Northwestern University

McCormick School of Engineering

CE361 Computer Architecture Project - Group 11

Single Cycle Processor Design

Xi Chen

Yankai Jiang

Matthew Schilling

2021 November

1. Introduction of Project

In this project, we were asked to design, implement, and test a single-cycle 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

In computer architecture, a major component of the computer is the processor. This unit essentially is in charge of making computations and carrying out instructions for the computer. For this project, we were tasked with designing a central processing unit that completes instructions within a single cycle. To implement the given instructions, we had to design different components, the major ones being the control unit, the data memory, the instruction memory, a register file, a program counter, and an ALU. Along with these, different extenders and adders were used along the way to carry out certain instructions.

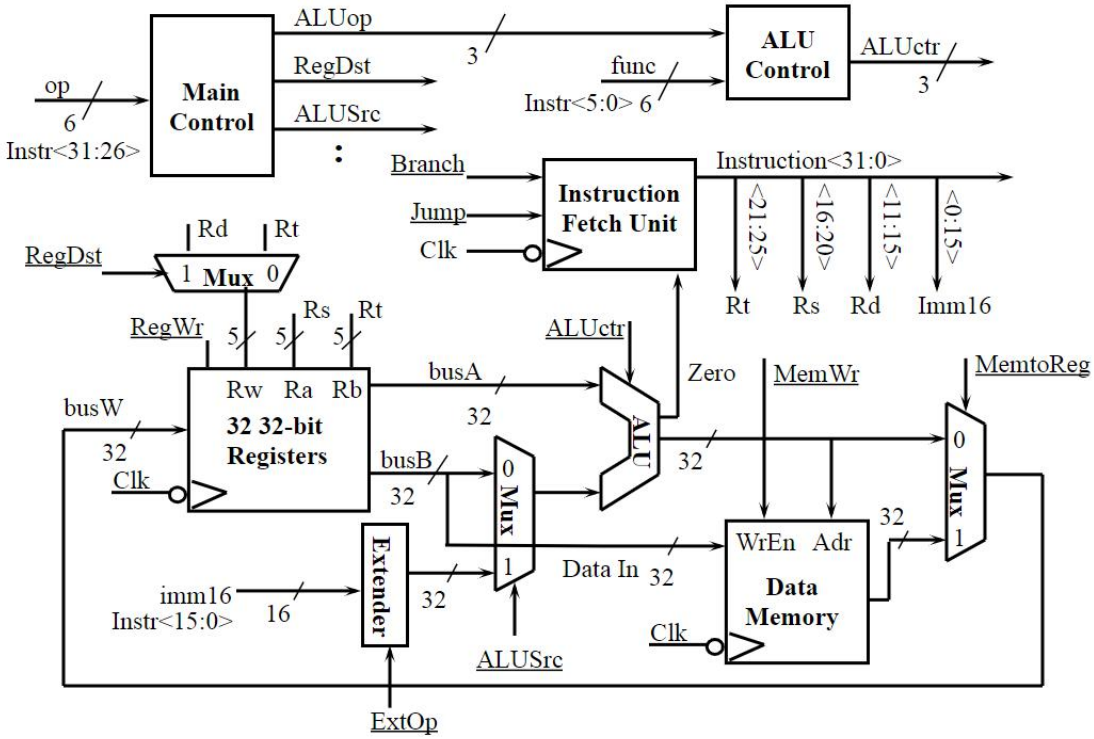

Figure 1: *Single Cycler Processor Design (Slide 8, Lecture 52)*

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

Component	File Name
32-bit Adder	adder_32bit.v
ALU control	alu_control.v
ALU	alu.vhd
Main Control	scp_controller.v
32 32-bit Registers	register_file.v
Extender	extender.v
Data Memory	sram.v
Instruction Memory	sram.v
Instruction Fetch Unit	Instr_fetch_unit.v
SCP Top File	scp_top.v

2. Control Logic Design

2.1 SCP Main Controller

Control Unit Operation (Based on OpCode):

