McCormick School of Engineering

CE361 Computer Architecture Project - Group 11

Pipelined Processor Design

Xi Chen

Yankai Jiang

2021 December

1. Introduction of Project

In this project, we were asked to design, implement, and test a pipelined processor that is capable of handling the below subset of the MIPS instruction set:

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

Also, we implement hazard detector and forwarding in the pipelined processor, consider different types of hazard.

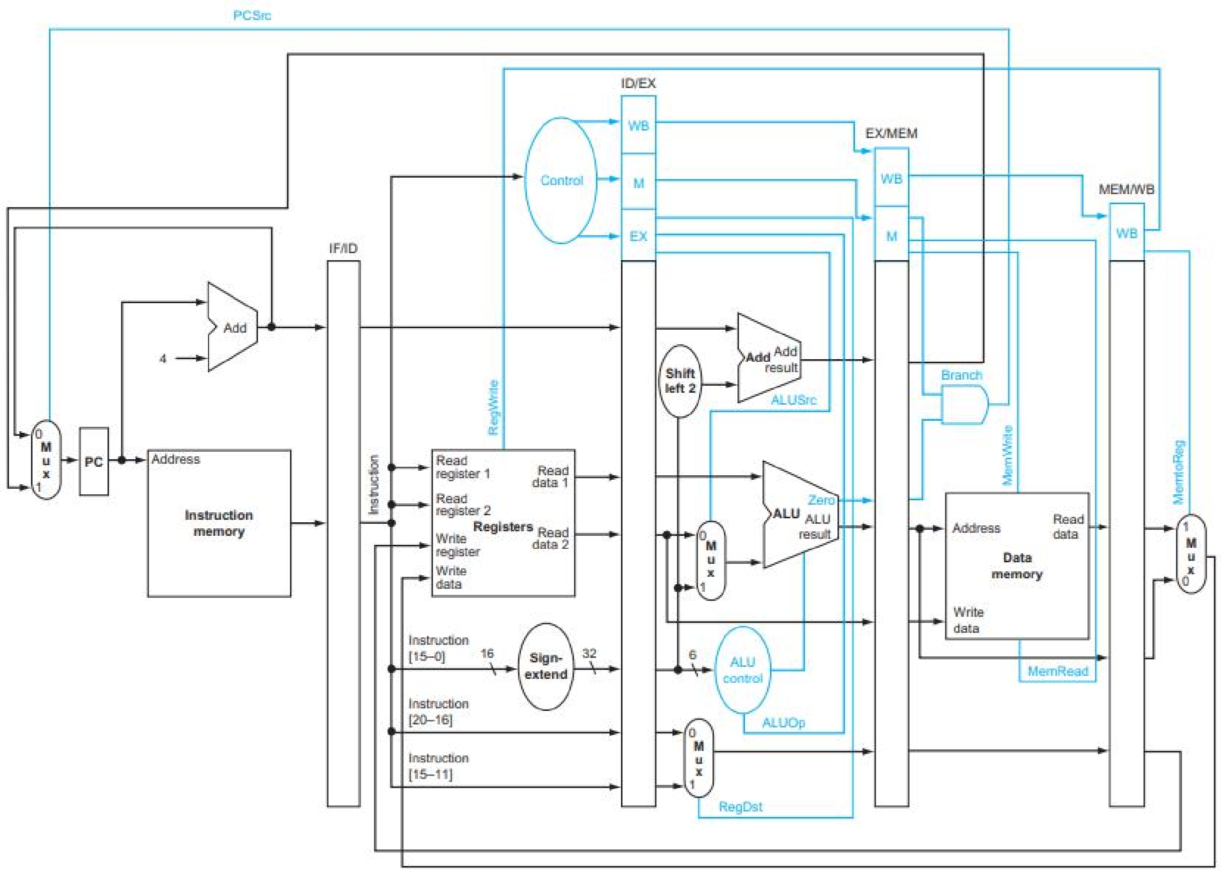


Figure 1: *Pipelined Processor Design (from textbook)*

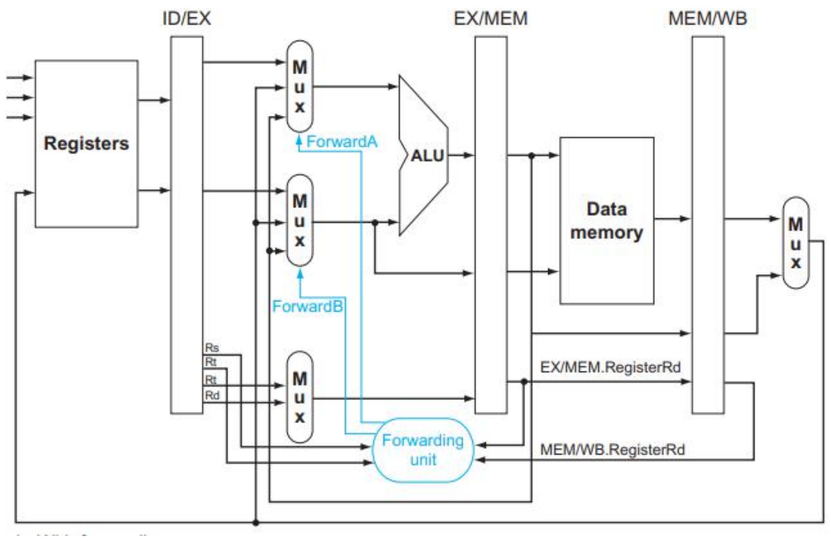


Figure 2: *Design with Forwarding (from textbook)*

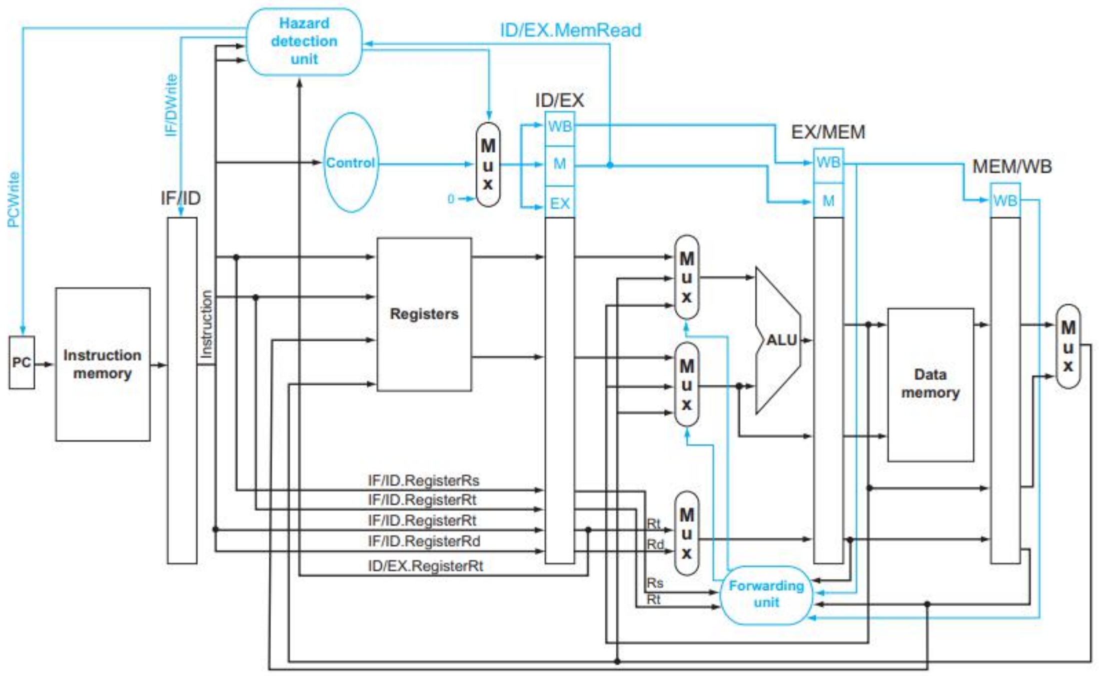


Figure 3: *Design with Hazard Detection Unit (from textbook)*

We have the correct ALU file and the lib which is used for building processor.

Table1: *Main Components in Design*

Component	File Name
IF Stage	stage1_if.v
ID Stage	stage2_id.v
EX Stage	stage3_ex.v
MEM Stage	stage4_mem.v
WB Stage	stage5_wb.v
Forwarding Unit	forwarding.v
Hazard Detect Unit	hazard_dect.v
Main Control Unit	main_control.v
ALU Control Unit	alu_control.v
Pipelined Processor Control Path	pp_ctrl.v
Pipelined Processor	pp_top.v
Register	register_file.v
Instruction Memory	sram_fix.v
Data Memory	sram_fix.v

2. Pipelined Processor Control Design

The design of main control and the design of ALU control are similar with single cycle processor design. In piplined processor, we must consider the procrastination of each control signal (almost every control signal must “go through” a register). Also, some control signals may be “flushed” because of multiple types of hazard, and the “flush” signal is controlled by the hazard detect unit.

