PIPELINED CPU PROJECT REPORT

Justin Enciso-Anaya , Ruby Huynh , Anusha Kankipati

CSEN-122, Winter 2024

Dr. Weijia Shang

03/16/2024

Santa Clara University

School of Engineering

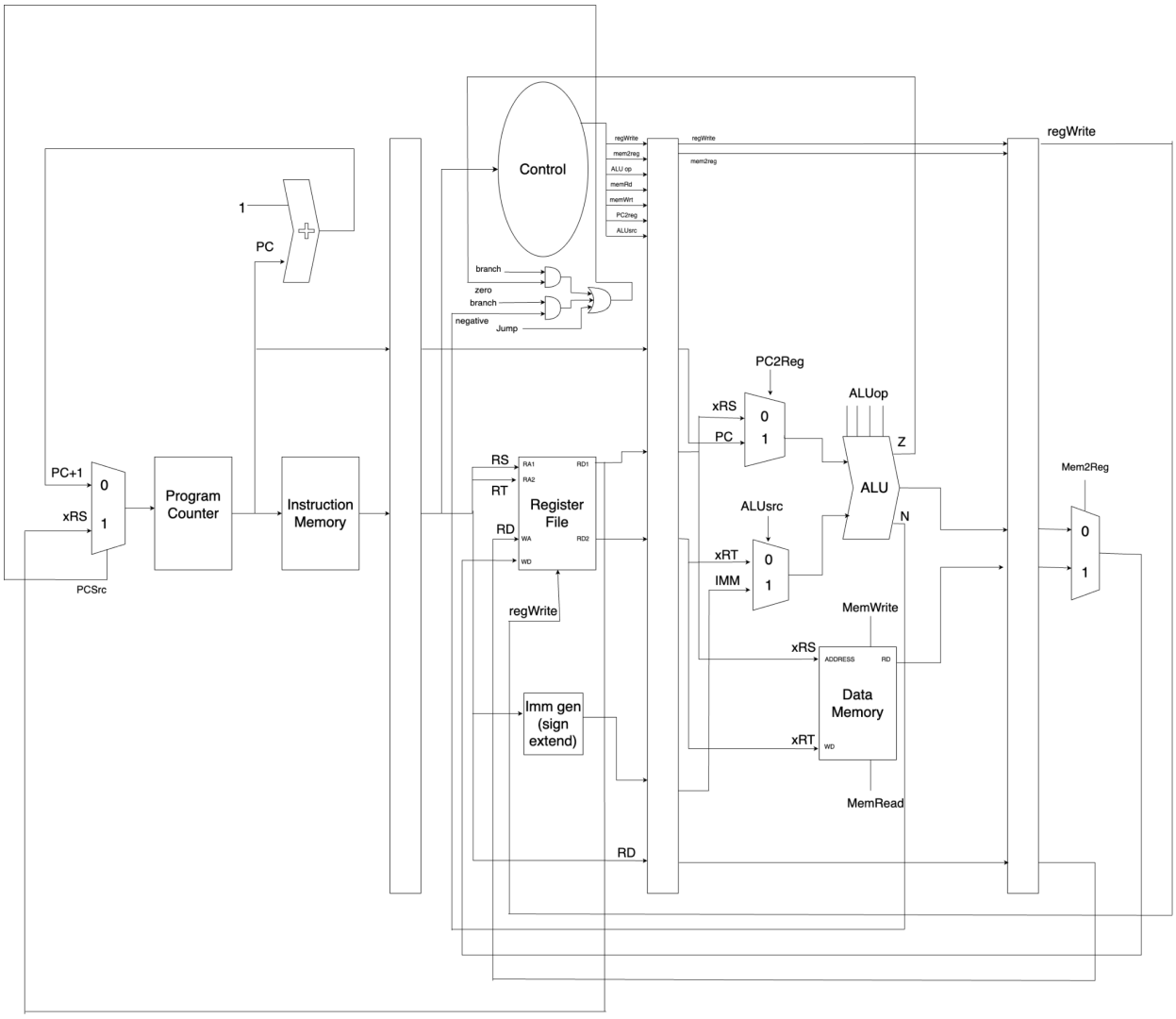
Abstract

This project involves designing a 32-bit pipelined CPU using Verilog HDL, targeted at executing a specific set of 13 instructions defined within the SCU ISA. The design emphasizes implementing a structural model that accommodates the unique instruction set. The project requires us as a team to develop two assembly program versions to benchmark the CPU's capabilities. The evaluation criteria include the CPU design's effectiveness in handling the instruction set, and the efficiency of the pipeline stages. In the end, we were able to piece together all necessary components to build a working CPU.

