KJ Architecture

ISA REVIEW

Instruction formats:

R-type:

, pe.			
opcode: 4 bits	rs: 2 bits rd: 2		rd: 2 bits
B-type:			
opcode: 4 bits	opt: 1 bit	rs: 1 bit	rt: 2 bits
A-type:			
opcode: 4 bits	opt: 1 bit rs: 3 bits		
J-type:			
opcode: 4 bits	const: 4 bits		

Operations with R-type format:

instruction	opcode	rs	rd
halt	0	(always) 0	(always) 0
add (add two registers, write to rd, implicitly	1	if rs = 0, then use \$c	if rd = 0, then use \$r1
writes carry out bit to \$c)		if rs = 1, then use \$r1	if rd = 1, then use \$r3
		if rs = 2, then use \$r3	if rd = 2, then use \$r4
		if rs = 3, then use \$r4	if rd = 3, then use \$r5
diff (find absolute difference rs between rd	2	if rs = 0, then use \$r1	if rd = 0, then use \$r3
where rs and rd are signed, write to rs)		if rs = 1, then use \$r2	if rd = 1, then use \$r4
		if rs = 2, then use \$r6	if rd = 2, then use \$r5
		if rs = 3, then use \$r8	if rd = 3, then use \$r6
movl (move data in rs to rd)	3	if rs = 0, then use \$r1	if rd = 0, then use \$r2
		if rs = 1, then use \$r2	if rd = 1, then use \$r4
		if rs = 2, then use \$r3	if rd = 2, then use \$r7
		if rs = 3, then use \$r4	if rd = 3, then use \$r8
movh (move data in rs to rd)	4	if rs = 0, then use \$r5	if rd = 0, then use \$r1
		if rs = 1, then use \$r6	if rd = 1, then use \$r3
		if rs = 2, then use \$r7	if rd = 2, then use \$r7
		if rs = 3, then use \$r8	if rd = 3, then use \$r8
udiff (find absolute difference rs between rd	13	if rs = 0, then use \$r1	if rd = 0, then use \$r3
where rs and rd are unsigned, write to rs)		if rs = 1, then use \$r2	if rd = 1, then use \$r4
		if rs = 2, then use \$r6	if rd = 2, then use \$r5
		if rs = 3, then use \$r8	if rd = 3, then use \$r6
II (shift left logical, write to rd, implicitly writes	15	if rs = 0, then sll 1 bit	if rd = 0, then use \$r2
carry out bit to \$c)		if rs = 1, then sll 2 bits	if rd = 1, then use \$r3
		if rs = 2, then sll 4 bits	if rd = 2, then use \$r4
		if rs = 3, then sll 6 bits	if rd = 3, then use \$r8

Operations with B-type format:

instruction	opcode	opt	rs	rt
be (rs == rt, if true	5	if opt = 0, then compare lower 4 bits of	if rs = 0, then use \$r1	if rt = 0, then use \$r1
then newPC = PC +		operands	if rs = 1, then use \$r2	if rt = 1, then use \$r2
1, else then newPC		if opt = 1, then compare all 8 bits of		if rt = 2, then use \$r3

= PC + 2)		operands		if rt = 3, then use \$r4
bge ((rs ≥ rt), if true	6	if opt = 0, use register set #1	register set #1:	register set #1:
then newPC = PC +		if opt = 1, use register set #2	if rs = 0, then use \$r1	if rt = 0, then use \$r3
1, else then newPC			if rs = 1, then use \$r2	if rt = 1, then use \$r4
= PC + 2)			register set #2:	if rt = 2, then use \$r5
			if rs = 0, then use \$r3	if rt = 3, then use \$r7
			if rs = 1, then use \$r6	register set #2:
				if rt = 0, then use \$r1
				if rt = 1, then use \$r2
				if rt = 2, then use \$r3
				if rt = 3, then use \$r7

Operations with A-type format:

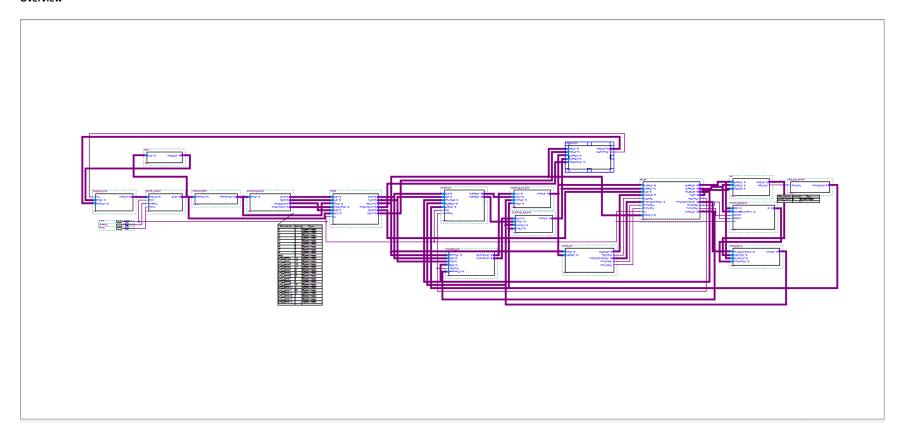
instruction	opcode	opt	r	rs
st (implicitly stores data from	7	if opt = 0, store data to	if rs = 0, then use \$r1	if rs = 4, then use \$r5
rs to memory address in \$r8		memory address	if rs = 1, then use \$r2	if rs = 5, then use \$r6
OR writes \$c to \$temp)		if opt = 1, write \$c to	if rs = 2, then use \$r3	if rs = 6, then use \$r7
		\$temp	if rs = 3, then use \$r4	if rs = 7, then use \$r8
ld (implicitly loads data from	8	if opt = 0, load data from	if rs = 0, then use \$r1	if rs = 4, then use \$r5
memory address in \$8, writes		memory address	if rs = 1, then use \$r2	if rs = 5, then use \$r6
to rs OR writes \$temp to \$c)		if opt = 1, write \$temp to	if rs = 2, then use \$r3	if rs = 6, then use \$r7
		\$c	if rs = 3, then use \$r4	if rs = 7, then use \$r8
addi (increments or	9	if opt = 0, increment	if rs = 0, then use \$r1	if rs = 4, then use \$r5
decrements to rs)		if opt = 1, decrement	if rs = 1, then use \$r2	if rs = 5, then use \$r6
			if rs = 2, then use \$r3	if rs = 6, then use \$r7
			if rs = 3, then use \$r4	if rs = 7, then use \$r8
bz (implicitly uses \$r6 (zero	10	(always) 0	if rs = 0, then use \$r1	if rs = 4, then use \$r5
register). \$r6 == rs, if true			if rs = 1, then use \$r2	if rs = 5, then use \$r6
then newPC = PC + 1, else			if rs = 2, then use \$r3	if rs = 6, then use \$r7
then $newPC = PC + 2$)			if rs = 3, then use \$r4	if rs = 7, then use \$r8
btwo (checks if rs % 2 == 0, if	11	(always) 0	if rs = 0, then use \$r1	if rs = 4, then use \$r5
true then newPC = PC + 1,			if rs = 1, then use \$r2	if rs = 5, then use \$r6
else then newPC = PC + 2)			if rs = 2, then use \$r3	if rs = 6, then use \$r7
			if rs = 3, then use \$r4	if rs = 7, then use \$r8
srl (shift right logical, write to	14	if opt = 0, srl 1 bit	if rs = 0, then use \$r1	if rs = 4, then use \$r5
rs)		if opt = 1, srl 8 bits	if rs = 1, then use \$r2	if rs = 5, then use \$r6
			if rs = 2, then use \$r3	if rs = 6, then use \$r7
			if rs = 3, then use \$r4	if rs = 7, then use \$r8

Operations with J-type format:

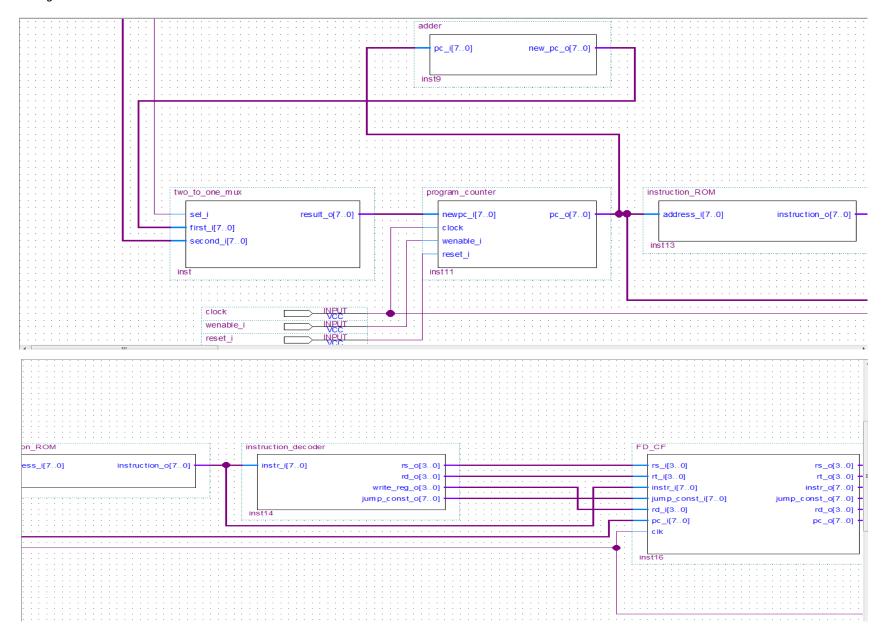
Instruction	opcode			const
jump (jump to instruction	12	if const = 0, use -19	if const = 5, use -8	if const = 10, use 6
address, newPC = PC + const)		if const = 1, use = -17	if const = 6, use -7	if const = 11, use 13
		if const = 2, use -14	if const = 7, use -6	if const = 12, use 14
		if const = 3, use -13	if const = 8, use 2	if const = 13, use 16
		if const = 4, use -9	if const = 9, use 5	if const = 14, use 25
				if const = 15, use 47

SCHEMATICS

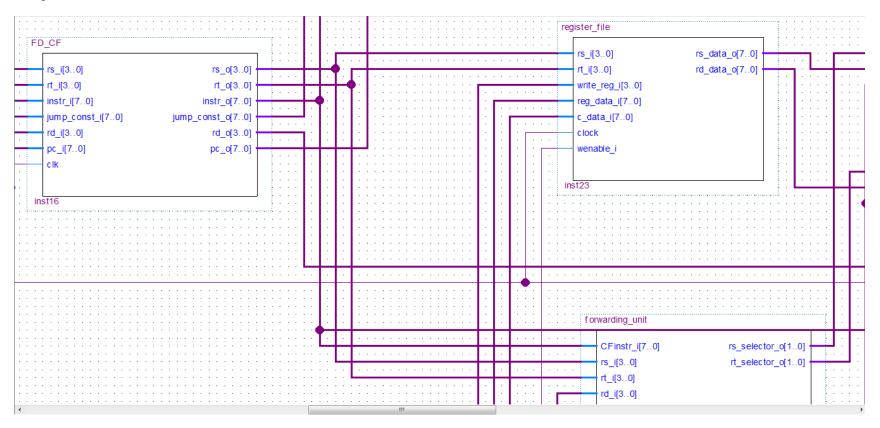
Overview

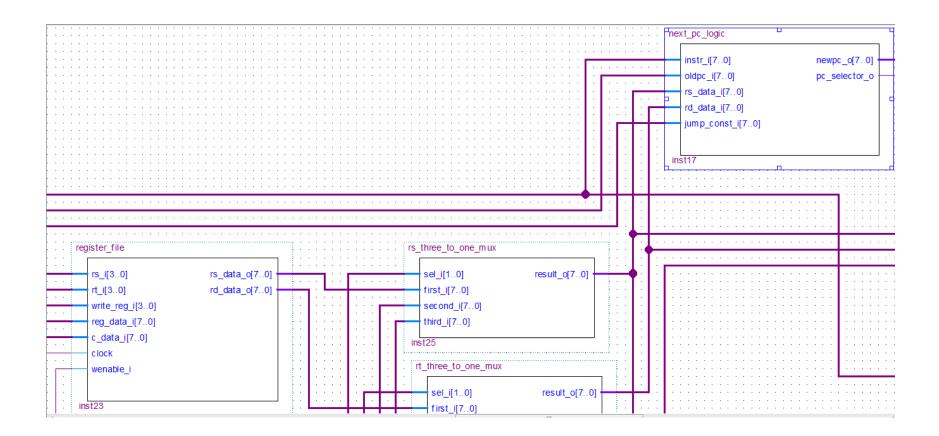


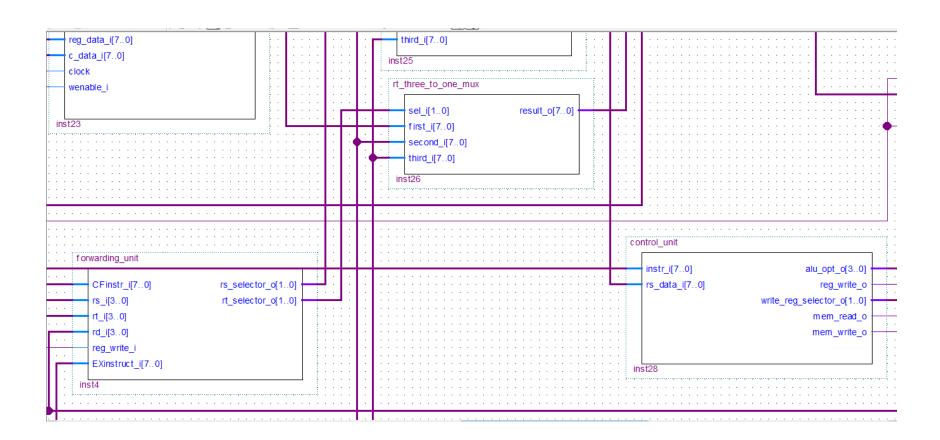
FD Stage



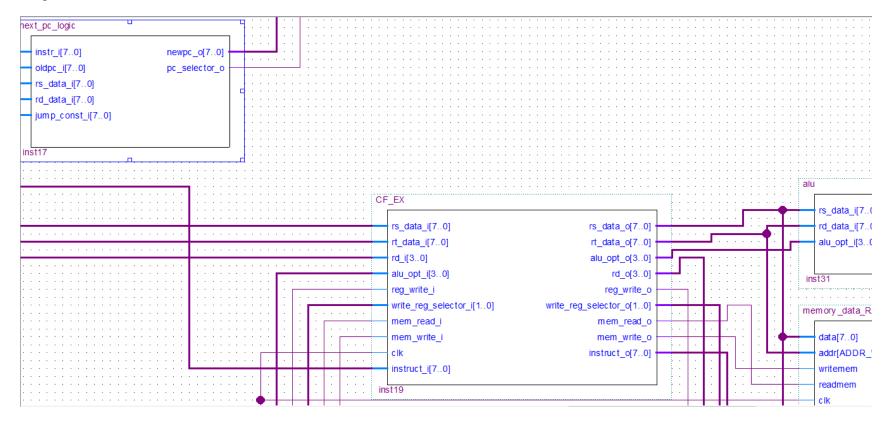
CF Stage

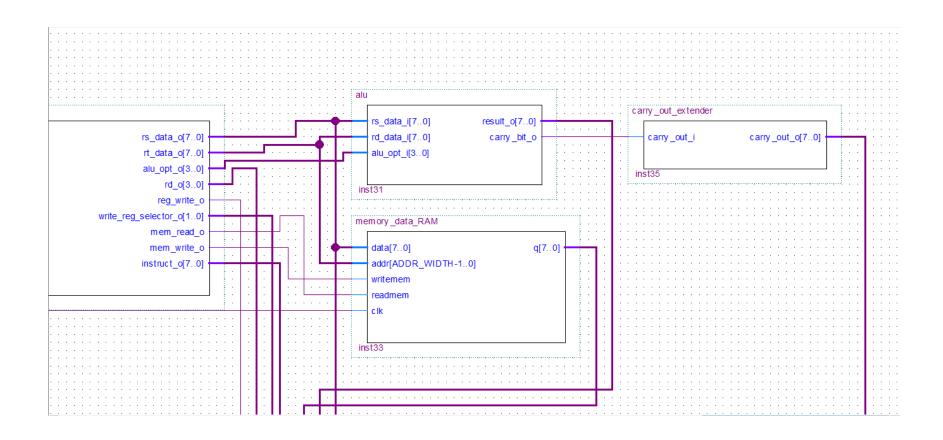


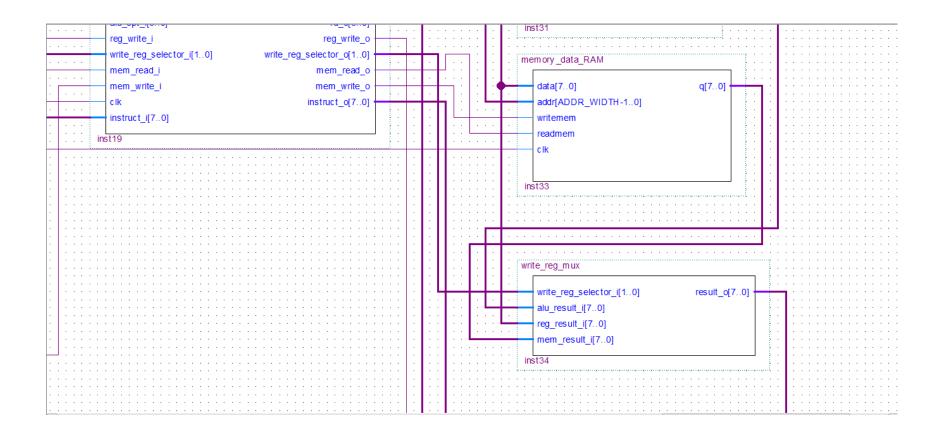




EX Stage







VERILOG CODE

module two_to_one_mux (input logic sel_i, input logic [7:0] first_i, input logic [7:0] second_i, output logic [7:0] result_o); always_comb begin case(sel_i) 0: result_o = first_i;	module alu (input [7:0] rs_data_i, input [7:0] rd_data_i, input [3:0] alu_opt_i, output logic [7:0] result_o, output logic carry_bit_o); always comb begin	result_o=(~rs_data_i+1)+rd_data_i; //rd_data_i is negative and rs_data_i is positive else if(rs_data_i[7]==1'b0 && rd_data_i[7]==1'b1) result_o=(~rd_data_i+1)+rs_data_i; //rs_data_i is negative	end //shift left 2 6:begin result_o=rs_data_i<<2; carry_bit_o=0; end //shift left 4 7:begin result_o=rs_data_i<<4; carry_bit_o=0; end	module carry_out_extender (input logic carry_out_i, output logic [7:0] carry_out_o); always_comb begin carry_out_o = {7'b0000000, carry_out_i}; end endmodule
1: result_o = second_i; endcase end endmodule	case(alu_opt_i) //add 0:begin result_o = rs_data_i+rd_data_i;	and rd_data_i is negative else begin	//shift left 6 8:begin result_o=rs_data_i<<6; carry_bit_o=0; end	module CF_EX (input logic [7:0] rs_data_i, input logic [7:0] rt_data_i, input logic [3:0] rd_i, input logic [3:0] alu_opt_i,
module write_reg_mux (input logic [1:0] write_reg_selector_i, input logic [7:0] alu_result_i, input logic [7:0] reg_result_i, input logic [7:0] mem_result_i, output logic [7:0] result_o); always_comb begin case(write_reg_selector_i) 0: result_o = alu_result_i; 1: result_o = reg_result_i;	if(255 <rs_data_i+rd_data_i) 1="" 1:begin="" carry_bit_o="rs_data_i[7];" difference<="" else="" end="" left="" result_o="rs_data_i<1;" shift="" td=""><td><pre>if((~rs_data_i+1)>(~rd_data_i+1))</pre></td><td>//shift right 8 9:begin result_o=rs_data_i>>8; carry_bit_o=0; end 10: begin if(rs_data_i > rd_data_i) begin result_o = rs_data_i - rd_data_i; carry_bit_o = 0; end else</td><td>input logic reg_write_i, input logic [1:0] write_reg_selector_i, input logic mem_read_i, input logic mem_write_i, input clk, input [7:0] instruct_i, output logic [7:0] rs_data_o, output logic [7:0] rt_data_o, output logic [3:0] alu_opt_o, output logic [3:0] rd_o, output logic reg_write_o, output logic [1:0]</td></rs_data_i+rd_data_i)>	<pre>if((~rs_data_i+1)>(~rd_data_i+1))</pre>	//shift right 8 9:begin result_o=rs_data_i>>8; carry_bit_o=0; end 10: begin if(rs_data_i > rd_data_i) begin result_o = rs_data_i - rd_data_i; carry_bit_o = 0; end else	input logic reg_write_i, input logic [1:0] write_reg_selector_i, input logic mem_read_i, input logic mem_write_i, input clk, input [7:0] instruct_i, output logic [7:0] rs_data_o, output logic [7:0] rt_data_o, output logic [3:0] alu_opt_o, output logic [3:0] rd_o, output logic reg_write_o, output logic [1:0]
2: result_o = mem_result_i; default: result_o = alu_result_i; endcase end endmodule module adder (input logic [7:0] pc_i, output logic [7:0] new_pc_o); always comb	2:begin //both number are positive if(rs_data_i[7]==1'b0 && rd_data_i[7]==1'b0) begin if(rs_data_i>rd_data_i) result_o = rs_data_i- rd_data_i; else result_o=rd_data_i-	3:begin if(255 <rs_data_i+1) 4:begin="" carry_bit_o="0;" decrement="" else="" end="" result_o="rs_data_i-1;</td"><td><pre>begin result_o = rd_data_i - rs_data_i; carry_bit_o = 0; end end default: begin result_o=0; carry_bit_o=0; end end end</pre></td><td>write_reg_selector_o, output logic mem_read_o, output logic mem_write_o, output logic [7:0] instruct_o); logic [7:0] rs_data; logic [7:0] rt_data; logic [3:0] rd; logic [3:0] alu_opt; logic reg_write;</td></rs_data_i+1)>	<pre>begin result_o = rd_data_i - rs_data_i; carry_bit_o = 0; end end default: begin result_o=0; carry_bit_o=0; end end end</pre>	write_reg_selector_o, output logic mem_read_o, output logic mem_write_o, output logic [7:0] instruct_o); logic [7:0] rs_data; logic [7:0] rt_data; logic [3:0] rd; logic [3:0] alu_opt; logic reg_write;
begin new_pc_o=pc_i+1; end endmodule	rs_data_i; end //rs_data_i is negative and rd_data_i is positive else if(rs_data_i[7]==1'b1 && rd_data_i[7]==1'b0)	<pre>carry_bit_o=0; end //shift right 1 5:begin carry_bit_o=rs_data_i[0]; result_o=rs_data_i>>1;</pre>	end endmodule	logic [1:0] write_reg_selector; logic mem_read; logic mem_write; logic [7:0]instruct; always_comb