2.1 Main Controller

Control Unit Operation (Based on OpCode):

Table2: *Truth Table of Main Controller*

Operation	Opcode	RegDst	RegWr	ALU Src	MemWr	MemReg	ExtOp	Branch	ALU op
R Type	000000	1	1	0	0	0	x	0	100
BEQ	000100	0	0	0	0	0	1	1(==0)	001
BNE	000101	0	0	0	0	0	1	1(!=0)	001
BGTZ	000111	0	0	0	0	0	1	1(>0)	001
AddI	001000	0	1	1	0	0	0	0	000
LW	100011	0	1	1	0	1	1	0	000
SW	101011	X	0	1	1	X	1	0	000

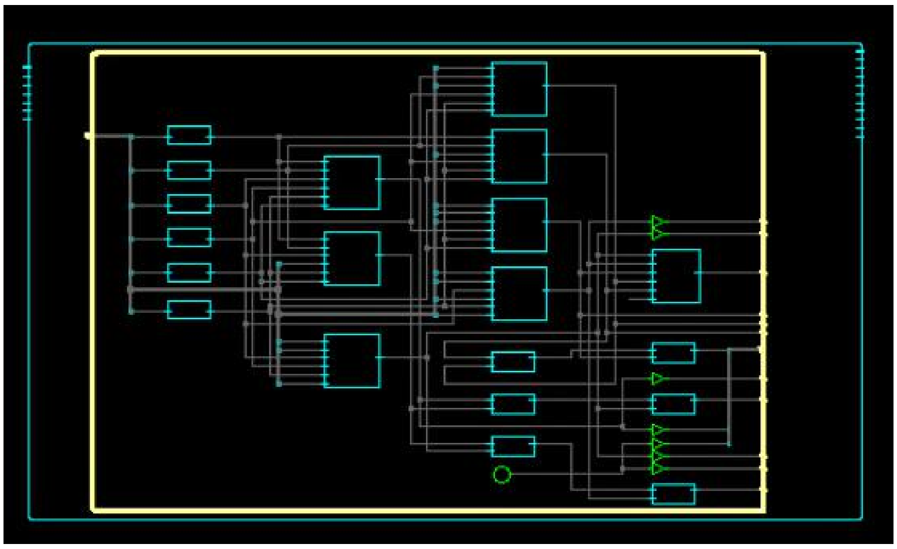


Figure 4: *Schematic Tracer of Main Controller*

The main controller is similar to the SCP main controller, the difference is the extender for branch, so the ext_op signal for branch is 1 (signed extended).

2.2 ALU Controller

Table2: *Truth Table of Main Controller*

Operation	ALU op	func	ALU Ctrl
ADD	100	100000	010
ADDU	100	100001	100
SUB	100	100010	110
SUBU	100	100011	110
AND	100	100100	000
OR	100	100101	001
SLL	100	000000	101
SLT	100	101010	011
SLTU	100	101011	111
LW	000	X	010
SW	000	X	010
BEQ	001	X	110
BNE	001	X	110
BGTZ	001	X	110
ADDI	000	X	010

ALU controller takes operation signals from control unit and the *function* field of each instruction as inputs and generates the corresponding signals for ALU.

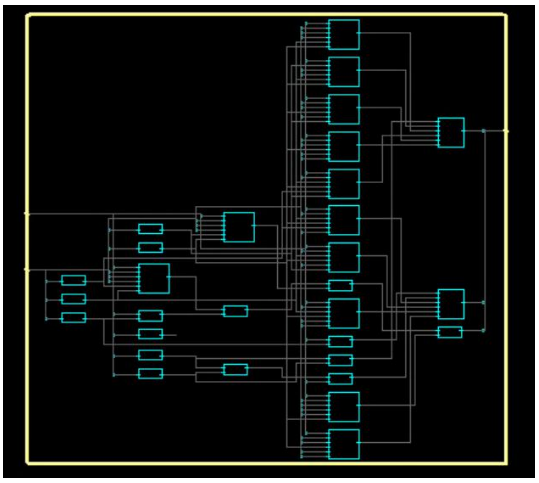


Figure 5: *Schematic Tracer of ALU Controller*

A control unit should take the *opcode* of the instructions as input and generate control signals. Also, the generated ALU Op code should become the input for the ALU controller, and ALU controller generate the ALU control signal to tell ALU which instruction should operate. The design of controller is quite simple, just implement PLA using the truth table.