KEYWORDS: Verilog, Assembly, Pipeline, 32-bit, SCU ISA

I. Datapath Control + Truth Table

In our project, we embarked on the challenging endeavor of designing a 32-bit pipelined CPU using Verilog HDL, tailored to efficiently execute a specified set of instructions within the SCU ISA framework. As a team, we focused on a structural approach to model our CPU, ensuring it could adeptly manage a unique set of 13 instructions. This project challenged us to apply our theoretical knowledge in a practical setting and provided us with invaluable insights into the intricacies of CPU design and optimization, putting what we learned over the winter quarter into practice–as well as what we have learned in the prerequisite courses.



Instruction	op-code	regWrt	mem2reg	PC2reg	branch	jump	memRd	memWrt	ALUsrc	ALUop
NOP	0000	0	0	x	0	0	0	0	x	0100
SavePC	1111	1	0	1	0	0	0	0	1	0100
Load	1110	1	1	x	0	0	1	0	x	0100
Store	0011	0	x	x	0	0	0	1	x	0100
Add	0100	1	0	0	0	0	0	0	0	0000
Increment	0101	1	0	0	0	0	0	0	1	0001
Negate	0110	1	0	0	0	0	0	0	x	0010
Subtract	0111	1	0	0	0	0	0	0	0	0011
Jump	1000	0	x	x	0	1	0	0	x	0100
Branch if Z	1001	0	x	x	1	0	0	0	x	0100
Branch if N	1011	0	x	x	1	0	0	0	x	0100

This datapath provides a comprehensive overview of our controlled datapath within a CPU's architecture, integral to the processing of instructions. At the core of our datapath is the Control unit, which orchestrates the flow of operations by emitting various control signals. The Program Counter and Instruction Memory are indicative of the fetch stage, where instructions are sequentially retrieved for execution.

IF Stage:

In the IF stage, the CPU begins by fetching and decoding instructions. The program counter keeps track of the current instruction to be executed, while the instruction memory holds the set of instructions. Each instruction includes truth table information and destination address data. Upon fetching an instruction, its details are passed to the next stage. The program counter is adjusted to proceed to the next instruction or to execute a jump.

ID Stage:

The ID stage houses the Register File, serving as the active data working set. Registers are read from and written to as necessary. The control module decodes the Opcode from the previous

stage to determine the appropriate control signals. Instructions are decoded here, and addresses are forwarded to the register for potential read or write operations.

EX/MEM Stage:

The EX/MEM stage carries out significant CPU operations. The ALU performs arithmetic operations as needed. Additionally, this stage determines the address for the next instruction in case of branch or jump instructions. For BRZ or BRN instructions, the ALU's Z and N values are stored for use in branching in the next cycle. Data memory operations, including writes and reads, are also handled here. While the data and register memory size is constrained by simulator limitations, in a real-world scenario, they would be much larger.

WB Stage:

The WB stage is crucial for finalizing operations. Any operations requiring writing back to the register file are managed here. Additionally, the final elements for determining the next instruction address are held in this stage, which is then written to the program counter.

