KJ Architecture

ISA REVIEW

Instruction formats:

R-type:

opcode: 4 bits	rs: 2 bits	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 writes carry out bit to $c)	1	if rs = 0, then use $c
		if rs = 1, then use $r1
		if rs = 2, then use $r3
		if rs = 3, then use $r4
			if rd = 0, then use $r1
			if rd = 1, then use $r3
			if rd = 2, then use $r4
			if rd = 3, then use $r5

diff (find absolute difference rs between rd where rs and rd are signed, write to rs)	2	if rs = 0, then use $r1
		if rs = 1, then use $r2
		if rs = 2, then use $r6
		if rs = 3, then use $r8
			if rd = 0, then use $r3
			if rd = 1, then use $r4
			if rd = 2, then use $r5
			if rd = 3, then use $r6

movl (move data in rs to rd)	3	if rs = 0, then use $r1
		if rs = 1, then use $r2
		if rs = 2, then use $r3
		if rs = 3, then use $r4
			if rd = 0, then use $r2
			if rd = 1, then use $r4
			if rd = 2, then use $r7
			if rd = 3, then use $r8

movh (move data in rs to rd)	4	if rs = 0, then use $r5
		if rs = 1, then use $r6
		if rs = 2, then use $r7
		if rs = 3, then use $r8
			if rd = 0, then use $r1
			if rd = 1, then use $r3
			if rd = 2, then use $r7
			if rd = 3, then use $r8

udiff (find absolute difference rs between rd where rs and rd are unsigned, write to rs)	13	if rs = 0, then use $r1
		if rs = 1, then use $r2
		if rs = 2, then use $r6
		if rs = 3, then use $r8
			if rd = 0, then use $r3
			if rd = 1, then use $r4
			if rd = 2, then use $r5
			if rd = 3, then use $r6

sll (shift left logical, write to rd, implicitly writes carry out bit to $c)	15	if rs = 0, then sll 1 bit
		if rs = 1, then sll 2 bits
		if rs = 2, then sll 4 bits
		if rs = 3, then sll 6 bits
			if rd = 0, then use $r2
			if rd = 1, then use $r3
			if rd = 2, then use $r4
			if rd = 3, then use $r8

Operations with B-type format:

instruction	**opcode**	**opt**	**rs**	**rt**

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

= PC + 2)		operands		if rt = 3, then use $r4

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

st (implicitly stores data from rs to memory address in $r8 OR writes $c to $temp)	7	if opt = 0, store data to memory address
		if opt = 1, write $c to $temp
			if rs = 0, then use $r1 if rs = 4, then use $r5
			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

ld (implicitly loads data from memory address in $8, writes to rs OR writes $temp to $c)	8	if opt = 0, load data from memory address
		if opt = 1, write $temp to $c
			if rs = 0, then use $r1 if rs = 4, then use $r5
			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

addi (increments or decrements to rs)	9	if opt = 0, increment
		if opt = 1, decrement
			if rs = 0, then use $r1 if rs = 4, then use $r5
			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 register). $r6 == rs, if true then newPC = PC + 1, else then newPC = PC + 2)	10	(always) 0	if rs = 0, then use $r1 if rs = 4, then use $r5
			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

btwo (checks if rs % 2 == 0, if true then newPC = PC + 1, else then newPC = PC + 2)	11	(always) 0	if rs = 0, then use $r1 if rs = 4, then use $r5
			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

srl (shift right logical, write to rs)	14	if opt = 0, srl 1 bit
		if opt = 1, srl 8 bits
			if rs = 0, then use $r1 if rs = 4, then use $r5
			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

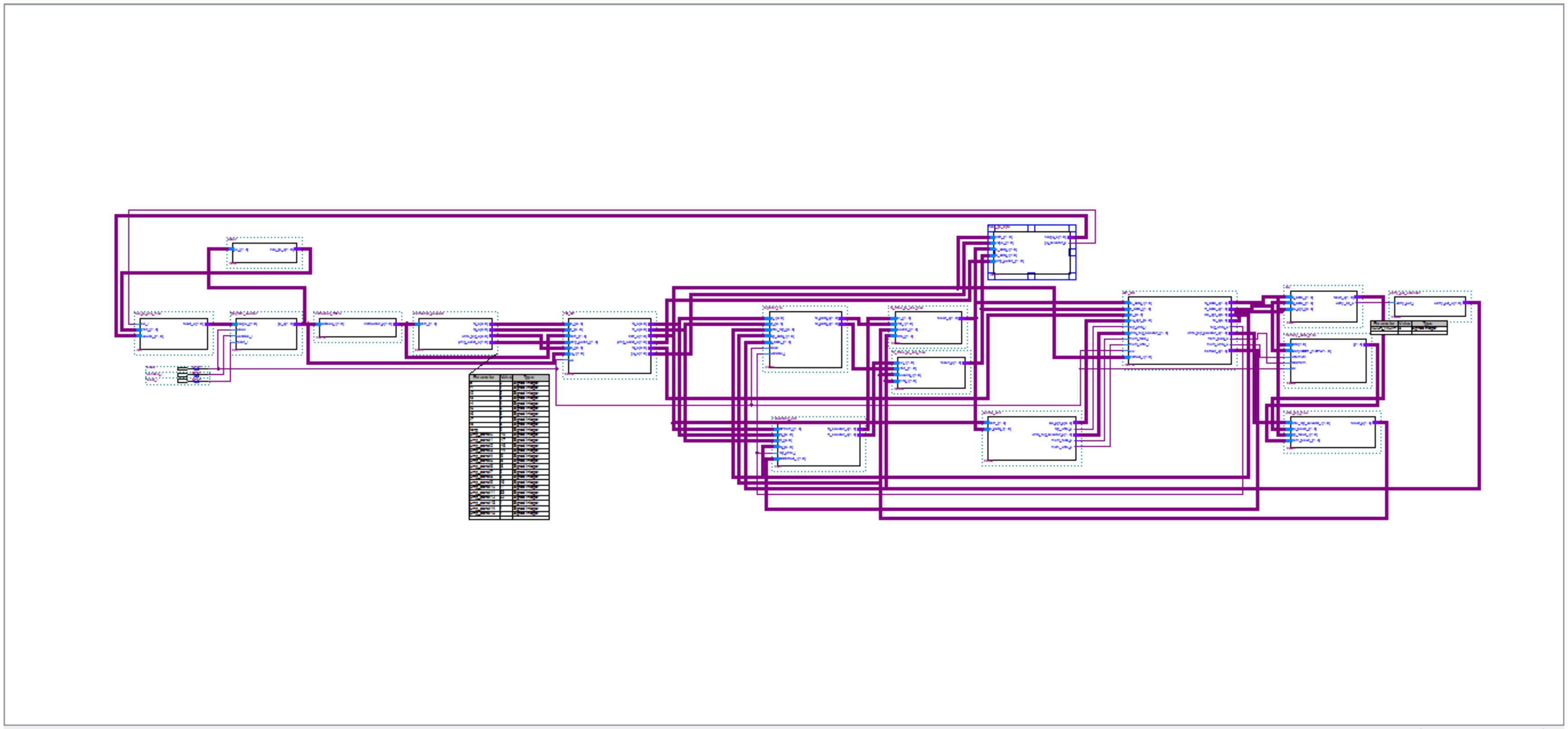
Operations with J-type format:

Instruction	opcode	const

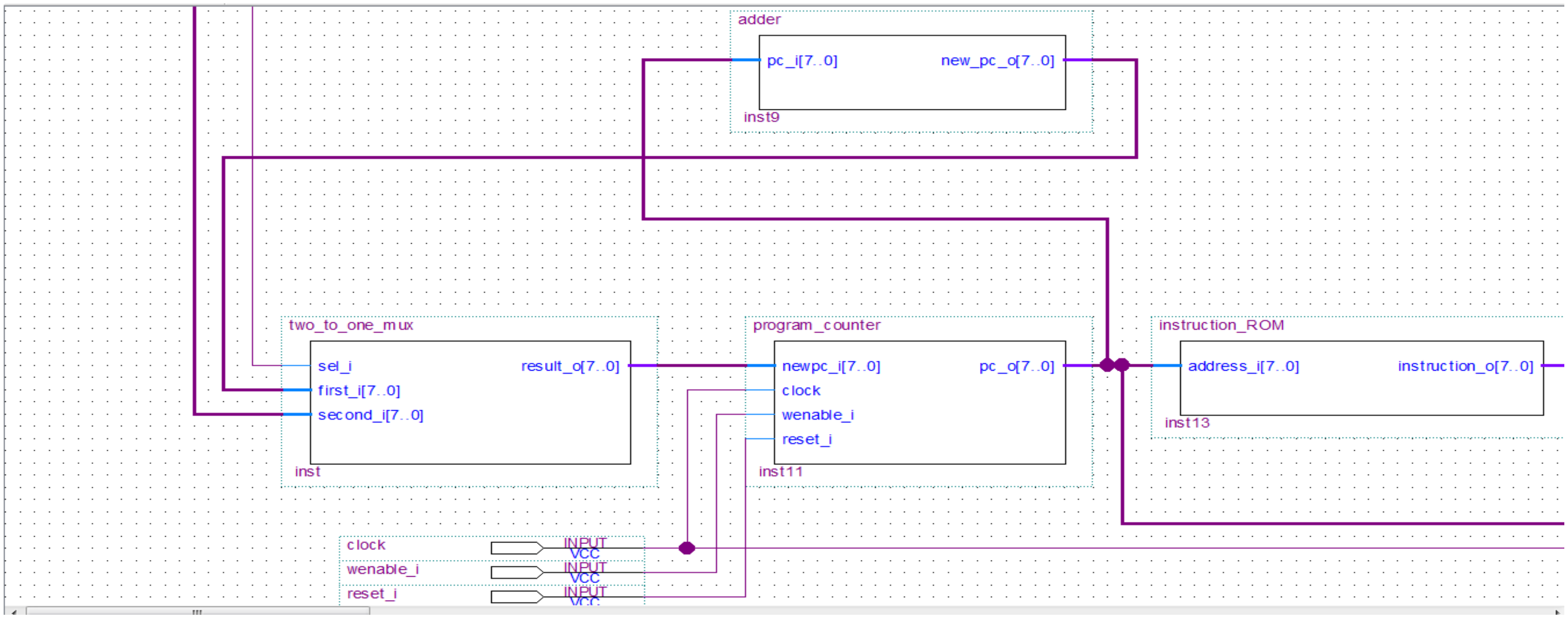
jump (jump to instruction address, newPC = PC + const)	12	if const = 0, use -19 if const = 5, use -8 if const = 10, use 6
		if const = 1, use = -17 if const = 6, use -7 if const = 11, use 13
		if const = 2, use -14 if const = 7, use -6 if const = 12, use 14
		if const = 3, use -13 if const = 8, use 2 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

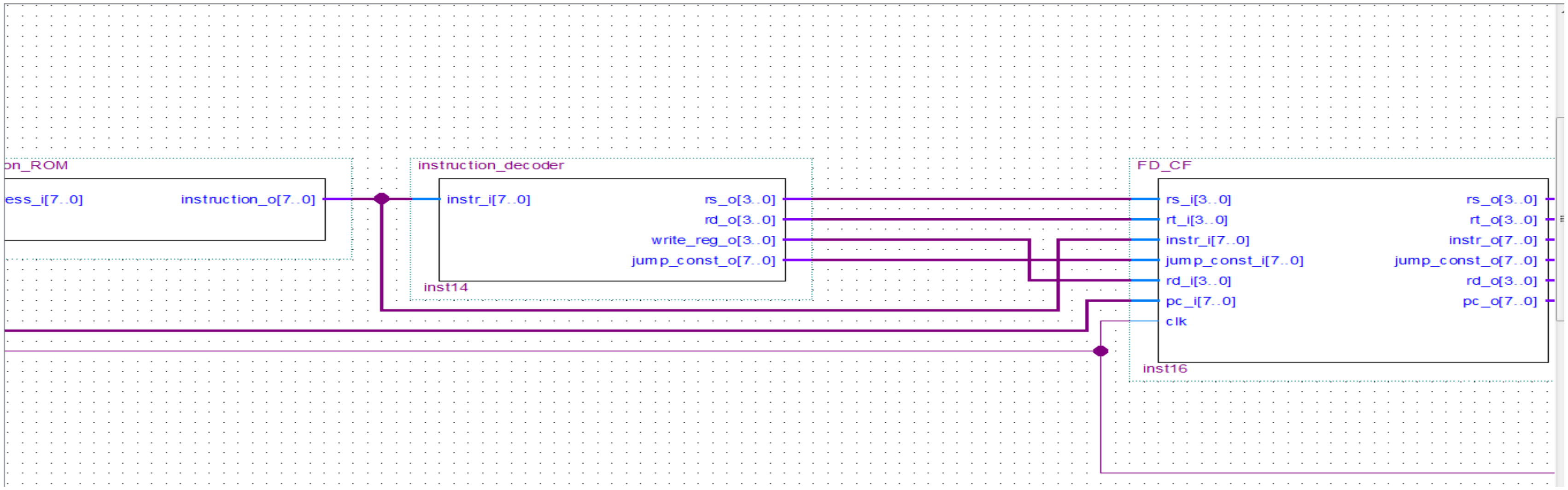
SCHEMATICS

Overview

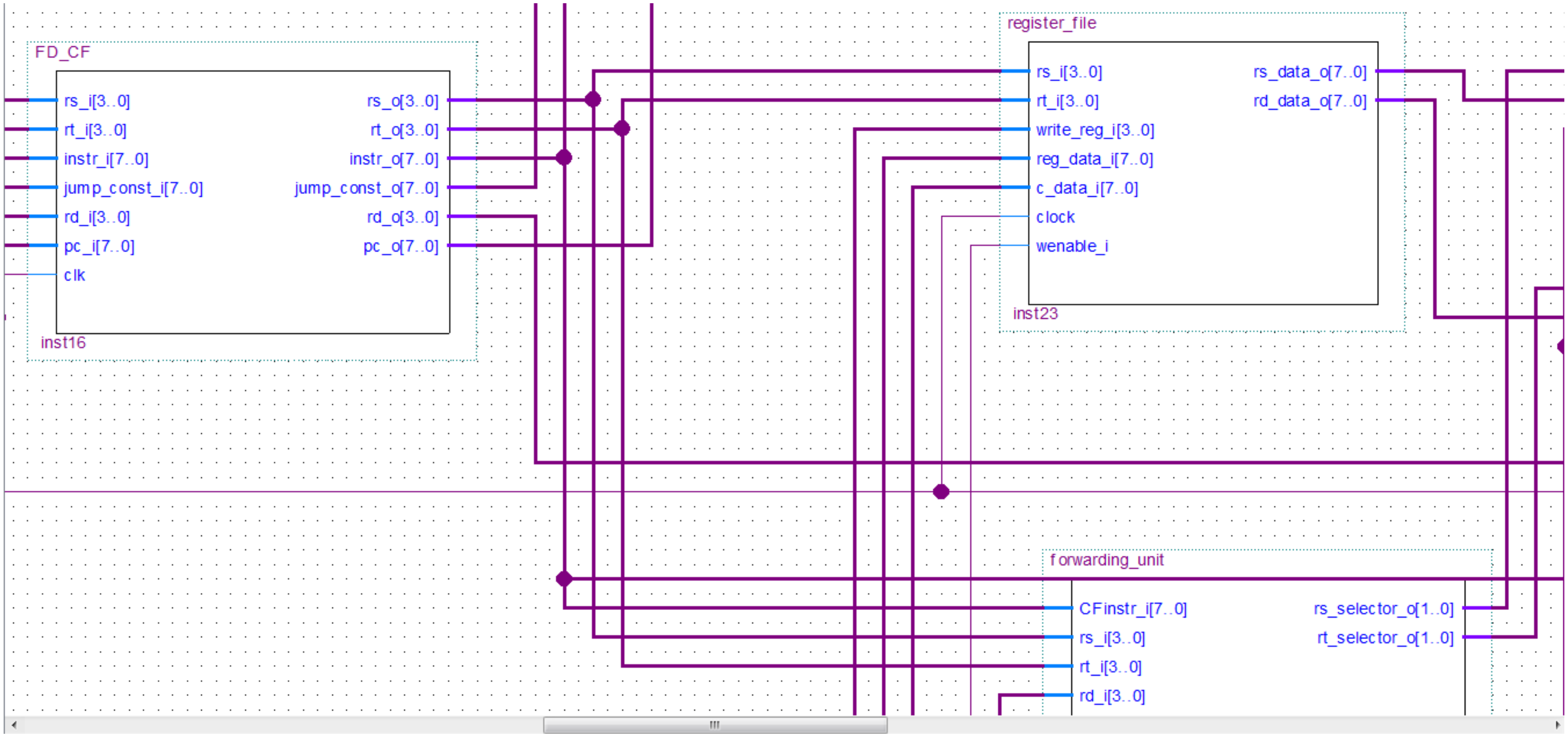


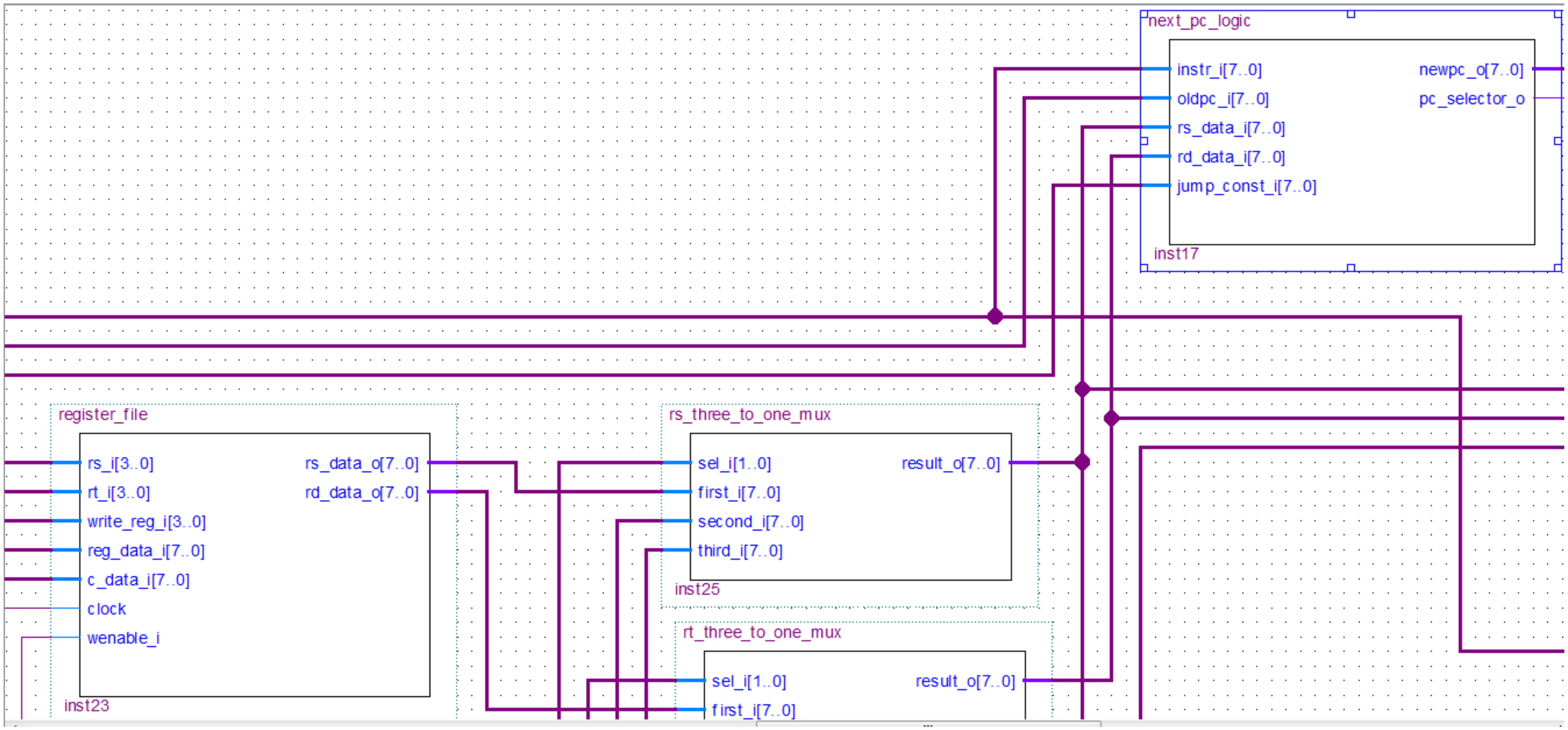
FD Stage

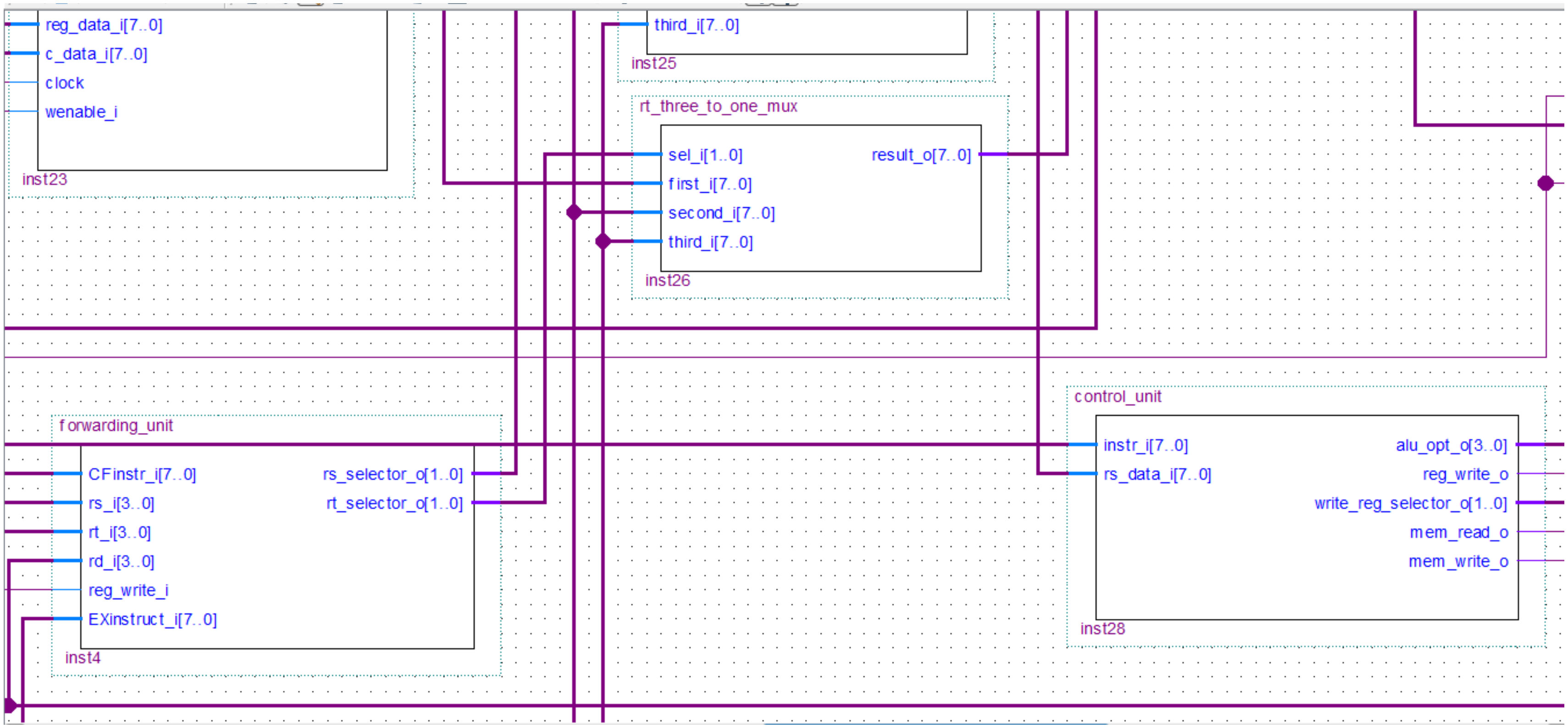

adder
pc_i[7..0]
new_pc_o[7..0]
inst9
two_to_one_mux
sel_i
first_i[7..0]
second_i[7..0]
result_o[7..0]
inst
program_counter
newpc_i[7..0]
clock
wenable_i
reset_i
pc_o[7..0]
inst11
instruction_ROM
address_i[7..0]
instruction_o[7..0]
inst13
clock
wenable_i
reset_i
INPUT
VCC
INPUT
VCC
INPUT
VCC


on_ROM
ess_i[7..0]
instruction_o[7..0]
instruction_decoder
instr_i[7..0]
rs_o[3..0]
rd_o[3..0]
write_reg_o[3..0]
jump_const_o[7..0]
inst14
FD_CF
rs_i[3..0]
rt_i[3..0]
instr_i[7..0]
jump_const_i[7..0]
rd_i[3..0]
pc_i[7..0]
clk
rs_o[3..0]
rt_o[3..0]
instr_o[7..0]
jump_const_o[7..0]
rd_o[3..0]
pc_o[7..0]
inst16