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	X	1(==0)	001
BNE	000101	0	0	0	0	0	X	1(!=0)	001
BGTZ	000111	0	0	0	0	0	X	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

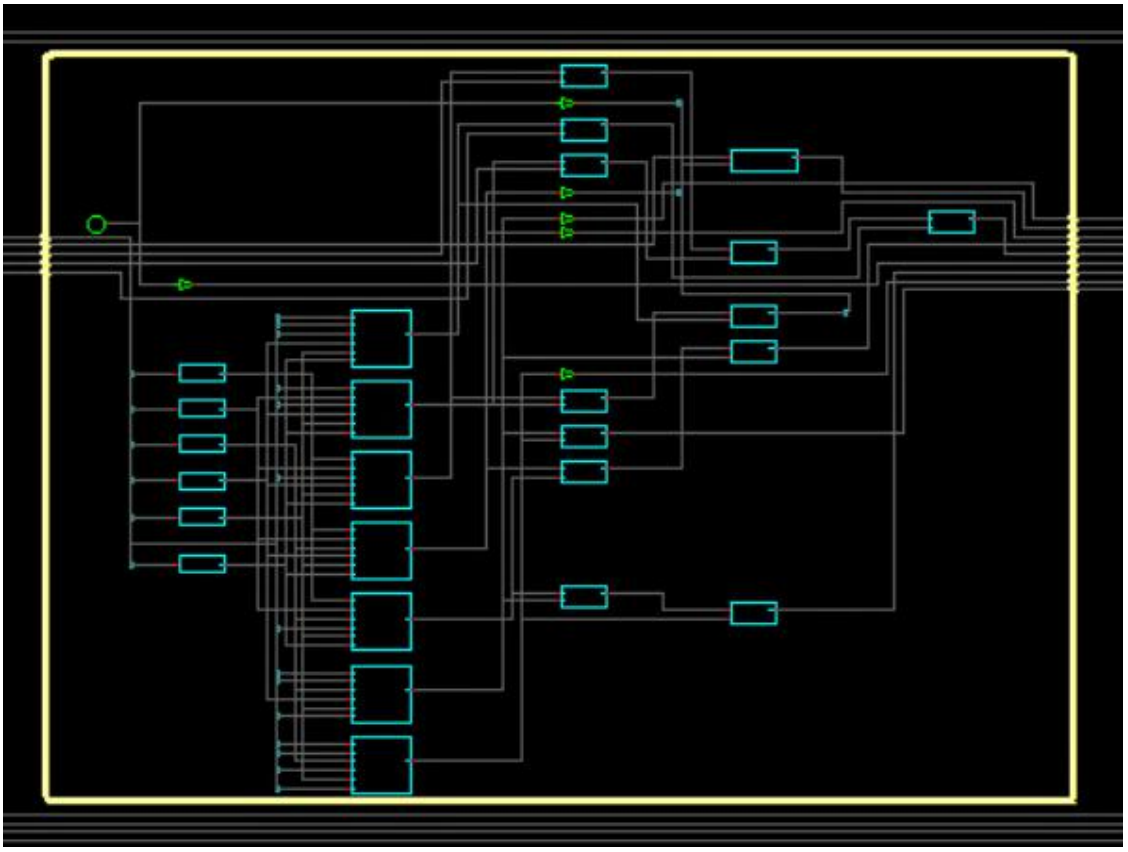

Figure 2: *Schematic Tracer of SCP Controller*

Control unit is one of our biggest components that requires designing and testing. A control unit should take the *opcode* of the instructions as input and generate control signals for each *multiplexer,* D-mem, Register file and ALU. 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, use lots of gates, and realize the SCP controller.

