Term Project 2 Report

20210225 Rojin Park

1. Introduction

I implemented Cortex M0 CPU with 5 stage pipeline including forwarding logic, a simple branch predictor. I passed all tests in the give ALU_TEST testbench, and also passed all tests in my personal testbench.

2. Design

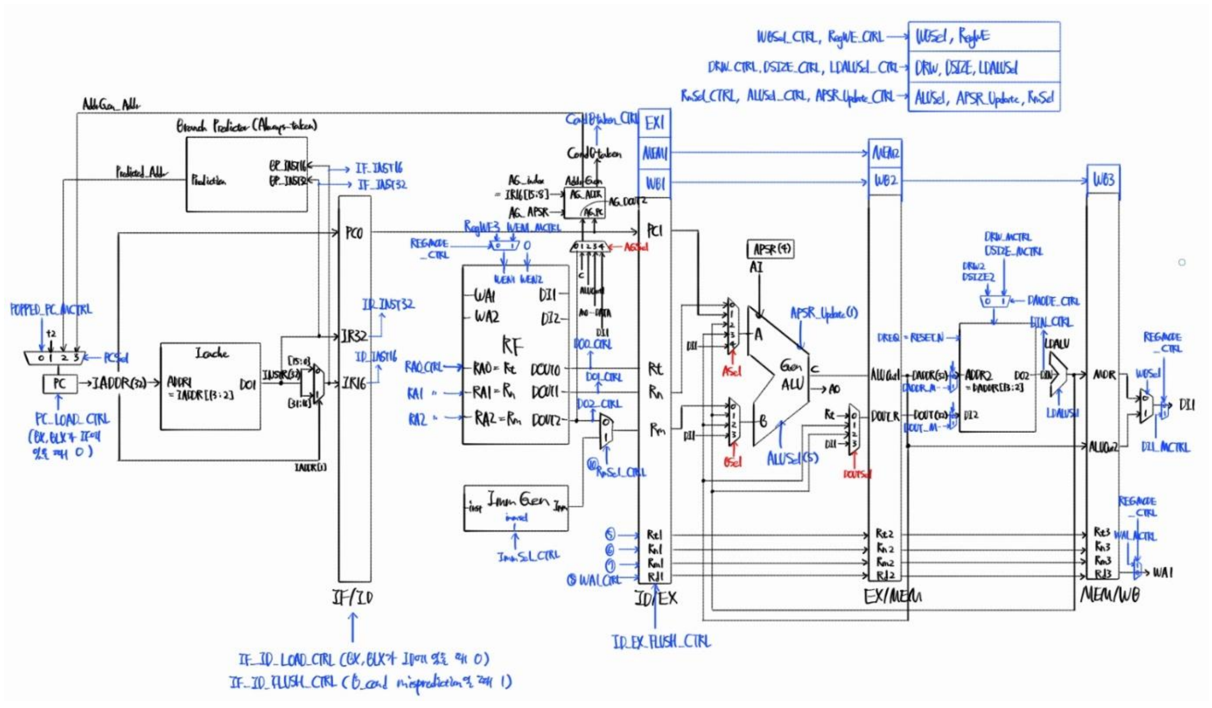


Fig1. CPU Diagram

The upper diagram is the entire design of my CPU. It is composed of several modules – ALU, LDALU, STALU, Branch_Predictor, ImmGen, AddrGen, CTRL, and Forwarding. Data flows are colored by black, while control flows are colored by blue. Moreover, red signals are generated by Forwarding logic. Please note that STALU is omitted in the diagram since the space is limited.

Blue rectangles upon each black rectangle between each stage indicate the pipeline registers, control signal transmitting registers to be specific.

The pipeline is updated at negative edge of clock. Most data hazards are eliminated by forwarding logic, but control flow hazards represented by BX, BLX, branch mispredictions still exist. However, all hazards are resolved in one cycle. For multiple data transfer instructions – PUSH, POP, STM, LDM – the CPU wait for 3 cycles to empty EX, MEM, WB pipelines, then sequentially moves data from the register file to the memory, and vice versa. For hazards and multiple data transfer instructions, pipeline stalls are required. This is controlled by specific control signals.