2.3 Pipelined Processor Controller

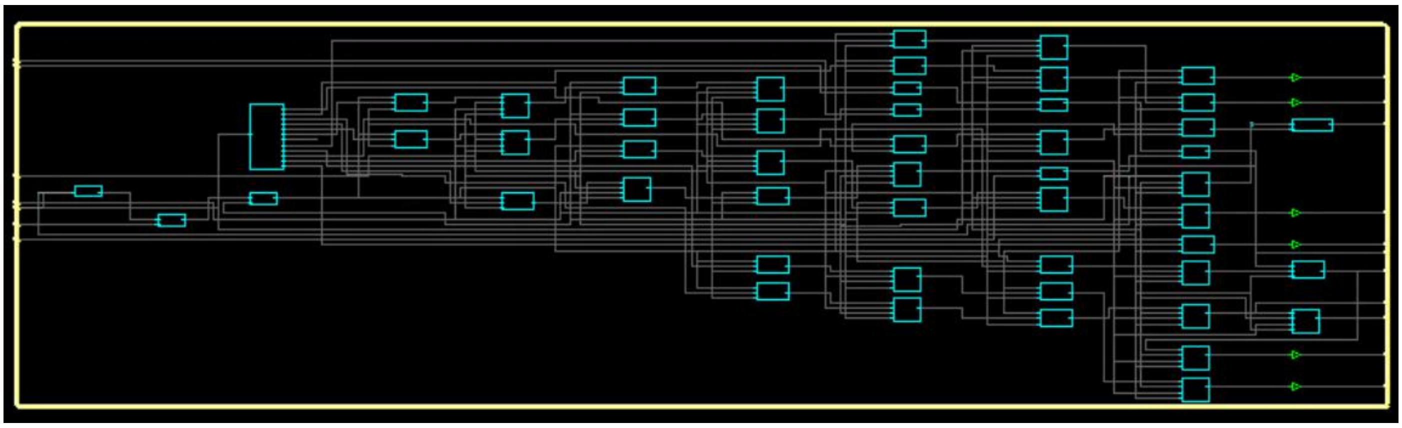


Figure 6: *Schematic Tracer of Pipelined Processor Controller*

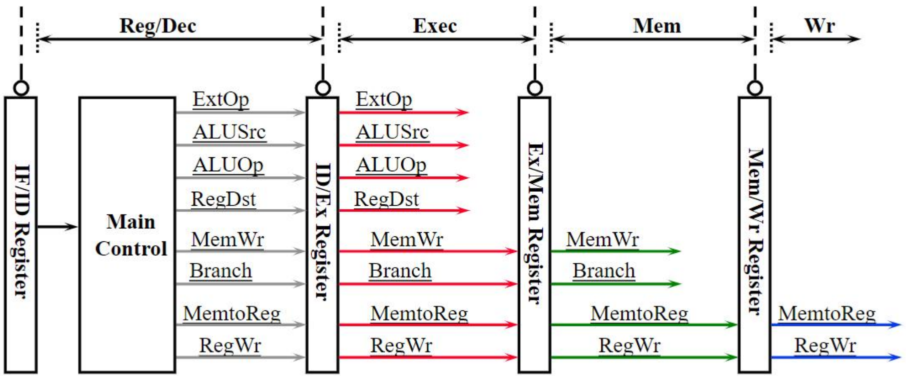


Figure 7: *Transmission of Control Signals in PP Controller (Lec12 Slides 3)*

If instruction is a all-zero instruction or there is a clear signal provided by the hazard detect unit, the control signals in ID stage will be set 0. Also if branch signal in Mem stage is one, set all control signals in EX stage to be zero. (stall)

