EECS-361 COMPUTER ARCHITECTURE- I
GROUP PROJECT-2 - PIPELINED PROCESSOR

GROUP 5- MEMBERS

MATTHEW LENK
YUE WANG
NITHIN KRISHNAN

CONTENTS:

Introduction:

Pipelining is the method of splitting instructions into several small portions and executing them in every clock cycle. However, instead of waiting for the current instruction to finish before starting execution of the next one, the first stage of the second instruction starts executing on the second clock cycle (first instruction is executing its second stage). Pipelining a processor doesn't improve latency of every instruction, but it improves the overall throughput of the system.

Design:

The pipelined processor is realised with the help of ModelSim software, with the test programs loaded into the memory unit of the processor. It consists mainly of an Instruction Fetch Unit, a Register file, an Arithmetic and Logic Unit, a Memory Unit, hazard detection and control, a forwarding unit and registers in between the stages of the pipeline.

The instruction set for this processor contains the following instructions:

1. • arithmetic: add, addi, addu, sub, subu
2. • logical: and, or, sll
3. • data transfer: lw, sw
4. • conditional branch: beq, bne, bgtz, slt, sltu.

The different types of instructions in the MIPS architecture is

1. R-type instruction
2. I-type instruction
3. J-type instruction

The instruction fetch unit gets the new instruction from the Program Counter and decodes it.
The register file stores the input operands for the operation to be performed as well any data that has to be written back.
There are additional registers in-between the different stages of the pipeline in order to store the results. In the case of a hazard or a pipeline flush, the previous address of data is still available and the program counter does not have to be reset again.
The hazard detection unit, detects 2 types of data hazards(type A and B) and returns a value back. Which helps in deciding where the signal should be forwarded and which signals should be forwarded.
The ALU performs the required arithmetic or logic operation ,which is selected by an ALU control, and produces an output.

Finally, data is stored in the memory unit, which can then be written back to the register file depending on the instruction.

New Main Entities:

(The provided Single Cycle Processor was used for the non-pipeline components)

Registers between Stages: The registers between the various stages of the pipeline take in the previous stage values as inputs and provide them as inputs to the next stage on the next clock edge. This ensures that only the latest values are stored in the registers pertaining to the corresponding stage, as the stages execute concurrently. Here is an example of our pipeline registers. This is the register between the instruction fetch and decode stages.

```
entity Register_IF_ID is
port(
clk : in std_logic;
IF_ID_write_enable : in std_logic;
pc_plus4_in :in std_logic_vector(31 downto 0);
opcode,func :in std_logic_vector(5 downto 0);
Rs,Rt,Rd,shamt: in std_logic_vector(4 downto 0);
imm16: in std_logic_vector(15 downto 0);

pc_plus4_out :out std_logic_vector(31 downto 0);
opcode_out,func_out :out std_logic_vector(5 downto 0);
Rs_out,Rt_out,Rd_out,shamt_out: out std_logic_vector(4 downto 0);
imm16_out: out std_logic_vector(15 downto 0));
end entity;


--Inside the architecture
inst_in(31 downto 26)<=opcode;
inst_in(20 downto 16)<=Rt;
inst_in(25 downto 21)<=Rs;
inst_in(15 downto 11)<=Rd;
inst_in(10 downto 6)<=shamt;
inst_in(5 downto 0)<=func;
inst_in(15 downto 0)<=imm16;

opcode_out<=inst_out(31 downto 26);
```

```
Rt_out<=inst_out(20 downto 16);
Rs_out<=inst_out(25 downto 21);
Rd_out<=inst_out(15 downto 11);
shamt_out<=inst_out(10 downto 6);
func_out<=inst_out(5 downto 0);
imm16_out<=inst_out(15 downto 0);

store_PC: Reg_32_bit port map (clk, IF_ID_write_enable, pc_plus4_in, pc_plus4_out);
store_instruction: Reg_32_bit port map (clk, IF_ID_write_enable, inst_in, inst_out);
```

Hazard Detection and Forwarding: Our hazard detection and forwarding was based upon the book's recommended design. This design allows for us to deal with both execution and memory hazards. Hazard detection is based upon the read and write registers and the write enable signals at various points of the processor. The forwarding signals went into multiplexers in the processor to decide what the inputs should be to the ALU.

