

UNIVERSITY OF WATERLOO AQ  
Faculty of Engineering   
      Department of Electrical and Computer Engineering         
ECE 621- Computer Organization

**Register File and Execute Stage**

Group 2

**Jordan *Ross,* Mustafa Faraj**   
j25ross, mabdulaz 31 October, 2014

# ALU Module Code

This section contains the code for our ALU module. It takes in the two source operands as well as an operation control signal (set in the decode stage). It outputs the result of the arithmetic operation as well as set two condition bits (zero and negative).

**module** alu**(**

**input** **[**31**:**0**]** op1**,** // operand 1 (always from rs)

**input** **[**31**:**0**]** op2**,** // operand 2

**input** **[**5**:**0**]** operation**,** // The arithmatic operation to perform

**input** **[**5**:**0**]** shift\_amount**,** // The number of bits to shift

**output** **reg** **[**31**:**0**]** result**,** // The arithmatic result based on the operation

**output** **reg** zero**,** // Indicates if the result of the operation is zero.

**output** **reg** neg // Indicates if the result is negative or not.

**);**

// The operations that the ALU supports are:

// 0: Addition TODO: Is there a difference here with unisgned vs. signed?

// 1: Subtraction

// 2: Multiply

// 3: Divide

// 4: Shift logical left

// 5: Shift logical right

// 6: Set on less than

// 7: And

// 8: Or

// 9: Xor

// 10: Nor

// 11: SRA

// 12: Load upper

**always** **@(**op1 **or** op2 **or** operation **or** shift\_amount**)** **begin**

**case** **(**operation**)**

0**:** result **=** op1**+**op2**;**

1**:** result **=** op1**-**op2**;**

2**:** result **=** op1**\***op2**;**

3**:** **begin**

**if** **(**op2 **!=** 0**)** **begin**

result **=** op1**/**op2**;**

**end**

// TODO: Otherwise we should trap here

**end**

4**:** result **=** op2 **<<** shift\_amount**;**

5**:** result **=** op2 **>>** shift\_amount**;**

6**:** result **=** op1 **<** op2 **?** 1 **:** 0**;**

7**:** result **=** op1 **&** op2**;**

8**:** result **=** op1 **|** op2**;**

9**:** result **=** op1 **^** op2**;**

10**:** result **=** **~(**op1 **|** op2**);**

11**:** result **=** op2 **>>>** shift\_amount**;**

12**:** result **=** **(**op2 **<<** 16**)&**32'hffff0000**;**

**endcase**

zero **=** result **==** 0 **?** 1 **:** 0**;**

neg **=** result**[**31**];**

**end**

**endmodule**

# Branch Resolve Unit

This section contains our logic for selecting whether a branch should be taken or not based on the two condition bits set by the ALU.

**module** branch\_resolve**(**

**input** zero**,**

**input** neg**,**

**input** **[**1**:**0**]**branch\_type**,**

**input** is\_branch**,**

**output** **reg** branch\_taken // Indicates if the branch is taken or not

**);**

**always** **@(**zero **or** neg **or** branch\_type **or** is\_branch**)** **begin**

**if** **(**is\_branch **==** 1'b1**)** **begin**

**case** **(**branch\_type**)**

0**:** branch\_taken **=** zero **==** 0 **?** 1 **:** 0**;** // BEQ

1**:** branch\_taken **=** zero **!=** 0 **?** 1 **:** 0**;** // BNE

2**:** branch\_taken **=** **(**neg **|** zero**)** **==** 0 **?** 1 **:** 0**;** // BLEZ

3**:** branch\_taken **=** neg **!=** 0 **?** 1 **:** 0**;** // BGTZ

**endcase**

**end** **else** **begin**

branch\_taken **=** 0**;**

**end**

**end**

**endmodule**

# Processor Module Code

This section contains the code for the processor module. The processor module is our top level module where we instantiate our fetch, decode, memory, register file, and ALU so far. This is also where we have our control signals being set. The control is only being done in the decode stage and the control signals that are needed in later stages are latched in pipeline registers. The processor module currently has the IF/ID and ID/IX pipeline registers implemented.

**module** processor**(**

**input** clk**,** // The system clock

**input** srec\_parse // If the SREC parser is active or not.

**);**

// Decoder signals

**wire** **[**4**:**0**]**rs**;**

**wire** **[**4**:**0**]**rt**;**

**wire** **[**4**:**0**]**rd**;**

**wire** **[**4**:**0**]**sha**;**

**wire** **[**5**:**0**]**func**;**

**wire** **[**15**:**0**]**immed**;**

**wire** **[**25**:**0**]**target**;**

**wire** **[**5**:**0**]**opcode**;**

**wire** **[**31**:**0**]**pc\_out**;**

**wire** **[**31**:**0**]**insn\_out**;**

// Control signals

**reg** stall**;**

**wire** **[**1**:**0**]**insn\_access\_size**;**

**wire** **[**1**:**0**]**fetch\_access\_size**;**

**wire** fetch\_rw**;**

**wire** insn\_rw**;**

**reg** reg\_file\_write\_enable**;**

// Address lines

**wire** **[**31**:**0**]**pc**;**

// Data lines

**wire** **[**31**:**0**]**insn\_data\_out**;**

**wire** **[**31**:**0**]**insn\_address**;**

// SREC registers (only used for helping the parser write to instruction memory)

**reg** **[**31**:**0**]**srec\_address**;**

**reg** **[**31**:**0**]**srec\_data\_in**;**

**reg** srec\_rw**;**

**reg** **[**1**:**0**]**srec\_access\_size**;**

// Decode wires

**wire** **[**31**:**0**]**decode\_pc**;** // We define this as the PC for next instruction to be executed

**wire** **[**31**:**0**]**decode\_ir**;** // We define this as the current instruction being decoded

**wire** **[**31**:**0**]**dec\_A**;**

**wire** **[**31**:**0**]**dec\_B**;**

**wire** **[**4**:**0**]**dest\_reg**;**

// Decode control signals

**reg** dest\_reg\_sel**;**

**reg** dec\_illegal\_insn**;**

**reg** **[**5**:**0**]** dec\_alu\_op**;**

**reg** dec\_is\_branch**;**

**reg** dec\_op2\_sel**;**

**reg** **[**5**:**0**]** dec\_shift\_amount**;**

**reg** **[**1**:**0**]**dec\_branch\_type**;** // 0-BEQ, 1-BNE, 2-BLEZ, 3-BGTZ

**reg** dec\_is\_jump**;**

// Execute wires

**wire** **[**31**:**0**]**exe\_pc**;** // We define this as the PC for next instruction to be executed

**wire** **[**31**:**0**]**exe\_ir**;** // We define this as the current instruction being executied

**wire** **[**31**:**0**]**exe\_A**;**

**wire** **[**31**:**0**]**exe\_B**;**

**wire** **[**31**:**0**]**exe\_op2**;**

**wire** **[**31**:**0**]**exe\_O**;**

**wire** **[**31**:**0**]**exe\_extended**;** // The wire the comes for the sign extender

**wire** exe\_zero**;**

**wire** **[**5**:**0**]** exe\_alu\_op**;**

**wire** exe\_is\_branch**;**

**wire** exe\_op2\_sel**;**

**wire** **[**5**:**0**]** exe\_shift\_amount**;**

**wire** **[**31**:**0**]** exe\_shift\_immed**;**

**wire** **[**31**:**0**]** exe\_shift\_target**;**

**wire** **[**31**:**0**]** exe\_jump\_effective\_address**;**

**wire** **[**31**:**0**]** exe\_branch\_effective\_address**;**

**wire** exe\_neg**;**

**wire** **[**1**:**0**]**exe\_branch\_type**;**

**wire** exe\_branch\_taken**;**

**wire** **[**31**:**0**]**exe\_next\_pc**;**

**wire** **[**31**:**0**]**exe\_branch\_next\_pc**;**

**wire** exe\_is\_jump**;**

// Execute control signals

// Instantiate mux's for each of the SREC registers to aid the SREC parser.