II. Simulation Verification

```
module test_bench();
        wire [31:0] pc_increment;
        wire [31:0] rs_val;
    wire pc_src;
        wire ex_jumpMem_out;
        wire [31:0] pc_mux_out;
        wire aluZ, aluN;

        wire [31:0] PCOut;
        reg clock;

        //INSTRUCTION FETCH STAGE_____________________
          mux test(pc_increment,rsOut , pc_src, pc_mux_out); //very 1st mux before PC
          program_counter test1(pc_mux_out, PCOut, clock); //unsure of which module
module alu

        wire N, Z;

        //PC adder output: pc_increment
        alu test1_5(PCOut, 1, 4'b0100, pc_increment, N, Z); //PC = PC+1 CHANGE TO
ADDER??

        initial
        begin
                clock = 0;
                forever #5 clock = ~clock;
        end

        wire [31:0] im_out; //instruction memory out
```

```
//instruction memory output: im_out
inst_mem test2(PCOut, clock, im_out);

wire [31:0] if_im_out; //inst fetch inst mem out
wire [31:0] if_PCOut;

//IF/ID buffer output: if_im_out, if_PCOut
ifid_buffer test3(clock, im_out, PCOut, if_im_out, if_PCOut);

//INSTRUCTION DECODE STAGE_____________________
wire [31:0] signOut;

//Sign extension output: signOut
sign_extender test4(if_im_out[31:28], if_im_out[21:0], signOut);

wire [31:0] rsOut;
wire [31:0] rtOut;
wire [31:0] data_write;
wire [31:0] ex_regWrite_out;

//PROBLEM: Reset not initialized. Ask what it is for reg_file.v
reg_file test5(clock, if_im_out[21:16], if_im_out[15:10], ex_rd_out,
ex_regWrite_out, ex_regWrite, rsOut, rtOut);

wire [3:0] aluOp;
wire regWrite, memtoReg, PCtoReg, jump, memRead, memWrite, jumpMem,
branchz, branchn;

//Control output: regWrite, aluSrc, memtoReg, memRead, memWrite, branch,
PCtoReg, jump, aluOp
```

```
        control test6(if_im_out[31:28], clock, regWrite, aluSrc, memtoReg, memRead,
memWrite, branchz, PCtoReg, jump, branchn, aluOp);

        //GETING PC_SRC AS OUTPUT
        wire and1_out;
        and_gate test12(branchz, aluZ, and1_out);

        wire and2_out;
        and_gate test13(branchn, aluN, and2_out);

        wire or1_out;
        or_gate test14(and1_out, and2_out, or1_out, clock);

        wire or2_out;
        or_gate test15(or1_out, jump, pc_src, clock);

        //Used for outputs in idex_buffer
        wire [31:0] rsOut_out;
        wire [31:0] rtOut_out;
        wire [5:0] rd_out;
        //wire [31:0] alu2_out_out;
        wire [3:0] aluOp_out;
        wire [31:0] PCOut_Out, signOut_Out;
        wire memRead_out, memWrite_out, PCtoReg_out, memtoReg_out, regWrite_out,
jumpMem_out, aluSrcOut;

    //ID/EX Buffer outputs: rd_out, rsOut_out, rtOut_out, PCOut_Out, regWrite_out,
memtoReg_out, aluOp_out, memRead_out, memWrite_out, PCtoReg_out, aluSrcOut,
signOut_Out
        idex_buffer test7(clock, if_im_out[27:22], rsOut, rtOut, if_PCOut, regWrite,
memtoReg, aluOp, memRead, memWrite, PCtoReg, aluSrc, signOut,
```