1. *EX hazard:*

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

2. *MEM hazard:*

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Screen shots taken form Computer Organization and Design 5th Edition by Patterson and Hennesy

```
nz0: not_zero_compare port map(rd_EX_MEM, a0);
ec0: equality_checker port map(rd_EX_MEM, rs_ID_EX, a1);
and0: and_gate port map(a0, a1, a2);
and1: and_gate port map(WriteEnable_EX_MEM, a2, a3); -- a3 means ex hazard FA=10
```

```
n0: not_gate port map(a3,b0); -- not a ex  hazard
nz1: not_zero_compare port map(rd_EX_MEM, b1);
ec3: equality_checker port map(rd_MEM_WB, rs_ID_EX, b2);

and4: and_gate port map(WriteEnable_MEM_WB, b1, b3);
and5: and_gate port map(b3, b0, b4);
and6: and_gate port map(b4, b2, b5); --b9 means mem hazard FA=01

A(1)<= a3; --Ex Hazard
A(0) <=b5;--Mem Hazard



nz2: not_zero_compare port map(rd_EX_MEM, c0);
ec4: equality_checker port map(rd_EX_MEM, rt_ID_EX, c1);
and7: and_gate port map(WriteEnable_EX_MEM, c0, c2);
and8: and_gate port map(c2, c1, c3);

n1: not_gate port map (c3, d0);
nz3: not_zero_compare port map (rd_MEM_WB, d1);
ec5: equality_checker port map(rd_MEM_WB, rt_ID_EX, d2);
and9: and_gate port map(d0, d1, d4);
and10: and_gate port map(d4,d2, d5);

B(1)<= c3;
B(0)<= d5;
```

Generating Stalls:

When we detect either a load word or a branch instruction, our design was to have a single cycle stall.

Top-Level Entity: The top level entity(datapath), connects all the components together. As expected, this assignment created a lot more connections than the single cycle processor. Clear labeling was a necessity to keep track of each signal and ensure that it was connecting the correct components.

VHDL IMplementation:

```
begin
--IF STAGE
w1:  mux_32 port map(Branch,pc_plus4,branchaddress,nextAddress);
w2:  pc port map(clk, reset, start, nextAddress, instAddress);
w3:  inst_mem generic map (mem_file=>mem_file)
```