mux\_2\_1\_32\_bit srec\_insn\_address\_mux**(**

**.**line0**(**pc**),**

**.**line1**(**srec\_address**),**

**.**select**(**srec\_parse**),**

**.**output\_line**(**insn\_address**)**

**);**

mux\_2\_1\_1\_bit srec\_insn\_rw\_mux**(**

**.**line0**(**fetch\_rw**),**

**.**line1**(**srec\_rw**),**

**.**select**(**srec\_parse**),**

**.**output\_line**(**insn\_rw**)**

**);**

mux\_2\_1\_2\_bit srec\_insn\_access\_size\_mux**(**

**.**line0**(**fetch\_access\_size**),**

**.**line1**(**srec\_access\_size**),**

**.**select**(**srec\_parse**),**

**.**output\_line**(**insn\_access\_size**)**

**);**

// Instantiate the fetch module

fetch fetch**(**

**.**clk\_in**(**clk**),**

**.**stall\_in**(**stall**),**

**.**pc\_out**(**pc**),**

**.**rw\_out**(**fetch\_rw**),**

**.**access\_size\_out**(**fetch\_access\_size**)**

**);**

// Instantiate the instruction memory module

memory insn\_memory**(**

**.**data\_out**(**insn\_data\_out**),**

**.**address**(**insn\_address**),**

**.**data\_in**(**srec\_data\_in**),** // We can tie the srec\_data\_in wire to this port since we should never be writing to instruction memory unless we are srec parsing

**.**write**(**insn\_rw**),**

**.**clk**(**clk**),**

**.**access\_size**(**insn\_access\_size**)**

**);**

// Instatiate the IF/ID pipeline register to kep the PC and IR

if\_id\_pipleline\_reg if\_id\_pipleline\_reg**(**

**.**clk**(**clk**),**

**.**pc\_in**(**pc**),**

**.**ir\_in**(**insn\_data\_out**),**

**.**pc\_out**(**decode\_pc**),**

**.**ir\_out**(**decode\_ir**)**

**);**

// Instantiate a mux for selecting which destination to choose

mux\_2\_1\_5\_bit dest\_reg\_mux**(**

**.**line0**(**rt**),**

**.**line1**(**rd**),**

**.**select**(**dest\_reg\_sel**),** // TODO: Implement this control singal

**.**output\_line**(**dest\_reg**)**

**);**

// Instantiate the register file

reg\_file reg\_file**(**

**.**clk**(**clk**),**

**.**write\_enable**(**reg\_file\_write\_enable**),**

**.**source1**(**rs**),**

**.**source2**(**rt**),**

**.**dest**(**dest\_reg**),**

**.**destVal**(**exe\_O**),** // TODO: Change this to be the output from WB stage

**.**s1val**(**dec\_A**),**

**.**s2val**(**dec\_B**)**

**);**

// Instantiate the decode module

decode decoder**(**

**.**clk**(**clk**),**

**.**stall**(**stall**),**

**.**insn\_in**(**decode\_ir**),**

**.**pc\_in**(**pc**),** // TODO: I don't see what this is needed

**.**rs**(**rs**),**

**.**rt**(**rt**),**

**.**rd**(**rd**),**

**.**sha**(**sha**),**

**.**func**(**func**),**

**.**immed**(**immed**),**

**.**target**(**target**),**

**.**opcode**(**opcode**),**

**.**pc\_out**(**pc\_out**),** // TODO: I don't see why this is needed

**.**insn\_out**(**insn\_out**)**

**);**

// Instatiate the ID/IX pipeline register

id\_ix\_pipleline\_reg id\_ix\_pipleline\_reg**(**

**.**clk**(**clk**),**

**.**pc\_in**(**decode\_pc**),**

**.**ir\_in**(**decode\_ir**),**

**.**A\_in**(**dec\_A**),**

**.**B\_in**(**dec\_B**),**

**.**alu\_op\_in**(**dec\_alu\_op**),**

**.**is\_branch\_in**(**dec\_is\_branch**),**

**.**is\_jump\_in**(**dec\_is\_jump**),**

**.**op2\_sel\_in**(**dec\_op2\_sel**),**

**.**shift\_amount\_in**(**dec\_shift\_amount**),**

**.**branch\_type\_in**(**dec\_branch\_type**),**

**.**pc\_out**(**exe\_pc**),**

**.**ir\_out**(**exe\_ir**),**

**.**A\_out**(**exe\_A**),**

**.**B\_out**(**exe\_B**),**

**.**alu\_op\_out**(**exe\_alu\_op**),**

**.**is\_branch\_out**(**exe\_is\_branch**),**

**.**is\_jump\_out**(**exe\_is\_jump**),**

**.**op2\_sel\_out**(**exe\_op2\_sel**),**

**.**shift\_amount\_out**(**exe\_shift\_amount**),**

**.**branch\_type\_out**(**exe\_branch\_type**)**

**);**

sign\_extender sign\_extender**(**

**.**in\_data**(**exe\_ir**[**15**:**0**]),**

**.**out\_data**(**exe\_extended**)**

**);**

**assign** exe\_shift\_target **=** exe\_ir**[**25**:**0**];**

**assign** exe\_jump\_effective\_address **=** **(**exe\_pc **&** 32'hf0000000**)** **|** **((**exe\_shift\_target **<<** 2**)** **&** 32'h0fffffff**);**

**assign** exe\_shift\_immed **=** exe\_extended **<<** 2**;**

**assign** exe\_branch\_effective\_address **=** exe\_shift\_immed **+** exe\_pc**;**

// Instantiate a 32-bit mux for selecting which operand to provide to op2 of the ALU

mux\_2\_1\_32\_bit alu\_op2\_sel\_mux**(**

**.**line0**(**exe\_B**),**

**.**line1**(**exe\_extended**),**

**.**select**(**exe\_op2\_sel**),**

**.**output\_line**(**exe\_op2**)**

**);**

alu alu**(**

**.**op1**(**exe\_A**),** // operand 1 (always from rs)

**.**op2**(**exe\_op2**),** // operand 2

**.**operation**(**exe\_alu\_op**),** // The arithmatic operation to perform

**.**shift\_amount**(**exe\_shift\_amount**),** // The number of bits to shift

**.**result**(**exe\_O**),** // The arithmatic result based on the operation

**.**zero**(**exe\_zero**),** // Indicates if the result of the operation is zero.

**.**neg**(**exe\_neg**)**

**);**

branch\_resolve branch\_resolve**(**

**.**zero**(**exe\_zero**),**

**.**neg**(**exe\_neg**),**

**.**branch\_type**(**exe\_branch\_type**),**

**.**is\_branch**(**exe\_is\_branch**),**

**.**branch\_taken**(**exe\_branch\_taken**)** // Indicates if the branch is taken or not

**);**

mux\_2\_1\_32\_bit branch\_next\_pc\_mux**(**

**.**line0**(**exe\_pc**),**

**.**line1**(**exe\_branch\_effective\_address**),**

**.**select**(**exe\_branch\_taken**),**

**.**output\_line**(**exe\_branch\_next\_pc**)**

**);**

mux\_2\_1\_32\_bit jump\_next\_pc\_mux**(**

**.**line0**(**exe\_branch\_next\_pc**),**

**.**line1**(**exe\_jump\_effective\_address**),**

**.**select**(**exe\_is\_jump**),**

**.**output\_line**(**exe\_next\_pc**)**

**);**

// Control

**always** **@(posedge** clk**)** **begin**

reg\_file\_write\_enable **=** 0**;** // TODO: Change this later

// ----------- Decode Stage Control Signal Logic --------------------------- //

dec\_illegal\_insn **=** 0**;**

dest\_reg\_sel **=** 0**;**

dec\_alu\_op **=** 0**;**

dec\_op2\_sel **=** 0**;**

dec\_is\_branch **=** 0**;**

dec\_is\_jump **=** 0**;**

dec\_branch\_type **=** 0**;**

**if** **(((**opcode **&** 6'b111000**)>>** 3**)** **==** 3'h0**)** **begin**

**if** **((**opcode **&** 6'b000111**)** **==** 3'h0**)** **begin**

// We are in the SPECIAL Opcode encoding table

// This is an R-type instruction

dest\_reg\_sel **=** 0**;**

**if** **(**func **==** 6'b100000 **||** func **==** 6'b100001**)** // ADD, ADDU

dec\_alu\_op **=** 0**;** // Do an add operation

**else** **if** **(**func **==** 6'b100010 **||** func **==** 6'b100011**)** // SUB, SUBU

dec\_alu\_op **=** 1**;**

**else** **if** **(**func **==** 6'b011000 **||** func **==** 6'b011001**)** // MULT, MULTU

dec\_alu\_op **=** 2**;**

**else** **if** **(**func **==** 6'b011010 **||** func **==** 6'b011011**)** // DIV, DIVU

dec\_alu\_op **=** 3**;**

**else** **if** **(**func **==** 6'b000000**)** **begin** // SLL

dec\_alu\_op **=** 4**;**

dec\_shift\_amount **=** sha**;**

**end**

**else** **if** **(**func **==** 6'b000010**)** **begin** // SLL

dec\_alu\_op **=** 5**;**

dec\_shift\_amount **=** sha**;**

**end**

**else** **if** **(**func **==** 6'b101010 **||** func **==** 6'b101011**)** // SLT, SLTU

dec\_alu\_op **=** 6**;**

**else** **begin**

dec\_illegal\_insn **=** 1**;**

**end**

**end** **else** **if** **(((**opcode **&** 6'b000111**)** **==** 3'd2**)** **||** **((**opcode **&** 6'b000111**)** **==** 3'd3**))** **begin**