begin	if(instr_i[3] == 0)	write_reg_selector_o = 0;	reg_write_o = 1;	write_reg_selector_o = 0;
rs_data_o=rs_data;	begin	mem_read_o = 0;	write_reg_selector_o = 1;	mem_read_o = 0;
rt_data_o=rt_data;	alu_opt_o = 3;	mem_write_o = 0;	mem_read_o = 0;	mem_write_o = 0;
alu_opt_o=alu_opt;	end	end	mem_write_o = 0;	end
rd_o=rd;	else	2: begin	end	endcase
reg_write_o=reg_write;	begin	reg_write_o = 1;	end	end
<u> </u>	alu_opt_o = 4;	write_reg_selector_o = 0;	9: begin	endmodule
	end	mem_read_o = 0;	reg_write_o = 1;	
write_reg_selector_o=write_reg_	end	mem_write_o = 0;	write_reg_selector_o = 0;	module FD_CF (
selector;	11: alu_opt_o = 5;	end	mem_read_o = 0;	input logic [3:0] rs_i,
mem_read_o=mem_read;	13: alu opt o = 10;	3: begin	mem write o = 0;	input logic [3:0] rt i,
mem_write_o=mem_write;	14: begin	reg_write_o = 1;	end	input logic [7:0] instr i,
instruct_o=instruct;	if(instr_i[3] == 0)	write_reg_selector_o = 1;	11: begin	input byte jump_const_i,
end	begin	mem read $o = 0$;	if(rs_data_i[0]==0)	input logic [3:0] rd i,
always # @ (accorded alls)	alu_opt_o = 5;	mem_write_o = 0;	begin	input logic [7:0] pc_i,
always_ff @ (negedge clk)	end	end	reg_write_o = 0;	input clk,
begin	else	4: begin	write_reg_selector_o = 0;	output logic [3:0] rs_o,
rs_data=rs_data_i;	begin	reg_write_o = 1;	mem_read_o = 0;	output logic [3:0] rt_o,
rt_data=rt_data_i;	alu_opt_o = 9;	write_reg_selector_o = 1;	mem_write_o = 0;	output logic [7:0] instr_o,
alu_opt=alu_opt_i;	end	mem_read_o = 0;	end	output byte jump_const_o,
rd=rd_i;	end	mem_write_o = 0;	else	output logic [3:0] rd_o,
reg_write=reg_write_i;	15: begin	end	begin	output logic [7:0] pc_o
	case(instr i[3:2])	7: begin	reg_write_o = 0;);
write_reg_selector=write_reg_sel	0: begin	if(instr_i[3] == 0)	write_reg_selector_o = 0;	// logic [3:0] rs;
ector_i;	alu opt o = 1;	begin	mem read $o = 0$;	logic [3:0] rt;
mem_read=mem_read_i;	end	reg_write_o = 0;	mem write o = 0;	logic [7:0] instr;
mem_write=mem_write_i;	1: begin	write_reg_selector_o = 0;	end	byte jump_const;
instruct=instruct_i;	alu_opt_o = 6;	mem read $o = 0$;	end	logic [3:0] rd;
end	end	mem_write_o = 1;	13: begin	logic [7:0] pc;
endmodule	2: begin	end	reg_write_o = 1;	-0 -1 -31-7
	alu_opt_o = 7;	else	write_reg_selector_o = 0;	al a consult
module control_unit (end	begin	mem_read_o = 0;	always_comb
input logic [7:0] instr_i,	3: begin	reg_write_o = 1;	mem_write_o = 0;	begin
input [7:0] rs_data_i,	alu_opt_o = 8;	write reg selector o = 1;	end,	rs_o=rs;
output logic [3:0] alu_opt_o,	end	mem_read_o = 0;	14: begin	rt_o=rt;
output logic reg_write_o,	endcase	mem_write_o = 0;	reg_write_o = 1;	instr_o=instr;
output logic [1:0]	end	end	write_reg_selector_o = 0;	jump_const_o=jump_const;
write_reg_selector_o,	default: alu_opt_o = 0;	end	mem_read_o = 0;	rd_o=rd;
output logic mem_read_o,	endcase	8: begin	mem_write_o = 0;	pc_o=pc;
output logic mem_write_o	end	if(instr_i[3] == 0)	end	end
);		begin	15: begin	
	// Determine reg_write_o,	reg_write_o = 1;	reg_write_o = 1;	always_ff @ (negedge clk)
// Determine alu_opt_o	write_reg_selector_o,	write_reg_selector_o = 2;	write_reg_selector_o = 0;	begin
always_comb begin	mem_read_o, mem_write_o	mem_read_o = 1;	mem_read_o = 0;	rs=rs_i;
case(instr_i[7:4])	always comb begin	mem_write_o = 0;	mem_write_o = 0;	rt=rt_i;
1: alu_opt_o = 0;	case(instr_i[7:4])	end	end	instr=instr_i;
2: alu_opt_o = 2;	1: begin	else	default: begin	jump_const=jump_const_i;
9: begin	reg_write_o = 1;	begin	reg_write_o = 0;	rd=rd_i;
	00,			

pc=pc_i;	else	begin	begin	end
end	rt_selector_o = 2;	rs_selector_o = 0;	rs_selector_o = 2;	end
endmodule	end	end	end	8: begin
	else if(rt_i == rd_i)	rt_selector_o = 0;	else if(rs_i == rd_i)	if(rs_i == rd_i)
module forwarding_unit (begin	end	begin	begin
input logic [7:0] CFinstr_i,	rt_selector_o = 1;	4: begin	rs_selector_o = 1;	rs_selector_o = 1;
input [3:0] rs_i,	end	if(rs_i==0 && rd_i==0)	end - ,	end ,
input [3:0] rt_i,	else	begin	else	else
input [3:0] rd_i,	begin	rs_selector_o = 2;	begin	begin
input reg_write_i,	rt_selector_o = 0;	end,	rs_selector_o = 0;	rs_selector_o = 0;
input logic [7:0] EXinstruct_i,	end	else if(rs_i == rd_i)	end	end
output logic [1:0] rs_selector_o,	end	begin	if(rt_i==0)	if(rt_i==0)
output logic [1:0] rt_selector_o	2: begin	rs_selector_o = 1;	begin	begin
);	if(rs_i==0)	end	rt_selector_o = 2;	rt_selector_o = 2;
// if selector == 0, don't forward	begin	else	end	end
// if 1, forward result	rs_selector_o = 2;	begin	else if(rt_i == rd_i)	else if(rt_i == rd_i)
// if 2, forward carry out	end	rs_selector_o = 0;	begin	begin
always_comb	else if(rs_i == rd_i)	end	rt_selector_o = 1;	rt_selector_o = 1;
begin	begin	rt_selector_o = 0;	end	end
if(reg_write_i)	rs_selector_o = 1;	end	else	else
begin	end	5: begin	begin	begin
case(CFinstr_i[7:4])	else	if(rs_i==0 && rd_i==0)	rt_selector_o = 0;	rt_selector_o = 0;
0: begin	begin	begin	end	end
_	rs_selector_o = 0;		end	end
rs_selector_o = 0;	end	rs_selector_o = 2; end	7: begin	9: begin
rt_selector_o = 0; end				_
	if(rt_i==0)	else if(rs_i == rd_i)	if(rs_i==0)	if(rs_i==0)
1: begin if(rs_i==0)	begin	begin	begin	begin
· -	rt_selector_o = 2; end	rs_selector_o = 1; end	rs_selector_o=2; end	rs_selector_o = 2;
begin				end
if(EXinstruct_i[7:4]==8 &&	else if(rt_i == rd_i)	else	else if(rs_i == rd_i)	else if(rs_i == rd_i)
EXinstruct_i[3]==1)	begin	begin	begin	begin
rs_selector_o=1;	rt_selector_o = 1;	rs_selector_o = 0;	rs_selector_o = 1;	rs_selector_o = 1;
else	end	end	end	end
rs_selector_o = 2;	else	if(rt_i==0)	else	else
end	begin	begin	begin	begin
else if(rs_i == rd_i)	rt_selector_o = 0;	rt_selector_o = 2;	rs_selector_o = 0;	rs_selector_o = 0;
begin	end	end	end	end
rs_selector_o = 1;	end	else if(rt_i == rd_i)	if(rt_i==0)	rt_selector_o = 0;
end	3: begin	begin	begin	end
else	if(rs_i==0 && rd_i==0)	rt_selector_o = 1;	rt_selector_o = 2;	10: begin
begin	begin	end	end	if(rs_i==0)
rs_selector_o = 0;	rs_selector_o = 2;	else	else if(rt_i == rd_i)	begin
end	end	begin	begin	rs_selector_o = 2;
if(rt_i==0)	else if(rs_i == rd_i)	rt_selector_o = 0;	rt_selector_o = 1;	end
begin	begin	end	end	else if(rs_i == rd_i)
if(EXinstruct_i[7:4]==8 &&	rs_selector_o = 1;	end	else	begin
EXinstruct_i[3]==1)	end	6: begin	begin	rs_selector_o = 1;
rt_selector_o=1;	else	if(rs_i==0 && rd_i==0)	rt_selector_o = 0;	end

else	rt_selector_o = 0;	rt_selector_o = 2;	end	parameter jump_const12 = 6
begin	end	end		parameter jump_const13 = 1
rs_selector_o = 0;	end	else if(rt_i == rd_i)		parameter jump_const14 = 1
end	14: begin	begin	endmodule	parameter jump_const15 = 1
rt_selector_o = 0;	if(rs_i==0)	rt_selector_o = 1;		
end	begin	end	module instruction_decoder	// Decodes instruction for
11: begin	rs_selector_o = 2;	else	(write_reg_o
if(rs_i==0)	end	begin	input [7:0] instr_i,	always_comb
begin	else if(rs_i == rd_i)	rt_selector_o = 0;	output logic [3:0] rs_o,	begin
rs_selector_o = 2;	begin	end	output logic [3:0] rd_o,	case(instr_i[7:4])
end	rs_selector_o = 1;	end	output logic [3:0]	4'b0000: begin
else if(rs_i == rd_i)	end	endcase	write_reg_o,	write_reg_o = c;
begin	else	end	output byte jump_const_o	end
rs selector o = 1;	begin	else);	4'b0001: begin
end	rs_selector_o = 0;	begin	•	case(instr i[1:0])
else	end,	rt selector o=0;	parameter c = 0;	2'b00: begin
begin	rt_selector_o = 0;	rs selector o=0;	parameter r1 = 1;	write_reg_o = r1;
rs_selector_o = 0;	end	end,	parameter r2 = 2;	end ,
end,	15: begin	end	parameter r3 = 3;	2'b01: begin
rt_selector_o = 0;	if(rs_i==0)		parameter r4 = 4;	write_reg_o = r3;
end	begin		parameter r5 = 5;	end
12: begin	rs_selector_o = 2;	endmodule	parameter r6 = 6;	2'b10: begin
rs_selector_o = 0;	end	eaou a	parameter r7 = 7;	write_reg_o = r4;
rt selector o = 0;	else if(rs_i == rd_i)	module	parameter r8 = 8;	end
end	begin	hazard detection unit (parameter temp = 9;	2'b11: begin
13: begin	rs_selector_o = 1;	input logic taken i,	parameter temp 3,	write_reg_o = r5;
if(rs i==0)	end	input logic hazard i,	parameter jump const0 =	end
begin	else	input logic [7:0] instr i,	-19;	default: begin
rs selector o = 2;	begin	output logic hazard o	parameter jump const1 =	write_reg_o = r1;
end	rs_selector_o = 0;);	-17;	end
else if(rs i == rd i)	end	"	parameter jump const2 =	endcase
begin	rt_selector_o = 0;	always_comb	-16;	end
rs selector o = 1;	end	begin	parameter jump const3 =	4'b0010: begin
end	default: begin	if(taken i==0)	-14;	case(instr_i[3:2])
	if(rs i==0 && rd i==0)	. = /		
else	, = = ,	begin	parameter jump_const4 =	2'b00: begin
begin	begin	hazard_o = 0;	-9;	write_reg_o = r1;
rs_selector_o = 0;	rs_selector_o = 2;	end	parameter jump_const5 =	end
end	end	else	-8;	2'b01: begin
if(rt_i==0)	else if(rs_i == rd_i)	begin	parameter jump_const6 =	write_reg_o = r2;
begin	begin	if(hazard_i==0)	-6;	end
rt_selector_o = 2;	rs_selector_o = 1;	begin	parameter jump_const7 = 2;	2'b10: begin
end	end	hazard_o = 1;	parameter jump_const8 = 6;	write_reg_o = r6;
else if(rt_i == rd_i)	else	end	parameter jump_const9 =	end
begin	begin	else	16;	2'b11: begin
rt_selector_o = 1;	rs_selector_o = 0;	begin	parameter jump_const10 =	write_reg_o = r8;
end	end	hazard_o = 0;	17;	end
else	if(rt_i==0)	end	parameter jump_const11 =	default: begin
begin	begin	end	33;	write reg $o = r1$;

end	write_reg_o = temp;	3'b010: begin	end	3'b110: begin
endcase	end	write_reg_o = r3;	3'b111: begin	write_reg_o = r7;
end	4'b1000: begin	end	write_reg_o = r8;	end
4'b0011: begin	case(instr_i[3])	3'b011: begin	end	3'b111: begin
case(instr_i[1:0])	1'b0: begin	write_reg_o = r4;	default: begin	write_reg_o = r8;
2'b00: begin	case(instr_i[2:0])	end	write_reg_o = r1;	end
write reg o = r2;	3'b000: begin	3'b100: begin	end	default: begin
end	write_reg_o = r1;	write_reg_o = r5;	endcase	write_reg_o = r1;
2'b01: begin	end	end	end	end
write_reg_o = r4;	3'b001: begin	3'b101: begin	4'b1100: begin	endcase
end	write_reg_o = r2;	write_reg_o = r6;	write_reg_o = c;	end
2'b10: begin	end	end	end	4'b1111: begin
write_reg_o = r7;	3'b010: begin	3'b110: begin	4'b1101: begin	case(instr i[1:0])
end	write_reg_o = r3;	write_reg_o = r7;	case(instr_i[3:2])	2'b00: begin
2'b11: begin	end	end	2'b00: begin	write reg o = r2;
write_reg_o = r8;	3'b011: begin	3'b111: begin	write reg o = r1;	end
end	write reg o = r4;	write reg $o = r8$;	end ,	2'b01: begin
default: begin	end	end	2'b01: begin	write_reg_o = r3;
write_reg_o = r2;	3'b100: begin	default: begin	write_reg_o = r2;	end
end	write reg o = r5;	write_reg_o = r1;	end	2'b10: begin
endcase	end	end	2'b10: begin	write_reg_o = r4;
end	3'b101: begin	endcase	write_reg_o = r6;	end
4'b0100: begin	write reg o = r6;	end	end	2'b11: begin
case(instr_i[1:0])	end	4'b1010: begin	2'b11: begin	write_reg_o = r8;
2'b00: begin	3'b110: begin	write reg o = c;	write reg o = r8;	end
write reg $o = r1$;	write reg o = r7;	end end	end	default: begin
end	end	4'b1011: begin	endcase	write reg o = r2;
2'b01: begin	3'b111: begin	case(instr i[2:0])	end	end
write_reg_o = r3;	write_reg_o = r8;	3'b000: begin	4'b1110: begin	endcase
end	end	write_reg_o = r1;	case(instr_i[2:0])	end
2'b10: begin	default: begin	end	3'b000: begin	default: begin
write_reg_o = r7;	write_reg_o = r1;	3'b001: begin	write_reg_o = r1;	write_reg_o = c;
end	end	write reg $o = r2$;	end	end
2'b11: begin	endcase	end	3'b001: begin	endcase
write_reg_o = r8;	end	3'b010: begin	write_reg_o = r2;	end
end	1'b1: begin	write reg $o = r3$;	end	5.10
default: begin	write_reg_o = c;	end	3'b010: begin	//Decodes instruction for rs o
write_reg_o = r1;	end	3'b011: begin	write_reg_o = r3;	always_comb
end	endcase	write_reg_o = r4;	end	begin
endcase	end	end	3'b011: begin	case(instr i[7:4])
end	4'b1001: begin	3'b100: begin	write reg o = r4;	4'b0000: begin
4'b0101: begin	case(instr_i[2:0])	write_reg_o = r5;	end	rs_o = r1;
write_reg_o = c;	3'b000: begin	end	3'b100: begin	end
end	write reg o = r1;	3'b101: begin	write reg o = r5;	4'b0001: begin
4'b0110: begin	end	write_reg_o = r6;	end	case(instr i[3:2])
write_reg_o = c;	3'b001: begin	end	3'b101: begin	2'b00: begin
end	write reg o = r2;	3'b110: begin	write reg $o = r6$;	rs_o = c;
4'b0111: begin	end	write_reg_o = r7;	end	end
·				

31h04 - h i -	d	4 lh 4 . h a min		
2'b01: begin	end	1'b1: begin	end	rs_o = r1; end
rs_o = r1; end	4'b0100: begin	case(instr_i[2])	endcase	
	case(instr_i[3:2])	1'b0: begin	end	endcase end
2'b10: begin	2'b00: begin	rs_o = r3; end	1'b1: begin	
rs_o = r3;	rs_o = r5;	*****	rs_o = c;	4'b1001: begin
end	end	1'b1: begin	end	case(instr_i[2:0])
2'b11: begin	2'b01: begin	rs_o = r6;	default: begin	3'b000: begin
rs_o = r4;	rs_o = r6;	end	rs_o = r1;	rs_o = r1;
end	end	default: begin	end	end
default: begin	2'b10: begin	rs_o = r3;	endcase	3'b001: begin
rs_o = r1;	rs_o = r7;	end	end	rs_o = r2;
end	end	endcase	4'b1000: begin	end
endcase	2'b11: begin	end	case(instr_i[3])	3'b010: begin
end	rs_o = r8;	default: begin	1'b0: begin	rs_o = r3;
4'b0010: begin	end	rs_o = r1;	case(instr_i[2:0])	end
case(instr_i[3:2])	default: begin	end	3'b000: begin	3'b011: begin
2'b00: begin	rs_o = r5;	endcase	rs_o = r1;	rs_o = r4;
rs_o = r1;	end	end	end	end
end	endcase	4'b0111: begin	3'b001: begin	3'b100: begin
2'b01: begin	end	case(instr_i[3])	rs_o = r2;	rs_o = r5;
rs_o = r2;	4'b0101: begin	1'b0: begin	end	end
end	case(instr_i[2])	case(instr_i[2:0])	3'b010: begin	3'b101: begin
2'b10: begin	1'b0: begin	3'b000: begin	rs_o = r3;	rs_o = r6;
rs_o = r6;	rs_o = r1;	rs_o = r1;	end	end
end	end	end	3'b011: begin	3'b110: begin
2'b11: begin	1'b1: begin	3'b001: begin	rs_o = r4;	rs_o = r7;
rs_o = r8;	rs_o = r2;	rs_o = r2;	end	end
end	end	end	3'b100: begin	3'b111: begin
default: begin	default: begin	3'b010: begin	rs_o = r5;	rs_o = r8;
rs_o = r1;	rs_o = r1;	rs_o = r3;	end	end
end	end	end	3'b101: begin	default: begin
endcase	endcase	3'b011: begin	rs_o = r6;	rs_o = r1;
end	end	rs_o = r4;	end ,	end ,
4'b0011: begin	4'b0110: begin	end	3'b110: begin	endcase
case(instr_i[3:2])	case(instr_i[3])	3'b100: begin	rs_o = r7;	end
2'b00: begin	1'b0: begin	rs_o = r5;	end	4'b1010: begin
rs o = r1;	case(instr i[2])	end	3'b111: begin	case(instr i[2:0])
end	1'b0: begin	3'b101: begin	rs o = r8;	3'b000: begin
2'b01: begin	rs o = r1;	rs o = r6;	end	rs o = r1;
rs o = r2;	end	end	default: begin	end
end	1'b1: begin	3'b110: begin	rs_o = r1;	3'b001: begin
2'b10: begin	rs o = r2;	rs_o = r7;	end	rs o = r2;
rs_o = r3;	end	13_0 = 17, end	endcase	13_0 = 12, end
15_0 = 15, end	default: begin	3'b111: begin	end	3'b010: begin
2'b11: begin	rs o = r1;	rs o = r8;	1'b1: begin	rs o = r3;
•	rs_0 = r1; end	= *	5	rs_0 = rs; end
rs_o = r4;		end	rs_o = temp;	
end	endcase	default: begin	end	3'b011: begin
endcase	end	rs_o = r1;	default: begin	rs_o = r4;

end	end	end	rd_o = r5;	rd_o = r1;
3'b100: begin	4'b1100: begin	default: begin	end	end
rs_o = r5;	rs_o = r1;	rs_o = r1;	default: begin	2'b01: begin
end	end	end	rd_o = r1;	rd_o = r3;
3'b101: begin	4'b1101: begin	endcase	end	end
rs_o = r6;	case(instr_i[3:2])	end	endcase	2'b10: begin
end	2'b00: begin	4'b1111: begin	end	rd_o = r7;
3'b110: begin	rs_o = r1;	case(instr_i[1:0])	4'b0010: begin	end
rs_o = r7;	end	2'b00: begin	case(instr_i[1:0])	2'b11: begin
end	2'b01: begin	rs_o = r2;	2'b00: begin	rd_o = r8;
3'b111: begin	rs_o = r2;	end	rd_o = r3;	end
rs_o = r8;	end	2'b01: begin	end	default: begin
end	2'b10: begin	rs_o = r3;	2'b01: begin	rd_o = r1;
default: begin	rs_o = r6;	end	rd_o = r4;	end
rs_o = r1;	end	2'b10: begin	end	endcase
end	2'b11: begin	rs_o = r4;	2'b10: begin	end
endcase	rs o = r8;	end	rd o = r5;	4'b0101: begin
end	end	2'b11: begin	end	case(instr i[1:0])
4'b1011: begin	default: begin	rs_o = r8;	2'b11: begin	2'b00: begin
case(instr i[2:0])	rs o = r8;	end	rd o = r6;	rd o = r1;
3'b000: begin	end	default: begin	end	end
rs o = r1;	endcase	rs_o = r2;	default: begin	2'b01: begin
end	end	end	rd o = r3;	rd o = r2;
3'b001: begin	4'b1110: begin	endcase	end	end
rs_o = r2;	case(instr_i[2:0])	end	endcase	2'b10: begin
end	3'b000: begin	endcase	end	rd o = r3;
	<u> </u>	end	4'b0011: begin	ru_0 = 15, end
3'b010: begin	rs_o = r1; end	ena	ū	
rs_o = r3;	****	//Danadas instruction for	case(instr_i[1:0])	2'b11: begin
end	3'b001: begin	//Decodes instruction for	2'b00: begin	rd_o = r4;
3'b011: begin	rs_o = r2;	rd_o	rd_o = r2;	end
rs_o = r4;	end	always_comb	end	default: begin
end	3'b010: begin	begin	2'b01: begin	rd_o = r1;
3'b100: begin	rs_o = r3;	case(instr_i[7:4])	rd_o = r4;	end
rs_o = r5;	end	4'b0000: begin	end	endcase
end	3'b011: begin	rd_o = c;	2'b10: begin	end
3'b101: begin	rs_o = r4;	end	rd_o = r7;	4'b0110: begin
rs_o = r6;	end	4'b0001: begin	end	case(instr_i[3])
end	3'b100: begin	case(instr_i[1:0])	2'b11: begin	1'b0: begin
3'b110: begin	rs_o = r5;	2'b00: begin	rd_o = r8;	case(instr_i[1:0])
rs_o = r7;	end	rd_o = r1;	end	2'b00: begin
end	3'b101: begin	end	default: begin	rd_o = r3;
3'b111: begin	rs_o = r6;	2'b01: begin	rd_o = r2;	end
rs_o = r8;	end	rd_o = r3;	end	2'b01: begin
end	3'b110: begin	end	endcase	rd_o = r4;
default: begin	rs_o = r7;	2'b10: begin	end	end
rs_o = r1;	end	rd o = r4;	4'b0100: begin	2'b10: begin
		_ ′	G	o o
end	3'b111: begin	end	case(instr i[1:0])	rd o = r5;