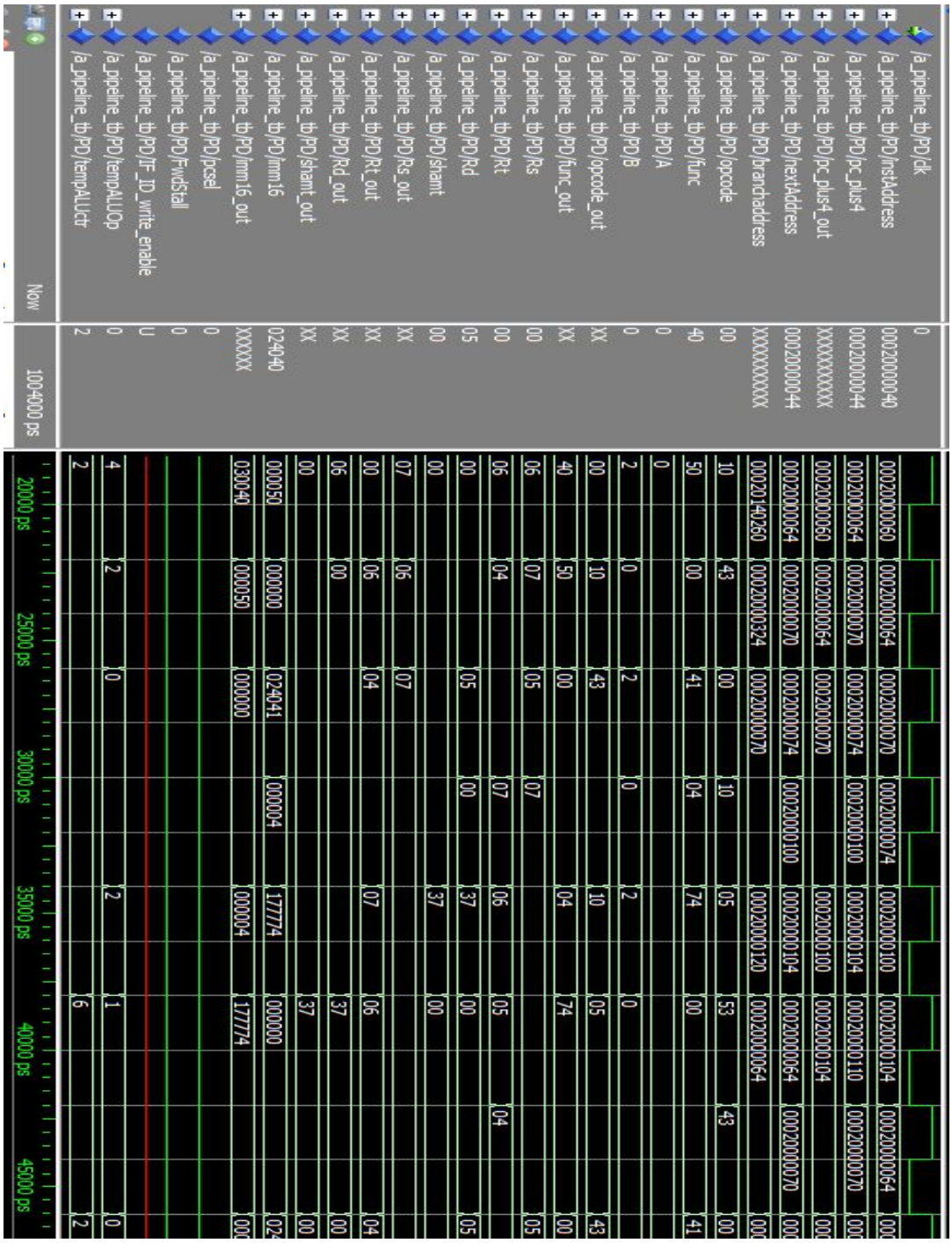
```
            port map (instAddress, opcode, func, Rs, Rt, Rd, shamt, imm16);
w4:  full_adder_32bit port map (cin=>'0', x=>instAddress, y=>x"00000004",
z=>pc_plus4);
w5:  stall_generate port map(clk,reset,opcode,stall_signal);
w6:  Register_IF_ID port
map(clk,stall_signal,pc_plus4,opcode,func,Rs,Rt,Rd,shamt,imm16,
            pc_plus4_out, opcode_out,
func_out,Rs_out,Rt_out,Rd_out,shamt_out,imm16_out);


--ID STAGE
w7:  ControlUnit port
map(opcode_out,tempALUOp,tempRegDst,tempALUSrc,tempMemToReg,
    tempRegWrite,tempMemRead,tempMemWrite ,tempBranchNE ,tempBranch
,tempBranchTZ,
    tempExtOP);
w8:  register_file port map(clk,reset,tempRegWrite,Rs_out, Rt_out, WB_Rw,
WB_data_in, tempA,tempB);
w9:  ALUCU port map(tempALUOp, func_out, tempALUctr);
w10: register_ID_EX port
map(clk,tempALUOp,tempRegDst,tempALUSrc,tempMemToReg,tempRegWrite,

tempMemRead,tempMemWrite,tempBranchNE,tempBranch,tempBranchTZ,tempExtOP
,shamt_out,
    pc_plus4_out,tempA,tempB,imm16_out,Rt_out,Rd_out,tempALUctr,


EX_ALUOp,EX_RegDst,EX_ALUSrc,EX_MemToReg,EX_RegWr,EX_MemRead,EX_M
emWr,EX_BranchNE,

EX_Branch,EX_BranchTZ,EX_ExtOP,EX_shamt,EX_pc,EX_tempA,EX_tempB,EX_imm
16,EX_Rt,EX_Rd,
    EX_ALUctr);


--EX STAGE
w11: sign_ext port map(EX_imm16, signEX_imm32);
w12: Extender port map(EX_imm16,EX_eximm32);
w13: mux_32 port map(EX_ExtOp,EX_eximm32,signEX_imm32,MUX_EX_imm32);
```

```
w14: mux_32 port map(EX_ALUSrc,EX_tempB,MUX_EX_imm32,EX_imm32);
w15: mux4to1 port map(A,EX_imm32,WB_data_in,MEM_Forward,
    "00000000000000000000000000000000",opA);
w16: mux4to1 port map(B,EX_imm32,WB_data_in,MEM_Forward,
    "00000000000000000000000000000000",opB);
