Spring
2015

MyRISC
A VHDL implementation of a 16-bit RISC processor

Final Project for ENGS 128: Advanced Digital Systems Design

Professor Hansen

BRETT R. NICHOLAS
MATTHEW T. METZLER

Table of Contents

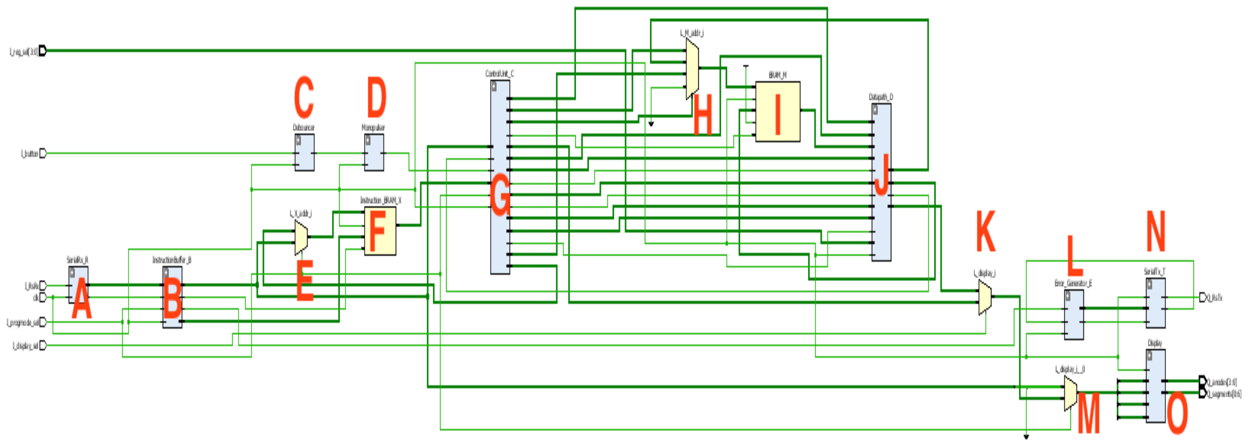
Project Introduction

Overview

In the project we have designed a simple reduced instruction set architecture (RISC) microprocessor implemented on a field-programmable gate array (FPGA). This processor executes 16-bit instructions stored in instruction random access memory (RAM) one at a time using 16 general purpose registers and a general purpose block RAM. We began by storing our instructions in read-only memory, but after completing the processing functionality of the design, we decided to allow user interaction with the processor by allowing real-time programming through serial communication with the board. The user can type instructions in hexadecimal format into *PuTTy* to program the board, and then step through program execution one instruction at a time using a push button.

High-Level Overview

Top-Level Block Diagram



Key	Component	Key	Component
A	**Serial Receiver**	**I**	**General Purpose RAM**
B	**Instruction Buffer**	**J**	**Datapath**
C	**Debouncer**	**K**	**7-segment display MUX1**
D	**Monopulser**	**L**	**Error Generator**
E	**Instruction RAM Address MUX**	**M**	**7-segment display MUX2**
F	**Instruction RAM**	**N**	**Serial Transmitter**
G	**Control Unit**	**O**	**7-segment Display**
H	**General Purpose RAM Address MUX**		

Functionality

Control Unit

The control unit consists of an FSM controller, four special purpose registers, and a small amount of dedicated combinational logic. The FSM controller guides the processor through the various generic states of program execution – *Fetch, Decode, Load Operands, Execute, Store* – and also handles other component's access to memory. The processor utilizes a "Harvard Architecture", which means instructions are stored in a separate memory from the general purpose memory, so the controller must handle simultaneous communication with both memories. The four special purpose registers contained in the control unit are the Program Counter (PC), the Instruction Register (IR), the Link Register (LR), and the Stack Pointer (SP). A more detailed description of controller functionality is provided in the Component Architecture section.

Datapath

The Datapath consists of a 16x16-bit multipurpose register file with two pipelined outputs, an Arithmetic Logic Unit (ALU), and various input and output multiplexors. The controller is responsible for selecting from the register file the appropriate registers from which to read and write. Any two of the 16 registers can be multiplexed into the two designated read pipeline registers (`Preg` and `Qreg`), and any one of the 16 registers can be written to from a number of different sources as designated by the controller. A more detailed description of Datapath functionality is provided in the Component Architecture section.

Instruction Buffer

The instruction buffer is a state machine-controlled character buffer that allows for the user to program the instruction memory (XRAM) over USART. The instruction buffer shifts each received character from the serial receiver into the buffer until a carriage return or error is detected. When a valid instruction is fully buffered, it is then written to instruction memory, and the buffer is ready to accept the next command. When programming is complete, the buffer stores the memory address of the last instruction so that the processor can halt execution after the final instruction. A more detailed description of the Instruction Buffer functionality is provided in the Component Architecture section.

General Purpose RAM (BRAM)

For general purpose memory, the processor employs a single-port read/write-enable 18k BRAM, instantiated as a Xilinx IP core using the Block Memory Generator. Memory access is handled by the control unit. A more detailed description of BRAM functionality is provided in the Component Architecture section.

Instruction RAM (XRAM)

For instruction memory, the processor employs another single-port read/write-enable 18k BRAM, instantiated as a Xilinx IP core using the Block Memory Generator. Memory is programmable by the user over USART, and access is shared by the control unit and the Instruction Buffer. A more detailed description of the XRAM functionality is provided in the Component Architecture section.

Serial Receiver & Transmitter

These pre-built components interact with the RS-232 serial port for serial communication between the board and the user

Debouncer & Monopulser

The debouncer uses a counter and output register to filter out electrical noise during push button operation. The debounced signal is then fed into a monopulser that creates a one clock-width tick on any rising edge of the signal.

7-segment Display

The display driver controls the four 7-segment displays on the board. It does this by cycling through the four anodes with the corresponding 4-bit vector.

Top-level Port Descriptions

Port	Type	Peripheral Connectivity	Description
`clk`	std_logic		100 MHz system clock
`I_RsRx`	std_logic	USB-RS232 (pkg pin B18)	Serial data in
`I_progmode_sel`	std_logic	Switch SW0 (pkg pin V17)	Allows user to toggle between program mode and execute mode
`I_button`	std_logic	Center board button BTNC (pkg pin U18)	Allows user to step through program execution, one instruction at a time
`I_reg_sel`	std_logic_vector (3 downto 0)	Switches SW15 downto SW12 (pkg pins R2, T1, U1, W2)	Allows user to select which register's contents are displayed on the 7-segment LED display
`Q_RsTx`	std_logic	USB-RS232 (pkg pin A18)	Serial data out
`Q_segments`	std_logic_vector (7 downto 0)	7-segment display	7-segment LED controls
`Q_anodes`	std_logic_vector (3 downto 0)	7-segment display	7-segment LED controls

Functional Description

Modes of operation

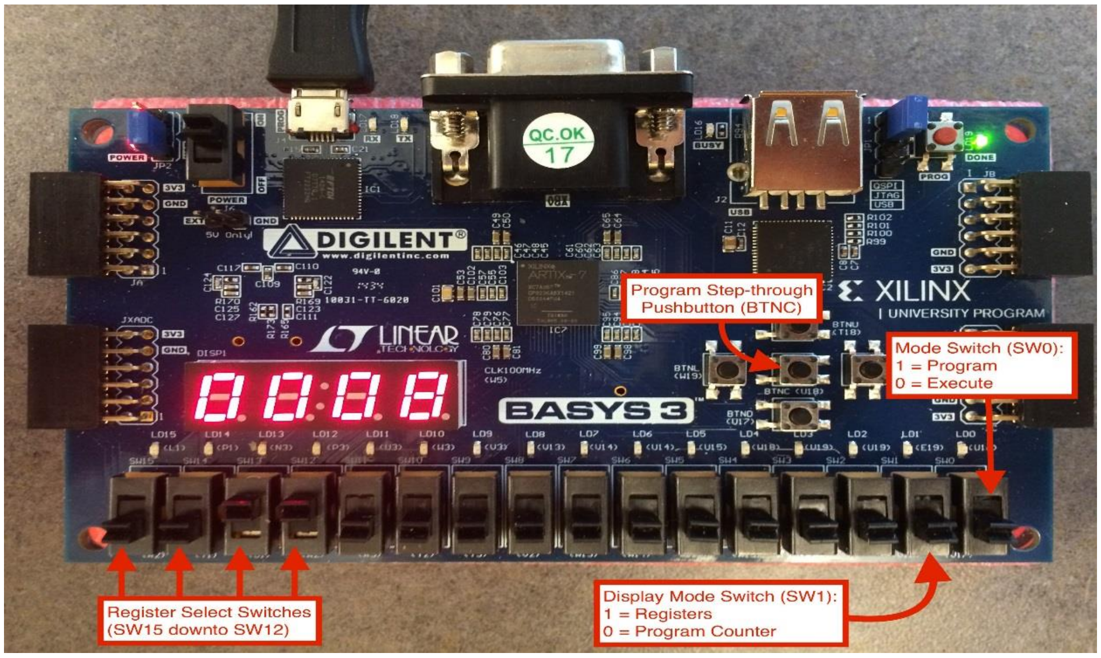
The processor can be operated in two modes: *program* mode and *execution* mode. The user can toggle between these two modes using a slide switch. P*rogram* mode, accessed by setting **SW0** to **HIGH**, allows users to enter instructions over USART which are then stored in instruction memory when the mode is changed. *Execution* mode, entered by setting **SW0** to **LOW,** waits for the user to press the center button (**BTNC**), upon which it will load the next instruction into the instruction register, execute the instruction, and then either wait for the next button press to continue or terminate execution when the last instruction is reached.

Program Mode

In the *program* mode the user enters processor instructions into *PuTTy* to be transmitted over the serial interface on the board. These commands, entered as four-character hexadecimal words followed by a carriage return (CR), are converted into instructions and saved to the instruction memory. The 7-segment display on the board displays which address in the instruction memory is currently being written to. The processor returns an "ERROR" message over the serial interface when the entered command word does not match the format specified below, and the command is ignored.

Execution Mode

In *execution* mode the user can step through the instructions using a pushbutton on the board. The user can use an array of switch slides to control the data shown on the 7-segment LED displays. The **SW1** switch toggles between displaying the program counter and the register file. When looking at the register file, four separate switch slides designate the binary address of the register the user wants to look at. While looking at the program counter on the display, the instruction held in that spot in memory is not executed until the user presses the pushbutton. The user can continue executing instructions until all the instructions read in the previous *program* mode have been performed, after which the processor idles until the user returns **SW0** to **LOW** to enter new commands.


Program Step-through Pushbutton (BTNC)
Mode Switch (SW0): 1 = Program 0 = Execute
Register Select Switches (SW15 downto SW12)
Display Mode Switch (SW1): 1 = Registers 0 = Program Counter

Instructions

Instruction Set

Command	Description	Opcode		Operation
NOP	No operation	0000	x“0”	N/A
LDR	Load direct	0001	x“1”	R_a <= M[d]
STR	Store direct	0010	x“2”	M[d] <= R_a
MOV	Move	0011	x“3”	R_a <= R_b
LDC	Load constant	0100	x“4”	R_a <= d
ADD	Add	0101	x“5”	R_a <= R_b + R_c
SUB	Subtract	0110	x“6”	R_a <= R_b – R_c
BZ	Branch if zero	0111	x“7”	if(R_a = 0) then PC <= PC + d + 1
BD	Branch direct	1000	x“8”	PC <= d
BL	Branch with link	1001	x“9”	LR <= PC and PC <= d
LIN	Load indirect	1010	x“a”	R_a <= M[R_c]
SIN	Store indirect	1011	x“b”	M[R_c] <= R_b
PSH	Push registers to stack	1100	x“c”	Push(v); v is one-hot register selection vector ‘*PSH 010101010101*’ pushes r0, r2, r4,r6, r8, r10 onto stack
POP	Pop registers off stack	1101	x“d”	Pop(v); v is one-hot register selection vector ‘*POP 010101010101*’ pops r0, r2, r4,r6, r8, r10 from stack, assuming registers were pushed using the same order

Instruction Format

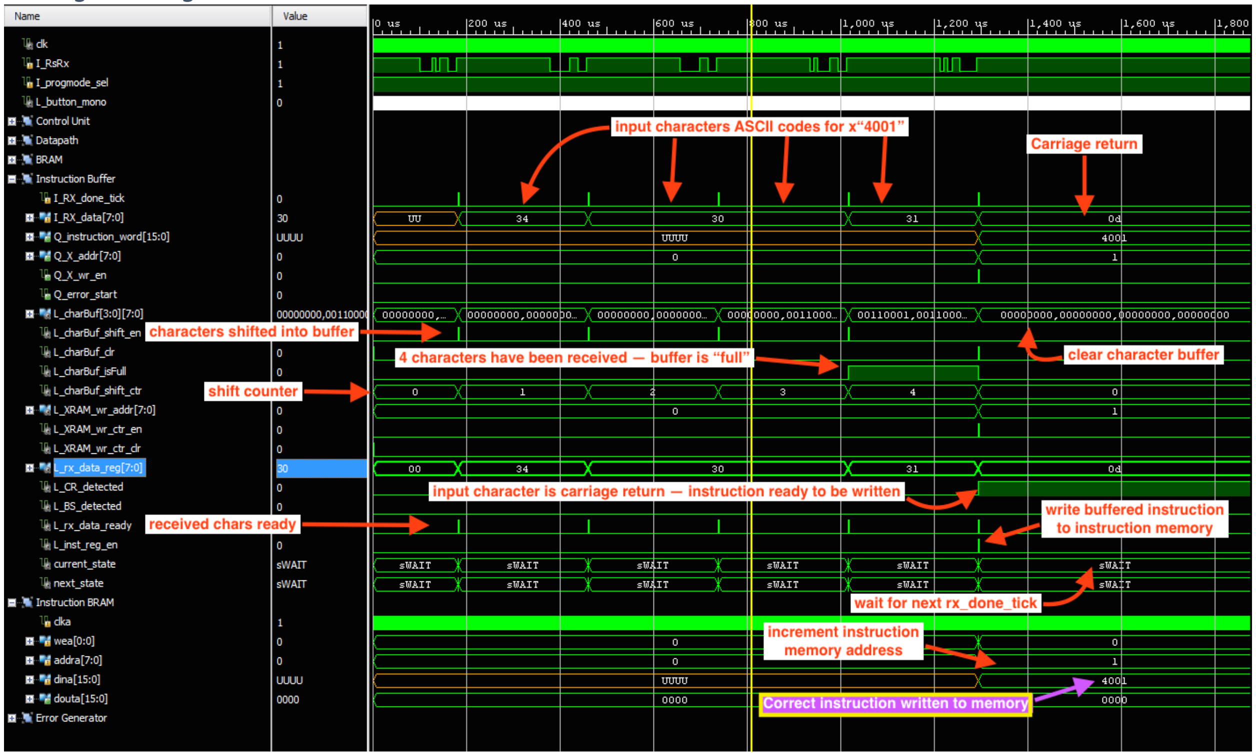
3 instruction types:

(15 downto 12)	(11 downto 8)	(7 downto 4)	(3 downto 0)
< Opcode >	**< R_a >**	**< R_b >**	**< R_c >**
	< R_a >	**< d >**	
	< Register Selection Vector >		

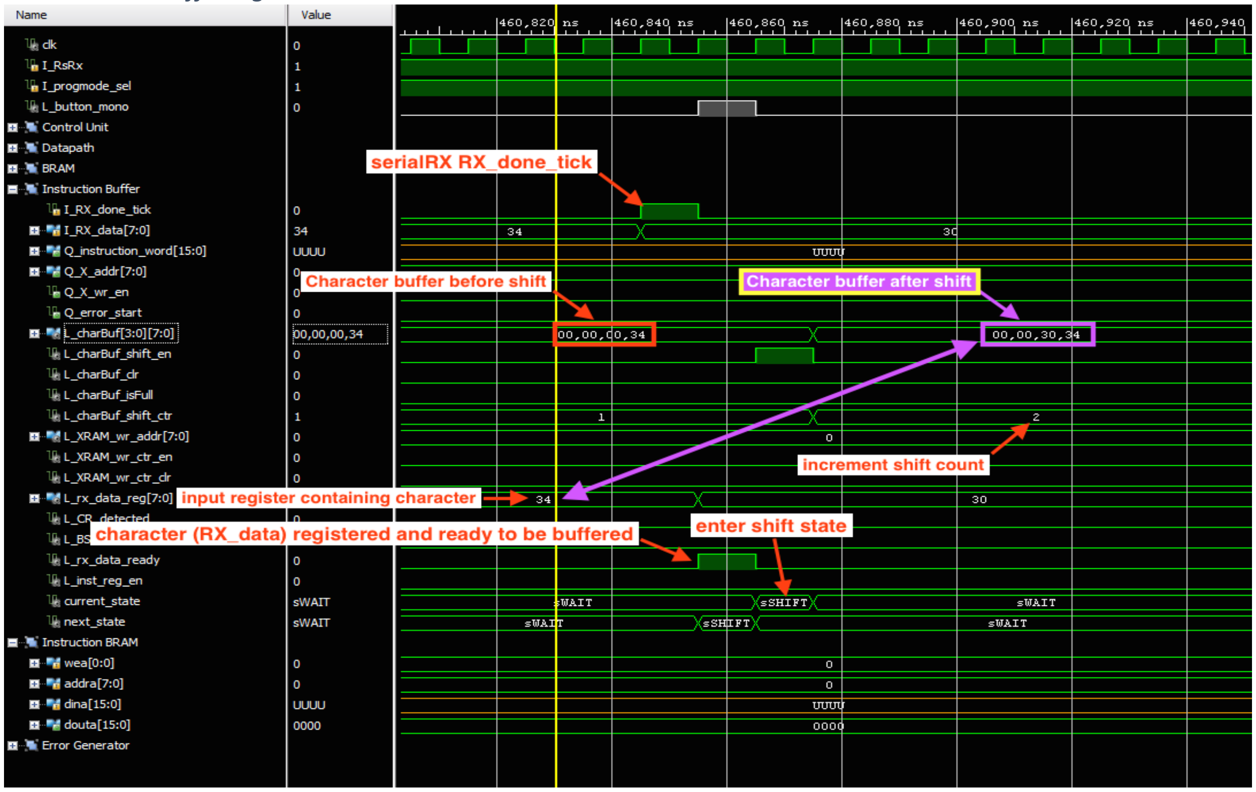
Simulation Waveforms

Program Mode

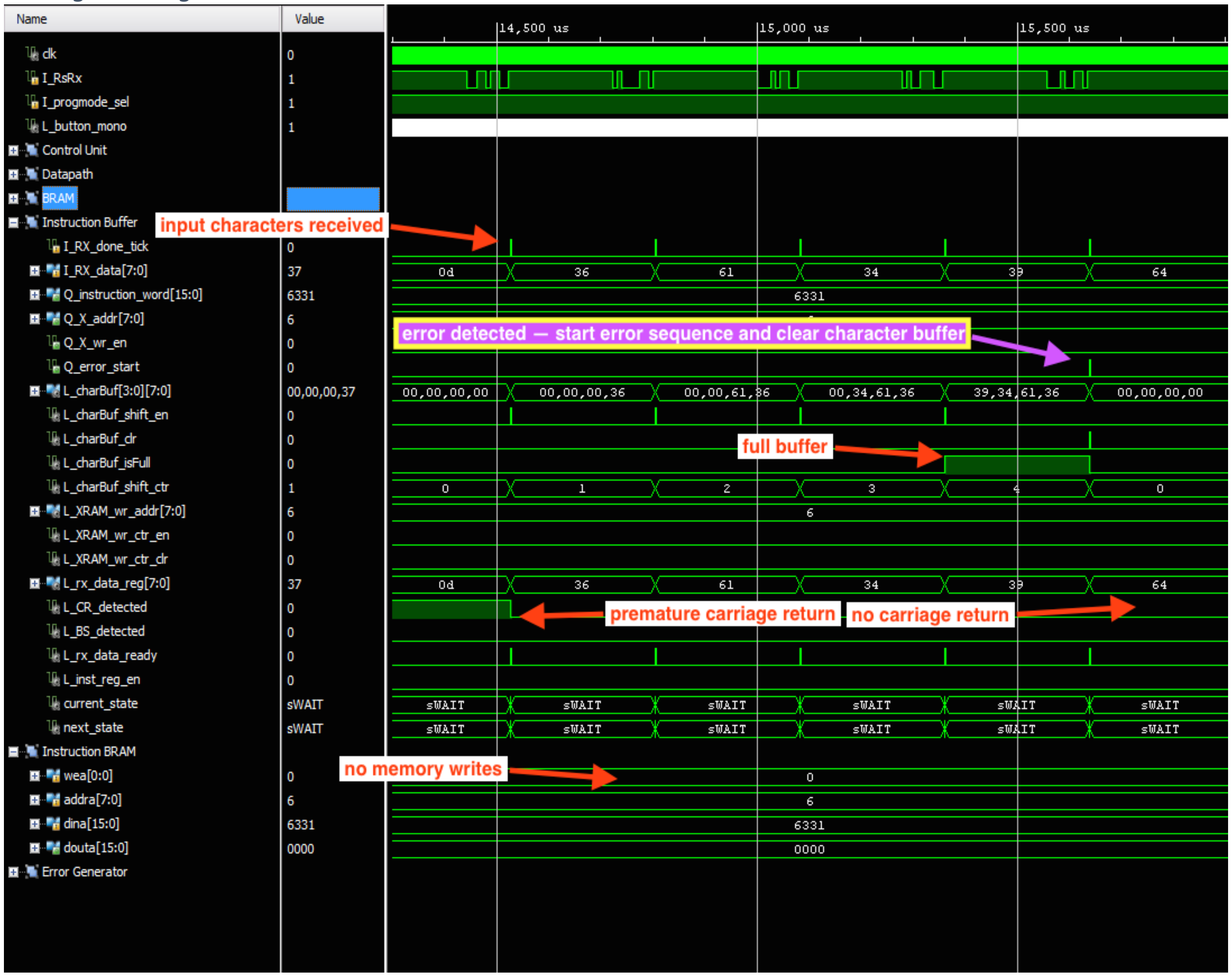
1: Programming XRAM



2: Character Buffering

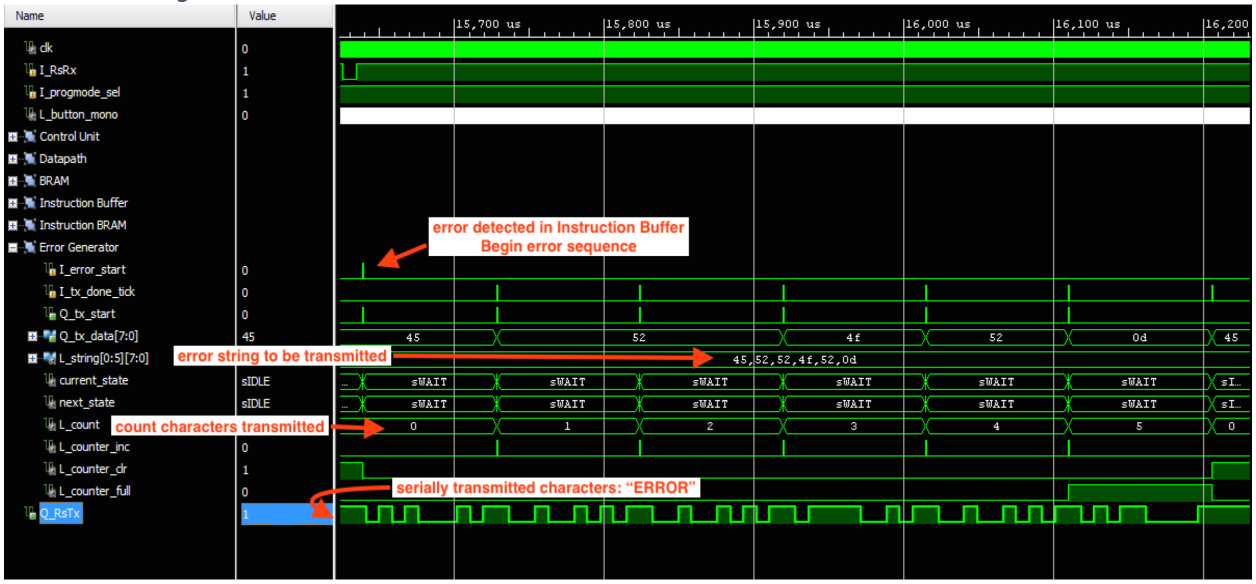


3: Programming Error Detection



Name	Value
clk	0
I_RsRx	1
I_progmode_sel	1
L_button_mono	1
Control Unit	
Datapath	
BRAM	
Instruction Buffer	
I_RX_done_tick	0
I_RX_data[7:0]	37
Q_instruction_word[15:0]	6331
Q_X_addr[7:0]	6
Q_X_wr_en	0
Q_error_start	0
L_charBuf[3:0][7:0]	00,00,00,37
L_charBuf_shift_en	0
L_charBuf_clr	0
L_charBuf_isFull	0
L_charBuf_shift_ctr	1
L_XRAM_wr_addr[7:0]	6
L_XRAM_wr_ctr_en	0
L_XRAM_wr_ctr_clr	0
L_rx_data_reg[7:0]	37
L_CR_detected	0
L_BS_detected	0
L_rx_data_ready	0
L_inst_reg_en	0
current_state	sWAIT
next_state	sWAIT
Instruction BRAM	
wea[0:0]	0
addra[7:0]	6
dina[15:0]	6331
douta[15:0]	0000
Error Generator	