2'b11: begin	rd_o = c;	jump_const_o =	jump_const_o =	12 : instruction_o =
rd_o = r7;	end	jump_const2;	jump_const14;	8'b01101101;
end	4'b1101: begin	end	end	13 : instruction_o =
default: begin	case(instr_i[1:0])	4'b0011: begin	4'b1111: begin	8'b11001010;
rd_o = r3;	2'b00: begin	jump_const_o =	jump_const_o =	14 : instruction_o =
end	rd_o = r3;	jump_const3;	jump_const15;	8'b10110001;
endcase	end	end	end	15 : instruction_o =
end	2'b01: begin	4'b0100: begin	default: begin	8'b11001000;
1'b1: begin	rd_o = r4;	jump_const_o =	jump_const_o =	16 : instruction_o =
case(instr_i[1:0])	end	jump_const4;	jump_const8;	8'b10011001;
2'b00: begin	2'b10: begin	end	end	17 : instruction o =
rd_o = r1;	rd_o = r5;	4'b0101: begin	endcase	8'b00010110;
end	end	jump_const_o =	end	18 : instruction o =
2'b01: begin	2'b11: begin	jump_const5;	endmodule	8'b00010011;
rd_o = r2;	rd_o = r6;	end		19 : instruction o =
end ,	end ,	4'b0110: begin	module instruction ROM (8'b00011011;
2'b10: begin	default: begin	jump_const_o =	input logic [7:0] address i,	20 : instruction_o =
rd o = r3;	rd o = r4;	jump_const6;	output logic [7:0]	8'b11000101;
end	end	end	instruction_o	,
2'b11: begin	endcase	4'b0111: begin);	
rd o = r7;	end	jump const o =	,,	21 : instruction_o =
end	4'b1110: begin	jump_const7;	always_comb	8'b11100001;
default: begin	rd o = c;	end	begin	22 : instruction_o =
rd_o = r1;	end	4'b1000: begin	case (address_i)	8'b00110011;
end	4'b1111: begin	jump_const_o =	0 : instruction o =	23 : instruction_o =
endcase	rd_o = c;	jump_const8;	8'b11101101;	8'b11110011;
end	end	end	1 : instruction_o =	
default: begin	default: begin	4'b1001: begin	8'b11101010;	24 : instruction_o =
rd_o = r1;	rd_o = c;	jump_const_o =	2 : instruction_o =	8'b01111000;
end	end	jump_const9;	8'b11101011;	25 : instruction_o =
endcase	endcase	end	3 : instruction o =	8'b01001100;
end	end	4'b1010: begin	8'b11101111;	26 : instruction_o =
4'b0111: begin	Cita	jump const o =	4 : instruction o =	8'b11110001;
rd_o = r8;	//Decodes instruction for	jump_const10;	8'b11101100;	27 : instruction_o =
end	jump const o	end	5 : instruction o =	8'b10001000;
4'b1000: begin	always_comb	4'b1011: begin	8'b10010111;	28 : instruction o =
rd o = r8;	begin	jump_const_o =	6: instruction o =	8'b00010001;
end	case(instr i[3:0])	jump_const11;	8'b10000000;	29 : instruction o =
4'b1001: begin	4'b0000: begin	end	7 : instruction o =	8'b11000001;
rd_o = c;	jump_const_o =	4'b1100: begin	8'b10010111;	30 : instruction o =
end	jump_const0;	jump_const_o =	8 : instruction_o =	8'b00011100;
4'b1010: begin	end	jump_const12;	8'b10000001;	31 : instruction_o =
rd_o = c;	4'b0001: begin	end	9 : instruction_o =	8'b00010011;
end	jump_const_o =	4'b1101: begin	8'b10010101;	32 : instruction_o =
4'b1011: begin	jump_const1;	jump const o =	10 : instruction o =	8'b00011011;
rd_o = c;	end	jump_const13;	8'b10100001;	33 : instruction o =
end	4'b0010: begin	end	11 : instruction o =	8'b01000001;
4'b1100: begin	+ 50010. 5CBIII	4'b1110: begin	8'b11001100;	•
. DIIO. DCB		- DIIIO. DESIII	0.011001100,	

34 : instruction_o =	57 : instruction_o =	79 : instruction_o =	101 : instruction_o =	125 : instruction_o =
8'b11101111;	8'b11110001;	8'b01110101;	8'b00111011;	8'b00000000;
35 : instruction_o =	58 : instruction_o =	80 : instruction_o =	102 : instruction_o =	
8'b10010111;	8'b10001000;	8'b01110101;	8'b11100111;	//Program 3
36 : instruction_o =	59 : instruction_o =	81 : instruction_o =	103 : instruction_o =	126 : instruction o =
8'b10010111;	8'b00010001;	8'b01110101;	8'b11101000;	8'b11101111;//srl
37 : instruction_o =	60 : instruction_o =	82 : instruction_o =	104 : instruction_o =	127 : instruction o =
8'b10010111;	8'b11000001;	8'b00000000;	8'b10100010;	8'b10010111;//addi
	61 : instruction_o =		105 : instruction_o =	128 : instruction o =
38 : instruction o =	8'b00010110;	//program 2	8'b11001001;	8'b11110111;//sll
8'b10000001;	62 : instruction_o =	83 : instruction o =	106 : instruction_o =	129 : instruction_o =
39 : instruction o =	8'b00010011;	8'b11101101;	8'b10000011;	8'b11110011;//sll
8'b10100001;	63 : instruction_o =	84 : instruction o =	107 : instruction_o =	130 : instruction o =
40 : instruction o =	8'b00011011;	8'b11101111;	8'b01010111;	8'b10011111;//addi
8'b11001011;		85 : instruction o =	108 : instruction_o =	131 : instruction o =
41 : instruction o =	64 : instruction o =	8'b10010111;	8'b11001000;	8'b01001101;//movh
8'b11101011;	8'b11101111;	86 : instruction o =	109 : instruction_o =	132 : instruction o =
-	65 : instruction o =	8'b11110111;	8'b11100011;	8'b10010010;//addi
42 : instruction_o =	—		110 : instruction o =	
8'b11101100;	8'b10010111;	87 : instruction_o =	8'b10010000;	133 : instruction_o =
43 : instruction_o =	66 : instruction_o =	8'b10010111;	111 : instruction_o =	8'b10010010;//addi
8'b01101101;	8'b10010111;	88 : instruction_o =	8'b01100011;	134 : instruction_o =
44 : instruction_o =	67 : instruction_o =	8'b10010111;	112 : instruction o =	8'b11110001;//sll
8'b11001010;	8'b10010111;	89 : instruction_o =	8'b11001000;	135 : instruction_o =
45 : instruction_o =	68 : instruction_o =	8'b10000001;	113 : instruction o =	8'b10010010;//addi
8'b10110001;	8'b10010111;	90 : instruction_o =	8'b11000110;	136 : instruction_o =
46 : instruction_o =		8'b11111000;	114 : instruction_o =	8'b00111001;//movl
8'b11001000;	69 : instruction_o =	91 : instruction_o =	8'b10010100;	137 : instruction_o =
47 : instruction_o =	8'b01110100;	8'b11100001;	115 : instruction_o =	8'b11101000;
8'b10011001;	70 : instruction o =	92 : instruction_o =	8'b10011010;	138 : instruction_o =
48 : instruction_o =	8'b10010111;	8'b11100001;	116 : instruction o =	8'b11101001;
8'b00010110;	71 : instruction o =	93 : instruction_o =	8'b10010111;	139 : instruction_o =
49 : instruction_o =	8'b01110011;	8'b11100001;	117 : instruction o =	8'b11101111;
8'b00010011;	72 : instruction o =	94 : instruction_o =	8'b11000011;	140 : instruction_o =
50 : instruction_o =	8'b0000000;	8'b11100001;		8'b10010111;//r8++
8'b00011011;	73 : instruction o =	95 : instruction_o =	118 : instruction_o =	141 : instruction_o =
51 : instruction_o =	8'b11101111;	8'b11101100;	8'b10010111;	8'b10010111;//r8++
8'b11000101;	74 : instruction o =	96 : instruction_o =	119 : instruction_o =	142 : instruction_o =
52 : instruction_o =	8'b10010111;	8'b11101010;	8'b10011010;	8'b1111111;
8'b11100001;	75 : instruction o =	97 : instruction_o =	120 : instruction_o =	143 : instruction_o =
53 : instruction_o =	8'b10010111;	8'b10010010;	8'b11000001;	8'b10000100;
8'b00110011;	76 : instruction o =	98 : instruction_o =	121 : instruction_o =	144 : instruction_o =
54 : instruction_o =	8'b10010111;	8'b01001110;	8'b01001011;	8'b10010111;//r8++
8'b11110011;	77 : instruction o =	99 : instruction_o =	122 : instruction_o =	145 : instruction_o =
55 : instruction_o =	8'b10010111;	8'b10011110;	8'b10010111;	8'b10000101;// ld
8'b01111000;	•	100 : instruction o =	123 : instruction_o =	146 : instruction o =
56 : instruction_o =	78 : instruction_o = 8'b11101101:	8'b11111101;	8'b10010111;	8'b10010001;//j++
8'b01001100;	8 011101101;	•	124 : instruction_o =	
·			8'b01110100:	

8'b01110100;

147 : instruction_o =	171 : instruction_o =	ram[1] = 8'd19;	ram[74] = 8'hfe;	begin
8'b00101010;//diff	8'b10010111;	ram[2] = 8'd61;	ram[75] = 8'hed;	// Write
148 : instruction_o =	172 : instruction_o =	ram[3] = 8'd23;	ram[76] = 8'h03;	if (writemem)
8'b01000110;//movh	8'b11111111;		ram[77] = 8'h69;	ram[addr] <= data;
149 : instruction_o =	173 : instruction_o =	ram[6] = 8'h05;	ram[78] = 8'hcf;	end
8'b10010000;//r1++	8'b10011111;		ram[79] = 8'h25;	
150 : instruction o =	174: instruction o =	ram[32] = 8'h12;	ram[80] = 8'h8a;	
8'b10010111;//r8++	8'b01110110;	ram[33] = 8'h34;	ram[81] = 8'hd1;	// Continuous assignment
151 : instruction o =	175 : instruction o =	ram[34] = 8'h56;	ram[82] = 8'h47;	// Continuous assignment
8'b10000101;//ld	8'b11010000;	ram[35] = 8'h78;	ram[83] = 8'had;	implies read returns NEW data.
152 : instruction o =	default: instruction o =	ram[36] = 8'h9a;	ram[84] = 8'h15;	// This is the natural behavior of
8'b00101010;//diff	8'b0000000;	ram[37] = 8'hbc;	ram[85] = 8'h9d;	the TriMatrix memory
153 : instruction o =	endcase	ram[38] = 8'hde;	ram[86] = 8'h01;	// blocks in Single Port mode.
8'b01101111;//bge	end	ram[39] = 8'hf0;	ram[87] = 8'h23;	assign q = (readmem)?
154 : instruction o =	endmodule	ram[40] = 8'h11;	ram[88] = 8'h34;	ram[addr] : 8'bxxxxxxxx;
8'b11000111;//jump	enamodale	ram[41] = 8'h22;	ram[89] = 8'h56;	
155 : instruction o =	// Quartus II Verilog	ram[42] = 8'h33;	ram[90] = 8'h78;	endmodule
=				
8'b01000110;//movh	Template	ram[43] = 8'h44;	ram[91] = 8'h9a;	module next_pc_logic
156 : instruction_o =	// Single port RAM with	ram[44] = 8'h55;	ram[92] = 8'hbc;	(
8'b10010001;//addi	single read/write address	ram[45] = 8'h66;	ram[93] = 8'hde;	input [7:0] instr_i,
157 : instruction_o =	and initial contents	ram[46] = 8'h77;	ram[94] = 8'hf0;	input [7:0] oldpc_i,
8'b01011110;//be	// specified with an initial	ram[47] = 8'h88;	ram[95] = 8'h00;	input [7:0] rs_data_i,
158 : instruction_o =	block	ram[48] = 8'h99;		input [7:0] rd_data_i,
8'b11000111;//jump		ram[49] = 8'haa;	ram[128] = 8'd2;	input byte jump const i,
159 : instruction_o =	module memory_data_RAM	ram[50] = 8'hde;	ram[129] = -8'd23;	output logic [7:0] newpc o,
8'b11000100;//jump	#(parameter	ram[51] = 8'had;	ram[130] = -8'd33;	output logic pc_selector_o
160 : instruction_o =	ADDR_WIDTH=8)	ram[52] = 8'hbe;	ram[131] = 8'd63;);
8'b10011011;//addi	(ram[53] = 8'hef;	ram[132] = 8'd18;	"
161 : instruction_o =	input [7:0] data,	ram[54] = 8'h02;	ram[133] = 8'd23;	always_comb
8'b11011101;//udiff	input [(ADDR_WIDTH-1):0]	ram[55] = 8'h46;	ram[133] = -8'd52;	begin
162 : instruction_o =	addr,	ram[56] = 8'h8a;	ram[135] = 8'd28;	case(instr i[7:4])
8'b10000100;//ld	input writemem,	ram[57] = 8'hce;	ram[136] = 8'd9;	4'b0000: begin
163 : instruction_o =	readmem, clk,	ram[58] = 8'h13;	·	9
8'b11101001;//srl	output [7:0] q	ram[59] = 8'h57;	ram[137] = -8'd8;	newpc_o = oldpc_i;
164 : instruction o =);	ram[60] = 8'h9a;	ram[138] = 8'd6;	pc_selector_o=1;
8'b11101000;//srl	,,	ram[61] = 8'hcf;	ram[139] = 8'd105;	end
165 : instruction o =	// Declare the RAM	ram[62] = 8'h39;	ram[140] = 8'd94;	4'b0001: begin
8'b10011010;//addi	variable	ram[63] = 8'haa;	ram[141] = -8'd83;	newpc_o = oldpc_i + 1;
166 : instruction o =	reg [7:0]	ram[64] = 8'hbc;	ram[142] = 8'd55;	pc_selector_o=0;
8'b01100000;	ram[2**ADDR WIDTH-1:0];	ram[65] = 8'hf1;	ram[143] = 8'd100;	end
167 : instruction_o =	ramiz Abbit_Wibiti-1.0],	ram[66] = 8'h00;	ram[144] = -8'd3;	4'b0010: begin
8'b11000111;	// Variable to hold the	ram[67] = 8'hf0;	ram[145] = -8'd72;	newpc_o = oldpc_i + 1;
•			ram[146] = 8'd65;	pc_selector_o=0;
168 : instruction_o =	registered read address	ram[68] = 8'h57;	ram[147] = -8'd47;	end
8'b11000000;	reg [ADDR_WIDTH-1:0]	ram[69] = 8'h68;		4'b0011: begin
169 : instruction_o =	addr_reg;	ram[70] = 8'h9a;	end	newpc_o = oldpc_i + 1;
8'b11101111;		ram[71] = 8'h56;	CIIU	pc_selector_o=0;
170 : instruction_o =	initial	ram[72] = 8'hbe;	always @ (posedge clk)	end
8'b10010111;	begin	ram[73] = 8'hde;	aiways @ (poseuge cik)	

