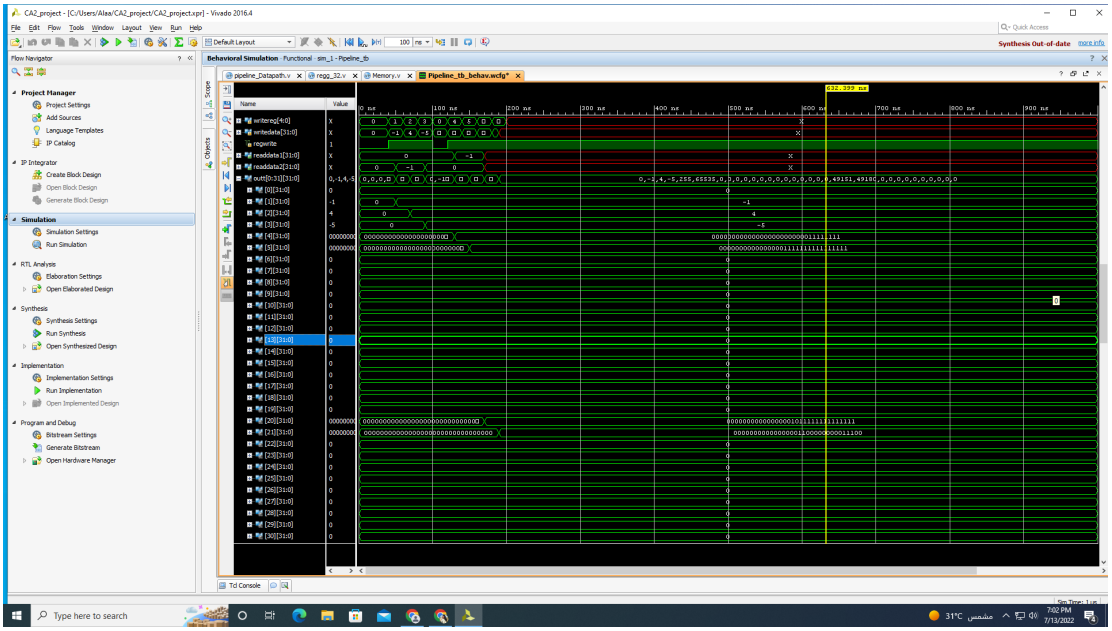
```
add x0, x0, x0                  //  x0= 0
addi x2, x0, 6                  //x2= 6
addi x3, x0, 8                  // x3 = 8
or x4, x2, x3                   // x4 =14
L:add x2, x4, x3                //x2 = 22
bne x2, x2, L                   // will not be taken
add x2, x2, x4                  //x2 = 36
sw x2, 0(x5)                    // x2 = 36
lw x6, 0(x5)                    // x6 =36
addi x6, x6, 0                  //x6 = 36
jal x8, L2                      // will jump to L2 Jalr
slti x9, x3, 1                  //0
slt x9, x3, x4                  // X9 =1
beq x0, x0, L3                  // WILL BRANCH , will got to L3
L2: jalr x0, 0(x8)              // X8 = 44
L3: xori x10, x6, 9             // x10 = 45
ori x11, x6, 3                  // x11 = 39
andi x12, x6, 3                 //x12 =0
slli x13, x6, 4                 // x13 = 576
srai x14, x6, 2                 //x14=9
srli x10, x6, 2                 //x10 = 9
sub x11, x6, x10                //x11 = 27
sll x15, x11, x10               // x15 = 13824
srl x16, x13, x10               // x16 = 1
sra x17, x13, x10               //x17 =1
and x18, x11, x10               // x18 = 9
#ebreak
```



Test case 2)

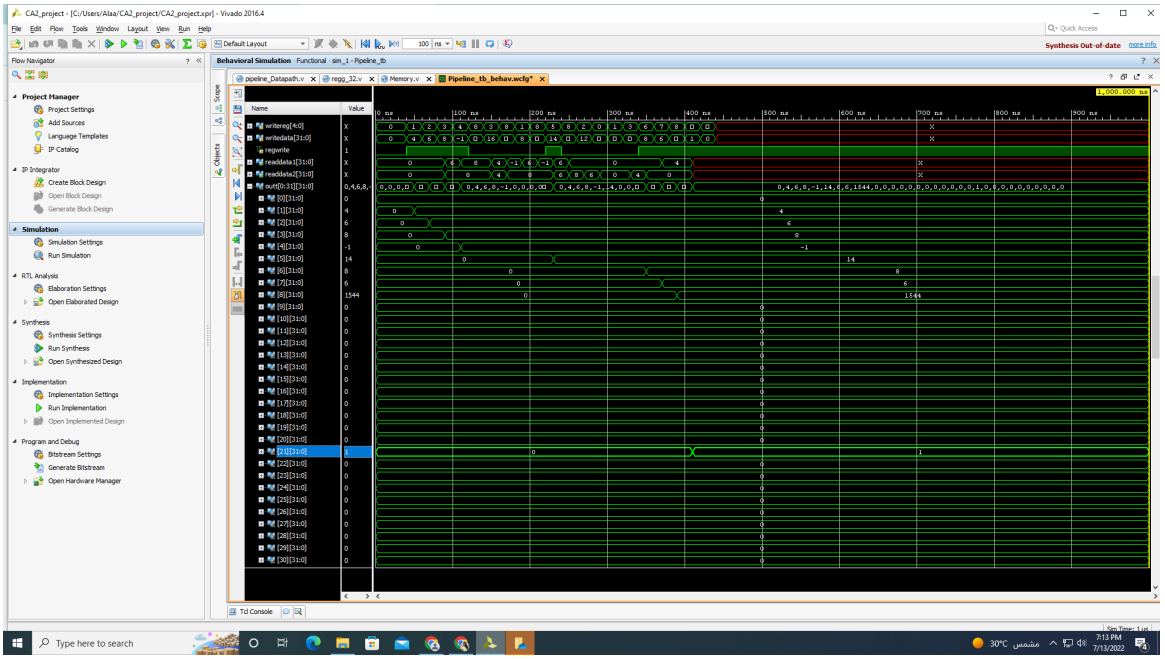
```
addi x1, x0, -1        // x1=-1
addi x2, x0, 4         //x2=4
xor x3, x2, x1         //x3=-5
sw x1, 0(x0)
lbu x4, 0(x0)          //x4 = 255
lhu x5, 0(x0)          // x5 = 65535
lui x20, 12            //x20 =49151
auipc x21, 12          //x21 =49180
#ebreak
```