4: Error Message Transmission

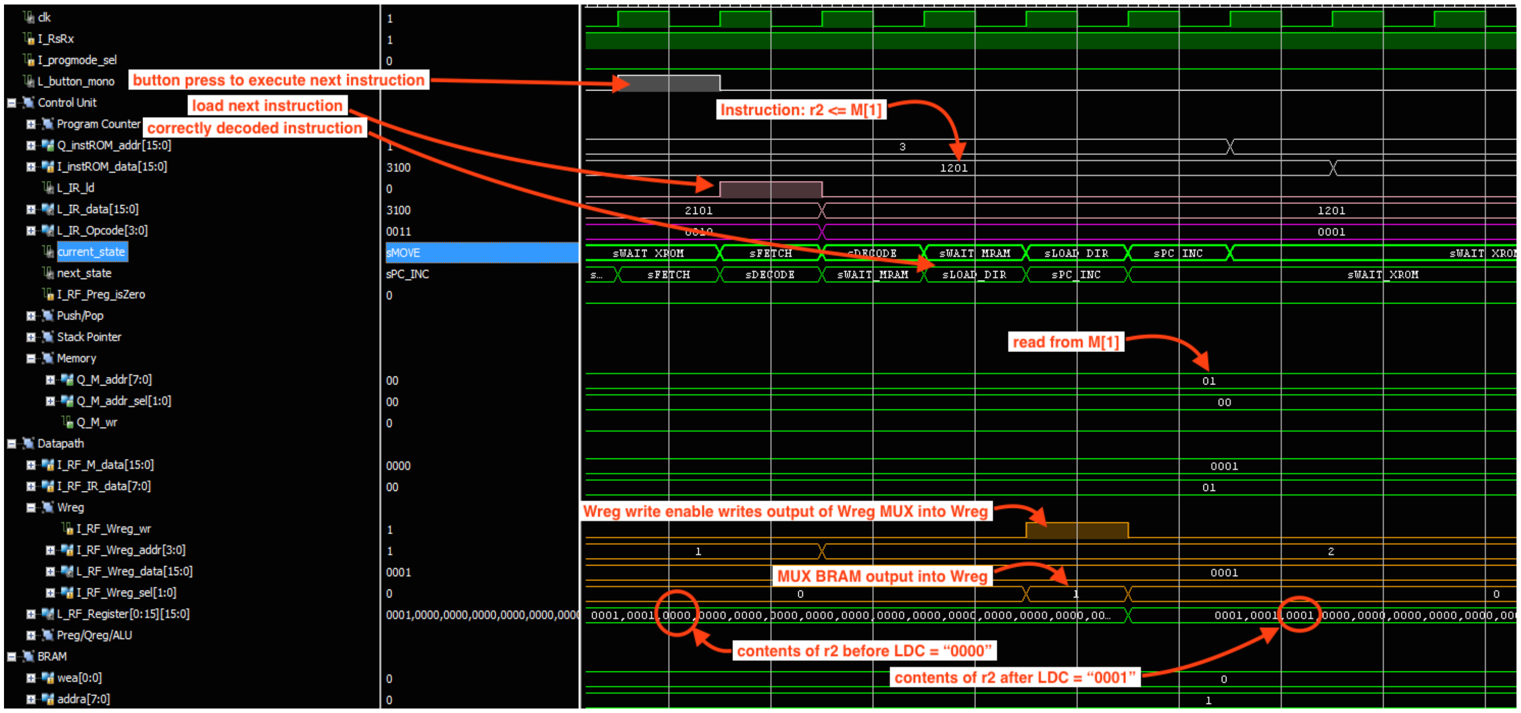


Name	Value
clk	0
I_RsRx	1
I_progmode_sel	1
L_button_mono	0
Control Unit	
Datapath	
BRAM	
Instruction Buffer	
Instruction BRAM	
Error Generator	
I_error_start	0
I_tx_done_tick	0
Q_tx_start	0
Q_tx_data[7:0]	45
L_string[0:5][7:0]	
current_state	sIDLE
next_state	sIDLE
L_count	0
L_counter_inc	0
L_counter_clr	1
L_counter_full	0
Q_RsTx	1

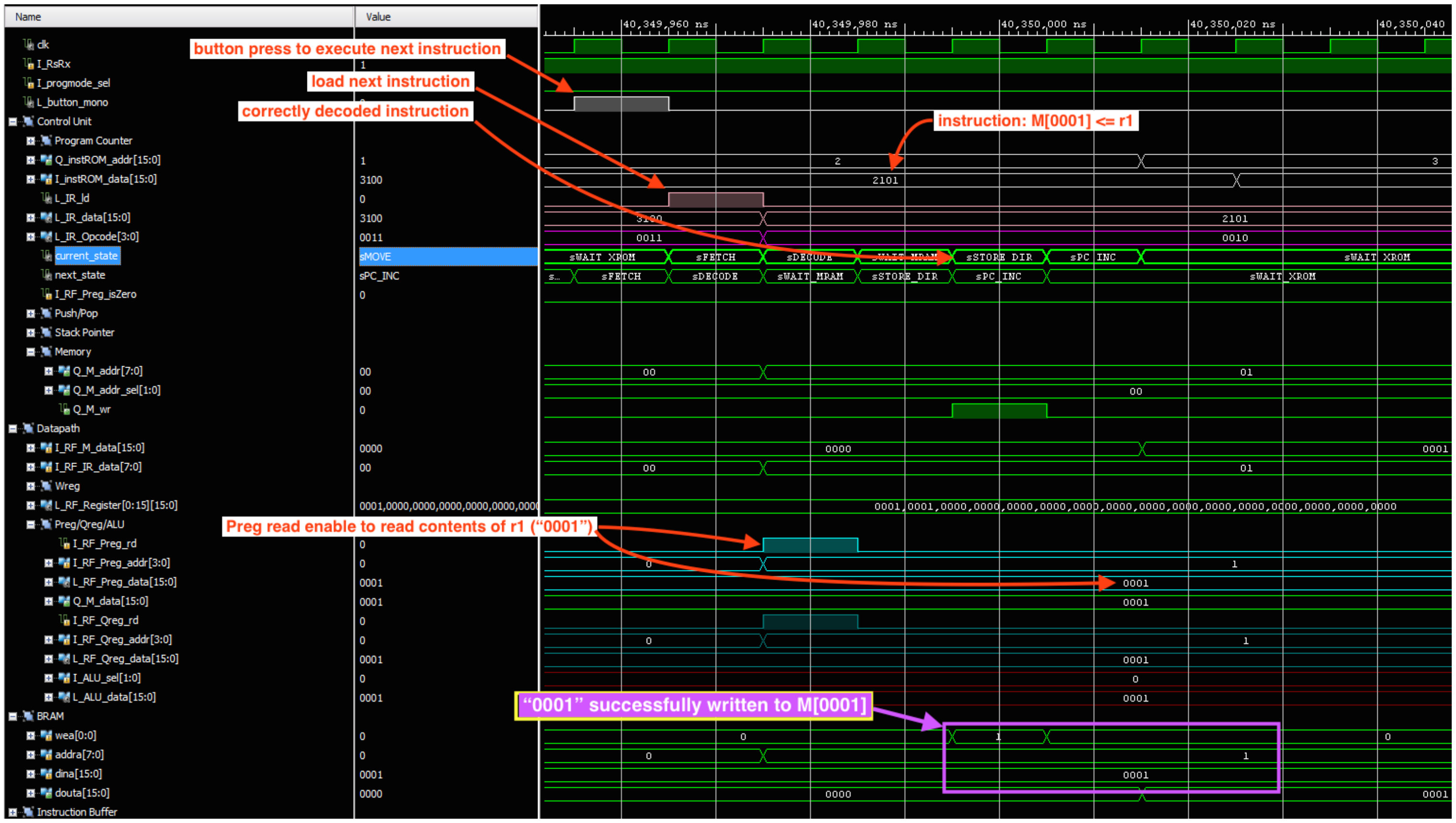
15,700 us
15,800 us
15,900 us
16,000 us
16,100 us
16,200
error detected in Instruction Buffer
Begin error sequence
45
52
52
4f
52
0d
45
error string to be transmitted
45,52,52,4f,52,0d
sWAIT
count characters transmitted
0
1
2
3
4
5
0
serially transmitted characters: "ERROR"

Execution Mode

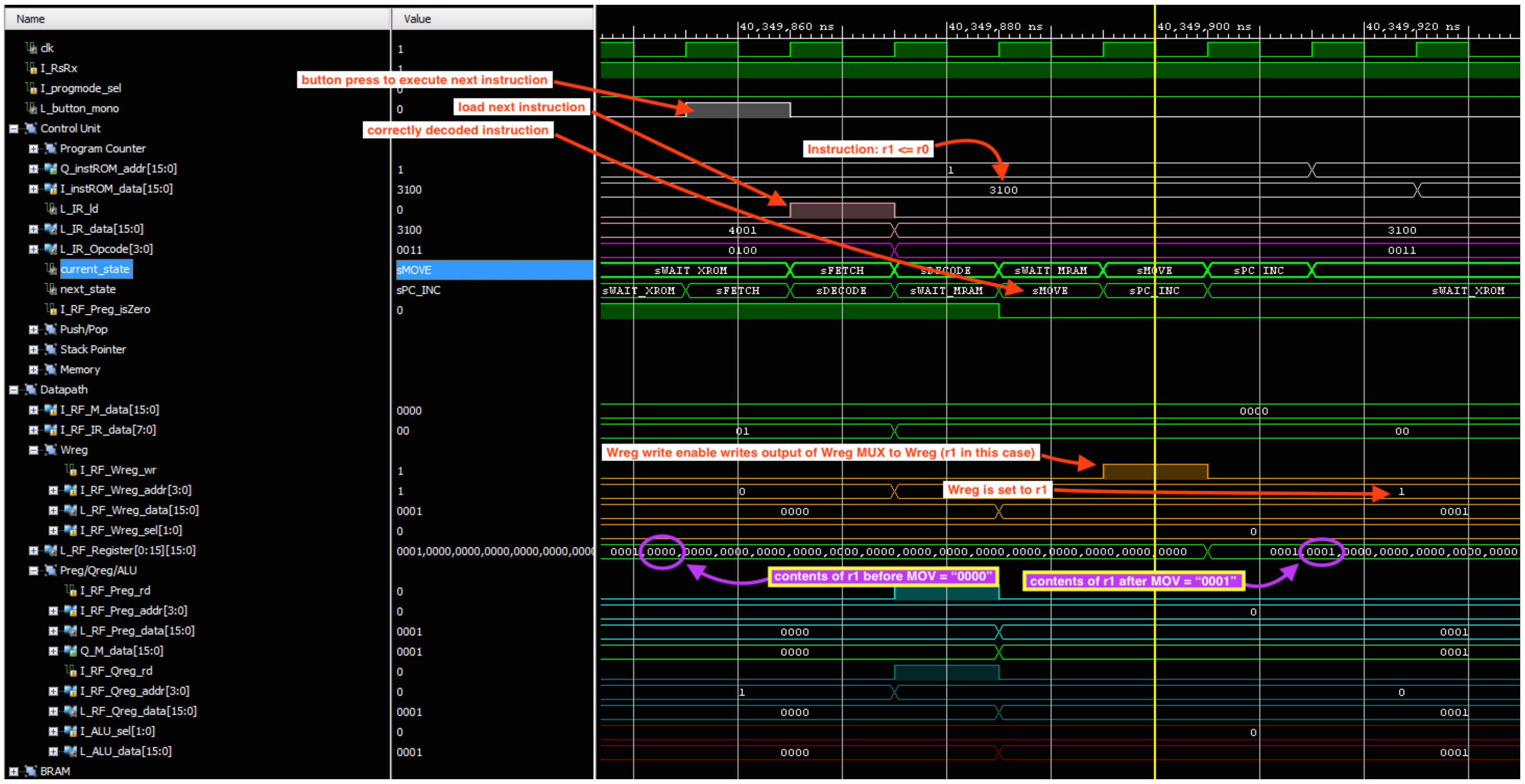
5: Load Direct


button press to execute next instruction
load next instruction
correctly decoded instruction
Instruction: r2 <= M[1]
read from M[1]
Wreg write enable writes output of Wreg MUX into Wreg
MUX BRAM output into Wreg
contents of r2 before LDC = "0000"
contents of r2 after LDC = "0001"

6: Store Direct


button press to execute next instruction
load next instruction
correctly decoded instruction
instruction: M[0001] <= r1
Preg read enable to read contents of r1 ("0001")
"0001" successfully written to M[0001]

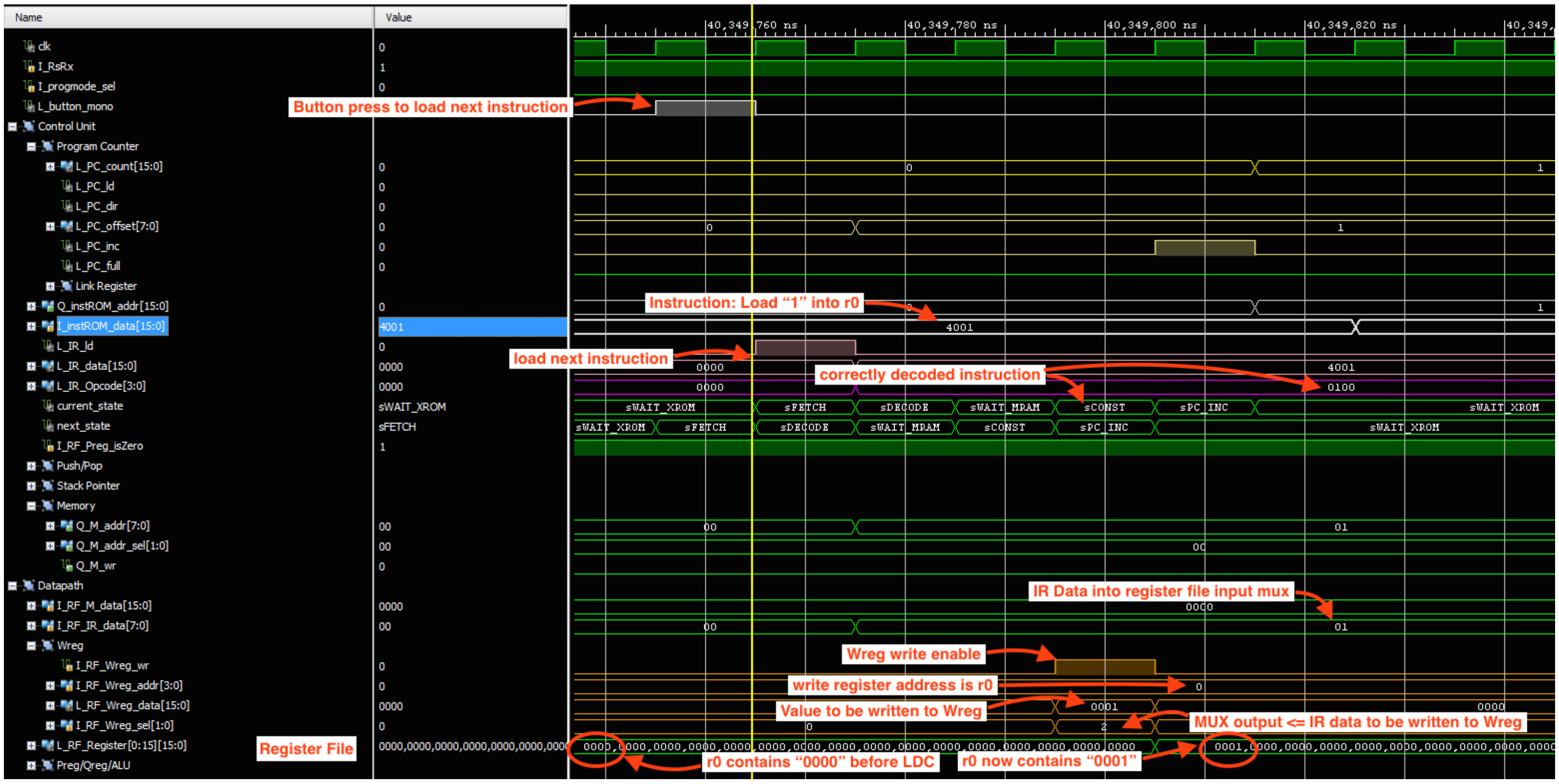
7: Move



Name	Value
clk	1
I_RsRx	1
I_progmode_sel	0
L_button_mono	0
Control Unit	
Program Counter	
Q_instROM_addr[15:0]	1
I_instROM_data[15:0]	3100
L_IR_ld	0
L_IR_data[15:0]	3100
L_IR_Opcode[3:0]	0011
current_state	sMOVE
next_state	sPC_INC
I_RF_Preg_isZero	0
Push/Pop	
Stack Pointer	
Memory	
Datapath	
I_RF_M_data[15:0]	0000
I_RF_IR_data[7:0]	00
Wreg	
I_RF_Wreg_wr	1
I_RF_Wreg_addr[3:0]	1
L_RF_Wreg_data[15:0]	0001
I_RF_Wreg_sel[1:0]	0
L_RF_Register[0:15][15:0]	0001,0000,0000,0000,0000,0000,0000,000
Preg/Qreg/ALU	
I_RF_Preg_rd	0
I_RF_Preg_addr[3:0]	0
L_RF_Preg_data[15:0]	0001
Q_M_data[15:0]	0001
I_RF_Qreg_rd	0
I_RF_Qreg_addr[3:0]	0
L_RF_Qreg_data[15:0]	0001
I_ALU_sel[1:0]	0
L_ALU_data[15:0]	0001
BRAM	

40,349,860 ns
40,349,880 ns
40,349,900 ns
40,349,920 ns
button press to execute next instruction
load next instruction
correctly decoded instruction
Instruction: r1 <= r0
Wreg write enable writes output of Wreg MUX to Wreg (r1 in this case)
Wreg is set to r1
contents of r1 before MOV = "0000"
contents of r1 after MOV = "0001"
sWAIT_XROM
sFETCH
sDECODE
sWAIT_MRAM
sMOVE
sPC_INC

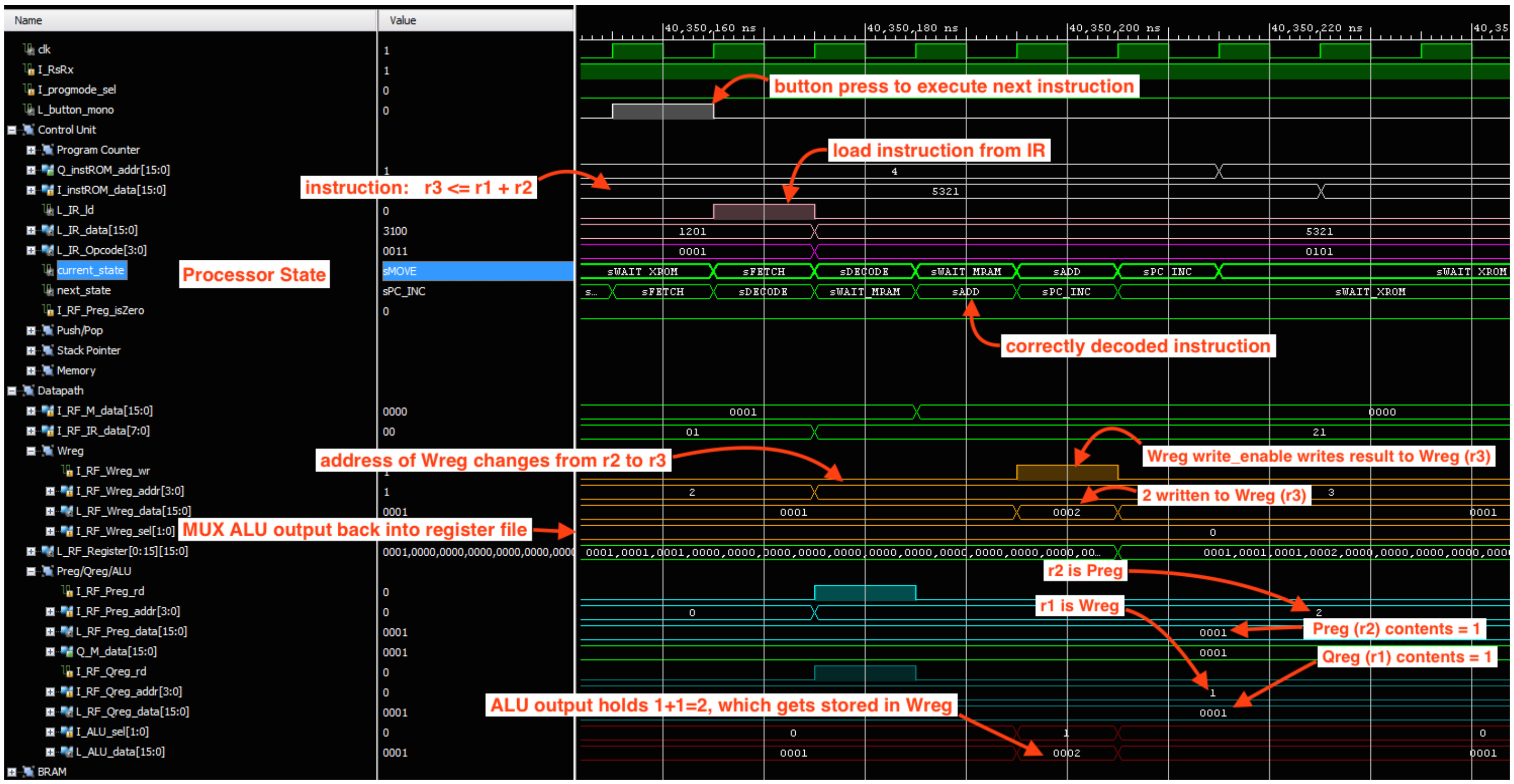
8: Load Constant



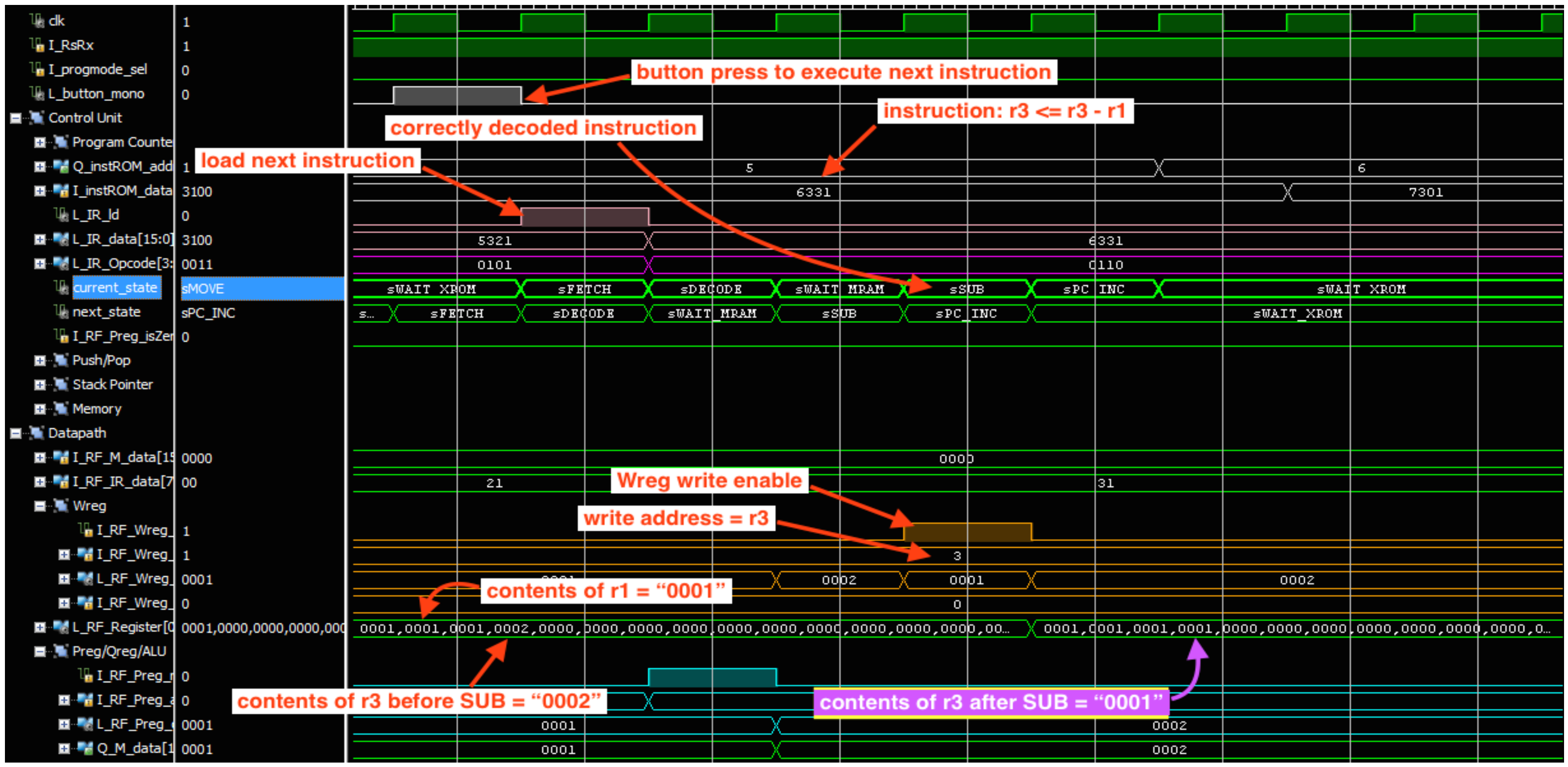
Name	Value
clk	0
I_RsRx	1
I_progmode_sel	0
L_button_mono	0
Control Unit	
Program Counter	
L_PC_count[15:0]	0
L_PC_ld	0
L_PC_dir	0
L_PC_offset[7:0]	0
L_PC_inc	0
L_PC_full	0
Link Register	
Q_instROM_addr[15:0]	0
I_instROM_data[15:0]	4001
L_IR_ld	0
L_IR_data[15:0]	0000
L_IR_Opcode[3:0]	0000
current_state	sWAIT_XROM
next_state	sFETCH
I_RF_Preg_isZero	1
Push/Pop	
Stack Pointer	
Memory	
Q_M_addr[7:0]	00
Q_M_addr_sel[1:0]	00
Q_M_wr	0
Datapath	
I_RF_M_data[15:0]	0000
I_RF_IR_data[7:0]	00
Wreg	
I_RF_Wreg_wr	0
I_RF_Wreg_addr[3:0]	0
L_RF_Wreg_data[15:0]	0000
I_RF_Wreg_sel[1:0]	0
L_RF_Register[0:15][15:0]	0000,0000,0000,0000,0000,0000,0000,000
Preg/Qreg/ALU	