2-1. ALU

This module performs the general arithmetic and logic operations at EX stage. Additionally, it also updates APSR as well. The elucidation of roles of ALU is shown below.

0	LSL imm	11	MOV imm	22	SXTB
1	LSR imm	12	MOV reg	23	UXTH
2	ASR imm	13	AND	24	UXTB
3	LSL reg	14	OR	25	REV
4	LSR reg	15	NOT	26	REV16
5	ASR reg	16	XOR	27	REVSH
6	ADD	17	BIC	28	BLX
7	SUB	18	ROR	29	BL
8	ADC	19	MUL	30	RSB
9	SBC	20	Reglist cnt	31	default
10	ADR/LDR	21	SXTH		

2-2. LDALU, STALU

Since the memory is byte addressable, sophisticated generation of DOUT and modification of DI are strongly required. Therefore, I implemented two additional ALUs to address this. LDALU modifies the DI value and STALU generates DOUT value to be suitable for DSIZE and DADDR[1:0].

2-3. Branch Predictor

There are 5 kinds of control flow instructions – BX, BLX, B(conditional), B(unconditional), BL. BX, BLX are unpredictable in IF stage since they requires the regfile read, so the branch predictor generates the prediction only for BL, B(conditional), B(unconditional). For B(conditional), the prediction algorithm is always taken. The prediction is completely accurate for BL and B(unconditional), but B(conditional) might have an error. This is resolved by AddrGen, which generates accurate branch address.

2-4. ImmGen

This module generates the immediate value by properly extracting portions from the instruction. The generating method is determined by ImmSel control signal, just like ALUSel of ALU. Note that for certain instructions like load and store, some multiplications with 2 or 4 of immediate value are needed. This is properly considered in ImmGen module.

2-5. AddrGen

This module generates the accurate branch address by looking aside the APSR value generated by ALU in EX stage. This module facilitates the branch taken determinations.

2-6. Forwarding

My implementation supports full forwarding of data. Please note that the pipeline is updated at negative clock edge, which is different from memory and register file, thereby eliminating data hazards regarded to load instructions. Consequently, the pipeline does not stall except for control hazards, including BX, BLX, branch misprediction, and little data hazards.

Although the design includes full forwarding, some data hazards still exist since BX and BLX also need forwarding. If the required values of BX and BLX are generated by immediately preceding instructions at MEM stage, 1 cycle stall is required. There are no other data hazards except for this case.

2-7. CTRL

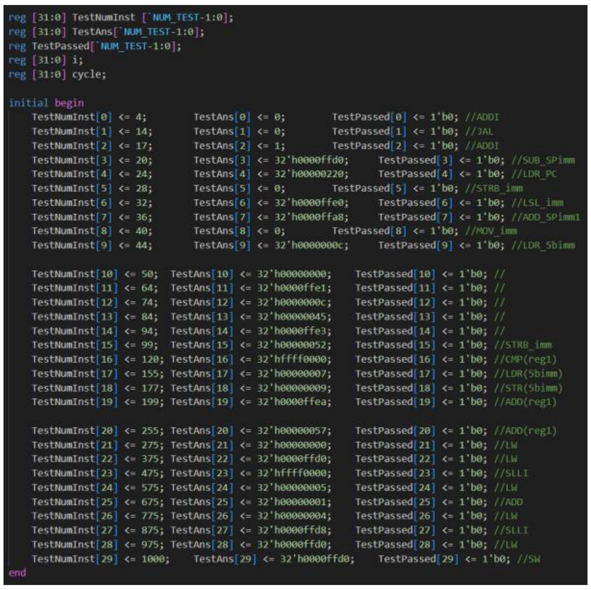
This module is the most important logic in my implementation. It controls every behaviors of the entire CPU. Deviding into several sector, firstly, it determines Rt, Rm, Rn, Rd fields of the instructions to put into the pipeline. Secondly, it judges whether the pipeline stalls are needed, then execute them by manipulating some signals such as PC_LOAD_CTRL, IF_ID_LOAD_CTRL, ID_EX_FLUSH_CTRL, IF_ID_FLUSH_CTRL. Thirdly, it determines numerous control signals to be put into the pipeline from the instruction. Lastly, for multiple data transfer instructions, the module supports certain modes – REGMODE, DMODE – to update the register file and the memory not caring pipelining. For instance, when STM instruction is detected at ID stage, the control module stops IF, ID pipeline and empty other pipelines by waiting 3 cycles while fetching register values. After that, the module directly transfer those data into the data memory by turning on DMODE, consuming multiple cycles. Finally, it turns of DMODE and turns on REGMODE to update proper register in the file, then turns off REGMODE to return to original pipelining.