2.4 Pipelined Processor Controller Wave Form

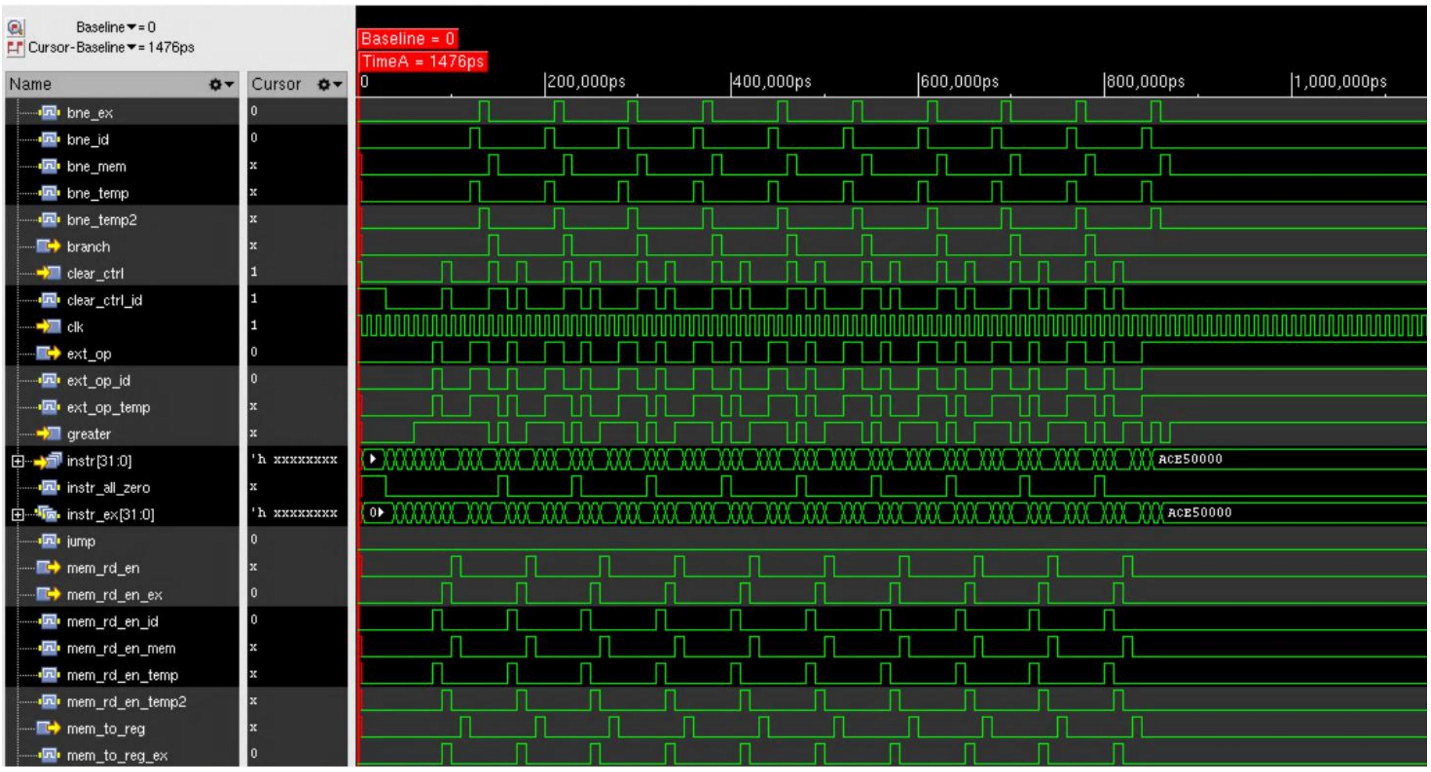


Figure 8: *Part of Wave Form (pp controller)*

3. Stage 1 IF Design

IF stage contain 2 parts, the first part is PC controller, and second part is IF/ID register controlled by the clock signal. The inputs and outputs are:

Table3: *Inputs of IF Stage*

Inputs	clk	rstb	pc_src	pc_plus4_plusimm16	hold_if	hold_pc	if_flush
Outputs	pc_plus4				instr		

Declaration： *clk* is clock signal, *rstb* is reset signal, *pc_src* is branch signal, *pc_plus4_plusimm16* is the immediate pc. *hold_if, hold_pc, if_flush* are signals generated from hazard detect unit. *pc_plus4* is the new pc, *instr* is the instruction which is fetched.

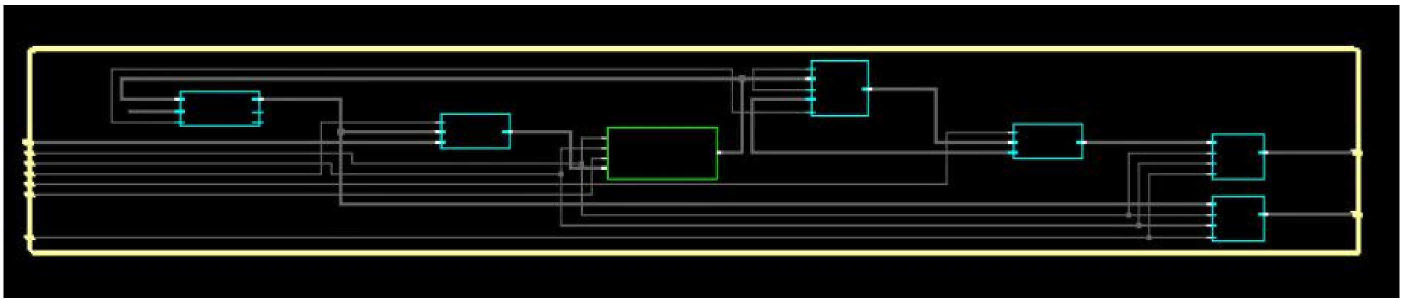


Figure 9: *Schematic Tracer of IF Stage*

The IF stage only contains the datapath, control path is designed in PP controller.
The IF stage consists of the Instruction Memory, an adder which adds a constant 4, and a MUX select from a feedback address or the next instruction. The feedback address may come from IF stage by adding the immediate number in the case of forwarding, or come from the EX/MEM registers in regular cases.

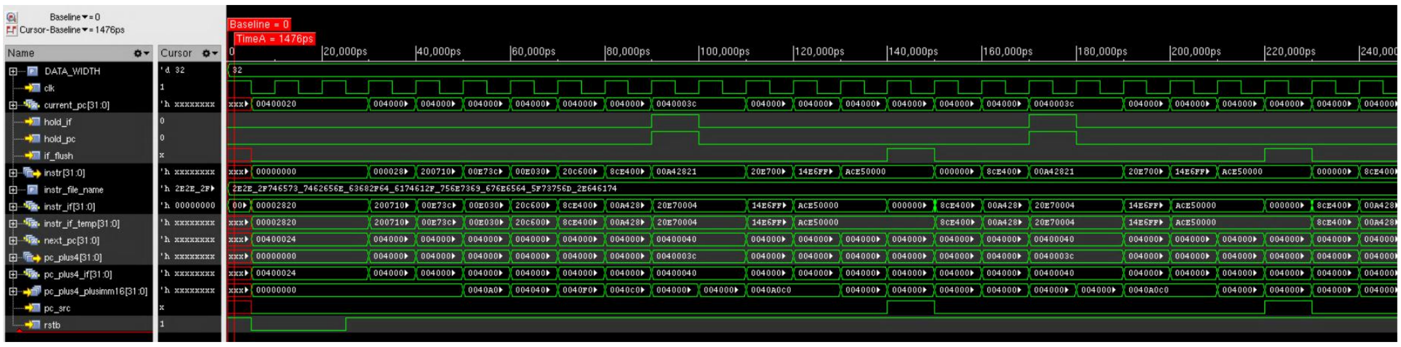


Figure 10: *Part of Wave Form (IF Stage)*

4. Stage 2 ID Design

The inputs are: *clk, rstb, reg_wr_en, pc_plus4_in, instr_in, reg_wr_addr, reg_wr_data, ext_ctrl, if_flush.*
The outputs are: *pc_plus4_out, regA_rd_data, regB_rd_data, imm_exted, regS_addr_id, regT_addr_id, regS_addr, regT_addr, regD_addr*
Declaration: *regS_addr_id, regT_addr_id* are used as inputs of hazard detect unit.