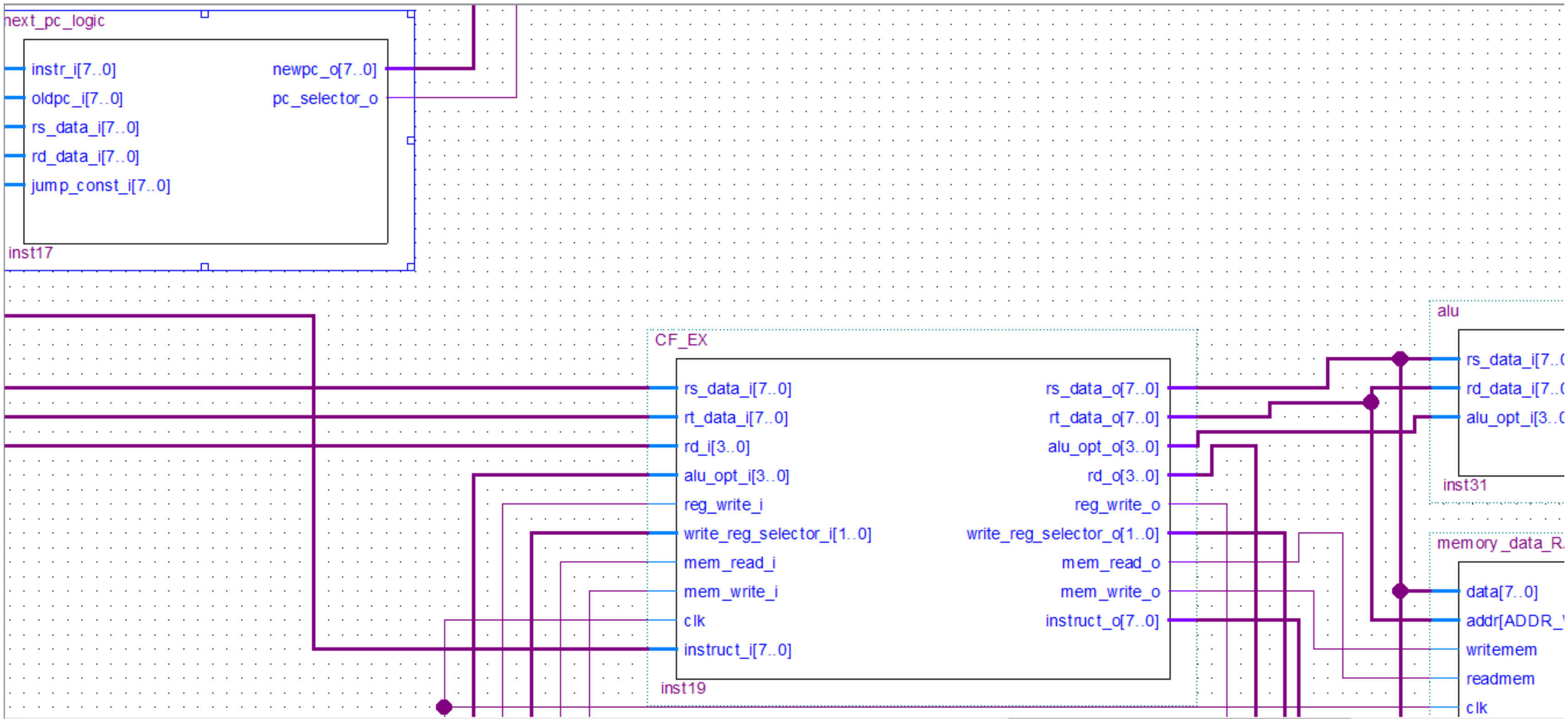
CF Stage

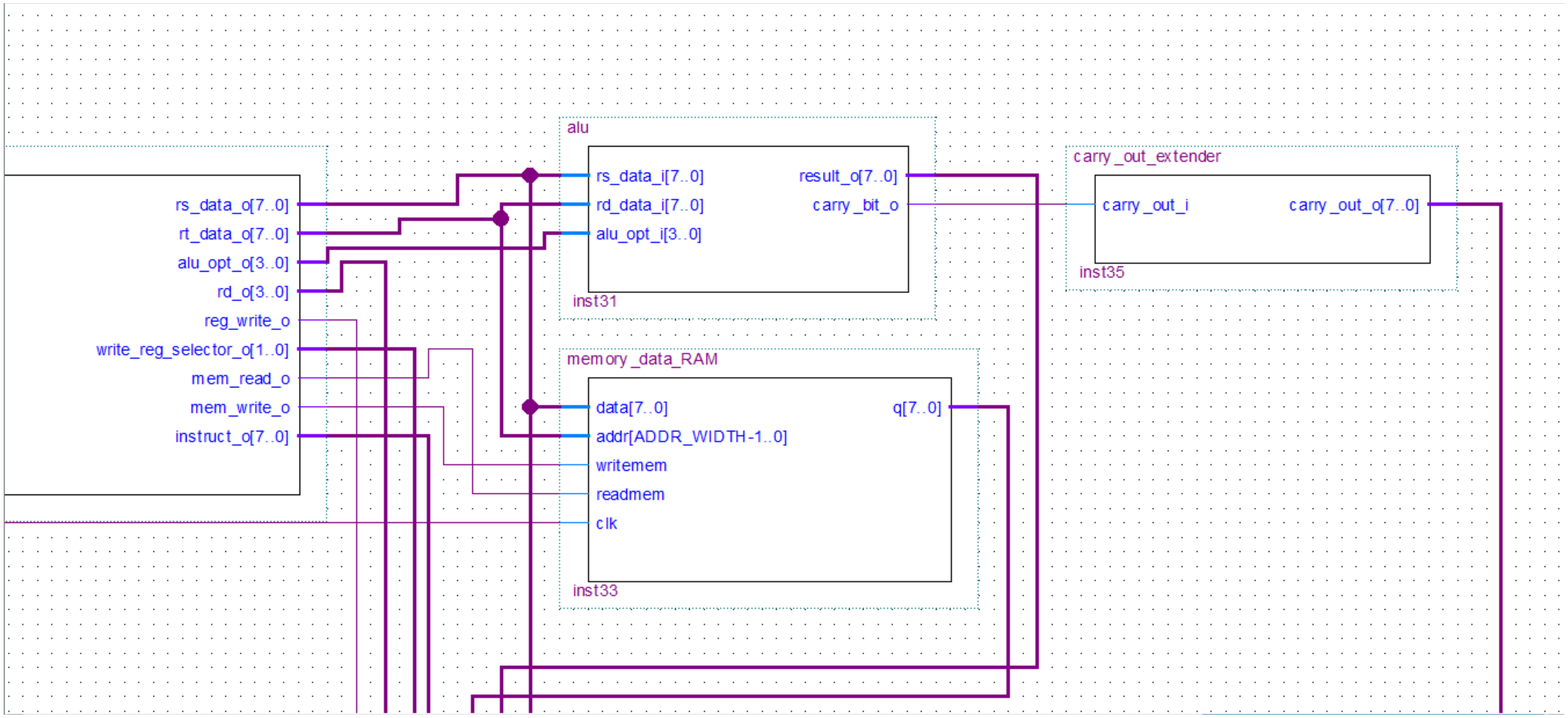

FD_CF
rs_i[3..0]
rt_i[3..0]
instr_i[7..0]
jump_const_i[7..0]
rd_i[3..0]
pc_i[7..0]
clk
rs_o[3..0]
rt_o[3..0]
instr_o[7..0]
jump_const_o[7..0]
rd_o[3..0]
pc_o[7..0]
inst16
register_file
rs_i[3..0]
rt_i[3..0]
write_reg_i[3..0]
reg_data_i[7..0]
c_data_i[7..0]
clock
wenable_i
rs_data_o[7..0]
rd_data_o[7..0]
inst23
forwarding_unit
CFinstr_i[7..0]
rs_i[3..0]
rt_i[3..0]
rd_i[3..0]
rs_selector_o[1..0]
rt_selector_o[1..0]


next_pc_logic
instr_i[7..0]
oldpc_i[7..0]
rs_data_i[7..0]
rd_data_i[7..0]
jump_const_i[7..0]
newpc_o[7..0]
pc_selector_o
inst17
register_file
rs_i[3..0]
rt_i[3..0]
write_reg_i[3..0]
reg_data_i[7..0]
c_data_i[7..0]
clock
wenable_i
rs_data_o[7..0]
rd_data_o[7..0]
inst23
rs_three_to_one_mux
sel_i[1..0]
first_i[7..0]
second_i[7..0]
third_i[7..0]
result_o[7..0]
inst25
rt_three_to_one_mux
sel_i[1..0]
first_i[7..0]
result_o[7..0]

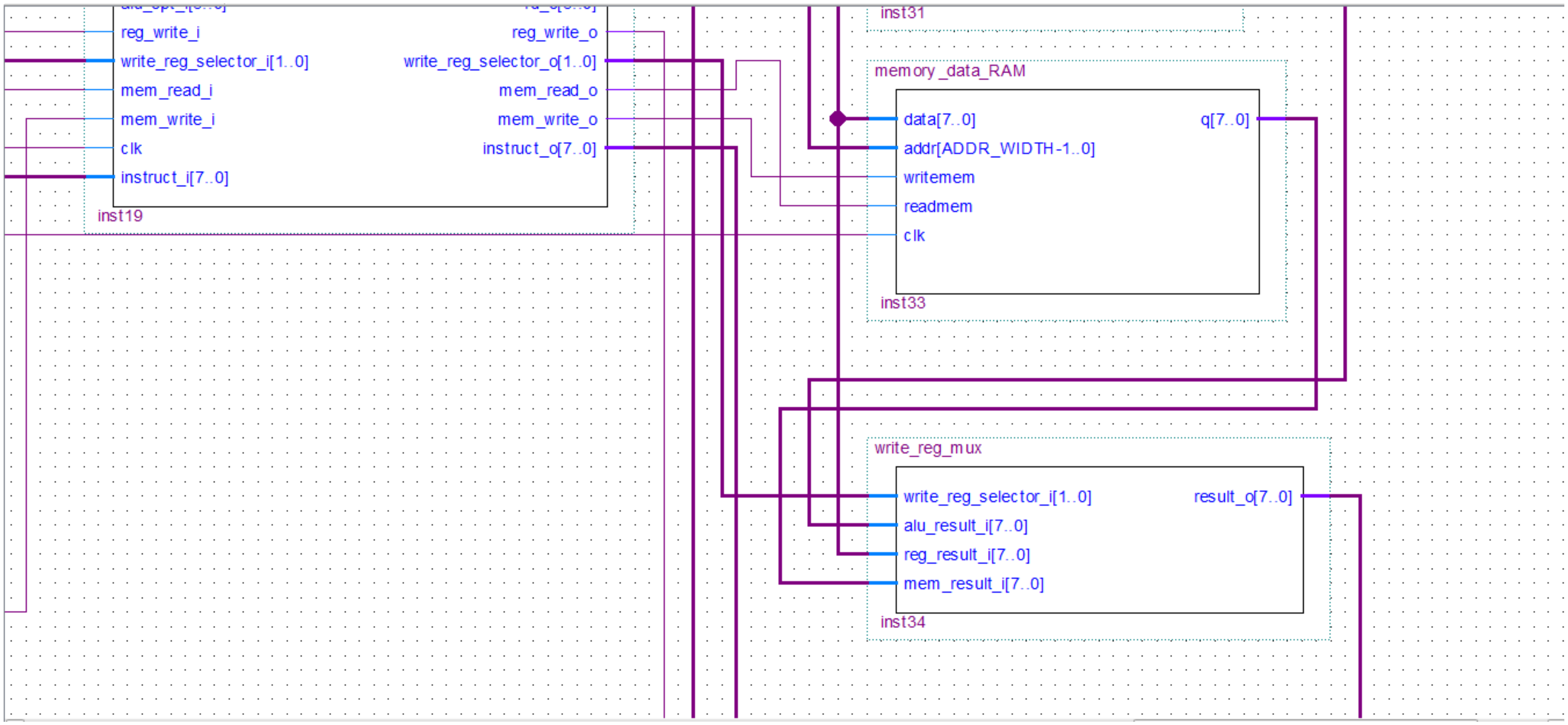

reg_data_i[7..0]
c_data_i[7..0]
clock
wenable_i
inst23
third_i[7..0]
inst25
rt_three_to_one_mux
sel_i[1..0]
first_i[7..0]
second_i[7..0]
third_i[7..0]
result_o[7..0]
inst26
forwarding_unit
CFinstr_i[7..0]
rs_i[3..0]
rt_i[3..0]
rd_i[3..0]
reg_write_i
EXinstruct_i[7..0]
rs_selector_o[1..0]
rt_selector_o[1..0]
inst4
control_unit
instr_i[7..0]
rs_data_i[7..0]
alu_opt_o[3..0]
reg_write_o
write_reg_selector_o[1..0]
mem_read_o
mem_write_o
inst28

EX Stage


next_pc_logic
instr_i[7..0]
oldpc_i[7..0]
rs_data_i[7..0]
rd_data_i[7..0]
jump_const_i[7..0]
newpc_o[7..0]
pc_selector_o
inst17
CF_EX
rs_data_i[7..0]
rt_data_i[7..0]
rd_i[3..0]
alu_opt_i[3..0]
reg_write_i
write_reg_selector_i[1..0]
mem_read_i
mem_write_i
clk
instruct_i[7..0]
rs_data_o[7..0]
rt_data_o[7..0]
alu_opt_o[3..0]
rd_o[3..0]
reg_write_o
write_reg_selector_o[1..0]
mem_read_o
mem_write_o
instruct_o[7..0]
inst19
alu
rs_data_i[7..(
rd_data_i[7..(
alu_opt_i[3..(
inst31
memory_data_R
data[7..0]
addr[ADDR_'
writemem
readmem
clk



rs_data_o[7..0]
rt_data_o[7..0]
alu_opt_o[3..0]
rd_o[3..0]
reg_write_o
write_reg_selector_o[1..0]
mem_read_o
mem_write_o
instruct_o[7..0]
alu
rs_data_i[7..0]
rd_data_i[7..0]
alu_opt_i[3..0]
result_o[7..0]
carry_bit_o
inst31
carry_out_extender
carry_out_i
carry_out_o[7..0]
inst35
memory_data_RAM
data[7..0]
addr[ADDR_WIDTH-1..0]
writemem
readmem
clk
q[7..0]
inst33


alu_opt_i[3..0]
reg_write_i
write_reg_selector_i[1..0]
mem_read_i
mem_write_i
clk
instruct_i[7..0]
rd_o[3..0]
reg_write_o
write_reg_selector_o[1..0]
mem_read_o
mem_write_o
instruct_o[7..0]
inst19
inst31
memory_data_RAM
data[7..0]
addr[ADDR_WIDTH-1..0]
writemem
readmem
clk
q[7..0]
inst33
write_reg_mux
write_reg_selector_i[1..0]
alu_result_i[7..0]
reg_result_i[7..0]
mem_result_i[7..0]
result_o[7..0]
inst34

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;
  1:  result_o = second_i;
  endcase
end
endmodule

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;
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
begin
  new_pc_o=pc_i+1;
end

endmodule
```

```
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
  case(alu_opt_i)
    //add
    0:begin
      result_o =
rs_data_i+rd_data_i;


if(255<rs_data_i+rd_data_i)
        carry_bit_o=1;
      else
        carry_bit_o=0;
    end
    //shift left 1
    1:begin
      carry_bit_o=rs_data_i[7];
      result_o=rs_data_i<<1;
    end
    //difference
    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-
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)
```

```
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
and rd_data_i is negative
      else
      begin


if((~rs_data_i+1)>(~rd_data
_i+1))
          result_o =
(~rs_data_i+1)-
(~rd_data_i+1);
        else
          result_o
=(~rd_data_i+1)-
(~rs_data_i+1);
      end
      carry_bit_o=0;
    end
    //increment
    3:begin
      if(255<rs_data_i+1)
        carry_bit_o=1;
      else
        carry_bit_o=0;
      result_o=rs_data_i+1;
    end
    //decrement
    4:begin
      result_o=rs_data_i-1;
      carry_bit_o=0;
    end
    //shift right 1
    5:begin
      carry_bit_o=rs_data_i[0];
      result_o=rs_data_i>>1;
```

```
    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
    //shift left 6
    8:begin
      result_o=rs_data_i<<6;
      carry_bit_o=0;
    end
    //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
      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
  endcase


end
endmodule
```

```
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

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,
   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]
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;
  logic [1:0] write_reg_selector;
  logic mem_read;
  logic mem_write;
  logic [7:0]instruct;

always_comb
```

```
begin
  rs_data_o=rs_data;
  rt_data_o=rt_data;
  alu_opt_o=alu_opt;
  rd_o=rd;
  reg_write_o=reg_write;

write_reg_selector_o=write_reg_
selector;
  mem_read_o=mem_read;
  mem_write_o=mem_write;
  instruct_o=instruct;
end

always_ff @ (negedge clk)
begin
    rs_data=rs_data_i;
    rt_data=rt_data_i;
    alu_opt=alu_opt_i;
    rd=rd_i;
     reg_write=reg_write_i;

write_reg_selector=write_reg_sel
ector_i;
     mem_read=mem_read_i;
     mem_write=mem_write_i;
    instruct=instruct_i;
end
endmodule

module control_unit (
  input logic [7:0] instr_i,
  input [7:0] rs_data_i,
  output logic [3:0] alu_opt_o,
  output logic reg_write_o,
  output logic [1:0]
write_reg_selector_o,
  output logic mem_read_o,
  output logic mem_write_o
  );

 // Determine alu_opt_o
always_comb begin
  case(instr_i[7:4])
  1: alu_opt_o = 0;
  2: alu_opt_o = 2;
  9: begin
  if(instr_i[3] == 0)
  begin
    alu_opt_o = 3;
  end
  else
  begin
    alu_opt_o = 4;
  end
  end
  11: alu_opt_o = 5;
  13: alu_opt_o = 10;
  14: begin
  if(instr_i[3] == 0)
  begin
    alu_opt_o = 5;
  end
  else
  begin
    alu_opt_o = 9;
  end
  end
  15: begin
  case(instr_i[3:2])
  0: begin
    alu_opt_o = 1;
  end
  1: begin
    alu_opt_o = 6;
  end
  2: begin
    alu_opt_o = 7;
  end
  3: begin
    alu_opt_o = 8;
  end
  endcase
  end
  default: alu_opt_o = 0;
  endcase
end