2-8. IF stage

The instruction memory is pointed by the program counter, which is connected to IADDR. INSTR is divided into two halfwords, and IADDR[1] determines the proper choice between these two halfwords. However, since PC is updated in negative edge clock, which is different from instruction memory, some error occurs. Therefore, an indicator register updated at every positive edge clock gets IADDR value and be used to determine which halfword is the correct instruction at every cycle.

3. Test

As metioned earlier, the implementation passed all tests in ALU_TEST.v as well as my own testbench. I implemented a new testbench to specifically evaluate the design. The codes are shown below.



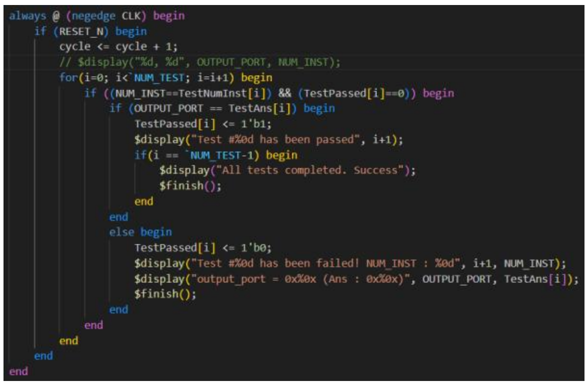
```verilog
reg [31:0] TestNumInst [`NUM_TEST-1:0];
reg [31:0] TestAns[`NUM_TEST-1:0];
reg TestPassed[`NUM_TEST-1:0];
reg [31:0] i;
reg [31:0] cycle;

initial begin
    TestNumInst[0] <= 4;        TestAns[0] <= 0;        TestPassed[0] <= 1'b0; //ADDI
    TestNumInst[1] <= 14;       TestAns[1] <= 0;        TestPassed[1] <= 1'b0; //JAL
    TestNumInst[2] <= 17;       TestAns[2] <= 1;        TestPassed[2] <= 1'b0; //ADDI
    TestNumInst[3] <= 20;       TestAns[3] <= 32'h0000ffd0;     TestPassed[3] <= 1'b0; //SUB_SPimm
    TestNumInst[4] <= 24;       TestAns[4] <= 32'h00000220;     TestPassed[4] <= 1'b0; //LDR_PC
    TestNumInst[5] <= 28;       TestAns[5] <= 0;        TestPassed[5] <= 1'b0; //STRB_imm
    TestNumInst[6] <= 32;       TestAns[6] <= 32'h0000ffe0;     TestPassed[6] <= 1'b0; //LSL_imm
    TestNumInst[7] <= 36;       TestAns[7] <= 32'h0000ffa8;     TestPassed[7] <= 1'b0; //ADD_SPimm1
    TestNumInst[8] <= 40;       TestAns[8] <= 0;        TestPassed[8] <= 1'b0; //MOV_imm
    TestNumInst[9] <= 44;       TestAns[9] <= 32'h0000000c;     TestPassed[9] <= 1'b0; //LDR_5bimm

    TestNumInst[10] <= 50;  TestAns[10] <= 32'h00000000;    TestPassed[10] <= 1'b0; //
    TestNumInst[11] <= 64;  TestAns[11] <= 32'h0000ffe1;    TestPassed[11] <= 1'b0; //
    TestNumInst[12] <= 74;  TestAns[12] <= 32'h0000000c;    TestPassed[12] <= 1'b0; //
    TestNumInst[13] <= 84;  TestAns[13] <= 32'h00000045;    TestPassed[13] <= 1'b0; //
    TestNumInst[14] <= 94;  TestAns[14] <= 32'h0000ffe3;    TestPassed[14] <= 1'b0; //
    TestNumInst[15] <= 99;  TestAns[15] <= 32'h00000052;    TestPassed[15] <= 1'b0; //STRB_imm