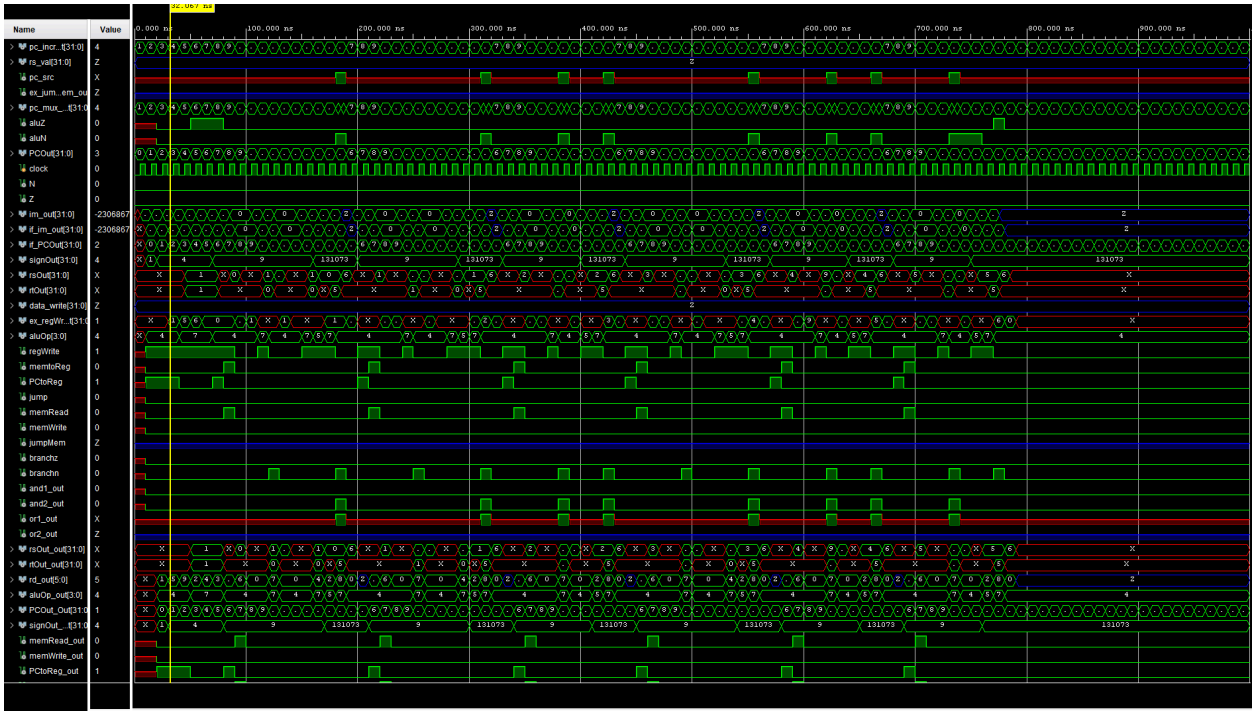
```
                                                    rd_out, rsOut_out,
rtOut_out, PCOut_Out, regWrite_out, memtoReg_out, aluOp_out, memRead_out,
memWrite_out, PCtoReg_out, aluSrcOut, signOut_Out);

        //EXECUTION STAGE_____________________
        wire [31:0] dataMemoryOut;
        wire [31:0] aluInput1, aluInput2;

        //First alu Input Mux output: aluInput1
        mux test17(rsOut_out, PCOut_Out, PCtoReg_out, aluInput1);

        //Second alu Input Mux output: aluInput2
        mux test18(rtOut_out, signOut_Out, aluSrcOut, aluInput2);

        wire [31:0] aluOut;

        //ALU output: aluZ, aluN, aluOut
        alu test9(aluInput1, aluInput2, aluOp_out, aluOut, aluN, aluZ);

        //data memory output: dataMemoryOut
        data_mem test8(clock, memWrite_out, memRead_out, rsOut_out, rtOut_out,
dataMemoryOut);

        wire [31:0] ex_alu_out, ex_dataMem_out;
        wire [5:0] ex_rd_out;
        wire ex_memtoReg_out, ex_regWrite;

        //WRITE BACK STAGE_____________________
        exwb_buffer test10(clock, dataMemoryOut, aluOut, regWrite_out,
memtoReg_out, rd_out,
```