2.2 ALU 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.

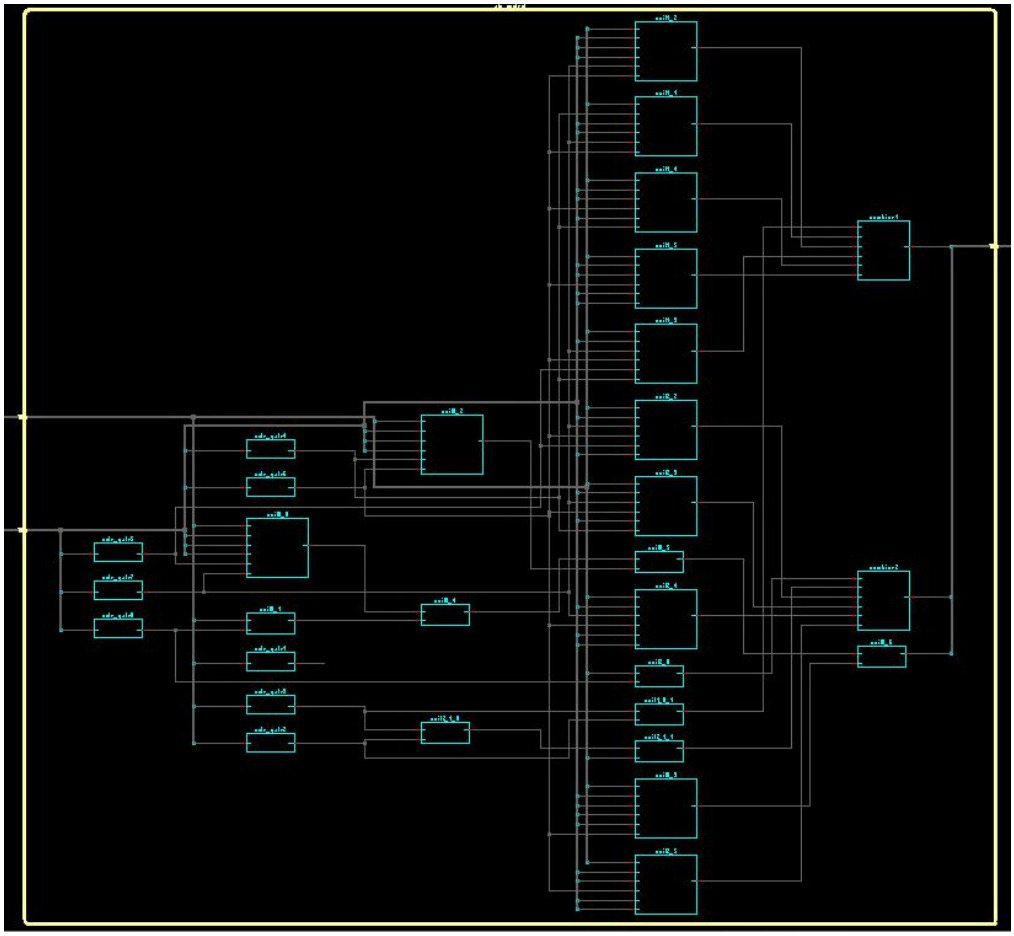

Figure 3: *Schematic Tracer of ALU Controller*

The design is similar to SCP controller, just use lots of logic gates to implement.

2.3 Declaration

The ALU controller is included in the SCP controller. The inputs and outputs of SCP controller are:

Inputs	Outputs
opCode	ALUctrl,
eqzero	RegDst,
not_eqzero	RegWr,
greater	Branch,
func	Jump,
	ALUSrc,
	MemWr,
	MemtoReg,
	ExtOp

There are 3 signals: eqzero, not_eqzero and greater. These signal are generated through the ALU. And it will jointly generate a branch signal and send it to the instruction fetch unit.
And the branch will be:

$$\text{Branch} = (\text{eqzero and beq}) \text{ or } (\text{not\_eqzero and bne}) \text{ or } (\text{greater and bgtz})$$

3. Instruction Fetch Unit Design

Instruction Fetch Unit is to generate RS, RT, RD and immediate value. Follow the image from the slides, the difference is the blue circle, the branch signal is already generated from the SCP controller.

