

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

**Main Memory and SREC Parser**

Group 2

**Jordan *Ross,* Mustafa Faraj**   
j25ross, mabdulaz 17 October, 2014

# Fetch Module Code

This section contains the code for the fetch module. It will determine what instruction to read next by placing the address in the PC. For right now it simply puts the current value of the PC in the PC register and increments by 4 for the next instruction.

**module** fetch**(**

**input** clk\_in**,** // Clock signal for the fetch module

**input** stall\_in**,** // Indicates a stall

**output** **reg** **[**31**:**0**]**pc\_out**,** // Supplies the address in the PC register to the address input of the MM and decode stage

**output** **reg** rw\_out**,** // Indicates whether the fetch stage is performing a read or write to MM

**output** **reg** **[**1**:**0**]**access\_size\_out // Supplies the size of the word that is being accessed by the fetch module

**);**

// Parameter to define the address spaces

**parameter** instruction\_offset **=** 32'h80020000**;**

**reg** **[**31**:**0**]**pc **=** instruction\_offset**;** // The PC register

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

pc\_out **=** pc**;**

rw\_out **=** 0**;**

access\_size\_out **=** 2'b10**;**

**case(**stall\_in**)**

0**:** **begin**

// Increment the pc by 4 so we read the next

// instruction on the next clock cycle.

pc **<=** pc **+** 4**;**

**end**

1**:** **begin**

// TODO: Stalls not implemented

pc **<=** pc**;**

**end**

**default:** **begin**

$display**(**"Unknown signal!"**);**

**end**

**endcase**

**end**

**endmodule**

# Decode Module Code

This section contains the code for the decode module. The decode module simply slices up the instruction into separate wires (i.e. rs, rt, immed). It however makes no distinction of which wires will be used in getting registers from the register file since we will leave that up to the control unit.

**module** decode**(**

**input** clk**,**

**input** stall**,**

**input** **[**31**:**0**]**insn\_in**,** // Instruction bits to decode

**input** **[**31**:**0**]**pc\_in**,** // Program counter for the instruction.

**output** **wire** **[**4**:**0**]**rs**,**

**output** **wire** **[**4**:**0**]**rt**,**

**output** **wire** **[**4**:**0**]**rd**,**

**output** **wire** **[**4**:**0**]**sha**,**

**output** **wire** **[**5**:**0**]**func**,**

**output** **wire** **[**15**:**0**]**immed**,**

**output** **wire** **[**25**:**0**]**target**,**

**output** **wire** **[**5**:**0**]**opcode**,**

**output** **wire** **[**31**:**0**]**pc\_out**,**

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

**);**

//reg illegal\_insn;

**reg** **[**31**:**0**]** pc\_in\_r**;**

**assign** pc\_out **=** pc\_in\_r**;**

**assign** insn\_out **=** **(**stall**)?** 32'b0**:** insn\_in**;**

**assign** **{**opcode**,** rs**,** rt**,** rd**,** sha**,** func**}** **=** insn\_in**;**

**assign** target **=** insn\_in**[**25**:**0**];**

**assign** immed **=** insn\_in**[**15**:**0**];**

**always** **@** **(posedge** clk**)** pc\_in\_r **<=** pc\_in**;**

**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 and memory so far. This module will grow with later PDs but for now it simply connects our fetch, decode and memory modules. The processor also instantiates a set of multiplexors to add some control for when the parser is running. Only one of the multiplexors is shown below since all are very similar.

**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**;**

// 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**;**

// 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**)**

**);**

// Instantiate the decode module

decode decoder**(**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**);**

**endmodule**

**module** mux\_2\_1\_32\_bit**(**

**input** **[**31**:**0**]** line0**,**

**input** **[**31**:**0**]** line1**,**

**input** select**,**

**output** **reg** **[**31**:**0**]** output\_line

**);**

**always** **@(**line0 **or** line1 **or** select**)** **begin**

**case(**select**)**

0**:** output\_line **=** line0**;**

1**:** output\_line **=** line1**;**

**default:** output\_line **=** 32'hx**;**

**endcase**

**end**

**endmodule**

# Processor Test Bench

This is the test bench that was created to test the fetch and decode modules that were implemented in this PD. The test bench was built off of the old SREC parser in the previous PD since we needed to initialize our memory. Then we have added some logic to print out the instruction that was fetched, its operands, and its operation. We use this test bench in the following section for our verification demonstration.