40,349,760 ns
40,349,780 ns
40,349,800 ns
40,349,820 ns
Button press to load next instruction
Instruction: Load "1" into r0
load next instruction
correctly decoded instruction
IR Data into register file input mux
Wreg write enable
write register address is r0
Value to be written to Wreg
MUX output <= IR data to be written to Wreg
Register File
r0 contains "0000" before LDC
r0 now contains "0001"
sWAIT_XROM
sFETCH
sDECODE
sWAIT_MRAM
sCONST
sPC_INC

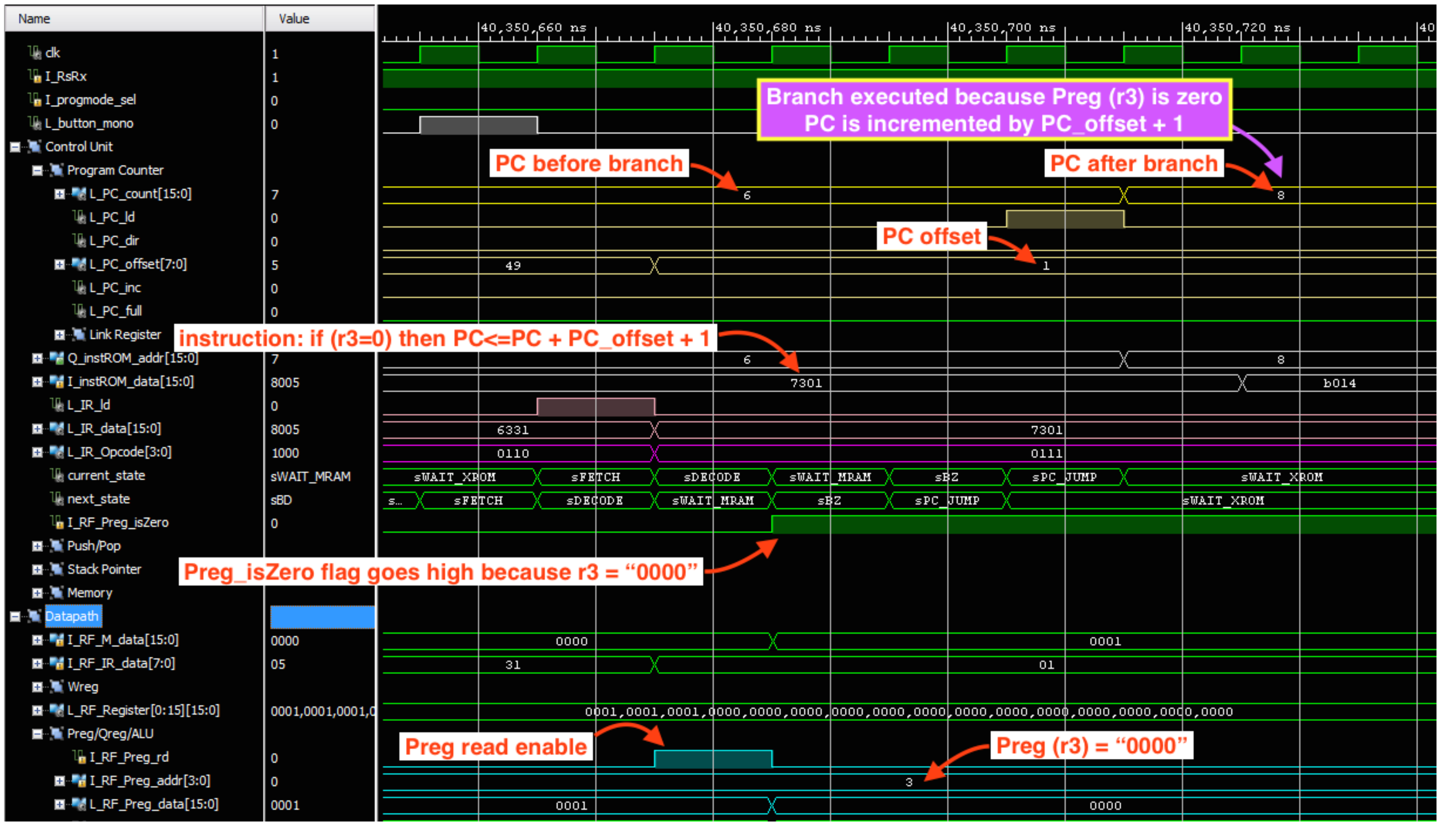
9: Add


button press to execute next instruction
load instruction from IR
instruction: r3 <= r1 + r2
Processor State
correctly decoded instruction
address of Wreg changes from r2 to r3
Wreg write_enable writes result to Wreg (r3)
2 written to Wreg (r3)
MUX ALU output back into register file
r2 is Preg
r1 is Wreg
Preg (r2) contents = 1
Qreg (r1) contents = 1
ALU output holds 1+1=2, which gets stored in Wreg
sWAIT_XROM
sFETCH
sDECODE
sWAIT_MRAM
sADD
sPC_INC

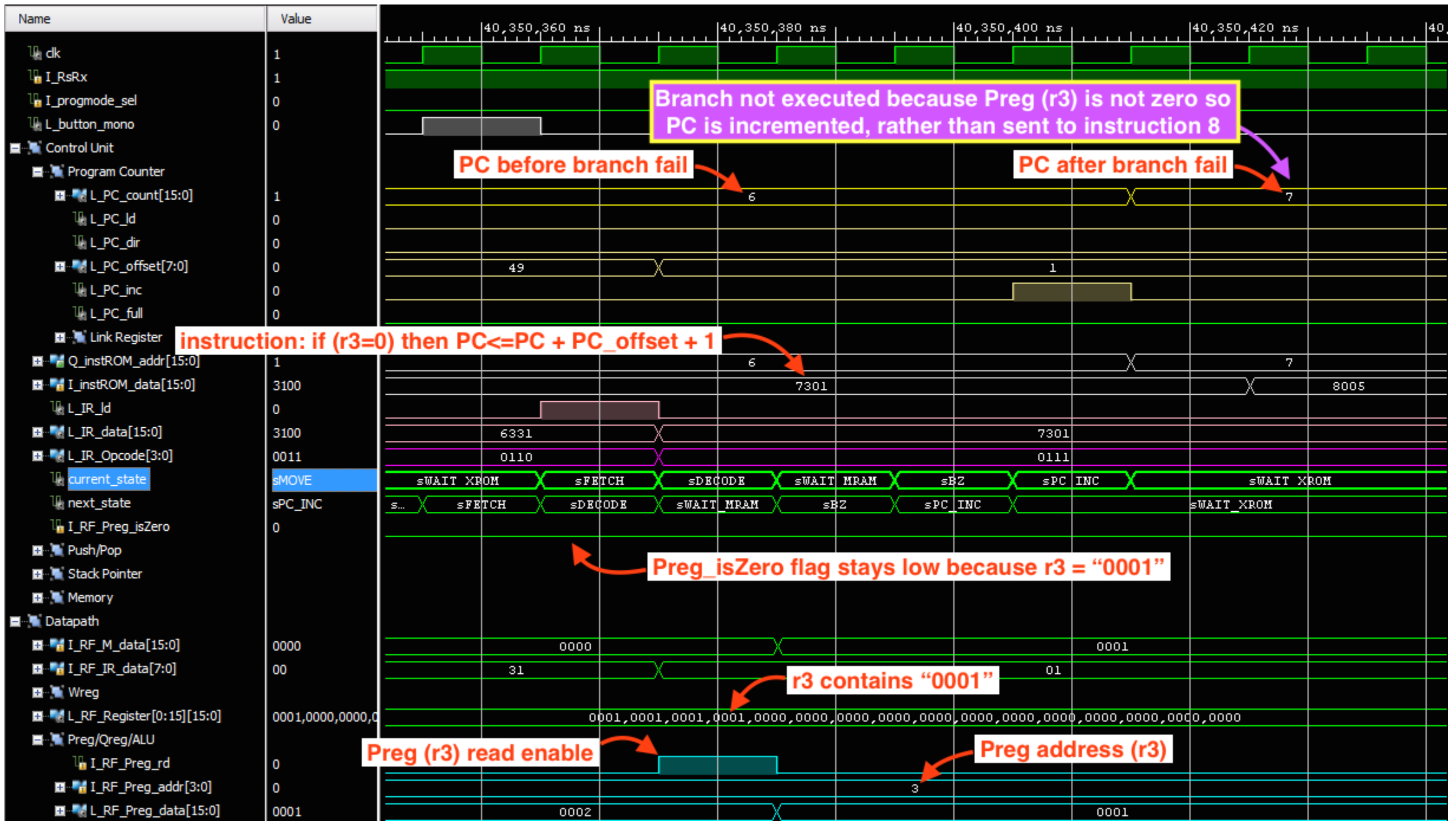
10: Subtract


button press to execute next instruction
instruction: r3 <= r3 - r1
correctly decoded instruction
load next instruction
Wreg write enable
write address = r3
contents of r1 = "0001"
contents of r3 before SUB = "0002"
contents of r3 after SUB = "0001"
sWAIT_XROM
sFETCH
sDECODE
sWAIT_MRAM
sSUB
sPC_INC

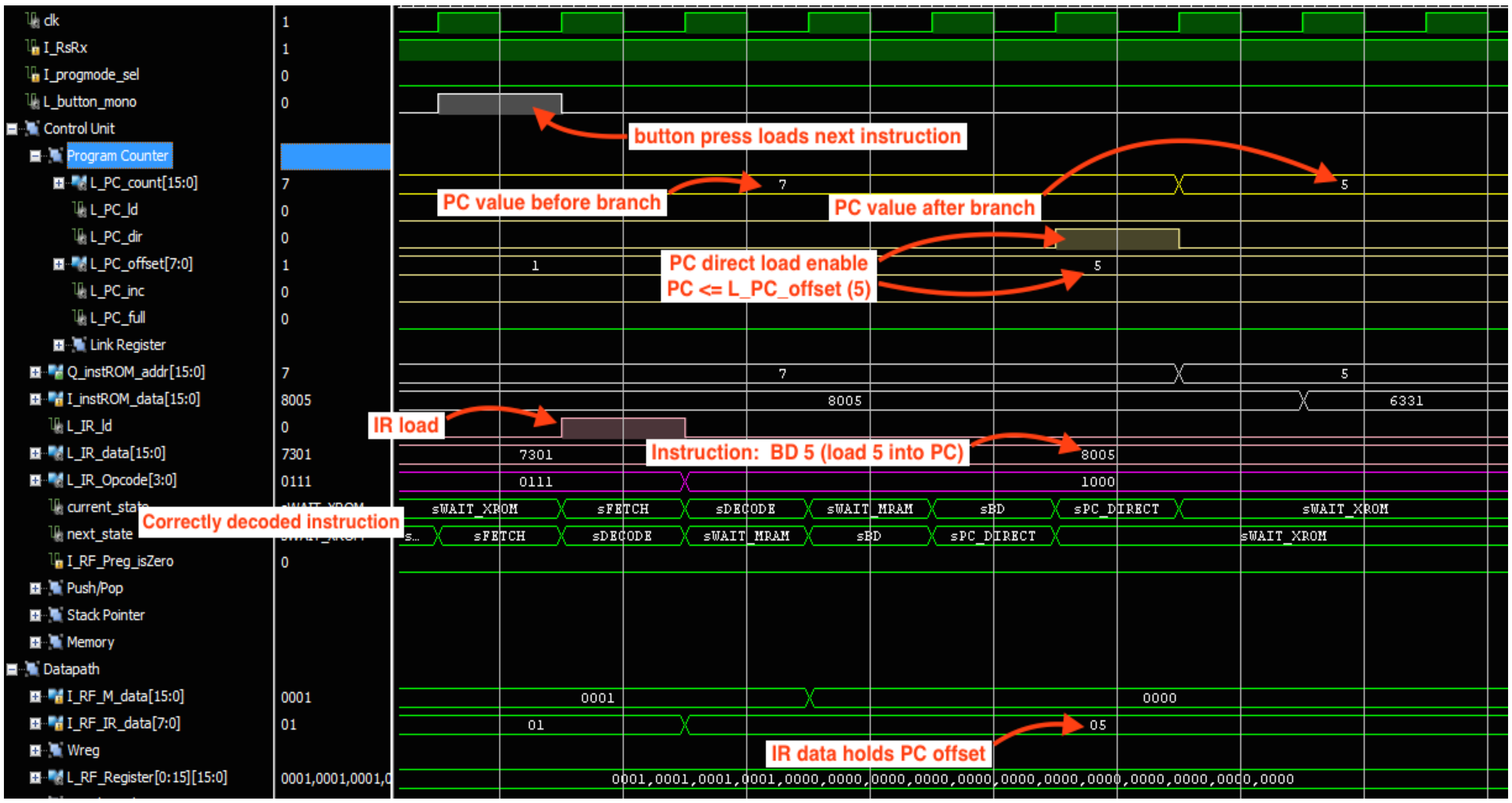
11: Branch if Zero (Successful branch)



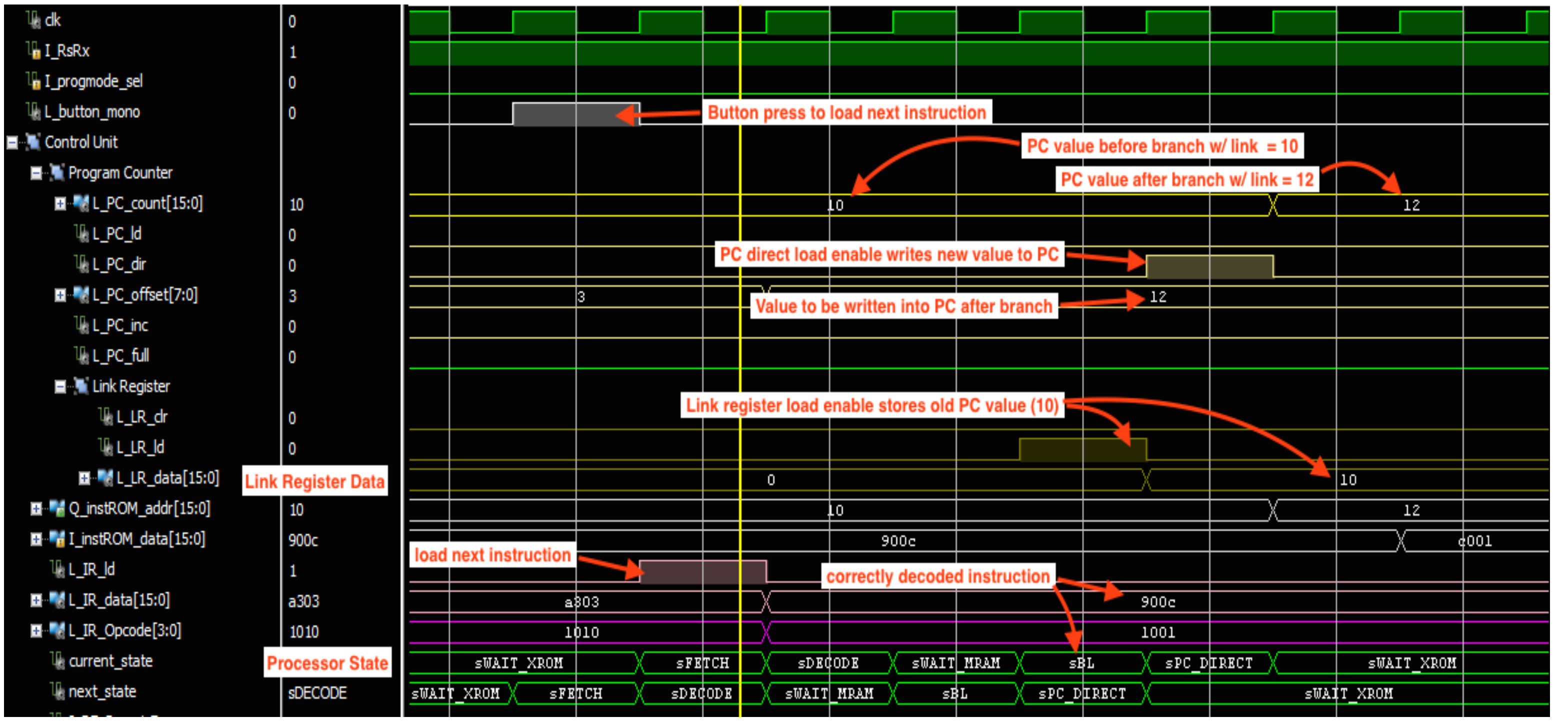
12: Branch if Zero (Failed Attempt)