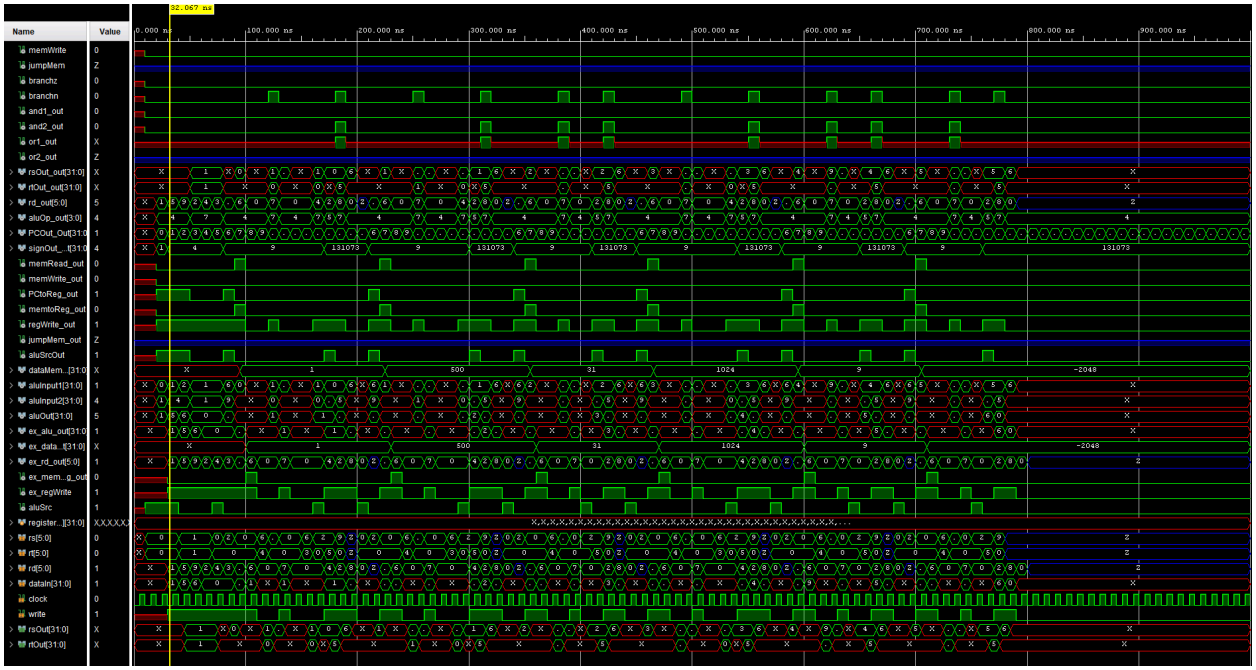
ex_dataMem_out, ex_alu_out, ex_regWrite, ex_memtoReg_out, ex_rd_out);

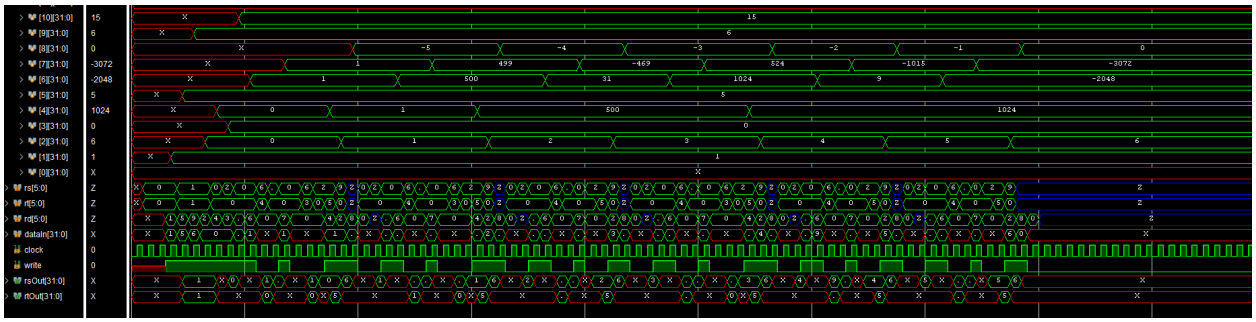
//WB Mux output: ex_regWrite_out

mux test11(ex_alu_out, ex_dataMem_out, ex_memtoReg_out, ex_regWrite_out);

endmodule




Name
Value
0.000 ns
100.000 ns
200.000 ns
300.000 ns
400.000 ns
500.000 ns
600.000 ns
700.000 ns
800.000 ns
900.000 ns
memWrite
jumpMem
branchz
branchn
and1_out
and2_out
or1_out
or2_out
rsOut_out[31:0]
rtOut_out[31:0]
rd_out[5:0]
aluOp_out[3:0]
PCOut_Out[31:0]
signOut_...t[31:0]
memRead_out
memWrite_out
PCtoReg_out
memtoReg_out
regWrite_out
jumpMem_out
aluSrcOut
dataMem...[31:0]
aluInput1[31:0]
aluInput2[31:0]
aluOut[31:0]
ex_alu_out[31:0]
ex_data...t[31:0]
ex_rd_out[5:0]
ex_mem...g_out
ex_regWrite
aluSrc
register...[31:0]
rs[5:0]
rt[5:0]
rd[5:0]
dataIn[31:0]
clock
write
rsOut[31:0]
rtOut[31:0]