// J and JAL instructions

// This is J-type instruction

dest\_reg\_sel **=** 0**;**

dec\_is\_jump **=** 1**;**

**if** **(**opcode **==** 6'b000011**)** **begin**

// JAL instruction

dec\_illegal\_insn **=** 1**;**

// TODO: We need to store the return address into reg[31]

**end**

**end** **else** **begin**

// BEQ, BNE, BLEZ, BGTZ

// This is an I-type instruction

dec\_is\_branch **=** 1**;**

**if** **(**opcode **==** 6'b000100**)** **begin**

// BEQ

dec\_alu\_op **=** 1**;** // We want to do a test if the result is zero from a subtract

dec\_branch\_type **=** 0**;**

**end** **else** **if** **(**opcode **==** 6'b000101**)** **begin**

// BNE

dec\_alu\_op **=** 1**;** // We want to do a test if the result is zero from a subtract

dec\_branch\_type **=** 1**;**

**end** **else** **if** **(**opcode **==** 6'b000110**)** **begin**

// BLEZ

// TODO: Test that rt has zero in it

dec\_illegal\_insn **=** 1**;**

dec\_alu\_op **=** 0**;**

dec\_branch\_type **=** 2**;**

**end** **else** **begin**

// BGTZ

// TODO: Test that rt has zero in it

dec\_illegal\_insn **=** 1**;**

dec\_alu\_op **=** 0**;**

dec\_branch\_type **=** 3**;**

**end**

**end**

**end** **else** **if** **(((**opcode **&** 6'b111000**)>>** 3**)** **==** 3'd1**)** **begin**

// ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI

// This is an I-type instruction

**case** **(**opcode**)**

6'b001000**:** dec\_alu\_op **=** 0**;** // ADDI

6'b001001**:** dec\_alu\_op **=** 0**;** // ADDIU

6'b001010**:** dec\_alu\_op **=** 6**;** // SLTI

6'b001011**:** dec\_alu\_op **=** 6**;** // SLTUI

6'b001100**:** dec\_alu\_op **=** 7**;** // ANDI

6'b001101**:** dec\_alu\_op **=** 8**;** // ORI

6'b001110**:** dec\_alu\_op **=** 9**;** // XORI

6'b001111**:** dec\_alu\_op **=** 12**;** // LUI

**endcase**

dest\_reg\_sel **=** 1**;**

dec\_op2\_sel **=** 1**;** // We want op2 to be the immediate in the ALU

**end** **else** **if** **(((**opcode **&** 6'b111000**)>>** 3**)** **==** 3'd3**)** **begin**

// No idea?

dest\_reg\_sel **=** 0**;**

dec\_illegal\_insn **=** 1**;**

**end** **else** **if** **(((**opcode **&** 6'b111000**)>>** 3**)** **==** 3'd4**)** **begin**

// LB, LH, LWL, LW, LBU, LHU, LWR

// This is an I-type instruction

dest\_reg\_sel **=** 1**;**

dec\_op2\_sel **=** 1**;**

dec\_alu\_op **=** 0**;** // We want to add the value of rs to the immediate

**end** **else** **if** **(((**opcode **&** 6'b111000**)>>** 3**)** **==** 3'd5**)** **begin**

// SB, SH, SWL, SW, SWR

// This is an I-type instruction

dest\_reg\_sel **=** 1**;**

dec\_op2\_sel **=** 1**;**

dec\_alu\_op **=** 0**;** // We want to add the value of rs to the immediate

**end** **else** **begin**

dec\_illegal\_insn **=** 1**;**

**end**

**end**

**endmodule**

# Processor Test Bench

We did not change anything in our test bench actually but the code is shown here again for reference.

**module** processor\_tb**;**

**reg** clk**;**

**reg** srec\_parse**;** // enable signal for srec parser.

// Registers, writes, and variables for parser

**integer** fh **=** 0**;** // file handler for input

**integer** i **=** 0**;** // loop variable

**integer** data\_byte **=** 0**;** // variable to keep track of what byte we are on.

**integer** data\_offset **=** 0**;** // keep track of the offset from the data address to write the next byte.

**reg** **[**1**:**0**]**nibble\_count **=** 0**;** // keep track of which nibble is being written

**reg** **[**7**:**0**]**rec\_type**;** // record type number

**reg** **[**7**:**0**]** byte\_count**;** //byte count for address, data, and checksum

**integer** record\_code**;** // A record\_code is equivalent to 1 ASCII digit/letter in the .srec file

**reg** **[**31**:**0**]** rec\_address **=** 'b0**;** // the address given by the record.

**integer** highest\_address **=** 0**;** // highest address written by parser

**reg** **[**7**:**0**]**rec\_data**;** // A single byte of the data from the record.

**reg** **[**7**:**0**]**temp**;** // a temporary byte used for place holding.

**reg** done **=** 0**;** // this will set high when we are done parsing the file.

**reg** **[**7**:**0**]** file\_char **=** 8'h0A**;** // Set the initial character from the file read to be the new line character

// Instantiate the processor as the unit under test.

processor processor\_uut**(.**clk**(**clk**),** **.**srec\_parse**(**srec\_parse**));**

**initial** **begin**

//---------------------------------------------------------

// Parsing stage of testbench - memory is not valid until

// after the parser has finished!

//---------------------------------------------------------

// Before we begin parsing we want to make sure the fetch module does nothing until the memory has been populated with instructions. We will do this by asserting stall.

processor\_uut**.**stall **=** 1**;**

srec\_parse **=** 1**;**

$monitor**(**"Starting the SREC parser..."**);**

// Open the SREC file to read

fh **=** $fopen**(**"D:/ECE621\_PiplinedProcessor/BubbleSort.srec"**,** "r"**);**

// Start the clock high

clk **=** 1**;**

// loop until we set the done bit

**while** **(**done **==** 0**)** **begin**

**#**100**;** // Delay 1 clock cycle.

// Read the first/next character from the file.

file\_char **=** $fgetc**(**fh**);**

**if** **(**file\_char **==** 8'hff**)** **begin**

done **=** 1**;**

file\_char **=** 8'h0A**;**

**end**

// Reset the record byte which keeps track of the current byte of the line you are reading in.

record\_code **=** 0**;**

// Loop until we reach a new line character (new record).