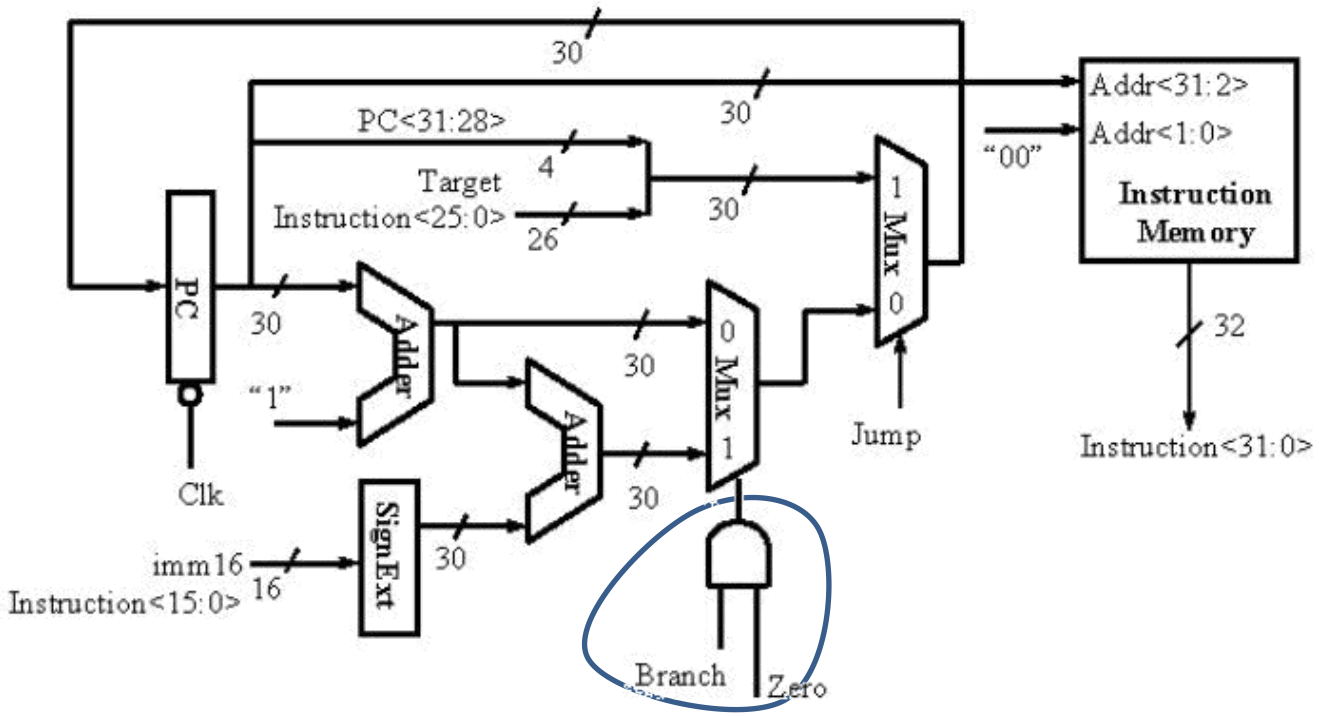

Figure 4: *Instruction Fetch Unit Design*

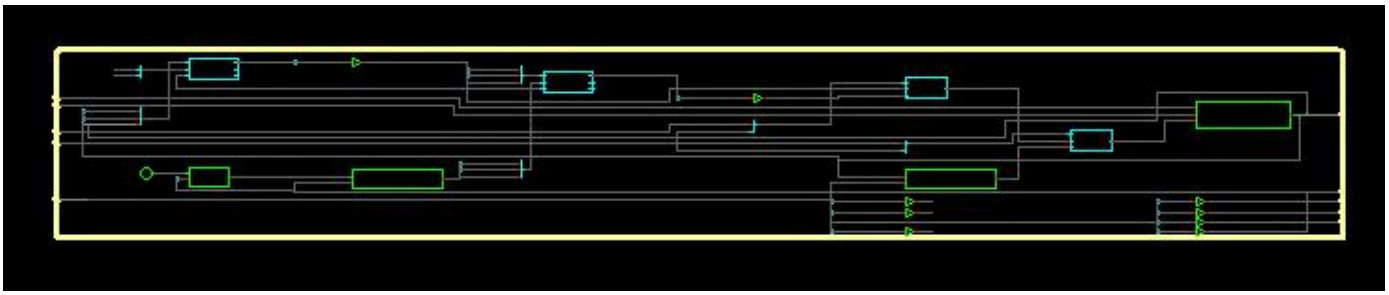

Figure 4: *Schematic Tracer of Instruction Fetch Unit*

4. Register File Design

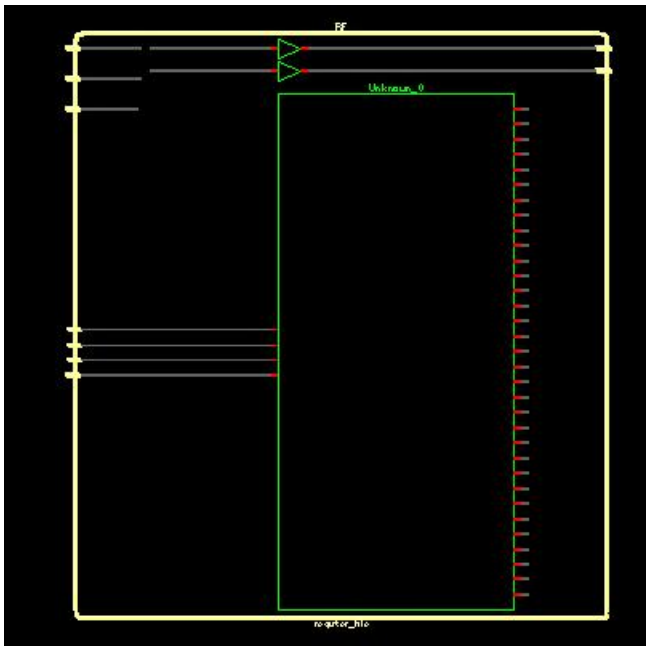

Figure 5: *Schematic Tracer of 32 Register(32-bit)*