w17: ALU port
map(EX_ALUctr,opA,opB,EX_shamt,EX_carryout,EX_overflow,EX_zero,EX_result);
w18: not_gate port map(EX_zero,Nzero);
w19: not_gate port map(x=>EX_carryout,z=>EX_carryout_not);
w20: and_gate PORT MAP(EX_BranchNE, Nzero, BranchNE_out);
w21: and_gate PORT MAP(EX_Branch, EX_zero, Branch_out);
w22: and_gate PORT MAP(EX_BranchTZ, EX_carryout_not, BTZ_out_1);
w23: and_gate PORT MAP(BTZ_out_1, Nzero, BranchTZ_out);
w24: or_gate PORT MAP(BranchNE_out, Branch_out, Btemp);
w25: or_gate PORT MAP(BranchTZ_out, Btemp, EX_BranchTaken);
w26: mux_5 port map(EX_RegDst, EX_Rt, EX_Rd, EX_Rw);

adder_shift(31 downto 2)<=signEX_imm32(29 downto 0);
adder_shift(1 downto 0)<="00";

w27: adder_32 port map(EX_pc,adder_shift,branchaddress);
w28: and_gate port map(EX_BranchTaken,EX_Branch,Branch);
w29: Register_EX_MEM port
map(clk,EX_result,EX_tempB,EX_Rw,EX_MemToReg,EX_RegWr,
    EX_MemRead,EX_MemWr,


MEM_result,MEM_tempB,MEM_Rw,MEM_MemToReg,MEM_RegWr,MEM_MemRead,
MEM_MemWr);



--MEM STAGE
w30: or_gate port map(MEM_MemWr,MEM_MemRead,MEM_MemRw);
w31: and_gate port map(clk,MEM_MemRw,MEM_MemClk);
w32: data_mem generic map(data_mem_file=>mem_file)
          port map(MEM_MemClk, MEM_MemRead, MEM_MemWr, MEM_result,
MEM_tempB, MEM_MemOut);
```

```
w33: mux_32 port map(MEM_MemToReg, MEM_result, MEM_MemOut,
MEM_Forward);
w34: Register_MEM_WB port
map(clk,MEM_result,MEM_MemOut,MEM_Rw,MEM_MemToReg,MEM_RegWr,
    WB_result,WB_MemOut,WB_Rw,WB_MemToReg,WB_RegWr);

--WB STAGE
w35: mux_32 port map(WB_MemToReg, WB_result, WB_MemOut, WB_data_in);

Forwarding: forwarding_unit port map(EX_Rs, EX_Rt, MEM_Rw, WB_Rw,
MEM_RegWr, WB_RegWr, MEM_MemToReg, A, B, stallF);

end architecture structural;
```