// Determine reg_write_o,
write_reg_selector_o,
mem_read_o, mem_write_o
always_comb begin
case(instr_i[7:4])
  1: begin
    reg_write_o = 1;
  write_reg_selector_o = 0;
  mem_read_o = 0;
  mem_write_o = 0;
end
2: begin
  reg_write_o = 1;
  write_reg_selector_o = 0;
  mem_read_o = 0;
  mem_write_o = 0;
end
3: begin
  reg_write_o = 1;
  write_reg_selector_o = 1;
  mem_read_o = 0;
  mem_write_o = 0;
end
4: begin
  reg_write_o = 1;
  write_reg_selector_o = 1;
  mem_read_o = 0;
  mem_write_o = 0;
end
7: begin
if(instr_i[3] == 0)
begin
  reg_write_o = 0;
  write_reg_selector_o = 0;
  mem_read_o = 0;
  mem_write_o = 1;
end
else
begin
  reg_write_o = 1;
  write_reg_selector_o = 1;
  mem_read_o = 0;
  mem_write_o = 0;
end
end
8: begin
if(instr_i[3] == 0)
begin
  reg_write_o = 1;
  write_reg_selector_o = 2;
  mem_read_o = 1;
  mem_write_o = 0;
end
else
begin
  reg_write_o = 1;
  write_reg_selector_o = 1;
  mem_read_o = 0;
  mem_write_o = 0;
end
end
9: begin
  reg_write_o = 1;
  write_reg_selector_o = 0;
  mem_read_o = 0;
  mem_write_o = 0;
end
11: begin
if(rs_data_i[0]==0)
begin
  reg_write_o = 0;
  write_reg_selector_o = 0;
  mem_read_o = 0;
  mem_write_o = 0;
end
else
begin
  reg_write_o = 0;
  write_reg_selector_o = 0;
  mem_read_o = 0;
  mem_write_o = 0;
end
end
13: begin
  reg_write_o = 1;
  write_reg_selector_o = 0;
  mem_read_o = 0;
  mem_write_o = 0;
end
14: begin
  reg_write_o = 1;
  write_reg_selector_o = 0;
  mem_read_o = 0;
  mem_write_o = 0;
end
15: begin
  reg_write_o = 1;
  write_reg_selector_o = 0;
  mem_read_o = 0;
  mem_write_o = 0;
end
default: begin
  reg_write_o = 0;
    write_reg_selector_o = 0;
    mem_read_o = 0;
    mem_write_o = 0;
  end
endcase
end
endmodule

module FD_CF (
  input logic [3:0] rs_i,
  input  logic [3:0] rt_i,
  input  logic [7:0] instr_i,
  input  byte jump_const_i,
  input  logic [3:0] rd_i,
  input logic [7:0] pc_i,
  input clk,
  output logic [3:0] rs_o,
  output logic [3:0] rt_o,
  output logic [7:0] instr_o,
  output byte jump_const_o,
  output logic [3:0] rd_o,
  output logic [7:0] pc_o
);
  logic [3:0] rs;
  logic [3:0] rt;
  logic [7:0] instr;
  byte jump_const;
  logic [3:0] rd;
  logic [7:0] pc;

always_comb
begin
  rs_o=rs;
  rt_o=rt;
  instr_o=instr;
  jump_const_o=jump_const;
  rd_o=rd;
  pc_o=pc;
end

always_ff @ (negedge clk)
begin
    rs=rs_i;
    rt=rt_i;
    instr=instr_i;
    jump_const=jump_const_i;
    rd=rd_i;
    pc=pc_i;
end
endmodule

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

```
 else
 begin
  rs_selector_o = 0;
 end
 rt_selector_o = 0;
end
11: begin
 if(rs_i==0)
 begin
  rs_selector_o = 2;
 end
 else if(rs_i == rd_i)
 begin
  rs_selector_o = 1;
 end
 else
 begin
  rs_selector_o = 0;
 end
 rt_selector_o = 0;
end
12: begin
 rs_selector_o = 0;
 rt_selector_o = 0;
end
13: begin
 if(rs_i==0)
 begin
  rs_selector_o = 2;
 end
 else if(rs_i == rd_i)
 begin
  rs_selector_o = 1;
 end
 else
 begin
  rs_selector_o = 0;
 end
 if(rt_i==0)
 begin
  rt_selector_o = 2;
 end
 else if(rt_i == rd_i)
 begin
  rt_selector_o = 1;
 end
 else
 begin
  rt_selector_o = 0;
 end
end
14: begin
 if(rs_i==0)
 begin
  rs_selector_o = 2;
 end
 else if(rs_i == rd_i)
 begin
  rs_selector_o = 1;
 end
 else
 begin
  rs_selector_o = 0;
 end
 rt_selector_o = 0;
end
15: begin
 if(rs_i==0)
 begin
  rs_selector_o = 2;
 end
 else if(rs_i == rd_i)
 begin
  rs_selector_o = 1;
 end
 else
 begin
  rs_selector_o = 0;
 end
 rt_selector_o = 0;
end
default: begin
 if(rs_i==0 && rd_i==0)
 begin
  rs_selector_o = 2;
 end
 else if(rs_i == rd_i)
 begin
  rs_selector_o = 1;
 end
 else
 begin
  rs_selector_o = 0;
 end
 if(rt_i==0)
 begin
    rt_selector_o = 2;
   end
   else if(rt_i == rd_i)
   begin
    rt_selector_o = 1;
   end
   else
   begin
    rt_selector_o = 0;
   end
  end
 endcase
 end
 else
 begin
  rt_selector_o=0;
  rs_selector_o=0;
 end
end


endmodule

module
hazard_detection_unit (
 input logic taken_i,
 input logic hazard_i,
 input logic [7:0] instr_i,
 output logic hazard_o
);

always_comb
begin
 if(taken_i==0)
 begin
  hazard_o = 0;
 end
 else
 begin
  if(hazard_i==0)
  begin
   hazard_o = 1;
  end
  else
  begin
   hazard_o = 0;
  end
 end
end


endmodule

module instruction_decoder
(
 input [7:0] instr_i,
 output logic [3:0] rs_o,
 output logic [3:0] rd_o,
 output logic [3:0]
write_reg_o,
 output byte jump_const_o
);

parameter c = 0;
parameter r1 = 1;
parameter r2 = 2;
parameter r3 = 3;
parameter r4 = 4;
parameter r5 = 5;
parameter r6 = 6;
parameter r7 = 7;
parameter r8 = 8;
parameter temp = 9;

parameter jump_const0 =
-19;
parameter jump_const1 =
-17;
parameter jump_const2 =
-16;
parameter jump_const3 =
-14;
parameter jump_const4 =
-9;
parameter jump_const5 =
-8;
parameter jump_const6 =
-6;
parameter jump_const7 = 2;
parameter jump_const8 = 6;
parameter jump_const9 =
16;
parameter jump_const10 =
17;
parameter jump_const11 =
33;
parameter jump_const12 = 62;
parameter jump_const13 = 1;
parameter jump_const14 = 1;
parameter jump_const15 = 1;

// Decodes instruction for
write_reg_o
always_comb
begin
 case(instr_i[7:4])
 4'b0000: begin
  write_reg_o = c;
 end
 4'b0001: begin
   case(instr_i[1:0])
    2'b00: begin
     write_reg_o = r1;
     end
    2'b01: begin
     write_reg_o = r3;
     end
    2'b10: begin
     write_reg_o = r4;
     end
    2'b11: begin
     write_reg_o = r5;
     end
    default: begin
     write_reg_o = r1;
    end
  endcase
 end
 4'b0010: begin
   case(instr_i[3:2])
    2'b00: begin
     write_reg_o = r1;
     end
    2'b01: begin
     write_reg_o = r2;
     end
    2'b10: begin
     write_reg_o = r6;
     end
    2'b11: begin
     write_reg_o = r8;
     end
    default: begin
     write_reg_o = r1;
    end
  endcase
end
4'b0011: begin
  case(instr_i[1:0])
    2'b00: begin
      write_reg_o = r2;
      end
    2'b01: begin
      write_reg_o = r4;
      end
    2'b10: begin
      write_reg_o = r7;
      end
    2'b11: begin
      write_reg_o = r8;
      end
    default: begin
      write_reg_o = r2;
    end
  endcase
end
4'b0100: begin
  case(instr_i[1:0])
    2'b00: begin
      write_reg_o = r1;
      end
    2'b01: begin
      write_reg_o = r3;
      end
    2'b10: begin
      write_reg_o = r7;
      end
    2'b11: begin
      write_reg_o = r8;
      end
    default: begin
      write_reg_o = r1;
    end
  endcase
end
4'b0101: begin
  write_reg_o = c;
end
4'b0110: begin
  write_reg_o = c;
end
4'b0111: begin
  write_reg_o = temp;
end
4'b1000: begin
case(instr_i[3])
1'b0: begin
  case(instr_i[2:0])
    3'b000: begin
      write_reg_o = r1;
      end
    3'b001: begin
      write_reg_o = r2;
      end
    3'b010: begin
      write_reg_o = r3;
      end
    3'b011: begin
      write_reg_o = r4;
      end
      3'b100: begin
      write_reg_o = r5;
      end
    3'b101: begin
      write_reg_o = r6;
      end
    3'b110: begin
      write_reg_o = r7;
      end
    3'b111: begin
      write_reg_o = r8;
      end
    default: begin
      write_reg_o = r1;
    end
  endcase
  end
 1'b1: begin
  write_reg_o = c;
end
endcase
end
4'b1001: begin
  case(instr_i[2:0])
    3'b000: begin
      write_reg_o = r1;
      end
    3'b001: begin
      write_reg_o = r2;
      end
    3'b010: begin
      write_reg_o = r3;
      end
    3'b011: begin
      write_reg_o = r4;
      end
      3'b100: begin
      write_reg_o = r5;
      end
    3'b101: begin
      write_reg_o = r6;
      end
    3'b110: begin
      write_reg_o = r7;
      end
    3'b111: begin
      write_reg_o = r8;
      end
    default: begin
      write_reg_o = r1;
    end
  endcase
end
4'b1010: begin
  write_reg_o = c;
end
4'b1011: begin
  case(instr_i[2:0])
    3'b000: begin
      write_reg_o = r1;
      end
    3'b001: begin
      write_reg_o = r2;
      end
    3'b010: begin
      write_reg_o = r3;
      end
    3'b011: begin
      write_reg_o = r4;
      end
      3'b100: begin
      write_reg_o = r5;
      end
    3'b101: begin
      write_reg_o = r6;
      end
    3'b110: begin
      write_reg_o = r7;
      end
    3'b111: begin
      write_reg_o = r8;
      end
    default: begin
      write_reg_o = r1;
    end
  endcase
end
4'b1100: begin
  write_reg_o = c;
end
4'b1101: begin
  case(instr_i[3:2])
    2'b00: begin
      write_reg_o = r1;
      end
    2'b01: begin
      write_reg_o = r2;
      end
    2'b10: begin
      write_reg_o = r6;
      end
    2'b11: begin
      write_reg_o = r8;
      end
   endcase
end
4'b1110: begin
  case(instr_i[2:0])
    3'b000: begin
      write_reg_o = r1;
      end
    3'b001: begin
      write_reg_o = r2;
      end
    3'b010: begin
      write_reg_o = r3;
      end
    3'b011: begin
      write_reg_o = r4;
      end
      3'b100: begin
      write_reg_o = r5;
      end
    3'b101: begin
      write_reg_o = r6;
      end
      3'b110: begin
        write_reg_o = r7;
        end
      3'b111: begin
        write_reg_o = r8;
        end
      default: begin
        write_reg_o = r1;
      end
    endcase
  end
  4'b1111: begin
    case(instr_i[1:0])
      2'b00: begin
        write_reg_o = r2;
        end
      2'b01: begin
        write_reg_o = r3;
        end
      2'b10: begin
        write_reg_o = r4;
        end
      2'b11: begin
        write_reg_o = r8;
        end
      default: begin
        write_reg_o = r2;
      end
    endcase
  end
  default: begin
    write_reg_o = c;
  end
  endcase
end

//Decodes instruction for rs_o
always_comb
begin
   case(instr_i[7:4])
  4'b0000: begin
    rs_o = r1;
  end
  4'b0001: begin
     case(instr_i[3:2])
      2'b00: begin
        rs_o = c;
        end
    2'b01: begin
      rs_o = r1;
      end
    2'b10: begin
      rs_o = r3;
      end
    2'b11: begin
      rs_o = r4;
      end
    default: begin
      rs_o = r1;
    end
  endcase
end
4'b0010: begin
   case(instr_i[3:2])
    2'b00: begin
      rs_o = r1;
      end
    2'b01: begin
      rs_o = r2;
      end
    2'b10: begin
      rs_o = r6;
      end
    2'b11: begin
      rs_o = r8;
      end
    default: begin
      rs_o = r1;
    end
  endcase
end
4'b0011: begin
   case(instr_i[3:2])
    2'b00: begin
      rs_o = r1;
      end
    2'b01: begin
      rs_o = r2;
      end
    2'b10: begin
      rs_o = r3;
      end
    2'b11: begin
      rs_o = r4;
      end
  endcase
```

```
end
4'b0100: begin
   case(instr_i[3:2])
    2'b00: begin
      rs_o = r5;
      end
    2'b01: begin
      rs_o = r6;
      end
    2'b10: begin
      rs_o = r7;
      end
    2'b11: begin
      rs_o = r8;
      end
    default: begin
      rs_o = r5;
    end
  endcase
end
4'b0101: begin
   case(instr_i[2])
  1'b0: begin
    rs_o = r1;
  end
  1'b1: begin
    rs_o = r2;
  end
  default: begin
    rs_o = r1;
  end
  endcase
end
4'b0110: begin
  case(instr_i[3])
  1'b0: begin
    case(instr_i[2])
    1'b0: begin
      rs_o = r1;
    end
    1'b1: begin
      rs_o = r2;
    end
    default: begin
      rs_o = r1;
    end
    endcase
  end
```

```
  1'b1: begin
    case(instr_i[2])
    1'b0: begin
      rs_o = r3;
    end
    1'b1: begin
      rs_o = r6;
    end
    default: begin
      rs_o = r3;
    end
    endcase
  end
  default: begin
    rs_o = r1;
  end
  endcase
end
4'b0111: begin
 case(instr_i[3])
1'b0: begin
   case(instr_i[2:0])
  3'b000: begin
     rs_o = r1;
  end
  3'b001: begin
     rs_o = r2;
  end
  3'b010: begin
     rs_o = r3;
  end
  3'b011: begin
     rs_o = r4;
  end
  3'b100: begin
     rs_o = r5;
  end
  3'b101: begin
     rs_o = r6;
  end
  3'b110: begin
     rs_o = r7;
  end
  3'b111: begin
     rs_o = r8;
  end
  default: begin
    rs_o = r1;
```

```
  end
  endcase
end
1'b1: begin
  rs_o = c;
end
default: begin
  rs_o = r1;
end
endcase
end
4'b1000: begin
case(instr_i[3])
1'b0: begin
   case(instr_i[2:0])
  3'b000: begin
     rs_o = r1;
  end
  3'b001: begin
     rs_o = r2;
  end
  3'b010: begin
     rs_o = r3;
  end
  3'b011: begin
     rs_o = r4;
  end
  3'b100: begin
     rs_o = r5;
  end
  3'b101: begin
     rs_o = r6;
  end
  3'b110: begin
     rs_o = r7;
  end
  3'b111: begin
     rs_o = r8;
  end
  default: begin
    rs_o = r1;
  end
  endcase
end
1'b1: begin
  rs_o = temp;
end
default: begin
```

```
  rs_o = r1;
end
endcase
end
4'b1001: begin
   case(instr_i[2:0])
  3'b000: begin
     rs_o = r1;
  end
  3'b001: begin
     rs_o = r2;
  end
  3'b010: begin
     rs_o = r3;
  end
  3'b011: begin
     rs_o = r4;
  end
  3'b100: begin
     rs_o = r5;
  end
  3'b101: begin
     rs_o = r6;
  end
  3'b110: begin
     rs_o = r7;
  end
  3'b111: begin
     rs_o = r8;
  end
  default: begin
    rs_o = r1;
  end
  endcase
end
4'b1010: begin
   case(instr_i[2:0])
  3'b000: begin
     rs_o = r1;
  end
  3'b001: begin
     rs_o = r2;
  end
  3'b010: begin
     rs_o = r3;
  end
  3'b011: begin
     rs_o = r4;
  end
  3'b100: begin
     rs_o = r5;
  end
  3'b101: begin
     rs_o = r6;
  end
  3'b110: begin
     rs_o = r7;
  end
  3'b111: begin
     rs_o = r8;
  end
  default: begin
    rs_o = r1;
  end
  endcase
end
4'b1011: begin
   case(instr_i[2:0])
  3'b000: begin
     rs_o = r1;
  end
  3'b001: begin
     rs_o = r2;
  end
  3'b010: begin
     rs_o = r3;
  end
  3'b011: begin
     rs_o = r4;
  end
  3'b100: begin
     rs_o = r5;
  end
  3'b101: begin
     rs_o = r6;
  end
  3'b110: begin
     rs_o = r7;
  end
  3'b111: begin
     rs_o = r8;
  end
  default: begin
    rs_o = r1;
  end
  endcase
end
4'b1100: begin
  rs_o = r1;
end
4'b1101: begin
  case(instr_i[3:2])
    2'b00: begin
      rs_o = r1;
      end
    2'b01: begin
      rs_o = r2;
      end
    2'b10: begin
      rs_o = r6;
      end
    2'b11: begin
      rs_o = r8;
      end
    default: begin
      rs_o = r8;
    end
  endcase
end
4'b1110: begin
   case(instr_i[2:0])
  3'b000: begin
     rs_o = r1;
  end
  3'b001: begin
     rs_o = r2;
  end
  3'b010: begin
     rs_o = r3;
  end
  3'b011: begin
     rs_o = r4;
  end
  3'b100: begin
     rs_o = r5;
  end
  3'b101: begin
     rs_o = r6;
  end
  3'b110: begin
     rs_o = r7;
  end
  3'b111: begin
     rs_o = r8;
    end
    default: begin
      rs_o = r1;
    end
    endcase
  end
  4'b1111: begin
     case(instr_i[1:0])
      2'b00: begin
        rs_o = r2;
        end
      2'b01: begin
        rs_o = r3;
        end
      2'b10: begin
        rs_o = r4;
        end
      2'b11: begin
        rs_o = r8;
        end
      default: begin
        rs_o = r2;
      end
    endcase
  end
  endcase
end

//Decodes instruction for
rd_o
always_comb
begin
case(instr_i[7:4])
4'b0000: begin
  rd_o = c;
end
4'b0001: begin
  case(instr_i[1:0])
  2'b00: begin
    rd_o = r1;
  end
  2'b01: begin
    rd_o = r3;
  end
  2'b10: begin
    rd_o = r4;
  end
  2'b11: begin
    rd_o = r5;
  end
  default: begin
    rd_o = r1;
  end
  endcase
end
4'b0010: begin
  case(instr_i[1:0])
  2'b00: begin
    rd_o = r3;
  end
  2'b01: begin
    rd_o = r4;
  end
  2'b10: begin
    rd_o = r5;
  end
  2'b11: begin
    rd_o = r6;
  end
  default: begin
    rd_o = r3;
  end
  endcase
end
4'b0011: begin
case(instr_i[1:0])
  2'b00: begin
    rd_o = r2;
  end
  2'b01: begin
    rd_o = r4;
  end
  2'b10: begin
    rd_o = r7;
  end
  2'b11: begin
    rd_o = r8;
  end
  default: begin
    rd_o = r2;
  end
  endcase
end
4'b0100: begin
case(instr_i[1:0])
  2'b00: begin
    rd_o = r1;
  end
  2'b01: begin
    rd_o = r3;
  end
  2'b10: begin
    rd_o = r7;
  end
  2'b11: begin
    rd_o = r8;
  end
  default: begin
    rd_o = r1;
  end
  endcase
end
4'b0101: begin
case(instr_i[1:0])
  2'b00: begin
    rd_o = r1;
  end
  2'b01: begin
    rd_o = r2;
  end
  2'b10: begin
    rd_o = r3;
  end
  2'b11: begin
    rd_o = r4;
  end
  default: begin
    rd_o = r1;
  end
  endcase
end
4'b0110: begin
case(instr_i[3])
1'b0: begin
case(instr_i[1:0])
  2'b00: begin
    rd_o = r3;
  end
  2'b01: begin
    rd_o = r4;
  end
  2'b10: begin
    rd_o = r5;
  end
```

```
 2'b11: begin
  rd_o = r7;
 end
 default: begin
  rd_o = r3;
 end
 endcase
end
1'b1: begin
case(instr_i[1:0])
 2'b00: begin
  rd_o = r1;
 end
 2'b01: begin
  rd_o = r2;
 end
 2'b10: begin
  rd_o = r3;
 end
 2'b11: begin
  rd_o = r7;
 end
 default: begin
  rd_o = r1;
 end
 endcase
end
default: begin
 rd_o = r1;
end
endcase
end
4'b0111: begin
 rd_o = r8;
end
4'b1000: begin
 rd_o = r8;
end
4'b1001: begin
 rd_o = c;
end
4'b1010: begin
 rd_o = c;
end
4'b1011: begin
 rd_o = c;
end
4'b1100: begin
 rd_o = c;
end
4'b1101: begin
  case(instr_i[1:0])
   2'b00: begin
    rd_o = r3;
    end
   2'b01: begin
    rd_o = r4;
    end
   2'b10: begin
    rd_o = r5;
    end
   2'b11: begin
    rd_o = r6;
    end
   default: begin
    rd_o = r4;
   end
  endcase
end
4'b1110: begin
 rd_o = c;
end
4'b1111: begin
 rd_o = c;
end
default: begin
 rd_o = c;
end
endcase
end

//Decodes instruction for
jump_const_o
always_comb
begin
   case(instr_i[3:0])
  4'b0000: begin
   jump_const_o =
jump_const0;
  end
  4'b0001: begin
   jump_const_o =
jump_const1;
  end
  4'b0010: begin
   jump_const_o =
jump_const2;
  end
  4'b0011: begin
   jump_const_o =
jump_const3;
  end
  4'b0100: begin
   jump_const_o =
jump_const4;
  end
  4'b0101: begin
   jump_const_o =
jump_const5;
  end
  4'b0110: begin
   jump_const_o =
jump_const6;
  end
  4'b0111: begin
   jump_const_o =
jump_const7;
  end
  4'b1000: begin
   jump_const_o =
jump_const8;
  end
  4'b1001: begin
   jump_const_o =
jump_const9;
  end
  4'b1010: begin
   jump_const_o =
jump_const10;
  end
  4'b1011: begin
   jump_const_o =
jump_const11;
  end
  4'b1100: begin
   jump_const_o =
jump_const12;
  end
  4'b1101: begin
   jump_const_o =
jump_const13;
  end
  4'b1110: begin
   jump_const_o =
jump_const14;
  end
  4'b1111: begin
   jump_const_o =
jump_const15;
  end
  default: begin
   jump_const_o =
jump_const8;
  end
  endcase
end
endmodule

module instruction_ROM (
 input logic [7:0] address_i,
 output logic [7:0]
instruction_o
 );

always_comb
 begin
  case (address_i)
   0 : instruction_o =
8'b11101101;
   1 : instruction_o =
8'b11101010;
   2 : instruction_o =
8'b11101011;
   3 : instruction_o =
8'b11101111;
  4 : instruction_o =
8'b11101100;
   5 : instruction_o =
8'b10010111;
  6 : instruction_o =
8'b10000000;
   7 : instruction_o =
8'b10010111;
  8 : instruction_o =
8'b10000001;
  9 : instruction_o =
8'b10010101;
   10 : instruction_o =
8'b10100001;
   11 : instruction_o =
8'b11001100;
   12 : instruction_o =
8'b01101101;
  13 : instruction_o =
8'b11001010;
   14 : instruction_o =
8'b10110001;
   15 : instruction_o =
8'b11001000;
   16 : instruction_o =
8'b10011001;
  17 : instruction_o =
8'b00010110;
   18 : instruction_o =
8'b00010011;
   19 : instruction_o =
8'b00011011;
   20 : instruction_o =
8'b11000101;

  21 : instruction_o =
8'b11100001;
  22 : instruction_o =
8'b00110011;
  23 : instruction_o =
8'b11110011;

   24 : instruction_o =
8'b01111000;
  25 : instruction_o =
8'b01001100;
   26 : instruction_o =
8'b11110001;
  27 : instruction_o =
8'b10001000;
   28 : instruction_o =
8'b00010001;
   29 : instruction_o =
8'b11000001;
   30 : instruction_o =
8'b00011100;
   31 : instruction_o =
8'b00010011;
   32 : instruction_o =
8'b00011011;
  33 : instruction_o =
8'b01000001;
```