This unit has 32 registers, and initiated 0, every time, it can generate value with providing addresses of registers. Also, the clock signal and rstb to control this register file.

5. SCP Overall Design

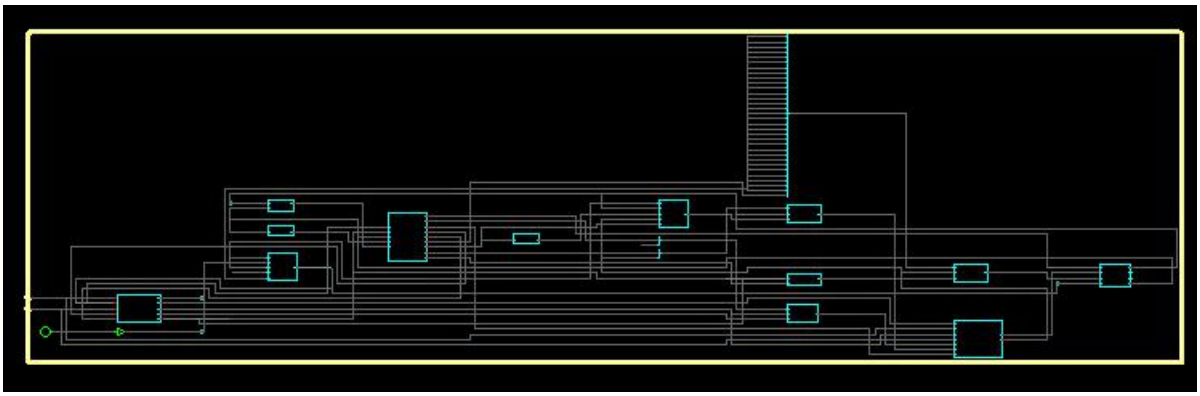

Figure 6: *Schematic Tracer of Single Cycle Processor*

The inputs of SCP are just clk and rstb. The cycle time is 10 ns.