newpc_0 = oldpc_1 + 1;	4'b0100: begin	end	pc_selector_o=0;	input wenable_i,	end
Part	<pre>newpc_o = oldpc_i + 1;</pre>	4'b1000: begin	end	output logic [7:0]	
	pc_selector_o=0;	newpc_o = oldpc_i + 1;	default: begin	rs_data_o,	
	end	pc_selector_o=0;	newpc_o = oldpc_i + 1;	output logic [7:0]	// Read data from register rd
Degin Newpc_0 = oldpc_i+1; endcase reg[7:0] c, ri, r2, r3, r4, r5, 0 c) begin rd_data_0 = c; end r6, r7, r8, temp; rd_data_0 = c; end r6, r7, r8, temp; rd_data_0 = c; end rewpc_0 = oldpc_i+1; end rewpc_0 = oldpc_i+2; reg[7:0] pc, pcnext; reg[7:0]	4'b0101: begin	end	pc_selector_o=0;	rd_data_o	always_comb
iff(s, data_ 3.0] = modulation pc_selector_o=0. end mendedule mendedule reg[7:0] c_1, 1, 2, 7, 4, 7, 4, 7, 7, 7, 4, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,	if(instr_i[3]==0)	4'b1001: begin	end);	begin
rd_data_ij3(30) end endmodule rf, 77, 78, temp; rd_data_o = c; begin 4*b1010: begin module program_counter //Read data from register rs 1: begin newpc_o = oldpc_i + 1; begin (module program_counter //Read data from register rs 1: begin ed newpc_o = oldpc_i + 2; input (rsc) begin end else pc_selector_o=0; input (rsc) case(rs.); 2: begin newpc_o = oldpc_i + 2; end input reset_i, rs_data_o = c; end end pc_selector_o=1; begin output (7:0) pc_o end 3: begin end pc_selector_o=1; reg (7:0) pc_pcnext; end 4: begin else end rcg (7:0) pc_pcnext; end 4: begin begin end end 4: begin rd, data_o = r3; else end pc_selector_o=1; end 5: begin rd_data_o = r3; else end pc_selector_o=0; end 5: begin rd_data_o = r3; end pc_selector_o=0	begin	newpc_o = oldpc_i + 1;	endcase		case(rt_i)
Degin Ab1010: begin newpc_o = oldpc_i + 1; lf(r_s, data_i = 0) nodule program_counter //Read data from register rs 1: begin rd, data_o = r1; end newpc_o = oldpc_i + 1; input (7:0) newpc_i begin end newpc_o = oldpc_i + 2; disc nput reset_i, 0: begin rd, data_o = r2; end newpc_o = oldpc_i + 2; disc nput reset_i, 0: begin rd, data_o = r3; end rs_data_o = c; end 3: begin rd, data_o = r3; end rd, data_o = r4; end rd, data_o = r6;	if(rs_data_i[3:0] ==	pc_selector_o=0;	end	reg[7:0] c, r1, r2, r3, r4, r5,	0: begin
newpc_0 = oldpc_i + 1;	rd_data_i[3:0])	end	endmodule	r6, r7, r8, temp;	rd_data_o = c;
pc, selector 0 = 0; begin (always_comb rd_data_0 = 1; end newpc_0 = oldpc_i + 1; input [7:0] newpc_i, begin end else pc_selector_0 = 0; input clock, case(s, i) 2; begin end newpc_0 = oldpc_i + 2; else input wenable_i, 0; begin rd_data_0 = r2; end newpc_0 = oldpc_i + 2; begin output [7:0] pc_0 end 3; begin end newpc_0 = oldpc_i + 2; ; 1; begin rd_data_0 = r3; end pc_selector_0 = 1; end else end pc_selector_0 = 1; end else end end end 2; begin rd_data_0 = r4; else end end 5; begin rd_data_0 = r4; else end pc_selector_0 = 0; if (rs_data_0) = r5; end else end pc_selector_0 = 0; if (rs_data_0) = r5; end else end pc_selector_0 = 0; if (rs_data_0) = r5; end else end pc_selector_0 = 0; end else if (wenable_0) rs_data_0 = r4; end else enwpc_0 = oldpc_i + 1; pc_selector_0 = 0; end else if (wenable_0) end rs_data_0 = r6; end else enwpc_0 = oldpc_i + 1; pc_selector_0 = 0; end end end end end end end else enwpc_0 = oldpc_i + 1; pc_selector_0 = 0; end e	begin	4'b1010: begin			end
end	newpc_o = oldpc_i + 1;	if(rs_data_i == 0)	module program_counter	//Read data from register rs	1: begin
else pc_selector_o=0; input clock, case(rs_i) 2; begin begin rd_data_o = r2; end end input wenable_i, rs_data_o = c; end data_o = r3; end data_o = r4; end data_o = r5; end data_o = r5; end data_o = r5; end data_o = r6; end data_o = r7; end data_o = r7; end data_o = r7; end data_o = r8; end data_o = r8; end end end data_o = r8; end	pc_selector_o=0;	begin	(always_comb	rd_data_o = r1;
begin end	end	newpc_o = oldpc_i + 1;	input [7:0] newpc_i,	begin	end
newpc_o = oldpc_i + 2; pc_selector_o=1; else begin newpc_o = oldpc_i + 2; pc_selector_o=1; input reset_i. rs_data_o = c; end end 3: begin rd_data_o = 73; end end 3: begin rd_data_o = 73; end end 4: begin rd_data_o = 73; end 4: begin rd_data_o = 73; end 4: begin rd_data_o = 74; end end 4: begin rd_data_o = 74; end 5: begin rd_data_o = 74; end 5: begin rd_data_o = 74; end 5: begin rd_data_o = 74; end 6: begin rd_data_o = 74; end 6: begin rd_data_o = 75; end 6: begin rd_data_o = 76; end 6: begin rd_data_o = 77; end 6: begin rd_data_o = 76; end 6: begin rd_da	else	pc_selector_o=0;	input clock,	case(rs_i)	2: begin
pc, selector_o=1; begin output [7:0] pc_o end 3: begin end newpc_o = oldpc_i + 2;); 1: begin rd_data_o = r1; end else end rs_data_o = r1; end 4! begin end 4! begin rd_data_o = r4; end 4! begin rd_data_o = r4; end 4! begin rd_data_o = r4; end ssign pc_o = pc; rs_data_o = r2; end end 5: begin rd_data_o = r4; end 5: begin rd_data_o = r4; end 5: begin rd_data_o = r5; end end 6: begin rd_data_o = r5; end end 6: begin rd_data_o = r5; end end 6: begin rd_data_o = r3; end end 6: begin rd_data_o = r5; end end 6: begin rd_data_o = r6; end end 6: begin rd_data_o = r6; end end end end rd_data_o = r6; end end end rd_data_o = r6; end end end end end end end end </td <td>begin</td> <td>end</td> <td>input wenable_i,</td> <td>0: begin</td> <td>rd_data_o = r2;</td>	begin	end	input wenable_i,	0: begin	rd_data_o = r2;
end	newpc_o = oldpc_i + 2;	else	input reset_i,	rs_data_o = c;	end
end	pc_selector_o=1;	begin	output [7:0] pc_o	end	3: begin
else	end	<pre>newpc_o = oldpc_i + 2;</pre>);	1: begin	rd_data_o = r3;
begin end fif(rs_data_i = rd_data_i) 4'b1011: begin iff(rs_data_i = rd_data_i) 4'b1011: begin iff(rs_data_i = rd_data_i) 4'b1011: begin assign pc_o = pc; rs_data_o = r2; end 5: begin rd_data_o = r5; end 5: begin rd_data_o = r5; end end 6: begin rd_data_o = r5; end end 6: begin rd_data_o = r6; end end 6: begin rd_data_o = r6; end end 6: begin rd_data_o = r6; end	end	pc_selector_o=1;		rs_data_o = r1;	end
if(rs_data_i = = rd_data_i) 4'b1011: begin assign pc_0 = pc; rs_data_o = r2; end 5: begin newpc_0 = oldpc_i + 1; begin always_comb 3: begin rd_data_o = r5; pc_selector_o=0; newpc_o = oldpc_i + 1; begin rs_data_o = r3; end end pc_selector_o=0; if (reset_i) begin end 6: begin else end pc.next = 0; 4: begin rd_data_o = r6; begin else end else if (wenable_i) pc.selector_o=1; end newpc_o = oldpc_i + 2; else 5: begin rd_data_o = r6; end pc_selector_o=1; pcnext = newpc_i; end 7: begin end pc_selector_o=1; pcnext = pc; end 8: begin end end 6: begin rg_data_o = r6; end end end 6: begin rg_data_o = r6; end if(rs_data_i) = red_data_i) newpc_o = oldpc_i + 1 pc_selector_o=0; rg_data_o = r6; end end end end end	else	end	reg [7:0] pc, pcnext;	end	4: begin
begin iff(rs_data_i[0] = 1'b0) assign pt_0 = pt. end 5: begin newpc_o = oldpc_i + 1; begin always_comb 3: begin r_data_o = r5; end pc_selector_o=0; newpc_o = oldpc_i + 1; begin end 6: begin else end pc_selector_o=0; if (reset_i) begin end 6: begin newpc_o = oldpc_i + 2; begin end else if (wenable_i) rs_data_o = r4; end end pc_selector_o=1; end else if (wenable_i) rs_data_o = r4; end end pc_selector_o=1; end else if (wenable_i) end 7: begin end pc_selector_o=1; pc_selector_o=0; rs_data_o = r5; end end end end 8: begin rd_data_o = r7; end end 6: begin rs_data_o = r6; end end end 9: begin rs_data_o = r6; end if(rs_data_i >= rd_data_i) pc_selector_o=0; pc_selector_o=0; rs_data_o = r6; end begin pc_selector_o=0;	begin	end		2: begin	rd_data_o = r4;
begin fif(rs_data_ 0 == 1'b0) sin site sin site sin site sin rd_data_o = 15; begin rd_data_o = 15; begin rs_data_o = 13; end site sin rs_data_o = 13; end site sit	if(rs_data_i == rd_data_i)	4'b1011: begin	assign no o = no:	rs_data_o = r2;	end
pc_selector_o=0;	begin	if(rs_data_i[0] == 1'b0)	assign pc_0 = pc,	end	5: begin
pc_selector_o=0; newpc_o = oldpc_i+1; begin rs_data_o = r3; end else end pc_selector_o=0; if (reset_i) begin end 6: begin rd_data_o = r6; begin else end pc_selector_o=0; else end else if (wenable_i) rs_data_o = r4; end rd_data_o = r6; pc_selector_o=1; newpc_o = oldpc_i+2; pc_selector_o=1; pc_selector_o=1; pc_selector_o=1; pc_selector_o=1; pc_selector_o=1; pc_selector_o=0; end	newpc_o = oldpc_i + 1;	begin	always comb	3: begin	rd_data_o = r5;
end else end pc_selector_o=0; if (reset_i) begin end 6: begin rd_data_o = r6; else end pcnext = 0; end else if (wenable_i) rs_data_o = r4; end 7: begin rd_data_o = r6; end else if (wenable_i) rs_data_o = r4; end 7: begin rd_data_o = r7; end 7: begin pcnext = newpc_i; else end pc_selector_o=1; end pc_selector_o=1; pcnext = pc; end end 8: begin rd_data_o = r7; end	pc_selector_o=0;	newpc_o = oldpc_i + 1;	· —	rs_data_o = r3;	end
else begin else end else pcnext = 0; 4: begin rd_data_o = r6; begin newpc_o = oldpc_i + 2; begin pcnext = newpc_i; else end else if (wenable_i) pcnext = newpc_i; else end rd_data_o = r7; begin pcnext = newpc_i; else end rd_data_o = r7; end pcnext = pc; else rs_data_o = r5; end	end	pc_selector_o=0;	9	end	6: begin
begin else end else if (wenable_i) rs_data_o = r4; end rd_data_o = r7; begin rd_data_o = r7; begin rd_data_o = r7; begin rd_data_o = r7; else end rd_data_o = r7; begin rd_data_o = r7; else end rd_data_o = r7; else end rd_data_o = r8; end rd_data_o = r8; end rd_data_o = r8; end rd_data_o = r6; end rd_data_o = r7; end	else	end		4: begin	rd_data_o = r6;
newpc_o = oldpc_i + 2; begin pc_next = newpc_i; end 7: begin pc_selector_o=1; newpc_o = oldpc_i + 2; else 5: begin rd_data_o = r7; end pc_selector_o=1; pcnext = pc; rs_data_o = r5; end end end 8: begin rd_data_o = r8; end rs_data_o = r6; end if(rs_data_i) = rd_data_i) newpc_o = oldpc_i + always_ff @(posedge clock) end 9: begin begin jump_const_i; pc <= pcnext;	begin	else	•	rs_data_o = r4;	end
pc_selector_o=1; newpc_o = oldpc_i + 2; else 5: begin rd_data_o = r7; end end end 8: begin end end 6: begin rd_data_o = r8; 4'b0110: begin 4'b1100: begin rs_data_o = r6; end if(rs_data_i >= rd_data_i) newpc_o = oldpc_i + always_ff @(posedge clock) end 9: begin begin jump_const_i; pc <= pcnext;	newpc_o = oldpc_i + 2;	begin	, =,	end	7: begin
end	pc_selector_o=1;	<pre>newpc_o = oldpc_i + 2;</pre>		5: begin	rd_data_o = r7;
end end end end 6: begin rd_data_o = r8; 4'b0110: begin 4'b1100: begin newpc_o = oldpc_i + always_ff @(posedge clock) end 9: begin rd_data_o = r8; begin jump_const_i; pc <= pcnext; 7: begin rd_data_o = r7; end default: begin rd_data_o = temp; newpc_o = oldpc_i + 1; pc_selector_o=1; end end default: begin rd_data_o = r1; end 4'b1101: begin end end end default: begin rd_data_o = r1; else newpc_o = oldpc_i + 1; module register_file rs_data_o = r8; newpc_o = oldpc_i + 2; end	end	<pre>pc_selector_o=1;</pre>		rs_data_o = r5;	end
end	end	end		end	8: begin
if(rs_data_i) = rd_data_i) begin	end	end	ena	6: begin	rd_data_o = r8;
begin jump_const_i; pc <= pcnext; 7: begin rd_data_o = temp; newpc_o = oldpc_i + 1; pc_selector_o=1; rs_data_o = r7; end pc_selector_o=0; end end default: begin rd_data_o = r1; else newpc_o = oldpc_i + 1; module register_file rs_data_o = r8; end begin pc_selector_o=0; (end endcase newpc_o = oldpc_i + 2; end input [3:0] rs_i, 9: begin end pc_selector_o=1; 4'b1110: begin input [3:0] rt_i, rs_data_o = temp; end newpc_o = oldpc_i + 1; input [3:0] write_reg_i, end end pc_selector_o=0; input [7:0] reg_data_i, default: begin // Write 4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b1111: begin input clock, end begin	4'b0110: begin	4'b1100: begin		rs_data_o = r6;	end
newpc_o = oldpc_i + 1;	if(rs_data_i >= rd_data_i)	newpc_o = oldpc_i +	, =	end	9: begin
pc_selector_o=0; end endmodule end default: begin end 4'b101: begin 8: begin rd_data_o = r1; else newpc_o = oldpc_i + 1; module register_file rs_data_o = r8; end begin pc_selector_o=0; (end endcase newpc_o = oldpc_i + 2; end input [3:0] rs_i, 9: begin end pc_selector_o=1; 4'b1110: begin input [3:0] rt_i, rs_data_o = temp; end newpc_o = oldpc_i + 1; input [3:0] write_reg_i, end end pc_selector_o=0; input [7:0] reg_data_i, default: begin // Write 4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b111: begin input clock, end begin	begin	jump_const_i;	pc <= pcnext;	7: begin	rd_data_o = temp;
end 4'b1101: begin 8: begin rd_data_o = r1; else newpc_o = oldpc_i + 1; module register_file rs_data_o = r8; end begin pc_selector_o=0; (end endcase newpc_o = oldpc_i + 2; end input [3:0] rs_i, 9: begin end pc_selector_o=1; 4'b1110: begin input [3:0] rt_i, rs_data_o = temp; end newpc_o = oldpc_i + 1; input [3:0] write_reg_i, end end pc_selector_o=0; input [7:0] reg_data_i, default: begin // Write 4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b1111: begin input clock, end begin	newpc_o = oldpc_i + 1;	<pre>pc_selector_o=1;</pre>		rs_data_o = r7;	end
else newpc_o = oldpc_i + 1; module register_file rs_data_o = r8; end begin pc_selector_o=0; (end endcase newpc_o = oldpc_i + 2; end input [3:0] rs_i, 9: begin end pc_selector_o=1; 4'b1110: begin input [3:0] rt_i, rs_data_o = temp; end newpc_o = oldpc_i + 1; input [3:0] write_reg_i, end end pc_selector_o=0; input [7:0] reg_data_i, default: begin // Write 4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b1111: begin input clock, end begin	pc_selector_o=0;	end	endmodule	end	default: begin
begin pc_selector_o=0; (end endcase newpc_o = oldpc_i + 2; end input [3:0] rs_i, 9: begin end pc_selector_o=1; 4'b1110: begin input [3:0] rt_i, rs_data_o = temp; end newpc_o = oldpc_i + 1; input [3:0] write_reg_i, end end pc_selector_o=0; input [7:0] reg_data_i, default: begin // Write 4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b1111: begin input clock, end begin	end	4'b1101: begin		8: begin	rd_data_o = r1;
begin pc_selector_o=0; (end endcase newpc_o = oldpc_i + 2; end input [3:0] rs_i, 9: begin end pc_selector_o=1; 4'b1110: begin input [3:0] rt_i, rs_data_o = temp; end newpc_o = oldpc_i + 1; input [3:0] write_reg_i, end end pc_selector_o=0; input [7:0] reg_data_i, default: begin // Write 4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b111: begin input clock, end begin	else	newpc_o = oldpc_i + 1;	module register file	rs_data_o = r8;	end
pc_selector_o=1; 4'b1110: begin input [3:0] rt_i, rs_data_o = temp; end newpc_o = oldpc_i + 1; input [3:0] write_reg_i, end end pc_selector_o=0; input [7:0] reg_data_i, default: begin // Write 4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b111: begin input clock, end begin	begin	pc_selector_o=0;	(end	endcase
end newpc_o = oldpc_i + 1; input [3:0] write_reg_i, end end pc_selector_o=0; input [7:0] reg_data_i, default: begin // Write 4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b111: begin input clock, end begin	newpc_o = oldpc_i + 2;	end	input [3:0] rs_i,	9: begin	end
end pc_selector_o=0; input [7:0] reg_data_i, default: begin // Write 4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b111: begin input clock, end begin	pc_selector_o=1;	4'b1110: begin	input [3:0] rt_i,	rs_data_o = temp;	
4'b0111: begin end input [7:0] c_data_i, rs_data_o = r1; always_ff @ (negedge clock) newpc_o = oldpc_i + 1; 4'b1111: begin input clock, end begin	end	newpc_o = oldpc_i + 1;	input [3:0] write_reg_i,	end	
newpc_o = oldpc_i + 1; 4'b1111: begin input clock, end begin	end	pc_selector_o=0;		default: begin	// Write
newpc_o = oldpc_i + 1; 4'b1111: begin input clock, end begin	4'b0111: begin	end	input [7:0] c_data_i,	rs_data_o = r1;	always_ff @ (negedge clock)
pc_selector_o=0; newpc_o = oldpc_i + 1; endcase if(wenable_i)	newpc_o = oldpc_i + 1;	4'b1111: begin		end	begin
	pc_selector_o=0;	newpc_o = oldpc_i + 1;		endcase	if(wenable_i)

begin		endcase	byte FDjump_const;	done = 0;
case(write reg i)	and	end	byte CFjump const;	new pc = 0; // Set PC to
0: begin	end	endmodule	reg CFmem read;	program
c = reg_data_i;	else		reg CFmem_write;	FDhazard=0;
end	begin	module	reg EXmem read;	CFhazard=0;
1: begin	// Don't write	pipeline_testbench();	reg EXmem_write;	FDnew_pc=0;
r1 = reg_data_i;		// Declare inputs as regs and	reg CFtaken;	instr counter=0;
c = c_data_i;	end	outputs as wires	reg FDhazard;	st554e. 5)
end	end	reg init;	reg CFhazard;	
2: begin		reg clk;	reg CFpc sel;	// initial value of new_pc
r2 = reg_data_i;	endmodule	reg done;	reg CFrs_sel;	changes depending on program
c = c_data_i;		reg done,	reg CFrt sel;	#10 init = 1; new_pc = 126;
end	module	reg [15:0] cycle_counter;	reg Cirt_sei,	cycle_counter = 0;
3: begin	rs_three_to_one_mux (reg [7:0] FDnew pc;	reg taken;	
	input logic [1:0] sel_i,			end
r3 = reg_data_i;	input logic [7:0] first_i,	reg [7:0] FDpc;	reg[7:0] EXinstruction;	2.10
c = c_data_i;	input logic [7:0] second_i,	reg [7:0] CFpc;	reg pc_selector;	// Clock generator
end	input logic [7:0] third_i,	reg [7:0] FDinstruction;	reg [1:0] CFrs_selector;	always begin
4: begin	output logic [7:0] result_o	reg [7:0] CFinstruction;	reg [1:0] CFrt_selector;	#5 clk = $^{\circ}$ clk; if(clk) begin
r4 = reg_data_i;);	reg [3:0] FDrs;	reg [7:0] CFrs_result;	
c = c_data_i;		reg [3:0] FDrt;	reg [7:0] CFrt_result;	cycle_counter++; end
end	always comb	reg [3:0] CFrs;	reg [7:0] CFnew_pc;	if(clk &&!CFhazard)begin
5: begin	begin	reg [3:0] CFrt;	reg [9:0] instr_counter;	instr_counter++; end// Toggle
r5 = reg_data_i;	case(sel i)	reg [3:0] EXrs;		clock every 5 ticks
c = c_data_i;	0: result o = first i;	reg [3:0] EXrt;	reg [7:0] new_pc;	// this makes the clock
end	1: result o = second i;	reg [3:0] FDrd;	reg [7:0] pc;	cycle 10 ticks
6: begin	2: result o=third i;	reg [3:0] CFrd;	reg [7:0] instruction;	
r6 = reg_data_i;	default: result_o=first_i;	reg [3:0] EXrd;	reg [3:0] rs;	
c = c_data_i;	endcase	reg [3:0] FDwrite_reg;	reg [3:0] rt;	
end		reg [3:0] EXwrite_reg;	reg [3:0] write_reg;	end
7: begin	end	reg [7:0] EXreg_data;	reg [7:0] reg_data;	
r7 = reg_data_i;	endmodule	reg [7:0] EXc data;	reg [7:0] c_data;	program_counter b2v_inst(
c = c_data_i;		reg CFreg_wenable;	reg reg_wenable;	.clock(clk),
end	module	reg EXreg_wenable;	reg [7:0] rs_data;	.wenable_i(init),
8: begin	rt_three_to_one_mux (reg [7:0] CFrs_data;	reg [7:0] rd_data;	.reset_i(0),
r8 = reg_data_i;	input logic [1:0] sel_i,	reg [7:0] CFrt_data;	reg [3:0] alu_opt;	.newpc_i(new_pc),
c = c_data_i;	input logic [7:0] first_i,	reg [7:0] EXrs data;	reg [7:0] result;	.pc_o(FDpc));
end	input logic [7:0] second_i,	reg [7:0] EXrd_data;	reg [7:0] q;	
9: begin	input logic [7:0] third_i,	reg [7:0] EXrd_data;	reg [1:0] q,	
temp = reg_data_i;	output logic [7:0] result_o	reg [3:0] CFalu opt;	write register selector;	//alu instance
end);		_ = = -	alu b3v inst(
		reg [3:0] EXalu_opt;	reg carry_out_bit;	.rs_data_i(EXrs_data),
13: begin	always_comb	reg [7:0] EXresult_write;	byte jump_const;	.rd_data_i(EXrt_data),
r2 = reg_data_i;	begin	reg [7:0] EXresult_alu;	reg mem_read;	
c = c_data_i;	case(sel_i)	reg [7:0] EXq;	reg mem_write;	.alu_opt_i(EXalu_opt),
end	0: result_o = first_i;	reg [1:0]		.result_o(EXresult_alu),
default: begin	1: result_o = second_i;	CFwrite_register_selector;		.carry_bit_o(EXcarry_out_bit));
r1 = reg_data_i;	2: result o=third i;	reg [1:0]	initial begin	
end	default: result o=first i;	EXwrite_register_selector;	clk = 1;	// instruction_decoder instance
endcase	,	reg EXcarry_out_bit;	init = 0;	instruction_decoder b1v_inst(

.instr_i(FDinstruction), .rs_o(FDrs), .rd_o(FDrt), .write_reg_o(FDrd), .jump_const_o(FDjump_const));	// control unit instance control_unit b8v_inst (.instr_i(CFinstruction), .rs_data_i(CFrs_result), .alu_opt_o(CFalu_opt),	.data(EXrs_data), .addr(EXrt_data), .writemem(EXmem_write), .readmem(EXmem_read), .clk(clk), .q(EXq)	.rd_i(FDrd), .pc_i(FDpc), .clk(clk), .rs_o(CFrs), .rt_o(CFrt), .instr_o(CFinstruction),	.result_o(CFrs_result)); rt_three_to_one_mux b20v_inst(.sel_i(CFrt_selector), .first_i(CFrt_data), .second_i(EXresult_write),
// register_file instance register_file b5v_inst(.rs_i(CFrs), .rt_i(CFrt), .write_reg_i(EXrd), .reg_data_i(EXresult_write), .c_data_i(EXc_data), .clock(clk), .wenable_i(EXreg_wenable),	<pre>.reg_write_o(CFreg_wenabl e), .write_reg_selector_o(CFwri te_register_selector), .mem_read_o(CFmem_read)</pre>); carry_out_extender b11v_inst(.carry_out_i(EXcarry_out_bi t), .carry_out_o(EXc_data));	.jump_const_o(CFjump_const), .rd_o(CFrd), .pc_o(CFpc)); forwarding_unit b17v_inst(.CFinstr_i(CFinstruction),	<pre>.third_i(EXc_data), .result_o(CFrt_result)); CF_EX b19v_inst(.rs_data_i(CFrs_result), .rt_data_i(CFrt_result), .rd_i(CFrd), .alu_opt_i(CFalu_opt), .reg_write_i(CFreg_wenable),</pre>
.rs_data_o(CFrs_data), .rd_data_o(CFrt_data)); // next_pc_logic instance next_pc_logic b6v_inst (.instr_i(CFinstruction), .oldpc_i(CFpc), .rs_data_i(CFrs_result),	.mem_write_o(CFmem_writ e)); //write_reg_mux instance write_reg_mux b9v_inst (two_to_one_mux b12v_inst(.sel_i(pc_selector), .first_i(FDnew_pc), .second_i(CFnew_pc), .result_o(new_pc));	<pre>.rs_i(CFrs), .rt_i(CFrt), .rd_i(EXrd), .EXinstruct_i(EXinstruction), .reg_write_i(EXreg_wenable),</pre>	.write_reg_selector_i(CFwrite_reg ister_selector), .mem_read_i(CFmem_read), .mem_write_i(CFmem_write), .clk(clk), .instruct_i(CFinstruction), .rs_data_o(EXrs_data),
<pre>.rd_data_i(CFrt_result), .jump_const_i(CFjump_const), .newpc_o(CFnew_pc), .pc_selector_o(pc_selector));</pre>	<pre>.write_reg_selector_i(EXwrit e_register_selector), .alu_result_i(EXresult_alu), .reg_result_i(EXrs_data), .mem_result_i(EXq), .result_o(EXresult_write));</pre>	adder b13v_inst(.pc_i(FDpc), .new_pc_o(FDnew_pc)); //add in hazard FD_CF b18v_inst(.rs_i(FDrs),	<pre>.rs_selector_o(CFrs_selector), .rt_selector_o (CFrt_selector)); rs_three_to_one_mux</pre>	<pre>.rt_data_o(EXrt_data), .alu_opt_o(EXalu_opt), .rd_o(EXrd), .reg_write_o(EXreg_wenable), .write_reg_selector_o(EXwrite_re gister_selector), .mem_read_o(EXmem_read),</pre>
<pre>// instruction_ROM instance instruction_ROM b7v_inst (.address_i(FDpc), .instruction_o(FDinstruction));</pre>	memory_data_RAM b10v_inst (<pre>.rt_i(FDrt), .instr_i(FDinstruction), .jump_const_i(FDjump_cons t),</pre>	b14v_inst(.sel_i(CFrs_selector), .first_i(CFrs_data), .second_i(EXresult_write), .third_i(EXc_data),	.mem_write_o(EXmem_write), .instruct_o(EXinstruction)); endmodule

TIMING DIAGRAMS

Some notes before reading diagrams:

We will label each section according to the cycle counter.