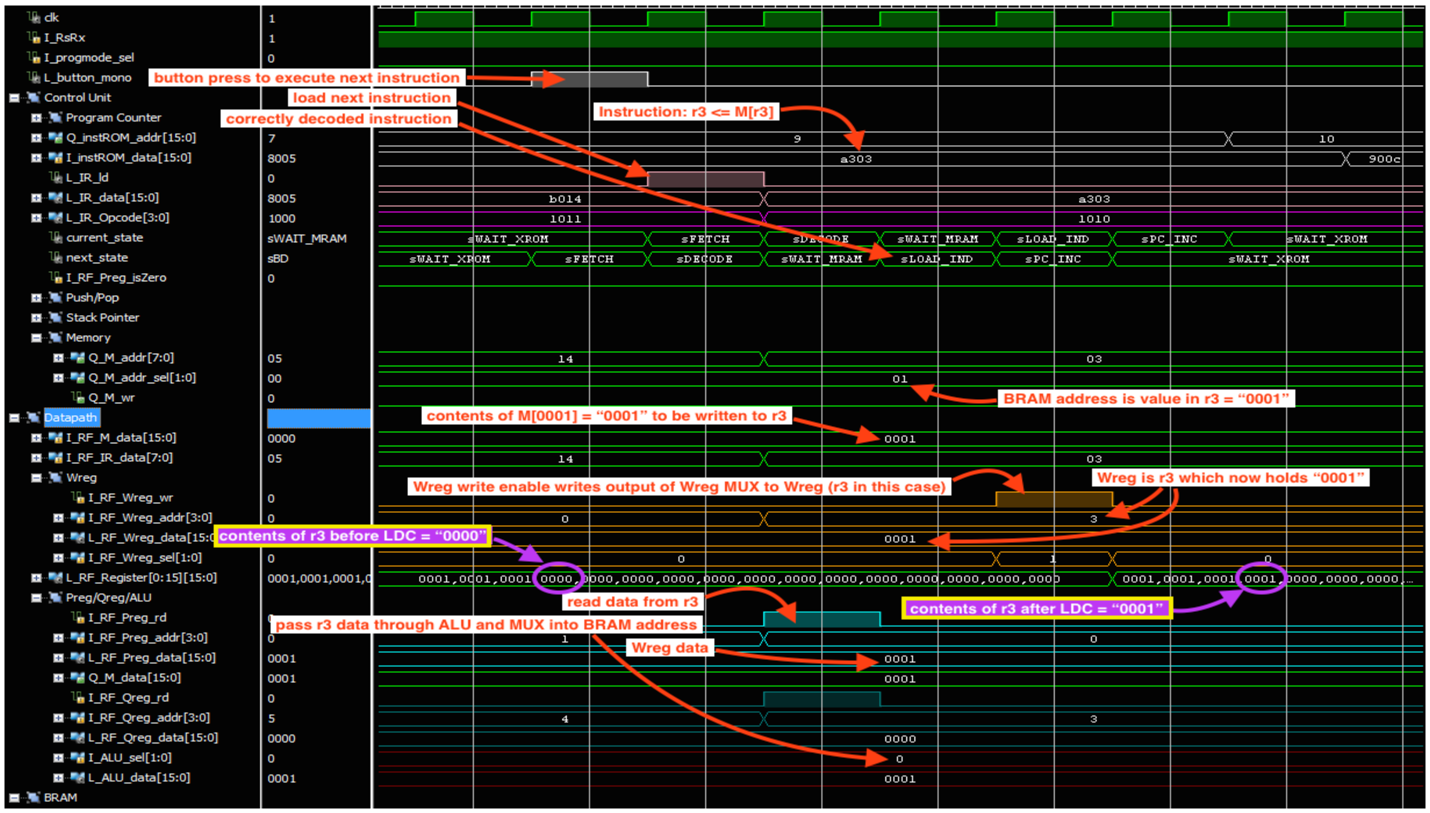
13: Branch Direct



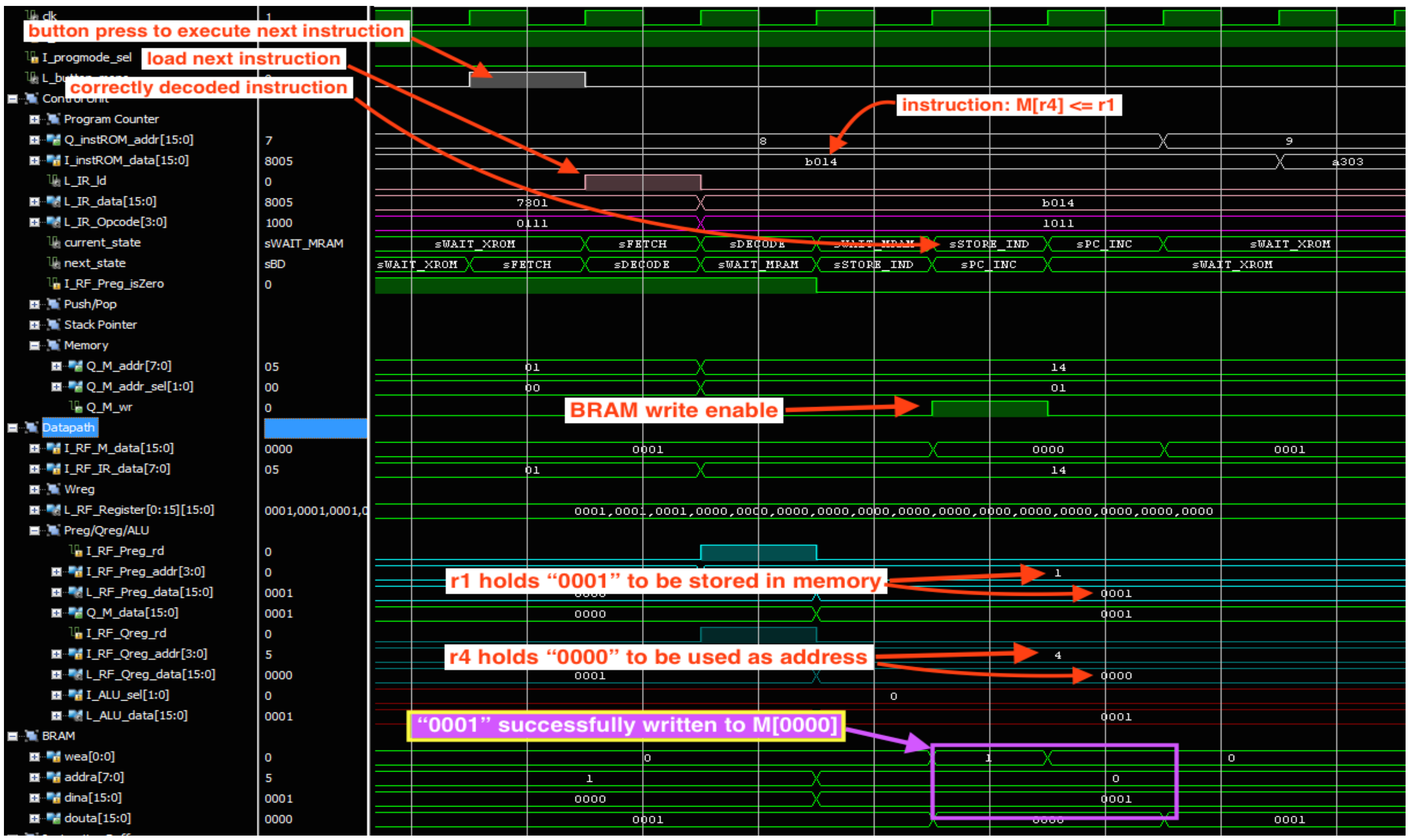
14: Branch with Link



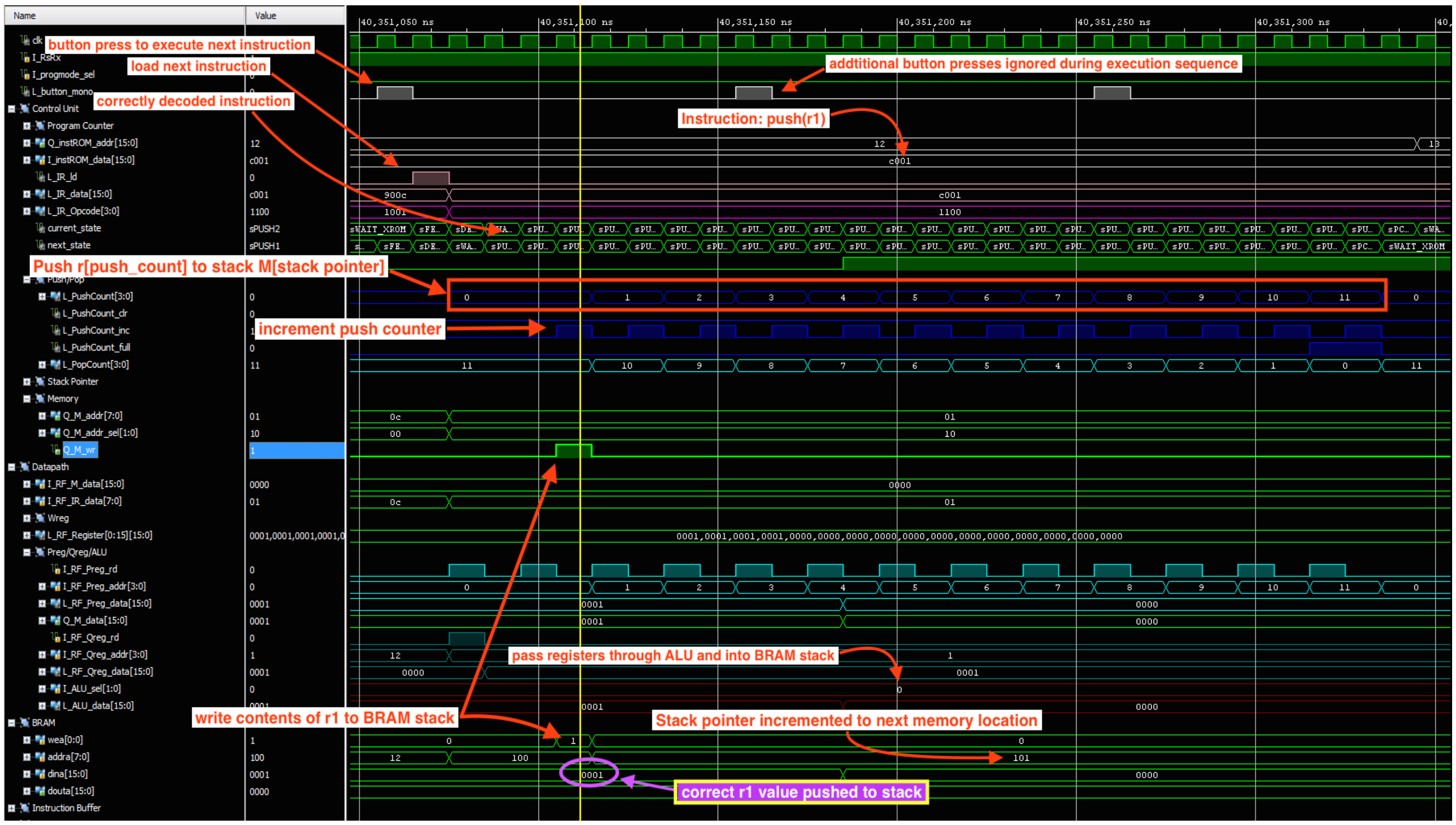
15: Load Indirect



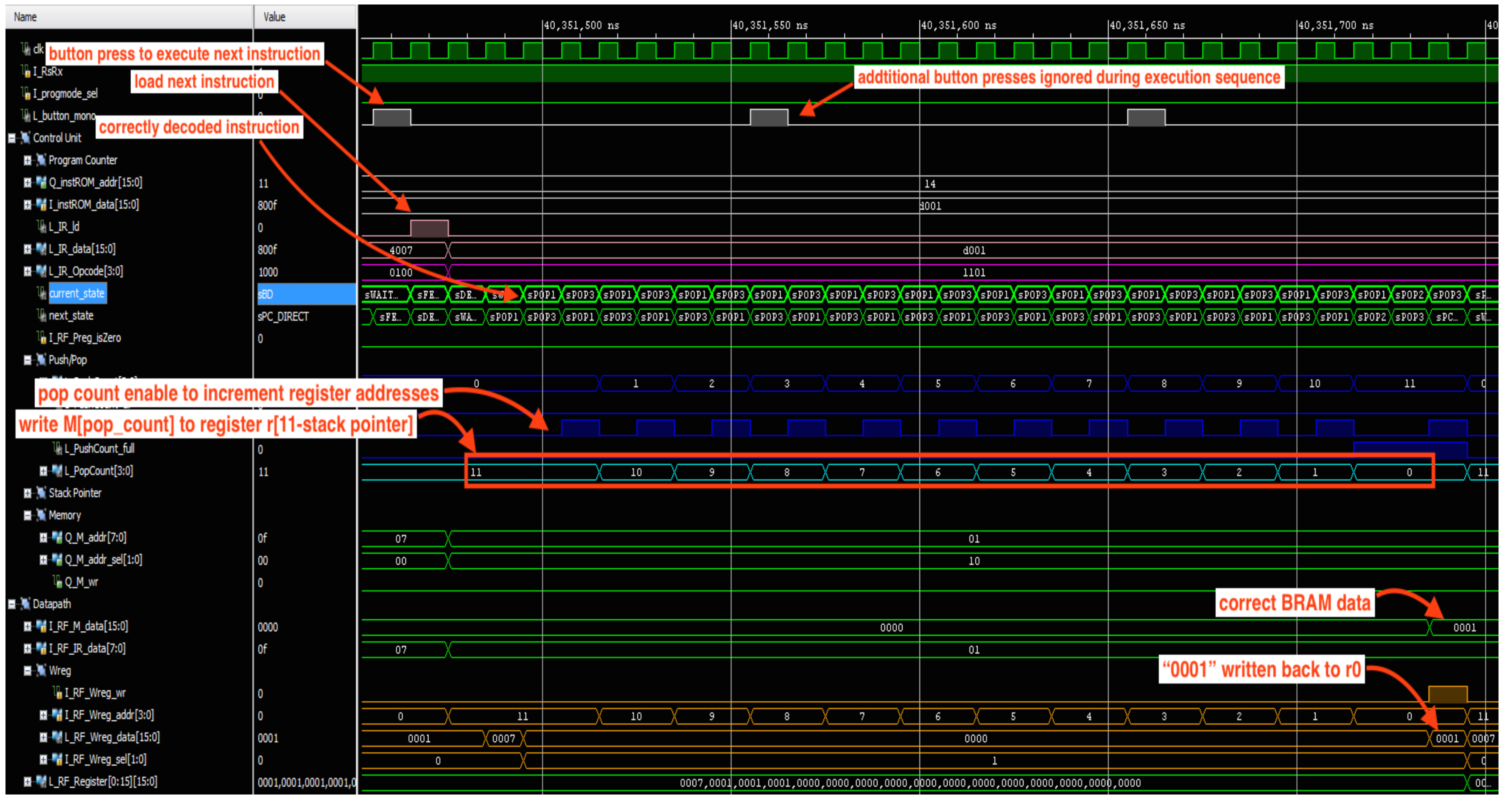
16: Store Indirect



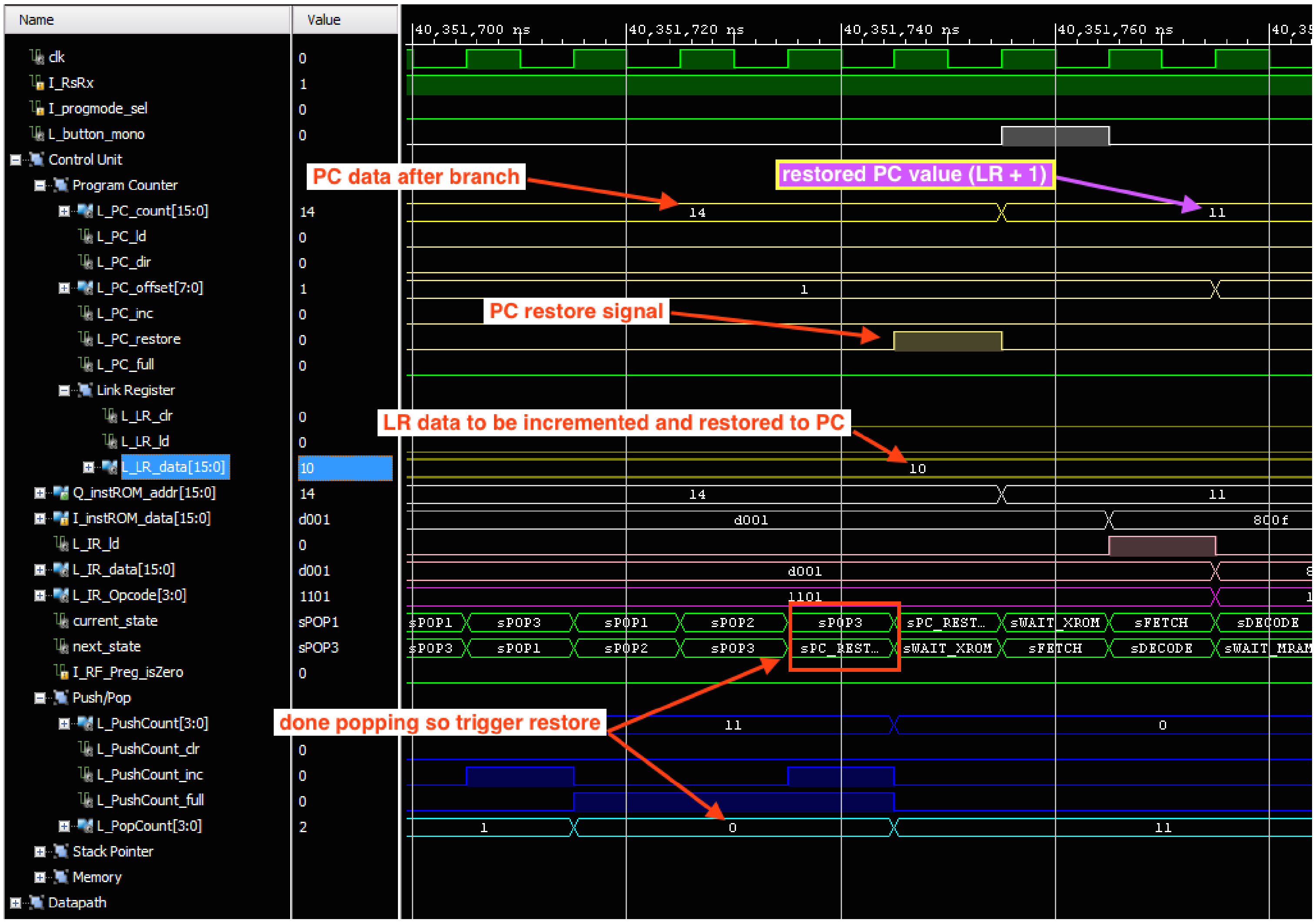
17: Push


button press to execute next instruction
load next instruction
correctly decoded instruction
addtitional button presses ignored during execution sequence
Instruction: push(r1)
Push r[push_count] to stack M[stack pointer]
increment push counter
pass registers through ALU and into BRAM stack
write contents of r1 to BRAM stack
Stack pointer incremented to next memory location
correct r1 value pushed to stack

18: Pop

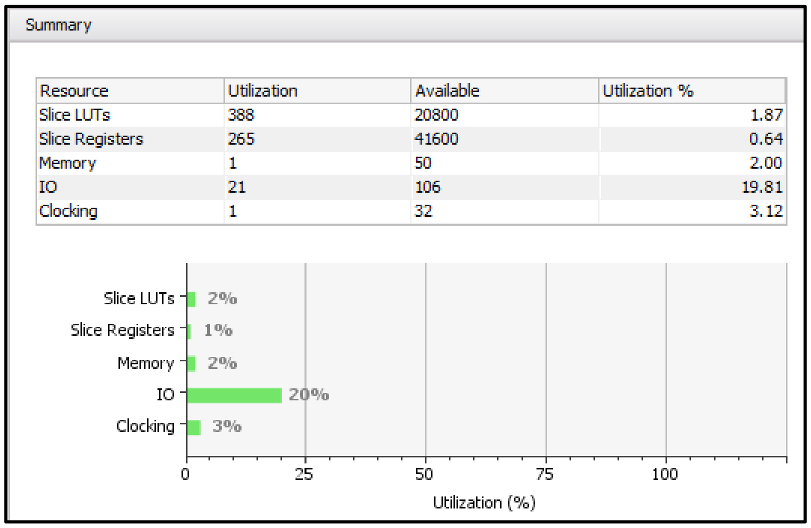

button press to execute next instruction
load next instruction
correctly decoded instruction
addtitional button presses ignored during execution sequence
pop count enable to increment register addresses
write M[pop_count] to register r[11-stack pointer]
correct BRAM data
"0001" written back to r0

19: Restore PC after pop



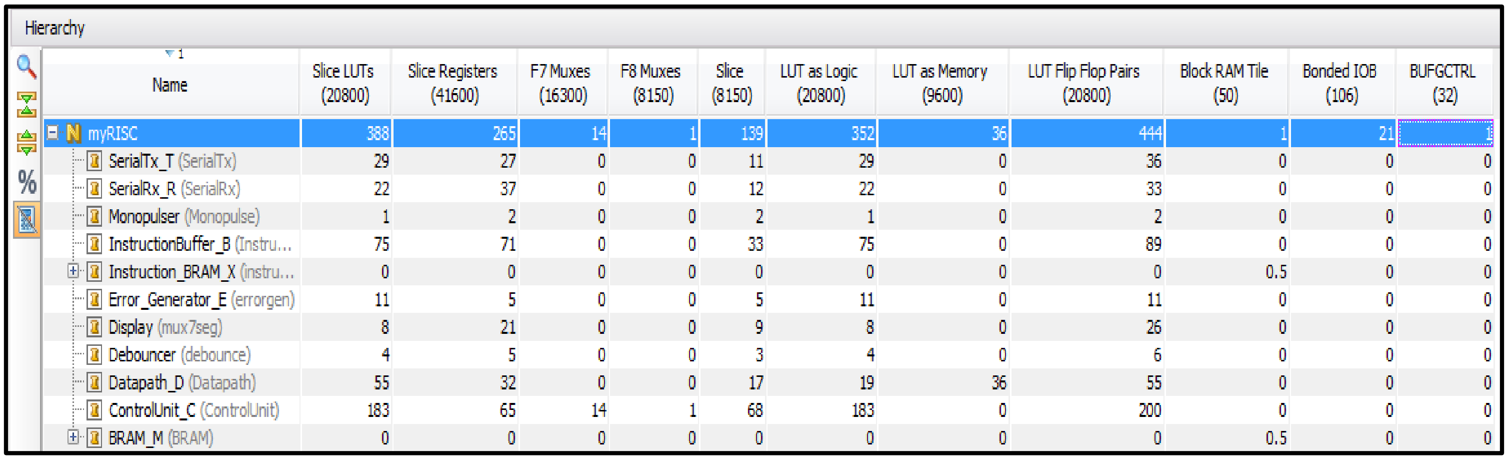
Hardware Implementation

Utilization Summary



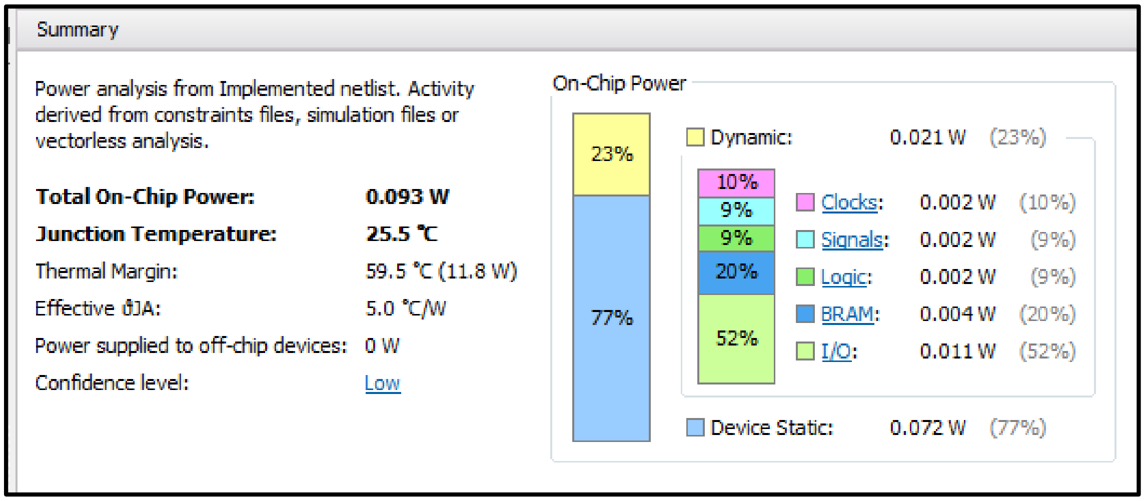
Summary

Resource	Utilization	Available	Utilization %
Slice LUTs	388	20800	1.87
Slice Registers	265	41600	0.64
Memory	1	50	2.00
IO	21	106	19.81
Clocking	1	32	3.12



Hierarchy

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
myRISC	388	265	14	1	139	352	36	444	1	21	1
SerialTx_T (SerialTx)	29	27	0	0	11	29	0	36	0	0	0
SerialRx_R (SerialRx)	22	37	0	0	12	22	0	33	0	0	0
Monopulser (Monopulse)	1	2	0	0	2	1	0	2	0	0	0
InstructionBuffer_B (Instru...	75	71	0	0	33	75	0	89	0	0	0
Instruction_BRAM_X (instru...	0	0	0	0	0	0	0	0	0.5	0	0
Error_Generator_E (errorgen)	11	5	0	0	5	11	0	11	0	0	0
Display (mux7seg)	8	21	0	0	9	8	0	26	0	0	0
Debouncer (debounce)	4	5	0	0	3	4	0	6	0	0	0
Datapath_D (Datapath)	55	32	0	0	17	19	36	55	0	0	0
ControlUnit_C (ControlUnit)	183	65	14	1	68	183	0	200	0	0	0
BRAM_M (BRAM)	0	0	0	0	0	0	0	0	0.5	0	0



Summary

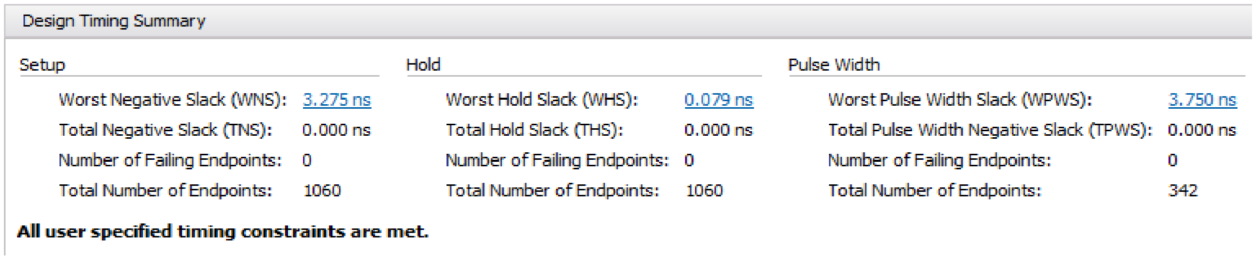
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.093 W**
Junction Temperature:	**25.5 ℃**
Thermal Margin:	59.5 ℃ (11.8 W)
Effective ϑJA:	5.0 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

Dynamic:	0.021 W	(23%)
Clocks:	0.002 W	(10%)
Signals:	0.002 W	(9%)
Logic:	0.002 W	(9%)
BRAM:	0.004 W	(20%)
I/O:	0.011 W	(52%)
Device Static:	0.072 W	(77%)

Timing Summary

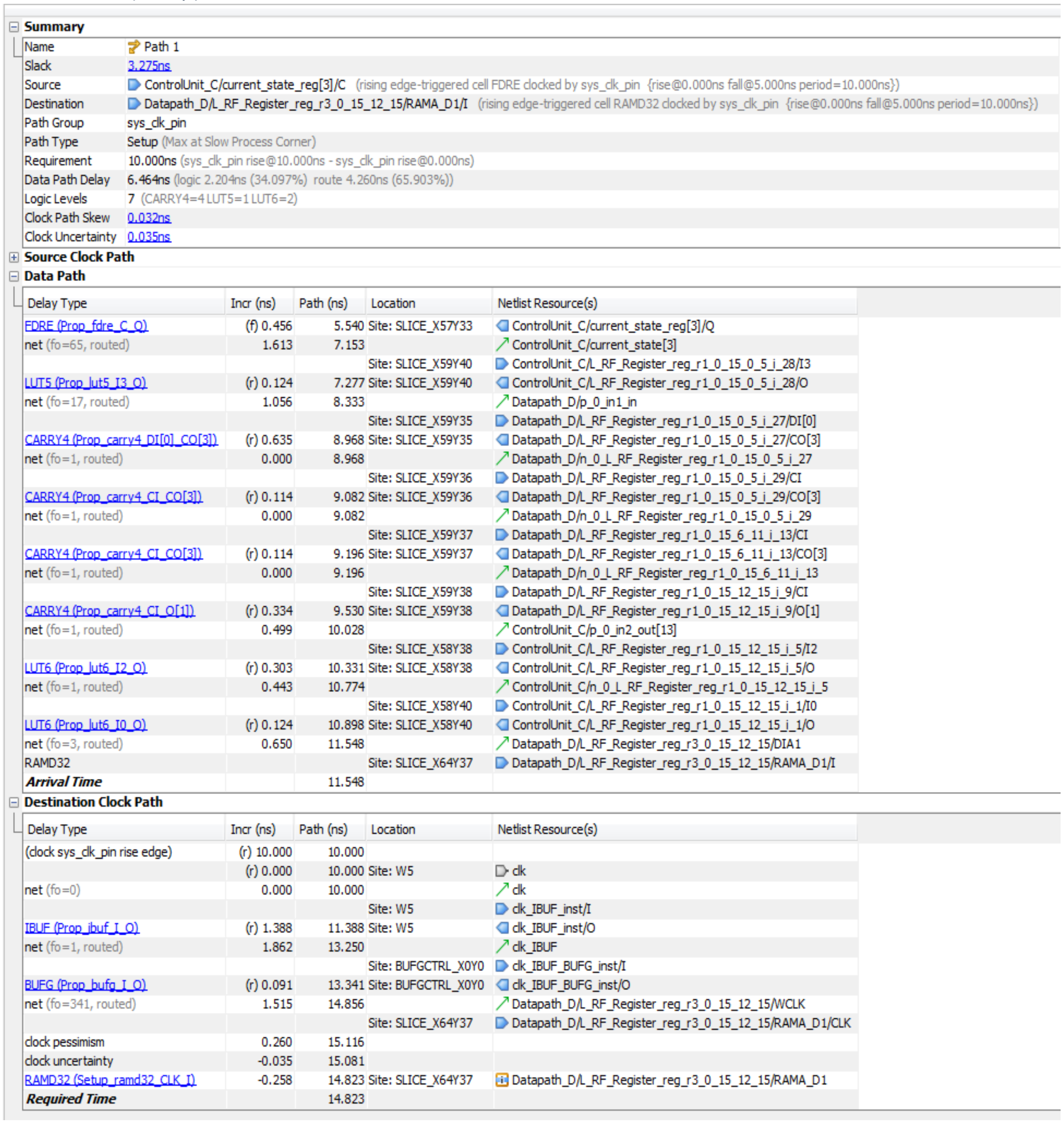


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.275 ns	Worst Hold Slack (WHS):	0.079 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1060	Total Number of Endpoints:	1060	Total Number of Endpoints:	342

All user specified timing constraints are met.