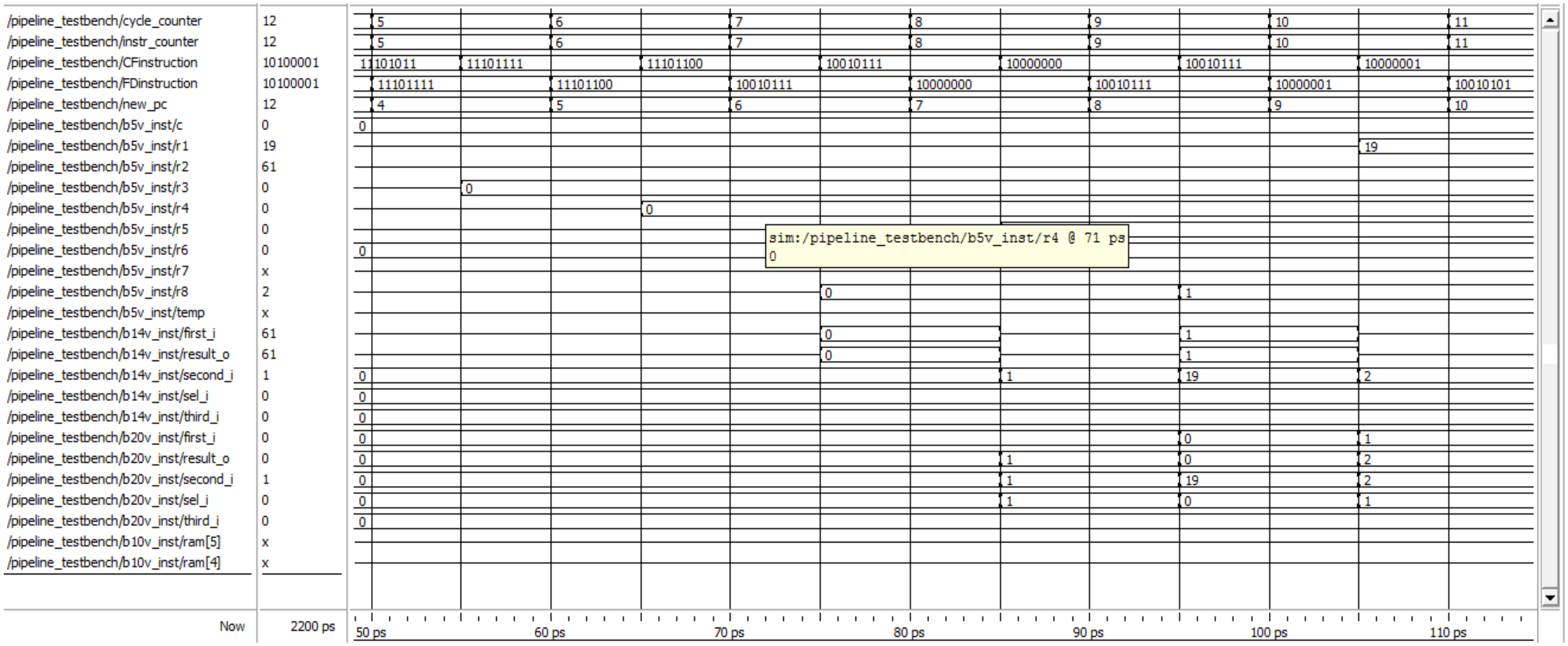
```
     34 : instruction_o =
8'b11101111;
   35 : instruction_o =
8'b10010111;
   36 : instruction_o =
8'b10010111;
   37 : instruction_o =
8'b10010111;


   38 : instruction_o =
8'b10000001;
   39 : instruction_o =
8'b10100001;
   40 : instruction_o =
8'b11001011;
   41 : instruction_o =
8'b11101011;
   42 : instruction_o =
8'b11101100;
   43 : instruction_o =
8'b01101101;
   44 : instruction_o =
8'b11001010;
   45 : instruction_o =
8'b10110001;
   46 : instruction_o =
8'b11001000;
   47 : instruction_o =
8'b10011001;
   48 : instruction_o =
8'b00010110;
   49 : instruction_o =
8'b00010011;
   50 : instruction_o =
8'b00011011;
   51 : instruction_o =
8'b11000101;
   52 : instruction_o =
8'b11100001;
   53 : instruction_o =
8'b00110011;
   54 : instruction_o =
8'b11110011;
   55 : instruction_o =
8'b01111000;
   56 : instruction_o =
8'b01001100;
   57 : instruction_o =
8'b11110001;
   58 : instruction_o =
8'b10001000;
   59 : instruction_o =
8'b00010001;
   60 : instruction_o =
8'b11000001;
   61 : instruction_o =
8'b00010110;
   62 : instruction_o =
8'b00010011;
   63 : instruction_o =
8'b00011011;


   64 : instruction_o =
8'b11101111;
   65 : instruction_o =
8'b10010111;
   66 : instruction_o =
8'b10010111;
   67 : instruction_o =
8'b10010111;
   68 : instruction_o =
8'b10010111;


   69 : instruction_o =
8'b01110100;
   70 : instruction_o =
8'b10010111;
   71 : instruction_o =
8'b01110011;
   72 : instruction_o =
8'b00000000;
   73 : instruction_o =
8'b11101111;
   74 : instruction_o =
8'b10010111;
   75 : instruction_o =
8'b10010111;
   76 : instruction_o =
8'b10010111;
   77 : instruction_o =
8'b10010111;
   78 : instruction_o =
8'b11101101;
   79 : instruction_o =
8'b01110101;
   80 : instruction_o =
8'b01110101;
   81 : instruction_o =
8'b01110101;
   82 : instruction_o =
8'b00000000;


   //program 2
   83 : instruction_o =
8'b11101101;
   84 : instruction_o =
8'b11101111;
   85 : instruction_o =
8'b10010111;
   86 : instruction_o =
8'b11110111;
   87 : instruction_o =
8'b10010111;
   88 : instruction_o =
8'b10010111;
   89 : instruction_o =
8'b10000001;
   90 : instruction_o =
8'b11111000;
   91 : instruction_o =
8'b11100001;
   92 : instruction_o =
8'b11100001;
   93 : instruction_o =
8'b11100001;
   94 : instruction_o =
8'b11100001;
   95 : instruction_o =
8'b11101100;
   96 : instruction_o =
8'b11101010;
   97 : instruction_o =
8'b10010010;
   98 : instruction_o =
8'b01001110;
   99 : instruction_o =
8'b10011110;
   100 : instruction_o =
8'b11111101;
   101 : instruction_o =
8'b00111011;
   102 : instruction_o =
8'b11100111;
   103 : instruction_o =
8'b11101000;
   104 : instruction_o =
8'b10100010;
   105 : instruction_o =
8'b11001001;
   106 : instruction_o =
8'b10000011;
   107 : instruction_o =
8'b01010111;
   108 : instruction_o =
8'b11001000;
   109 : instruction_o =
8'b11100011;
   110 : instruction_o =
8'b10010000;
   111 : instruction_o =
8'b01100011;
   112 : instruction_o =
8'b11001000;
   113 : instruction_o =
8'b11000110;
   114 : instruction_o =
8'b10010100;
   115 : instruction_o =
8'b10011010;
   116 : instruction_o =
8'b10010111;
   117 : instruction_o =
8'b11000011;
   118 : instruction_o =
8'b10010111;
   119 : instruction_o =
8'b10011010;
   120 : instruction_o =
8'b11000001;
   121 : instruction_o =
8'b01001011;
   122 : instruction_o =
8'b10010111;
   123 : instruction_o =
8'b10010111;
   124 : instruction_o =
8'b01110100;
   125 : instruction_o =
8'b00000000;

   //Program 3
   126 : instruction_o =
8'b11101111;//srl
   127 : instruction_o =
8'b10010111;//addi
   128 : instruction_o =
8'b11110111;//sll
   129 : instruction_o =
8'b11110011;//sll
   130 : instruction_o =
8'b10011111;//addi
   131 : instruction_o =
8'b01001101;//movh
   132 : instruction_o =
8'b10010010;//addi
   133 : instruction_o =
8'b10010010;//addi
   134 : instruction_o =
8'b11110001;//sll
   135 : instruction_o =
8'b10010010;//addi
   136 : instruction_o =
8'b00111001;//movl
   137 : instruction_o =
8'b11101000;
   138 : instruction_o =
8'b11101001;
   139 : instruction_o =
8'b11101111;
   140 : instruction_o =
8'b10010111;//r8++
   141 : instruction_o =
8'b10010111;//r8++
   142 : instruction_o =
8'b11111111;
   143 : instruction_o =
8'b10000100;
   144 : instruction_o =
8'b10010111;//r8++
   145 : instruction_o =
8'b10000101;// ld
   146 : instruction_o =
8'b10010001;//j++
   147 : instruction_o =
8'b00101010;//diff
   148 : instruction_o =
8'b01000110;//movh
   149 : instruction_o =
8'b10010000;//r1++
   150 : instruction_o =
8'b10010111;//r8++
   151 : instruction_o =
8'b10000101;//ld
   152 : instruction_o =
8'b00101010;//diff
   153 : instruction_o =
8'b01101111;//bge
   154 : instruction_o =
8'b11000111;//jump
   155 : instruction_o =
8'b01000110;//movh
   156 : instruction_o =
8'b10010001;//addi
   157 : instruction_o =
8'b01011110;//be
   158 : instruction_o =
8'b11000111;//jump
   159 : instruction_o =
8'b11000100;//jump
   160 : instruction_o =
8'b10011011;//addi
   161 : instruction_o =
8'b11011101;//udiff
   162 : instruction_o =
8'b10000100;//ld
   163 : instruction_o =
8'b11101001;//srl
   164 : instruction_o =
8'b11101000;//srl
   165 : instruction_o =
8'b10011010;//addi
   166 : instruction_o =
8'b01100000;
   167 : instruction_o =
8'b11000111;
   168 : instruction_o =
8'b11000000;
   169 : instruction_o =
8'b11101111;
   170 : instruction_o =
8'b10010111;
   171 : instruction_o =
8'b10010111;
   172 : instruction_o =
8'b11111111;
   173 : instruction_o =
8'b10011111;
   174 : instruction_o =
8'b01110110;
   175 : instruction_o =
8'b11010000;
     default: instruction_o =
8'b00000000;
   endcase
  end
endmodule

// Quartus II Verilog
Template
// Single port RAM with
single read/write address
and initial contents
// specified with an initial
block

module memory_data_RAM
#(parameter
ADDR_WIDTH=8)
(
  input [7:0] data,
  input [(ADDR_WIDTH-1):0]
addr,
  input writemem,
readmem, clk,
  output [7:0] q
);

  // Declare the RAM
variable
  reg [7:0]
ram[2**ADDR_WIDTH-1:0];

  // Variable to hold the
registered read address
  reg [ADDR_WIDTH-1:0]
addr_reg;

  initial
  begin
ram[1] = 8'd19;
ram[2] = 8'd61;
ram[3] = 8'd23;

ram[6] = 8'h05;

ram[32] = 8'h12;
ram[33] = 8'h34;
ram[34] = 8'h56;
ram[35] = 8'h78;
ram[36] = 8'h9a;
ram[37] = 8'hbc;
ram[38] = 8'hde;
ram[39] = 8'hf0;
ram[40] = 8'h11;
ram[41] = 8'h22;
ram[42] = 8'h33;
ram[43] = 8'h44;
ram[44] = 8'h55;
ram[45] = 8'h66;
ram[46] = 8'h77;
ram[47] = 8'h88;
ram[48] = 8'h99;
ram[49] = 8'haa;
ram[50] = 8'hde;
ram[51] = 8'had;
ram[52] = 8'hbe;
ram[53] = 8'hef;
ram[54] = 8'h02;
ram[55] = 8'h46;
ram[56] = 8'h8a;
ram[57] = 8'hce;
ram[58] = 8'h13;
ram[59] = 8'h57;
ram[60] = 8'h9a;
ram[61] = 8'hcf;
ram[62] = 8'h39;
ram[63] = 8'haa;
ram[64] = 8'hbc;
ram[65] = 8'hf1;
ram[66] = 8'h00;
ram[67] = 8'hf0;
ram[68] = 8'h57;
ram[69] = 8'h68;
ram[70] = 8'h9a;
ram[71] = 8'h56;
ram[72] = 8'hbe;
ram[73] = 8'hde;
  ram[74] = 8'hfe;
  ram[75] = 8'hed;
  ram[76] = 8'h03;
  ram[77] = 8'h69;
  ram[78] = 8'hcf;
  ram[79] = 8'h25;
  ram[80] = 8'h8a;
  ram[81] = 8'hd1;
  ram[82] = 8'h47;
  ram[83] = 8'had;
  ram[84] = 8'h15;
  ram[85] = 8'h9d;
  ram[86] = 8'h01;
  ram[87] = 8'h23;
  ram[88] = 8'h34;
  ram[89] = 8'h56;
  ram[90] = 8'h78;
  ram[91] = 8'h9a;
  ram[92] = 8'hbc;
  ram[93] = 8'hde;
  ram[94] = 8'hf0;
  ram[95] = 8'h00;


  ram[128] =  8'd2;
  ram[129] = -8'd23;
  ram[130] = -8'd33;
  ram[131] =  8'd63;
  ram[132] =  8'd18;
  ram[133] =  8'd23;
  ram[134] = -8'd52;
  ram[135] =  8'd28;
  ram[136] =  8'd9;
  ram[137] = -8'd8;
  ram[138] =  8'd6;
  ram[139] =  8'd105;
  ram[140] =  8'd94;
  ram[141] = -8'd83;
  ram[142] =  8'd55;
  ram[143] =  8'd100;
  ram[144] = -8'd3;
  ram[145] = -8'd72;
  ram[146] =  8'd65;
  ram[147] = -8'd47;


end

always @ (posedge clk)
  begin
    // Write
    if (writemem)
      ram[addr] <= data;
  end


  // Continuous assignment
implies read returns NEW data.
  // This is the natural behavior of
the TriMatrix memory
  // blocks in Single Port mode.
  assign q = (readmem)?
ram[addr] : 8'bxxxxxxxx;

endmodule

module next_pc_logic
(
  input [7:0] instr_i,
  input [7:0] oldpc_i,
  input [7:0] rs_data_i,
  input [7:0] rd_data_i,
  input byte jump_const_i,
  output logic [7:0] newpc_o,
  output logic pc_selector_o
);

always_comb
begin
  case(instr_i[7:4])
  4'b0000: begin
    newpc_o = oldpc_i;
    pc_selector_o=1;
  end
  4'b0001: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  4'b0010: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  4'b0011: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  4'b0100: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  4'b0101: begin
  if(instr_i[3]==0)
  begin
    if(rs_data_i[3:0] ==
rd_data_i[3:0])
    begin
      newpc_o = oldpc_i + 1;
      pc_selector_o=0;
    end
    else
    begin
      newpc_o = oldpc_i + 2;
      pc_selector_o=1;
    end
  end
  else
  begin
    if(rs_data_i == rd_data_i)
    begin
      newpc_o = oldpc_i + 1;
      pc_selector_o=0;
    end
    else
    begin
      newpc_o = oldpc_i + 2;
      pc_selector_o=1;
    end
  end
  end
  4'b0110: begin
    if(rs_data_i >= rd_data_i)
    begin
      newpc_o = oldpc_i + 1;
      pc_selector_o=0;
    end
    else
    begin
      newpc_o = oldpc_i + 2;
      pc_selector_o=1;
    end
  end
  4'b0111: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  4'b1000: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  4'b1001: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  4'b1010: begin
    if(rs_data_i == 0)
    begin
      newpc_o = oldpc_i + 1;
      pc_selector_o=0;
    end
    else
    begin
      newpc_o = oldpc_i + 2;
      pc_selector_o=1;
    end
  end
  4'b1011: begin
    if(rs_data_i[0] == 1'b0)
    begin
      newpc_o = oldpc_i + 1;
      pc_selector_o=0;
    end
    else
    begin
      newpc_o = oldpc_i + 2;
      pc_selector_o=1;
    end
  end
  4'b1100: begin
    newpc_o = oldpc_i +
jump_const_i;
    pc_selector_o=1;
  end
  4'b1101: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  4'b1110: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  4'b1111: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
  default: begin
    newpc_o = oldpc_i + 1;
    pc_selector_o=0;
  end
endcase
end
endmodule

module program_counter
(
  input [7:0] newpc_i,
  input clock,
  input wenable_i,
  input reset_i,
  output [7:0] pc_o
);

reg [7:0] pc, pcnext;


assign pc_o = pc;

always_comb
  begin
  if (reset_i) begin
    pcnext = 0;
  end else if (wenable_i)
    pcnext = newpc_i;
  else
    pcnext = pc;
  end


always_ff @(posedge clock)
  pc <= pcnext;


endmodule

module register_file
(
  input [3:0] rs_i,
  input [3:0] rt_i,
  input [3:0] write_reg_i,
  input [7:0] reg_data_i,
  input [7:0] c_data_i,
  input clock,
  input wenable_i,
  output logic [7:0]
rs_data_o,
  output logic [7:0]
rd_data_o
);

reg[7:0] c, r1, r2, r3, r4, r5,
r6, r7, r8, temp;

//Read data from register rs
always_comb
begin
  case(rs_i)
  0: begin
    rs_data_o = c;
  end
  1: begin
    rs_data_o = r1;
  end
  2: begin
    rs_data_o = r2;
  end
  3: begin
    rs_data_o = r3;
  end
  4: begin
    rs_data_o = r4;
  end
  5: begin
    rs_data_o = r5;
  end
  6: begin
    rs_data_o = r6;
  end
  7: begin
    rs_data_o = r7;
  end
  8: begin
    rs_data_o = r8;
  end
  9: begin
    rs_data_o = temp;
  end
  default: begin
    rs_data_o = r1;
  end
endcase
end


// Read data from register rd
always_comb
begin
  case(rt_i)
  0: begin