FD represents our first stage of our pipelined process where we fetch and decode the instruction.

CF represents the second stage of our pipelined process where we prepare the register value and resolve the next PC value.

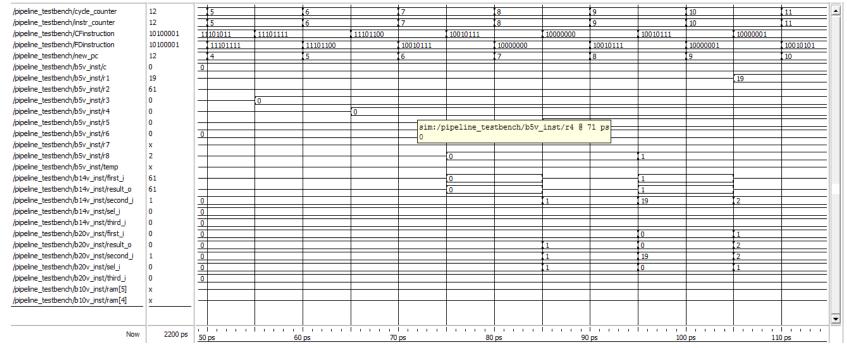
EX represents the third stage which is our execution stage including memory access.

Branch and jump resolutions are performed in CF stage.

Our design includes forwarding between CF and EX tage.

Our design does not have any branching hazards.

Program 1 (Calculates the product of three unsigned integers):



This timing diagram shows the beginning as we initialize the variables and load out the value of A from memory. It also shows a case where we forward our execution result to on next instruction.

5. FD: PC= instruction for shift \$r5 right logical by 8

CD: Preparing data to execute shift \$r8 by 8

EX: Executing shift \$r3 logical by 8, clearing \$r3, \$r3=0

- 6. FD: PC=instruction for increment \$r8
 - CF: Preparing data to execute shift \$r5 right logical by 8
 - EX: Executing shift \$r4 logical by 8, clearing \$r4, \$r4=0
- 7. FD: PC=instruction for loading memory address \$r8 to \$r1
 - CF: Preparing data to increment \$r8
 - EX: Executing shift \$r5 right logical by 8, \$r5=0
- 8. FD: PC=instruction to increment \$r8
 - CF: Preparing data to load memory address \$r8 to \$r1
 - EX: Executing increment \$r8, \$r8=1, we also forward this value to load instruction because it uses \$r8.
 - Our b20v sel i is 1, which shows our mux choose the forwarded data, second i, and not old \$78 value.
- 9. FD: PC=instruction to load data in address \$r8 to \$r2
 - CF: Preparing data to load memory address \$r8 to \$r1
 - EX: Execute load memory address \$r8 which is 1 from the forwarded value to \$r1
- 10. FD: PC=instruction to increment \$r6
 - CF: Preparing data to load data in address \$r8 to \$r2
 - EX: Executing increment \$r8, \$r8=2

/pipeline testbench/cycle counter	220	14	15	1	16	1	17	1	18	1	19	<u> </u>	20	<u> </u>	21	_
/pipeline_testbench/instr_counter	220	14	15		16		17		18		19		120		21	=
/pipeline testbench/CFinstruction	00000000	1011000	1	10011001		00010110		00010011		00011011		11000101		01101101		
/pipeline testbench/FDinstruction	00000000	101	10011001		00010110		00010011		00011011		11000101		01101101		1011000	01
/pipeline_testbench/new_pc	72	16		17	18		19		20		21	12		14		
/pipeline_testbench/b5v_inst/c	0	0														=
/pipeline_testbench/b5v_inst/r1	112	19														=
/pipeline_testbench/b5v_inst/r2	1	61						60								
pipeline_testbench/b5v_inst/r3	72	0														=
pipeline_testbench/b5v_inst/r4	33	0								19						
/pipeline_testbench/b5v_inst/r5	104	0														
pipeline_testbench/b5v_inst/r6	1	1														
pipeline_testbench/b5v_inst/r7	x					-	-		-			-		-	+	
oipeline_testbench/b5v_inst/r8	5	2														
pipeline_testbench/b5v_inst/temp	0					-	-		-					-	+	
pipeline_testbench/b14v_inst/first_i	112	61				19		0				19		1		
pipeline_testbench/b14v_inst/result_o	112	61				19		0				19		1		
pipeline_testbench/b14v_inst/second_i	112	62		30		60		19		0				19		
pipeline_testbench/b14v_inst/sel_i	0	0						2		0						
pipeline_testbench/b14v_inst/third_i	0	0		1		0										
pipeline_testbench/b20v_inst/first_i	0	0												60		
pipeline_testbench/b20v_inst/result_o	0	0								0				60		
pipeline_testbench/b20v_inst/second_i	112	62		30		60		19		0				19		
pipeline_testbench/b20v_inst/sel_i	0	0								1		0				
pipeline_testbench/b20v_inst/third_i	0	0		1		0										
pipeline_testbench/b10v_inst/ram[5]	33														+	
/pipeline_testbench/b10v_inst/ram[4]	104						-		-					-	+	_
Now	2200 ps	150			l i i i i		l i i i i		l i i i i		l i i i i		l i i i i		0 ps	1 1

This timing diagram shows an example of branch where there is no hazard as we figure out that 61 is not divisible by 2. It will also show when there is an hazard with the jump instruction and how it is handled.

- 15. FD: PC=Instruction to decrement \$r2
 - CF: Prepare value to check if \$r2 is divisible by 2
 - EX: Operation for branch, but branch has no effect during this stage
- 16. FD: PC=Instruction to add \$r1 to \$r4
 - CF: Prepare value to decrement \$r2
 - EX: Operation for branch, but branch has no effect during this stage
- 17. FD: PC=Instruction to add \$c to \$r5
 - CF: Prepare value to add \$r1 to \$r4
 - EX: Execute to decrement \$r2. \$r2=60
- 18. FD: PC=jump instruction
 - CF: Prepare value to add \$c to \$r5
 - EX: Execute to add \$r1 to \$r4, \$r4=19
- 19. FD: PC=instruction to shift right logical
 - CF: Execute jump instruction
 - EX: Execute to add \$c to \$r5

20. FD: PC =instruction to check if \$r2 is divisible by 2

CF: Execute check \$r6>=\$r2, false, so we increment program counter by 2, pc=14

EX: Non action jump instruction

/pipeline_testbench/cycle_counter	58	52		53		54		55		56		57		.58
/pipeline_testbench/instr_counter	58	52		53		54		55		56		57		58
/pipeline_testbench/CFinstruction	01111000	01101101	10110001		11001000		11100001		00110011		11110011		01111000	
/pipeline_testbench/FDinstruction	01001100	10110001		11001000		11100001		00110011		11110011		01111000		01001100
/pipeline_testbench/new_pc	26	14	15	16	21		22	23		24		25		26
/pipeline_testbench/b5v_inst/c	0	0												
pipeline_testbench/b5v_inst/r1	76	76												
pipeline_testbench/b5v_inst/r2	7	14									7			
/pipeline_testbench/b5v_inst/r3	0	0												
/pipeline_testbench/b5v_inst/r4	95	0 95												
pipeline_testbench/b5v_inst/r5	0	0												
pipeline_testbench/b5v_inst/r6	1	1												
pipeline_testbench/b5v_inst/r7	x	-					+					-		++-
pipeline_testbench/b5v_inst/r8	76	76												
pipeline_testbench/b5v_inst/temp	0	0												
pipeline_testbench/b14v_inst/first_i	0	1	14		76		14		76				0	
pipeline_testbench/b14v_inst/result_o	0	1	14		76		14		76				0	
pipeline_testbench/b14v_inst/second_i	152	76	15		7		76		7		76		152	
pipeline_testbench/b14v_inst/sel_i	2	0									1		2	
pipeline_testbench/b14v_inst/third_i	0	0												
pipeline_testbench/b20v_inst/first_i	76	14	0						76		0		76	
pipeline_testbench/b20v_inst/result_o	152	14	0						76		0		152	
pipeline_testbench/b20v_inst/second_i	152	76	15		7		76		7		76		152	
pipeline_testbench/b20v_inst/sel_i	1	0											1	
pipeline_testbench/b20v_inst/third_i	0	0												
pipeline_testbench/b10v_inst/ram[5]	x			+			1			-			-	++-
pipeline_testbench/b10v_inst/ram[4]	x	1		+			+	+		+			+	++-
		-		1										11
							<u> </u>	<u> </u>				<u> </u>		<u> </u>
Now	2200 ps	520 ps		1 30 ps		liiii Ops		liiii Ops		liiii Ops		liiii Ops		80 ps

This timing diagram shows an example of branch where there is a hazard as we figure out that 61 is divisible by 2 because we take the jump instruction.

- 52. FD: PC=instruction to jump
 - CF: Load data to check if \$r2 is divisible by 2
 - EX: Operation for branch, but branch has no effect during this stage
- 53. FD: PC=instruction to shift \$r2 right logical by 1
 - CF: Execute to jump instruction because that logic is in the CF stage
 - EX: Operation for branch, but branch has no effect during this stage
- 54. FD: PC=instruction to move value in \$r1 to \$r8
 - CF: Prepare data to shift \$r2 right logical by 1
 - EX: Operation for jump, but jump has no effect during this stage
- 55. FD: PC=instruction to shift \$r2 right logical by 1
 - CF: Prepare data to move value in \$r1 to \$r8
 - EX: Execute shift \$r2 right logical by 1
- 56. FD: PC=instruction to Move temp to c
 - CF: Prepare data to shift \$r2 right logical by 1

EX: Execute move value in \$r1 to \$r8

57. FD: PC=Move value in \$r8 to \$r1

CF: Prepare data to move temp to c

EX: Execute shift \$r2 right logical by 1, \$r2=7

pipeline_testbench/cycle_counter	58	192		193		194		195		196		197		198	
pipeline_testbench/instr_counter	58	192		193		194		195		196		197		198	
pipeline_testbench/CFinstruction	01111000	1001	0111				01110100		10010111		01110011		00000000		
pipeline_testbench/FDinstruction	01001100	1001	0111			01110100		10010111		01110011		00000000			
pipeline_testbench/new_pc	26	68		69		70		71		72		.73	72		
pipeline_testbench/b5v_inst/c	0	0													
pipeline_testbench/b5v_inst/r1	76	112													
pipeline_testbench/b5v_inst/r2	7	1													
pipeline_testbench/b5v_inst/r3	0	72													
pipeline_testbench/b5v_inst/r4	95	33													
pipeline_testbench/b5v_inst/r5	0	104													
pipeline_testbench/b5v_inst/r6	1	1													
pipeline_testbench/b5v_inst/r7	x		-					-	-			-	-		
oipeline_testbench/b5v_inst/r8	76	0	1		2		3		4				5		
pipeline_testbench/b5v_inst/temp	0	0													
pipeline_testbench/b14v_inst/first_i	0	0	1		2		104		4		33		112		
pipeline_testbench/b14v_inst/result_o	0	1	2		3		104		4		33		112		
pipeline_testbench/b14v_inst/second_i	152	1	2		3		4		108		5		38		112
pipeline_testbench/b14v_inst/sel_i	2	1					0								
pipeline_testbench/b14v_inst/third_i	0	0													
pipeline_testbench/b20v_inst/first_i	76	0					3		0		4		0		
pipeline_testbench/b20v_inst/result_o	152	0					4		0		5		0		
pipeline_testbench/b20v_inst/second_i	152	1	2		3		4		108		5		38		112
pipeline_testbench/b20v_inst/sel_i	1	0					1		0		1		0		
pipeline_testbench/b20v_inst/third_i	0	0													
pipeline_testbench/b10v_inst/ram[5]	x	-												33	
/pipeline_testbench/b10v_inst/ram[4]	x				1			-	-	104					
		-		1	<u> </u>				<u> </u>	<u> </u>				<u> </u>	
Now	2200 ps	'		liiii 30 ps	194	Ops	195	liiii Ops		liiii Sops		liiii Ops	198	liiii Bops	

This timing diagram is the end as we are storing are values into memory ram.

- 192. FD: PC=instruction increment \$r8
 - CF: Prepare data to increment \$r8
 - EX: Execute increment \$r8, \$r8=2
- 193. FD: PC=instruction to store \$r4 value to the address in \$r8
 - CF: Prepare data to increment \$r8
 - EX: Execute increment \$r8, \$r8=3
- 194. FD: PC=instruction increment \$r8
 - CF: Prepare data to store \$r4 value to the address in \$r8
 - EX: Execute increment \$r8, \$r8=4, forward the \$r8 value to store instruction, which is shown by the b20v selector_i being 1. We are choosing the forwarded data instead of the old register value of \$r8
- 195. FD: PC= instruction store \$5 value to the address in \$18
 - CF: Prepare data to increment \$r8
 - EX: Execute store \$r4 value to the address in \$r8 which has the value 4 which was forwarded to it. Next cycle it will be written in the memory ram
- 196. Ram[4]=104
 - FD: PC= instruction to halt
 - CF: Prepare data to store \$5 value to the address in \$r8

EX: Execute increment \$r8, \$r8=5, forward the \$r8 value to store instruction, which is shown by the b20v selector_i being 1. We are choosing the forwarded data instead of the old register value of \$r8

197. FD: Halt CF: Halt

EX: Execute store \$r5 value to the address in \$r8 which has the value 4 which was forwarded to it. Next cycle it will be written in the memory ram 198. Ram[5]=33

Program 2 (Counts the number of entries in an 64-byte array which contain a 4-bit string): (NOTE: The listed numbers are the cycle_counter values to help guide you).

The first set of sel_i, first_i, second_i, third_i signals refer to the forwarding selector for rs. The second set refers to the fowarding selector for rt.

When sel_i = 0, do not foward data from EX (use first_i value). If sel_i = 1, forward data from EX (use second_i value). If sel_i=2, forward carry out from EX (use third_i value). sel_i values should be checked on the rising edge (that's when reads should happen).

(THE FOLLOWING DIAGRAMS SHOW INITIALIZATION OF PROGRAM 2)

/pipeline_testbench/cycle_counter	2000	<u> </u>		1		2		3		4	X	5	6		. 7	7	X 8	3		9		10	X	11	=
/pipeline_testbench/instr_counter	2000	(0		1		2	X	3	<u> </u>	4		5	(6		1,7	7) E	3		9		10	X	11	
/pipeline_testbench/FDpc	125					83	X	84		85	X	86	8	7	(8	88		39		90		91	X	92	
/pipeline_testbench/FDinstruction	00000000	(000000				1110110		111011	11	100101	11 (11110111	. 1	001011	1		χ:	10000001		1111100	00	111000	001		\perp
/pipeline_testbench/CFinstruction	00000000		000000	00			1110110	1	111011	11	1001011	1 (1110111		10010111			1	000000	1	111110	00	1110000	1	\perp
/pipeline_testbench/EXinstruction	00000000				0000000	00			1110110)1	1110111	1 (0010111		11110111	. (:	10010111	1			100000	01	1111100	0	111
/pipeline_testbench/b5v_inst/r1	0																								+
/pipeline_testbench/b5v_inst/r2	5																						5		80
/pipeline_testbench/b5v_inst/r3	0																								-
/pipeline_testbench/b5v_inst/r4	0													_											+-
/pipeline_testbench/b5v_inst/r5	14																								+-
/pipeline_testbench/b5v_inst/r6	0										0														
/pipeline_testbench/b5v_inst/r7	5																								_
/pipeline_testbench/b5v_inst/r8	7											—— <u>(</u>		X	1) 4	4	(5			6				
/pipeline_testbench/CFrs	1		1				6		8									(2							\perp
/pipeline_testbench/CFrt	0		-{0															8			0				
/pipeline_testbench/EXrd	0				0				6		8										2				
/pipeline_testbench/b3v_inst/result_o	0	(0	}						(0			χ:			4	X:	5	(6					80		40
/pipeline_testbench/b14v_inst/sel_i	0	(0									1							(0			1				
/pipeline_testbench/b14v_inst/first_i	0											——{ <u>c</u>		X.	1) <u>-</u>	4	\longrightarrow					-{5		80
/pipeline_testbench/b14v_inst/second_i	0	(0							0			χ:		χ.	4	χ.	5	(6			5		80		40
/pipeline_testbench/b14v_inst/third_i	0	(0																							
/pipeline_testbench/b20v_inst/sel_i	0	(0																1			0				\perp
/pipeline_testbench/b20v_inst/first_i	0										0							, 5			0				
/pipeline_testbench/b20v_inst/second_i	0	(0							0			χ:		X.	4	χ.	5	(6			5		80		40
/pipeline_testbench/b20v_inst/third_i	0	(0																							
/pipeline_testbench/b10v_inst/ram[7]	14																								+
			11111	 										1111						1111		1			
Now	20000 ps	os			20				40				60 ps				80 p) ps			12

2: FD: Fetch/decode srl 8, \$r6 instruction

CF: Nothing EX: Nothing

3: FD: Fetch/decode srl 8, \$r8 instruction

CF: Prepare rs = \$r6 for shift right logical 8 (clear)

EX: Nothing

4: FD: Fetch/decode addi 0, \$r8 instruction

CF: Prepare rs = \$r8 for shift right logical 8 (clear)

EX: Clear \$r6, write to \$r6, \$r6 = 0

5: FD: Fetch/decode sll 2, \$r8 instruction

CF: Prepare rs = \$r8 for \$r8++, forward \$r8 value from EX (the first sel i = 1 (rs gets forwarded), the second sel i = 0 (rt not forwarded))

- EX: Clear \$r8, write to \$r8, \$r8= 0
- 6: FD: Fetch/decode addi 0, \$r8 instruction
 - CF: Prepare rs = \$r8 for shift left logical 2, forward \$r8 value from EX
 - EX: \$r8++, write to \$r8, \$r8 = 1
- 7: FD: Fetch/decode addi 0, \$r8 instruction
 - CF: Prepare rs = \$r8 for \$r8++, forward \$r8 value from EX
 - EX: sll 2 \$r8, write to \$r8, \$r8 = 4
- 8: FD: Fetch/decode Id \$r2 instruction
 - CF: Prepare rs = \$r8 for \$r8++, forward \$r8 value from EX
 - EX: \$r8++, write to \$r8, \$r8 = 5
- 9: FD: Fetch/decode sll 4, \$r2 instruction
 - CF: Prepare rs = \$r2, rt = \$r8 for load, forward \$r8 value from EX (the first sel i = 0 (rs does not get forwarded), but the second sel i = 1 (rt gets forwarded))
 - EX: \$r8++, write to \$r8, \$r8 = 6
- 10: FD: Fetch/decode srl 1,\$r2 instruction
 - CF: Prepare rs = \$r2 for shift left logical 4, forward \$r2 value from EX
 - EX: load into \$r2, now \$r2 holds 8-bit string containing the 4-bit substring
- 11: FD: Fetch/decode srl 1, \$r2 instruction
 - CF: Prepare rs = r^2 for shift right logical 1, forward r^2 value from EX (first sel i = 1 (rs forwarded), second sel i = 0 (rt not forwarded))
 - EX: sll 4 \$r2, write to \$r2

/pipeline_testbench/cycle_counter	2000	(12	X :	13	X	14		15		16		17		18	(19		20		21		22	X	23	
/pipeline_testbench/instr_counter	2000	(12	X:	13	X	14		15		16		17		18	19		20		21		22	X	23	
/pipeline_testbench/FDpc	125	93) <u> </u>	94	X	95	1	96		97		98		99	100		101		102		103	X	104	
/pipeline_testbench/FDinstruction	00000000	1110000)1		X	1110110	00	111010	10	100100	10	010011	10	100111	10 (11111	101	001110	11	111001	11	111010	00 (101000	10
/pipeline_testbench/CFinstruction	00000000	1110000)1			X	111011	00	111010	10	100100	10	010011	10	10011110	111111	01	001110	11	111001	11	1110100	0	101
/pipeline_testbench/EXinstruction	00000000	1110000)1						111011	00	111010	10	100100	10	01001110	100111	10	111111	01	001110	11	1110011	1	111
/pipeline_testbench/b5v_inst/r1	0																_	_						
/pipeline_testbench/b5v_inst/r2	5	80	40	X	20	X	10		5															
/pipeline_testbench/b5v_inst/r3	0												(0		<u>) 1</u>					64				
/pipeline_testbench/b5v_inst/r4	0																							
/pipeline_testbench/b5v_inst/r5	14										0													
/pipeline_testbench/b5v_inst/r6	0	0																						
/pipeline_testbench/b5v_inst/r7	5															-{6		15						
/pipeline_testbench/b5v_inst/r8	7	6																				64		32
/pipeline_testbench/CFrs	1	2				X	5		(3				8		17	(3				(8		1		13
/pipeline_testbench/CFrt	0	0											7		χo			(8		(o				
/pipeline_testbench/EXrd	0	2							15		3				17			13		8				1 1
/pipeline_testbench/b3v_inst/result_o	0	40	20	X	10		5		O				1)	-{5		64		70		32		χo
/pipeline_testbench/b14v_inst/sel_i	0	1				x	0				1		i o		11	χo		11				0		
/pipeline_testbench/b14v_inst/first_i	0	80 1	40	X	20								6			1				(6				64
/pipeline_testbench/b14v_inst/second_i	0	40 X	20	Ý	10	<u> </u>	5		(O				1		16	15		64				32		χo
/pipeline_testbench/b14v_inst/third_i	0	0																						
/pipeline_testbench/b20v_inst/sel_i	0	0																						
/pipeline_testbench/b20v_inst/first_i	0	0											} ——		{0			16		ίο				
/pipeline_testbench/b20v_inst/second_i	0	40 X	20	Ţ,	10	X	5		(O				1		16	15		64				32		(0
/pipeline_testbench/b20v_inst/third_i	0	0																						
/pipeline_testbench/b10v_inst/ram[7]	14																							
Now	20000 ps	11111	1111		140					D ps) ps	inintin		liiii Ops		Livia		D ps		11111	24