Critical Path (Setup)



Summary

Name	Path 1
Slack	3.275ns
Source	ControlUnit_C/current_state_reg[3]/C (rising edge-triggered cell FDRE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Destination	Datapath_D/L_RF_Register_reg_r3_0_15_12_15/RAMA_D1/I (rising edge-triggered cell RAMD32 clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group	sys_clk_pin
Path Type	Setup (Max at Slow Process Corner)
Requirement	10.000ns (sys_clk_pin rise@10.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay	6.464ns (logic 2.204ns (34.097%) route 4.260ns (65.903%))
Logic Levels	7 (CARRY4=4 LUT5=1 LUT6=2)
Clock Path Skew	0.032ns
Clock Uncertainty	0.035ns

Source Clock Path

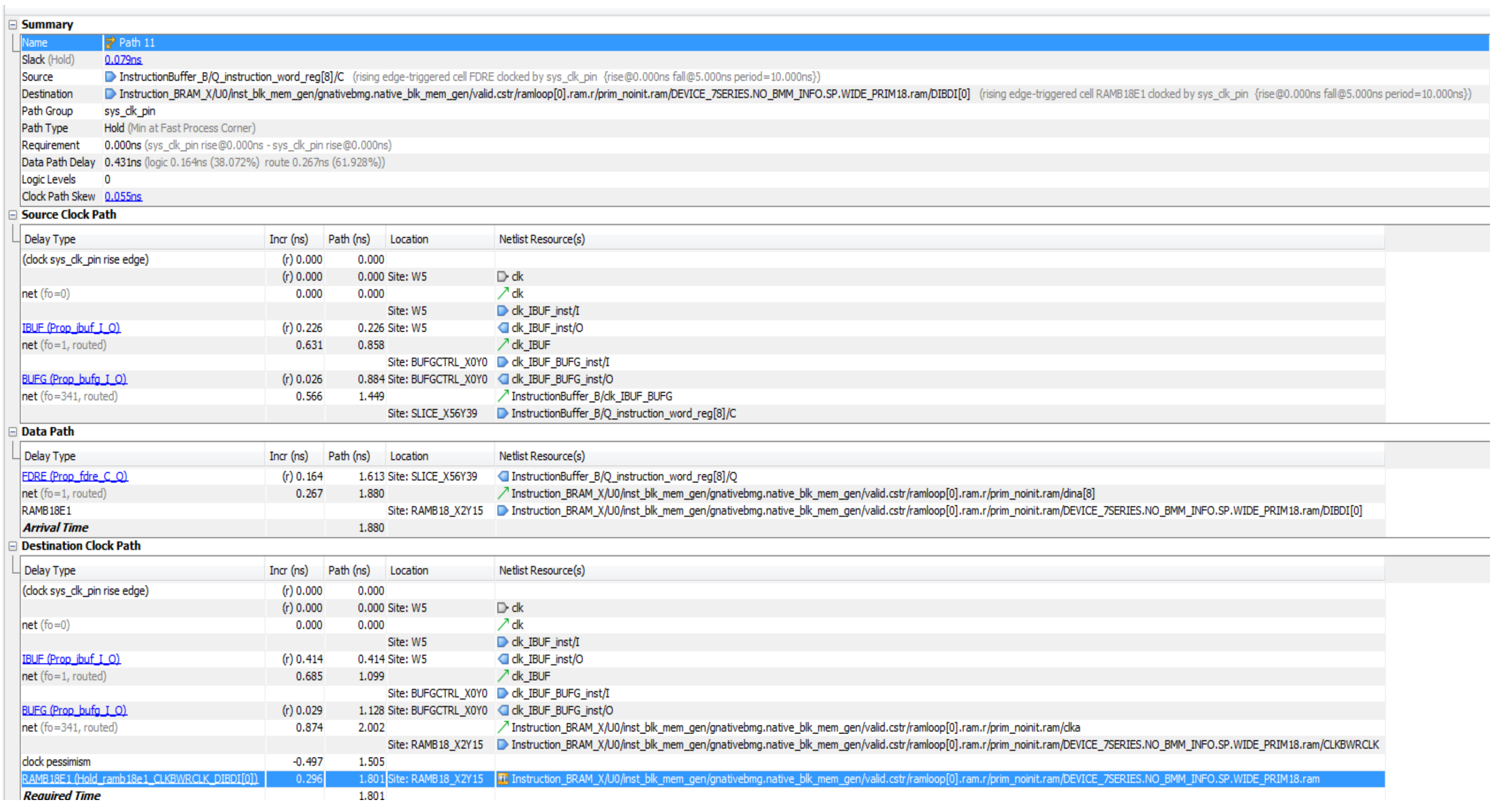
Data Path

Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
FDRE (Prop_fdre_C_Q)	(f) 0.456	5.540	Site: SLICE_X57Y33	ControlUnit_C/current_state_reg[3]/Q
net (fo=65, routed)	1.613	7.153		ControlUnit_C/current_state[3]
			Site: SLICE_X59Y40	ControlUnit_C/L_RF_Register_reg_r1_0_15_0_5_i_28/I3
LUT5 (Prop_lut5_I3_O)	(r) 0.124	7.277	Site: SLICE_X59Y40	ControlUnit_C/L_RF_Register_reg_r1_0_15_0_5_i_28/O
net (fo=17, routed)	1.056	8.333		Datapath_D/p_0_in1_in
			Site: SLICE_X59Y35	Datapath_D/L_RF_Register_reg_r1_0_15_0_5_i_27/DI[0]
CARRY4 (Prop_carry4_DI[0]_CO[3])	(r) 0.635	8.968	Site: SLICE_X59Y35	Datapath_D/L_RF_Register_reg_r1_0_15_0_5_i_27/CO[3]
net (fo=1, routed)	0.000	8.968		Datapath_D/n_0_L_RF_Register_reg_r1_0_15_0_5_i_27
			Site: SLICE_X59Y36	Datapath_D/L_RF_Register_reg_r1_0_15_0_5_i_29/CI
CARRY4 (Prop_carry4_CI_CO[3])	(r) 0.114	9.082	Site: SLICE_X59Y36	Datapath_D/L_RF_Register_reg_r1_0_15_0_5_i_29/CO[3]
net (fo=1, routed)	0.000	9.082		Datapath_D/n_0_L_RF_Register_reg_r1_0_15_0_5_i_29
			Site: SLICE_X59Y37	Datapath_D/L_RF_Register_reg_r1_0_15_6_11_i_13/CI
CARRY4 (Prop_carry4_CI_CO[3])	(r) 0.114	9.196	Site: SLICE_X59Y37	Datapath_D/L_RF_Register_reg_r1_0_15_6_11_i_13/CO[3]
net (fo=1, routed)	0.000	9.196		Datapath_D/n_0_L_RF_Register_reg_r1_0_15_6_11_i_13
			Site: SLICE_X59Y38	Datapath_D/L_RF_Register_reg_r1_0_15_12_15_i_9/CI
CARRY4 (Prop_carry4_CI_O[1])	(r) 0.334	9.530	Site: SLICE_X59Y38	Datapath_D/L_RF_Register_reg_r1_0_15_12_15_i_9/O[1]
net (fo=1, routed)	0.499	10.028		ControlUnit_C/p_0_in2_out[13]
			Site: SLICE_X58Y38	ControlUnit_C/L_RF_Register_reg_r1_0_15_12_15_i_5/I2
LUT6 (Prop_lut6_I2_O)	(r) 0.303	10.331	Site: SLICE_X58Y38	ControlUnit_C/L_RF_Register_reg_r1_0_15_12_15_i_5/O
net (fo=1, routed)	0.443	10.774		ControlUnit_C/n_0_L_RF_Register_reg_r1_0_15_12_15_i_5
			Site: SLICE_X58Y40	ControlUnit_C/L_RF_Register_reg_r1_0_15_12_15_i_1/I0
LUT6 (Prop_lut6_I0_O)	(r) 0.124	10.898	Site: SLICE_X58Y40	ControlUnit_C/L_RF_Register_reg_r1_0_15_12_15_i_1/O
net (fo=3, routed)	0.650	11.548		Datapath_D/L_RF_Register_reg_r3_0_15_12_15/DIA1
RAMD32			Site: SLICE_X64Y37	Datapath_D/L_RF_Register_reg_r3_0_15_12_15/RAMA_D1/I
Arrival Time		11.548		

Destination Clock Path

Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
(clock sys_clk_pin rise edge)	(r) 10.000	10.000		
	(r) 0.000	10.000	Site: W5	clk
net (fo=0)	0.000	10.000		clk
			Site: W5	clk_IBUF_inst/I
IBUF (Prop_ibuf_I_O)	(r) 1.388	11.388	Site: W5	clk_IBUF_inst/O
net (fo=1, routed)	1.862	13.250		clk_IBUF
			Site: BUFGCTRL_X0Y0	clk_IBUF_BUFG_inst/I
BUFG (Prop_bufg_I_O)	(r) 0.091	13.341	Site: BUFGCTRL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=341, routed)	1.515	14.856		Datapath_D/L_RF_Register_reg_r3_0_15_12_15/WCLK
			Site: SLICE_X64Y37	Datapath_D/L_RF_Register_reg_r3_0_15_12_15/RAMA_D1/CLK
clock pessimism	0.260	15.116		
clock uncertainty	-0.035	15.081		
RAMD32 (Setup_ramd32_CLK_I)	-0.258	14.823	Site: SLICE_X64Y37	Datapath_D/L_RF_Register_reg_r3_0_15_12_15/RAMA_D1
Required Time		14.823		

Critical Path (Hold)



Summary

Name	Path 11
Slack (Hold)	0.079ns
Source	InstructionBuffer_B/Q_instruction_word_reg[8]/C (rising edge-triggered cell FDRE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Destination	Instruction_BRAM_X/U0/inst_blk_mem_gen/gnativebmg.native_blk_mem_gen/valid.cstr/ramloop[0].ram.r/prim_noinit.ram/DEVICE_7SERIES.NO_BMM_INFO.SP.WIDE_PRIM18.ram/DIBDI[0] (rising edge-triggered cell RAMB18E1 clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group	sys_clk_pin
Path Type	Hold (Min at Fast Process Corner)
Requirement	0.000ns (sys_clk_pin rise@0.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay	0.431ns (logic 0.164ns (38.072%) route 0.267ns (61.928%))
Logic Levels	0
Clock Path Skew	0.055ns

Source Clock Path

Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
(clock sys_clk_pin rise edge)	(r) 0.000	0.000		
	(r) 0.000	0.000	Site: W5	clk
net (fo=0)	0.000	0.000		clk
			Site: W5	clk_IBUF_inst/I
IBUF (Prop_ibuf_I_O)	(r) 0.226	0.226	Site: W5	clk_IBUF_inst/O
net (fo=1, routed)	0.631	0.858		clk_IBUF
			Site: BUFGCTRL_X0Y0	clk_IBUF_BUFG_inst/I
BUFG (Prop_bufg_I_O)	(r) 0.026	0.884	Site: BUFGCTRL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=341, routed)	0.566	1.449		InstructionBuffer_B/clk_IBUF_BUFG
			Site: SLICE_X56Y39	InstructionBuffer_B/Q_instruction_word_reg[8]/C

Data Path

Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
FDRE (Prop_fdre_C_Q)	(r) 0.164	1.613	Site: SLICE_X56Y39	InstructionBuffer_B/Q_instruction_word_reg[8]/Q
net (fo=1, routed)	0.267	1.880		Instruction_BRAM_X/U0/inst_blk_mem_gen/gnativebmg.native_blk_mem_gen/valid.cstr/ramloop[0].ram.r/prim_noinit.ram/dina[8]
RAMB18E1			Site: RAMB18_X2Y15	Instruction_BRAM_X/U0/inst_blk_mem_gen/gnativebmg.native_blk_mem_gen/valid.cstr/ramloop[0].ram.r/prim_noinit.ram/DEVICE_7SERIES.NO_BMM_INFO.SP.WIDE_PRIM18.ram/DIBDI[0]
Arrival Time		1.880		

Destination Clock Path

Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
(clock sys_clk_pin rise edge)	(r) 0.000	0.000		
	(r) 0.000	0.000	Site: W5	clk
net (fo=0)	0.000	0.000		clk
			Site: W5	clk_IBUF_inst/I
IBUF (Prop_ibuf_I_O)	(r) 0.414	0.414	Site: W5	clk_IBUF_inst/O
net (fo=1, routed)	0.685	1.099		clk_IBUF
			Site: BUFGCTRL_X0Y0	clk_IBUF_BUFG_inst/I
BUFG (Prop_bufg_I_O)	(r) 0.029	1.128	Site: BUFGCTRL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=341, routed)	0.874	2.002		Instruction_BRAM_X/U0/inst_blk_mem_gen/gnativebmg.native_blk_mem_gen/valid.cstr/ramloop[0].ram.r/prim_noinit.ram/clka
			Site: RAMB18_X2Y15	Instruction_BRAM_X/U0/inst_blk_mem_gen/gnativebmg.native_blk_mem_gen/valid.cstr/ramloop[0].ram.r/prim_noinit.ram/DEVICE_7SERIES.NO_BMM_INFO.SP.WIDE_PRIM18.ram/CLKBWRCLK
clock pessimism	-0.497	1.505		
RAMB18E1 (Hold_ramb18e1_CLKBWRCLK_DIBDI[0])	0.296	1.801	Site: RAMB18_X2Y15	Instruction_BRAM_X/U0/inst_blk_mem_gen/gnativebmg.native_blk_mem_gen/valid.cstr/ramloop[0].ram.r/prim_noinit.ram/DEVICE_7SERIES.NO_BMM_INFO.SP.WIDE_PRIM18.ram
Required Time		1.801		

Component Architecture

Control Unit

Functionality

The control unit controls the flow of data through the datapath during the execution mode of the processor. This component contains the controller FSM along with the program counter, link register, instruction register, and stack pointer. Flow of data is controlled over a series of MUX select inputs which determine what information is able to enter the register file and ALU. The control unit also handles access requests from both the instruction RAM and general purpose RAM.

FSM

The finite state machine generates all of the control signals for the adjacent registers in the control unit and for the datapath and memory. The general operation of the FSM consists of these basic states – program count, fetch instruction, decode, and execute. The program count state controls the program counter, either incrementing it or loading in a predetermined memory address (depending on the previous instruction). After operating the program counter, the processor waits for the pushbutton before it fetches the next instruction from the memory. In the fetch instruction state the instruction register grabs the data from the address in the memory determined by the program counter. In the decode state the processor looks at the Opcode of the instruction held in the IR and moves to the appropriate execute state. In the execute state the FSM interacts with the datapath and the memory, moving the appropriate data between the two locations.

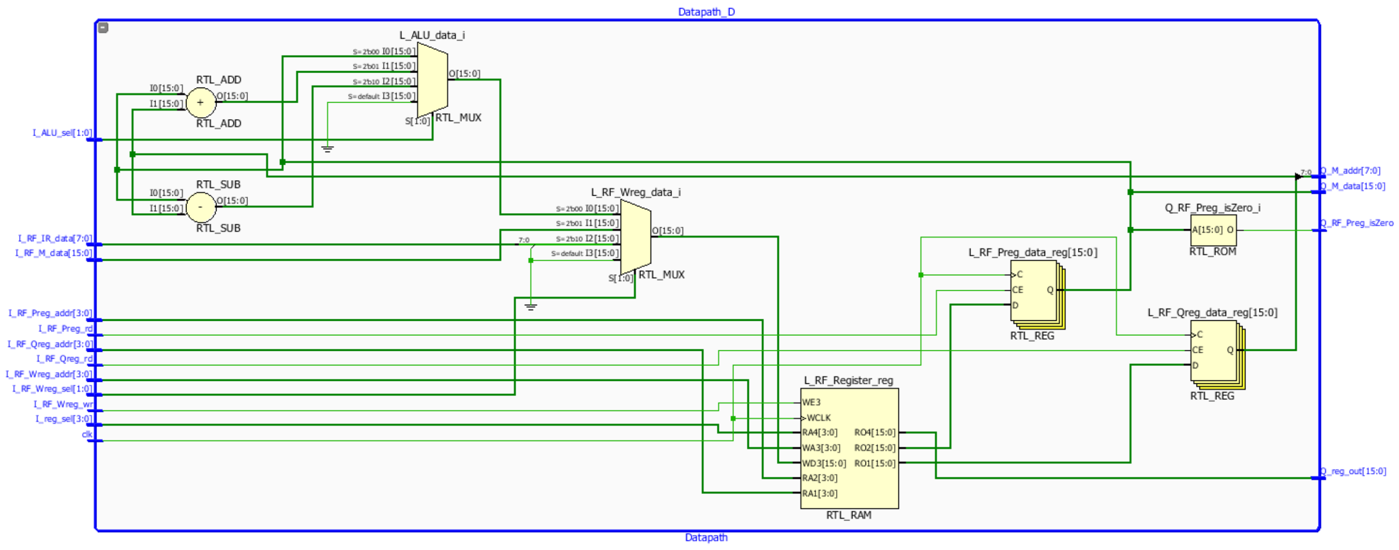
(See **Appendix L** for FSM diagram)

Datapath

Functionality

The datapath reacts to control signals from the control unit FSM and moves data between the 16 general purpose registers and the general purpose BRAM through two output pipeline registers (Preg and Qreg), an arithmetic logic unit (ALU), and an input multiplexer, all controlled by the control unit FSM. The synthesis tool chose to implement the general purpose registers as single-port write-enable/3-port read random access memory (RAM), though this could also be implemented as regular registers. The two read-output pipeline registers, Preg and Qreg, both send data into the ALU, which can add or subtract the data in the two registers or pass through the Preg data to be written back into a different register. The Preg is also used to send data directly into the BRAM, while the Qreg is used for indirect addressing of the BRAM. Data to be written to a register can come from the ALU, the BRAM, or directly from the instruction register in the control unit. These inputs are selected by a FSM-controlled multiplexer.

RTL Schematic

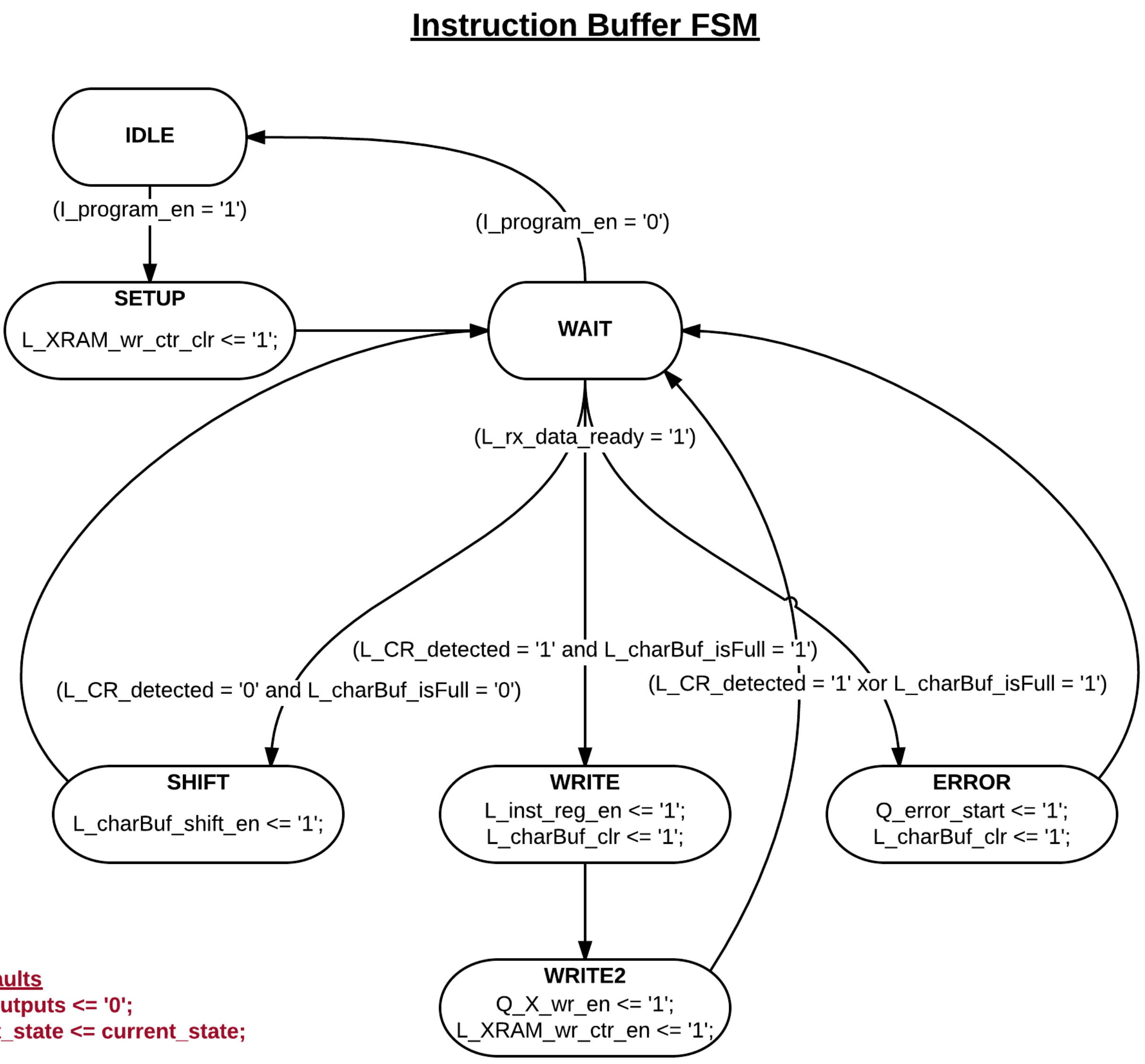


Instruction Buffer

Functionality

The instruction buffer receives 8-bit parallel data from the serial receiver and buffers it into a 4-word character buffer. Once four characters have been shifted in (detected by a counter that increments with every shift), the instruction buffer waits for a carriage return (CR) from the serial receiver. If the carriage return is recognized, the four previous characters are loaded out of the character buffer, converted from ASCII codes into the appropriate hexadecimal numbers, concatenated, and saved to the instruction memory (XRAM). A counter holds the address for the XRAM and increments every time an instruction is written. This counter holds its value once the user enters *execution* mode and is read by the control unit as a limit for the program counter. This value is cleared once the user re-enters *program* mode. If the received characters do not fit the (4-character then CR) format, the instruction is ignored and the instruction buffer asserts an error flag, which triggers a separate error sequence (see Error Generator).

FSM



Defaults
all outputs <= '0';
next_state <= current_state;

General Purpose Memory (BRAM)

Functionality

The single-port read/write enable block RAM serves for both general purpose memory and houses the Stack memory. The BRAM is controlled by the control unit and its data interacts purely with the datapath. The BRAM is written to with an enable signal from the control unit FSM. Its data input reads from the P register in the datapath. The BRAM address is MUX'ed between the instruction register (direct addressing), the Q register output (indirect addressing), and the Stack Pointer, selected asynchronously based the Opcode in the instruction register.

Instruction Memory (XRAM)

Functionality

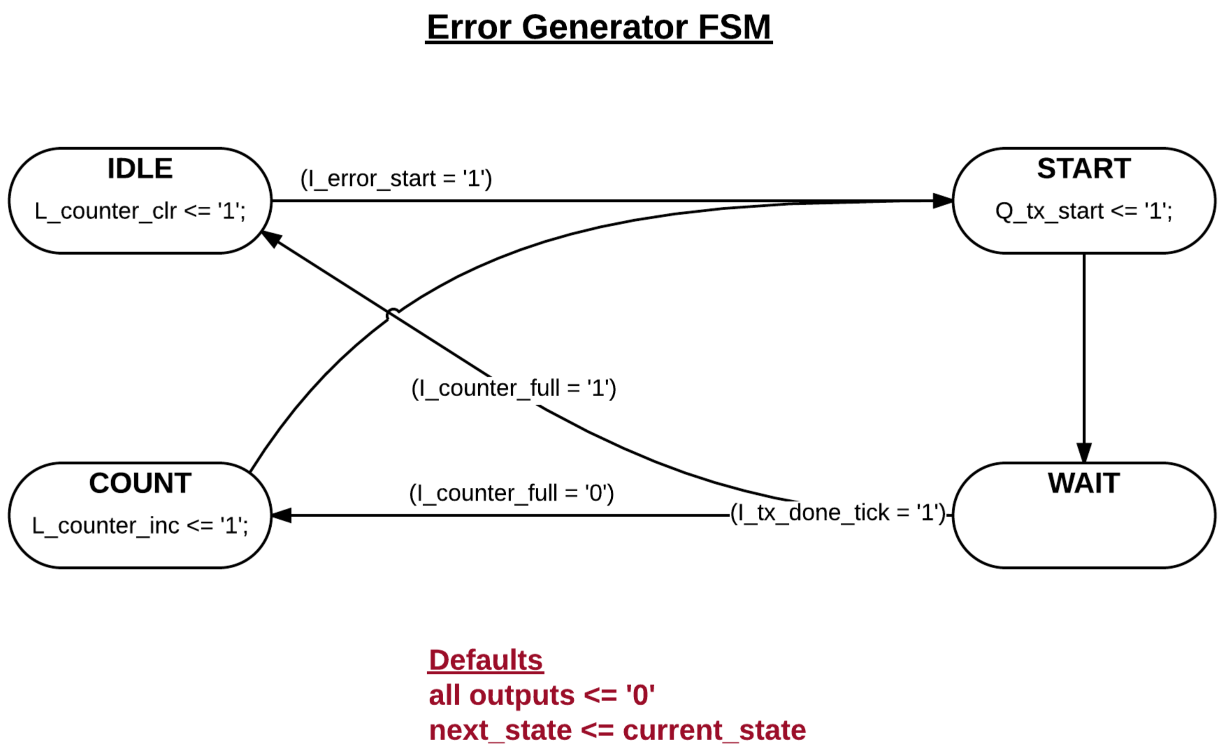
The instruction memory is a single-port read/write enable block RAM that serves to hold the instructions sent in through the serial receiver and the instruction buffer. The address port is MUX'ed between the address output of the instruction buffer during *program* mode and the control unit during *execution* mode. The XRAM is written to using a write enable signal from the instruction buffer. The data out of the XRAM is sent to the control unit and into the instruction register.

Error Generation

Functionality

The error generator is a small module that sends an "ERROR\n" message through the serial transmitter when it is given a start tick signal. The module uses a counter to cycle through the 8-bit ASCII codes for each character in the "ERROR\n" string and sends each character to the serial transmitter one at a time.

FSM



Results and Discussion

Ultimately, the project was successfully implemented in hardware. The processor functioned exactly as desired, and performed to the design specifications. In addition to rigorous testing of each instruction for a known set of inputs, the processor also was able to execute a program to compute the Fibonacci series (the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... where each number in the series is found by adding up the two numbers before it), which was the original "reach" design goal.

We are very happy with the level of success we have achieved in building a simple RISC processor. Success did not come without its share of hurdles, as there were many issues that we ran into along the way. One of the biggest struggles we faced was keeping track of the signal names and components, once the design reached a critical mass of complexity. This was resolved by adopting a very strict VHDL naming convention – detailed in Appendix A, which enabled us to easily determine the parent modules for signals and components. Another problem we ran into was debugging in hardware, given that most of the low-level operations that are critical to processor function can't be observed from the top level design. This issue was overcome by methodically debugging each module individually, and by not proceeding with another portion of the design until we had verified hardware functionality for the current module. We also mapped a large number of control signals to the PMOD sockets so we could observe their values with an oscilloscope.