    rd_data_o = c;
  end
  1: begin
    rd_data_o = r1;
  end
  2: begin
    rd_data_o = r2;
  end
  3: begin
    rd_data_o = r3;
  end
  4: begin
    rd_data_o = r4;
  end
  5: begin
    rd_data_o = r5;
  end
  6: begin
    rd_data_o = r6;
  end
  7: begin
    rd_data_o = r7;
  end
  8: begin
    rd_data_o = r8;
  end
  9: begin
    rd_data_o = temp;
  end
  default: begin
    rd_data_o = r1;
  end
endcase
end


// Write
always_ff @ (negedge clock)
begin
if(wenable_i)
```

```
begin
 case(write_reg_i)
 0: begin
   c = reg_data_i;
 end
 1: begin
   r1 = reg_data_i;
   c = c_data_i;
 end
 2: begin
   r2 = reg_data_i;
   c = c_data_i;
 end
 3: begin
   r3 = reg_data_i;
   c = c_data_i;
 end
 4: begin
   r4 = reg_data_i;
   c = c_data_i;
 end
 5: begin
   r5 = reg_data_i;
   c = c_data_i;
 end
 6: begin
   r6 = reg_data_i;
   c = c_data_i;
 end
 7: begin
   r7 = reg_data_i;
   c = c_data_i;
 end
 8: begin
   r8 = reg_data_i;
   c = c_data_i;
 end
 9: begin
   temp = reg_data_i;
 end
 13: begin
   r2 = reg_data_i;
   c = c_data_i;
 end
 default: begin
   r1 = reg_data_i;
 end
 endcase

end
else
begin
// Don't write

end
end

endmodule

module
rs_three_to_one_mux (
  input logic [1:0] sel_i,
  input  logic [7:0] first_i,
  input  logic [7:0] second_i,
  input  logic [7:0] third_i,
  output logic [7:0] result_o
);

always_comb
begin
case(sel_i)
  0: result_o = first_i;
  1:  result_o = second_i;
  2:  result_o=third_i;
  default: result_o=first_i;
endcase
end
endmodule

module
rt_three_to_one_mux (
  input logic [1:0] sel_i,
  input  logic [7:0] first_i,
  input  logic [7:0] second_i,
  input  logic [7:0] third_i,
  output logic [7:0] result_o
);

always_comb
begin
case(sel_i)
  0: result_o = first_i;
  1:  result_o = second_i;
  2:  result_o=third_i;
  default: result_o=first_i;
endcase
end
endmodule

module
pipeline_testbench();
// Declare inputs as regs and
outputs as wires
reg init;
reg clk;
reg done;

reg [15:0] cycle_counter;
reg [7:0] FDnew_pc;
reg [7:0] FDpc;
reg [7:0] CFpc;
reg [7:0] FDinstruction;
reg [7:0] CFinstruction;
reg [3:0] FDrs;
reg [3:0] FDrt;
reg [3:0] CFrs;
reg [3:0] CFrt;
reg [3:0] EXrs;
reg [3:0] EXrt;
reg [3:0] FDrd;
reg [3:0] CFrd;
reg [3:0] EXrd;
reg [3:0] FDwrite_reg;
reg [3:0] EXwrite_reg;
reg [7:0] EXreg_data;
reg [7:0] EXc_data;
reg CFreg_wenable;
reg EXreg_wenable;
reg [7:0] CFrs_data;
reg [7:0] CFrt_data;
reg [7:0] EXrs_data;
reg [7:0] EXrd_data;
reg [7:0] EXrt_data;
reg [3:0] CFalu_opt;
reg [3:0] EXalu_opt;
reg [7:0] EXresult_write;
reg [7:0] EXresult_alu;
reg [7:0] EXq;
reg [1:0]
CFwrite_register_selector;
reg [1:0]
EXwrite_register_selector;
reg EXcarry_out_bit;
byte FDjump_const;
byte CFjump_const;
reg CFmem_read;
reg CFmem_write;
reg EXmem_read;
reg EXmem_write;
reg CFtaken;
reg FDhazard;
reg CFhazard;
reg CFpc_sel;
reg CFrs_sel;
reg CFrt_sel;

reg taken;
reg[7:0] EXinstruction;
reg pc_selector;
reg [1:0] CFrs_selector;
reg [1:0] CFrt_selector;
reg [7:0] CFrs_result;
reg [7:0] CFrt_result;
reg [7:0] CFnew_pc;
reg [9:0] instr_counter;

reg [7:0] new_pc;
reg [7:0] pc;
reg [7:0] instruction;
reg [3:0] rs;
reg [3:0] rt;
reg [3:0] write_reg;
reg [7:0] reg_data;
reg [7:0] c_data;
reg reg_wenable;
reg [7:0] rs_data;
reg [7:0] rd_data;
reg [3:0] alu_opt;
reg [7:0] result;
reg [7:0] q;
reg [1:0]
write_register_selector;
reg carry_out_bit;
byte jump_const;
reg mem_read;
reg mem_write;


initial begin
  clk = 1;
  init = 0;
  done = 0;
  new_pc = 0; // Set PC to
program
  FDhazard=0;
  CFhazard=0;
  FDnew_pc=0;
  instr_counter=0;


   // initial value of new_pc
changes depending on program
  #10 init = 1; new_pc = 126;
cycle_counter = 0;


end

// Clock generator
always begin
   #5  clk = ~clk; if(clk) begin
cycle_counter++; end
       if(clk &&!CFhazard)begin
instr_counter++; end// Toggle
clock every 5 ticks
           // this makes the clock
cycle 10 ticks



end

program_counter  b2v_inst(
  .clock(clk),
  .wenable_i(init),
  .reset_i(0),
  .newpc_i(new_pc),
  .pc_o(FDpc));


//alu instance
alu b3v_inst(
  .rs_data_i(EXrs_data),
  .rd_data_i(EXrt_data),
  .alu_opt_i(EXalu_opt),
  .result_o(EXresult_alu),
   .carry_bit_o(EXcarry_out_bit));

// instruction_decoder instance
instruction_decoder  b1v_inst(
  .instr_i(FDinstruction),
  .rs_o(FDrs),
  .rd_o(FDrt),
  .write_reg_o(FDrd),
  .jump_const_o(FDjump_const));

// 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),
  .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),
  .rd_data_i(CFrt_result),
  .jump_const_i(CFjump_const),
  .newpc_o(CFnew_pc),
  .pc_selector_o(pc_selector)
);


// instruction_ROM instance
instruction_ROM b7v_inst (
  .address_i(FDpc),
  .instruction_o(FDinstruction)
);

// control unit instance
control_unit b8v_inst (
  .instr_i(CFinstruction),
  .rs_data_i(CFrs_result),
  .alu_opt_o(CFalu_opt),

.reg_write_o(CFreg_wenable),

.write_reg_selector_o(CFwrite_register_selector),

.mem_read_o(CFmem_read),

.mem_write_o(CFmem_write)
  );

 //write_reg_mux instance
write_reg_mux b9v_inst (

.write_reg_selector_i(EXwrite_register_selector),
  .alu_result_i(EXresult_alu),
  .reg_result_i(EXrs_data),
  .mem_result_i(EXq),
  .result_o(EXresult_write)
);


memory_data_RAM
b10v_inst
(
  .data(EXrs_data),
  .addr(EXrt_data),
  .writemem(EXmem_write),
  .readmem(EXmem_read),
  .clk(clk),
  .q(EXq)
);

carry_out_extender
b11v_inst(

.carry_out_i(EXcarry_out_bit),
  .carry_out_o(EXc_data)
);

two_to_one_mux
b12v_inst(
  .sel_i(pc_selector),
  .first_i(FDnew_pc),
  .second_i(CFnew_pc),
  .result_o(new_pc)
);

adder b13v_inst(
  .pc_i(FDpc),
  .new_pc_o(FDnew_pc)
);

//add in hazard
FD_CF b18v_inst(
  .rs_i(FDrs),
  .rt_i(FDrt),
  .instr_i(FDinstruction),

.jump_const_i(FDjump_const),
  .rd_i(FDrd),
  .pc_i(FDpc),
  .clk(clk),
  .rs_o(CFrs),
  .rt_o(CFrt),
  .instr_o(CFinstruction),

.jump_const_o(CFjump_const),
  .rd_o(CFrd),
  .pc_o(CFpc)
);

forwarding_unit b17v_inst(
  .CFinstr_i(CFinstruction),
  .rs_i(CFrs),
  .rt_i(CFrt),
  .rd_i(EXrd),

.EXinstruct_i(EXinstruction),

.reg_write_i(EXreg_wenable
),

.rs_selector_o(CFrs_selector
),
  .rt_selector_o
(CFrt_selector)
);

rs_three_to_one_mux
b14v_inst(
  .sel_i(CFrs_selector),
  .first_i(CFrs_data),
  .second_i(EXresult_write),
  .third_i(EXc_data),
  .result_o(CFrs_result)
);
rt_three_to_one_mux b20v_inst(
  .sel_i(CFrt_selector),
  .first_i(CFrt_data),
  .second_i(EXresult_write),
  .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),

.write_reg_selector_i(CFwrite_register_selector),
   .mem_read_i(CFmem_read),
   .mem_write_i(CFmem_write),
  .clk(clk),
  .instruct_i(CFinstruction),
  .rs_data_o(EXrs_data),
  .rt_data_o(EXrt_data),
  .alu_opt_o(EXalu_opt),
  .rd_o(EXrd),
  .reg_write_o(EXreg_wenable),

.write_reg_selector_o(EXwrite_register_selector),
   .mem_read_o(EXmem_read),
   .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):



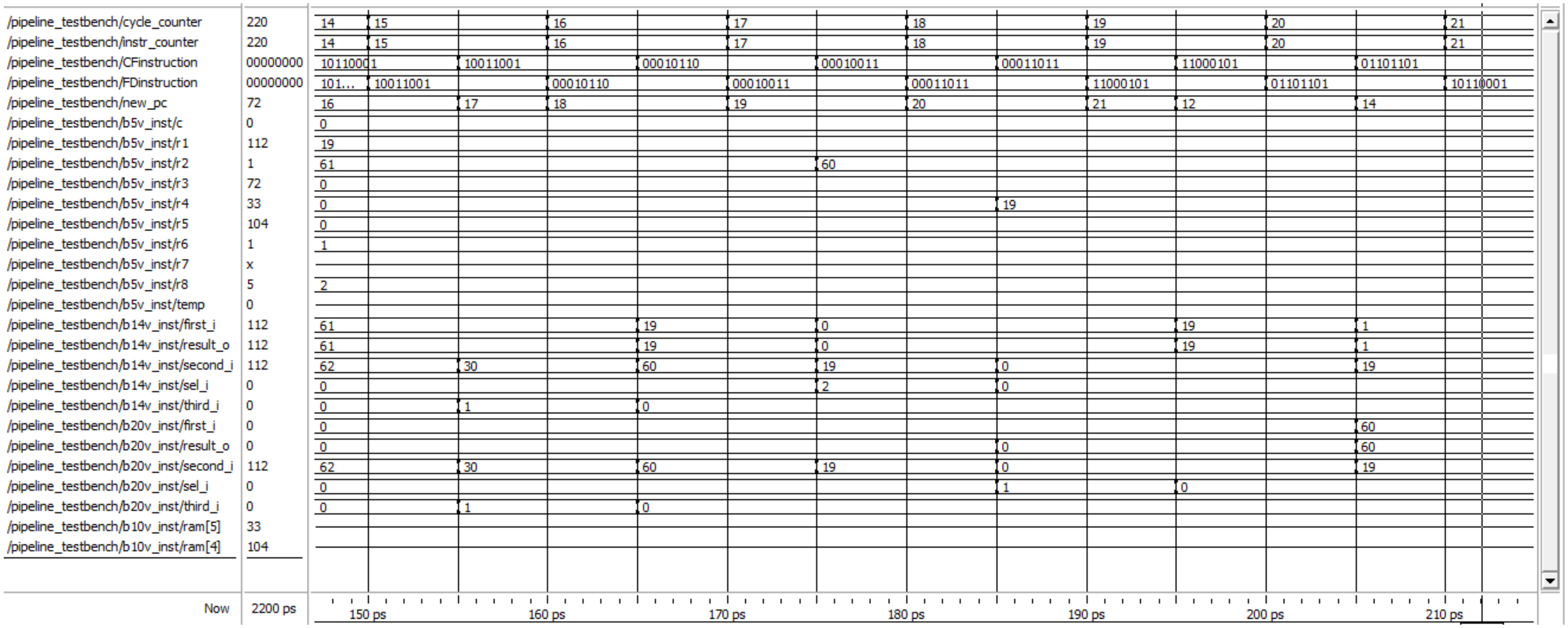
/pipeline_testbench/cycle_counter	12
/pipeline_testbench/instr_counter	12
/pipeline_testbench/CFinstruction	10100001
/pipeline_testbench/FDinstruction	10100001
/pipeline_testbench/new_pc	12
/pipeline_testbench/b5v_inst/c	0
/pipeline_testbench/b5v_inst/r1	19
/pipeline_testbench/b5v_inst/r2	61
/pipeline_testbench/b5v_inst/r3	0
/pipeline_testbench/b5v_inst/r4	0
/pipeline_testbench/b5v_inst/r5	0
/pipeline_testbench/b5v_inst/r6	0
/pipeline_testbench/b5v_inst/r7	x
/pipeline_testbench/b5v_inst/r8	2
/pipeline_testbench/b5v_inst/temp	x
/pipeline_testbench/b14v_inst/first_i	61
/pipeline_testbench/b14v_inst/result_o	61
/pipeline_testbench/b14v_inst/second_i	1
/pipeline_testbench/b14v_inst/sel_i	0
/pipeline_testbench/b14v_inst/third_i	0
/pipeline_testbench/b20v_inst/first_i	0
/pipeline_testbench/b20v_inst/result_o	0
/pipeline_testbench/b20v_inst/second_i	1
/pipeline_testbench/b20v_inst/sel_i	0
/pipeline_testbench/b20v_inst/third_i	0
/pipeline_testbench/b10v_inst/ram[5]	x
/pipeline_testbench/b10v_inst/ram[4]	x
Now	2200 ps

sim:/pipeline_testbench/b5v_inst/r4 @ 71 ps
0

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

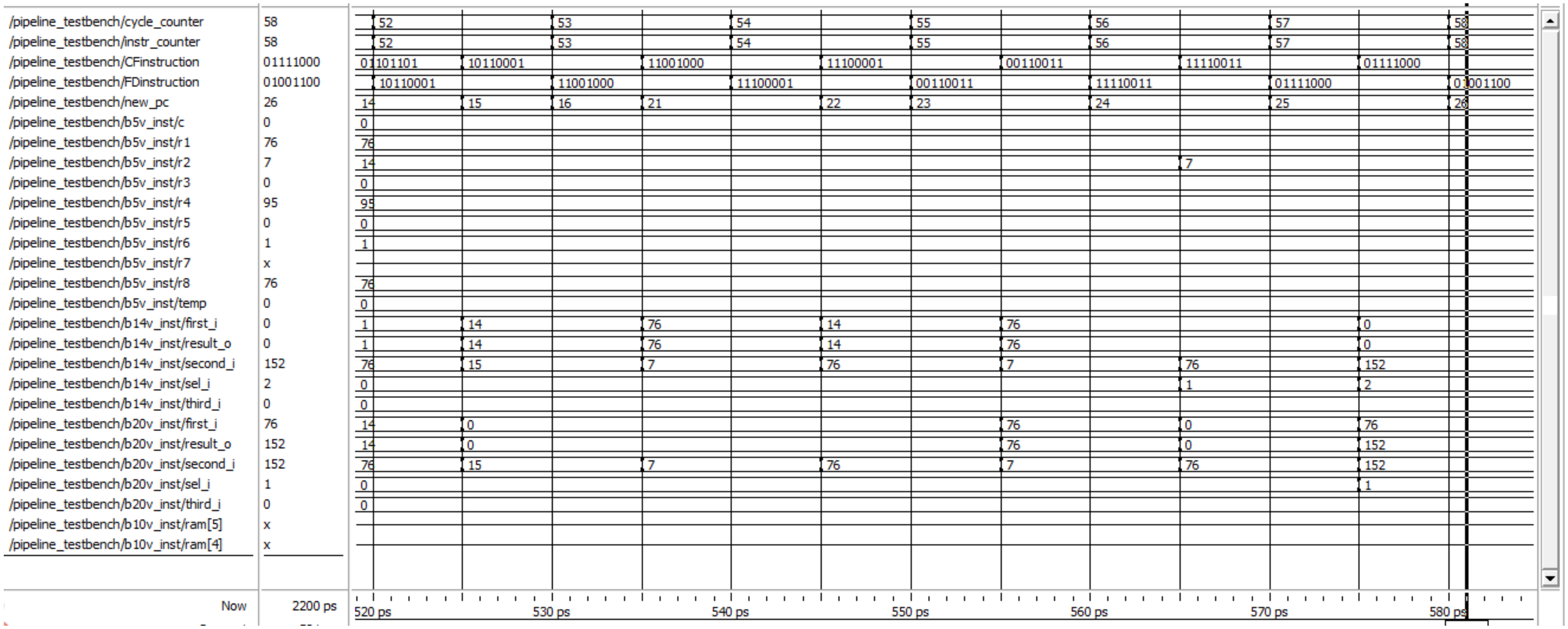
6. FD: PC=instruction for increment $r8
   CF: Preparing data to execute shift $r5 right logical by 8
   EX: Executing shift $r4 logical by 8, clearing $r4, $r4=0
7. FD: PC=instruction for loading memory address $r8 to $r1
   CF: Preparing data to increment $r8
   EX: Executing shift $r5 right logical 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 $r8 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
/pipeline_testbench/instr_counter 220
/pipeline_testbench/CFinstruction 00000000
/pipeline_testbench/FDinstruction 00000000
/pipeline_testbench/new_pc 72
/pipeline_testbench/b5v_inst/c 0
/pipeline_testbench/b5v_inst/r1 112
/pipeline_testbench/b5v_inst/r2 1
/pipeline_testbench/b5v_inst/r3 72
/pipeline_testbench/b5v_inst/r4 33
/pipeline_testbench/b5v_inst/r5 104
/pipeline_testbench/b5v_inst/r6 1
/pipeline_testbench/b5v_inst/r7 x
/pipeline_testbench/b5v_inst/r8 5
/pipeline_testbench/b5v_inst/temp 0
/pipeline_testbench/b14v_inst/first_i 112
/pipeline_testbench/b14v_inst/result_o 112
/pipeline_testbench/b14v_inst/second_i 112
/pipeline_testbench/b14v_inst/sel_i 0
/pipeline_testbench/b14v_inst/third_i 0
/pipeline_testbench/b20v_inst/first_i 0
/pipeline_testbench/b20v_inst/result_o 0
/pipeline_testbench/b20v_inst/second_i 112
/pipeline_testbench/b20v_inst/sel_i 0
/pipeline_testbench/b20v_inst/third_i 0
/pipeline_testbench/b10v_inst/ram[5] 33
/pipeline_testbench/b10v_inst/ram[4] 104
Now 2200 ps
150 ps
160 ps
170 ps
180 ps
190 ps
200 ps
210 ps

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
/pipeline_testbench/instr_counter	58
/pipeline_testbench/CFinstruction	01111000
/pipeline_testbench/FDinstruction	01001100
/pipeline_testbench/new_pc	26
/pipeline_testbench/b5v_inst/c	0
/pipeline_testbench/b5v_inst/r1	76
/pipeline_testbench/b5v_inst/r2	7
/pipeline_testbench/b5v_inst/r3	0
/pipeline_testbench/b5v_inst/r4	95
/pipeline_testbench/b5v_inst/r5	0
/pipeline_testbench/b5v_inst/r6	1
/pipeline_testbench/b5v_inst/r7	x
/pipeline_testbench/b5v_inst/r8	76
/pipeline_testbench/b5v_inst/temp	0
/pipeline_testbench/b14v_inst/first_i	0
/pipeline_testbench/b14v_inst/result_o	0
/pipeline_testbench/b14v_inst/second_i	152
/pipeline_testbench/b14v_inst/sel_i	2
/pipeline_testbench/b14v_inst/third_i	0
/pipeline_testbench/b20v_inst/first_i	76
/pipeline_testbench/b20v_inst/result_o	152
/pipeline_testbench/b20v_inst/second_i	152
/pipeline_testbench/b20v_inst/sel_i	1
/pipeline_testbench/b20v_inst/third_i	0
/pipeline_testbench/b10v_inst/ram[5]	x
/pipeline_testbench/b10v_inst/ram[4]	x
Now	2200 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

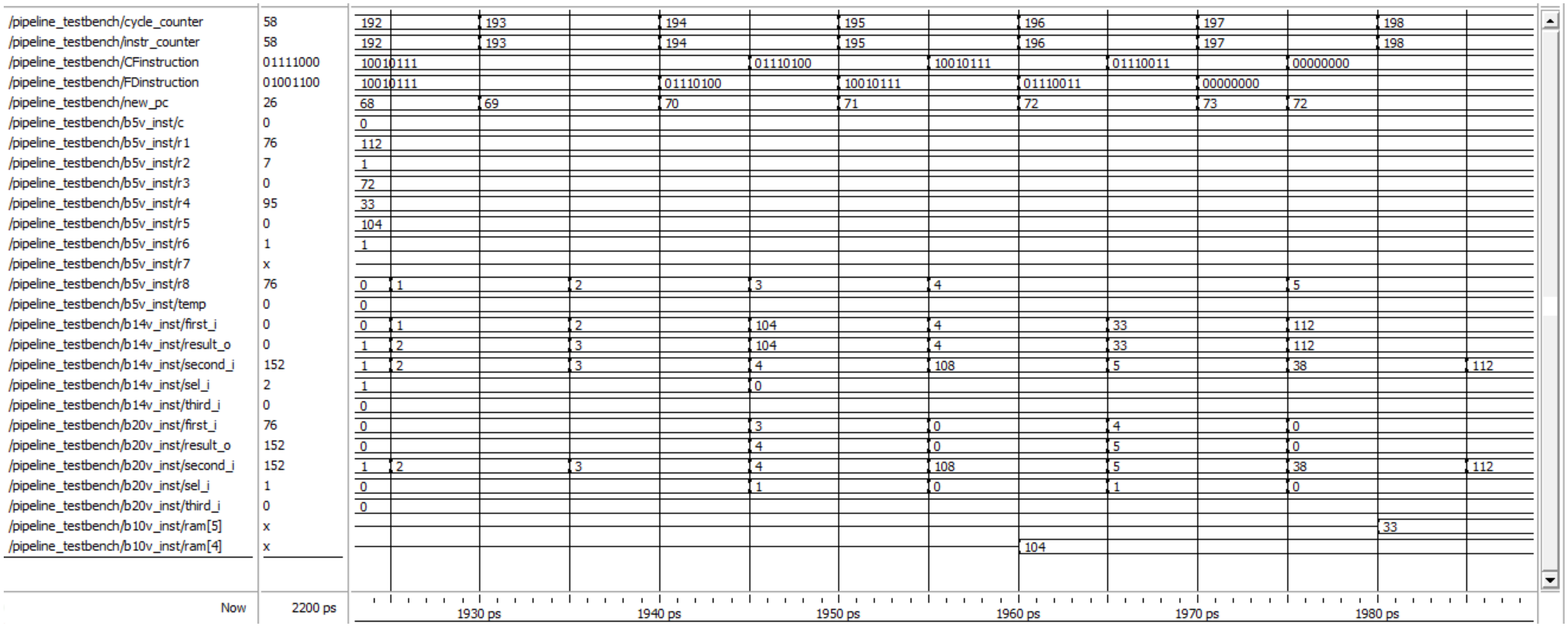
    EX: Execute shift $r2 right logical by 1

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
/pipeline_testbench/instr_counter 58
/pipeline_testbench/CFinstruction 01111000
/pipeline_testbench/FDinstruction 01001100
/pipeline_testbench/new_pc 26
/pipeline_testbench/b5v_inst/c 0
/pipeline_testbench/b5v_inst/r1 76
/pipeline_testbench/b5v_inst/r2 7
/pipeline_testbench/b5v_inst/r3 0
/pipeline_testbench/b5v_inst/r4 95
/pipeline_testbench/b5v_inst/r5 0
/pipeline_testbench/b5v_inst/r6 1
/pipeline_testbench/b5v_inst/r7 x
/pipeline_testbench/b5v_inst/r8 76
/pipeline_testbench/b5v_inst/temp 0
/pipeline_testbench/b14v_inst/first_i 0
/pipeline_testbench/b14v_inst/result_o 0
/pipeline_testbench/b14v_inst/second_i 152
/pipeline_testbench/b14v_inst/sel_i 2
/pipeline_testbench/b14v_inst/third_i 0
/pipeline_testbench/b20v_inst/first_i 76