///////////////////////////////////////////////////////////////////////////////

// Company: University of Waterloo

// Author(s): Jordan Ross, Mustafa Faraj

//

// Created Date: 08/10/2014

// Design Name:

// Module Name: fetch

// Project Name: ECE621\_PipelinedProcessor

// Description: This module is the fetch module for the execution loop of the

// pipelined processor. It will read the instruction from main memory given by

// the address in the program counter (PC) register. This instruction will then

// be fed into a decode module.

//

// Dependencies: memory.v

//

// Revision:

// 0.01 - File Created.

//

// Additional Comments:

//

///////////////////////////////////////////////////////////////////////////////

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

**reg** clk**;**

**reg** srec\_parse**;** // control signal for if the srec parser is active or not.

// Registers, writes, and variables for parser

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

**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 (upper/lower).

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

**reg** **[**7**:**0**]** byte\_count**;** // the number of bytes for the 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**;**

**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 the memory has been

// populated with instructions. We will do this by setting the stall\_in to 1 so nothing happens.

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

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

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

// Open the SREC file to read

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

//fh = $fopen("D:/Dropbox/Grad/01 Fall14/ECE621/Labs/L1/code/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.

// This is equivalent to 1 ASCII code from the file.

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

// Loop until we reach a new line character which signifies a 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 the record type is for 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 have reached the end of the data

**if** **(**data\_byte **<** byte\_count **-** 2 **-** 1**)** **begin** // Make sure we are less than the byte count minus the address size in bytes and checksum

// We are reading data so we want to create a lower and an upper nibble of a byte then write it to memory when we have both.

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**

// We have both nibbles so we should write the byte to memory

// set all the lines on the falling edge of the clock.

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 the record type is for a 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 have reached the end of the data

**if** **(**data\_byte **<** byte\_count **-** 3 **-** 1**)** **begin** // Make sure we are less than the byte count minus the address size in bytes and checksum

// We are reading data so we want to create a lower and an upper nibble of a byte then write it to memory when we have both.

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**

// We have both nibbles so we should write the byte to memory

// set all the lines on the falling edge of the clock.

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 the record type is for 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 have reached the end of the data

**if** **(**data\_byte **<** byte\_count **-** 4 **-** 1**)** **begin** // Make sure we are less than the byte count minus the address size in bytes and checksum

// We are reading data so we want to create a lower and an upper nibble of a byte then write it to memory when we have both.

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**

// We have both nibbles so we should write the byte to memory

// set all the lines on the falling edge of the clock.

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**;**

// Set the stall in to be 0 just read out the pc, rw, and access size.

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

//$monitor("%h: %h ", processor\_uut.pc\_out, processor\_uut.insn\_out);

**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

//write logic to grab instruction and its operands

//$monitor("Instruction = %h", processor\_uut.decoder.insn\_in);

#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

There are two demonstrations that are shown in this section. The first is that the fetch module is retrieving an instruction every clock cycle and that it is providing the correct PC to the decode module. The second is that for one of the benchmarks the test bench correctly fetches and decodes all of the instructions.

The waveform below shows that the processor is getting the next PC (PC 🡨 PC+4) every clock cycle since a stall has not been triggered. It also shows that the instruction is being fetched from memory every clock cycle and decoded.

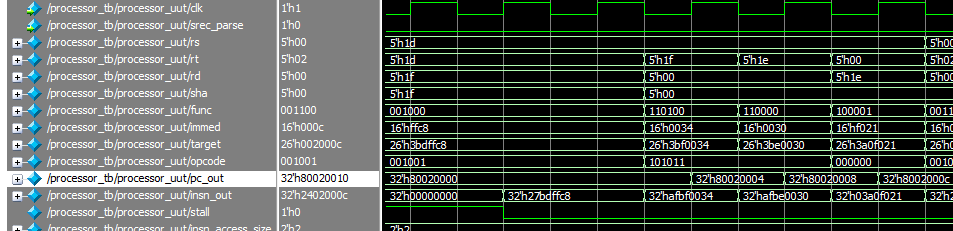


Figure 1. Waveform showing that the PC is being incremented by 4 each clock cycle and the memory is providing the instruction at that that PC on the next clock cycle.