12: FD: Fetch/decode srl 1, \$r2 instruction

CF: Prepare rs = \$r2 for shift right logical 1, forward \$r2 value from EX

EX: srl 1 \$r2, write to \$r2

13: FD: Fetch/decode srl 1, \$r2 instruction

CF: Prepare rs = \$r2 for shift right logical 1, forward \$r2 value from EX

EX: srl 1 \$r2, write to \$r2

14: FD: Fetch/decode srl 8, \$r5 instruction

CF: Prepare rs = \$r2 for shift right logical 1, forward \$r2 value from EX

EX: srl 1 \$r2, write to \$r2

15: FD: Fetch/decode srl 8, \$r3 instruction

CF: Prepare rs = \$r5 for shift right logical 8 (clear), no forwarding (first sel_i = 0, second sel_i = 0)

EX: srl 1 \$r2, write to \$r2, now \$r2 holds 4-bit substring

16: FD: Fetch/decode addi 0, \$r3 instruction

CF: Prepare rs = \$3 for shift right logical 8 (clear), no forwarding (first sel i = 0, second sel i = 0)

EX: clear \$r5, write to \$r5, \$r5 = 0 (matched counter)

17: FD: Fetch/decode movh \$r8, \$r7 instruction

CF: Prepare rs = \$r3 for \$r3++, forward \$r3 from EX (first sel_i = 1)

EX: clear \$r3, write to \$r3, \$r3 = 0

18: FD: Fetch/decode addi 1, \$r7 instruction

CF: Prepare rs = \$r8 for move, no forwarding (first sel_i = 0)

```
EX: \$r3++, write to \$r3, \$r3 = 1
```

19: FD: Fetch/decode sll 6, \$r3 instruction

CF: Prepare rs = \$r7 for \$r7--, forward \$r7 from EX (first sel_i = 1)

EX: move \$r8 value to \$r7, \$r7 = \$r8

20: FD: Fetch/decode movl \$r3, \$r8 instruction

CF: Prepare rs = \$r3 for shift left logical 6, no forwarding (first sel_i = 0)

EX: \$r7--, write to \$r7, \$r7 = 5

21: FD: Fetch/decode srl 1, \$r8 instruction

CF: Prepare rs = \$r3 for move, forward \$r3 from EX (first sel i = 1)

EX: sll \$r3, write to \$r3, \$r3 = 64 (strings checked counter)

22: FD: Fetch/decode srl 8, \$r1 instruction

CF: Prepare rs = \$r8 for shift right logical 1, forward \$r8 from EX (first sel_i = 1)

EX: move \$r3 value to \$r8, write \$r8, \$r8 = \$r3

23: FD: Fetch/decode bz \$r3 instruction (this is the outer loop which iterates per string we need to check)

CF: Prepare rs = \$r1 for shift right logical 8 (clear), no forwarding (first sel i = 0)

EX: srl 1 \$r8, write to \$r8, \$r8 = 32 (base address of the string array)

/pipeline_testbench/cycle_counter	2000	24	(25	5	26		27		28		29) 30		31		32		33		34	Į:	35	
/pipeline_testbench/instr_counter	2000	(24	25	5	26		27	X	28		29		30		31		32		33		34	χ:	35	
/pipeline_testbench/FDpc	125	106	(10)7	109		110	1	111		113		107		109		110		111		113	χ.	107	
/pipeline_testbench/FDinstruction	00000000	100000	11 (01	1010111	111000	11	100100	00	0110001	11	110001	10	010101	11	111000	11	100100	000	011000	11	110001	10)(0101011	11
/pipeline_testbench/CFinstruction	00000000	101	10000011	01010	111	111000	11	1001000	00	011000	11	110001	10	010101	11	111000	11	100100	000	011000	11	11000110) (010
/pipeline_testbench/EXinstruction	00000000	111	10100010	100000	11	010101	11	111000	11	1001000	00	011000	11	110001	10	010101	11	111000	011	100100	000	0110001		110
/pipeline_testbench/b5v_inst/r1	0		(0									1										2		
/pipeline_testbench/b5v_inst/r2	5	5																					=	
/pipeline_testbench/b5v_inst/r3	0	64																						
/pipeline_testbench/b5v_inst/r4	0					18)	9										(4				
/pipeline_testbench/b5v_inst/r5	14	0																						
/pipeline_testbench/b5v_inst/r6	0	0																						
/pipeline_testbench/b5v_inst/r7	5	5																						
/pipeline_testbench/b5v_inst/r8	7	32																						
/pipeline_testbench/CFrs	1	3	4	(2		4		1						2		4		1						2
/pipeline_testbench/CFrt	0	0	8	14		(o)	7		(0		4		0				7		0	X	4
/pipeline_testbench/EXrd	0	1	(0	4		ļο		4		1		X O						4		1		0	=	
/pipeline_testbench/b3v_inst/result_o	0	0	64	o		23		9)	1		(6		1		14		(4		(2		7		2
/pipeline_testbench/b14v_inst/sel_i	0	0							,	1		(0								1		0		
/pipeline_testbench/b14v_inst/first_i	0	64	}			18		X 0				1		,5		9		1				2		5
/pipeline_testbench/b14v_inst/second_i	0	0	64	18		23		9	<u> </u>	1		6		1		14		4		(2		7		2
/pipeline_testbench/b14v_inst/third_i	0	0																1		X o				
/pipeline_testbench/b20v_inst/sel_i	0	0		1		ļο																		
/pipeline_testbench/b20v_inst/first_i	0	0	32	\Longrightarrow —		(0)	5		(0		9		0				15		0	X	4
/pipeline_testbench/b20v_inst/second_i	0	0	64	18		23		9)	1		6		1		14		(4		(2		7		2
/pipeline_testbench/b20v_inst/third_i	0	0																(1		(0				
/pipeline_testbench/b10v_inst/ram[7]	14																	_					\rightarrow	
Now	20000 ps	1111		26	liiii Ops		Lili	280		1111			liiii Ops	11111	Liii		liiii Dos	11111	Tirri		1 10 ps			36

(THIS DIAGRAM SHOWS STRING CHECKING IN WHICH THE STRING DOES NOT MATCH)

24: FD: Fetch/decode ld \$r4 instruction

CF: Prepare rs = \$r3, check if equal to 0, not taken (have not checked all 64 strings yet), branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction

EX: clear \$r1, write to \$r1, \$r1 = 0 (counter for checking if we have performed 4 shift left logicals; if so, we have finished checking this string for matching)

25: FD: Fetch/decode be 0, \$r2, \$r4 instruction (inner loop which iterates per bit to check if the 4-bit substring matches)

CF: Prepare rs = \$r4, rt = \$r8 for load, no forwarding (first sel_i = 0, second sel_i = 0)

EX: bz \$r3 does nothing in EX stage

26: FD: Fetch/decode srl 1, \$r4 instruction

CF: Prepare rs = r^2 , rt = r^4 , check if they are equal in lower 4-bits, not taken, branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, forward r^4 from EX (first sel_i = 0, second sel_i = 1)

EX: load into \$r4, write \$r4 (8-bit string that we are checking)

27: FD: Fetch/decode addi 0, \$r1 instruction

CF: Prepare rs = \$r4 for shift right logical 1, no forwarding (first sel i = 0, second sel i = 0)

EX: be 0, \$r2, \$r4 does nothing in EX stage

28: FD: Fetch/decode bge \$r1, \$r7 instruction

CF: Prepare rs = \$r1 for \$r1++, no forwarding

EX: srl 1 \$r4, write \$r4 (shift bits to right once so we can check the lower 4-bits for matching again)

29: FD: Fetch/decode jump loop_next_bit (PC = PC + (-6))

CF: Prepare rs= \$r1, rt = \$r7, check if greater than or equal, not taken (we have not shifted the string four times yet), branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, forward \$r1 from EX (first sel_i = 1, second sel_i = 0)

EX: \$r1++, write to \$r1, \$r1 = 1 (we have shifted the string once so far)

30: FD: Fetch/decode be 0, \$r2, \$r4 instruction (inner loop which iterates per bit to check if the 4-bit substring matches)

CF: jump logic, jump resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding (first sel_i = 0, second sel i = 0)

EX: bge \$r1, \$r7 does nothing in EX stage

31: FD: Fetch/decode srl 1, \$r4 instruction

CF: Prepare rs = \$r2, rt = \$r4, check if they are equal, not taken (lower 4 bits do not match our 4-bit substring), branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding (first sel i = 0, second sel i = 0)

EX: jump does nothing in EX stage

32: FD: Fetch/decode addi 0, \$r1 instruction

CF: Prepare rs = \$r4 for shift right logical 1, no forwarding (first sel_i = 0, second sel_i = 0)

EX: be 0, \$r2, \$r4 does nothing in EX stage

33: FD: Fetch/decode bge \$r1, \$r7 instruction

CF: Prepare rs = \$r1 for \$r1++, no forwarding

EX: srl 1 \$r4, write \$r4 (shift bits to right once so we can check the lower 4-bits for matching again)

34: FD: Fetch/decode jump loop_next_bit (PC = PC + (-6))

CF: Prepare rs= \$r1, rt = \$r7, check if greater than or equal, not taken (we have not shifted the string four times yet), branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, forward \$r1 from EX (first sel_i = 1, second sel_i = 0)

EX: \$r1++, write to \$r1, \$r1 = 1 (we have shifted the string once so far)

35: FD: Fetch/decode be 0, \$r2, \$r4 instruction (inner loop which iterates per bit to check if the 4-bit substring matches)

CF: jump logic, jump resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding (first sel_i = 0, second sel i = 0)

EX: bge \$r1, \$r7 does nothing in EX stage

(This loop continues two more times until we have checked all necessary bits and it finds out that the string doesn't match)

(THE FOLLOWING DIAGRAM SHOWS LOADING IN THE NEXT STRING TO CONTINUE CHECKING)

/pipeline_testbench/cycle_counter	2000	(48	49	(5	50	51	52	2	53	X	54	55	(56		57		58	X	59	
/pipeline_testbench/instr_counter	2000	48	(49	(5	50	51	52		53	X	54	55	, 56		57		58	X	59	
/pipeline_testbench/FDpc	125	[111	112	(1	118	119	12	10	103	X	104	106	107		109		110	X	111	
/pipeline_testbench/FDinstruction	00000000	01100011	1 (110010	000 (1	10010111	100110	10 11	.000001	111010	00 X	10100010	100000	11 (010)	0111	111000	011	1001000	00	01100011	
/pipeline_testbench/CFinstruction	00000000	100	01100011	11001000	100	010111	10011010	11000	001	1110100	0 101	00010	10000011	01010	111	111000	11	1001000	0 01	11
/pipeline_testbench/EXinstruction	00000000	111 1	100 10000	01100011	1 (110	001000	10010111	10011	010	1100000	1 111	01000	10100010	10000	011	010101	11	1110001	1 10	00
/pipeline_testbench/b5v_inst/r1	0	4		15									X ₀							
/pipeline_testbench/b5v_inst/r2	5	5																		
/pipeline_testbench/b5v_inst/r3	0	64								63										
/pipeline_testbench/b5v_inst/r4	0	1 (0	0													52			26	6
/pipeline_testbench/b5v_inst/r5	14	0																		
/pipeline_testbench/b5v_inst/r6	0	0																		_
/pipeline_testbench/b5v_inst/r7	5	5																		
/pipeline_testbench/b5v_inst/r8	7	32						33												
/pipeline_testbench/CFrs	1	1			8		3	1			3		14	2		4		1		
/pipeline_testbench/CFrt	0	0 (7	7	X ₀									(8	4		(o			. 7	_
/pipeline_testbench/EXrd	0	4 1	1	X o			8	(3		X o	1		(O	4		(o		4	1	
/pipeline_testbench/b3v_inst/result_o	0	0 (5	5	10	, 5		33	63		<u>, 5</u>	(0		63	33		57		26	1	
/pipeline_testbench/b14v_inst/sel_i	0	0 1	1	(0															1	
/pipeline_testbench/b14v_inst/first_i	0	4		15	32		64	,5			63		X ₀	, 5		52		0		
/pipeline_testbench/b14v_inst/second_i	0	0 (5	5	10	, 5		33	63		<u>, 5</u>	(0		63	52		57		26	1	
/pipeline_testbench/b14v_inst/third_i	0	1 (0)																	
/pipeline_testbench/b20v_inst/sel_i	0	0												1		(o				
/pipeline_testbench/b20v_inst/first_i	0	0 (5	5	X ₀									33	(O		(o			.5	
/pipeline_testbench/b20v_inst/second_i	0	0 (5	5	10	, 5		33	63		15	(0		63	52		57		26	1	
/pipeline_testbench/b20v_inst/third_i	0	1 (0	0																	_
/pipeline_testbench/b10v_inst/ram[7]	14																			
Now	20000 ps	11111		500 p		uluu	520 ps		İ	540		nim	560 ps	111111	İmm		0 ps			600

49: FD: Fetch/decode jump end string instruction

CF: Prepare rs = \$r1, rt = \$r7 for branch greater than or equal, branch taken, branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, forward \$r1 (first sel_i = 1, second sel_i = 0)

EX: \$r1++, write to \$r1, \$r1 = 5

50: FD: Fetch/decode addi 0, \$r8 instruction

CF: jump logic, jump resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding

EX: bge \$r1, \$r7 does nothing in EX stage

51: FD: Fetch/decode addi 1, \$r3 instruction

CF: Prepare rs = \$r8 for \$r8++, no forwarding

EX: jump does nothing in EX stage

52: FD: Fetch/decode jump loop_next_string instruction

CF: Prepare rs = \$r3 for \$r3--, no forwarding

EX: \$r8++, write to \$r8, \$r8 =33

53: FD: Fetch/decode srl 8, \$r1 instruction

CF: jump logic, jump resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding

EX: \$r3--, write to \$r3, \$r3 = 63

54: FD: Fetch/decode bz \$r3 instruction

CF: Prepare rs = \$r1 for shift right logical 8 (clear), no forwarding

EX: jump does nothing in EX stage

55: FD: Fetch/decode ld \$r4 instruction

CF: Prepare rs = \$r3 for branch equals 0, not taken, branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding

EX: clear \$r1, write to \$r1, \$r1 = 0

56: FD: Fetch/decode be 0, \$r2, \$r4 instruction

CF: Prepare rs = \$r4, rt = \$r8 for load, no forwarding

EX: be \$r3 does nothing in EX stage

57: FD: Fetch/decode srl 1, \$r4 instruction

CF: Prepare rs = \$r2, rt = \$r4, check if they are equal in lower 4-bits, not taken, branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, forward \$r4 from EX (first sel i = 0, second sel i = 1)

EX: load to \$r4, write to \$r4 (next string to be checked if matching with the 4-bit substring)

(This main loop continues on...)

(THE FOLLOWING DIAGRAM SHOWS WHAT HAPPENS WHEN A STRING MATCHES)

/pipeline_testbench/cycle_counter	2000	95	, X	96	97		98	<u> </u>	99	X	100		101		102	;	103		104		105	χ	106
/pipeline_testbench/instr_counter	2000	(95	X	96	97		98		99	X	100		101		102	;	103		104		105	X	106
/pipeline_testbench/FDpc	125	111	X	113	107		108		114	X	115		116		117		103		104		106	X	107
/pipeline_testbench/FDinstruction	00000000	011000	11 (11000110	010101	11	1100100	0	1001010	0 (1001101	10	100101	11	110000	11	111010	00	101000	10	1000001	1)	01010111
/pipeline_testbench/CFinstruction	00000000	10010000	0110001	1 110	00110	010101	11)	1100100	00)	1001010	0)	100110	10	100101	11	110000	11	111010	00	101000	10)	1000001	1
/pipeline_testbench/EXinstruction	00000000	11100011	1001000	0 (011	00011	110001	10 (010101	11 (1100100	0)	100101	00	100110	10	100101	11	110000	11	1110100	00 (1010001	0
/pipeline_testbench/b5v_inst/r1	0	1		(2																	X	0	
/pipeline_testbench/b5v_inst/r2	5	5																					
/pipeline_testbench/b5v_inst/r3	0	62														61							
/pipeline_testbench/b5v_inst/r4	0	43	21																				
/pipeline_testbench/b5v_inst/r5	14	0												1									
/pipeline_testbench/b5v_inst/r6	0	0																					
/pipeline_testbench/b5v_inst/r7	5	5																					
/pipeline_testbench/b5v_inst/r8	7	34																35					
/pipeline_testbench/CFrs	1	1				2	X	1	X	5		3		8		1				3	X	4	2
/pipeline_testbench/CFrt	0	0	7	χo		4	X	0													X	8	4
/pipeline_testbench/EXrd	0	4	1	(0							, X	5		3		8		0		1	X	0	4
/pipeline_testbench/b3v_inst/result_o	0	21	2	.7		2	X	26	X	2	X	1		61		35		2		0	X	61	56
/pipeline_testbench/b14v_inst/sel_i	0	0	1	0																			
/pipeline_testbench/b14v_inst/first_i	0	1		12		15	X	2	X	0	,	62		34		2				61	X	21	(5
/pipeline_testbench/b14v_inst/second_i	0	21	2	17		(2	X	26	X	2	X	1		61		35		2		(0	X	61	
/pipeline_testbench/b14v_inst/third_i	0	1	0																				
/pipeline_testbench/b20v_inst/sel_i	0	0																					1
/pipeline_testbench/b20v_inst/first_i	0	0	(5	χo		21	X	0													X	35	21
/pipeline_testbench/b20v_inst/second_i	0	21	2	17		2	X	26	X	2	Ľ	1		61		35		2		X o	X	61	
/pipeline_testbench/b20v_inst/third_i	0	1	(0																				
/pipeline_testbench/b10v_inst/ram[7]	14																						
Now	20000 ps	111111	960	ns		980		1111	11111	1000		1111		102		1111	1		liiii Ops		1	1060	

96: FD: Fetch/decode jump loop next bit instruction

CF: Prepare rs = \$r1, rt = \$r7 for branch greater than or equal, not taken, branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, forward \$r1 from EX (first sel_i = 1, second sel_i = 0)

EX: \$r1++, write to \$r1, \$r1 = 2

97: FD: Fetch/decode be 0, \$r2, \$r4 instruction

CF: jump logic, jump resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding (first sel_i = 0, second sel_i = 0)

EX: bge \$r1, \$r7 does nothing in EX stage

98: FD: Fetch/decode jump equal string instruction

CF: Prepare rs = \$r2, rt = \$r4 for branch lower 4-bit equals, branch taken, branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding (first sel_i = 0, second sel_i = 0)

EX: jump loop next bit does nothing in EX stage

99: FD: Fetch/decode addi 0, \$r5 instruction

CF: jump logic, jump resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding (first sel_i = 0, second sel_i = 0)

EX: be 0, \$r2, \$r4 does nothing in EX

100: FD: Fetch/decode addi 1, \$r3 instruction

```
CF: Prepare rs = $r5 for $r5++, no forwarding (first sel_i = 0, second sel_i = 0)
```

EX: jump equal_string does nothing in EX

101: FD: Fetch/decode addi 0, \$r8 instruction

CF: Prepare rs = \$r3 for \$r3--, no forwarding (first sel_i = 0, second sel_i = 0)

EX: \$r5++, write to \$r5, \$r5 = 1 (matched_counter)

102: FD: Fetch/decode jump loop next string instruction

CF: Prepare rs = \$r8 for \$r8++, no forwarding (first sel i = 0, second sel i = 0)

EX: \$r3--, write to \$r3, \$r3 = 63 (strings_checked_counter)

103: FD: Fetch/decode srl 8, \$r1 instruction

CF: jump logic, jump resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding (first sel_i = 0, second sel_i = 0)

EX: \$r8++, write to \$r8, \$r8 = 35 (address of next string to check)

(Main loop continues on until finish checking all strings.)