Simulation:

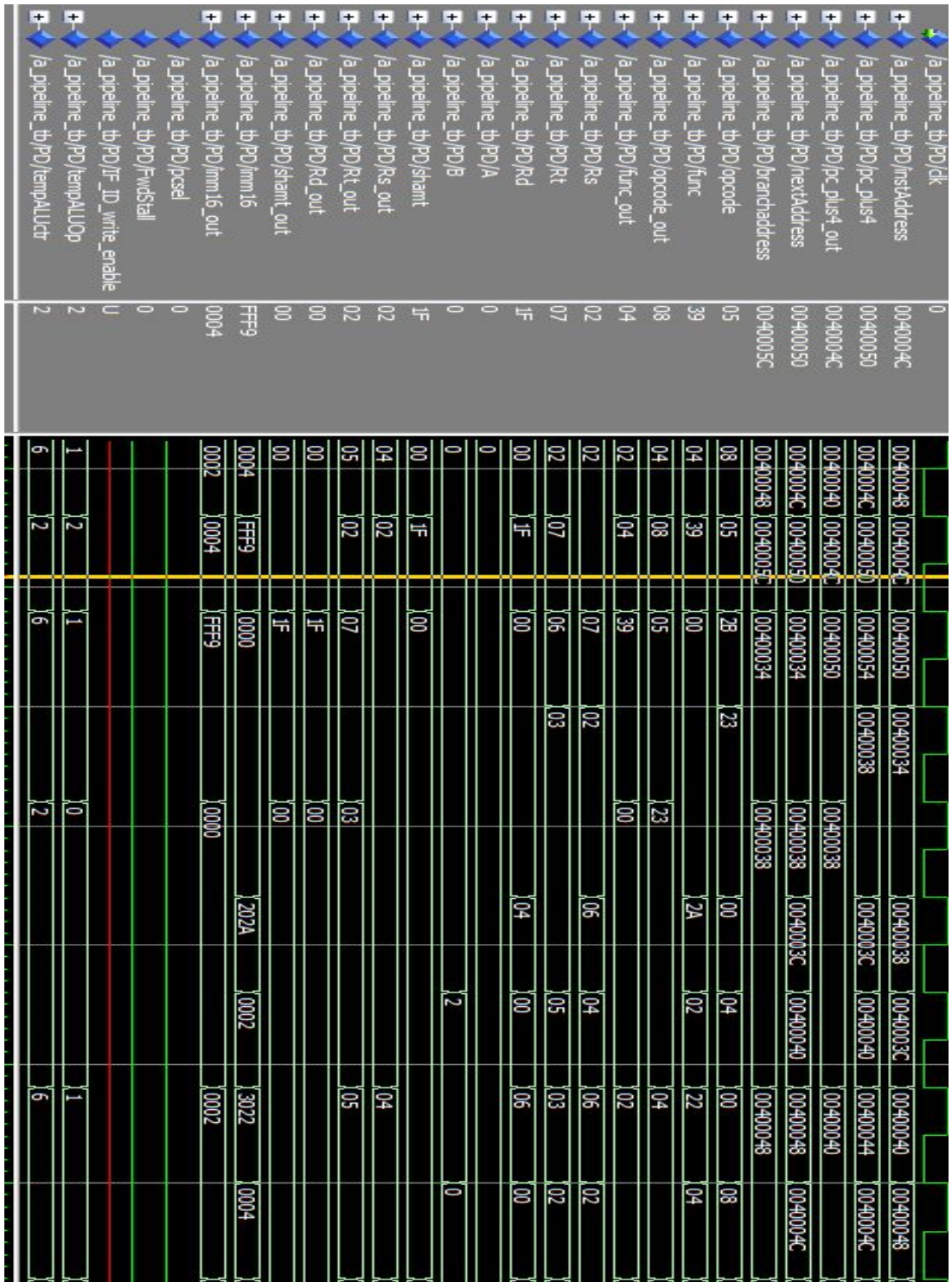
Unsigned Sum



Screen Capture of Unsigned Sum

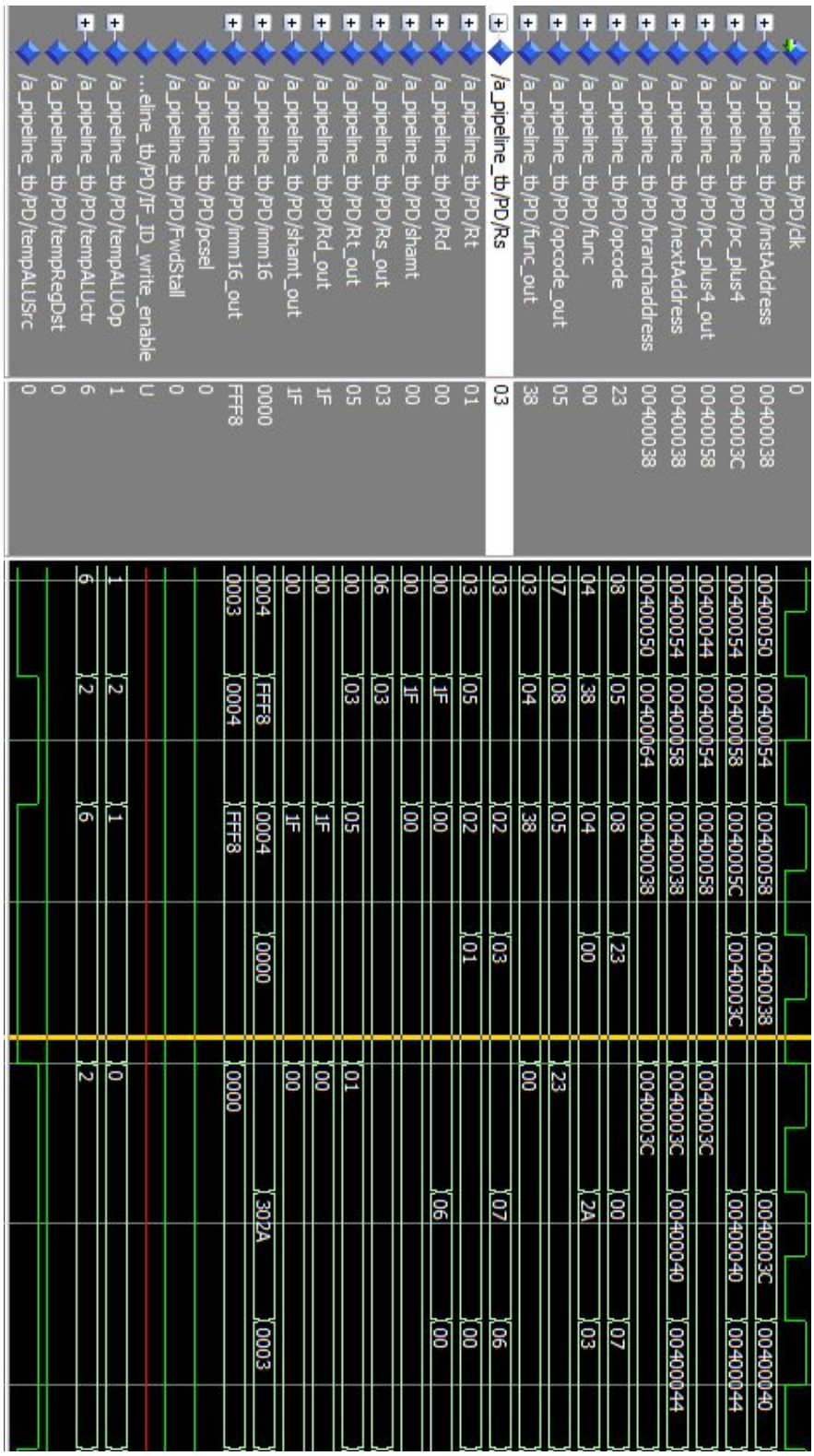
A and B refer to execution forward and memory forward respectively

Bills Branch



Screen capture of bills branch

***Sort Branch*:**



Screen Capture of *Sort Branch*

Results:

Our pipeline structure and the stalling of our programs appeared to be working correctly. There was some issues with our forwarding and the programs end up in infinite loops so they never reach completion.