    TestNumInst[16] <= 120; TestAns[16] <= 32'hffff0000;    TestPassed[16] <= 1'b0; //CMP(reg1)
    TestNumInst[17] <= 155; TestAns[17] <= 32'h00000007;    TestPassed[17] <= 1'b0; //LDR(5bimm)
    TestNumInst[18] <= 177; TestAns[18] <= 32'h00000009;    TestPassed[18] <= 1'b0; //STR(5bimm)
    TestNumInst[19] <= 199; TestAns[19] <= 32'h0000ffea;    TestPassed[19] <= 1'b0; //ADD(reg1)

    TestNumInst[20] <= 255; TestAns[20] <= 32'h00000057;    TestPassed[20] <= 1'b0; //ADD(reg1)
    TestNumInst[21] <= 275; TestAns[21] <= 32'h00000000;    TestPassed[21] <= 1'b0; //LW
    TestNumInst[22] <= 375; TestAns[22] <= 32'h0000ffd0;    TestPassed[22] <= 1'b0; //LW
    TestNumInst[23] <= 475; TestAns[23] <= 32'hffff0000;    TestPassed[23] <= 1'b0; //SLLI
    TestNumInst[24] <= 575; TestAns[24] <= 32'h00000005;    TestPassed[24] <= 1'b0; //LW
    TestNumInst[25] <= 675; TestAns[25] <= 32'h00000001;    TestPassed[25] <= 1'b0; //ADD
    TestNumInst[26] <= 775; TestAns[26] <= 32'h00000004;    TestPassed[26] <= 1'b0; //LW
    TestNumInst[27] <= 875; TestAns[27] <= 32'h0000ffd8;    TestPassed[27] <= 1'b0; //SLLI
    TestNumInst[28] <= 975; TestAns[28] <= 32'h0000ffd0;    TestPassed[28] <= 1'b0; //LW
    TestNumInst[29] <= 1000;    TestAns[29] <= 32'h0000ffd0;    TestPassed[29] <= 1'b0; //SW
end
```

Fig2-1. Testbench part1



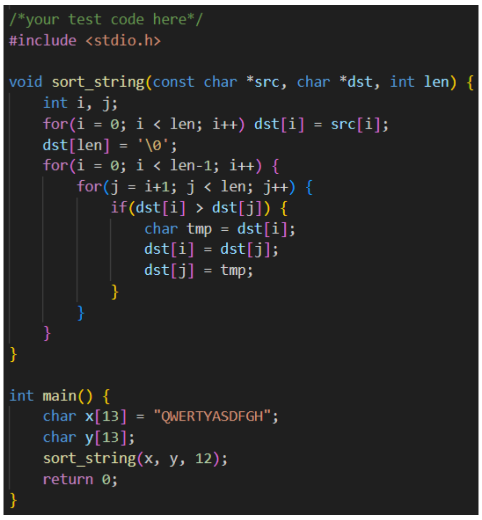
```verilog
always @ (negedge CLK) begin
    if (RESET_N) begin
        cycle <= cycle + 1;
        // $display("%d, %d", OUTPUT_PORT, NUM_INST);
        for(i=0; i<`NUM_TEST; i=i+1) begin
            if ((NUM_INST==TestNumInst[i]) && (TestPassed[i]==0)) begin
                if (OUTPUT_PORT == TestAns[i]) begin
                    TestPassed[i] <= 1'b1;
                    $display("Test #%0d has been passed", i+1);
                    if(i == `NUM_TEST-1) begin
                        $display("All tests completed. Success");
                        $finish();
                    end
                end
                else begin
                    TestPassed[i] <= 1'b0;
                    $display("Test #%0d has been failed! NUM_INST : %0d", i+1, NUM_INST);
                    $display("output_port = 0x%0x (Ans : 0x%0x)", OUTPUT_PORT, TestAns[i]);
                    $finish();
                end
            end
        end
    end
end
```