The table below shows a snapshot of the print out for the processor test bench when running the BubbleSort.srec file on the right and the actual disassembly of the benchmark on the right. The line highlighted in yellow is a pseudo instruction MOVE in the disassembly and it is converted into a supported instruction by simply adding 0 to the source register and placing it in the destination register. The highlighted green line shows another pseudo instruction but this is LI which is converted by adding the immediate to the zero register and placing it in the destination register.

|  |  |
| --- | --- |
| **ModelSim Console Printout** | **Benchmark Disassembly** |
| # 80020000: 27bdffc8 addiu 1d, 1d, 65480 | 80020000: 27bdffc8 addiu sp,sp,-56 |
| # 80020004: afbf0034 sw 1f, 52(1d) | 80020004: afbf0034 sw ra,52(sp) |
| # 80020008: afbe0030 sw 1e, 48(1d) | 80020008: afbe0030 sw s8,48(sp) |
| # 8002000c: 03a0f021 addu 1e, 1d, 00 | 8002000c: 03a0f021 move s8,sp |
| # 80020010: 2402000c addiu 02, 00, 12 | 80020010: 2402000c li v0,12 |
| # 80020014: afc20010 sw 02, 16(1e) | 80020014: afc20010 sw v0,16(s8) |
| # 80020018: 24020009 addiu 02, 00, 9 | 80020018: 24020009 li v0,9 |
| # 8002001c: afc20014 sw 02, 20(1e) | 8002001c: afc20014 sw v0,20(s8) |
| # 80020020: 24020004 addiu 02, 00, 4 | 80020020: 24020004 li v0,4 |
| # 80020024: afc20018 sw 02, 24(1e) | 80020024: afc20018 sw v0,24(s8) |
| # 80020028: 24020063 addiu 02, 00, 99 | 80020028: 24020063 li v0,99 |
| # 8002002c: afc2001c sw 02, 28(1e) | 8002002c: afc2001c sw v0,28(s8) |
| # 80020030: 24020078 addiu 02, 00, 120 | 80020030: 24020078 li v0,120 |
| # 80020034: afc20020 sw 02, 32(1e) | 80020034: afc20020 sw v0,32(s8) |
| # 80020038: 24020001 addiu 02, 00, 1 | 80020038: 24020001 li v0,1 |
| # 8002003c: afc20024 sw 02, 36(1e) | 8002003c: afc20024 sw v0,36(s8) |
| # 80020040: 24020003 addiu 02, 00, 3 | 80020040: 24020003 li v0,3 |
| # 80020044: afc20028 sw 02, 40(1e) | 80020044: afc20028 sw v0,40(s8) |
| # 80020048: 2402000a addiu 02, 00, 10 | 80020048: 2402000a li v0,10 |
| # 8002004c: afc2002c sw 02, 44(1e) | 8002004c: afc2002c sw v0,44(s8) |
| # 80020050: 27c20010 addiu 02, 1e, 16 | 80020050: 27c20010 addiu v0,s8,16 |
| # 80020054: 00402021 addu 04, 02, 00 | 80020054: 00402021 move a0,v0 |
| # 80020058: 24050008 addiu 05, 00, 8 | 80020058: 24050008 li a1,8 |
| # 8002005c: 0c008020 jal 80020080 | 8002005c: 0c008020 jal 80020080 <bubble\_srt> |
| # 80020060: 00000000 NOP | 80020060: 00000000 nop |
| # 80020064: 00001021 addu 02, 00, 00 | 80020064: 00001021 move v0,zero |
| # 80020068: 03c0e821 addu 1d, 1e, 00 | 80020068: 03c0e821 move sp,s8 |
| # 8002006c: 8fbf0034 lw 1f, 52(1d) | 8002006c: 8fbf0034 lw ra,52(sp) |
| # 80020070: 8fbe0030 lw 1e, 48(1d) | 80020070: 8fbe0030 lw s8,48(sp) |
| # 80020074: 27bd0038 addiu 1d, 1d, 56 | 80020074: 27bd0038 addiu sp,sp,56 |