/pipeline_testbench/b20v_inst/result_o 152
/pipeline_testbench/b20v_inst/second_i 152
/pipeline_testbench/b20v_inst/sel_i 1
/pipeline_testbench/b20v_inst/third_i 0
/pipeline_testbench/b10v_inst/ram[5] x
/pipeline_testbench/b10v_inst/ram[4] x
Now 2200 ps
1930 ps
1940 ps
1950 ps
1960 ps
1970 ps
1980 ps

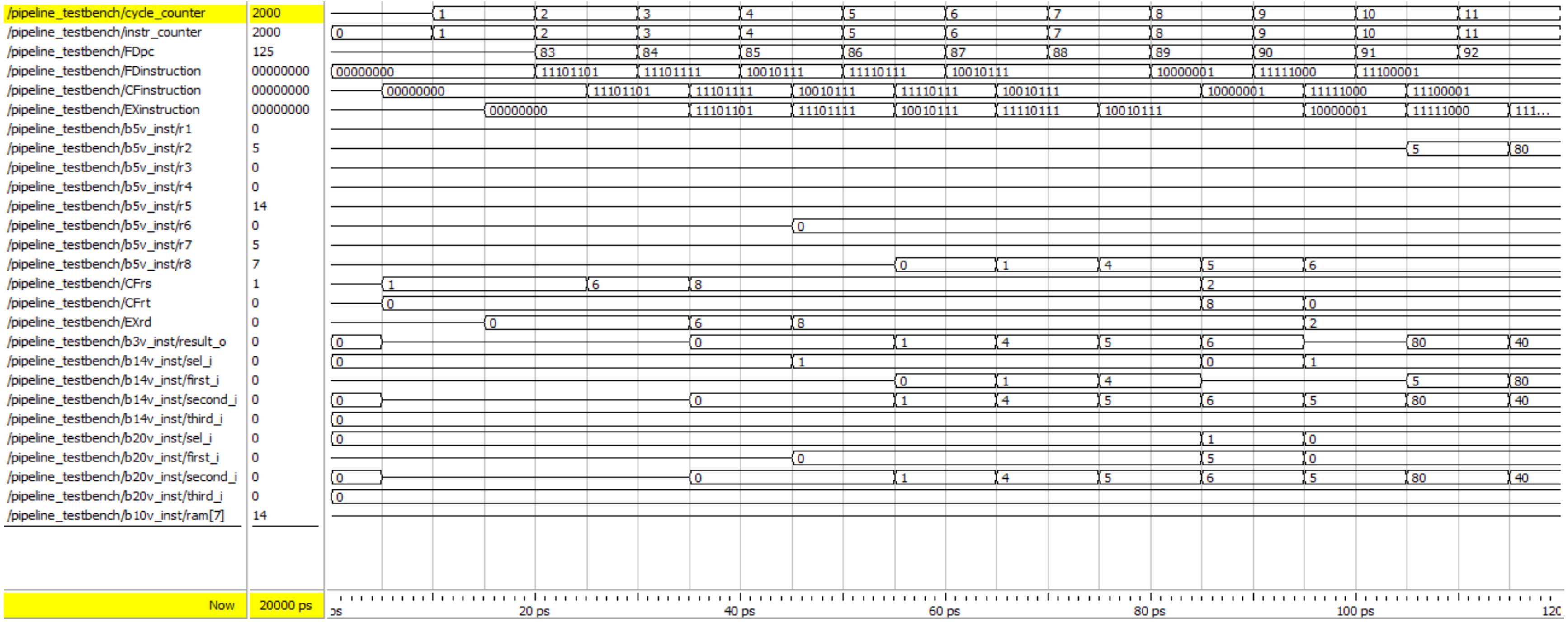
This timing diagram is the end as we are storing are values into memory ram.

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 $r8
     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
/pipeline_testbench/instr_counter	2000
/pipeline_testbench/FDpc	125
/pipeline_testbench/FDinstruction	00000000
/pipeline_testbench/CFinstruction	00000000
/pipeline_testbench/EXinstruction	00000000
/pipeline_testbench/b5v_inst/r1	0
/pipeline_testbench/b5v_inst/r2	5
/pipeline_testbench/b5v_inst/r3	0
/pipeline_testbench/b5v_inst/r4	0
/pipeline_testbench/b5v_inst/r5	14
/pipeline_testbench/b5v_inst/r6	0
/pipeline_testbench/b5v_inst/r7	5
/pipeline_testbench/b5v_inst/r8	7
/pipeline_testbench/CFrs	1
/pipeline_testbench/CFrt	0
/pipeline_testbench/EXrd	0
/pipeline_testbench/b3v_inst/result_o	0
/pipeline_testbench/b14v_inst/sel_i	0
/pipeline_testbench/b14v_inst/first_i	0
/pipeline_testbench/b14v_inst/second_i	0
/pipeline_testbench/b14v_inst/third_i	0
/pipeline_testbench/b20v_inst/sel_i	0
/pipeline_testbench/b20v_inst/first_i	0
/pipeline_testbench/b20v_inst/second_i	0
/pipeline_testbench/b20v_inst/third_i	0
/pipeline_testbench/b10v_inst/ram[7]	14
Now	20000 ps

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 ld $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

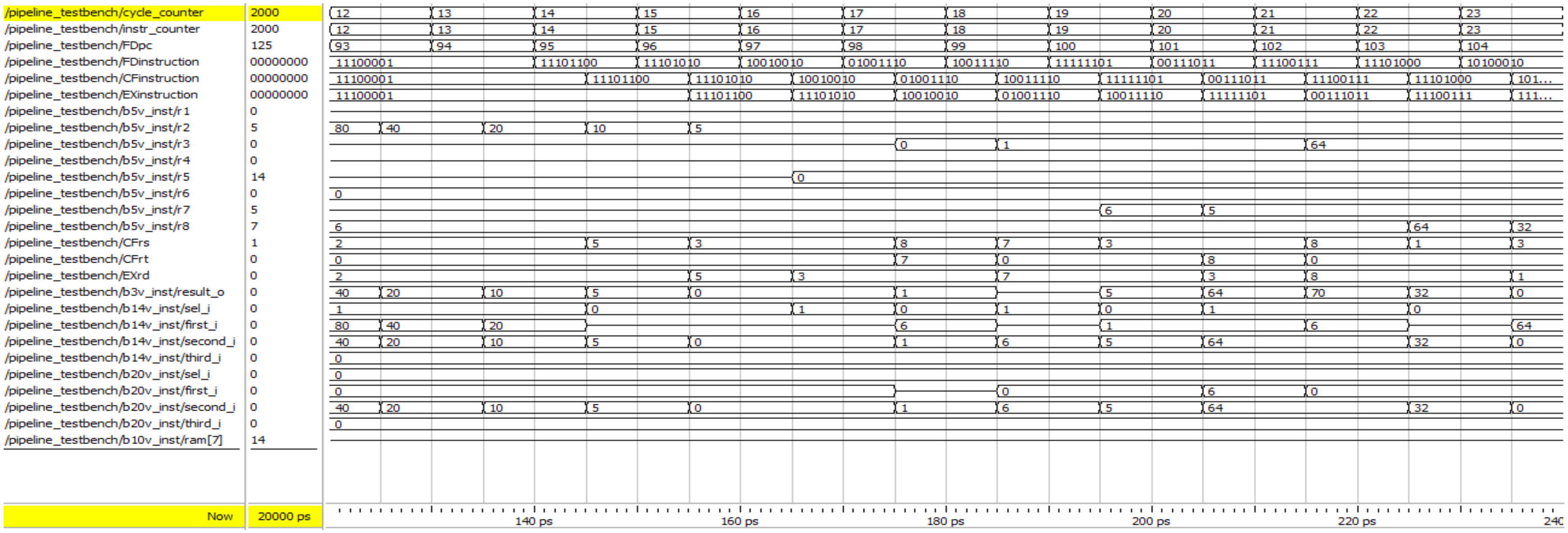
CF: Prepare rs = $r2 for shift left logical 4, forward $r2 value from EX

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 = $r2 for shift right logical 1, forward $r2 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 13 14 15 16 17 18 19 20 21 22 23
/pipeline_testbench/instr_counter 2000 | 12 13 14 15 16 17 18 19 20 21 22 23
/pipeline_testbench/FDpc 125 | 93 94 95 96 97 98 99 100 101 102 103 104
/pipeline_testbench/FDinstruction 00000000 | 11100001 11101100 11101010 10010010 01001110 10011110 11111101 00111011 11100111 11101000 10100010
/pipeline_testbench/CFinstruction 00000000 | 11100001 11101100 11101010 10010010 01001110 10011110 11111101 00111011 11100111 11101000 101...
/pipeline_testbench/EXinstruction 00000000 | 11100001 11101100 11101010 10010010 01001110 10011110 11111101 00111011 11100111 111...
/pipeline_testbench/b5v_inst/r1 0
/pipeline_testbench/b5v_inst/r2 5 | 80 40 20 10 5
/pipeline_testbench/b5v_inst/r3 0 | 0 1 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 5
/pipeline_testbench/b5v_inst/r8 7 | 6 64 32
/pipeline_testbench/CFrs 1 | 2 5 3 8 7 3 8 1 3
/pipeline_testbench/CFrt 0 | 0 7 0 8 0
/pipeline_testbench/EXrd 0 | 2 5 3 7 3 8 1
/pipeline_testbench/b3v_inst/result_o 0 | 40 20 10 5 0 1 5 64 70 32 0
/pipeline_testbench/b14v_inst/sel_i 0 | 1 0 1 0 1 0 1 0
/pipeline_testbench/b14v_inst/first_i 0 | 80 40 20 6 1 6 64
/pipeline_testbench/b14v_inst/second_i 0 | 40 20 10 5 0 1 6 5 64 32 0
/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 6 0
/pipeline_testbench/b20v_inst/second_i 0 | 40 20 10 5 0 1 6 5 64 32 0
/pipeline_testbench/b20v_inst/third_i 0 | 0
/pipeline_testbench/b10v_inst/ram[7] 14
Now 20000 ps
140 ps
160 ps
180 ps
200 ps
220 ps
240

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

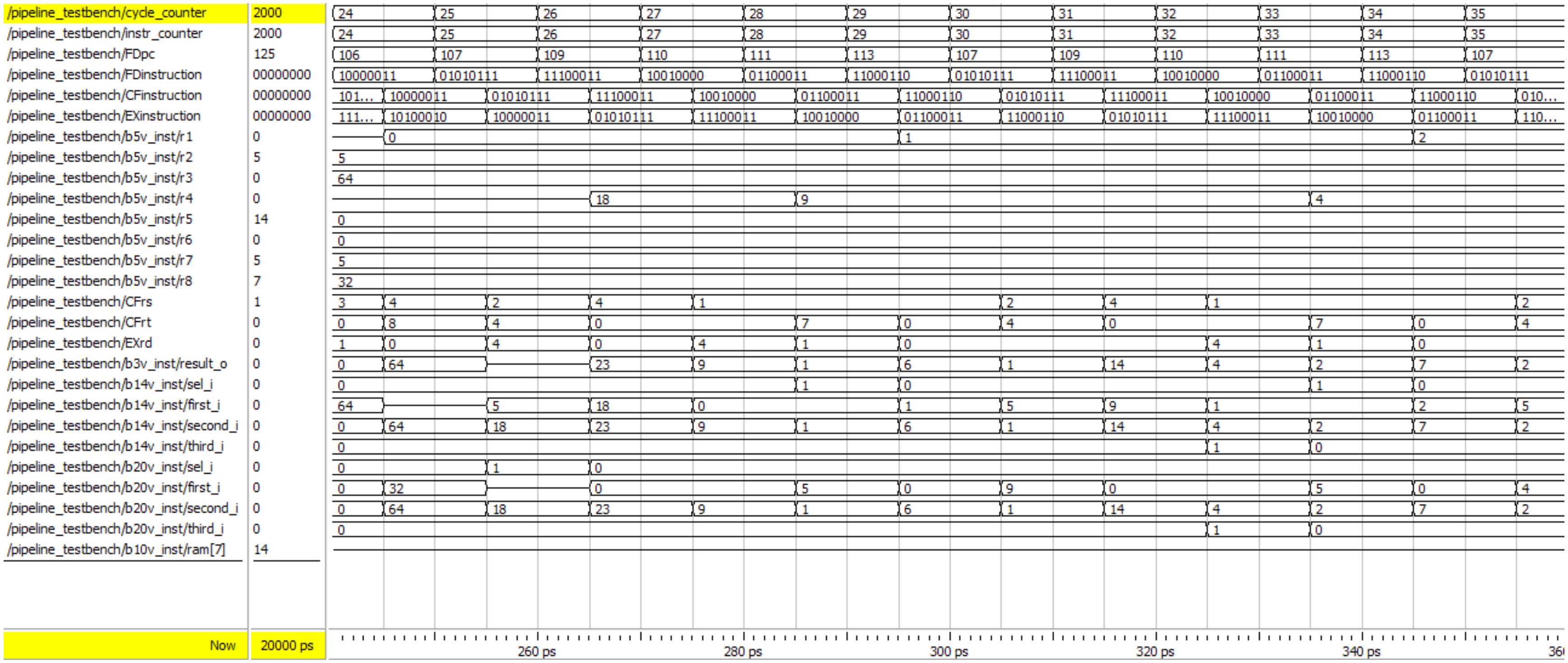
19: FD: Fetch/decode sll 6, $r3 instruction
CF: Prepare rs = $r7 for $r7--, forward $r7 from EX (first sel_i = 1)
EX: move $r8 value to $r7, $r7 = $r8

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
/pipeline_testbench/instr_counter	2000
/pipeline_testbench/FDpc	125
/pipeline_testbench/FDinstruction	00000000
/pipeline_testbench/CFinstruction	00000000
/pipeline_testbench/EXinstruction	00000000
/pipeline_testbench/b5v_inst/r1	0
/pipeline_testbench/b5v_inst/r2	5
/pipeline_testbench/b5v_inst/r3	0
/pipeline_testbench/b5v_inst/r4	0
/pipeline_testbench/b5v_inst/r5	14
/pipeline_testbench/b5v_inst/r6	0
/pipeline_testbench/b5v_inst/r7	5
/pipeline_testbench/b5v_inst/r8	7
/pipeline_testbench/CFrs	1
/pipeline_testbench/CFrt	0
/pipeline_testbench/EXrd	0
/pipeline_testbench/b3v_inst/result_o	0
/pipeline_testbench/b14v_inst/sel_i	0
/pipeline_testbench/b14v_inst/first_i	0
/pipeline_testbench/b14v_inst/second_i	0
/pipeline_testbench/b14v_inst/third_i	0
/pipeline_testbench/b20v_inst/sel_i	0
/pipeline_testbench/b20v_inst/first_i	0
/pipeline_testbench/b20v_inst/second_i	0
/pipeline_testbench/b20v_inst/third_i	0
/pipeline_testbench/b10v_inst/ram[7]	14
Now	20000 ps

(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 = $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 into $r4, write $r4 (8-bit string that we are checking)

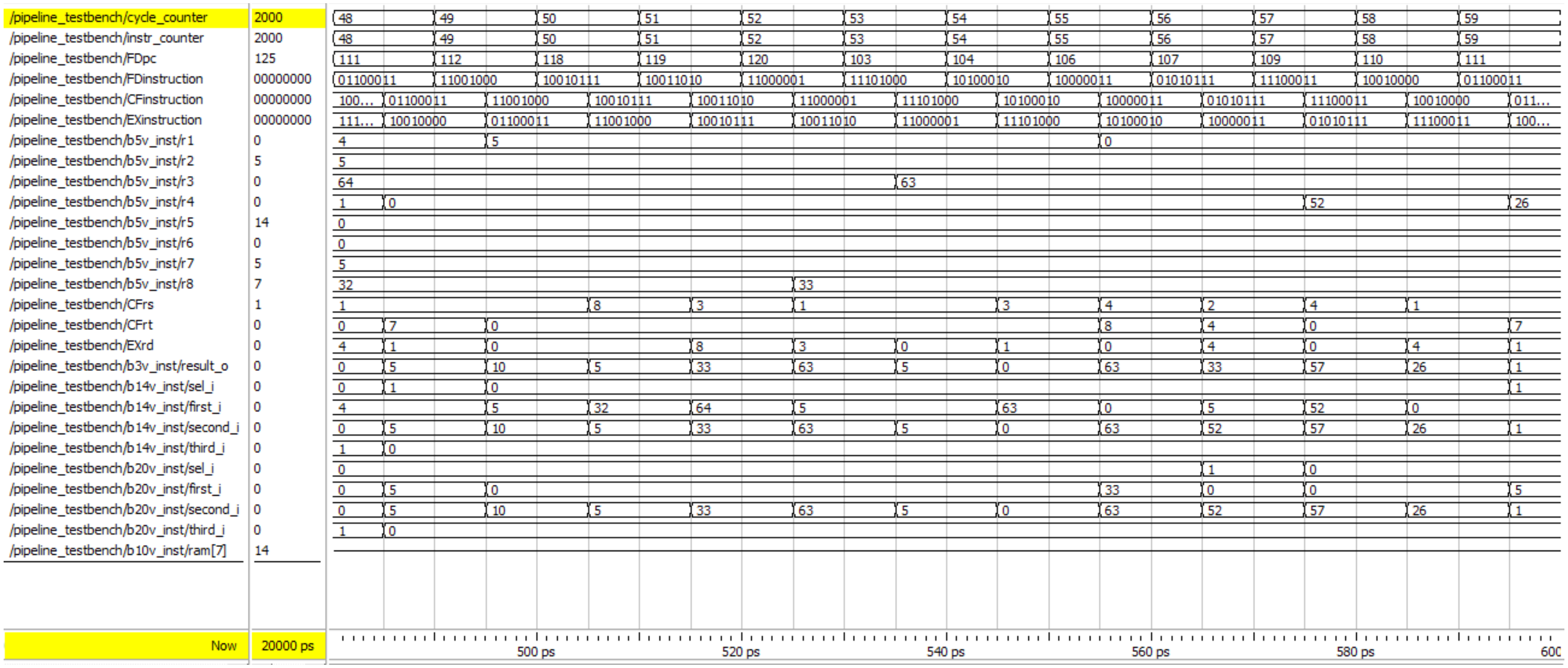
27: FD: Fetch/decode addi 0, $r1 instruction

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
/pipeline_testbench/instr_counter	2000
/pipeline_testbench/FDpc	125
/pipeline_testbench/FDinstruction	00000000
/pipeline_testbench/CFinstruction	00000000
/pipeline_testbench/EXinstruction	00000000
/pipeline_testbench/b5v_inst/r1	0
/pipeline_testbench/b5v_inst/r2	5
/pipeline_testbench/b5v_inst/r3	0
/pipeline_testbench/b5v_inst/r4	0
/pipeline_testbench/b5v_inst/r5	14
/pipeline_testbench/b5v_inst/r6	0
/pipeline_testbench/b5v_inst/r7	5
/pipeline_testbench/b5v_inst/r8	7
/pipeline_testbench/CFrs	1
/pipeline_testbench/CFrt	0
/pipeline_testbench/EXrd	0
/pipeline_testbench/b3v_inst/result_o	0
/pipeline_testbench/b14v_inst/sel_i	0
/pipeline_testbench/b14v_inst/first_i	0
/pipeline_testbench/b14v_inst/second_i	0
/pipeline_testbench/b14v_inst/third_i	0
/pipeline_testbench/b20v_inst/sel_i	0
/pipeline_testbench/b20v_inst/first_i	0
/pipeline_testbench/b20v_inst/second_i	0
/pipeline_testbench/b20v_inst/third_i	0
/pipeline_testbench/b10v_inst/ram[7]	14
Now	20000 ps

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

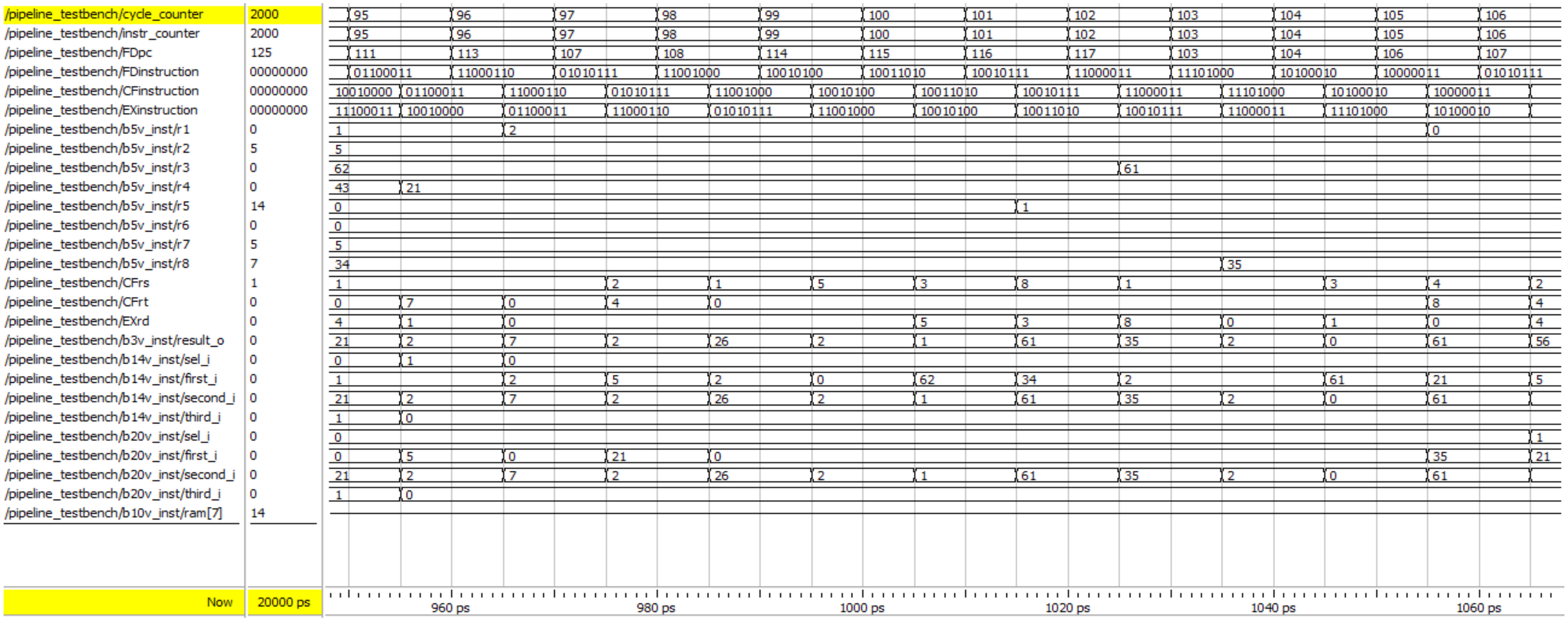
57: FD: Fetch/decode srl 1, $r4 instruction

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 96 97 98 99 100 101 102 103 104 105 106
/pipeline_testbench/instr_counter 2000 95 96 97 98 99 100 101 102 103 104 105 106
/pipeline_testbench/FDpc 125 111 113 107 108 114 115 116 117 103 104 106 107
/pipeline_testbench/FDinstruction 00000000 01100011 11000110 01010111 11001000 10010100 10011010 10010111 11000011 11101000 10100010 10000011 01010111
/pipeline_testbench/CFinstruction 00000000 10010000 01100011 11000110 01010111 11001000 10010100 10011010 10010111 11000011 11101000 10100010 10000011
/pipeline_testbench/EXinstruction 00000000 11100011 10010000 01100011 11000110 01010111 11001000 10010100 10011010 10010111 11000011 11101000 10100010
/pipeline_testbench/b5v_inst/r1 0 1 2 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 1 5 3 8 1 3 4 2
/pipeline_testbench/CFrt 0 0 7 0 4 0 8 4
/pipeline_testbench/EXrd 0 4 1 0 5 3 8 0 1 0 4
/pipeline_testbench/b3v_inst/result_o 0 21 2 7 2 26 2 1 61 35 2 0 61 56
/pipeline_testbench/b14v_inst/sel_i 0 0 1 0
/pipeline_testbench/b14v_inst/first_i 0 1 2 5 2 0 62 34 2 61 21 5
/pipeline_testbench/b14v_inst/second_i 0 21 2 7 2 26 2 1 61 35 2 0 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 0 21 0 35 21
/pipeline_testbench/b20v_inst/second_i 0 21 2 7 2 26 2 1 61 35 2 0 61
/pipeline_testbench/b20v_inst/third_i 0 1 0
/pipeline_testbench/b10v_inst/ram[7] 14
Now 20000 ps
960 ps
980 ps
1000 ps
1020 ps
1040 ps
1060 ps

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

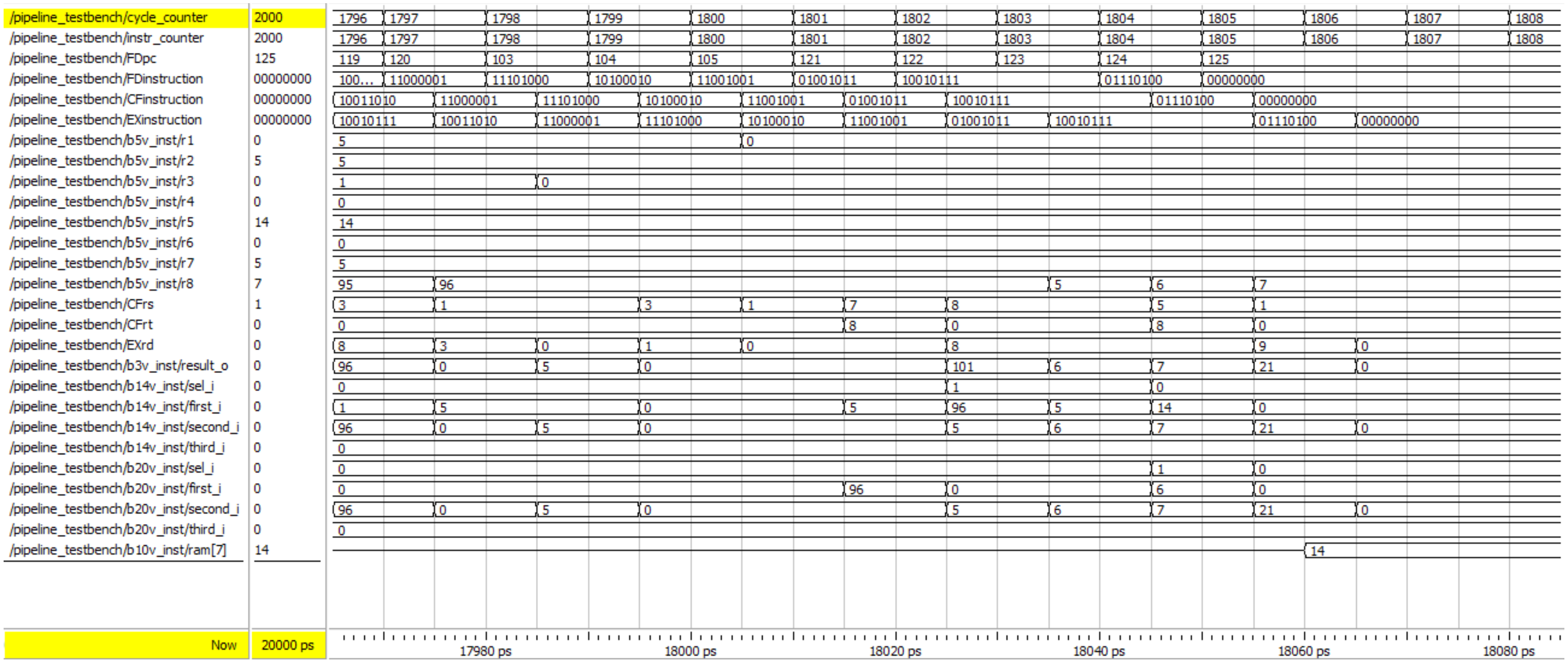
99: FD: Fetch/decode addi 0, $r5 instruction

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
/pipeline_testbench/instr_counter	2000
/pipeline_testbench/FDpc	125
/pipeline_testbench/FDinstruction	00000000
/pipeline_testbench/CFinstruction	00000000
/pipeline_testbench/EXinstruction	00000000
/pipeline_testbench/b5v_inst/r1	0
/pipeline_testbench/b5v_inst/r2	5
/pipeline_testbench/b5v_inst/r3	0
/pipeline_testbench/b5v_inst/r4	0
/pipeline_testbench/b5v_inst/r5	14
/pipeline_testbench/b5v_inst/r6	0
/pipeline_testbench/b5v_inst/r7	5
/pipeline_testbench/b5v_inst/r8	7
/pipeline_testbench/CFrs	1
/pipeline_testbench/CFrt	0
/pipeline_testbench/EXrd	0
/pipeline_testbench/b3v_inst/result_o	0
/pipeline_testbench/b14v_inst/sel_i	0
/pipeline_testbench/b14v_inst/first_i	0
/pipeline_testbench/b14v_inst/second_i	0
/pipeline_testbench/b14v_inst/third_i	0
/pipeline_testbench/b20v_inst/sel_i	0
/pipeline_testbench/b20v_inst/first_i	0
/pipeline_testbench/b20v_inst/second_i	0
/pipeline_testbench/b20v_inst/third_i	0
/pipeline_testbench/b10v_inst/ram[7]	14
Now	20000 ps

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)

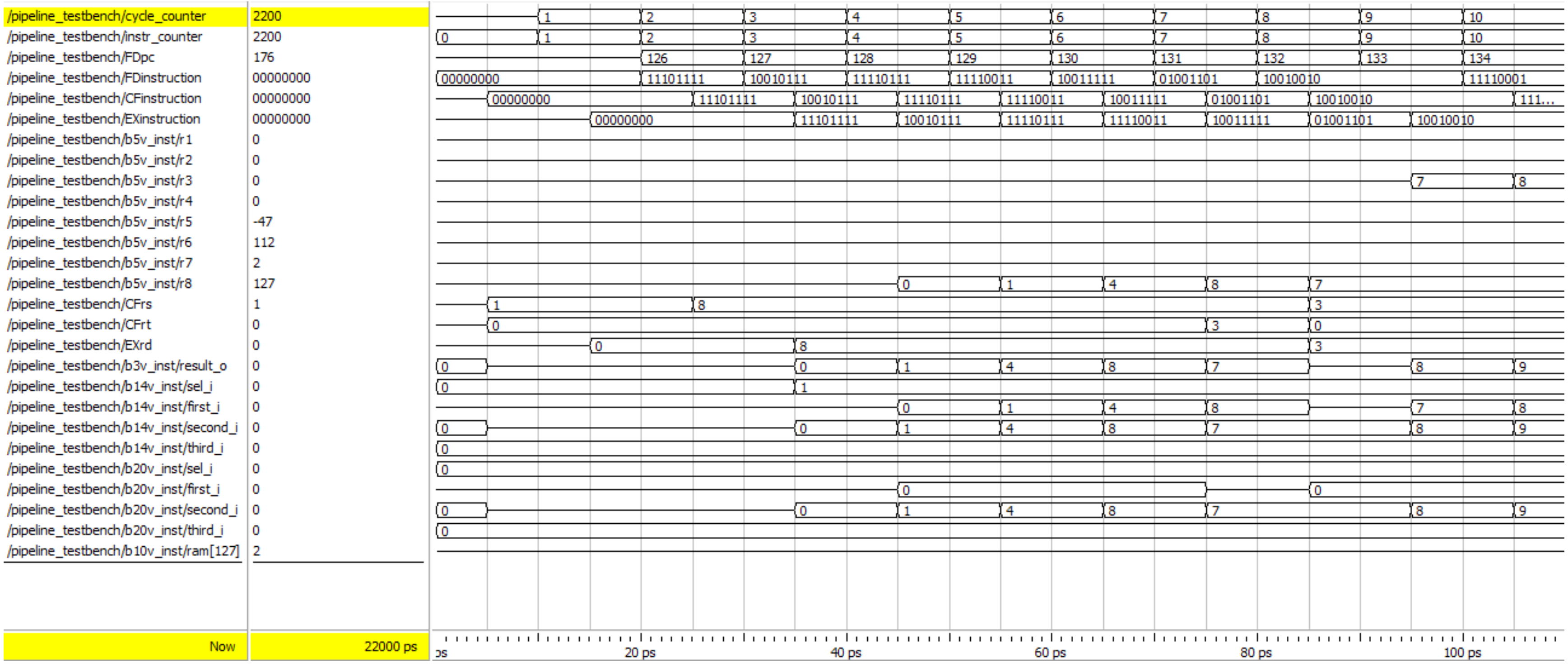
EX: $r8++, write to $r8, $r8 = 6
1805: FD: Fetch/decode halt instruction
CF: Prepare rs = $r5, rt = $r8 for store, forward $r8 from EX (first sel_i = 0, second sel_i = 1)
EX: $r8++, write to $r8, $r8 = 7
1806: FD: Fetch/decode halt instruction
CF: halt
EX: store $r5 to address of $r8, datamem[7] = $r5
(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)



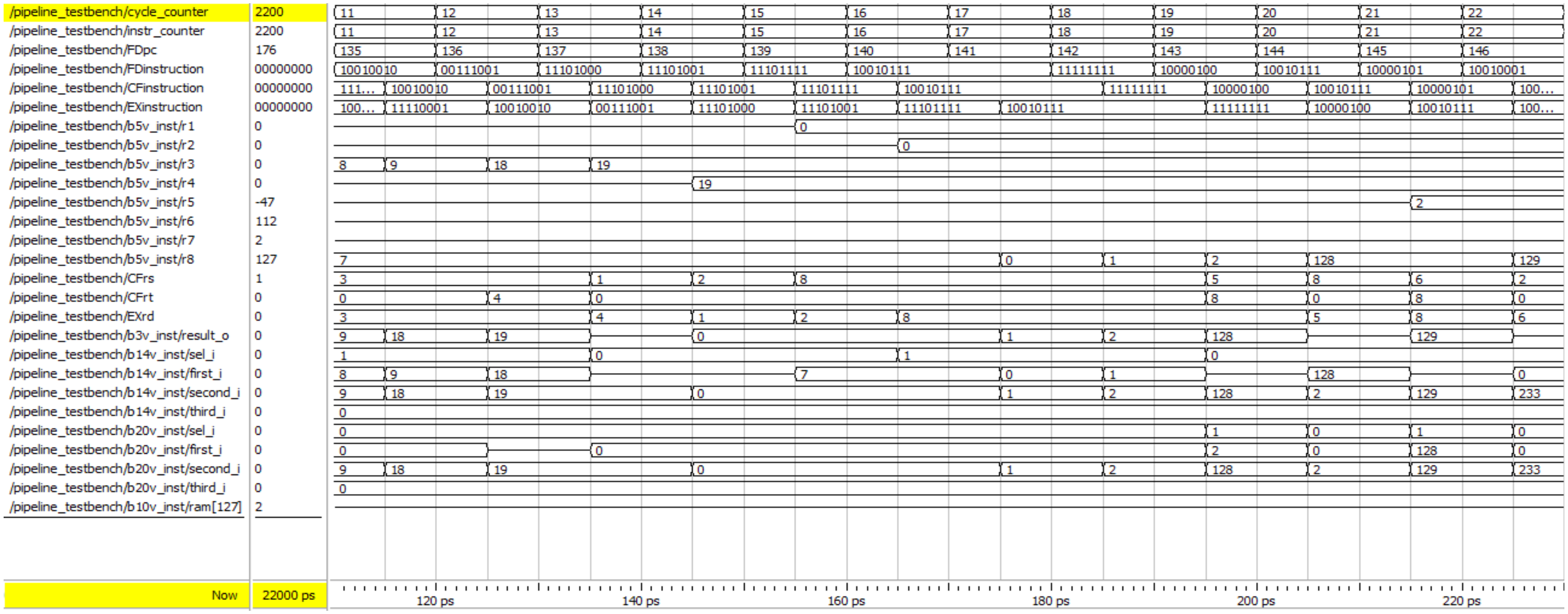
/pipeline_testbench/cycle_counter	2200
/pipeline_testbench/instr_counter	2200
/pipeline_testbench/FDpc	176
/pipeline_testbench/FDinstruction	00000000
/pipeline_testbench/CFinstruction	00000000
/pipeline_testbench/EXinstruction	00000000
/pipeline_testbench/b5v_inst/r1	0
/pipeline_testbench/b5v_inst/r2	0
/pipeline_testbench/b5v_inst/r3	0
/pipeline_testbench/b5v_inst/r4	0
/pipeline_testbench/b5v_inst/r5	-47
/pipeline_testbench/b5v_inst/r6	112
/pipeline_testbench/b5v_inst/r7	2
/pipeline_testbench/b5v_inst/r8	127
/pipeline_testbench/CFrs	1
/pipeline_testbench/CFrt	0
/pipeline_testbench/EXrd	0
/pipeline_testbench/b3v_inst/result_o	0
/pipeline_testbench/b14v_inst/sel_i	0
/pipeline_testbench/b14v_inst/first_i	0
/pipeline_testbench/b14v_inst/second_i	0
/pipeline_testbench/b14v_inst/third_i	0
/pipeline_testbench/b20v_inst/sel_i	0
/pipeline_testbench/b20v_inst/first_i	0
/pipeline_testbench/b20v_inst/second_i	0
/pipeline_testbench/b20v_inst/third_i	0
/pipeline_testbench/b10v_inst/ram[127]	2
Now	22000 ps

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
/pipeline_testbench/instr_counter          2200
/pipeline_testbench/FDpc                   176
/pipeline_testbench/FDinstruction          00000000
/pipeline_testbench/CFinstruction          00000000
/pipeline_testbench/EXinstruction          00000000
/pipeline_testbench/b5v_inst/r1            0
/pipeline_testbench/b5v_inst/r2            0
/pipeline_testbench/b5v_inst/r3            0
/pipeline_testbench/b5v_inst/r4            0
/pipeline_testbench/b5v_inst/r5            -47
/pipeline_testbench/b5v_inst/r6            112
/pipeline_testbench/b5v_inst/r7            2
/pipeline_testbench/b5v_inst/r8            127
/pipeline_testbench/CFrs                   1
/pipeline_testbench/CFrt                   0
/pipeline_testbench/EXrd                   0
/pipeline_testbench/b3v_inst/result_o      0
/pipeline_testbench/b14v_inst/sel_i        0
/pipeline_testbench/b14v_inst/first_i      0
/pipeline_testbench/b14v_inst/second_i     0
/pipeline_testbench/b14v_inst/third_i      0
/pipeline_testbench/b20v_inst/sel_i        0
/pipeline_testbench/b20v_inst/first_i      0
/pipeline_testbench/b20v_inst/second_i     0
/pipeline_testbench/b20v_inst/third_i      0
/pipeline_testbench/b10v_inst/ram[127]     2
Now                                        22000 ps
```