**while** **(**file\_char **!=** 8'h0A**)** **begin**

**#**50**;** // Delay 1/2 clock cycle.

highest\_address **=** **(**rec\_address **>** highest\_address**)?** rec\_address**:** highest\_address**;**

**if** **(**record\_code **==** 0**)** **begin**

// Clear out all the bit fields.

rec\_type **=** 8'h4**;**

byte\_count **=** 16'h0**;**

rec\_address **=** 32'h0**;**

rec\_data **=** 132'h0**;**

data\_offset **=** 0**;**

data\_byte **=** 0**;**

**end** **else** **if** **(**record\_code **==** 1**)** **begin**

// read the record type.

rec\_type**[**7**:**0**]** **=** atoh**(**file\_char**);**

**end** **else** **if** **(**record\_code **==** 2**)** **begin**

// read the upper byte of the byte count.

temp **=** atoh**(**file\_char**);**

byte\_count**[**7**:**4**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 3**)** **begin**

// read the lower byte of the byte count.

temp **=** atoh**(**file\_char**);**

byte\_count**[**3**:**0**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **>** 3**)** **begin**

**if** **(**rec\_type **==** 1**)** **begin** // If record is a 16 bit address.

rec\_address**[**31**:**16**]** **=** 16'h0000**;**

**if** **(**record\_code **==** 4**)** **begin**

// read the middle byte of the address.

temp **=** atoh**(**file\_char**);**

// remove the upper most nibble since we only have single digits to represent memory addresses

rec\_address**[**15**:**12**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 5**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**11**:**8**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 6**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**7**:**4**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 7**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**3**:**0**]** **=** temp**[**3**:**0**];**

**end** **else** **begin**

// Check to see if we reached the end of the data

**if** **(**data\_byte **<** byte\_count **-** 2 **-** 1**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_data **=** rec\_data **<<** 4**;**

rec\_data**[**3**:**0**]** **=** temp**[**3**:**0**];**

nibble\_count **=** nibble\_count **+** 1**;**

**#**50**;**

**if** **(**nibble\_count **>** 1**)** **begin**

processor\_uut**.**srec\_address **=** rec\_address**+**data\_offset**;**

processor\_uut**.**srec\_data\_in **=** rec\_data**;**

processor\_uut**.**srec\_access\_size **=** 2'b01**;**

processor\_uut**.**srec\_rw **=** 1**;**

**#**100**;** // Delay one clock cycle

processor\_uut**.**srec\_rw **=** 0**;**

// update the data\_offest.

data\_offset **=** data\_offset **+** 1**;**

// reset the nibble count

nibble\_count **=** 0**;**

data\_byte **=** data\_byte **+** 1**;**

**end**

**end**

**end**

**end**

**if** **(**rec\_type **==** 2**)** **begin** // If record is 24 bit address.

rec\_address**[**31**:**24**]** **=** 8'h00**;**

**if** **(**record\_code **==** 4**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**23**:**20**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 5**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**19**:**16**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 6**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**15**:**12**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 7**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**11**:**8**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 8**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**7**:**4**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 9**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**3**:**0**]** **=** temp**[**3**:**0**];**

**end** **else** **begin**

// Check to see if we reached the end of the data

**if** **(**data\_byte **<** byte\_count **-** 3 **-** 1**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_data **=** rec\_data **<<** 4**;**

rec\_data**[**3**:**0**]** **=** temp**[**3**:**0**];**

nibble\_count **=** nibble\_count **+** 1**;**

**#**50**;**

**if** **(**nibble\_count **>** 1**)** **begin**

processor\_uut**.**srec\_address **=** rec\_address**+**data\_offset**;**

processor\_uut**.**srec\_data\_in **=** rec\_data**;**

processor\_uut**.**srec\_access\_size **=** 2'b00**;**

processor\_uut**.**srec\_rw **=** 1**;**

**#**100**;** // Delay one clock cycle

processor\_uut**.**srec\_rw **=** 0**;**

// update the data\_offest.

data\_offset **=** data\_offset **+** 1**;**

// reset the nibble count

nibble\_count **=** 0**;**

data\_byte **=** data\_byte **+** 1**;**

**end**

**end**

**end**

**end**

**if** **(**rec\_type **==** 3**)** **begin** //If record is a 32 bit address.

**if** **(**record\_code **==** 4**)** **begin**

// read the upper most byte of the address.

temp **=** atoh**(**file\_char**);**

// remove the upper most nibble since we only have single digits to represent memory addresses

rec\_address**[**31**:**28**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 5**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**27**:**24**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 6**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**23**:**20**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 7**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**19**:**16**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 8**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**15**:**12**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 9**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**11**:**8**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 10**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**7**:**4**]** **=** temp**[**3**:**0**];**

**end** **else** **if** **(**record\_code **==** 11**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_address**[**3**:**0**]** **=** temp**[**3**:**0**];**

**end** **else** **begin**

// Check to see if we reached the end of the data

**if** **(**data\_byte **<** byte\_count **-** 4 **-** 1**)** **begin**

temp **=** atoh**(**file\_char**);**

rec\_data **=** rec\_data **<<** 4**;**

rec\_data**[**3**:**0**]** **=** temp**[**3**:**0**];**

nibble\_count **=** nibble\_count **+** 1**;**

**#**50**;**

**if** **(**nibble\_count **>** 1**)** **begin**

processor\_uut**.**srec\_address **=** rec\_address**+**data\_offset**;**

processor\_uut**.**srec\_data\_in **=** rec\_data**;**

processor\_uut**.**srec\_access\_size **=** 2'b00**;**

processor\_uut**.**srec\_rw **=** 1**;**

**#**100**;** // Delay one clock cycle

processor\_uut**.**srec\_rw **=** 0**;**

// update the data\_offest.

data\_offset **=** data\_offset **+** 1**;**

// reset the nibble count

nibble\_count **=** 0**;**

data\_byte **=** data\_byte **+** 1**;**

**end**

**end**

**end**

**end**

**end**

**#**50**;** // delay 1/2 clock cycle

// increment record\_code

record\_code **=** record\_code **+** 1**;**

// read the next character from the file.

file\_char **=** $fgetc**(**fh**);**

**end**

**end**

srec\_parse **=** 0**;**

**#**100**;**

// Close up the file

$fclose**(**fh**);**

$monitor**(**"Done parsing the SREC file!"**);**

**#**100**;**

// ------------------------------------------------------------

// Memory is ready to be used after this point!

// ------------------------------------------------------------

$monitor**(**"Beginning the fetch-decode-execute loop!"**);**

**#**100**;**

// Deassert stall just read out the pc, rw, and access size.

processor\_uut**.**stall **=** 0**;**

// ------------------------------------------------------------

// This section prints out the decoded instructions

// ------------------------------------------------------------

**while** **(**processor\_uut**.**pc **<=** highest\_address **-** 4**)** **begin**

**@(posedge** clk**);**

**case(**processor\_uut**.**opcode**)**

6'd0**:** **begin** //JR, ADD, ADDU, SUB SUBU, DIV, SLT, SLTU, SLL, SRL, SRA, AND, OR, XOR, NOR, NOP

**case(**processor\_uut**.**func**)**

6'd0**:** **begin** //SLL, NOP

**if** **(**processor\_uut**.**sha **==** 5'b0**)** $strobe**(**"%h: %h NOP"**,** processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**);**

**else** $strobe**(**"%h: %h sll %h, %h, %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rd**,** processor\_uut**.**rt**,** processor\_uut**.**sha**);**

**end**

6'd2**:** **begin** //SRL

$strobe**(**"%h: %h srl %h, %h, %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rd**,** processor\_uut**.**rt**,** processor\_uut**.**sha**);**

**end**

6'd3**:** **begin** //SRA

$strobe**(**"%h: %h sra %h, %h, %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rd**,** processor\_uut**.**rt**,** processor\_uut**.**sha**);**

**end**

6'd8**:** **begin** //JR

$strobe**(**"%h: %h jr %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rs**);**

**end**

6'd26**:** **begin** //DIV

$strobe**(**"%h: %h div %h, %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rs**,** processor\_uut**.**rt**);**

**end**

6'd32**:** **begin** //ADD

$strobe**(**"%h: %h add %h, %h, %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rd**,** processor\_uut**.**rs**,** processor\_uut**.**rt**);**

**end**

6'd33**:** **begin** //ADDU

$strobe**(**"%h: %h addu %h, %h, %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rd**,** processor\_uut**.**rs**,** processor\_uut**.**rt**);**

**end**

6'd34**:** **begin** //SUB

$strobe**(**"%h: %h sub %h, %h, %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rd**,** processor\_uut**.**rs**,** processor\_uut**.**rt**);**

**end**

6'd35**:** **begin** //SUBU

$strobe**(**"%h: %h subu %h, %h, %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rd**,** processor\_uut**.**rs**,** processor\_uut**.**rt**);**

**end**

6'd36**:** **begin** //AND

$strobe**(**"%h: %h and %h, %h, %h"**,**

processor\_uut**.**pc\_out**,** processor\_uut**.**insn\_out**,** processor\_uut**.**rd**,** processor\_uut**.**rs**,** processor\_uut**.**rt**);**