| # 80020078: 03e00008 jr 1f | 80020078: 03e00008 jr ra |
| # 8002007c: 00000000 NOP | 8002007c: 00000000 nop |
| # 80020080: 27bdffe8 addiu 1d, 1d, 65512 | 80020080: 27bdffe8 addiu sp,sp,-24 |
| # 80020084: afbe0014 sw 1e, 20(1d) | 80020084: afbe0014 sw s8,20(sp) |
| # 80020088: 03a0f021 addu 1e, 1d, 00 | 80020088: 03a0f021 move s8,sp |
| # 8002008c: afc40018 sw 04, 24(1e) | 8002008c: afc40018 sw a0,24(s8) |
| # 80020090: afc5001c sw 05, 28(1e) | 80020090: afc5001c sw a1,28(s8) |
| # 80020094: afc00000 sw 00, 0(1e) | 80020094: afc00000 sw zero,0(s8) |
| # 80020098: 0800805f j 8002017c | 80020098: 0800805f j 8002017c <bubble\_srt+0xfc> |
| # 8002009c: 00000000 NOP | 8002009c: 00000000 nop |
| # 800200a0: 24020001 addiu 02, 00, 1 | 800200a0: 24020001 li v0,1 |
| # 800200a4: afc20004 sw 02, 4(1e) | 800200a4: afc20004 sw v0,4(s8) |
| # 800200a8: 08008055 j 80020154 | 800200a8: 08008055 j 80020154 <bubble\_srt+0xd4> |
| # 800200ac: 00000000 NOP | 800200ac: 00000000 nop |
| # 800200b0: 8fc20004 lw 02, 4(1e) | 800200b0: 8fc20004 lw v0,4(s8) |
| # 800200b4: 2442ffff addiu 02, 02, 65535 | 800200b4: 2442ffff addiu v0,v0,-1 |
| # 800200b8: 00021080 sll 02, 02, 02 | 800200b8: 00021080 sll v0,v0,0x2 |
| # 800200bc: 8fc30018 lw 03, 24(1e) | 800200bc: 8fc30018 lw v1,24(s8) |
| # 800200c0: 00621021 addu 02, 03, 02 | 800200c0: 00621021 addu v0,v1,v0 |
| # 800200c4: 8c430000 lw 03, 0(02) | 800200c4: 8c430000 lw v1,0(v0) |
| # 800200c8: 8fc20004 lw 02, 4(1e) | 800200c8: 8fc20004 lw v0,4(s8) |
| # 800200cc: 00021080 sll 02, 02, 02 | 800200cc: 00021080 sll v0,v0,0x2 |
| # 800200d0: 8fc40018 lw 04, 24(1e) | 800200d0: 8fc40018 lw a0,24(s8) |
| # 800200d4: 00821021 addu 02, 04, 02 | 800200d4: 00821021 addu v0,a0,v0 |
| # 800200d8: 8c420000 lw 02, 0(02) | 800200d8: 8c420000 lw v0,0(v0) |
| # 800200dc: 0043102a slt 02, 02, 03 | 800200dc: 0043102a slt v0,v0,v1 |
| # 800200e0: 10400019 beq 02, 00, 0019 | 800200e0: 10400019 beqz v0,80020148 <bubble\_srt+0xc8> |
| # 800200e4: 00000000 NOP | 800200e4: 00000000 nop |
| # 800200e8: 8fc20004 lw 02, 4(1e) | 800200e8: 8fc20004 lw v0,4(s8) |
| # 800200ec: 2442ffff addiu 02, 02, 65535 | 800200ec: 2442ffff addiu v0,v0,-1 |
| # 800200f0: 00021080 sll 02, 02, 02 | 800200f0: 00021080 sll v0,v0,0x2 |
| # 800200f4: 8fc30018 lw 03, 24(1e) | 800200f4: 8fc30018 lw v1,24(s8) |
| # 800200f8: 00621021 addu 02, 03, 02 | 800200f8: 00621021 addu v0,v1,v0 |
| # 800200fc: 8c420000 lw 02, 0(02) | 800200fc: 8c420000 lw v0,0(v0) |
| # 80020100: afc20008 sw 02, 8(1e) | 80020100: afc20008 sw v0,8(s8) |
| # 80020104: 8fc20004 lw 02, 4(1e) | 80020104: 8fc20004 lw v0,4(s8) |
| # 80020108: 2442ffff addiu 02, 02, 65535 | 80020108: 2442ffff addiu v0,v0,-1 |
| # 8002010c: 00021080 sll 02, 02, 02 | 8002010c: 00021080 sll v0,v0,0x2 |
| # 80020110: 8fc30018 lw 03, 24(1e) | 80020110: 8fc30018 lw v1,24(s8) |