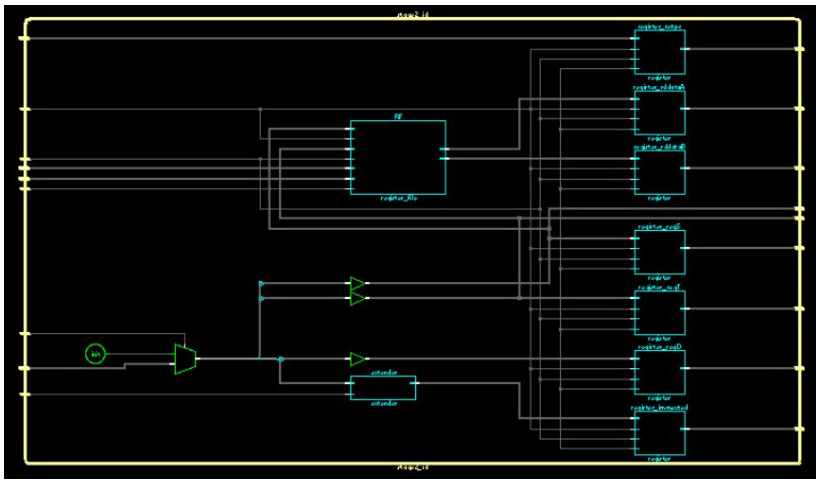


Figure 11: *Schematic Tracer of ID Stage*

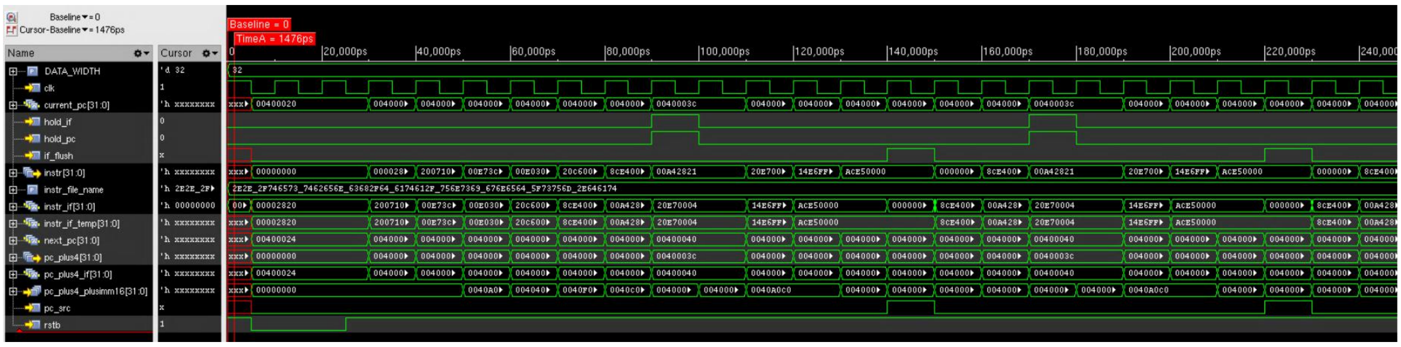


Figure 12: *Part of Wave Form (ID Stage)*

5. Stage 3 EX Design

For this stage, we have a shifter and an adder to compute next instruction address, a MUX selects from immediate number and the second operand by AluSrc, a MUX controlled by RegDst for the write back data, and definitely the ALU and corresponding ALU controller. Also there are 2 MUX for selecting inputs of ALU according to the forwarding signals.

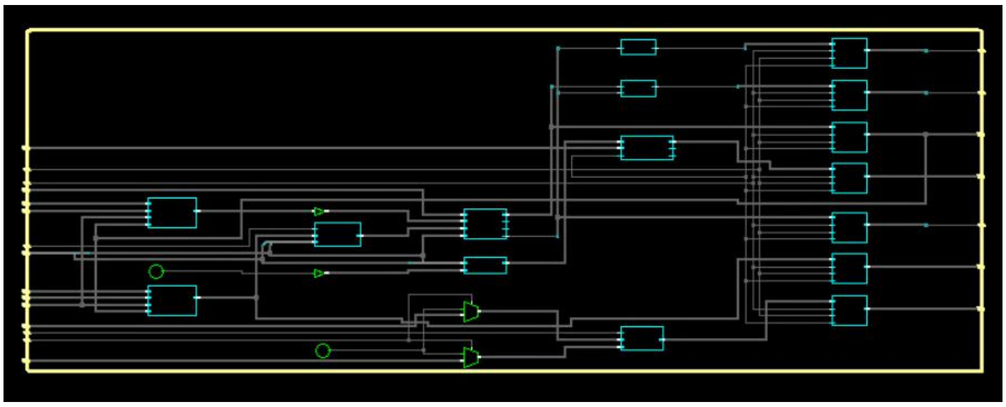


Figure 13: *Schematic Tracer of EX Stage*

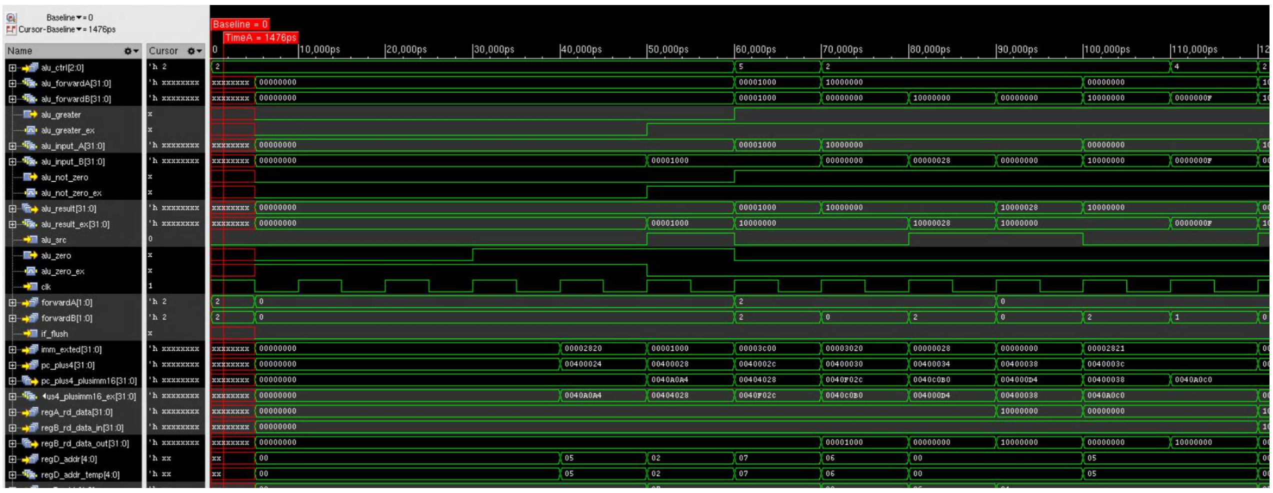


Figure 14: *Part of Wave Form (EX Stage)*

6. Stage 4 MEM Design

Basically only the data memory is in this stage.