**end**

6'd37**:** **begin** //OR

$strobe("%h: %h or %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rd, processor\_uut.rs, processor\_uut.rt);

end

6'd38: begin //XOR

$strobe("%h: %h xor %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rd, processor\_uut.rs, processor\_uut.rt);

end

6'd39: begin //NOR

$strobe("%h: %h nor %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rd, processor\_uut.rs, processor\_uut.rt);

end

6'd42: begin //SLT

$strobe("%h: %h slt %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rd, processor\_uut.rs, processor\_uut.rt);

end

6'd43: begin //SLTU

$strobe("%h: %h sltu %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rd, processor\_uut.rs, processor\_uut.rt);

end

default: begin

$strobe("ERROR! %h: %h", processor\_uut.pc\_out, processor\_uut.insn\_out);

$stop;

end

endcase

end

6'd1: begin //BLTZ, BGEZ

case(processor\_uut.rt)

5'd0: begin //BLTZ

$strobe("%h: %h bltz %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rs, processor\_uut.immed);

end

5'd1: begin //BGEZ

$strobe("%h: %h bgez %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rs, processor\_uut.immed);

end

default: begin

$strobe("ERROR! %h: %h", processor\_uut.pc\_out, processor\_uut.insn\_out);

$stop;

end

endcase

end

6'd2: begin //J

//display the destination address of the jump, not the offset.

$strobe("%h: %h j %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, {processor\_uut.pc\_out[31-:4], 28'b0}+{processor\_uut.target,2'b0});

end

6'd3: begin //JAL

//display the destination address of the jump, not the offset.

$strobe("%h: %h jal %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, {processor\_uut.pc\_out[31-:4], 28'b0}+{processor\_uut.target,2'b0});

end

6'd4: begin //BEQ

$strobe("%h: %h beq %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rs, processor\_uut.rt, processor\_uut.immed);

end

6'd5: begin //BNE

$strobe("%h: %h bne %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rs, processor\_uut.rt, processor\_uut.immed);

end

6'd6: begin //BLEZ

$strobe("%h: %h beq %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rs, processor\_uut.immed);

end

6'd7: begin //BGTZ

$strobe("%h: %h bgtz %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rs, processor\_uut.immed);

end

6'd9: begin //ADDIU

$strobe("%h: %h addiu %h, %h, %d",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rt, processor\_uut.rs, processor\_uut.immed);

end

6'd10: begin //SLTI

$strobe("%h: %h slti %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rt, processor\_uut.rs, processor\_uut.immed);

end

6'd13: begin //ORI

$strobe("%h: %h ori %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rt, processor\_uut.rs, processor\_uut.immed);

end

6'd15: begin //LUI

$strobe("%h: %h lui %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rt, processor\_uut.immed);

end

6'd28: begin //MUL

$strobe("%h: %h mul %h, %h, %h",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rd, processor\_uut.rs, processor\_uut.rt);

end

6'd32: begin //LB

$strobe("%h: %h lb %h, %d(%h)",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rt, processor\_uut.immed, processor\_uut.rs);

end

6'd35: begin //LW

$strobe("%h: %h lw %h, %d(%h)",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rt, processor\_uut.immed, processor\_uut.rs);

end

6'd36: begin //LBU

$strobe("%h: %h lbu %h, %d(%h)",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rt, processor\_uut.immed, processor\_uut.rs);

end

6'd40: begin //SB

$strobe("%h: %h sb %h, %d(%h)",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rt, processor\_uut.immed, processor\_uut.rs);

end

6'd43: begin //SW

$strobe("%h: %h sw %h, %d(%h)",

processor\_uut.pc\_out, processor\_uut.insn\_out, processor\_uut.rt, processor\_uut.immed, processor\_uut.rs);

end

default: begin

$strobe("ERROR! %h: %h", processor\_uut.pc\_out, processor\_uut.insn\_out);

$stop;

end

endcase

end

#100;

$stop;

end

always begin

#50 clk = !clk;

end

// A function to convert ASCII upper case letters and digits to their hexadecimal value.

function [7:0]atoh;

input [7:0]aCode;

begin

if (aCode >= 8'h30 && aCode <= 8'h39) begin

atoh = aCode - 8'h30;

end else if (aCode >= 8'h41 && aCode <= 8'h5A) begin

atoh = aCode - 8'h37;

end

end

endfunction

endmodule

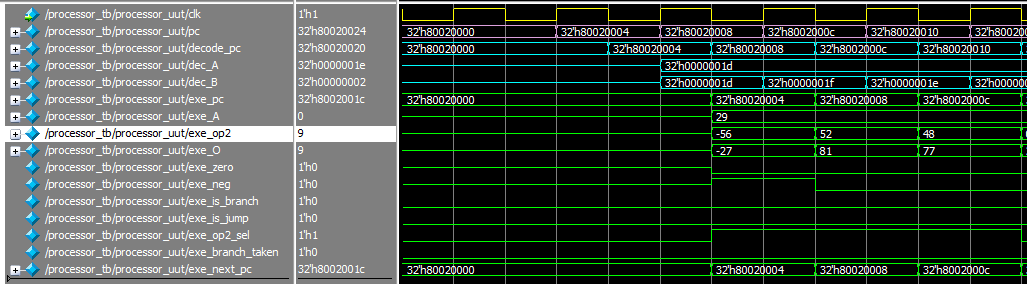
# Verification Demonstration

For the verification the setup for us that it is easiest to show is the waveform diagram as the program executes. For readability purposes we have only shown 4 instructions per waveform along with the instruction printout we get from our test bench (this was verified last PD). We show the operands to the ALU, and output from the ALU as decimal to improve readability. The plum colored wave forms are for the fetch stage while the cyan ones are for decode and the green for execute. We executed the bubble sort bench mark in the following diagrams.