| # 80020114: 00621021 addu 02, 03, 02 | 80020114: 00621021 addu v0,v1,v0 |
| # 80020118: 8fc30004 lw 03, 4(1e) | 80020118: 8fc30004 lw v1,4(s8) |
| # 8002011c: 00031880 sll 03, 03, 02 | 8002011c: 00031880 sll v1,v1,0x2 |
| # 80020120: 8fc40018 lw 04, 24(1e) | 80020120: 8fc40018 lw a0,24(s8) |
| # 80020124: 00831821 addu 03, 04, 03 | 80020124: 00831821 addu v1,a0,v1 |
| # 80020128: 8c630000 lw 03, 0(03) | 80020128: 8c630000 lw v1,0(v1) |
| # 8002012c: ac430000 sw 03, 0(02) | 8002012c: ac430000 sw v1,0(v0) |
| # 80020130: 8fc20004 lw 02, 4(1e) | 80020130: 8fc20004 lw v0,4(s8) |
| # 80020134: 00021080 sll 02, 02, 02 | 80020134: 00021080 sll v0,v0,0x2 |
| # 80020138: 8fc30018 lw 03, 24(1e) | 80020138: 8fc30018 lw v1,24(s8) |
| # 8002013c: 00621021 addu 02, 03, 02 | 8002013c: 00621021 addu v0,v1,v0 |
| # 80020140: 8fc30008 lw 03, 8(1e) | 80020140: 8fc30008 lw v1,8(s8) |
| # 80020144: ac430000 sw 03, 0(02) | 80020144: ac430000 sw v1,0(v0) |
| # 80020148: 8fc20004 lw 02, 4(1e) | 80020148: 8fc20004 lw v0,4(s8) |
| # 8002014c: 24420001 addiu 02, 02, 1 | 8002014c: 24420001 addiu v0,v0,1 |
| # 80020150: afc20004 sw 02, 4(1e) | 80020150: afc20004 sw v0,4(s8) |
| # 80020154: 8fc3001c lw 03, 28(1e) | 80020154: 8fc3001c lw v1,28(s8) |
| # 80020158: 8fc20000 lw 02, 0(1e) | 80020158: 8fc20000 lw v0,0(s8) |
| # 8002015c: 00621823 subu 03, 03, 02 | 8002015c: 00621823 subu v1,v1,v0 |
| # 80020160: 8fc20004 lw 02, 4(1e) | 80020160: 8fc20004 lw v0,4(s8) |
| # 80020164: 0043102a slt 02, 02, 03 | 80020164: 0043102a slt v0,v0,v1 |
| # 80020168: 1440ffd1 bne 02, 00, ffd1 | 80020168: 1440ffd1 bnez v0,800200b0 <bubble\_srt+0x30> |
| # 8002016c: 00000000 NOP | 8002016c: 00000000 nop |
| # 80020170: 8fc20000 lw 02, 0(1e) | 80020170: 8fc20000 lw v0,0(s8) |
| # 80020174: 24420001 addiu 02, 02, 1 | 80020174: 24420001 addiu v0,v0,1 |
| # 80020178: afc20000 sw 02, 0(1e) | 80020178: afc20000 sw v0,0(s8) |
| # 8002017c: 8fc30000 lw 03, 0(1e) | 8002017c: 8fc30000 lw v1,0(s8) |
| # 80020180: 8fc2001c lw 02, 28(1e) | 80020180: 8fc2001c lw v0,28(s8) |
| # 80020184: 0062102a slt 02, 03, 02 | 80020184: 0062102a slt v0,v1,v0 |
| # 80020188: 1440ffc5 bne 02, 00, ffc5 | 80020188: 1440ffc5 bnez v0,800200a0 <bubble\_srt+0x20> |
| # 8002018c: 00000000 NOP | 8002018c: 00000000 nop |
| # 80020190: 03c0e821 addu 1d, 1e, 00 | 80020190: 03c0e821 move sp,s8 |
| # 80020194: 8fbe0014 lw 1e, 20(1d) | 80020194: 8fbe0014 lw s8,20(sp) |
| # 80020198: 27bd0018 addiu 1d, 1d, 24 | 80020198: 27bd0018 addiu sp,sp,24 |
| # 8002019c: 03e00008 jr 1f | 8002019c: 03e00008 jr ra |
| # 800201a0: 00000000 NOP | 800201a0: 00000000 nop |

Table 1. Comparison table for BubbleSort disassembly. The left column shows the output from out processor test bench while the right shows the actual disable from the Bubble Sort benchmark.