6. Test

We just use utilized unsigned_sum.dat as an example to test individual components in our design. The MIPS code in that file is:

ADD $5 $0 $0
ADDI $7 $0 4096
SLL $7 $7 16
ADD $6 $7 $0
ADDI $6 $6 0x0028
LW $4 0($7)
ADDU $5 $5 $4
ADDI $7 $7 4
BNE $7 $6 0xFFFC
SW $5 0($7)

6.1 ALU Simulation Test

We use unsigned_sum to test the ALU, to make sure the ALU is correct. The wave form is:

Figure 7: *Wave Form of ALU (unsigned_sum)*

We just upload part of the wave form for testing the correctness of ALU, the first result of ALU is 0x00000000, it's correct, because the first instruction is *ADD $5 $0 $0*. As we check all the value of ALU, we find that the implementation of ALU is correct.

6.2 Instruction Fetch Simulation

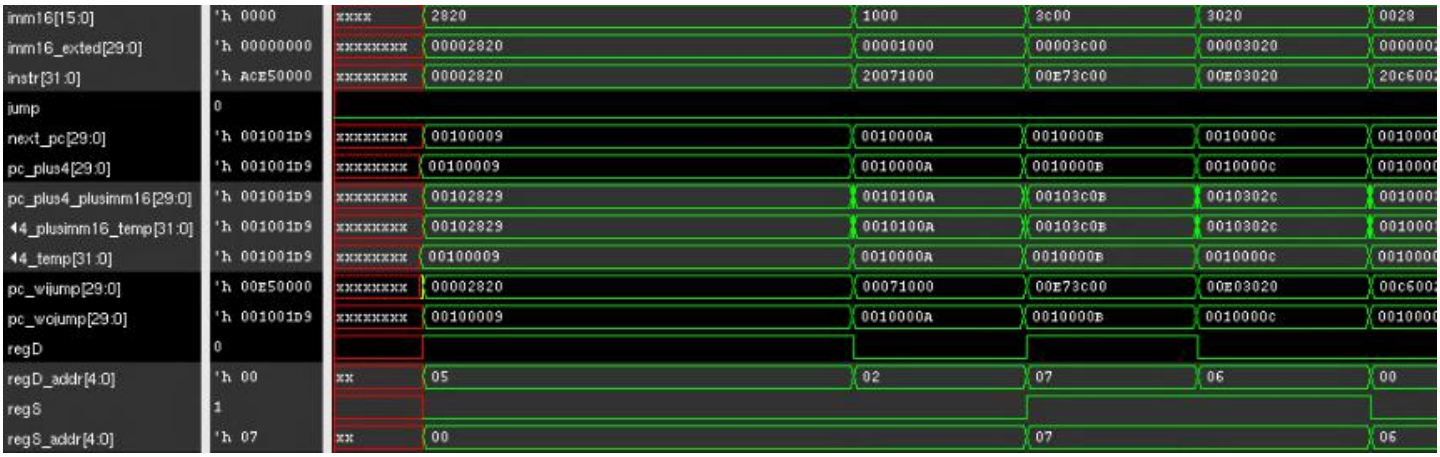

Figure 8: *Wave Form of Instruction Fetch Unit (unsigned_sum)*

We notice that for the for the wave form of *instr[31:0]*, the instructions are executed correctly. The correct instruction order is: 0x00002820, 0x20071000, 0x00e73c00, 0x00e03020... Also, the *regD_addr* and *regS_addr* are also correct.