If we had more time to complete the project, we would have liked to incorporate a Multiple Intermediate Pipeline Stages (MIPS) architecture scheme into the design of the processor. This would require a parallel pipeline to be implemented both within the controller and the datapath. In a MIPS architecture, a new instruction is loaded into the IR on every clock cycle, and each stage of an instruction's execution is simultaneously carried out and stored in the next pipeline register. This creates a faster processor by dramatically increasing instruction throughput, and also would allow us to clock our processor at higher speeds. Unfortunately, we determined that the control logic for a successful MIPS architecture stretches far beyond the scope of this course, and would have required an extra number of weeks to complete. The main difficulty in implementing a MIPS processor architecture arises when sequential instructions need to access the same register or memory address at a different stage in the pipeline. The controller now must not only ensure that both instructions are executed correctly, but also that the first instruction is fully executed before the second is performed – a daunting task when both instructions and data are moving through the pipeline at a different pace. Difficulties like this can be remedied by strategically placing No-Op commands in the pipeline to delay instructions that have these dependencies, ensuring the correct data is in the registers at the correct time. Part of this procedure is often done in software by the assembler, which is far beyond the scope of this course. We chose to first make sure that our control unit could properly operate the datapath, and ultimately decided that it would be more reasonable to add user interface functionality rather that attempt the parallel pipelined design.

We chose to interface with the processor over the familiar serial interface programming functionality and with a single-step pushbutton control. This idea, while not typical for a regular microprocessor, allowed us to demonstrate the functionality of the board in hardware rather than purely in simulation. While we initially attempted the very ambitious plan of creating a basic assembler in hardware, we quickly realized that this design would have been woefully inefficient and was not worth the effort.

While we are disappointed that we did not achieve the ambitious goal of creating our own assembler, we are glad that we were able to implement a fully functional design.

The biggest takeaways from this project have been a deepened understanding of processor operating principles, and exposure to different types of architectures. This project forced us to build for ourselves the defining features of a RISC microprocessor and required us to research many advanced features and architecture types before deciding on a specific design. Had we not chosen this ambitious project, we most likely would not have been exposed to any of the material and resources that we needed to utilize to achieve a successful result.

Appendix

Appendix A: VHDL Style Conventions

Due to the large quantity of complex code, a strict VHDL naming convention was followed to prevent mixing up signals in different scopes.

Entities, components, signals, ports, and VHDL keywords are written in lowercase. Each has a prefix according to the following rules:

- **I_** for input ports of a VHDL **entity**
- **Q_** for output ports of a VHDL **entity**
- **L_** for local signals within an entity or generated by a VHDL construct (e.g. by a signal assignment)
    - counters, internal control signals,
- **X_** with an uppercase X for signals generated **by an instantiated entit**y.
    - control signals, flags, TC, coming from a component

Each instantiated component is given some uppercase letter X (other than I, L, or Q) which will prefix all signals driven by that component. All signals driven by some component X can then either get the prefix **X_** (if the component drives an internal signal), or the prefix **Q_** (if the instantiated component drives an output port of the entity being defined). The resulting convention is that all signal names are prefixed by the component that drives them, unless it is connected to an output port of the current entity.

Enumerated types (states), constants, and variables are written in UPPERCASE. Prefixes have no underscores.

- states are prefixed by a lowercase “s” (sWAIT, sPOP)
- generic constants are prefixed by a lowercase “g” (gBUS_WIDTH)
- Numeric constants are prefixed by “r” if real, “I” if integer type, or “c” if STD_LOGIC
- variables are prefixed by lowercase “v” (vLOOP_NUM)

Appendix B: VHDL Source Listings

Top Level

```
-- Class:            ENGS 128 15S
-- Engineer:         Brett Nicholas and Matt Metzler
--
-- Create Date:      05/27/2015 03:43:05 PM
-- Design Name:      Final Project
-- Module Name:      myRISC - Behavioral
-- Project Name:     MyRISC
-- Target Devices:   Xilinx Arctix7 FPGA on Digilent BASYS3 project board
-- Tool Versions:    Xilinx Vivado
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
          use IEEE.STD_LOGIC_1164.ALL;
          use IEEE.NUMERIC_STD.ALL;
library UNISIM;
          use UNISIM.VComponents.all;

entity myRISC is
Port (     clk                  : in std_logic;                              -- clock in
          I_RsRx                : in std_logic;                              -- serial data in
          I_progmode_sel        : in std_logic;                              -- select between program and execute modes
          I_button              : in std_logic;                              -- button to step through commands
          I_display_sel         : in std_logic;                              -- select whether to display PC or registers
          I_reg_sel             : in std_logic_vector(3 downto 0);           -- select which register to display
          Q_RsTx                : out std_logic;                             -- serial data out
          Q_segments            : out std_logic_vector(0 to 6);              -- segments for display
          Q_anodes              : out std_logic_vector(3 downto 0)           -- anodes for display
          );
end myRISC;


architecture Behavioral of myRISC is

-- Control Unit contains FSM controller for fetch,decode/load/execute/store operations
-- as well as the Program Counter (PC), Instruction Register (IR) and necessary switching logic
component ControlUnit is
Port (    clk                   : in std_logic;
          I_instROM_data        : in std_logic_vector (15 downto 0);         -- instruction from ROM memory
          I_RF_Preg_isZero      : in std_logic;                              -- flag indicating Preg is zero (used for jump if zero)
          I_program_en          : in std_logic;                              -- Enables programming mode (no program execution during this time)
          I_button              : in std_logic;                              -- button to step through commands
          I_PC_limit            : in std_logic_vector (7 downto 0);          -- Maximum step to execute
          Q_instROM_addr        : out std_logic_vector (15 downto 0);        -- ROM address from which to receive instructions
          Q_instROM_rd          : out std_logic;                             -- ROM read enable signal to aquire instruction
          Q_M_addr              : out std_logic_vector (7 downto 0);         -- BRAM address
          Q_M_addr_sel          : out std_logic_vector (1 downto 0);         -- Select between direct and indirect memory access
          Q_M_wr                : out std_logic;                             -- BRAM write enable
          Q_RF_IR_data          : out std_logic_vector (7 downto 0);         -- Bottom 8 bits from the instruction register (for load constant)
          Q_RF_Wreg_sel         : out std_logic_vector (1 downto 0);         -- Mux select for inputs to register file
          Q_RF_Wreg_addr        : out std_logic_vector (3 downto 0);         -- Address of register to be written to (Wreg)
          Q_RF_Wreg_wr          : out std_logic;                             -- write enable for write register (Wreg)
          Q_RF_Preg_addr        : out std_logic_vector (3 downto 0);         -- Address of primary read register (Preg)
          Q_RF_Preg_rd          : out std_logic;                             -- read enable for Preg
          Q_RF_Qreg_addr        : out std_logic_vector (3 downto 0);         -- Address of secondary read register (Qreg - only used for arithmetic functions)
          Q_RF_Qreg_rd          : out std_logic;                             -- read enable for Qreg
          Q_ALU_sel             : out std_logic_vector (1 downto 0);         -- ALU MUX: 00=>Preg pass through, 01=>Preg+Qreg, 10=>Preg-Qreg
          Q_SP_addr             : out std_logic_vector (7 downto 0);         -- Stack Pointer address for memory
          Q_PC_count            : out std_logic_vector (15 downto 0)         -- Push the program count out to display
          );
end component;
          signal C_instROM_addr : std_logic_vector (15 downto 0);            -- ROM address from which to receive instructions
          signal C_instROM_rd   : std_logic;                                 -- ROM read enable signal to aquire instruction
          signal C_M_addr       : std_logic_vector (7 downto 0);             -- BRAM address
          signal C_M_addr_sel   : std_logic_vector (1 downto 0);             -- Select between direct and indirect memory access
          signal C_M_wr         : std_logic;                                 -- BRAM write enable
          signal C_RF_IR_data   : std_logic_vector (7 downto 0);             -- Bottom 8 bits from the instruction register (for load constant)
          signal C_RF_Wreg_sel  : std_logic_vector (1 downto 0);             -- Mux select for inputs to register file
          signal C_RF_Wreg_addr : std_logic_vector (3 downto 0);             -- Address of register to be written to (Wreg)
          signal C_RF_Wreg_wr   : std_logic;                                 -- write enable for write register (Wreg)
          signal C_RF_Preg_addr : std_logic_vector (3 downto 0);             -- Address of primary read register (Preg)
          signal C_RF_Preg_rd   : std_logic;                                 -- read enable for Preg
          signal C_RF_Qreg_addr : std_logic_vector (3 downto 0);             -- Address of secondary read register (Qreg - only used for arithmetic functions)
          signal C_RF_Qreg_rd   : std_logic;                                 -- read enable for Qreg
          signal C_ALU_sel      : std_logic_vector (1 downto 0);             -- ALU MUX: 00=>Preg pass through, 01=>Preg+Qreg, 10=>Preg-Qreg
          signal C_SP_addr      : std_logic_vector (7 downto 0);             -- Stack Pointer address for memory
          signal C_PC_count     : std_logic_vector (15 downto 0);            -- Push the program count out to display


-- Datapath contains data input MUX, MIPS, 16x1 Register File (RF), and ALU
component Datapath is
Port (    clk                   : in std_logic;
          I_RF_M_data           : in std_logic_vector (15 downto 0);         -- data from BRAM into datapath
          I_RF_IR_data          : in std_logic_vector (7 downto 0);          -- Bottom 8 bits from the instruction register (for load constant)
          I_RF_Wreg_addr        : in std_logic_vector (3 downto 0);          -- Address of register to be written to (Wreg)
          I_RF_Wreg_wr          : in std_logic;                              -- write enable for write register (Wreg)
```

```vhdl
          I_RF_Wreg_sel         : in std_logic_vector (1 downto 0);         -- Mux select for inputs to register file
          I_RF_Preg_addr        : in std_logic_vector (3 downto 0);         -- Address of primary read register (Preg)
          I_RF_Preg_rd          : in std_logic;                             -- read enable for Preg
          I_RF_Qreg_addr        : in std_logic_vector (3 downto 0);         -- Address of secondary read register (Qreg - only used for arithmetic functions)
          I_RF_Qreg_rd          : in std_logic;                             -- read enable for Qreg
          I_ALU_sel             : in std_logic_vector (1 downto 0);         -- ALU MUX: 00=>Preg pass through, 01=>Preg+Qreg, 10=>Preg-Qreg
          I_reg_sel             : in std_logic_vector (3 downto 0);         -- Select which register to display
          Q_M_data              : out std_logic_vector (15 downto 0);       -- data from datapath into BRAM
          Q_RF_Preg_isZero      : out std_logic;                            -- flag indicating Preg is zero (used for jump if zero)
          Q_M_addr              : out std_logic_vector (7 downto 0);        -- Memory address for inderect access
          Q_reg_out             : out std_logic_vector (15 downto 0)        -- register output for displaying
          );
end component;
          signal D_RF_Preg_isZero           : std_logic;                              -- flag indicating Preg is zero (used for jump if zero)
          signal D_M_data                   : std_logic_vector (15 downto 0);         -- data from datapath into BRAM
          signal D_M_addr                   : std_logic_vector (7 downto 0);          -- Memory address for inderect access
          signal D_reg_out                  : std_logic_vector (15 downto 0);         -- Register output for displaying


-- Instruction RAM for programmable processor instructions
-- Single-Port read/write BRAM, with read priority
COMPONENT instruction_BRAM
  PORT (
          clka       : IN std_logic;
          wea        : IN std_logic_vector(0 downto 0);
          addra      : IN std_logic_vector(7 downto 0);
          dina       : IN std_logic_vector(15 downto 0);
          douta      : OUT std_logic_vector(15 downto 0)
  );
end component;
          signal X_instROM_data             : std_logic_vector (15 downto 0);         -- instruction from ROM memory


-- Multipuropose single-port read/write BRAM, with read priority
-- Read/write operations have a 2 clock cycle latency
component BRAM
port (    clka       : in std_logic;                             -- master clock
          ena        : IN std_logic;                             -- clock enable
          wea        : in std_logic_vector(0 downto 0);          -- write enable
          addra      : in std_logic_vector(7 downto 0);          -- read/write address
          dina       : in std_logic_vector(15 downto 0);         -- write data in
          douta      : out std_logic_vector(15 downto 0) );      -- read data
end component;
          signal M_RF_M_data    : std_logic_vector (15 downto 0); -- data from BRAM into datapath


-- Serial Receiver to program XRAM over USART
component SerialRx
port (    clk                   : in std_logic;
          RsRx                  : in std_logic;
          Q_rx_data             : out std_logic_vector(7 downto 0);
          Q_rx_done_tick        : out std_logic  );
end component;
          signal R_RX_data      : std_logic_vector (7 downto 0) := x"00";
          signal R_RX_done_tick : std_logic := '0';


-- Serial Transmitter
component SerialTx is
Port (    clk                   : in  std_logic;
          tx_data               : in  std_logic_vector (7 downto 0);
          tx_start              : in  std_logic;
          tx                    : out  std_logic;     -- to Nexys 2 RS-232 port
          tx_done_tick          : out  std_logic);
end component;
          signal T_tx_done_tick : std_logic := '0';   -- Transmitter done tick


-- Instruction buffer : Buffers received characters until carriage return, then converts
-- the received word into a 16-bit instruction and stores the word in instruction memory
component InstructionBuffer is
Port (    clk                   : in std_logic;
          I_RX_done_tick        : in std_logic;                             -- SPI conversion done tick
          I_RX_data             : in std_logic_vector (7 downto 0);         -- SPI parallel data
          I_program_en          : in std_logic;                             -- Enables programming mode (no program execution during this time)
          Q_instruction_word    : out std_logic_vector (15 downto 0);       -- 16-bit instruction to be stored in XRAM
          Q_X_addr              : out std_logic_vector (7 downto 0);        -- XRAM write address
          Q_X_wr_en             : out std_logic;                            -- XRAM write enable
--        Q_X_busy              : out std_logic;                            -- Instruction programming in progress flag
          Q_error_start         : out std_logic                             -- start the error sequence
          );
end component;
          signal B_instruction_word         : std_logic_vector (15 downto 0) := x"0000";-- 16-bit instruction to be stored in XRAM
          signal B_X_addr                   : std_logic_vector (7 downto 0) := x"00";   -- XRAM write address
          signal B_X_wr_en                  : std_logic := '0';                       -- XRAM write enable
          signal B_X_busy                   : std_logic := '0';                       -- Instruction programming in progress flag
          signal B_error_start              : std_logic := '0';

component debounce IS
PORT(     clk     : IN  std_logic;  -- assumes 25Mhz clock (MIGHT NEED TO CHANGE)
          button  : IN  std_logic;  -- input signal to be debounced
          db_out  : OUT std_logic); -- debounced signal out
end component;
          signal L_button_db    : std_logic := '0';              -- debounced button
```

```
component Monopulse IS
Port (    clk                    : in std_logic;
          button                 : in std_logic;
          mono_out               : out std_logic
          );
end component;
          signal L_button_mono  : std_logic := '0';   -- monopulsed and debounced button

component mux7seg is
Port (    clk                    : in  std_logic;
          y0, y1, y2, y3         : in  std_logic_vector (3 downto 0);        -- digits
          seg                    : out  std_logic_vector(0 to 6);            -- segments (a...g)
          an                     : out  std_logic_vector (3 downto 0) );     -- anodes
end component;
          signal L_display      : std_logic_vector (15 downto 0) := x"0000";-- post-mux display vector


component errorgen is
Port (    clk                               : in std_logic;
          I_error_start                     : in std_logic;
          I_tx_done_tick                    : in std_logic;
          Q_tx_start                        : out std_logic;
          Q_tx_data                         : out std_logic_vector (7 downto 0)
          );
end component;
          signal E_tx_start                 : std_logic := '0';
          signal E_tx_data                  : std_logic_vector(7 downto 0) := x"00";


-- ADDRESS MUXes
          signal L_M_addr                   : std_logic_vector (7 downto 0) := x"00";   -- Memory address mux
          signal L_X_addr                   : std_logic_vector (7 downto 0) := x"00";   -- Instruction Memory address mux

BEGIN  -------------------------------------------------------------------------------------------------------------------------

-- Memory address MUX
-- switches between direct address, indirect address (within register),and the address held in the stack pointer.
with C_M_addr_sel select
          L_M_addr <=           C_M_addr when "00",
                                D_M_addr when "01",
                                C_SP_addr when "10",
                                x"00" when others;

-- Instruction Memory address MUX
-- Switches between address from controller (execute mode) and address from buffer (programmable mode)
with I_progmode_sel select
          L_X_addr <=           C_instROM_addr(7 downto 0) when '0',        -- address from controller (PC: execute mode)
                                B_X_addr when '1',                          -- address from buffer (programming mode)
                                x"00" when others;

-- Display Mux Process
DisplayMux: process(I_display_sel, I_progmode_sel, B_X_addr, D_reg_out, C_PC_count)
begin
          if (I_progmode_sel = '1') then
                    L_display <= x"00" & B_X_addr;
          elsif (I_display_sel = '1') then
                    L_display <= D_reg_out;
          else
                    L_display <= C_PC_count;
          end if;
end process;


ControlUnit_C: ControlUnit
Port map( clk => clk,
          I_instROM_data => X_instROM_data,           -- instruction from ROM memory
          I_RF_Preg_isZero => D_RF_Preg_isZero,       -- flag indicating Preg is zero (used for jump if zero)
          I_program_en => I_progmode_sel,             -- Enables programming mode (no program execution during this time)
          I_button => L_button_mono,                  -- button for stepping through instructions
          I_PC_limit => B_X_addr,                     -- Maximum step to execute
          Q_instROM_addr => C_instROM_addr,           -- ROM address from which to receive instructions
          Q_instROM_rd => C_instROM_rd,               -- ROM read enable signal to aquire instruction
          Q_M_addr => C_M_addr,                       -- BRAM address
          Q_M_addr_sel => C_M_addr_sel,               -- Select between direct and indirect memory access
          Q_M_wr => C_M_wr,                           -- BRAM write enable
          Q_RF_IR_data => C_RF_IR_data,               -- Bottom 8 bits from the instruction register (for load constant)
          Q_RF_Wreg_sel => C_RF_Wreg_sel,             -- Mux select for inputs to register file
          Q_RF_Wreg_addr => C_RF_Wreg_addr,           -- Address of register to be written to (Wreg)
          Q_RF_Wreg_wr => C_RF_Wreg_wr,               -- write enable for write register (Wreg)
          Q_RF_Preg_addr => C_RF_Preg_addr,           -- Address of primary read register (Preg)
          Q_RF_Preg_rd => C_RF_Preg_rd,               -- read enable for Preg
          Q_RF_Qreg_addr => C_RF_Qreg_addr,           -- Address of secondary read register (Qreg - only used for arithmetic functions)
          Q_RF_Qreg_rd => C_RF_Qreg_rd,               -- read enable for Qreg
          Q_ALU_sel => C_ALU_sel,                     -- ALU MUX: 00=>Preg pass through, 01=>Preg+Qreg, 10=>Preg-Qreg
          Q_SP_addr => C_SP_addr,                     -- Stack Pointer address for memory
          Q_PC_count => C_PC_count                    -- Push the program count out to display
          );


Datapath_D: Datapath
Port Map( clk => clk,
          I_RF_M_data => M_RF_M_data,                 -- data from BRAM into datapath
          I_RF_IR_data => C_RF_IR_data,               -- Bottom 8 bits from the instruction register (for load constant)
          I_RF_Wreg_addr => C_RF_Wreg_addr,           -- Address of register to be written to (Wreg)
          I_RF_Wreg_wr => C_RF_Wreg_wr,               -- write enable for write register (Wreg)
          I_RF_Wreg_sel => C_RF_Wreg_sel,             -- Mux select for inputs to register file
          I_RF_Preg_addr => C_RF_Preg_addr,           -- Address of primary read register (Preg)
          I_RF_Preg_rd => C_RF_Preg_rd,               -- read enable for Preg
          I_RF_Qreg_addr => C_RF_Qreg_addr,           -- Address of secondary read register (Qreg - only used for arithmetic functions)
          I_RF_Qreg_rd => C_RF_Qreg_rd,               -- read enable for Qreg
          I_ALU_sel => C_ALU_sel,                     -- ALU MUX: 00=>Preg pass through, 01=>Preg+Qreg, 10=>Preg-Qreg
          I_reg_sel => I_reg_sel,                     -- Select which register to display
          Q_M_data => D_M_data,                       -- data from datapath into BRAM
          Q_RF_Preg_isZero => D_RF_Preg_isZero,       -- flag indicating Preg is zero (used for jump if zero)
          Q_M_addr => D_M_addr,                       -- Memory address for inderect access
          Q_reg_out => D_reg_out                      -- register output for displaying
          );


Instruction_BRAM_X: instruction_BRAM
 PORT MAP (clka => clk,
          wea(0) => B_X_wr_en,
          addra => L_X_addr,
          dina => B_instruction_word,
          douta => X_instROM_data
          );


 BRAM_M: BRAM
 Port Map (clka => clk,          -- Port A (write port) clock
          ena => '1',            -- clock enable
          wea(0) => C_M_wr,      -- write enable (read if low, write if high)
          addra => L_M_addr,     -- read/write address
          dina => D_M_data,      -- write data: data from datapath into BRAM
          douta => M_RF_M_data             -- read data: from BRAM to Datapath
          );


 SerialRx_R: SerialRx
 Port Map (clk => clk,
          RsRx => I_RsRx,
          Q_rx_data => R_RX_data,
          Q_rx_done_tick => R_RX_done_tick
          );


  SerialTx_T: SerialTx
 Port Map (clk => clk,
          tx_data => E_tx_data,
          tx_start => E_tx_start,
          tx => Q_RsTx,  -- to Nexys 2 RS-232 port
          tx_done_tick => T_tx_done_tick
          );


 Error_Generator_E: errorgen
 Port Map (clk => clk,
          I_error_start => B_error_start,
          I_tx_done_tick => T_tx_done_tick,
          Q_tx_start => E_tx_start,
          Q_tx_data => E_tx_data
          );


 InstructionBuffer_B: InstructionBuffer
 Port Map (clk => clk,
          I_RX_done_tick => R_RX_done_tick,           -- SPI conversion done tick
          I_RX_data => R_rx_data,                     -- SPI parallel data
          I_program_en => I_progmode_sel,             -- Enables programming mode (no program execution during this time)
          Q_instruction_word => B_instruction_word,   -- 16-bit instruction to be stored in XRAM
          Q_X_addr => B_X_addr,                       -- XRAM write address (also PC limit)
          Q_X_wr_en => B_X_wr_en,                     -- XRAM write enable
          Q_X_busy => B_X_busy,                       -- Instruction programming in progress flag
          Q_error_start => B_error_start              -- start the error sequence
          );


  Debouncer: debounce
 Port Map (clk => clk,
          button => I_button,
          db_out => L_button_db
          );


  Monopulser: monopulse
 Port Map (clk => clk,
          button => L_button_db,
          mono_out => L_button_mono
          );

  Display: mux7seg
 Port Map (clk => clk,
          y0 => L_display(3 downto 0),
          y1 => L_display(7 downto 4),
          y2 => L_display(11 downto 8),
          y3 => L_display(15 downto 12),
          seg => Q_segments,
          an => Q_anodes
          );

 end Behavioral;
```