Fig2-2. Testbench part2

The upper photos are parts of my own testbench. I got ideas from the testbench of labs in EE312 course. I implemented two additional outputs at CortexM0 model; one is indicating the number of instructions went through the entire pipeline and the another is writing back data or storing data into the data memory.

Then, with the given instruction set simulator, I figured out the writing back value of instructions spaced by long distances, setting them as answers. Consequently, I could build a testbench which compares the answer and my implementation's output port, to eventually determine failure and success, at certain cycles which are spaced by long distances.

Moreover, I used string sorting program as an test program. Since sorting algorithms contains lots of branches, it enables me to evaluate the validity of the implementation in various environments. Behaviors about PUSH, POP, STM, LDM, BX, BLX, B, BL, and other load/store instructions can be strictly evaluated from this testbench as well as the sort program. Moreover, very specific instructions which are rarely used; for instance, UXTH, REV, XOR, can be evaluated through the test program in Fig 3-2. This program is the same with that I used in term project 1.

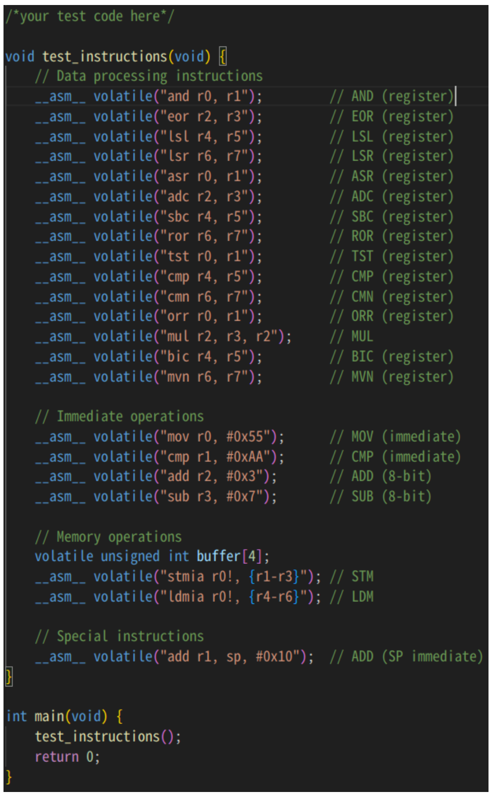


```c
/*your test code here*/
#include <stdio.h>

void sort_string(const char *src, char *dst, int len) {
    int i, j;
    for(i = 0; i < len; i++) dst[i] = src[i];
    dst[len] = '\0';
    for(i = 0; i < len-1; i++) {
        for(j = i+1; j < len; j++) {
            if(dst[i] > dst[j]) {
                char tmp = dst[i];
                dst[i] = dst[j];
                dst[j] = tmp;
            }
        }
    }
}

int main() {
    char x[13] = "QWERTYASDFGH";
    char y[13];
    sort_string(x, y, 12);
    return 0;
}
```