(THE FOLLOWING DIAGRAM SHOWS STORING THE RESULT)

/pipeline_testbench/cycle_counter	2000	1796 179		1798		1799		1800		1801		1802		1803		1804		1805		1806		1807	\Rightarrow	1808
/pipeline_testbench/instr_counter	2000	1796 179	7	1798		1799		1800		1801		1802		1803		1804		1805		1806		1807		1808
/pipeline_testbench/FDpc	125	119 120		103		104		105		121		122		123		124		125						
/pipeline_testbench/FDinstruction	00000000	100 110	00001	111010	00	101000	10	110010	01	010010	11	100101	11			011101	00	000000	00				=	
/pipeline_testbench/CFinstruction	00000000	(10011010	11000	0001	111010	00	101000	10	110010	01	010010	1	100101	11			011101	100	000000	000			\equiv	
/pipeline_testbench/EXinstruction	00000000	(10010111	10011	1010	110000	01	111010	00	101000	10	1100100)1	010010	11	100101	11			01110	100	000000	00	=	
/pipeline_testbench/b5v_inst/r1	0	5							0															_
/pipeline_testbench/b5v_inst/r2	5	5																						_
/pipeline_testbench/b5v_inst/r3	0	1			0																		\equiv	
/pipeline_testbench/b5v_inst/r4	0	0																						
/pipeline_testbench/b5v_inst/r5	14	14																						
/pipeline_testbench/b5v_inst/r6	0	0																						
/pipeline_testbench/b5v_inst/r7	5	5																						
/pipeline_testbench/b5v_inst/r8	7	95	96												5		(6		7					
/pipeline_testbench/CFrs	1	(3	1				3		1		7		8				5		1					
/pipeline_testbench/CFrt	0	0									8		0				8		(o					
/pipeline_testbench/EXrd	0	(8	(3		0		1		0				8						9		(0			
/pipeline_testbench/b3v_inst/result_o	0	(96	(o		5		0						101		6		7		21		(0			
/pipeline_testbench/b14v_inst/sel_i	0	0											1				(o							
/pipeline_testbench/b14v_inst/first_i	0	(1	5				0				5		96		5		14		(o					
/pipeline_testbench/b14v_inst/second_i	0	(96	(o		5		0						5		6		7		21		(0			
/pipeline_testbench/b14v_inst/third_i	0	0																						
/pipeline_testbench/b20v_inst/sel_i	0	0															1		(o					
/pipeline_testbench/b20v_inst/first_i	0	0									96		0				6		(o					
/pipeline_testbench/b20v_inst/second_i	0	(96	(o		5		0						5		6		7		21		(0			
/pipeline_testbench/b20v_inst/third_i	0	0																						
/pipeline_testbench/b10v_inst/ram[7]	14																			14				
Now	20000 ps		17	980 ps		Litte	1800			Linn	1802			Litt		1 40 ps		Litte		1 60 ps		Lini	18080	

1800: FD: Fetch/decode jump finish 2 instruction

CF: Prepare rs = \$r3 for branch equal to 0, branch taken, branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding (first sel_i = 0, second sel_i = 0)

EX: clear \$r1, write to \$r1

1801: FD: Fetch/decode movh \$r7, \$r8 instruction

CF: jump logic, jump resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding (first sel_i = 0, second sel i = 0)

EX: bz \$r3 does nothing in EX stage

1802: FD: Fetch/decode addi 0, \$r8 instruction

CF: Prepare rs = \$r7 for move, no forwarding (first sel i = 0, second sel i = 0)

EX: jump finish 2 does nothing in EX stage

1803: FD: Fetch/decode addi 0, \$r8 instruction

CF: Prepare rs = \$r8 for \$r8++, forward \$8 from EX (first sel_i = 1, second sel_i = 0)

EX: move \$r7 value to \$r8, \$r8 = \$r7

1804: FD: Fetch/decode store \$r5 instruction

CF: Prepare rs = \$r8 for \$r8++, forward \$8 from EX (first sel i = 1, second sel i = 0)

EX: \$r8++, write to \$r8, \$r8 = 6

1805: FD: Fetch/decode halt instruction

CF: Prepare rs = \$r5, rt = \$r8 for store, forward \$r8 from EX (first sel_i = 0, second sel_i = 1)

EX: \$r8++, write to \$r8, \$r8 = 7

1806: FD: Fetch/decode halt instruction

CF: halt

EX: store \$r5 to address of \$r8, datamem[7] = \$r5

(Halt instructions continue to execute through the pipeline.)

Program 3 (Finds the least distances between all pairs of values in an array of 20 bytes): (NOTE: The listed numbers are the cycle_counter values to help guide you).

The first set of sel_i, first_i, second_i, third_i signals refer to the forwarding selector for rs. The second set refers to the fowarding selector for rt.

When sel_i = 0, do not foward data from EX (use first_i value). If sel_i = 1, forward data from EX (use second_i value). If sel_i=2, forward carry out from EX (use third_i value). sel_i values should be checked on the rising edge (that's when reads should happen).

(THE FOLLOWING DIAGRAMS SHOW THE INITIALIZATION OF PROGRAM 3)

pipeline_testbench/cycle_counter	2200			1	X	2	(3		4		5		6		7		8		9		10	\equiv
ipeline_testbench/instr_counter	2200	(0	X	1	X	2	(3		4		5		6		7		8		9		10	
pipeline_testbench/FDpc	176				(126	127		128		129		130		131		132		133		134	
pipeline_testbench/FDinstruction	00000000	(000000	00		X	11101111	1001	0111	111101	11	111100	11	1001111	1	010011	01	100100	10			111100	01
pipeline_testbench/CFinstruction	00000000		(0000000	00		1	1101111	100101	11	111101	11	111100	11)	100111	11	010011	01	100100	10			111
oipeline_testbench/EXinstruction	00000000				0000000	00		111011	11	100101	11	111101	11)	111100	11	100111	11	010011	01	100100	10	
oipeline_testbench/b5v_inst/r1	0																					+-
oipeline_testbench/b5v_inst/r2	0																					+
pipeline_testbench/b5v_inst/r3	0																			7		(8
ipeline_testbench/b5v_inst/r4	0																					+-
oipeline_testbench/b5v_inst/r5	-47																					+-
oipeline_testbench/b5v_inst/r6	112																					+-
pipeline_testbench/b5v_inst/r7	2																					+
ipeline_testbench/b5v_inst/r8	127									0		1 1	ĭ	4		18		17				-
ipeline_testbench/CFrs	1		1			18												13				F
ipeline_testbench/CFrt	0		0													(3		X _O				-
ipeline_testbench/EXrd	0				0			(8										13				
ipeline_testbench/b3v_inst/result_o	0	(0						-₹0		1		4	, and	8		7		 		{8		9
ipeline_testbench/b14v_inst/sel_i	0	(0						1														\equiv
oipeline_testbench/b14v_inst/first_i	0									(0		1	X	4		8				7		8
ipeline_testbench/b14v_inst/second_i	0	(0						√ 0		1		4	,	8		7				8		9
ipeline_testbench/b14v_inst/third_i	0	(0																				=
ipeline_testbench/b20v_inst/sel_i	0	(0																				\equiv
ipeline_testbench/b20v_inst/first_i	0									(0								{0				F
ipeline_testbench/b20v_inst/second_i	0	(0	-					√ 0		1		4	ĭ	8		7				18		19
ipeline_testbench/b20v_inst/third_i	0	(0																				F
ipeline_testbench/b10v_inst/ram[127]	2																					F
Now	22000 ps	DS DS			20				liiii ps			60					ps			100		1 1

2: FD: Fetch/decode srl 8, \$r8 instruction

CF: Nothing EX: Nothing

3: FD: Fetch/decode addi 0, \$r8 instruction

CF: Prepare rs = \$r8 for shift right logical 8 (clear)

EX: Nothing

4: FD: Fetch/decode sll 4, \$r8 instruction

CF: Prepare rs = \$r8 for \$r8++, forward \$r8 from EX (first sel_i = 1, second sel_i = 0)

EX: Clear \$r8, write to \$r8, \$r8 = 0

```
5: FD: Fetch/decode sll 1, $r8 instruction
  CF: Prepare rs = $r8 for shift left logical 4, forward $r8 from EX (first sel i = 1, second sel i = 0)
  EX: \$r8++, write to \$r8, \$r8 = 1
6: FD: Fetch/decode addi 1, $r8 instruction
  CF: Prepare rs = $r8 for shift left logical 1 forward $r8 from EX (first sel i = 1, second sel i = 0)
  EX: sll 4, $r8, write to $r8, $r8 = 4
7: FD: Fetch/decode movh $r8, $r3 instruction
  CF: Prepare rs = \$r8 for \$r8--, forward \$r8 from EX (first sel i = 1, second sel i = 0)
  EX: sll 1, $r8, write to $r8, $r8 = 8
8: FD: Fetch/decode addi 0, $r3 instruction
  CF: Prepare rs = \$r8 for move, forward \$r8 from EX (first sel i = 1, second sel i = 0)
  EX: \$r8--, write to \$r8, \$r8 = 7
9: FD: Fetch/decode addi 0, $r3 instruction
  CF: Prepare rs = r3 for r3++, forward r3 from EX (first sel i = 1, second sel i = 0)
  EX: move $r8 value to $r3. $r3 = $r8
10: FD: Fetch/decode sll 1, $r3 instruction
  CF: Prepare rs = r3 for r3++, forward r3 from EX (first sel i = 1, second sel i = 0)
EX: \$r3++, write to \$r3, \$r3 = 8
11: FD: Fetch/decode addi 0, $r3 instruction (CHECK NEXT PAGE FOR NEXT DIAGRAM)
  CF: Prepare rs = $r3 for shift left logical 1, forward $r3 from EX (first sel i = 1, second sel i = 0)
  EX: \$r3++, write to \$r3, \$r3 = 9
12: FD: Fetch/decode movl $r3, $r4 instruction
  CF: Prepare rs = \$r3 for \$r3++, forward \$r3 from EX (first sel i = 1, second sel i = 0)
  EX: sll 1, \$r3, write to \$r3, \$r3 = 18
13: FD: Fetch/decode srl 8, $r1 instruction
  CF: Prepare rs = \$r3 for move, forward \$r3 from EX (first sel i = 1, second sel i = 0)
  EX: \$r8++, write to \$r8, \$r8 = 19
14: FD: Fetch/decode srl 8, $r2 instruction
  CF: Prepare rs = $r1 for shift right logical 8 (clear), no forwarding (first sel i = 0, second sel i = 0)
```

EX: move \$r3 to \$r4, \$r4 = \$r3

/pipeline_testbench/cycle_counter	2200	(11	X	12	13		14	Ĭ.	15	X	16	(1	17	X	18	19		20		21	1 2	22
/pipeline_testbench/instr_counter	2200	(11	X	12	13		14	χ.	15	X	16	(1	17	X	18	19		20		21	1 2	22
/pipeline_testbench/FDpc	176	135	X	136	137		138	χ.	139	X	140	(1	141	X	142	143		144		145	1 1	146
/pipeline_testbench/FDinstruction	00000000	100100	10)	00111001	111010	000	111010	01)	1110111	11	100101	11		X	111111	11 (100)	00100	10010	111	10000	101 1	10010001
/pipeline_testbench/CFinstruction	00000000	111	1001001	0 (0011	1001	111010	00	1110100	1 (1110111	1	10010111	L			11111111	10000	100	100101	11	10000101	100
/pipeline_testbench/EXinstruction	00000000	100	1111000	1 1001	0010	001110	01	11101000) (1110100	1	11101111	L)	100101	11		11111	111	100001	.00	10010111	100
/pipeline_testbench/b5v_inst/r1	0								 (0												
/pipeline_testbench/b5v_inst/r2	0				_							0										
/pipeline_testbench/b5v_inst/r3	0	8	9	18		19																
/pipeline_testbench/b5v_inst/r4	0							19														
/pipeline_testbench/b5v_inst/r5	-47																				2	
/pipeline_testbench/b5v_inst/r6	112																					
/pipeline_testbench/b5v_inst/r7	2																					
/pipeline_testbench/b5v_inst/r8	127	7											X	0		1	2		128			129
/pipeline_testbench/CFrs	1	3				1		2	X	8							5		(8		(6	(2
/pipeline_testbench/CFrt	0	0		(4		(o											8		ļο		(8	χo
/pipeline_testbench/EXrd	0	3				4		1	X	2		8							(5		(8	6
/pipeline_testbench/b3v_inst/result_o	0	9	18	19				0					X	1		2	128				129	\rightarrow
/pipeline_testbench/b14v_inst/sel_i	0	1				(o						1					(o					
/pipeline_testbench/b14v_inst/first_i	0	8	<u> </u>	18					 (7			X	0		1	\supset —		128			(o
/pipeline_testbench/b14v_inst/second_i	0	9	18	19				0					Ż	1		2	128		(2		129	233
/pipeline_testbench/b14v_inst/third_i	0	0																				
/pipeline_testbench/b20v_inst/sel_i	0	0															1		χo.		1	χo
/pipeline_testbench/b20v_inst/first_i	0	0		\longrightarrow		-{0											2		χo.		128	χo
/pipeline_testbench/b20v_inst/second_i	0	9	18	19				0					X	1		2	128		2		129	233
/pipeline_testbench/b20v_inst/third_i	0	0																				
/pipeline_testbench/b10v_inst/ram[127]	2																					
New	22000 ps	1111			dan 														11111	İ		
Now	22000 ps		120	ps		14	0 ps			160	ps			180	ps		20	00 ps			220 p	ıs

15: FD: Fetch/decode srl 8, \$r8 instruction

CF: Prepare rs = \$r2 for shift right logical 8 (clear), no forwarding (first sel_i = 0, second sel_i = 0)

EX: Clear \$r1, write to \$r1, \$r1 = 0 (i_counter for outer loop)

16: FD: Fetch/decode addi 0, \$r8 instruction

CF: Prepare rs = \$r8 for shift right logical 8 (clear), no forwarding (first sel_i = 0, second sel_i = 0)

EX: Clear \$r2, write to \$r2, \$r2 = 0 (j_counter for inner loop)

17: FD: Fetch/decode addi 0, \$r8 instruction

CF: Prepare rs = \$r8 for \$r8++, forward \$r8 from EX (first sel_i = 1, second sel_i = 0)

EX: Clear \$r8, write to \$r8, \$r8 =0

18: FD: Fetch/decode sll 6, \$r8 instruction

CF: Prepare rs = \$r8 for \$r8++, forward \$r8 from EX (first sel_i = 1, second sel_i = 0)

EX: \$r8++, write to \$r8, \$r8 = 1

19: FD: Fetch/decode ld \$r5 instruction

CF: Prepare rs = \$r8 for shift left logical 6, forward \$r8 from EX (first sel_i = 1, second sel_i = 0)

EX: \$r8++, write to \$r8, \$r8 = 2

20: FD: Fetch/decode addi 0, \$r8 instruction

CF: Prepare rs = \$r5, rt = \$r8, forward \$r8 from EX (first sel_i = 0, second sel_i = 1)

EX: sll 6, \$r8, write to \$r8, \$r8 = 128 (memory address of int array)

21: FD: Fetch/decode ld \$r6 instruction

CF: Prepare rs = \$r8 for \$r8++, no forwarding (first sel_i = 0, second sel_i = 0)

EX: load \$r5 with data from memory address \$r8, write to \$r5 (first integer to compare distance with)

22: FD: Fetch/decode addi 0, \$r2 instruction

CF: Prepare rs = \$r6, rt = \$r8 for load, forward \$r8 from EX (first sel_i = 0, second sel_i = 1)

EX: \$r8++, write to \$r8, \$r8 = 129

/pipeline_testbench/cycle_counter	2200	(23	. 24	4	25	26	27	(28		29	130	31	32	(33	(34	\bot
/pipeline_testbench/instr_counter	2200	23	124	4	25	26	27	28		29	(30	31	32	33	34	
/pipeline_testbench/FDpc	176	147	114	48	149	150	151	152	2	153	154	156	157	159	150	
/pipeline_testbench/FDinstruction	00000000	(001010	010 (0:	1000110	100100	00 (10010	111 (1000	0101 (001	01010	011011	11 (110001	11 (10010	0001 0101	1110 (11000	100 (1001)	3111
/pipeline_testbench/CFinstruction	00000000	100	00101010	01000	110	10010000	10010111	10000101	001010	10	01101111	11000111	10010001	01011110	11000100	100
/pipeline_testbench/EXinstruction	00000000	100	10010001	00101	010	01000110	10010000	10010111	100001	01	00101010	01101111	11000111	10010001	01011110	110
/pipeline_testbench/b5v_inst/r1	0	0						1 1								
/pipeline_testbench/b5v_inst/r2	0	0		(1											(2	
/pipeline_testbench/b5v_inst/r3	0	19														
/pipeline_testbench/b5v_inst/r4	0	19														
/pipeline_testbench/b5v_inst/r5	-47	2														
/pipeline_testbench/b5v_inst/r6	112		-23			25					.33	(35				
/pipeline_testbench/b5v_inst/r7	2						25									
/pipeline_testbench/b5v_inst/r8	127	129							130							
pipeline_testbench/CFrs	1	2	(6			1	(8	(6				11	(2		1 1	(8
pipeline_testbench/CFrt	0	0	X 5	7		X O		(8	<u> </u>		7	χo		(3	X _O	
/pipeline_testbench/EXrd	0	6	12	(6		17) 1	(8	16			χo		(2	χo	\pm
/pipeline_testbench/b3v_inst/result_o	0		1	25		}	1	130	155		35	(60	1	12	21	1
/pipeline_testbench/b14v_inst/sel_i	0	0		1 1		X O			1			χo		1	χo	
/pipeline_testbench/b14v_inst/first_i	0	0	233			X O	129	25			223	X 1			(1	130
/pipeline_testbench/b14v_inst/second_i	0	233	X 1	25			1	(130	223		35	(60	1	(2	21	1
pipeline_testbench/b14v_inst/third_i	0	0														
/pipeline_testbench/b20v_inst/sel_i	0	0						1	χo							
pipeline_testbench/b20v_inst/first_i	0	0	(2	\longrightarrow —		0		129	(2		25	χo		19	X _O	
/pipeline_testbench/b20v_inst/second_i	0	233	X 1	25			1	130	223		35	(60	1	(2	21	1
pipeline_testbench/b20v_inst/third_i	0	0														
/pipeline_testbench/b10v_inst/ram[127]	2															
Now	22000 ps		240 ps		İrri	260 ps	immino	280 ps		İrrir	300 ps	innin	320 ps	rice de la constancia d	340 ps	Time

(THIS DIAGRAM SHOWS WHAT HAPPENS WHEN DIFFERENCE IS NOT THE NEW MINIMUM DISTANCE)

23: FD: Fetch/decode diff \$r6, \$r5 instruction

EX: load \$r6 with data from memory address \$r8, write to \$r6 (second integer to compare distance with)

24: FD: Fetch/decode movh \$r6, \$r7 instruction

CF: Prepare rs = \$r5, rt = \$r6 for diff, no forwarding (first sel i = 0, second sel i = 0)

EX: r2++, write to r2, r2 = 1

25: FD: Fetch/decode addi 0, \$r1 instruction

CF: Prepare $rs = \frac{r}{r}$ for move, forward $\frac{r}{r}$ from EX(first sel i = 1, second sel i = 0)

EX: diff \$r6, \$r5, write to \$r6

26: FD: Fetch/decode addi 0, \$r8 instruction

CF: Prepare rs = \$r1 for \$r1++, no forwarding (first sel_i = 0, second sel_i = 0)

EX: move \$r6 value to \$r7 (minimum distance so far)

27: FD: Fetch/decode ld \$r6 instruction

CF: Prepare rs = \$r8 for \$r8++, no forwarding (first sel_i = 0, second sel_i = 0)

EX: \$r1++, write to \$r1, \$r1 = 1

28: FD: Fetch/decode diff \$r6, \$r5 instruction

CF: Prepare rs = \$r6, rt = \$r8, forward \$r8 from EX (first sel i = 0, second sel i = 1)

EX: \$r8++, write to \$r8, \$r8 = 130

29: FD: Fetch/decode bge \$r6, \$r7 instruction

CF: Prepare rs = \$r6, rt = \$r5 for diff (first sel i = 1, second sel i = 0)

EX: load \$r6 with data from memory address \$r8, write to \$r6 (next integer to compare distance)

30: FD: Fetch/decode jump j continue instruction

CF: Prepare rs = \$r6, rt = \$r7 for branch greater than or equal, branch taken (not a new minimum distance), branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, forward \$r6 from EX (first sel i = 1, second sel i = 0)

EX: diff \$r6, \$r5, write to \$r6 (compare and write the distance)

31: FD: Fetch/decode addi 0, \$r2 instruction

CF: jump logic, jump resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, no forwarding

EX: bge \$r6, \$r7 does nothing in EX stage

(The j loop and i loop continue iterating through the array to find new minimum distances...)