Controller

```
-- Class:             ENGS 128 15S
-- Engineer:          Brett Nicholas and Matt Metzler
--
-- Create Date:       05/27/2015 03:43:05 PM
-- Design Name:       Final Project
-- Module Name:       ControlUnit - Behavioral
-- Project Name:      MyRISC
-- Target Devices:    Xilinx Arctix7 FPGA on Digilent BASYS3 project board
-- Tool Versions:     Xilinx Vivado
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity ControlUnit is
Generic (STACK_TOP : integer := 100);  -- Address of the beginning of the stack
Port (      clk                 : in std_logic;
            I_instROM_data      : in std_logic_vector (15 downto 0);        -- instruction from ROM memory
            I_RF_Preg_isZero    : in std_logic;                             -- flag indicating Preg is zero (used for jump if zero)
            I_program_en        : in std_logic;                             -- Enables programming mode (no program execution during this time)
            I_button            : in std_logic;                             -- button to step through commands
            I_PC_limit          : in std_logic_vector (7 downto 0);         -- Maximum step to execute
            Q_instROM_addr      : out std_logic_vector (15 downto 0);       -- ROM address from which to receive instructions
            Q_instROM_rd        : out std_logic;                            -- ROM read enable signal to aquire instruction
            Q_M_addr            : out std_logic_vector (7 downto 0);        -- BRAM address
            Q_M_addr_sel        : out std_logic_vector (1 downto 0);        -- Select between direct and indirect memory access
            Q_M_wr              : out std_logic;                            -- BRAM write enable
            Q_RF_IR_data        : out std_logic_vector (7 downto 0);        -- Bottom 8 bits from the instruction register (for load constant)
            Q_RF_Wreg_sel       : out std_logic_vector (1 downto 0);        -- Mux select for inputs to register file
            Q_RF_Wreg_addr      : out std_logic_vector (3 downto 0);        -- Address of register to be written to (Wreg)
            Q_RF_Wreg_wr        : out std_logic;                            -- write enable for write register (Wreg)
            Q_RF_Preg_addr      : out std_logic_vector (3 downto 0);        -- Address of primary read register (Preg)
            Q_RF_Preg_rd        : out std_logic;                            -- read enable for Preg
            Q_RF_Qreg_addr      : out std_logic_vector (3 downto 0);        -- Address of secondary read register (Qreg - only used for arithmetic functions)
            Q_RF_Qreg_rd        : out std_logic;                            -- read enable for Qreg
            Q_ALU_sel           : out std_logic_vector (1 downto 0);        -- ALU MUX: 00=>Preg pass through, 01=>Preg+Qreg, 10=>Preg-Qreg
            Q_SP_addr           : out std_logic_vector (7 downto 0);        -- Stack Pointer address for memory
            Q_PC_count          : out std_logic_vector (15 downto 0)        -- Push the program count out to display
            );
end ControlUnit;


architecture Behavioral of ControlUnit is

            -- Signals for the Program Counter
            signal L_PC_count               : unsigned (15 downto 0) := x"0000";                 -- Program Count
            signal L_PC_ld                  : std_logic := '0';                                  -- PC load from IR offset
            signal L_PC_clr                 : std_logic := '0';                                  -- clear the PC
            signal L_PC_inc                 : std_logic := '0';                                  -- increment the PC
            signal L_PC_dir                 : std_logic := '0';                                  -- PC jump to direct address
            signal L_PC_restore             : std_logic := '0';                                  -- PC restore from LR
            signal L_PC_offset              : unsigned (7 downto 0) := x"00";                    -- PC offset value from IR
            signal L_PC_full                : std_logic := '0';                                  -- Flag when PC reaches the limit

            -- Signals for Instruction Register
            signal L_IR_data                : std_logic_vector (15 downto 0) := x"0000";         -- Instruction held in the IR
            signal L_IR_ld                  : std_logic := '0';                                  -- load enable for IR
            signal L_IR_Opcode              : std_logic_vector (3 downto 0);

            -- Signals for the Link Register
            signal L_LR_data                : unsigned (15 downto 0) := x"0000";                 -- Link Register
            signal L_LR_ld                  : std_logic := '0';                                  -- LR load from PC
            signal L_LR_clr                 : std_logic := '0';                                  -- clear the LR

            -- Signals for the stack register
            signal L_SP_addr                : unsigned (7 downto 0) := to_unsigned(STACK_TOP, 8); -- Stack pointer
            signal L_SP_inc                 : std_logic := '0';                                  -- Stack pointer increment
            signal L_SP_dec                 : std_logic := '0';                                  -- Stack pointer decrement
            signal L_SP_clr                 : std_logic := '0';                                  -- Stack pointer clear

            -- Signals for Push Counter
            signal L_PushCount              : unsigned (3 downto 0) := "0000";                   -- Counter for Push/Pop operations (cycles through registers)
            signal L_PushCount_clr          : std_logic := '0';                                  -- Clear the Push Counter
            signal L_PushCount_inc          : std_logic := '0';                                  -- increment the Push Counter
            signal L_PushCount_full         : std_logic := '0';                                  -- Flag when Push Counter = 12
            signal L_PopCount               : unsigned (3 downto 0) := x"C";                     -- Pop Count
            signal L_Push_ready             : std_logic := '0';                                  -- enable pushing!
            signal L_Pop_ready              : std_logic := '0';                                  -- enable popping!
```

```
          -- State Machine signals
          type state_type is (sIDLE, sWAIT_XROM, sWAIT_MRAM, sPC_INC, SPC_INC_pre, sPC_DIRECT, sPC_DIRECT_pre, sPC_JUMP, sPC_JUMP_pre,
                              sPC_RESTORE, sPC_RESTORE_pre, sFETCH, sDECODE, sLOAD_DIR, sSTORE_DIR, sMOVE, sCONST, sADD, sSUB, sBZ,
                              sBD, sBL, sLOAD_IND, sSTORE_IND, sNO_OP, sPUSH1, sPUSH2, sPOP1, sPOP2, sPOP3, sFINISHED, sFIRST);
          signal current_state, next_state : state_type := sIDLE;

 begin


Q_instROM_rd <= '1';
L_IR_Opcode <= L_IR_data(15 downto 12);

-- Asynchronous Address Outputs
 Q_instROM_addr <= std_logic_vector(L_PC_count);
 Q_M_addr <= L_IR_data(7 downto 0);
 Q_RF_Qreg_addr <= L_IR_data(3 downto 0);
 Q_SP_addr <= std_logic_vector(L_SP_addr);
Q_RF_Wreg_addr <= std_logic_vector(L_PopCount) when (L_IR_Opcode = x"D") else L_IR_data(11 downto 8);

with L_IR_opcode select
          Q_RF_Preg_addr <=     L_IR_data(11 downto 8) when x"2",
                                L_IR_data(11 downto 8) when x"7",
                                std_logic_vector(L_PushCount) when x"C",
                                L_IR_data(7 downto 4) when others;

with L_IR_opcode select
          Q_M_addr_sel <=       "01" when x"a",                  -- Memory address gets Qreg
                                "01" when x"b",
                                "10" when x"c",                  -- Memory address gets stack pointer
                                "10" when x"d",
                                "00" when others;

 --------------------------------------------
 ProgramCounter: process(clk)
 begin
          if rising_edge(clk) then
                    if ( L_PC_clr = '1' ) then
                              L_PC_count <= x"0000";
                    elsif ( L_PC_inc = '1' ) then
                              L_PC_count <= L_PC_count + 1;
                    elsif ( L_PC_dir = '1' ) then
                              L_PC_count <= x"00" & L_PC_offset;
                    elsif ( L_PC_ld = '1' ) then
                              L_PC_count <= L_PC_count + 1 + L_PC_offset;
                    elsif ( L_PC_restore = '1' ) then
                              L_PC_count <= L_LR_data + 1;
                    end if;
          end if;
 end process;
 Q_PC_count <= std_logic_vector(L_PC_count);
 L_PC_full <= '1' when L_PC_count >= (unsigned(I_PC_limit) - 1) else '0';


LinkRegister: process(clk)
 begin
          if rising_edge(clk) then
                    if ( L_LR_clr = '1' ) then
                              L_LR_data <= x"0000";
                    elsif ( L_LR_ld = '1' ) then
                              L_LR_data <= L_PC_count;
                    end if;
          end if;
 end process;


StackPointer: process(clk)
 begin
          if rising_edge(clk) then
                    if (L_SP_clr = '1') then
                              L_SP_addr <= to_unsigned(STACK_TOP, L_SP_addr'Length); -- reset SP to STACK_TOP
                    elsif (L_SP_dec = '1') then
                              L_SP_addr <= L_SP_addr - 1; -- decrement SP
                    elsif (L_SP_inc = '1') then
                              L_SP_addr <= L_SP_addr + 1; -- increment SP
                    end if;
          end if;
 end process;


PushCounter: process(clk)
 begin
          if rising_edge(clk) then
                    if ( L_PushCount_clr = '1' ) then
                              L_PushCount <= "0000";
                    elsif ( L_PushCount_inc = '1' ) then
                              if ( L_PushCount_full = '1' ) then
                                        L_PushCount <= "0000";
                              else
                                        L_PushCount <= L_PushCount + 1;
                              end if;
                    end if;
          end if;
 end process;
```

```
L_PushCount_full <= '1' when ( L_PushCount = x"B" ) else '0';               -- Push Count is full when 12
L_PopCount <= 11 - L_PushCount;
L_Push_ready <= '1' when ( L_IR_data(to_integer(L_PushCount)) = '1' ) else '0';
L_Pop_ready <= '1' when ( L_IR_data(to_integer(L_PopCount)) = '1' ) else '0';


InstructionRegister: process(clk)
begin
          if rising_edge(clk) then
                    if ( L_IR_ld = '1' ) then
                              L_IR_data <= I_instROM_data;
                    end if;
          end if;
end process;
L_PC_offset <= unsigned(L_IR_data(7 downto 0));
Q_RF_IR_data <= L_IR_data(7 downto 0);


StateUpdate: process(clk)
begin
          if rising_edge(clk) then
                    current_state <= next_state;
          end if;
end process;


StateMachine: process(current_state, L_IR_Opcode, I_RF_Preg_isZero, I_program_en, L_PushCount_full, L_Push_ready, L_Pop_ready, I_button, L_PC_full)
begin
          -- State Default
          next_state <= current_state;

          --Local signal defaults
          L_IR_ld <= '0';
          L_PC_clr <= '0';
          L_PC_inc <= '0';
          L_PC_ld <= '0';
          L_PC_dir <= '0';
          L_PC_restore <= '0';
          L_LR_ld <= '0';
          L_LR_clr <= '0';
          L_SP_clr <= '0';
          L_SP_inc <= '0';
          L_SP_dec <= '0';
          L_PushCount_clr <= '0';
          L_PushCount_inc <= '0';

          --Output defaults
          Q_RF_Preg_rd <= '0';
          Q_RF_Qreg_rd <= '0';
          Q_RF_Wreg_sel <= "00";
          Q_RF_Wreg_wr <= '0';
          Q_M_wr <= '0';
          Q_ALU_sel <= "00";

          case current_state is

                    when sIDLE =>
                              L_PC_clr <= '1';
                              L_LR_clr <= '1';
                              L_SP_clr <= '1';
                              if ( I_program_en = '0' ) then
                                        next_state <= sWAIT_XROM;
                              end if;


                    when sWAIT_XROM =>
                              if I_button = '1' then
                                        next_state <= sFETCH;
                              end if;

                    when sFETCH =>
                              L_IR_ld <= '1';
                              next_state <= sDECODE;

                    when sDECODE =>
                              Q_RF_Preg_rd <= '1';
                              Q_RF_Qreg_rd <= '1';
                              next_state <= sWAIT_MRAM;

                    when sWAIT_MRAM =>

                              case L_IR_Opcode is
                                        when x"0" => next_state <= sNO_OP;
                                        when x"1" => next_state <= sLOAD_DIR;
                                        when x"2" => next_state <= sSTORE_DIR;
                                        when x"3" => next_state <= sMOVE;
                                        when x"4" => next_state <= sCONST;
                                        when x"5" => next_state <= sADD;
                                        when x"6" => next_state <= sSUB;
                                        when x"7" => next_state <= sBZ;
                                        when x"8" => next_state <= sBD;
                                        when x"9" => next_state <= sBL;
                                        when x"A" => next_state <= sLOAD_IND;
                                        when x"B" => next_state <= sSTORE_IND;
                                        when x"C" => next_state <= sPUSH1;
                                        when x"D" => next_state <= sPOP1;
                                        when others => next_state <= sPC_INC;
                              end case;
```

```
when sLOAD_DIR =>
           Q_RF_Wreg_wr <= '1';
           Q_RF_Wreg_sel <= "01";
           if L_PC_full = '1' then
                      next_state <= sFINISHED;
           else next_state <= sPC_INC;
           end if;

when sSTORE_DIR =>
           Q_M_wr <= '1';
           if L_PC_full = '1' then
                      next_state <= sFINISHED;
           else next_state <= sPC_INC;
           end if;

when sMOVE =>
           Q_RF_Wreg_wr <= '1';
           if L_PC_full = '1' then
                      next_state <= sFINISHED;
           else next_state <= sPC_INC;
           end if;

when sCONST =>
           Q_RF_Wreg_sel <= "10";
           Q_RF_Wreg_wr <= '1';
           if L_PC_full = '1' then
                      next_state <= sFINISHED;
           else next_state <= sPC_INC;
           end if;

when sADD =>
           Q_ALU_sel <= "01";
           Q_RF_Wreg_wr <= '1';
           if L_PC_full = '1' then
                      next_state <= sFINISHED;
           else next_state <= sPC_INC;
           end if;

when sSUB =>
           Q_ALU_sel <= "10";
           Q_RF_Wreg_wr <= '1';
           if L_PC_full = '1' then
                      next_state <= sFINISHED;
           else next_state <= sPC_INC;
           end if;

when sBZ =>
           if ( I_RF_Preg_isZero = '1' ) then
                      next_state <= sPC_JUMP;
           else
                      if L_PC_full = '1' then
                                 next_state <= sFINISHED;
                      else next_state <= sPC_INC;
                      end if;
           end if;

when sBD =>
           next_state <= sPC_DIRECT;

when sBL =>
           L_LR_ld <= '1';
           next_state <= sPC_DIRECT;

when sLOAD_IND =>
           Q_RF_Wreg_wr <= '1';
           Q_RF_Wreg_sel <= "01";
           if L_PC_full = '1' then
                      next_state <= sFINISHED;
           else next_state <= sPC_INC;
           end if;

when sSTORE_IND =>
           Q_M_wr <= '1';
           if L_PC_full = '1' then
                      next_state <= sFINISHED;
           else next_state <= sPC_INC;
           end if;

when sNO_OP =>
           if L_PC_full = '1' then
                      next_state <= sFINISHED;
           else next_state <= sPC_INC;
           end if;

when sPC_INC =>
           L_PC_inc <= '1';
           next_state <= sWAIT_XROM;

when sPC_JUMP =>
           L_PC_ld <= '1';
           next_state <= sWAIT_XROM;

when sPC_DIRECT =>
           L_PC_dir <= '1';
           next_state <= sWAIT_XROM;

when sPC_RESTORE =>
```

```
                                L_PC_restore <= '1';
                                next_state <= sWAIT_XROM;

                        when sPUSH1 =>
                                Q_RF_Preg_rd <= '1';
                                next_state <= sPUSH2;

                        when sPUSH2 =>
                                L_PushCount_inc <= '1';
                                if ( L_Push_ready = '1' ) then
                                        Q_M_wr <= '1';
                                        L_SP_inc <= '1';
                                end if;

                                if ( L_PushCount_full = '1' ) then
                                        next_state <= sPC_INC;
                                else
                                        next_state <= sPUSH1;
                                end if;

                        when sPOP1 =>
                                Q_RF_Wreg_sel <= "01";                          -- Wreg gets data from Memory
                                if ( L_Pop_ready = '1' ) then
                                        L_SP_dec <= '1';
                                        next_state <= sPOP2;
                                else
                                        next_state <= sPOP3;
                                end if;

                        when sPOP2 =>
                                Q_RF_Wreg_sel <= "01";
                                next_state <= sPOP3;

                        when sPOP3 =>
                                Q_RF_Wreg_sel <= "01";
                                L_PushCount_inc <= '1';
                                if ( L_Pop_ready = '1' ) then
                                        Q_RF_Wreg_wr <= '1';
                                end if;

                                if (L_PushCount_full = '1') then
                                        next_state <= sPC_RESTORE;
                                else
                                        next_state <= sPOP1;
                                end if;

                        when sFINISHED =>

                        when others =>
                                next_state <= sPC_INC_pre;
                end case;

                if ( I_program_en = '1' ) then
                        next_state <= sIDLE;
                end if;

 end process;


end Behavioral;
```

Datapath

```
-- Class:            ENGS 128 15S
-- Engineer:         Brett Nicholas and Matt Metzler
--
-- Create Date:      05/27/2015 03:43:05 PM
-- Design Name:      Final Project
-- Module Name:      Datapath - Behavioral
-- Project Name:     MyRISC
-- Target Devices:   Xilinx Arctix7 FPGA on Digilent BASYS3 project board
-- Tool Versions:    Xilinx Vivado
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Datapath is
Port (    clk                   : in std_logic;
          I_RF_M_data           : in std_logic_vector (15 downto 0);        -- data from BRAM into datapath
          I_RF_IR_data          : in std_logic_vector (7 downto 0);         -- Bottom 8 bits from the instruction register (for load constant)
          I_RF_Wreg_addr        : in std_logic_vector (3 downto 0);         -- Address of register to be written to (Wreg)
          I_RF_Wreg_wr          : in std_logic;                             -- write enable for write register (Wreg)
          I_RF_Wreg_sel         : in std_logic_vector (1 downto 0);         -- Mux select for inputs to register file
          I_RF_Preg_addr        : in std_logic_vector (3 downto 0);         -- Address of primary read register (Preg)
          I_RF_Preg_rd          : in std_logic;                             -- read enable for Preg
          I_RF_Qreg_addr        : in std_logic_vector (3 downto 0);         -- Address of secondary read register (Qreg - only used for arithmetic functions)
          I_RF_Qreg_rd          : in std_logic;                             -- read enable for Qreg
          I_ALU_sel             : in std_logic_vector (1 downto 0);         -- ALU MUX: 00=>Preg pass through, 01=>Preg+Qreg, 10=>Preg-Qreg
          I_reg_sel             : in std_logic_vector (3 downto 0);         -- Select which register to display
          Q_M_data              : out std_logic_vector (15 downto 0);       -- data from datapath into BRAM
          Q_RF_Preg_isZero      : out std_logic;                            -- flag indicating Preg is zero (used for jump if zero)
          Q_M_addr              : out std_logic_vector (7 downto 0);        -- Memory address for inderect access
          Q_reg_out             : out std_logic_vector (15 downto 0)        -- register output for displaying
          );
end Datapath;


architecture Behavioral of Datapath is

          -- Structure for register file
          type Register_File is array(0 to 15) of std_logic_vector(15 downto 0);
                    signal L_RF_Register  : Register_File := (others => (others => '0'));


          signal L_RF_Preg_data : unsigned(15 downto 0) := x"0000";
          signal L_RF_Qreg_data : unsigned(15 downto 0) := x"0000";
          signal L_ALU_data     : unsigned(15 downto 0) := x"0000";
          signal L_RF_Wreg_data : std_logic_vector(15 downto 0) := x"0000";

begin


--Select register to display
Q_reg_out <= L_RF_Register(to_integer(unsigned(I_reg_sel)));


WregDataMux: process(I_RF_IR_data, I_RF_M_data, L_ALU_data, I_RF_Wreg_sel)
begin
          case I_RF_Wreg_sel is
                    when "00" =>
                              L_RF_Wreg_data <= std_logic_vector(L_ALU_data);
                    when "01" =>
                              L_RF_Wreg_data <= I_RF_M_data;
                    when "10" =>
                              L_RF_Wreg_data <= x"00" & I_RF_IR_data;
                    when others =>
                              L_RF_Wreg_data <= x"0000";
          end case;
end process;


RegisterWrite: process(clk)
begin
          if rising_edge(clk) then
                    if ( I_RF_Wreg_wr = '1' ) then
                              L_RF_Register(to_integer(unsigned(I_RF_Wreg_addr))) <= L_RF_Wreg_data;
                    end if;
          end if;
end process;

PregQreg: process(clk)
begin
```

```
          if rising_edge(clk) then
                    -- Register Preg from the array
                    if ( I_RF_Preg_rd = '1' ) then
                              L_RF_Preg_data <= unsigned(L_RF_Register(to_integer(unsigned(I_RF_Preg_addr))));
                    end if;
                    -- Register Qreg from the array
                    if ( I_RF_Qreg_rd = '1' ) then
                              L_RF_Qreg_data <= unsigned(L_RF_Register(to_integer(unsigned(I_RF_Qreg_addr))));
                    end if;
          end if;
 end process;


ArithmeticLogicUnit: process(L_RF_Preg_data, L_RF_Qreg_data, I_ALU_sel)
 begin
          case I_ALU_sel is
                    when "00" =>                                                          -- Preg pass through
                              L_ALU_data <= L_RF_Preg_data;
                    when "01" =>                                                          -- Preg + Qreg
                              L_ALU_data <= L_RF_Preg_data + L_RF_Qreg_data;
                    when "10" =>                                                          -- Preg - Qreg
                              L_ALU_data <= L_RF_Preg_data - L_RF_Qreg_data;
                    when others =>
                              L_ALU_data <= x"0000";
          end case;
 end process;


 -- Asynchronous Preg Zero flag
 Q_RF_Preg_isZero <= '1' when L_RF_Preg_data = x"0000" else '0';


 -- Data written to memory always comes from Preg
 Q_M_data <= std_logic_vector(L_RF_Preg_data);
 Q_M_addr <= std_logic_vector(L_RF_Qreg_data(7 downto 0));


 end Behavioral;
```

Instruction Buffer

```vhdl
-- Class:            ENGS 128 15S
-- Engineer:         Brett Nicholas and Matt Metzler
-- S
-- Create Date:      05/27/2015 03:43:05 PM
-- Design Name:      Final Project
-- Module Name:      InstructionBuffer - Behavioral
-- Project Name:     MyRISC
-- Target Devices:   Xilinx Arctix7 FPGA on Digilent BASYS3 project board
-- Tool Versions:    Xilinx Vivado
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;


-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
library UNISIM;
use UNISIM.VComponents.all;


entity InstructionBuffer is
Port (    clk                         : in std_logic;
          I_RX_done_tick              : in std_logic;                            -- SPI conversion done tick
          I_RX_data                   : in std_logic_vector (7 downto 0);        -- SPI parallel data
          I_program_en                : in std_logic;                            -- Enables programming mode (no program execution during this time)
          Q_instruction_word          : out std_logic_vector (15 downto 0);      -- 16-bit instruction to be stored in XRAM
          Q_X_addr                    : out std_logic_vector (7 downto 0);       -- XRAM write address
          Q_X_wr_en                   : out std_logic;                           -- XRAM write enable
          Q_error_start               : out std_logic                            -- Start the error sequence
          );
end InstructionBuffer;


architecture Behavioral of InstructionBuffer is

-- converts 8-bit ascii code for 0->f to hex representation
function ascii2hex (a: std_logic_vector(7 downto 0)) return std_logic_vector is
          variable tmp: std_logic_vector(3 downto 0) := x"0";
begin
          case a is
                    when x"30" => tmp := x"0";      -- 0
                    when x"31" => tmp := x"1";      -- 1
                    when x"32" => tmp := x"2";      -- 2
                    when x"33" => tmp := x"3";      -- 3
                    when x"34" => tmp := x"4";      -- 4
                    when x"35" => tmp := x"5";      -- 5
                    when x"36" => tmp := x"6";      -- 6
                    when x"37" => tmp := x"7";      -- 7
                    when x"38" => tmp := x"8";      -- 8
                    when x"39" => tmp := x"9";      -- 9
                    when x"61" => tmp := x"a";      -- a
                    when x"62" => tmp := x"b";      -- b
                    when x"63" => tmp := x"c";      -- c
                    when x"64" => tmp := x"d";      -- d
                    when x"65" => tmp := x"e";      -- e
                    when x"66" => tmp := x"f";      -- f
                    when others => tmp := x"0";
          end case;
          return tmp;
end ascii2hex;

-- ASCII character constants
constant NUL                          : std_logic_vector (7 downto 0) := x"00";
constant CR                           : std_logic_vector (7 downto 0) := x"0d";
constant BS                           : std_logic_vector (7 downto 0) := x"08";

-- character buffer shift register file
constant iCHARBUF_DEPTH               : integer := 4;                            -- max = 16 characters deep
constant iCHARBUF_WIDTH               : integer := 8;                            -- character = 8 bits
type regfile_type is array(iCHARBUF_DEPTH-1 downto 0) of std_logic_vector(iCHARBUF_WIDTH-1 downto 0);
signal L_charBuf                      : regfile_type := (others => NUL);         -- character buffer shift register file
signal L_charBuf_shift_en             : std_logic := '0';                        -- shift enable signal
signal L_charBuf_clr                  : std_logic := '0';                        -- Clear character buffer
signal L_charBuf_isFull               : std_logic := '0';                        -- Character buffer is full

-- character buffer shift counter
signal L_charBuf_shift_ctr            : integer range 0 to 4 := 0;               -- shift counter


    -- XRAM
    constant iXRAM_ADDR_WIDTH          : integer := 8;      -- width of instruction RAM address
```

```
     constant iXRAM_DEPTH                   : integer := 200;     -- depth of instruction RAM
     signal L_XRAM_wr_addr                  : unsigned(iXRAM_ADDR_WIDTH-1 downto 0) := (others=>'0'); --
     signal L_XRAM_wr_ctr_en                : std_logic := '0';
           signal L_XRAM_wr_ctr_clr         : std_logic := '0';

     -- input register
     signal L_rx_data_reg                   : std_logic_vector (7 downto 0) := (others => '0');
     signal L_CR_detected                   : std_logic := '0';              -- input character = carriage return
     signal L_BS_detected                   : std_logic := '0';              -- input character = backspace
     signal L_rx_data_ready                 : std_logic := '0';              -- input data is ready
     signal L_inst_reg_en                   : std_logic := '0';

     -- FSM state types
     type state_type is (sIDLE, sWAIT, sSHIFT, sWRITE, sWRITE2, sERROR, sSETUP);
     signal current_state, next_state: state_type;

begin


 -- Register input character on RX_done_tick
 input_reg: process(clk)
 begin
           if rising_edge(clk) then
                      if (I_rx_done_tick = '1') then
                                 L_rx_data_reg <= I_rx_data;
                                 L_rx_data_ready <= '1'; -- data ready flag
                      else
                                 L_rx_data_ready <= '0'; -- data not ready
                      end if;
           end if;
 end process;
 L_CR_detected <= '1' when L_rx_data_reg = CR else '0'; -- Async watchdog to detect CR
 L_BS_detected <= '1' when L_rx_data_reg = BS else '0'; -- Async watchdog to detect CR


-- Shifts characters into the char buffer on the enable signal. Also contains shift counter
shift_reg: process(clk, L_charBuf_shift_en, L_charBuf_clr)
begin
           if rising_edge(clk) then
                      if (L_charBuf_clr = '1') then
                                 L_charBuf <= (others => NUL); -- clear char buffer
                                 L_charBuf_shift_ctr <= 0;           -- clear shift counter
                      elsif (L_charBuf_shift_en = '1') then
                                 L_charBuf(L_charBuf_shift_ctr) <= L_rx_data_reg; -- shift in character
                                 L_charBuf_shift_ctr <= L_charBuf_shift_ctr + 1; -- Shift count ++
                      end if;
     end if;
 end process;
 L_charBuf_isFull <= '1' when L_charBuf_shift_ctr = 4 else '0'; -- ASYNC watchdog for full character buffer


 output_reg: process(clk, L_inst_reg_en)
 begin
           if rising_edge(clk) then
                      if(L_inst_reg_en = '1') then
                                 Q_instruction_word <= ascii2hex(L_charBuf(0)) & ascii2hex(L_charBuf(1)) & ascii2hex(L_charBuf(2)) & ascii2hex(L_charBuf(3));
                      end if;
           end if;
 end process;


 -- Counter for XRAM write address
 write_counter: process(clk, L_XRAM_wr_ctr_clr, L_XRAM_wr_ctr_en)
 begin
     if rising_edge(clk) then
         if (L_XRAM_wr_ctr_clr = '1') then
             L_XRAM_wr_addr <= to_unsigned(0, L_XRAM_wr_addr'length);
         elsif (L_XRAM_wr_ctr_en = '1') then
             L_XRAM_wr_addr <= L_XRAM_wr_addr + 1;
         end if;
     end if;
 end process;
 Q_X_addr <= std_logic_vector(L_XRAM_wr_addr);


-----------------------------------------------------------------
---------------------- FSM CONTROLLER --------------------------
-----------------------------------------------------------------
StateUpdate: process(clk)
begin
           if rising_edge(clk) then
                      current_state <= next_state;
           end if;
 end process;


StateMachine: process(current_state, L_rx_data_ready, L_CR_detected, I_program_en, L_BS_detected, L_charBuf_isFull)
begin
     -- Defaults
     L_XRAM_wr_ctr_clr <= '0';
     L_XRAM_wr_ctr_en <= '0';
     L_charBuf_shift_en <= '0';
     L_charBuf_clr <= '0';
     L_inst_reg_en <= '0';
     Q_X_wr_en <= '0';
     Q_error_start <= '0';
           next_state <= current_state;
```

```
    case current_state is
        -- idle until set into programming mode
        when sIDLE =>
                            L_charBuf_clr <= '1';
                if (I_program_en = '1') then
                            next_state <= sSETUP;
                end if;

        -- Clear the XRAM write address
        when sSETUP =>
                 L_XRAM_wr_ctr_clr <= '1';
                 next_state <= sWAIT;

        -- wait for new character on rx_done_tick
        when sWAIT =>
                if (I_program_en = '0') then -- go to idle b/c execute mode begins
                            next_state <= sIDLE;
                            elsif (L_rx_data_ready = '1') then
                                        if (L_BS_detected = '1') then
                                                    next_state <= sERROR;
                                        elsif (L_CR_detected = '1') then
                                                    if (L_charBuf_isFull = '1') then
                                                                next_state <= sWRITE;
                                                    else
                                                                next_state <= sERROR;
                                                    end if;
                                        elsif (L_charBuf_isFull = '1') then
                                                    next_state <= sERROR;
                                        else
                                                    next_state <= sSHIFT;
                                        end if;
                            end if;

                -- Shift received character into register file
        when sSHIFT =>
                            L_charBuf_shift_en <= '1';
                            next_state <= sWAIT;

                -- register instruction in output register
                when sWRITE =>
                            L_inst_reg_en <= '1'; -- register instruction
                            L_charBuf_clr <= '1'; -- clear character buffer now that command has been
                            next_state <= sWRITE2;

                -- write contents of output register to XRAM
                when sWRITE2 =>
                            Q_X_wr_en <= '1'; -- write output register to memory
                            L_XRAM_wr_ctr_en <= '1'; -- increment write counter
                            next_state <= sWAIT;

                when sERROR =>
                            Q_error_start <= '1';
                            L_charBuf_clr <= '1';
                            next_state <= sWAIT;

                when others =>
                            next_state <= sWAIT;

     end case;
 end process;

end Behavioral;
```