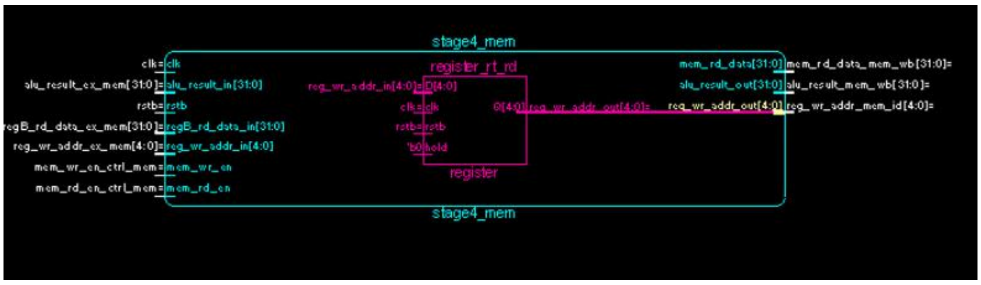


Figure 15: *Schematic Tracer of MEM Stage*

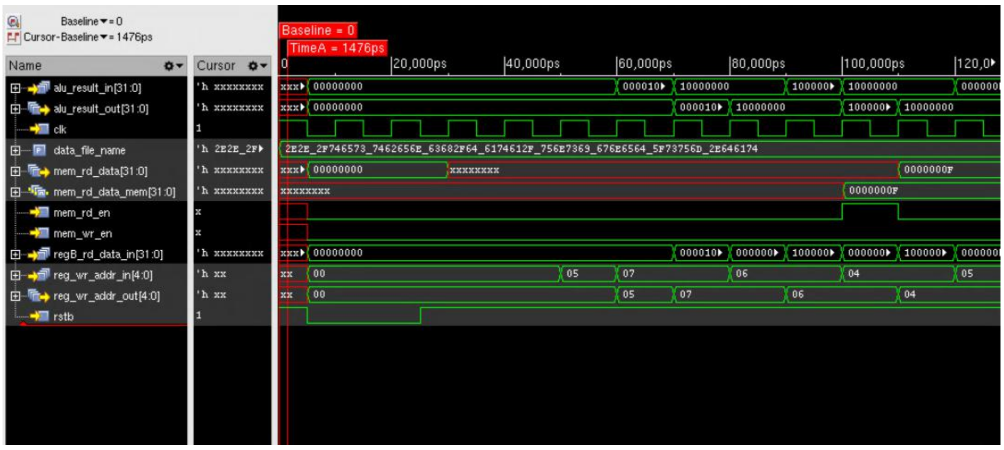


__Figure 16:__ Part of Wave Form (MEM Stage)

7. Stage 5 WB Design

In this stage, it's quite simple, it just includes a MUX to select the data which will be written in the register file.

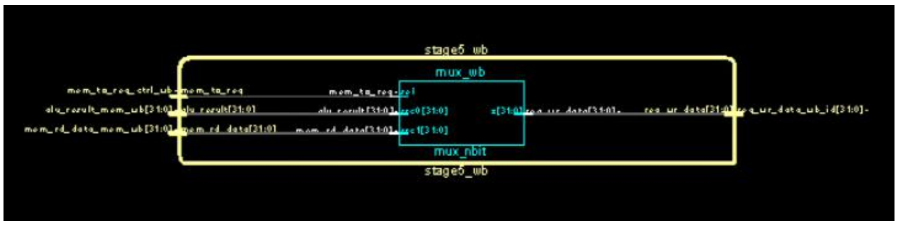


__Figure 17:__ Schematic Tracer of WB Stage

8. Forwarding Unit Design

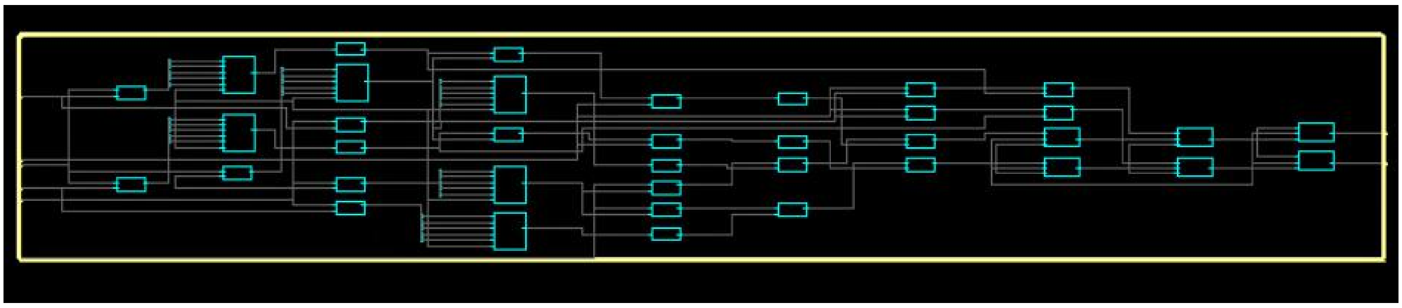


__Figure 18:__ Schematic Tracer of Forwarding Unit

a. 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

b. MEM Hazard

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd = 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

If MEM hazard and EX hazard both happened, forwarding signals will select EX hazard to be “10”.