6.3 Register File Simulation

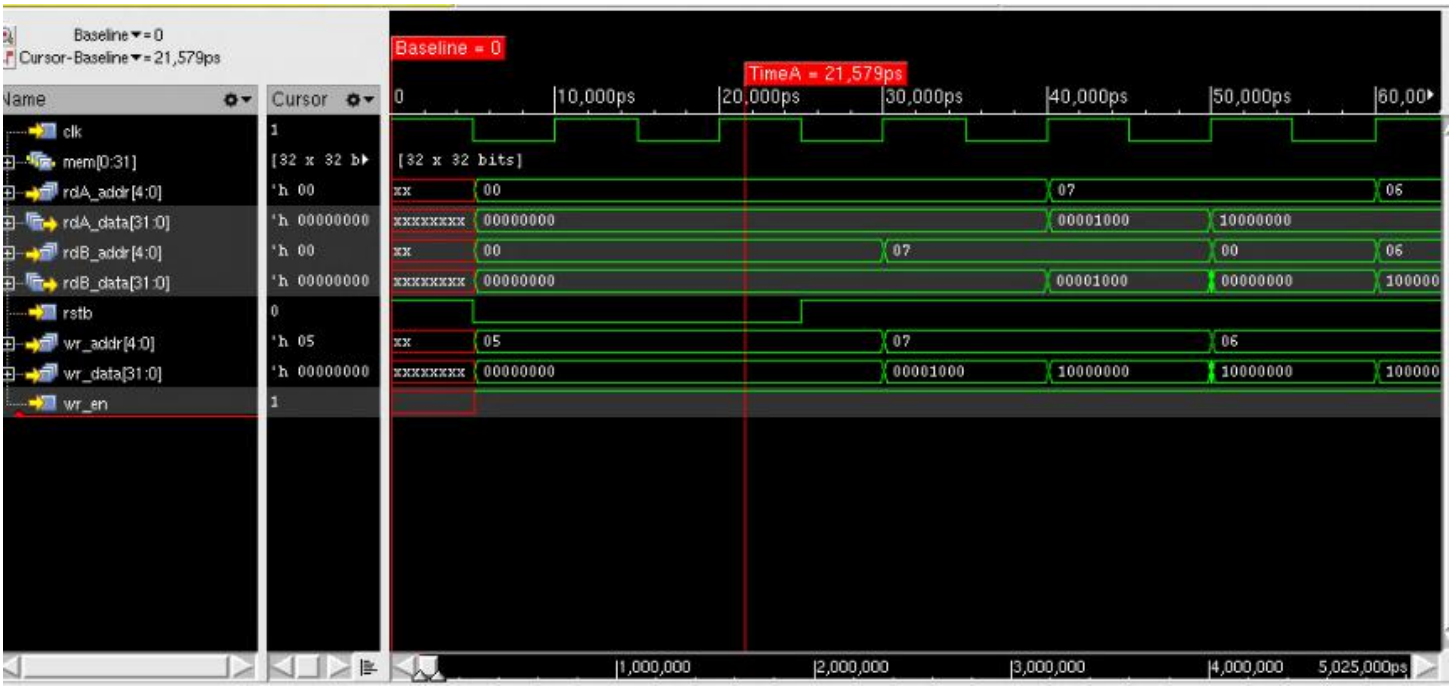

Figure 9: *Wave Form of 32 Register File (unsigned_sum)*

We notice that register file works well, and the result are related to Instruction Fetch Unit correctly, we also generate files with the results of registers and memory for each cycle, we can easily track the result to verify it.