Error Generator

```
 -- Class:           ENGS 128 15S
 -- Engineer:        Brett Nicholas and Matt Metzler
 --
 -- Create Date:     05/27/2015 03:43:05 PM
 -- Design Name:     Final Project
 -- Module Name:     errorgen - Behavioral
 -- Project Name:    MyRISC
 -- Target Devices:  Xilinx Arctix7 FPGA on Digilent BASYS3 project board
 -- Tool Versions:   Xilinx Vivado
 -- Description:
 --
 -- Dependencies:
 --
 -- Revision:
 -- Revision 0.01 - File Created
 -- Additional Comments:
 --
 ----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

 -- Uncomment the following library declaration if using
 -- arithmetic functions with Signed or Unsigned values
 --use IEEE.NUMERIC_STD.ALL;


 -- Uncomment the following library declaration if instantiating
 -- any Xilinx leaf cells in this code.
 --library UNISIM;
 --use UNISIM.VComponents.all;


entity errorgen is
Port (     clk                  : in std_logic;
           I_error_start        : in std_logic;
           I_tx_done_tick       : in std_logic;
           Q_tx_start           : out std_logic;
           Q_tx_data            : out std_logic_vector (7 downto 0));
 end errorgen;


 architecture Behavioral of errorgen is

           type string_type is array(0 to 5) of std_logic_vector(7 downto 0);
                      signal L_string : string_type;

           type state_type is (sIDLE, sSTART, sWAIT, sCOUNT);
                      signal current_state, next_state : state_type := sIDLE;

           --Counter signals
           signal L_count        : integer range 0 to 5;
           signal L_counter_inc  : std_logic := '0';
           signal L_counter_clr  : std_logic := '0';
           signal L_counter_full : std_logic := '0';

 begin

 --String Declarations
 L_string(0) <= x"45";
 L_string(1) <= x"52";
 L_string(2) <= x"52";
 L_string(3) <= x"4f";
 L_string(4) <= x"52";
 L_string(5) <= x"0d";


 --Output Assignment
 Q_tx_data <= L_string(L_count);


 Counter: process(clk)
 begin
           if rising_edge(clk) then
                      if (L_counter_clr = '1') then
                                 L_count <= 0;
                      elsif (L_counter_inc = '1') then
                                 L_count <= L_count + 1;
                      end if;
           end if;
 end process;
 L_counter_full <= '1' when (L_count = 5) else '0';


 StateUpdate: process(clk)
 begin
           if rising_edge(clk) then
                      current_state <= next_state;
           end if;
 end process;



StateMachine: process(current_state, I_error_start, I_tx_done_tick, L_counter_full)
```

```
begin

           next_state <= current_state;
           Q_tx_start <= '0';
           L_counter_inc <= '0';
           L_counter_clr <= '0';

           case current_state is

                      when sIDLE =>
                                 L_counter_clr <= '1';
                                 if I_error_start = '1' then
                                            next_state <= sSTART;
                                 end if;

                      when sSTART =>
                                 Q_tx_start <= '1';
                                 next_state <= sWAIT;

                      when sWAIT =>
                                 if (I_tx_done_tick = '1') then
                                            if (L_counter_full = '1') then
                                                       next_state <= sIDLE;
                                            else
                                                       next_state <= sCOUNT;
                                            end if;
                                 end if;

                      when sCOUNT =>
                                 L_counter_inc <= '1';
                                 next_state <= sSTART;

                      when others =>
                                 next_state <= sIDLE;

           end case;
 end process;


end Behavioral;
```

Serial Receiver

```
-- Company:          Engs 31 09X
-- Engineer:         E.W. Hansen
--
-- Create Date:      18:29:23 07/19/2008
-- Design Name:
-- Module Name:      SerialRx - Behavioral
-- Project Name:     Lab 5
-- Target Devices:   Spartan 3E / Nexys 2
-- Tool versions:    Foundation ISE 11.2
-- Description:      Serial asynchronous receiver for Nexys 2 RS-232 port
--
-- Dependencies:
--
-- Revision:
-- Revision
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use ieee.math_real.all;


entity SerialRx is
    Port ( clk : in  STD_LOGIC;
           RsRx  : in  STD_LOGIC;                                                    -- received bit stream
           Q_rx_data : out  STD_LOGIC_VECTOR (7 downto 0);      -- data byte
           Q_rx_done_tick : out  STD_LOGIC );                                -- data ready tick
end SerialRx;


architecture Behavioral of SerialRx is
          constant CLOCK_FREQUENCY : integer := 100000000;       -- 100 MHz clock
          constant BAUD_RATE : integer := 115200;                -- baud rate
          constant BAUD_RATE_COUNT : integer := CLOCK_FREQUENCY / BAUD_RATE; -- clocks per bit
          constant TICK_CTR_WIDTH : integer := integer(CEIL(LOG2(real(BAUD_RATE_COUNT))));

          signal rx_sync:          std_logic_vector(1 downto 0) := "11";                  -- double synchronizer
          signal rx_bit:     std_logic;                                           -- synchronized bit stream
          signal rx_reg:     std_logic_vector(9 downto 0) := (others=>'0');          -- start & data & stop


          signal tick_ctr:      unsigned(TICK_CTR_WIDTH-1 downto 0) := (others=>'0');   -- count clock ticks to find bits
          signal bit_ctr:       unsigned(3 downto 0) := x"0";              -- count received bits (0 to 10)
          signal rx_sample: std_logic;
          signal rx_shift:  std_logic;
          signal rx_reg_full: std_logic;
          signal rx_output:     std_logic;
          signal rx_init:                  std_logic;            -- load bit counter with 8 and reset shift counter

          type state_type is (s0, s1, s2, s3, s4);
          signal curr_state, next_state : state_type;
begin

Synchronizer:                                                                                   -- Double flop synchronizer
process(Clk)
begin
          if rising_edge(Clk) then
                      rx_sync <= RsRx & rx_sync(1);
          end if;
end process Synchronizer;
rx_bit <= rx_sync(0);                                                          -- sync'd output is the lsb of the pair


TickCounter:
process(Clk)
begin
          if rising_edge(Clk) then
                     rx_sample <= '0';
                     if rx_init='1' then
                                tick_ctr <= to_unsigned(BAUD_RATE_COUNT/2, TICK_CTR_WIDTH);
                     elsif tick_ctr = 0 then
                                rx_sample <= '1';
                                tick_ctr <= to_unsigned(BAUD_RATE_COUNT, TICK_CTR_WIDTH);
                     else
                                tick_ctr <= tick_ctr-1;
                     end if;
          end if;
end process TickCounter;









DataRegisters:
process(Clk)
```

```
begin
	if rising_edge(Clk) then
		if rx_shift='1' then		-- shift register
			rx_reg <= rx_bit & rx_reg(9 downto 1);
		end if;

		if rx_output='1' then		-- output register
			Q_rx_data <= rx_reg(8 downto 1);
		end if;
	end if;
end process DataRegisters;


BitCounter:
process(Clk, bit_ctr)
begin
	if rising_edge(Clk) then
		if rx_init = '1' then
			bit_ctr <= x"0";
		else
			if rx_shift='1' then
				bit_ctr <= bit_ctr+1;
			end if;
		end if;
	end if;

	rx_reg_full <= '0';
	if bit_ctr = "1010" then
		rx_reg_full <='1';
	end if;
end process BitCounter;


RxControllerReg:
process(Clk)
begin
	if rising_edge(Clk) then
		curr_state <= next_state;
	end if;
end process RxControllerReg;


RxControllerComb:
process(rx_bit, rx_sample, rx_reg_full, curr_state)
begin
	-- defaults
	rx_init <= '0'; rx_shift <= '0';
	rx_output <= '0'; Q_rx_done_tick <= '0';
	next_state <= curr_state;
	--

	case curr_state is
		when s0 => rx_init <= '1';				-- wait for start bit
			if rx_bit = '0'
				then next_state <= s1;
			end if;

		when s1 =>
			if rx_reg_full='1'						-- after 10 bits have been shifted
				then next_state <= s3;					-- go to the output state
			elsif rx_sample='1'
				then next_state <= s2;					-- go to the shift state
			end if;

		when s2 => rx_shift <= '1';				-- shift the register
			next_state <= s1;

		when s3 => rx_output <= '1';				-- transfer to the output register
			next_state <= s4;

		when s4 => Q_rx_done_tick <= '1';			-- tell we're done
			next_state <= s0;

		when others =>
			next_state <= s0;
	end case;
end process RxControllerComb;


end Behavioral;
```

Serial Transmitter

```vhdl
-- Company:                              Engs 31 14X
-- Engineer:                             E.W. Hansen
--
-- Create Date:      12:55:02 07/19/2008
-- Design Name:                 Lab 5
-- Module Name:      SerialTx - Behavioral
-- Project Name:                RS232
-- Target Devices:   Spartan 6 / Nexys 3
-- Tool versions:    Foundation ISE 14.4
-- Description:                 Serial asynchronous transmitter for Pmod RS-232 port
--
-- Revision:
-- Revision 0.01 - File Created
--        Rev (EWH) 7.17.2014, no external baud rate generator
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.math_real.all;


entity SerialTx is
    Port ( clk : in  STD_LOGIC;
           tx_data : in  STD_LOGIC_VECTOR (7 downto 0);
           tx_start : in  STD_LOGIC;
           tx : out  STD_LOGIC;                                  -- to Nexys 2 RS-232 port
           tx_done_tick : out  STD_LOGIC);
end SerialTx;


architecture Behavioral of SerialTx is
          constant CLOCK_FREQUENCY : integer := 100000000;
          constant BAUD_RATE : integer := 115200;
          constant BAUD_COUNT : integer := CLOCK_FREQUENCY / BAUD_RATE;
          constant TICK_CTR_WIDTH : integer := integer(CEIL(LOG2(real(BAUD_COUNT))));

          signal br_cnt:                   unsigned(TICK_CTR_WIDTH-1 downto 0) := (others=>'0');  -- baud rate counter

                     -- 12 bits can handle 4800 baud at 10 MHz clock
          signal br_tick:    std_logic;
          signal tx_reg:     std_logic_vector(9 downto 0) := "1111111111";   -- 1 start bit, 8 data bits, 1 stop bit, no parity
          signal tx_ctr:     unsigned(3 downto 0);                           -- count the bits that have been sent
          signal tx_load, tx_shift : std_logic;                                         -- register control bits
          signal tx_empty:   std_logic;                                                     -- register status bit
          type state_type is (sidle, ssync, sload, sshift, sdone, swait);   -- state machine
          signal curr_state, next_state: state_type;
begin


BaudRateClock:
process(Clk)
begin
          if rising_edge(Clk) then
                     if br_cnt = BAUD_COUNT-1 then
                                br_cnt <= (others=>'0');
                                br_tick <= '1';
                     else
                                br_cnt <= br_cnt+1;
                                br_tick <= '0';
                     end if;
          end if;
end process BaudRateClock;


DataRegister:
process( Clk )
begin
          if rising_edge( Clk ) then
                     if (tx_load = '1') then
                                tx_reg <= '1' & tx_data & '0';                          -- load with stop & data & start
                                                                                                             -- the register is always shifting
                     elsif br_tick = '1' then
                                tx_reg <= '1' & tx_reg(9 downto 1);                     -- shift right

                     end if;
          end if;
end process DataRegister;
tx <= tx_reg(0);                                                                                                                   -- serial output port <= lsb


ShiftCounter:
process ( Clk )
begin
          if rising_edge( Clk ) then
                     if (tx_load = '1') then                                                 -- load counter with 10 when register is loaded
                                tx_ctr <= x"A";
                     elsif br_tick = '1' then                                     -- count shifts (br_ticks) down to 0
                                if (tx_shift = '1') then
                                           if tx_ctr > 0 then
                                                      tx_ctr <= tx_ctr - 1;
                                           end if;
                                end if;
                     end if;
```

```
		end if;
	end process ShiftCounter;
	tx_empty <= '1' when tx_ctr = x"0" else '0';


	TxControllerComb:
	process ( tx_start, tx_empty, br_tick, curr_state )
	begin
		-- defaults
		next_state <= curr_state;
		tx_load <= '0';  tx_shift <= '0'; tx_done_tick <= '0';
		-- next state and output logic
		case curr_state is
			when sidle =>
				if tx_start = '1'				-- wait for start signal
					then next_state <= ssync;
				end if;
			when ssync =>								-- sync up with baud rate
				if br_tick = '1'
					then next_state <= sload;
				end if;
			when sload =>		tx_load <= '1';			-- load the data register
				next_state <= sshift;
			when sshift => tx_shift <= '1';			-- shift the bits out
				if tx_empty = '1'				-- wait for shift counter
					then next_state <= sdone;
				end if;
			when sdone => tx_done_tick <= '1';		-- raise the done flag
				next_state <= swait;
			when swait =>								-- wait for start signal to drop
				if tx_start = '0'
					then next_state <= sidle;
				end if;
			when others => next_state <= sidle;
		end case;
	end process TxControllerComb;


	TxControllerReg:
	process ( Clk )
	begin
		if rising_edge(Clk) then
				curr_state <= next_state;
		end if;
	end process TxControllerReg;

	end Behavioral;
```

Monopulser

```
-- Class:            ENGS 128 15S
-- Engineer:         Brett Nicholas and Matt Metzler
--
-- Create Date:      05/27/2015 03:43:05 PM
-- Design Name:      Final Project
-- Module Name:      Monopulse - Behavioral
-- Project Name:     MyRISC
-- Target Devices:   Xilinx Arctix7 FPGA on Digilent BASYS3 project board
-- Tool Versions:    Xilinx Vivado
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;


-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity Monopulse is
Port ( clk            : in std_logic;
       button         : in std_logic;
       mono_out       : out std_logic);
end Monopulse;


architecture Behavioral of Monopulse is

          signal old_button     : std_logic := '0';
          signal new_button     : std_logic := '0';

begin


Registers: process(clk)
begin
          if rising_edge(clk) then
                     old_button <= new_button;
                     new_button <= button;
          end if;
end process;


mono_out <= '1' when (new_button = '1') and (old_button = '0') else '0';




end Behavioral;
```

Debouncer

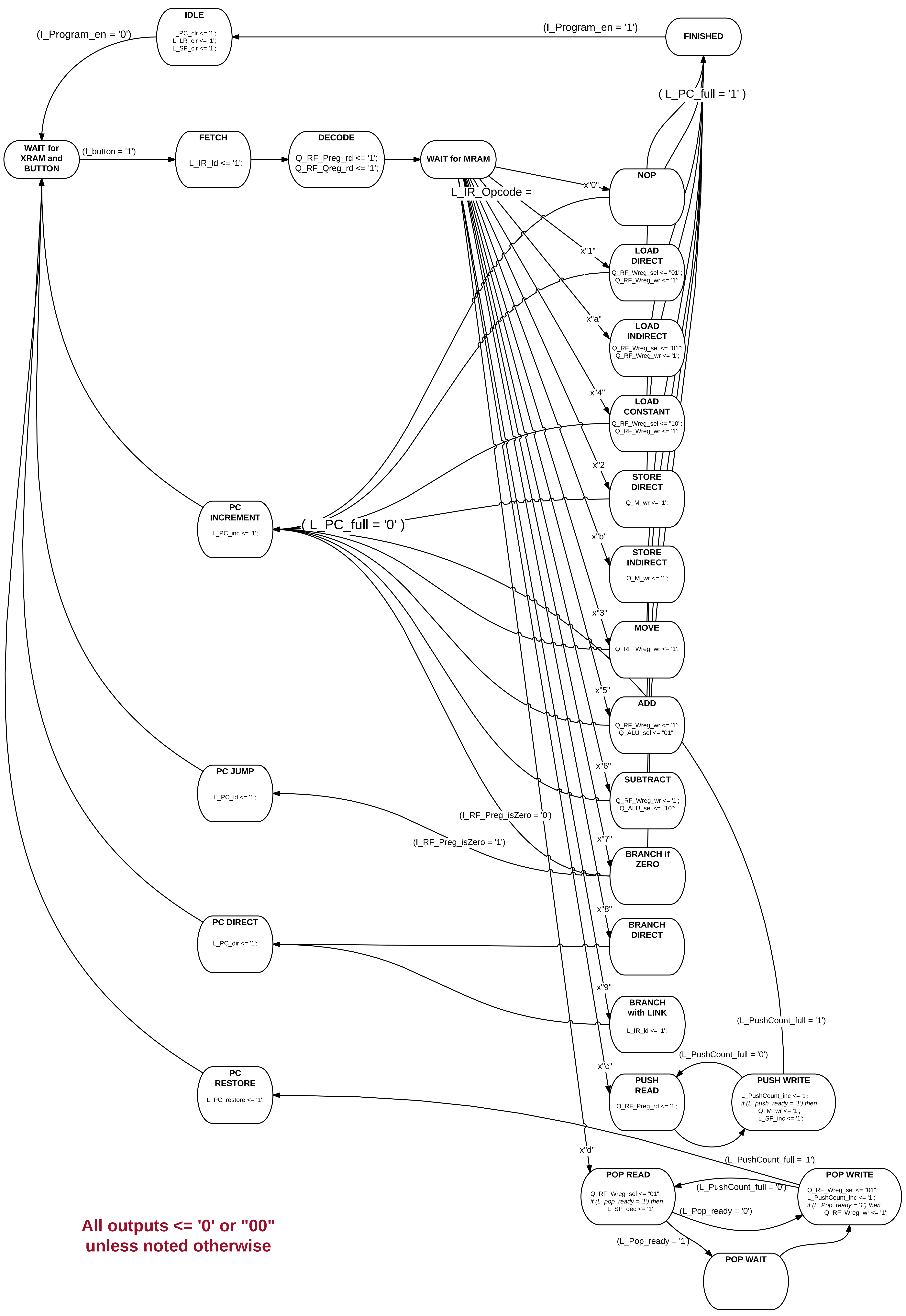
```
LIBRARY eee;
 USE ieee.std_logic_1164.all;
 USE ieee.std_logic_unsigned.all;


 ENTITY debounce IS


   PORT(
     clk     : IN  STD_LOGIC;  -- assumes 25Mhz clock
     button  : IN  STD_LOGIC;  -- input signal to be debounced
     db_out  : OUT STD_LOGIC); -- debounced signal out
 END debounce;


 ARCHITECTURE logic OF debounce IS
   constant NBITS : integer := 18;          -- change this if too fast or too slow
   SIGNAL flipflops   : STD_LOGIC_VECTOR(1 DOWNTO 0); --input flip flops (synchronizer)
   SIGNAL counter_set : STD_LOGIC;                    --sync reset to zero
   SIGNAL counter_out : STD_LOGIC_VECTOR(NBITS DOWNTO 0) := (OTHERS => '0');
      --counter size (19 bits gives approx 10ms delay with 25MHz clock)
 BEGIN


 DFS:  process(clk, flipflops)                                                  -- double flop synchronizer
 begin
           IF rising_edge(clk) THEN
       flipflops <= button & flipflops(1);
     end if;
           counter_set <= flipflops(0) xor flipflops(1);   -- determine when to start/reset counter
 end process DFS;


 -- Wait until the input has been stable (synchronizer holds "00" or "11" for
 -- about 10 msec
 DBtimer:  PROCESS(clk)
 BEGIN
           IF rising_edge(clk) THEN
                      If(counter_set = '1') THEN      -- reset counter because input is still bouncing
           counter_out <= (OTHERS => '0');
 --        ELSIF(counter_out(1) = '0') THEN       -- use for simulation only
       ELSIF(counter_out(NBITS) = '0') THEN   -- use for implementation
           counter_out <= counter_out + 1;
           ELSE                                   --stable input time is met
           db_out <= flipflops(1);
           END IF;
           END IF;
 END PROCESS DBtimer;


 END logic;
```

7-segment display

```
-- Company:          Engs 31 08X
-- Engineer:         E.W. Hansen
--
-- Create Date:      17:56:35 07/25/2008
-- Design Name:
-- Module Name:      mux7seg - Behavioral
-- Project Name:
-- Target Devices:   Digilent Nexys 2 board
-- Tool versions:    Foundation ISE 10.1.01i
-- Description:      Multiplexed seven-segment decoder for the display on the
--                   Nexys 2 board
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity mux7seg is
    Port ( clk : in  STD_LOGIC;
           y0, y1, y2, y3 : in  STD_LOGIC_VECTOR (3 downto 0);   -- digits
           seg : out  STD_LOGIC_VECTOR(0 to 6);                                                -- segments (a...g)
           an : out  STD_LOGIC_VECTOR (3 downto 0) );                                     -- anodes
end mux7seg;


architecture Behavioral of mux7seg is
          constant NCLKDIV: integer := 18;
          constant MAXCLKDIV: integer := 2**NCLKDIV-1;                                     -- max count of clock divider
          signal cdcount:       unsigned(NCLKDIV-1 downto 0) := (others => '0');                    -- clock divider count
          signal CE :           std_logic := '0';                                                                                                   -- clock enable
          signal adcount : unsigned(1 downto 0) := "00";                                                -- anode / mux selector count
          signal muxy : std_logic_vector(3 downto 0);                         -- mux output
          signal segh : std_logic_vector(0 to 6);                                     -- segments (high true)
begin


-- Frequency of CE is Clock Frequency / 2^NCLKDIV = 100 MHz / 2^20, approx 95 Hz
ClockDivider: process(clk)
begin
          if rising_edge(clk) then
             if cdcount < MAXCLKDIV then
                                cdcount <= cdcount+1;
                                CE <= '0';
       else
            cdcount <= (others => '0');
            CE <= '1';
       end if;
    end if;
end process;


AnodeDriver: process(clk, adcount)
begin
          if rising_edge(clk) then
                     if CE='1' then
                                adcount <= adcount + 1;
                     end if;
          end if;

          case adcount is
                     when "00" => an <= "1110";
                     when "01" => an <= "1101";
                     when "10" => an <= "1011";
                     when "11" => an <= "0111";
                     when others => an <= "1111";
          end case;
end process AnodeDriver;


Multiplexer:
process(adcount, y0, y1, y2, y3)
begin
          case adcount is
                     when "00" => muxy <= y0;
                     when "01" => muxy <= y1;
                     when "10" => muxy <= y2;
                     when "11" => muxy <= y3;
                     when others => muxy <= x"0";
          end case;
end process Multiplexer;




-- Seven segment decoder
with muxy select segh <=
          "1111110" when x"0",             -- active-high definitions
          "0110000" when x"1",
          "1101101" when x"2",
          "1111001" when x"3",
          "0110011" when x"4",
          "1011011" when x"5",
          "1011111" when x"6",
          "1110000" when x"7",
          "1111111" when x"8",
          "1111011" when x"9",
          "1110111" when x"a",
          "0011111" when x"b",
          "1001110" when x"c",
          "0111101" when x"d",
          "1001111" when x"e",
          "1000111" when x"f",
          "0000000" when others;
 seg <= not(segh);                                     -- Convert to active-low


end Behavioral;
```

Appendix C: Control Unit FSM Diagram

CONTROL UNIT FSM



All outputs <= '0' or "00" unless noted otherwise