[10][31:0]
[9][31:0]
[8][31:0]
[7][31:0]
[6][31:0]
[5][31:0]
[4][31:0]
[3][31:0]
[2][31:0]
[1][31:0]
[0][31:0]
rs[5:0]
rt[5:0]
rd[5:0]
dataIn[31:0]
clock
write
rsOut[31:0]
rtOut[31:0]

III. Assembly Code for Max Calculation

This first version is using our provided Instruction Set:

```
sub  x4, x4, x4  //Set x4 register to 0
add  x5, x2, x3  //Store the address of last element in array in x5

SVPC x9, 2       //Save the target branch address of PC+2 into x9
SVPC x10, 5      //Save the target branch address of PC+5 into x10

LD   x6, x2      //x6=mem[x2]
sub  x7, x6, x4  //x7=x6-x4 x7 holds negative val if x6<x4
BRN  x10         //If x6 is smaller than x4, skip instruction loading x6 to x4

INC  x4, x6, 0   //x4 = x6 + 0, else store the value in x6 into x4 if x6 !< x4

INC  x2, x2, 1   //x2=x2+1 (Move to next array element)
sub  x8, x2, x5  //x8=x2-x5 If at last element in array, x8 will hold 0
BRN  x9          //else if not yet at last element in array, loop back
BRZ  x9          //if at last element in array, loop back to process last element

STOP
```

This version is assuming we have a MAX instruction:

```
MAX x4, x2, x3
STOP
```

In our program, we ended up using this assembly program to find the maximum number

```
26
27      instr[0] = 32'b1111_000001_000000_000000_0000000001; //SVPC x1, 1
28      instr[1] = 32'b1111_000101_000000_000000_0000000100; //SVPC x5, 4  Store the address of last element in array in x5
29      instr[2] = 32'b1111_001001_000000_000000_0000000100; //SVPC x9, 4 Save the target branch address of PC+4 into x9
30      instr[3] = 32'b0111_000010_000001_000001_0000000000; //sub x2, x1, x1 Set x2 register to 0
31      instr[4] = 32'b0111_000100_000001_000001_0000000000; //sub x4, x1, x1 Set x4 register to 0
32      instr[5] = 32'b0111_000011_000001_000001_0000000000; //sub x3, x1, x1 Set x3 register to 0
33
34      instr[6] = 32'b1111_001010_000000_000000_0000001001; //SVPC x10, 9  Save the target branch address of PC+5 into x10
35
36      instr[7] = 32'b1110_000110_000010_000000_0000000000; //LD   x6, x2      //x6=mem[x2]
37      instr[8] = 32'b00000000000000000000000000000000;     //NOP
38      instr[9] = 32'b00000000000000000000000000000000;     //NOP
39
40
41      instr[10] = 32'b0111_000111_000110_000100_0000000000; //sub  x7, x6, x4  x7=x6-x4 x7 holds negative val if x6<x4
42      instr[11] = 32'b1011_000000_001010_000000_0000000000; //BRN  x10         //If x6 is smaller than x4, skip instruction loading x6 to x4
43      instr[12] = 32'b00000000000000000000000000000000;    //NOP
44      instr[13] = 32'b00000000000000000000000000000000;    //NOP
45      instr[14] = 32'b0111_000100_000110_000011_0000000000; //sub x4, x6, x3 Set x4 register to 0
46
47      instr[15] = 32'b0101_000010_000010_000000_0000000001; //INC  x2, x2, 1   //x2=x2+1 (Move to next array element)
48
49      instr[16] = 32'b0111_001000_000010_000101_0000000000; //sub  x8, x2, x5  //x8=x2-x5 If at last element in array, x8 will hold 0
50      instr[17] = 32'b1011_000000_001001_000000_0000000000; //BRN  x9          //else if not yet at last element in array, loop back
```

IV. Execution Time Analysis

Our system has 4 NOP instructions which are each 4 cycles. We have 14 regular instructions, which come out to 56 + 16 CPU clock cycles. Theoretically, the amount of time to get through each step in the data path should be 10ns, since we set it in the simulation. 56 + 16 = 72 CPU clock cycles. 72 clock cycles * 10ns/clock cycle = 720ns. This is similar to the 750 ns execution time our waveform produced in which the data memory 6 values were fully traversed and compared with register 4's value.