# 80020004: 27bdffc8 addiu 1d, 1d, 65480

# 80020008: afbf0034 sw 1f, 52(1d)

# 8002000c: afbe0030 sw 1e, 48(1d)

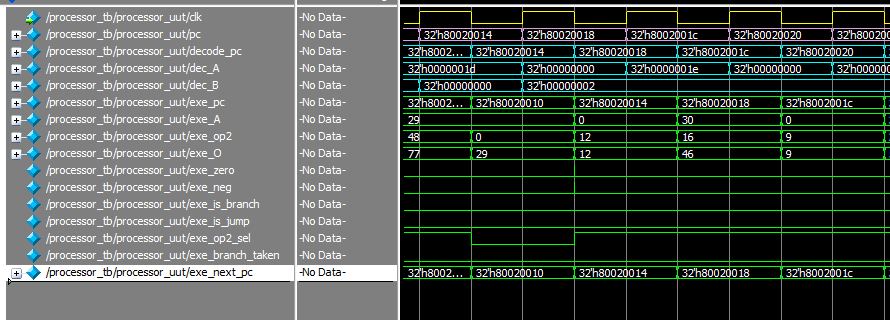


# 80020010: 03a0f021 addu 1e, 1d, 00

# 80020014: 2402000c addiu 02, 00, 12

# 80020018: afc20010 sw 02, 16(1e)

# 8002001c: 24020009 addiu 02, 00, 9

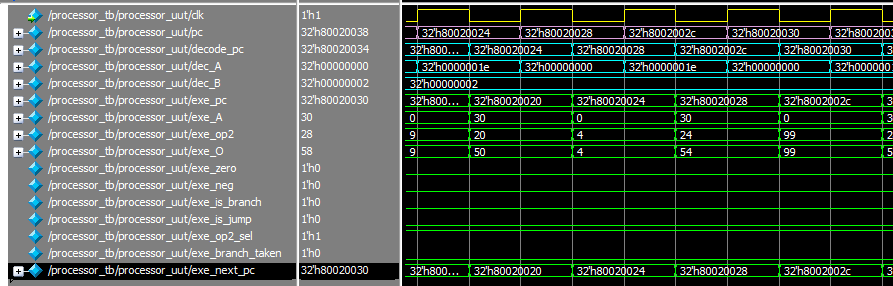


# 80020020: afc20014 sw 02, 20(1e)

# 80020024: 24020004 addiu 02, 00, 4

# 80020028: afc20018 sw 02, 24(1e)

# 8002002c: 24020063 addiu 02, 00, 99

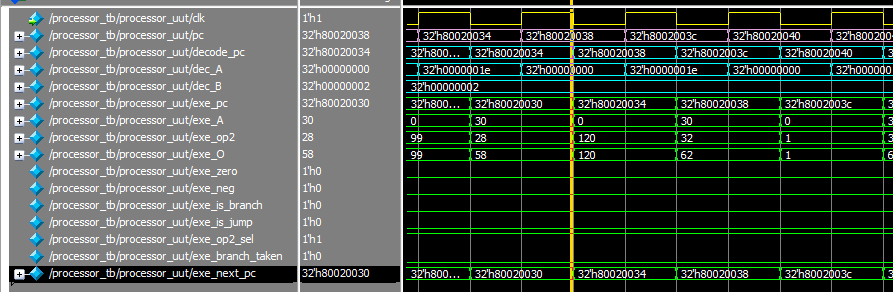


# 80020030: afc2001c sw 02, 28(1e)

# 80020034: 24020078 addiu 02, 00, 120

# 80020038: afc20020 sw 02, 32(1e)

# 8002003c: 24020001 addiu 02, 00, 1

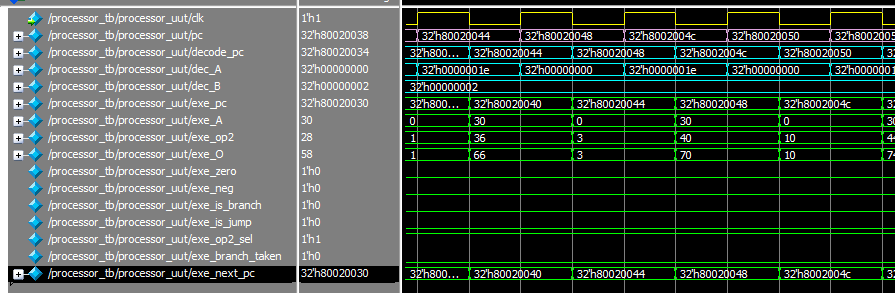


# 80020040: afc20024 sw 02, 36(1e)

# 80020044: 24020003 addiu 02, 00, 3

# 80020048: afc20028 sw 02, 40(1e)

# 8002004c: 2402000a addiu 02, 00, 10

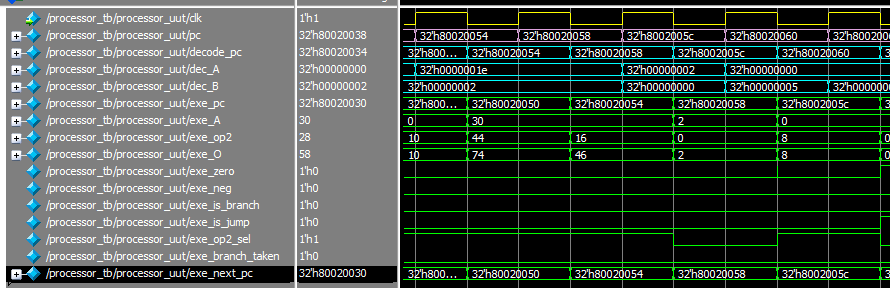


# 80020050: afc2002c sw 02, 44(1e)

# 80020054: 27c20010 addiu 02, 1e, 16

# 80020058: 00402021 addu 04, 02, 00

# 8002005c: 24050008 addiu 05, 00, 8

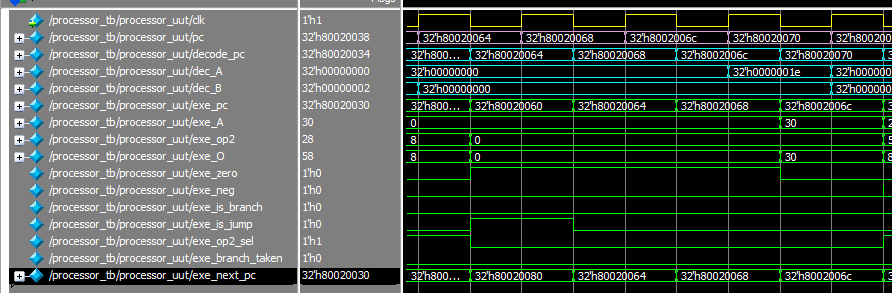


# 80020060: 0c008020 jal 80020080

# 80020064: 00000000 NOP

# 80020068: 00001021 addu 02, 00, 00

# 8002006c: 03c0e821 addu 1d, 1e, 00

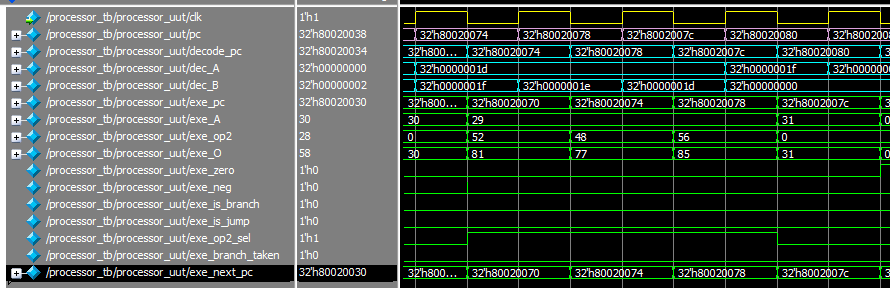


# 80020070: 8fbf0034 lw 1f, 52(1d)

# 80020074: 8fbe0030 lw 1e, 48(1d)

# 80020078: 27bd0038 addiu 1d, 1d, 56

# 8002007c: 03e00008 jr 1f

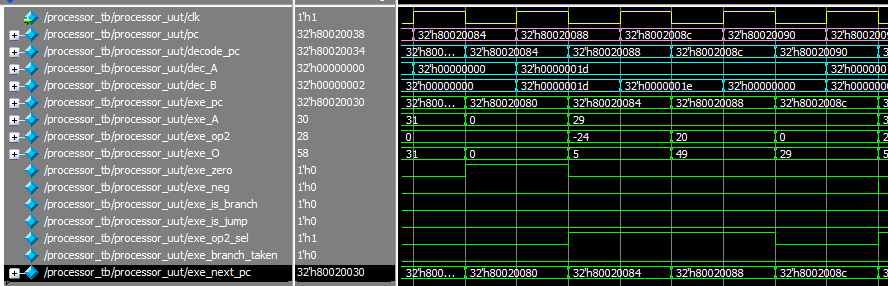


# 80020080: 00000000 NOP

# 80020084: 27bdffe8 addiu 1d, 1d, 65512

# 80020088: afbe0014 sw 1e, 20(1d)

# 8002008c: 03a0f021 addu 1e, 1d, 00

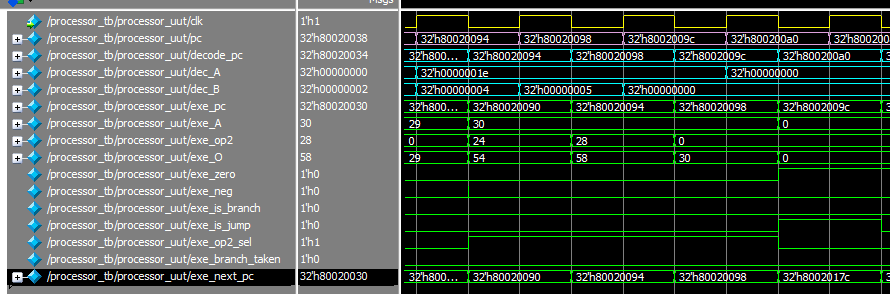