Fig3-1. Test program1 – sort program



```c
/*your test code here*/

void test_instructions(void) {
    // Data processing instructions
    __asm__ volatile("and r0, r1");          // AND (register)
    __asm__ volatile("eor r2, r3");          // EOR (register)
    __asm__ volatile("lsl r4, r5");          // LSL (register)
    __asm__ volatile("lsr r6, r7");          // LSR (register)
    __asm__ volatile("asr r0, r1");          // ASR (register)
    __asm__ volatile("adc r2, r3");          // ADC (register)
    __asm__ volatile("sbc r4, r5");          // SBC (register)
    __asm__ volatile("ror r6, r7");          // ROR (register)
    __asm__ volatile("tst r0, r1");          // TST (register)
    __asm__ volatile("cmp r4, r5");          // CMP (register)
    __asm__ volatile("cmn r6, r7");          // CMN (register)
    __asm__ volatile("orr r0, r1");          // ORR (register)
    __asm__ volatile("mul r2, r3, r2");      // MUL
    __asm__ volatile("bic r4, r5");          // BIC (register)
    __asm__ volatile("mvn r6, r7");          // MVN (register)

    // Immediate operations
    __asm__ volatile("mov r0, #0x55");       // MOV (immediate)
    __asm__ volatile("cmp r1, #0xAA");       // CMP (immediate)
    __asm__ volatile("add r2, #0x3");        // ADD (8-bit)
    __asm__ volatile("sub r3, #0x7");        // SUB (8-bit)

    // Memory operations
    volatile unsigned int buffer[4];
    __asm__ volatile("stmia r0!, {r1-r3}"); // STM
    __asm__ volatile("ldmia r0!, {r4-r6}"); // LDM

    // Special instructions
    __asm__ volatile("add r1, sp, #0x10");  // ADD (SP immediate)
}

int main(void) {
    test_instructions();
    return 0;
}
```

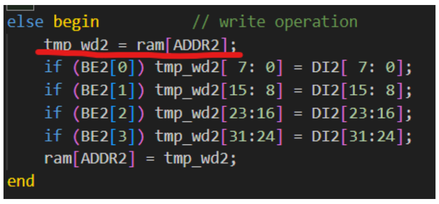
Fig3-2. Test program2 – assembly program

I passed all tests in my own testbench shown at Fig2 using Fig3-1, which are total 30. Also, I used the test program2 multiple times changing the code properly, finally verified that all tests are passed. Since I was able to figure out errors in the program easily through these test methods, it is reasonable to conclude that my test method is quite effective.

Additionally, I used gtkwave and iverilog for detailed simulations.

3. Notations

I think that memory model contains an error. Temp register is not initialized after the write(store) operations, so it modifies the memory data after store operations are finished. Therefore, I modified the memory file a little. The modification is shown below.



```
else begin          // write operation
    tmp_wd2 = ram[ADDR2];
    if (BE2[0]) tmp_wd2[ 7: 0] = DI2[ 7: 0];
    if (BE2[1]) tmp_wd2[15: 8] = DI2[15: 8];
    if (BE2[2]) tmp_wd2[23:16] = DI2[23:16];
    if (BE2[3]) tmp_wd2[31:24] = DI2[31:24];
    ram[ADDR2] = tmp_wd2;
end
```

Fig4. Modified memory part

I think it is legitimate. If there is no such red line code, the program has to read the memory before every write halfword or byte operation. This is not only inefficient but also absolutely not reasonable. Since I assumed this line is included in the memory file, if my guess is incorrect, there may be some malfunctions in my program.

4. Conclusion

In this project, I was able to learn how to implement and verify the pipelined CPU. I have several disappointment for my assignments. I wanted to implement more advanced branch prediction algorithms such as selective branch and out of order execution with Tomasulo algorithm. However, I do not have additional time to put for this project, so I finish here.

I think my program is quite seamless since I tested the program with several methods for multiple times, so in this aspect, I am satisfied with my assignment.

The project was arduous. I spent almost 50-60 hours, 1.5week. I hope there are no more projects left. I need to study other courses as well.

Thanks for great project. I learned a lot.