9. Hazard Detection Unit Design

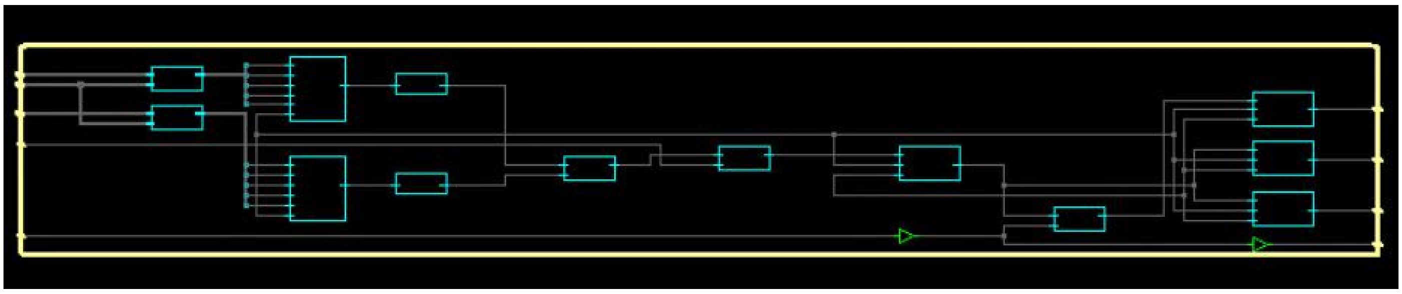
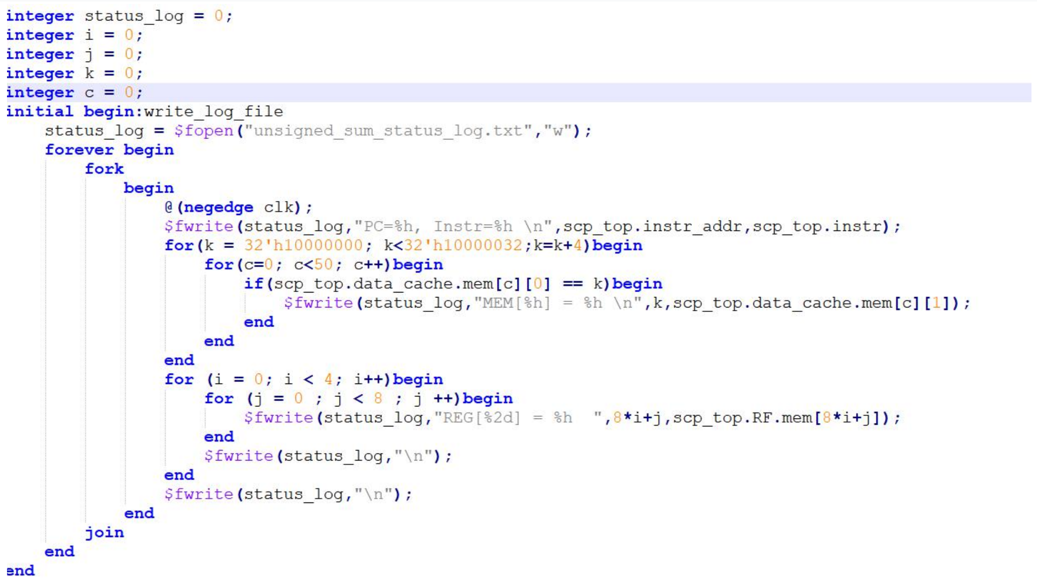


Figure 19: *Schematic Tracer of Hazard Detect Unit*

If branch =1, then flush =1; It will make instruction be all-zero instruction.
En = mem_rd_en_ex&&((regS_addr_id==regT_addr_ex)||(regT_addr_id==regT_addr_ex))
If then En =1, PC will be held, and IF/ID register will be held.
IF branch or En = 1 , then clear_ctrl = 1. All generated control signals will be cleared in the ID stage.

10. Verification

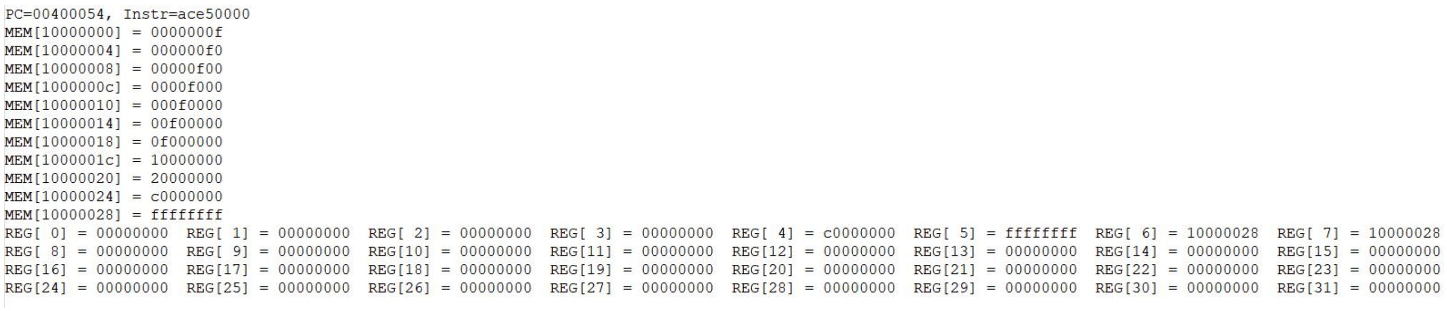
First, we manually calculated the values in all registers and memory of each cycle executed by the MIPS code. (You can check it in the verification folder) And just verify these values through ou design. And we set the cycle to be 10000, make sure that all MIPS codes are executed.



```
integer status_log = 0;
integer i = 0;
integer j = 0;
integer k = 0;
integer c = 0;
initial begin:write_log_file
    status_log = $fopen("unsigned_sum_status_log.txt","w");
    forever begin
        fork
            begin
                @(negedge clk);
                $fwrite(status_log,"PC=%h, Instr=%h \n",scp_top.instr_addr,scp_top.instr);
                for(k = 32'h10000000; k<32'h10000032;k=k+4)begin
                    for(c=0; c<50; c++)begin
                        if(scp_top.data_cache.mem[c][0] == k)begin
                            $fwrite(status_log,"MEM[%h] = %h \n",k,scp_top.data_cache.mem[c][1]);
                        end
                    end
                end
                for (i = 0; i < 4; i++)begin
                    for (j = 0 ; j < 8 ; j ++)begin
                        $fwrite(status_log,"REG[%2d] = %h  ",8*i+j,scp_top.RF.mem[8*i+j]);
                    end
                    $fwrite(status_log,"\n");
                end
                $fwrite(status_log,"\n");
            end
        join
    end
end
```