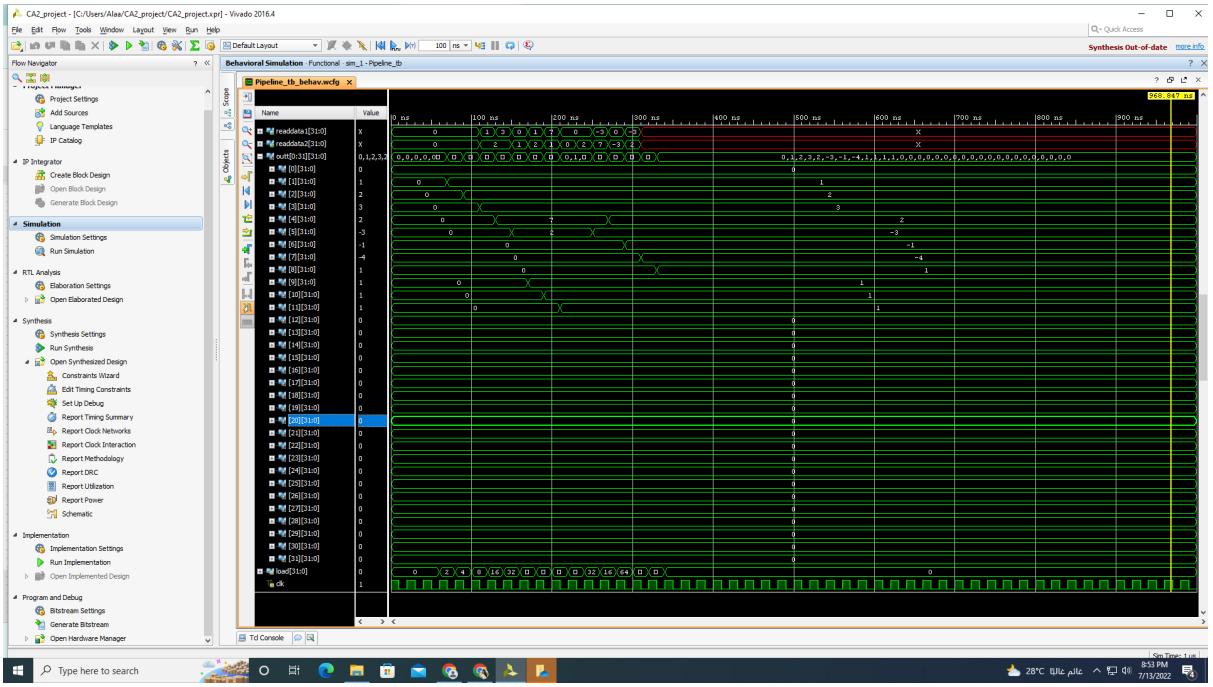
Test case 3)

```
#00000073 ecall
addi x1, x0, 4          //x1=4
addi x2, x0, 6          //x2=6
addi x3, x0, 8          //x3=8
addi x4, x0, -1         //x4=-1
blt x2, x3, L1          // WILL BRANCH , go to L1 bge
add x3, x3, x3          #skipped
L1: bge x3, x3, L2      // WILL BRANCH ,go to L2 bltu
add x1, x1, x1          #skipped
L2: bltu x4, x3, L3     //WILL NOT BRANCH
add x5, x2, x3          //x5=14
L3:bgeu x4, x3, L4      //WILL BRANCH , go to L4 sb
add x2, x2, x2
L4:sb x3, 0(x0)
sh x2, 1(x0)
sb x0, 3(x0)
lb x6, 0(x0)            //x6=8
lh x7, 1(x0)            //x7 = 6
lw x8, 0(x0)            // x8=1544
sltiu x21, x1, -1       //x21=1
sltu x22, x1, x20       //x22=0
#00100073 ebreak
```



Test case 4)

```
addi x0,x0,0            // x0=0
addi x1,x0,1             // x1=1
addi x2,x0,2             //  x2=2
addi x3,x0,3            // x3=3
addi x4 , x0 , 7          //x4=7
mul x5 , x1 ,x2          //x5=2
div x9 , x3 ,x2           //x9=1
divu x10  , x9,x1        //x10=1
rem x11 , x1,x2          //x11=1
remu x12 , x4 ,x1       //x12=0
addi x5 , x0 , -3        //x5=-3
addi x4 ,x0 , 2          //x4=2
mulh x6 , x5 ,x4          //x6=-1
mulhsu x7 ,x6, x5       //x7=-1
mulhu x8 , x5, x4       //x8=1
```


CA2_project - [C:/Users/Alaa/CA2_project/CA2_project.xpr] - Vivado 2016.4
Pipeline_tb_behav.wcfg
Behavioral Simulation - Functional sim_1 - Pipeline_tb
Type here to search