(THESE TWO FOLLOWING DIAGRAMS SHOWS WHAT HAPPENS WHEN A NEW MINIMUM DISTANCE IS FOUND)

/pipeline_testbench/cycle_counter	2200	35		36		37	X	38		39	X4	10		41		42	X	43		44		45	X	46	
/pipeline_testbench/instr_counter	2200	35		36		37		38		39) <u>-</u>	10		41		42	X	43		44		45		46	
/pipeline_testbench/FDpc	176	151		152		153	X	154		156	χ:	157		159		150	X	151		152		153		155	
/pipeline_testbench/FDinstruction	00000000	10000	101	001010	10	011011	11	110001	11	100100	01 (0	101111	0	1100010	00	100101	11 (100001	01	001010	10	011011	11 (010001	10
/pipeline_testbench/CFinstruction	00000000	100	100001	01	001010	10	0110111	1	110001	11	10010001		010111	10	1100010	00	1001011	1	1000010	01	001010	10	0110111	1	010
/pipeline_testbench/EXinstruction	00000000	110	100101	11	100001	01	0010101	.0	011011	11	11000111		1001000	01	010111	10	1100010	0	100101	11	100001	01	0010101	0	011
/pipeline_testbench/b5v_inst/r1	0	1																							
/pipeline_testbench/b5v_inst/r2	0	2													3										
/pipeline_testbench/b5v_inst/r3	0	19																						=	
/pipeline_testbench/b5v_inst/r4	0	19																							
/pipeline_testbench/b5v_inst/r5	-47	2																							
/pipeline_testbench/b5v_inst/r6	112	35					63		61														18		16
/pipeline_testbench/b5v_inst/r7	2	25																							
/pipeline_testbench/b5v_inst/r8	127	130			131																132				
/pipeline_testbench/CFrs	1	8	(6						1		2				1		(8		(6						
/pipeline_testbench/CFrt	0	0	(8		5		7		0			X	3		0				8		5		7		
/pipeline_testbench/EXrd	0	0	(8		6			1	0			X	2		0				8		6				0
/pipeline_testbench/b3v_inst/result_o	0	1	131		166		61		86		1		3		22		1		132		193		16		41
/pipeline_testbench/b14v_inst/sel_i	0	0			1				0				1		0						1				(O
/pipeline_testbench/b14v_inst/first_i	0	130	35				63		1		2				1		131		61				18		16
/pipeline_testbench/b14v_inst/second_i	0	1	131		63		61		86		1	X	3		22		1		132		18		16	=	41
/pipeline_testbench/b14v_inst/third_i	0	0																							
/pipeline_testbench/b20v_inst/sel_i	0	0	1		0														1		(O				
/pipeline_testbench/b20v_inst/first_i	0	0	130		2		25		0			X	19		0				131		2		25		
/pipeline_testbench/b20v_inst/second_i	0	1	131		63		61		86		1	X	3		22		1		132		18		16	=	41
/pipeline_testbench/b20v_inst/third_i	0	0																							
/pipeline_testbench/b10v_inst/ram[127]	2																								
				1		 														 	<u> </u>	1	1		<u> </u>
Now	22000 ps) ps			380				400 p				420				440				460		

44: FD: Fetch/decode diff \$r6, \$r5 instruction

CF: Prepare rs = \$r6, rt = \$r8 for load, forward \$r8 from EX (first sel_i = 0, second sel_i = 1)

EX: \$r8++, write to \$r8

45: FD: Fetch/decode bge \$r6, \$r7 instruction

CF: Prepare rs = \$r6, rt = \$r5 for diff, forward \$r6 from EX (first sel_i = 1, second sel_i = 0)

EX: load \$r6 with data from memory address \$r8, write to \$r6 (new integer to compare distance)

46: FD: Fetch/decode movh \$r6, \$r7 instruction

CF: Prepare rs = \$r6, rt = \$r7 for branch greater than or equal, branch not taken (it is a new minimum distance), branch resolved next pc value before program counter read on posedge, FD stage fetches correct instruction, forward \$r6 from EX (first sel_i = 1, second sel_i = 0)

EX: diff \$r6, \$r5, write to \$r6 (compare and write the distance)

/pipeline_testbench/cycle_counter	2200	47		48	49	(50		51	X	52	χ.	53	X	54		55	X	56		57		58	
/pipeline_testbench/instr_counter	2200	47		48	49	50		51	X	52	χ.	53	X	54		55	X	56		57	X	58	
/pipeline_testbench/FDpc	176	156		157	159	(15)	0	151	X	152	X	153	X	154		156	X	157		159	, and	150	
/pipeline_testbench/FDinstruction	00000000	(100100	001	01011110	110001	00 (10	010111	100001	01	0010101	0 (0110111	1	110001	11	1001000)1)	0101111	10	1100010	0)	100101	11
/pipeline_testbench/CFinstruction	00000000	010	1001000	010	11110	11000100	10010	111	1000010	1	0010101	0)	0110111	1	110001	11	1001000)1 (010111	10)	1100010	0	100
/pipeline_testbench/EXinstruction	00000000	011	010001	10 100	10001	01011110	11000	100	1001011	1	1000010	1	0010101	0	011011	11	110001	11 (1001000	01 X	0101111	0	110
/pipeline_testbench/b5v_inst/r1	0	1																					
/pipeline_testbench/b5v_inst/r2	0	3				(4														X	5		
/pipeline_testbench/b5v_inst/r3	0	19																					
/pipeline_testbench/b5v_inst/r4	0	19																					
/pipeline_testbench/b5v_inst/r5	-47	2																					
/pipeline_testbench/b5v_inst/r6	112	16										, and	23		21								
/pipeline_testbench/b5v_inst/r7	2	25		16																			
/pipeline_testbench/b5v_inst/r8	127	132								X	133												
/pipeline_testbench/CFrs	1	6	2			1	8		(6						1		2			X	1		8
/pipeline_testbench/CFrt	0	7	(0	(3		X ₀			(8	X	5	, i	7		(0			X	3	X	0		
/pipeline_testbench/EXrd	0	0	7	2		X 0			(8	χ.	6				(0			X	2	X	0		
/pipeline_testbench/b3v_inst/result_o	0	41		(4		23	1		133	X	149		21		37		1	X	5	X	24		1
/pipeline_testbench/b14v_inst/sel_i	0	0		1		X ₀				X	1				(0				1	X	0		
/pipeline_testbench/b14v_inst/first_i	0	16	(3			1	132		16			, and	23		1		4			X	1		133
/pipeline_testbench/b14v_inst/second_i	0	41	16	(4		23	1		133	X	23	, i	21		37		1	X	5	X	24		1
/pipeline_testbench/b14v_inst/third_i	0	0																					
/pipeline_testbench/b20v_inst/sel_i	0	0							1		0												
/pipeline_testbench/b20v_inst/first_i	0	25	(0	19		X ₀			132	X	2	, i	16	;	(0			X	19	X	0		
/pipeline_testbench/b20v_inst/second_i	0	41	16	(4		23	1		133	X	23	, i	21		37		1	X	5	X	24		1
/pipeline_testbench/b20v_inst/third_i	0	0																					
/pipeline_testbench/b10v_inst/ram[127]	2																						-
Now	22000 ps	1111	480	1111111		500 ps	11111		520				540				560		1111	1	1 1 1 1 I 580		

47: FD: Fetch/decode addi 0, \$r2 instruction

CF: Prepare rs = \$r6 for move, no forwarding (first sel_i = 0, second sel_i = 0)

EX: bge \$r6, \$r7 does nothing in EX stage.

48: FD: Fetch/decode be \$r2, \$r3 instruction

CF: Prepare rs = \$r2 for \$r2++, no forwarding (first sel_i = 0, second sel_i = 0)

EX: move \$r6 value to \$r7, write to \$r6 (new minimum distance)

(The j loop and i loop continue iterating through the array to find new minimum distances...)

(THE FOLLOWING DIAGRAM SHOWS STORING THE RESULT)

oipeline_testbench/cycle_counter	2200	170	7	1708	1709	1710	1711		1712	17	713	1714		1715	17	16	1717		1718
oipeline_testbench/instr_counter	2200	170	7	1708	1709	1710	1711	X	1712	17	713	1714		1715	17	16	1717		1718
pipeline_testbench/FDpc	176	167		169	170	171	172	X	173	17	74	175		176					
pipeline_testbench/FDinstruction	00000000	110	00111	1110111	1 (10010)	111	(1111	1111	1001111	11 01	110110	1101000	00	0000000	0				=
pipeline_testbench/CFinstruction	00000000	01100000	110001	111 (11101111	10010111		1111111	11	10011111	011101	10 X	110100	00)	00000000				
pipeline_testbench/EXinstruction	00000000	10011010	011000	000 (:	11000111	11101111	10010111		X	11111111	100111	11)	011101	10)	11010000	0000	00000		
pipeline_testbench/b5v_inst/r1	0	0																	=
pipeline_testbench/b5v_inst/r2	0	0																	_
pipeline_testbench/b5v_inst/r3	0	1	(0																_
pipeline_testbench/b5v_inst/r4	0	0																	_
pipeline_testbench/b5v_inst/r5	-47	-47																	_
pipeline_testbench/b5v_inst/r6	112	112																	=
pipeline_testbench/b5v_inst/r7	2	2																	_
pipeline_testbench/b5v_inst/r8	127	147					X ₀	1	X	2	128	X	127						_
pipeline_testbench/CFrs	1	1		18	3						7	X	1						
pipeline_testbench/CFrt	0	3	0								(8	X	3		0				
pipeline_testbench/EXrd	0	3	0			(8						X	9	X	1	(0			
oipeline_testbench/b3v_inst/result_o	0	0					1	2	X	128	127	X	129		0				_
pipeline_testbench/b14v_inst/sel_i	0	0				(1					(0								_
pipeline_testbench/b14v_inst/first_i	0	0		X:	147		X ₀	1	X	2	(2	, x	0						_
pipeline_testbench/b14v_inst/second_i	0	0					1	2	X	128	127	, x	129	1	0				
oipeline_testbench/b14v_inst/third_i	0	0																	
pipeline_testbench/b20v_inst/sel_i	0	1	0								1	X	0						
pipeline_testbench/b20v_inst/first_i	0	1	0								128	L X	0						
oipeline_testbench/b20v_inst/second_i	0	0					X 1	2	X	128	127	<u> </u>	129	X	0				
pipeline_testbench/b20v_inst/third_i	0	0																	
pipeline_testbench/b10v_inst/ram[127]	2													2					
Now	22000 ps	11111		1 1 1 1 1 1 1 80 ps		17100 ps		1712		mili		1 40 ps	1111		17160 p			17180	

- 1710: FD: Fetch/decode addi 0, \$r8 instruction
 - CF: Prepare rs = \$r8 for \$r8++, forward \$r8 from EX (first sel_i = 1, second sel_i = 0)
 - EX: Clear \$r8, write to \$r8, \$r8 = 0
- 1711: FD: Fetch/decode sll 6, \$r8 instruction
 - CF: Prepare rs = \$r8 for \$r8++, forward \$r8 from EX (first sel i = 1, second sel i = 0)
 - EX: \$r8++, write to \$r8, \$r8 = 1
- 1712: FD: Fetch/decode addi 1, \$r8 instruction
 - CF: Prepare rs = \$r8 for shift left logical 6, forward \$r8 from EX (first sel i = 1, second sel i = 0)
 - EX: \$r8++, write to \$r8, \$r8 = 2
- 1713: FD: Fetch/decode store \$r7 instruction
 - CF: Prepare rs = \$r8 for \$r8--, forward \$r8 from EX (first sel i = 1, second sel i = 0)
 - EX: sll 6, \$r8, write to \$r8, \$r8 = 128
- 1714: FD: Fetch/decode halt instruction
 - CF: Prepare rs = \$r7, rt = \$r8 for store, forward \$r8 from EX(first sel_i = 0, second sel_i = 1)

EX: \$r8--, write to \$r8, \$r8 = 127

1715: FD: Fetch/decode halt instruction

CF: halt

EX: store \$r7 value to address of \$r8, datamem[127] = \$r7 (Halt instructions continue to execute through the pipeline.)

```
ASSEMBLY CODE
Program 1:
srl
          8, $r6
                                          # srl 8, so clear $r6 (zero register)
srl
          8, $r3
                                          # srl 8, so clear $r3 (accumulator for A*B)
srl
          8, $r4
                                          # srl 8, so clear $r4 (accumulator for A*B)
srl
          8, $r8
                                          # srl 8, so clear $r8
srl
          8, $r5
                                          # clear $r5
addi
          0, $r8
                                          # $r8++, so now $r8 = 1, $r8 = &A
ld
          $r1
                                          # $r1 = A
                                          # $r8++, so now $r8 = 2, $r8 = &B
addi
          0, $r8
                                          # $r2 = B
ld
          $r2
          0, $r6
addi
          0, $r2
                                          # if (B==0), then jump to finish_1
bz
jump
          finish_1
                                          # newPC = PC + 65
bge
          1, $r6, $r2
                               # if (B==1), then jump to end B
jump
          end_B
                                          # newPC = PC + 16
btwo
          $r2
                                          # if (B%2==0), then B/=2 and jump to Bdiv_two
jump
          Bdiv_two
                               # newPC = PC + 6*
          1, $r2
addi
                                          # B-
          $r1, $r4
add
add
          $c, $r5
                                          # accumulator += A
add
          $r3, $r5
                                          #add to odd accumulator
jump
          addB_loop
                               # newPC = PC + (-8)*
          1, $r2
sll
          $r1, $r8
movl
          1, $r8
                                          # sll 1
sll
                                          # write $c to $temp
store
          $r8, $r1
movh
sll
          1, $r3
                                          # sll 1
                                          # write $temp to $c
ld
add
          $c, $r3
                                          # accumulator *=2
jump
          addB_loop
                               # newPC = PC + (-17)*
add
          $r4,$r1
add
          $c, $r5
                                          #add overflow value to r5
          $r3, $r5
                                          #add odd accumulator with even accumulator
add
movh
          $r5,$r3
srl
          8, $r8
          0,$r8
addi
addi
          0,$r8
addi
          0, $r8
                                          # $r8++, so now $r8 = 3, $r8 = &C
ld
          $r2
                                          # $r2 = C
bz
          0, $r2
                                          # if (C==0), then jump to finish_1
jump
          finish_1
                                          # newPC = PC + 35
          8, $r4
                                          # srl 8, so clear $r4 for lower 16 bit odd accumulator
srl
srl
          8, $r5
                                          # srl 8, so clear $r5 for higher 16 bit odd accumulator
bge
          1, $r6.$r2
                               # if (C==1), then jump to end C
                                          # newPC = PC + 17
          end_C
jump
btwo
          $r2
                                          # if (C%2==0), then C/=2 and jump to Cdiv_two
```

```
Cdiv_two
                               # newPC = PC + 6
jump
addi
          1, $r2
                                          # C-
add
          $r1, $r4
add
          $c, $r5
add
          $r3, $r5
                                         # accumulator += (A*B)
                               # newPC = PC + (-8)
jump
          addC_loop
sll
          1, $r2
movl
          $r1, $r8
          1, $r8
                                         # sll 1
sll
          $r8, $r1
movh
                                         # write $c to $temp
store
          1, $r3
                                         # sll 1
srl
ld
                                         # write $temp to $c
add
          $c, $r3
                                         # accumulator *=2
          addC_loop
jump
                               # newPC = PC + (-17)
add
          $r1, $r4
add
          $c, $r5
add
          $r3,$r5
                                          #accumulator=A*B*C
srl
          8, $r8
          0, $r8
addi
          0, $r8
addi
addi
          0, $r8
addi
          0, $r8
                                         # $r8 = 4
store
          $r5
                                         # store 4 high bits into memory address 4
addi
          0, $r8
                                         # $r8 = 5
          $r4
                                         # store 4 low bits into memory address 5
store
halt
srl
          1, $r8
          0, $r8
addi
          0, $r8
addi
addi
          0, $r8
addi
          0, $r8
                                         # $r8 = 4
srl
          1, $r6
store
          $r6
                                         # store 0 into memory address 4
          0, $r8
                                         # $r8 = 5
addi
                                         # store 0 into memory address 5
store
          $r6
halt
program_2:
                                         # srl 8, so clear $r6
srl
          8, $r6
srl
          8, $r8
                                         # srl 8, so clear $r8
addi
          0, $r8
                                         # $r8++, so now $r8 = 1
sll
          2, $r8
                                         # sll 2, so now $r8 = 4
          0, $r8
                                         # $r8++, so now $r8 = 5
addi
                                         # $r8++, so now $r8 = 6, so now $r8 = address of 4-bit string
addi
          0, $r8
                                         #$r2 holds 8-bit string containing the 4-bit substring
ld
          $r2
          4, $r2
                                         # sll 4
sll
srl
          1, $r2
                                         # srl 1
```

```
1, $r2
                                           # srl 1
srl
          1, $r2
                                           # srl 1
srl
srl
          1, $r2
                                           # srl 1, so now $r2 = 4-bit string
srl
          8, $r5
                                           # srl 8, so clear $r5 (matched_counter)
srl
          8, $r3
                                           # srl 8, so clear $r3
addi
          0, $r3
                                           # $r3++, so now $r3 = 1
movh
          $r8, $r7
                                           # $r7 = $r8
addi
          1, $r7
                                           # $r7--, so now $r7 = 5
sll
          6, $r3
                                           # sll 6, $r3 = 64 (strings_checked_counter)
          $r3, $r8
                                           # $r8 = $r3
movl
srl
          1, $r8
                                           # srl 1, so now $r8 = 32, $r8 = address of string array
loop_next_string:
srl
          8, $r1
                                           # srl 8, so clear $r1
bz
          $r3
                                           # if (\$r3 == 0), then jump to finish_2
          finish_2
jump
                                           # newPC = PC + 17
ld
          $r4
                                           # $r4 = 8 bit string
loop_next_bit:
be
          0, $r2, $r4
                                # if ($r2 == $r4), then jump to equal_string
                                           \# newPC = PC + 6
jump
          equal_string
          1, $r4
                                           # srl 1
srl
addi
          0, $r1
                                           # $r1++
bge
          $r1, $r7
                                           # if (\$r1 \ge \$r7), then jump to end_string
jump
          end_string
                                # newPC = PC + 6
          loop_next_bit
                                           # newPC = PC + (-6)
jump
equal_string:
                                           # $r5++ (matched_counter)
addi
          0, $r5
addi
          1, $r3
                                           # $r3-- (strings_checked_counter)
          0, $r8
                                           # $r8++ (address of string array)
addi
                                           \# newPC = PC + (-15)
jump
          loop_next_string
end_string:
addi
          0, $r8
                                           #$r8++ (address of string array)
          1, $r3
addi
                                           # $r3-- (strings_checked_counter)
          loop_next_string
                                           \# newPC = PC + (-18)
jump
finish_2:
          $r7, $r8
                                           # $r8 = 5
movh
                                           # $r8++, so now $r8 = 6
addi
          0, $r8
          0, $r8
                                           # $r8++, so now $r8 = 7
addi
          $r5
                                           # store matched_counter into memory address 7
store
halt
program_3:
srl
          8, $r8
                                           # srl 8, so clear $r8
addi
          0, $r8
                                           # $r8++, so now $r8 = 1
                                           # sll 2, so now $r8 = 4
sll
          2, $r8
sll
          1, $r8
                                           # sll 1, so now $r8 = 8
addi
          1, $r8
                                           # $r8--, so now $r8 = 7
                                           # $r3 = 7
          $r8, $r3
movh
addi
          0, $r3
                                           # $r3++, so now $r3 = 8
```

```
0, $r3
                                           # $r3++, so now $r3 = 9
addi
sll
          1, $r3
                                           # sll 1, so now $r3 = 18
addi
          0, $r3
                                           # $r3++, so now $r3 = 19
          $r3, $r4
                                           # $r4 = $r3
movl
srl
          8, $r1
                                           # srl 8, so clear $r1 (i_counter for outer loop)
srl
          8, $r2
                                           # srl 8, so clear $r2 (j_counter for inner loop)
srl
          8, $r8
                                           # srl 8, so clear $r8
addi
          0, $r8
                                           # $r8++, so now $r8 = 1
                                           # $r8++, so now $r8 = 2
addi
          0, $r8
sll
          6, $r8
                                           #$r8 = 128, $r8 = memory address of int array
ld
                                           # $r5 = first integer to compare distance
          $r5
addi
          0, $r8
                                           # $r8++, so now $r8 = 129
ld
          $r6
                                           # $r6 = second integer to compare distance
addi
          0, $r2
                                           # j++
diff
          $r6, $r5
                                           # $r5 = |$r6 - $r5|
movh
          $r6, $r7
                                           # $r7 = minimum_distance
i_loop:
addi
          0, $r1
                                           # $r1++
j_loop:
          0, $r8
                                           # $r8++
addi
ld
          $r6
                                           # $r6 = next second integer to compare distance
diff
          $r6, $r5
                                           # $r6 = |$r6 - $r5|
bge
          $r6, $r7
                                           # if (\$r6 \ge \$r7), then jump to j_continue
jump
          j_continue
                                           \# newPC = PC + 2
movh
          $r6, $r7
                                           # $r7 = new minimum_distance
j_continue:
addi
          0, $r2
                                           # j++
          $r2, $r3
                                           # if ($r2 == $r3), then jump to end_j, else jump to j_loop
be
                                           \# newPC = PC + 2
jump
          end_j
                                           # newPC = PC + (-10)
jump
          j_loop
end_j:
addi
          1, $r4
                                           # $r4--
udiff
          $r8, $r4
                                           # $r8 = |$r8 - $r4|
ld
          $r5
                                           # $r5 = next first integer to compare distance
          8, $r2
srl
                                           # srl 8, so clear $r2
srl
          8, $r1
                                           # srl 8, so clear $r1
addi
          1, $r3
                                           # $r3--
          $r1, $r3
                                           # if (\$r1 \ge \$r3), then jump to finish_3
bge
jump
          finish_3
                                           \# newPC = PC + 2
jump
          i_loop
                                           # newPC = PC + (-21)
finish_3:
srl
          8, $r8
                                           # srl 8, so clear $r8
addi
          0, $r8
                                           # $r8++, so now $r8 = 1
                                           # $r8++, so now $r8 = 2
addi
          0, $r8
sll
          6, $r8
                                           # sll 6, so $r8 = 128
addi
          1, $r8
                                           # $r8--, so now $r8 = 127
          $r7
                                           # store minimum_distance into memory address 127
store
halt
```

MACHINE CODE

0:11101101	36:10010111	72 : 00000000	106 : 10000011	140 : 10010111
1:11101010	37 : 10010111	73 : 11101111	107 : 01010111	141:10010111
2:11101011	38:10000001	74 : 10010111	108 : 11001000	142 : 11111111
3:11101111	39:10100001	75 : 10010111	109 : 11100011	143:10000100
4:11101100	40 : 11001011	76 : 10010111	110 : 10010000	144:10010111
5:10010111	41 : 11101011	77 : 10010111	111:01100011	145:10000101
6:10000000	42:11101100	78 : 11101101	112 : 11001000	146:10010001
7:10010111	43:01101101	79:01110101	113 : 11000110	147:00101010
8:10000001	44 : 11001010	80:01110101	114 : 10010100	148:01000110
9:10010101	45 : 10110001	81:01110101	115 : 10011010	149:10010000
10:10100001	46:11001000	82:00000000	116 : 10010111	150 : 10010111
11:11001100	47 : 10011001		117 : 11000011	151:10000101
12:01101101	48:00010110	// Program 2	118 : 10010111	152:00101010
13:11001010	49:00010011	83:11101101	119 : 10011010	153:01101111
14:10110001	50:00011011	84:11101111	120 : 11000001	154:11000111
15 : 11001000	51:11000101	85 : 10010111	121:01001011	155:01000110
16:10011001	52:11100001	86:11110111	122 : 10010111	156:10010001
17:00010110	53:00110011	87 : 10010111	123 : 10010111	157:01011110
18:00010011	54 : 11110011	88:10010111	124 : 01110100	158 : 11000111
19:00011011	55:01111000	89:10000001	125 : 00000000	159:11000100
20 : 11000101	56:01001100	90 : 11111000		160 : 10011011
21:11100001	57 : 11110001	91:11100001	//Program 3	161:11011101
22:00110011	58:10001000	92:11100001	126 : 11101111	162:10000100
23:11110011	59:00010001	93:11100001	127 : 10010111	163:11101001
24:01111000	60 : 11000001	94:11100001	128 : 11110111	164 : 11101000
25 : 01001100	61:00010110	95 : 11101100	129 : 11110011	165 : 10011010
26:11110001	62:00010011	96 : 11101010	130 : 10011111	166:01100000
27 : 10001000	63:00011011	97 : 10010010	131:01001101	167 : 11000111
28:00010001	64 : 11101111	98:01001110	132 : 10010010	168:11000000
29:11000001	65 : 10010111	99:10011110	133 : 10010010	169 : 11101111
30:00011100	66 : 10010111	100 : 11111101	134 : 11110001	170:10010111
31:00010011	67 : 10010111	101:00111011	135 : 10010010	171:10010111
32:00011011	68:10010111	102 : 11100111	136:00111001	172 : 11111111
33:01000001	69:01110100	103 : 11101000	137 : 11101000	173 : 10011111
34 : 11101111	70 : 10010111	104 : 10100010	138 : 11101001	174 : 01110110
35 : 10010111	71:01110011	105 : 11001001	139 : 11101111	175 : 11010000