Figure 20: *Part of Testbench for Generating Results*

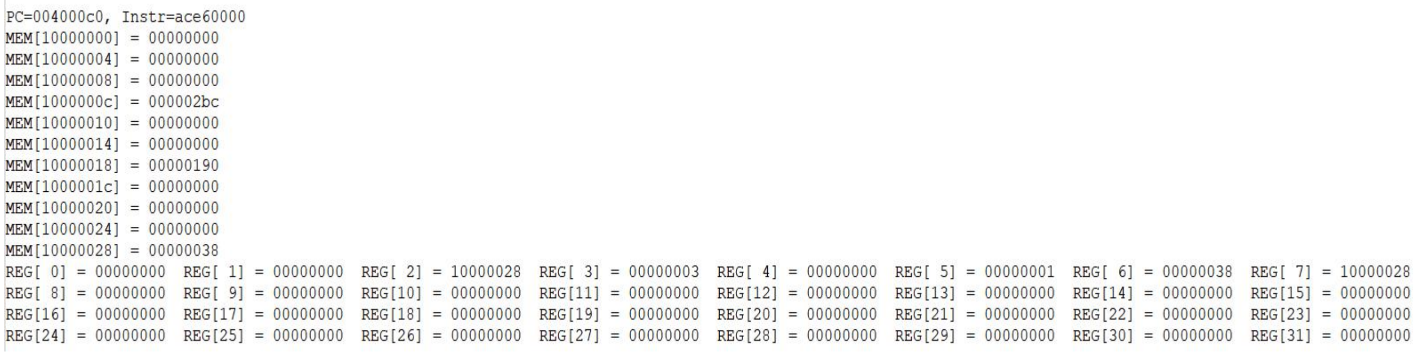
10.1 unsigned_sum.dat Verification



```
PC=00400054, Instr=ace50000
MEM[10000000] = 0000000f
MEM[10000004] = 000000f0
MEM[10000008] = 00000f00
MEM[1000000c] = 0000f000
MEM[10000010] = 000f0000
MEM[10000014] = 00f00000
MEM[10000018] = 0f000000
MEM[1000001c] = 10000000
MEM[10000020] = 20000000
MEM[10000024] = c0000000
MEM[10000028] = ffffffff
REG[ 0] = 00000000  REG[ 1] = 00000000  REG[ 2] = 00000000  REG[ 3] = 00000000  REG[ 4] = c0000000  REG[ 5] = ffffffff  REG[ 6] = 10000028  REG[ 7] = 10000028
REG[ 8] = 00000000  REG[ 9] = 00000000  REG[10] = 00000000  REG[11] = 00000000  REG[12] = 00000000  REG[13] = 00000000  REG[14] = 00000000  REG[15] = 00000000
REG[16] = 00000000  REG[17] = 00000000  REG[18] = 00000000  REG[19] = 00000000  REG[20] = 00000000  REG[21] = 00000000  REG[22] = 00000000  REG[23] = 00000000
REG[24] = 00000000  REG[25] = 00000000  REG[26] = 00000000  REG[27] = 00000000  REG[28] = 00000000  REG[29] = 00000000  REG[30] = 00000000  REG[31] = 00000000
```

Figure 21: *Values in Registers and Memory (Final Cycle)*

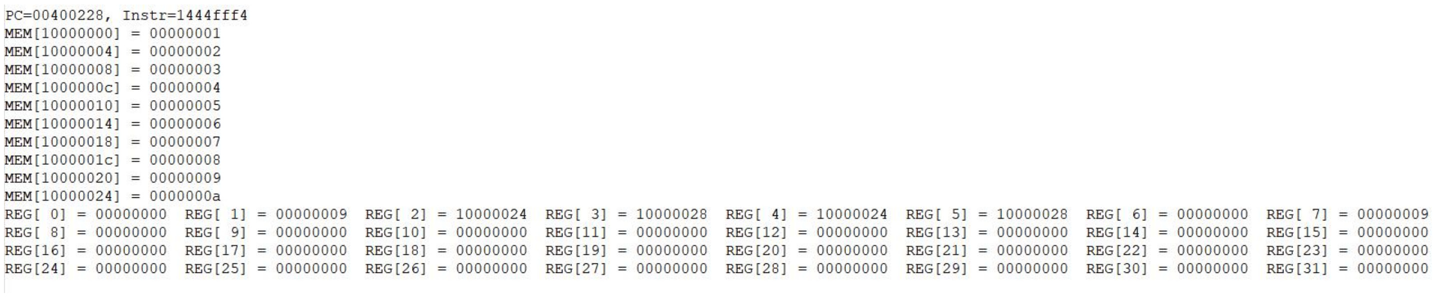
10.2 bills_branch.dat Verification



```
PC=004000c0, Instr=ace60000
MEM[10000000] = 00000000
MEM[10000004] = 00000000
MEM[10000008] = 00000000
MEM[1000000c] = 000002bc
MEM[10000010] = 00000000
MEM[10000014] = 00000000
MEM[10000018] = 00000190
MEM[1000001c] = 00000000
MEM[10000020] = 00000000
MEM[10000024] = 00000000
MEM[10000028] = 00000038
REG[ 0] = 00000000  REG[ 1] = 00000000  REG[ 2] = 10000028  REG[ 3] = 00000003  REG[ 4] = 00000000  REG[ 5] = 00000001  REG[ 6] = 00000038  REG[ 7] = 10000028
REG[ 8] = 00000000  REG[ 9] = 00000000  REG[10] = 00000000  REG[11] = 00000000  REG[12] = 00000000  REG[13] = 00000000  REG[14] = 00000000  REG[15] = 00000000
REG[16] = 00000000  REG[17] = 00000000  REG[18] = 00000000  REG[19] = 00000000  REG[20] = 00000000  REG[21] = 00000000  REG[22] = 00000000  REG[23] = 00000000
REG[24] = 00000000  REG[25] = 00000000  REG[26] = 00000000  REG[27] = 00000000  REG[28] = 00000000  REG[29] = 00000000  REG[30] = 00000000  REG[31] = 00000000
```

Figure 22: *Values in Registers and Memory (Final Cycle)*

10.3 sort_corrected_branch.dat Verification



```
PC=00400228, Instr=1444fff4
MEM[10000000] = 00000001
MEM[10000004] = 00000002
MEM[10000008] = 00000003
MEM[1000000c] = 00000004
MEM[10000010] = 00000005
MEM[10000014] = 00000006
MEM[10000018] = 00000007
MEM[1000001c] = 00000008
MEM[10000020] = 00000009
MEM[10000024] = 0000000a
REG[ 0] = 00000000  REG[ 1] = 00000009  REG[ 2] = 10000024  REG[ 3] = 10000028  REG[ 4] = 10000024  REG[ 5] = 10000028  REG[ 6] = 00000000  REG[ 7] = 00000009
REG[ 8] = 00000000  REG[ 9] = 00000000  REG[10] = 00000000  REG[11] = 00000000  REG[12] = 00000000  REG[13] = 00000000  REG[14] = 00000000  REG[15] = 00000000
REG[16] = 00000000  REG[17] = 00000000  REG[18] = 00000000  REG[19] = 00000000  REG[20] = 00000000  REG[21] = 00000000  REG[22] = 00000000  REG[23] = 00000000
REG[24] = 00000000  REG[25] = 00000000  REG[26] = 00000000  REG[27] = 00000000  REG[28] = 00000000  REG[29] = 00000000  REG[30] = 00000000  REG[31] = 00000000
```

Figure 23: *Values in Registers and Memory (Final Cycle)*