6.4 Data Memory Simulation

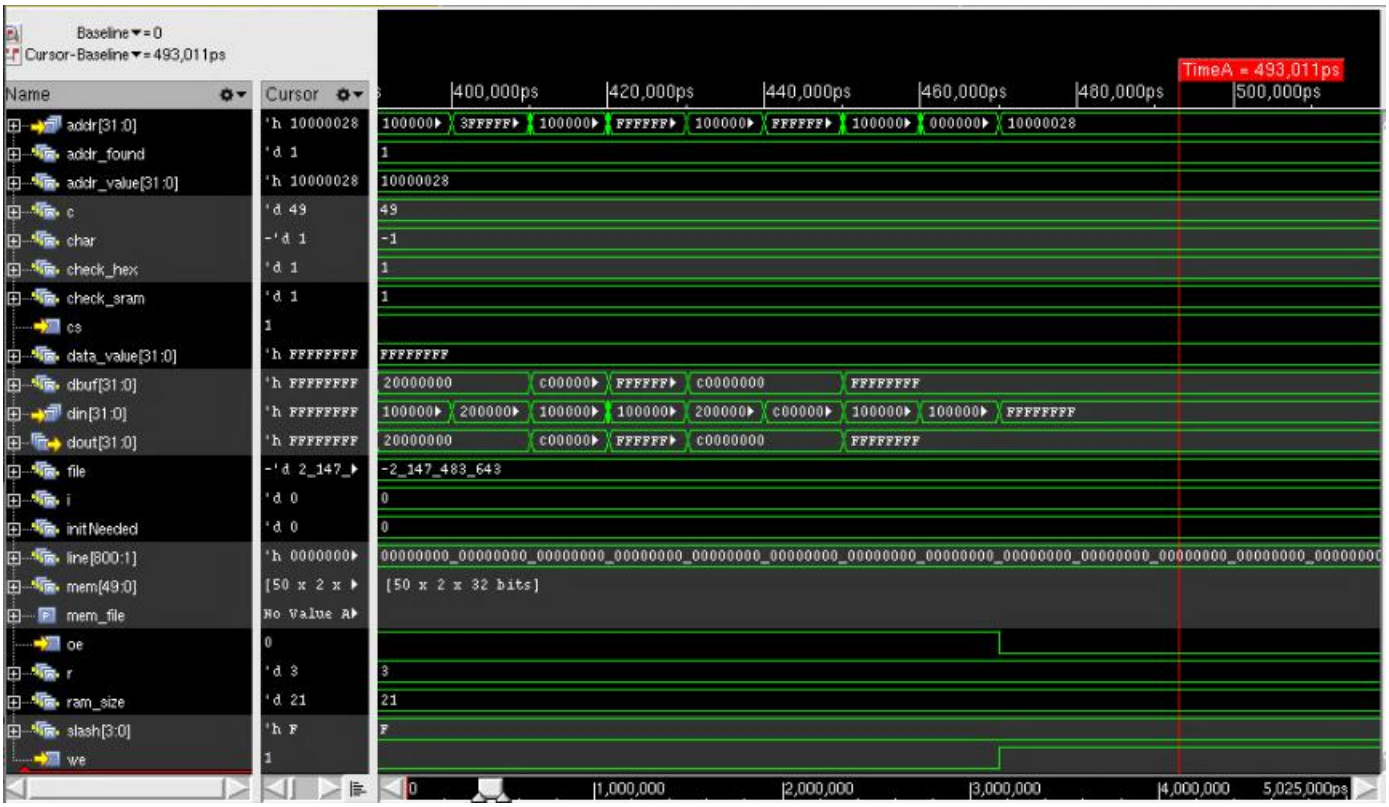

Figure 10: *Wave Form of Data Memory (unsigned_sum)*

Figure 10 shows the last cycle, and it just store the sum result ($5)=0xffffffff in the 10000028 address memory. In the part 7, we will show the values of registers and memory in each cycle.

7. 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 5000, 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 11: *Part of Testbench for Generating Results*

7.1 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 12: *Values in Registers and Memory (Final Cycle)*

7.2 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 13: *Values in Registers and Memory (Final Cycle)*

7.3 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 14: *Values in Registers and Memory (Final Cycle)*

8. Challenges

(1) . The libs and ALU have some issues, such as the mux_32.v in lib folder is different with it in the ALU folder.

(2) . Verification is very tricky, because we need to calculate values in registers and memory for every cycle. sort_corrected_branch.dat is the most difficult one, because it contains many 3 branches, calculate these values manually is really a disaster.