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

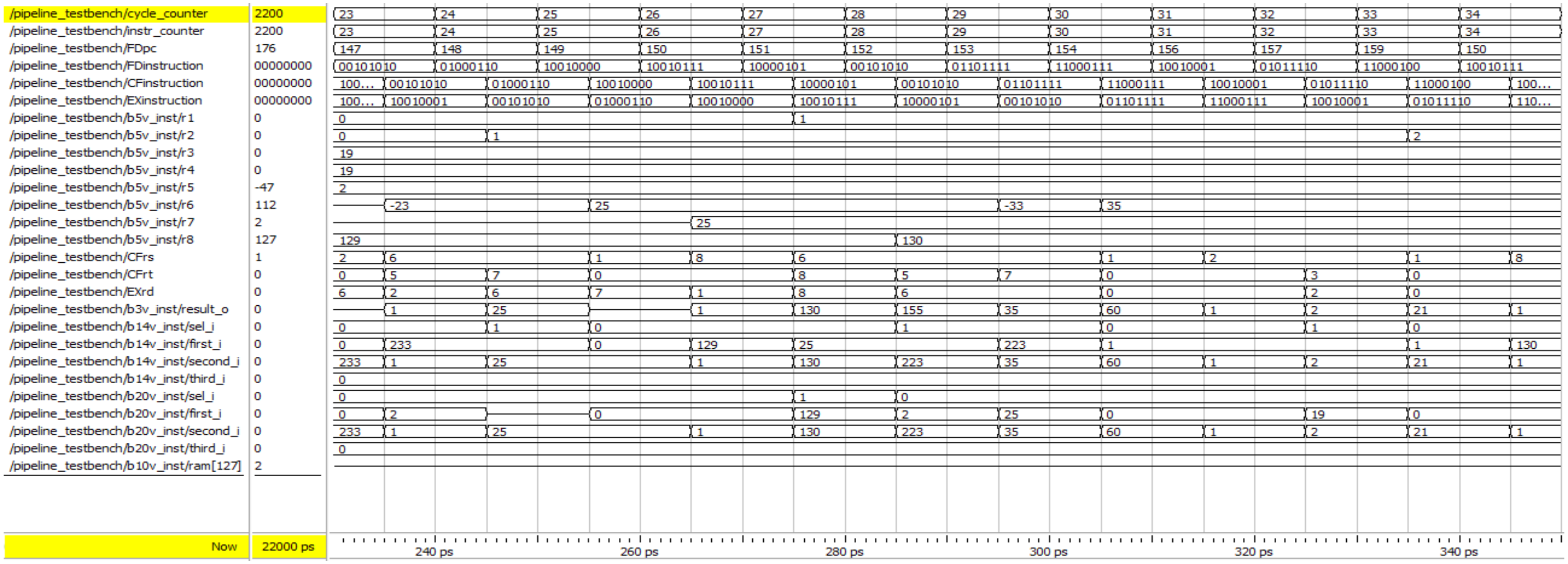
CF: Prepare rs = $r8 for $r8++, no forwarding (first sel_i = 0, second sel_i = 0)