# 80020090: afc40018 sw 04, 24(1e)

# 80020094: afc5001c sw 05, 28(1e)

# 80020098: afc00000 sw 00, 0(1e)

# 8002009c: 0800805f j 8002017c

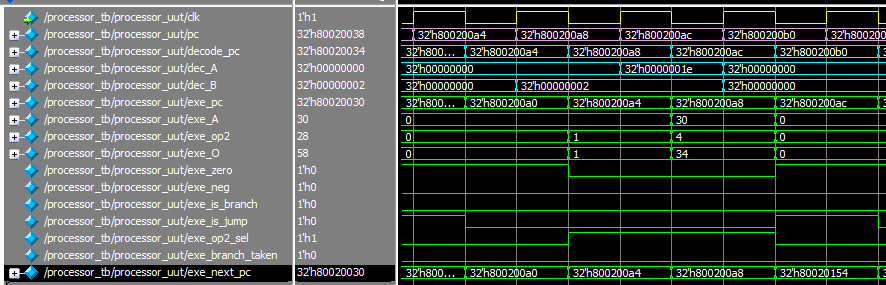


# 800200a0: 00000000 NOP

# 800200a4: 24020001 addiu 02, 00, 1

# 800200a8: afc20004 sw 02, 4(1e)

# 800200ac: 08008055 j 80020154

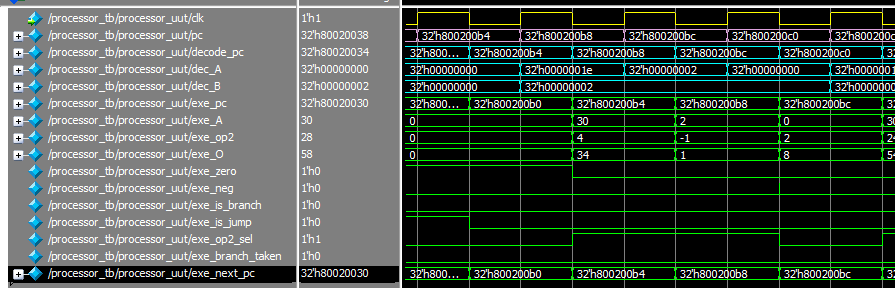


# 800200b0: 00000000 NOP

# 800200b4: 8fc20004 lw 02, 4(1e)

# 800200b8: 2442ffff addiu 02, 02, 65535

# 800200bc: 00021080 sll 02, 02, 02

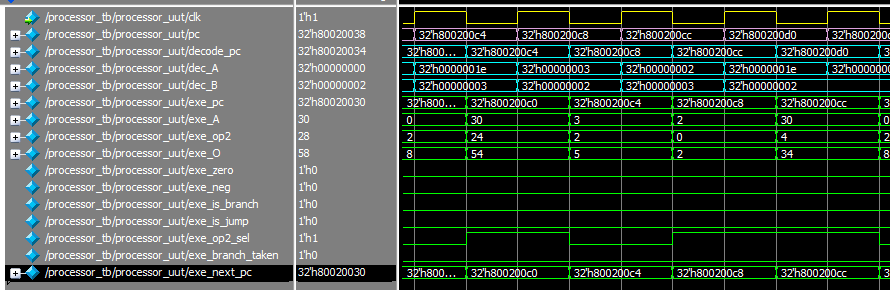


# 800200c0: 8fc30018 lw 03, 24(1e)

# 800200c4: 00621021 addu 02, 03, 02

# 800200c8: 8c430000 lw 03, 0(02)

# 800200cc: 8fc20004 lw 02, 4(1e)

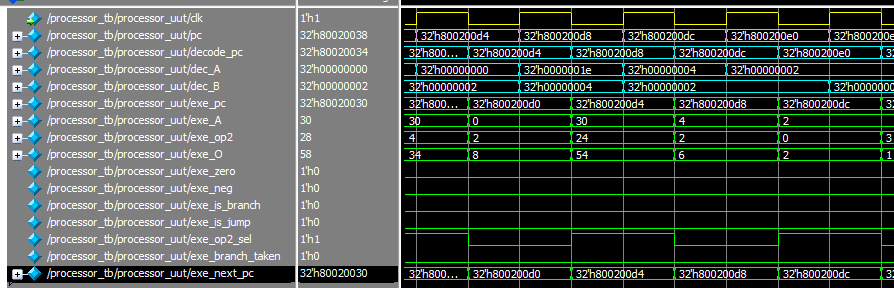


# 800200d0: 00021080 sll 02, 02, 02

# 800200d4: 8fc40018 lw 04, 24(1e)

# 800200d8: 00821021 addu 02, 04, 02

# 800200dc: 8c420000 lw 02, 0(02)

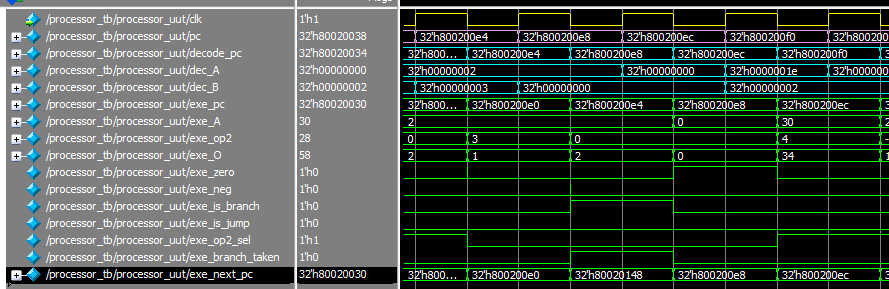


# 800200e0: 0043102a slt 02, 02, 03

# 800200e4: 10400019 beq 02, 00, 0019

# 800200e8: 00000000 NOP

# 800200ec: 8fc20004 lw 02, 4(1e)

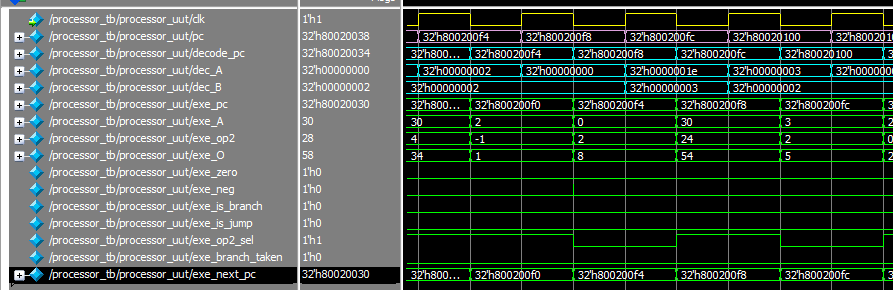


# 800200f0: 2442ffff addiu 02, 02, 65535

# 800200f4: 00021080 sll 02, 02, 02

# 800200f8: 8fc30018 lw 03, 24(1e)

# 800200fc: 00621021 addu 02, 03, 02

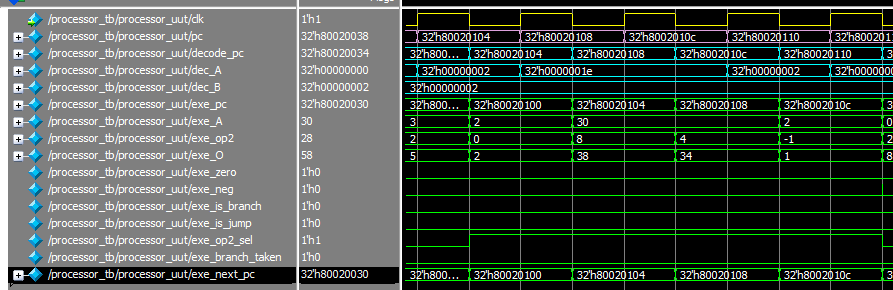


# 80020100: 8c420000 lw 02, 0(02)

# 80020104: afc20008 sw 02, 8(1e)

# 80020108: 8fc20004 lw 02, 4(1e)

# 8002010c: 2442ffff addiu 02, 02, 65535

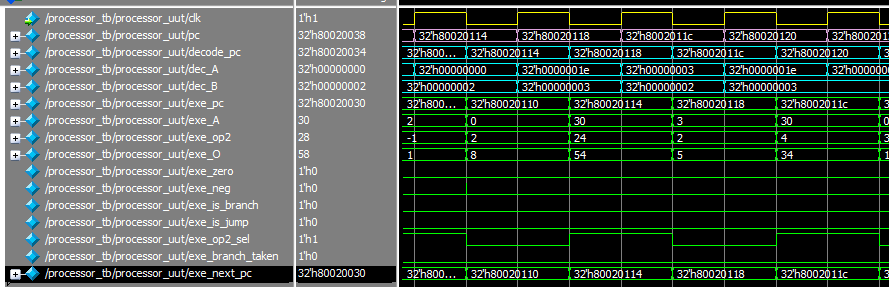


# 80020110: 00021080 sll 02, 02, 02

# 80020114: 8fc30018 lw 03, 24(1e)

# 80020118: 00621021 addu 02, 03, 02

# 8002011c: 8fc30004 lw 03, 4(1e)

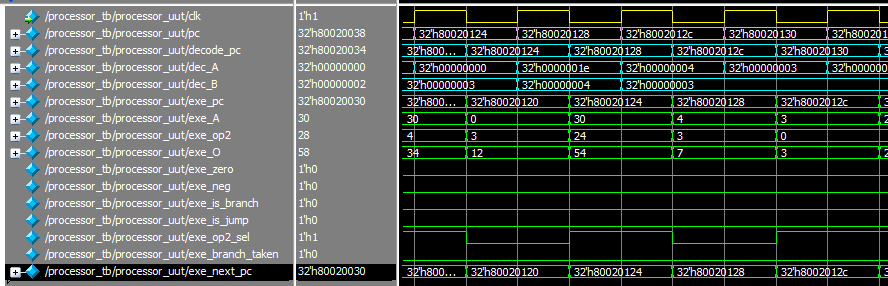


# 80020120: 00031880 sll 03, 03, 02

# 80020124: 8fc40018 lw 04, 24(1e)

# 80020128: 00831821 addu 03, 04, 03

# 8002012c: 8c630000 lw 03, 0(03)

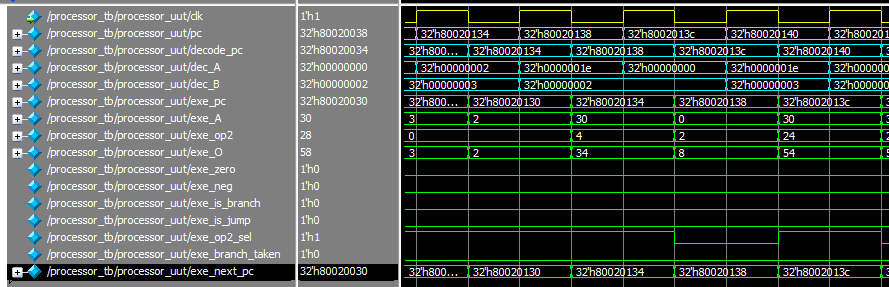


# 80020130: ac430000 sw 03, 0(02)

# 80020134: 8fc20004 lw 02, 4(1e)

# 80020138: 00021080 sll 02, 02, 02

# 8002013c: 8fc30018 lw 03, 24(1e)



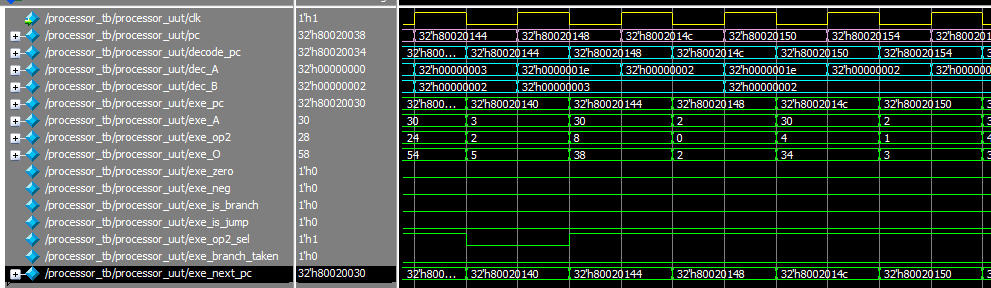
# 80020140: 00621021 addu 02, 03, 02

# 80020144: 8fc30008 lw 03, 8(1e)

# 80020148: ac430000 sw 03, 0(02)

# 8002014c: 8fc20004 lw 02, 4(1e)

# 80020150: 24420001 addiu 02, 02, 1

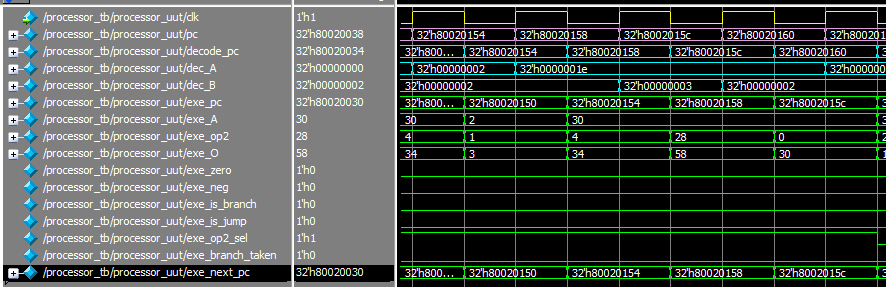


# 80020150: 24420001 addiu 02, 02, 1

# 80020154: afc20004 sw 02, 4(1e)

# 80020158: 8fc3001c lw 03, 28(1e)

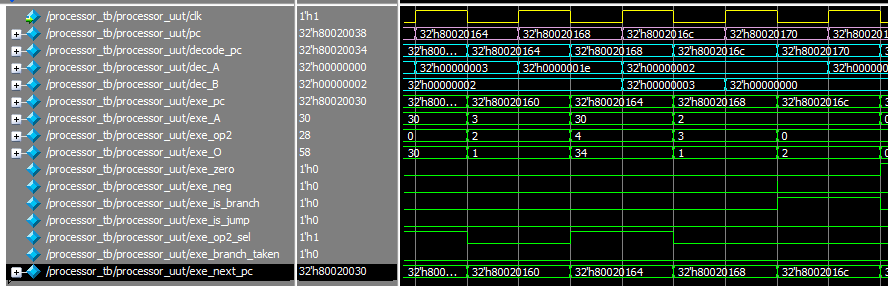
# 8002015c: 8fc20000 lw 02, 0(1e)



# 80020164: 8fc20004 lw 02, 4(1e)

# 80020168: 0043102a slt 02, 02, 03

# 8002016c: 1440ffd1 bne 02, 00, ffd1

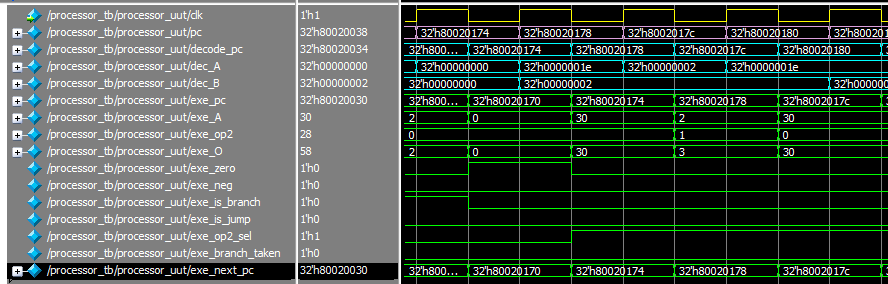


# 80020170: 00000000 NOP

# 80020174: 8fc20000 lw 02, 0(1e)

# 80020178: 24420001 addiu 02, 02, 1

# 8002017c: afc20000 sw 02, 0(1e)

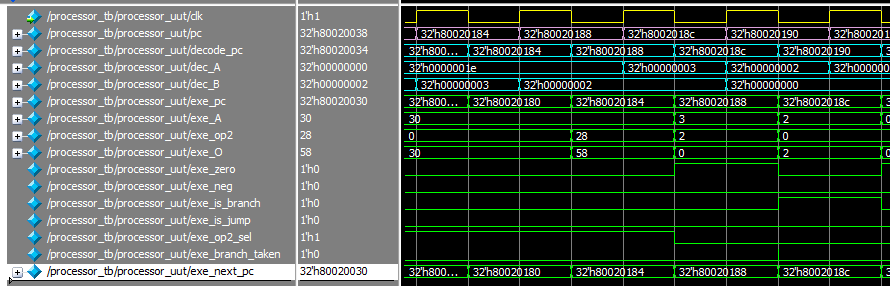


# 80020180: 8fc30000 lw 03, 0(1e)

# 80020184: 8fc2001c lw 02, 28(1e)

# 80020188: 0062102a slt 02, 03, 02

# 8002018c: 1440ffc5 bne 02, 00, ffc5



# 80020190: 00000000 NOP

# 80020194: 03c0e821 addu 1d, 1e, 00

# 80020198: 8fbe0014 lw 1e, 20(1d)

# 8002019c: 27bd0018 addiu 1d, 1d, 24