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
/pipeline_testbench/instr_counter         2200
/pipeline_testbench/FDpc                  176
/pipeline_testbench/FDinstruction         00000000
/pipeline_testbench/CFinstruction         00000000
/pipeline_testbench/EXinstruction         00000000
/pipeline_testbench/b5v_inst/r1           0
/pipeline_testbench/b5v_inst/r2           0
/pipeline_testbench/b5v_inst/r3           0
/pipeline_testbench/b5v_inst/r4           0
/pipeline_testbench/b5v_inst/r5           -47
/pipeline_testbench/b5v_inst/r6           112
/pipeline_testbench/b5v_inst/r7           2
/pipeline_testbench/b5v_inst/r8           127
/pipeline_testbench/CFrs                  1
/pipeline_testbench/CFrt                  0
/pipeline_testbench/EXrd                  0
/pipeline_testbench/b3v_inst/result_o     0
/pipeline_testbench/b14v_inst/sel_i       0
/pipeline_testbench/b14v_inst/first_i     0
/pipeline_testbench/b14v_inst/second_i    0
/pipeline_testbench/b14v_inst/third_i     0
/pipeline_testbench/b20v_inst/sel_i       0
/pipeline_testbench/b20v_inst/first_i     0
/pipeline_testbench/b20v_inst/second_i    0
/pipeline_testbench/b20v_inst/third_i     0
/pipeline_testbench/b10v_inst/ram[127]    2
Now                                       22000 ps
```

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

23: FD: Fetch/decode diff $r6, $r5 instruction
 CF: Prepare rs = $r2 for $r2++, no forwarding (first sel_i = 0, second sel_i = 0)
 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

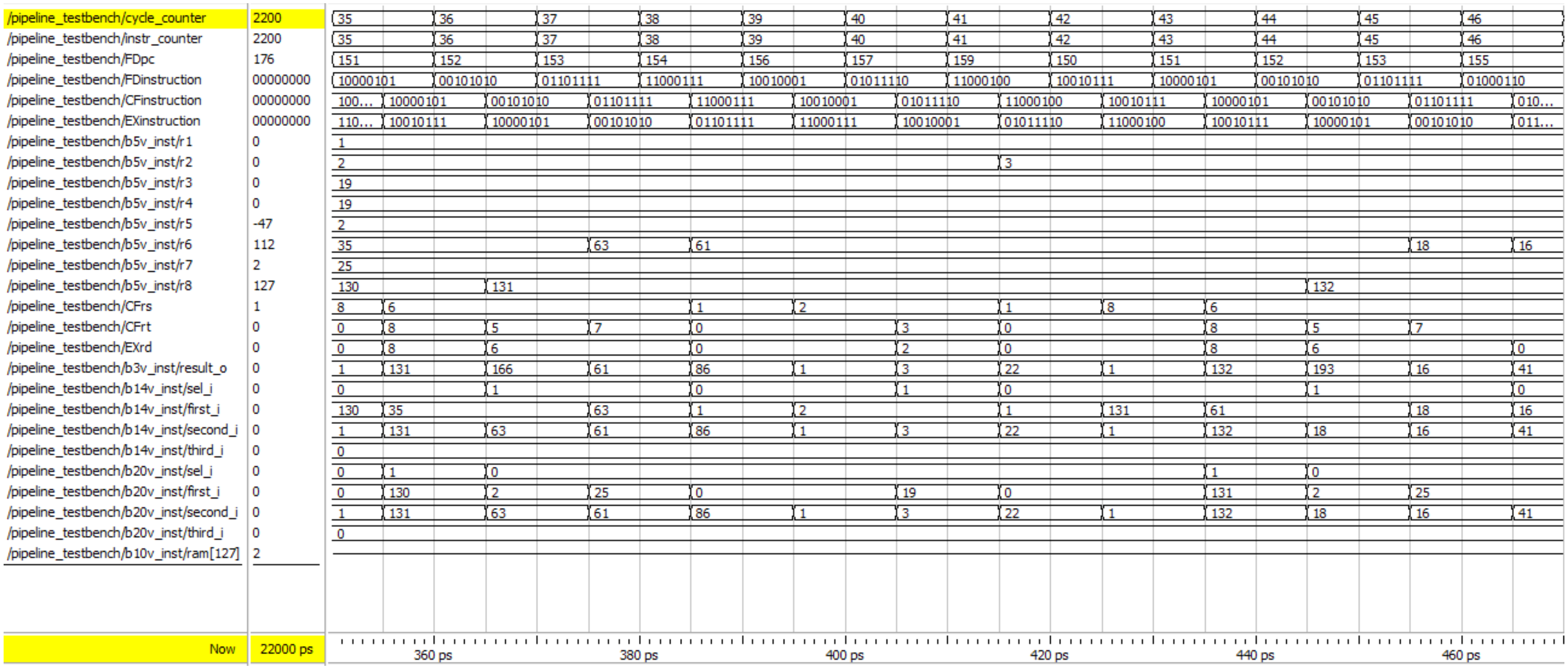
25: FD: Fetch/decode addi 0, $r1 instruction
 CF: Prepare rs = $r6,rt = $r7 for move, forward $r6 from EX(first sel_i = 1, second sel_i = 0)
 EX: diff $r6, $r5, write to $r6

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)



Signal	Value	Waveform
/pipeline_testbench/cycle_counter	2200	35 36 37 38 39 40 41 42 43 44 45 46
/pipeline_testbench/instr_counter	2200	35 36 37 38 39 40 41 42 43 44 45 46
/pipeline_testbench/FDpc	176	151 152 153 154 156 157 159 150 151 152 153 155
/pipeline_testbench/FDinstruction	00000000	10000101 00101010 01101111 11000111 10010001 01011110 11000100 10010111 10000101 00101010 01101111 01000110
/pipeline_testbench/CFinstruction	00000000	100... 10000101 00101010 01101111 11000111 10010001 01011110 11000100 10010111 10000101 00101010 01101111 010...
/pipeline_testbench/EXinstruction	00000000	110... 10010111 10000101 00101010 01101111 11000111 10010001 01011110 11000100 10010111 10000101 00101010 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 3 0 8 5 7
/pipeline_testbench/EXrd	0	0 8 6 0 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 0
/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 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 0
/pipeline_testbench/b20v_inst/first_i	0	0 130 2 25 0 19 0 131 2 25
/pipeline_testbench/b20v_inst/second_i	0	1 131 63 61 86 1 3 22 1 132 18 16 41
/pipeline_testbench/b20v_inst/third_i	0	0
/pipeline_testbench/b10v_inst/ram[127]	2	
Now	22000 ps	360 ps 380 ps 400 ps 420 ps 440 ps 460 ps

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

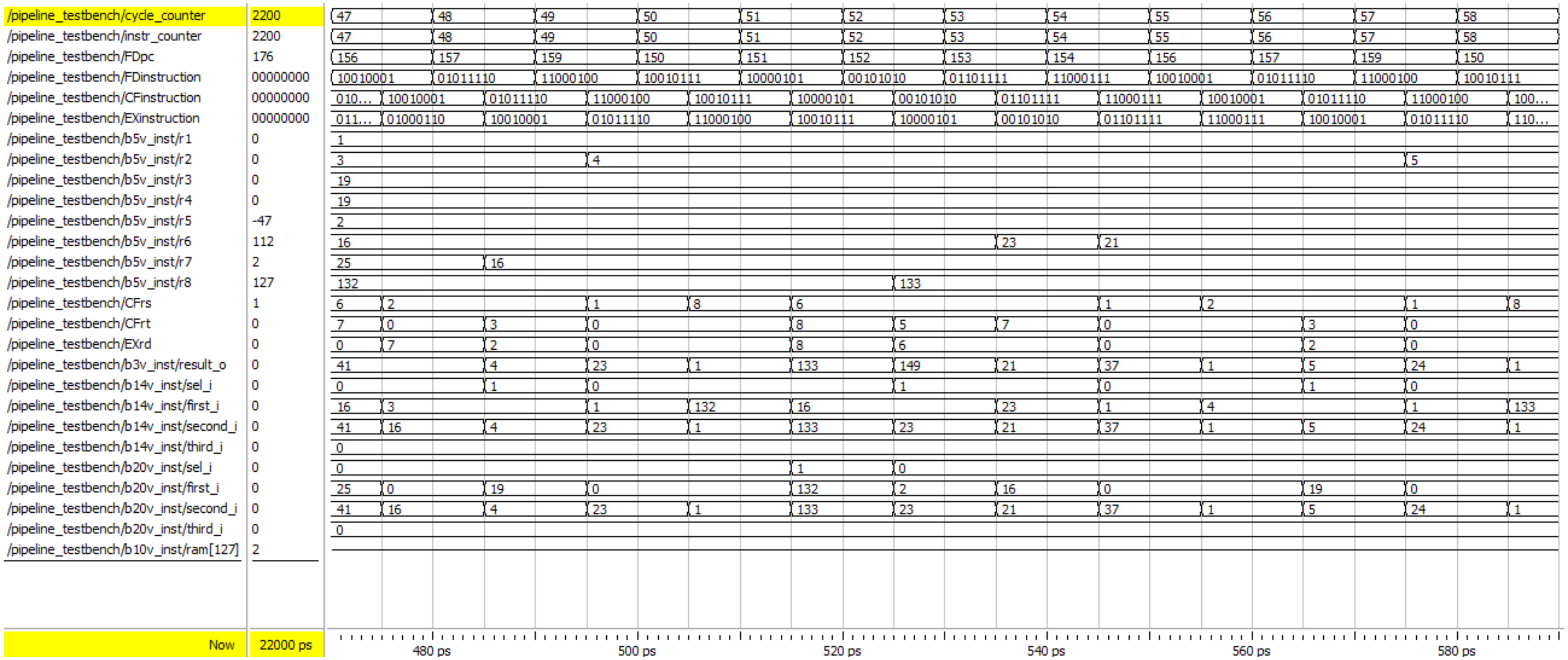
CF: Prepare rs = $r6, rt = $r5 for diff, forward $r6 from EX (first sel_i = 1, second sel_i = 0)

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 52 53 54 55 56 57 58
/pipeline_testbench/instr_counter        2200      47 48 49 50 51 52 53 54 55 56 57 58
/pipeline_testbench/FDpc                 176       156 157 159 150 151 152 153 154 156 157 159 150
/pipeline_testbench/FDinstruction        00000000  10010001 01011110 11000100 10010111 10000101 00101010 01101111 11000111 10010001 01011110 11000100 10010111
/pipeline_testbench/CFinstruction        00000000  010... 10010001 01011110 11000100 10010111 10000101 00101010 01101111 11000111 10010001 01011110 11000100 100...
/pipeline_testbench/EXinstruction        00000000  011... 01000110 10010001 01011110 11000100 10010111 10000101 00101010 01101111 11000111 10010001 01011110 110...
/pipeline_testbench/b5v_inst/r1          0         1
/pipeline_testbench/b5v_inst/r2          0         3 4 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 23 21
/pipeline_testbench/b5v_inst/r7          2         25 16
/pipeline_testbench/b5v_inst/r8          127       132 133
/pipeline_testbench/CFrs                 1         6 2 1 8 6 1 2 1 8
/pipeline_testbench/CFrt                 0         7 0 3 0 8 5 7 0 3 0
/pipeline_testbench/EXrd                 0         0 7 2 0 8 6 0 2 0
/pipeline_testbench/b3v_inst/result_o    0         41 4 23 1 133 149 21 37 1 5 24 1
/pipeline_testbench/b14v_inst/sel_i      0         0 1 0 1 0 1 0
/pipeline_testbench/b14v_inst/first_i    0         16 3 1 132 16 23 1 4 1 133
/pipeline_testbench/b14v_inst/second_i   0         41 16 4 23 1 133 23 21 37 1 5 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 0 132 2 16 0 19 0
/pipeline_testbench/b20v_inst/second_i   0         41 16 4 23 1 133 23 21 37 1 5 24 1
/pipeline_testbench/b20v_inst/third_i    0         0
/pipeline_testbench/b10v_inst/ram[127]   2

Now                                      22000 ps  480 ps   500 ps   520 ps   540 ps   560 ps   580 ps
```

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.

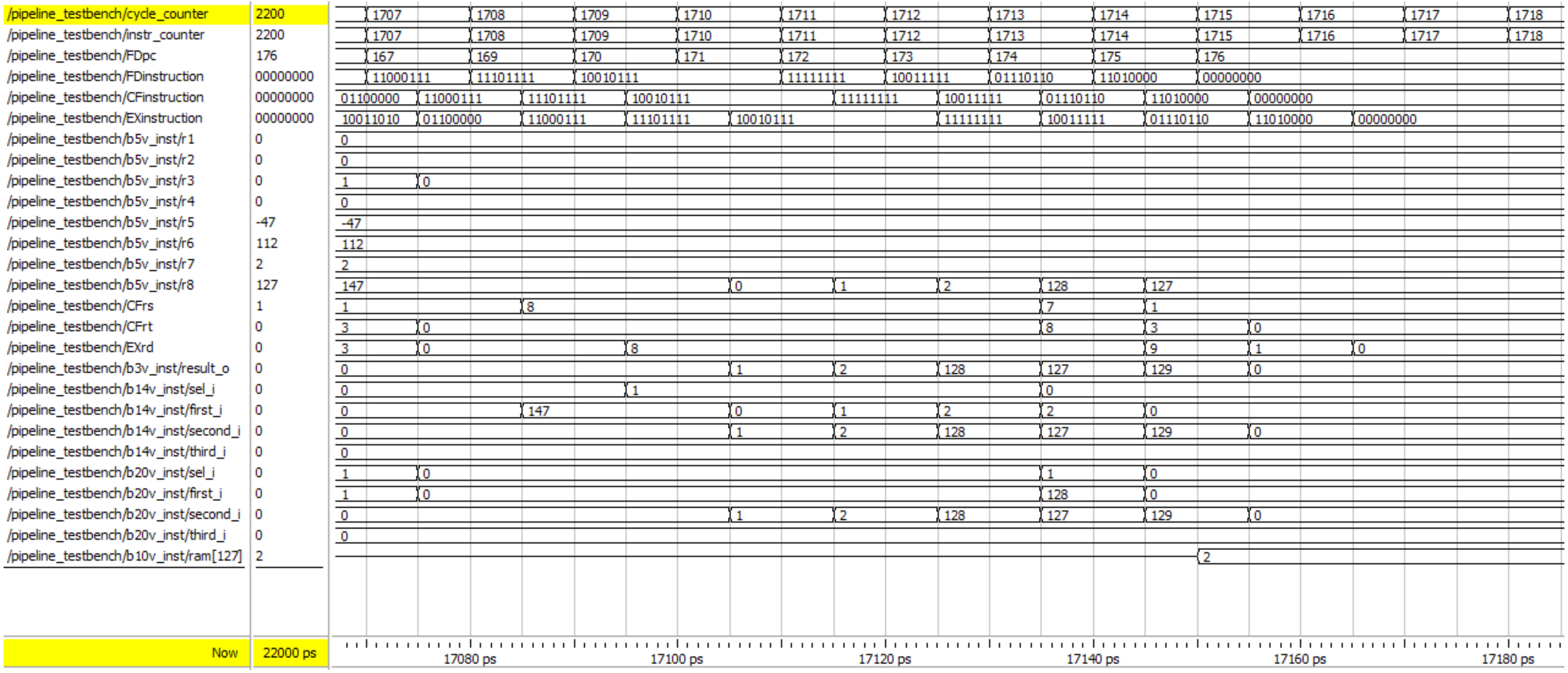
48: FD: Fetch/decode be $r2, $r3 instruction

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)



Signal	Value	Waveform
/pipeline_testbench/cycle_counter	2200	1707, 1708, 1709, 1710, 1711, 1712, 1713, 1714, 1715, 1716, 1717, 1718
/pipeline_testbench/instr_counter	2200	1707, 1708, 1709, 1710, 1711, 1712, 1713, 1714, 1715, 1716, 1717, 1718
/pipeline_testbench/FDpc	176	167, 169, 170, 171, 172, 173, 174, 175, 176
/pipeline_testbench/FDinstruction	00000000	11000111, 11101111, 10010111, 11111111, 10011111, 01110110, 11010000, 00000000
/pipeline_testbench/CFinstruction	00000000	01100000, 11000111, 11101111, 10010111, 11111111, 10011111, 01110110, 11010000, 00000000
/pipeline_testbench/EXinstruction	00000000	10011010, 01100000, 11000111, 11101111, 10010111, 11111111, 10011111, 01110110, 11010000, 00000000
/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, 0, 1, 2, 128, 127
/pipeline_testbench/CFrs	1	1, 8, 7, 1
/pipeline_testbench/CFrt	0	3, 0, 8, 3, 0
/pipeline_testbench/EXrd	0	3, 0, 8, 9, 1, 0
/pipeline_testbench/b3v_inst/result_o	0	0, 1, 2, 128, 127, 129, 0
/pipeline_testbench/b14v_inst/sel_i	0	0, 1, 0
/pipeline_testbench/b14v_inst/first_i	0	0, 147, 0, 1, 2, 2, 0
/pipeline_testbench/b14v_inst/second_i	0	0, 1, 2, 128, 127, 129, 0
/pipeline_testbench/b14v_inst/third_i	0	0
/pipeline_testbench/b20v_inst/sel_i	0	1, 0, 1, 0
/pipeline_testbench/b20v_inst/first_i	0	1, 0, 128, 0
/pipeline_testbench/b20v_inst/second_i	0	0, 1, 2, 128, 127, 129, 0
/pipeline_testbench/b20v_inst/third_i	0	0
/pipeline_testbench/b10v_inst/ram[127]	2	2
Now	22000 ps	17080 ps, 17100 ps, 17120 ps, 17140 ps, 17160 ps, 17180 ps

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
addi      0, $r8                        # $r8++, so now $r8 = 2, $r8 = &B
ld        $r2                           # $r2 = B
addi      0, $r6
bz        0, $r2                        # if (B==0), then jump to finish_1
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*
addi      1, $r2                        # B—
add       $r1, $r4
add       $c, $r5                       # accumulator += A
add       $r3, $r5                      #add to odd accumulator
jump      addB_loop           # newPC = PC + (-8)*
sll       1, $r2
movl      $r1, $r8
sll       1, $r8                        # sll 1
store                                   # write $c to $temp
movh      $r8, $r1
sll       1, $r3                        # sll 1
ld                                      # write $temp to $c
add       $c, $r3                       # accumulator *=2
jump      addB_loop           # newPC = PC + (-17)*
add       $r4,$r1
add       $c, $r5                       #add overflow value to r5
add       $r3, $r5                      #add odd accumulator with even accumulator
movh      $r5,$r3
srl       8, $r8
addi      0,$r8
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
srl       8, $r4                        # srl 8, so clear $r4 for lower 16 bit odd accumulator
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
jump      end_C                         # newPC = PC + 17
btwo      $r2                           # if (C%2==0), then C/=2 and jump to Cdiv_two
```

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

program_2:
srl       8, $r6                        # srl 8, so clear $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
addi      0, $r8                        # $r8++, so now $r8 = 5
addi      0, $r8                        # $r8++, so now $r8 = 6, so now $r8 = address of 4-bit string
ld        $r2                           # $r2 holds 8-bit string containing the 4-bit substring
sll       4, $r2                        # sll 4
srl       1, $r2                        # srl 1
srl       1, $r2                        # srl 1
srl       1, $r2                        # srl 1
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)
movl      $r3, $r8                      # $r8 = $r3
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
jump      finish_2                      # newPC = PC + 17
ld        $r4                           # $r4 = 8 bit string
loop_next_bit:
be        0, $r2, $r4         # if ($r2 == $r4), then jump to equal_string
jump      equal_string                  # newPC = PC + 6
srl       1, $r4                        # srl 1
addi      0, $r1                        # $r1++
bge       $r1, $r7                      # if ($r1 ≥ $r7), then jump to end_string
jump      end_string          # newPC = PC + 6
jump      loop_next_bit                 # newPC = PC + (-6)
equal_string:
addi      0, $r5                        # $r5++ (matched_counter)
addi      1, $r3                        # $r3-- (strings_checked_counter)
addi      0, $r8                        # $r8++ (address of string array)
jump      loop_next_string              # newPC = PC + (-15)
end_string:
addi      0, $r8                        # $r8++ (address of string array)
addi      1, $r3                        # $r3-- (strings_checked_counter)
jump      loop_next_string              # newPC = PC + (-18)
finish_2:
movh      $r7, $r8                      # $r8 = 5
addi      0, $r8                        # $r8++, so now $r8 = 6
addi      0, $r8                        # $r8++, so now $r8 = 7
store     $r5                           # store matched_counter into memory address 7
halt

program_3:
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
sll       1, $r8                        # sll 1, so now $r8 = 8
addi      1, $r8                        # $r8--, so now $r8 = 7
movh      $r8, $r3                      # $r3 = 7
addi      0, $r3                        # $r3++, so now $r3 = 8
addi      0, $r3                        # $r3++, so now $r3 = 9
sll       1, $r3                        # sll 1, so now $r3 = 18
addi      0, $r3                        # $r3++, so now $r3 = 19
movl      $r3, $r4                      # $r4 = $r3
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
addi      0, $r8                        # $r8++, so now $r8 = 2
sll       6, $r8                        # $r8 = 128, $r8 = memory address of int array
ld        $r5                           # $r5 = first integer to compare distance
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:
addi      0, $r8                        # $r8++
ld        $r6                           # $r6 = next second integer to compare distance
diff      $r6, $r5                      # $r6 = |$r6 - $r5|
bge       $r6, $r7                      # if ($r6 ≥ $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++
be        $r2, $r3                      # if ($r2 == $r3), then jump to end_j, else jump to j_loop
jump      end_j                         # newPC = PC + 2
jump      j_loop                        # newPC = PC + (-10)
end_j:
addi      1, $r4                        # $r4--
udiff     $r8, $r4                      # $r8 = |$r8 - $r4|
ld        $r5                           # $r5 = next first integer to compare distance
srl       8, $r2                        # srl 8, so clear $r2
srl       8, $r1                        # srl 8, so clear $r1
addi      1, $r3                        # $r3--
bge       $r1, $r3                      # if ($r1 ≥ $r3), then jump to finish_3
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
addi      0, $r8                        # $r8++, so now $r8 = 2
sll       6, $r8                        # sll 6, so $r8 = 128
addi      1, $r8                        # $r8--, so now $r8 = 127
store     $r7                           # store minimum_distance into memory address 127
halt
```

MACHINE CODE

```
0 : 11101101
1 : 11101010
2 : 11101011
3 : 11101111
4 : 11101100
5 : 10010111
6 : 10000000
7 : 10010111
8 : 10000001
9 : 10010101
10 : 10100001
11 : 11001100
12 : 01101101
13 : 11001010
14 : 10110001
15 : 11001000
16 : 10011001
17 : 00010110
18 : 00010011
19 : 00011011
20 : 11000101
21 : 11100001
22 : 00110011
23 : 11110011
24 : 01111000
25 : 01001100
26 : 11110001
27 : 10001000
28 : 00010001
29 : 11000001
30 : 00011100
31 : 00010011
32 : 00011011
33 : 01000001
34 : 11101111
35 : 10010111
36 : 10010111
37 : 10010111
38 : 10000001
39 : 10100001
40 : 11001011
41 : 11101011
42 : 11101100
43 : 01101101
44 : 11001010
45 : 10110001
46 : 11001000
47 : 10011001
48 : 00010110
49 : 00010011
50 : 00011011
51 : 11000101
52 : 11100001
53 : 00110011
54 : 11110011
55 : 01111000
56 : 01001100
57 : 11110001
58 : 10001000
59 : 00010001
60 : 11000001
61 : 00010110
62 : 00010011
63 : 00011011
64 : 11101111
65 : 10010111
66 : 10010111
67 : 10010111
68 : 10010111
69 : 01110100
70 : 10010111
71 : 01110011
72 : 00000000
73 : 11101111
74 : 10010111
75 : 10010111
76 : 10010111
77 : 10010111
78 : 11101101
79 : 01110101
80 : 01110101
81 : 01110101
82 : 00000000

// Program 2
83 : 11101101
84 : 11101111
85 : 10010111
86 : 11110111
87 : 10010111
88 : 10010111
89 : 10000001
90 : 11111000
91 : 11100001
92 : 11100001
93 : 11100001
94 : 11100001
95 : 11101100
96 : 11101010
97 : 10010010
98 : 01001110
99 : 10011110
100 : 11111101
101 : 00111011
102 : 11100111
103 : 11101000
104 : 10100010
105 : 11001001
106 : 10000011
107 : 01010111
108 : 11001000
109 : 11100011
110 : 10010000
111 : 01100011
112 : 11001000
113 : 11000110
114 : 10010100
115 : 10011010
116 : 10010111
117 : 11000011
118 : 10010111
119 : 10011010
120 : 11000001
121 : 01001011
122 : 10010111
123 : 10010111
124 : 01110100
125 : 00000000

//Program 3
126 : 11101111
127 : 10010111
128 : 11110111
129 : 11110011
130 : 10011111
131 : 01001101
132 : 10010010
133 : 10010010
134 : 11110001
135 : 10010010
136 : 00111001
137 : 11101000
138 : 11101001
139 : 11101111
140 : 10010111
141 : 10010111
142 : 11111111
143 : 10000100
144 : 10010111
145 : 10000101
146 : 10010001
147 : 00101010
148 : 01000110
149 : 10010000
150 : 10010111
151 : 10000101
152 : 00101010
153 : 01101111
154 : 11000111
155 : 01000110
156 : 10010001
157 : 01011110
158 : 11000111
159 : 11000100
160 : 10011011
161 : 11011101
162 : 10000100
163 : 11101001
164 : 11101000
165 : 10011010
166 : 01100000
167 : 11000111
168 : 11000000
169 : 11101111
170 : 10010111
171 : 10010111
172 : 11111111
173 : 10011111
174 : 01110110
175 : 11010000
```