EE446 Laboratory Work 3

Tunahan Aktaş

1. Preliminary Work

1.1. Single Cycle Processor Design with Verilog HDL

In this laboratory work, we will design a single cycle ARM based processor with small set of instructions given in Figure 1.

Mnemonic	Name		Operation
ADD	Addition	add rA, rB, rC	rA ← rB + rC
SUB	Subtraction	sub rA, rB, rC	rA ← rB - rC
AND	Logical And	and rA, rB, rC	rA ← rB & rC
ORR	Logical Or	orr rA, rB, rC	rA ← rB | rC
LSR	Logical shift right immediate	lsr rA, rB, imm	rA ← (rB >> imm)
LSL	Logical shift left immediate	lsl rA, rB, imm	rA ← (rB << imm)
CMP	Compare	cmp rA, rB, rC	set the flag if (rA - rB=0)
STR	Store	str rA, [rB, imm12]	Mem[rB + imm] ← rA
LDR	Load	ldr rA, [rB, imm12]	rA ← Mem[rB + imm]

Figure 1: Instruction set of single cycle computer design.

Additionally, my design allows conditional instructions such as ADDEQ, SUBSGT, ANDS. I used ARM instruction format in my design as shown in Figure 2

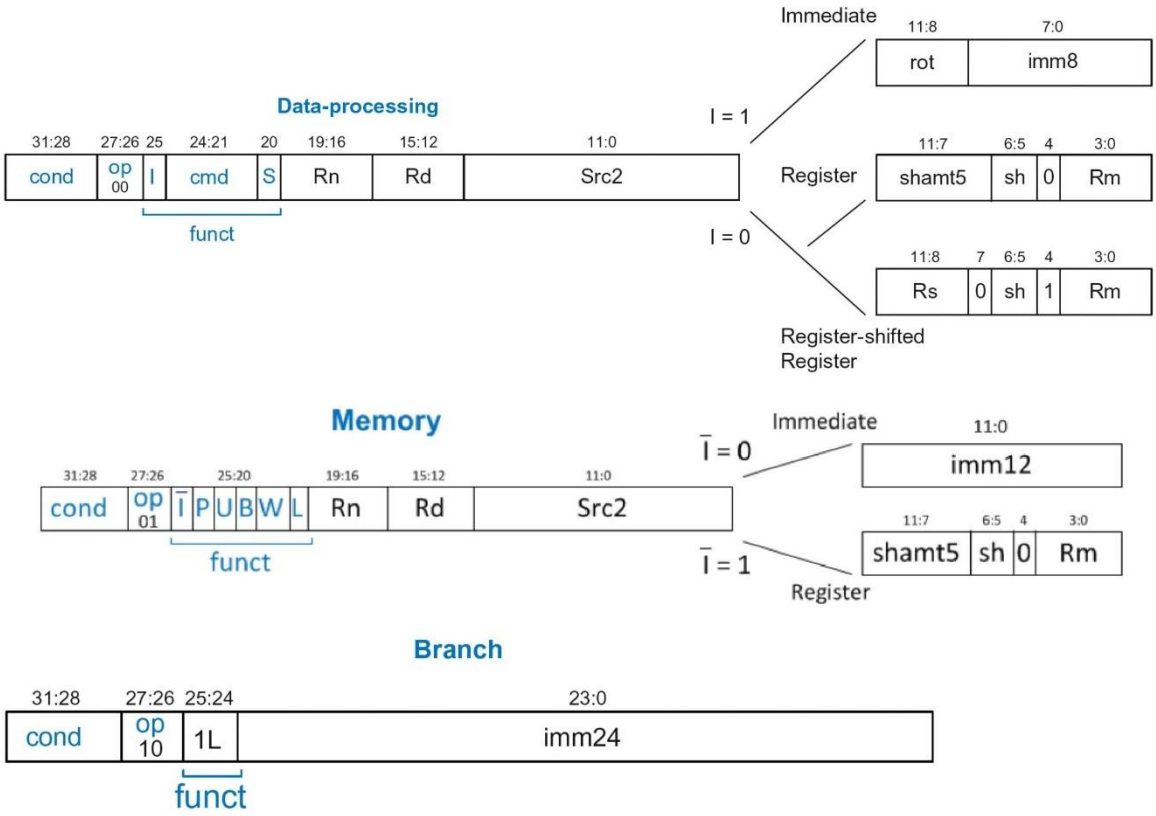


Figure 2: Shows 32-bit instruction formats.

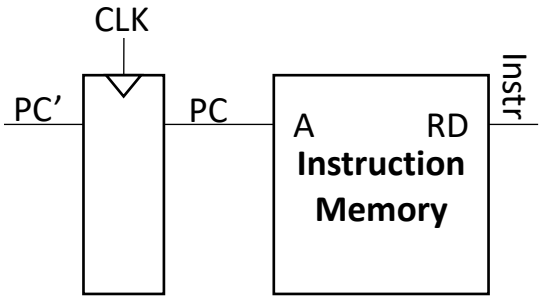
1.1.1.Datapath Design

1.1.1.1. LDR Instruction

LDR Rd, Rn, Imm
Rd =[Rn+Imm]

Fetch

To fetch the instructions, we need a program counter, which points to the instruction and an instruction memory. We have separate instruction memory because the single cycle processor that we will design has Harvard type architecture. *PC'* holds the next instruction address. When clock hits, *PC'* becomes *PC* and instruction is read from *Instruction Memory* as *Instr* at the same cycle.



Read Source Register and Increment PC

We have the current instruction at *Instr* at each cycle. Now, we will decode it. If we look at the instruction set in Figure 1, we see that for all instructions we need Rb register, which corresponds to Rn register in ARM instruction format given in Figure 2. Rn register is coded in [19:16] bits of the instruction. We will give the register number that we want to read to A1 port and the 32-bit register content will be ready at RD1 at the same cycle.

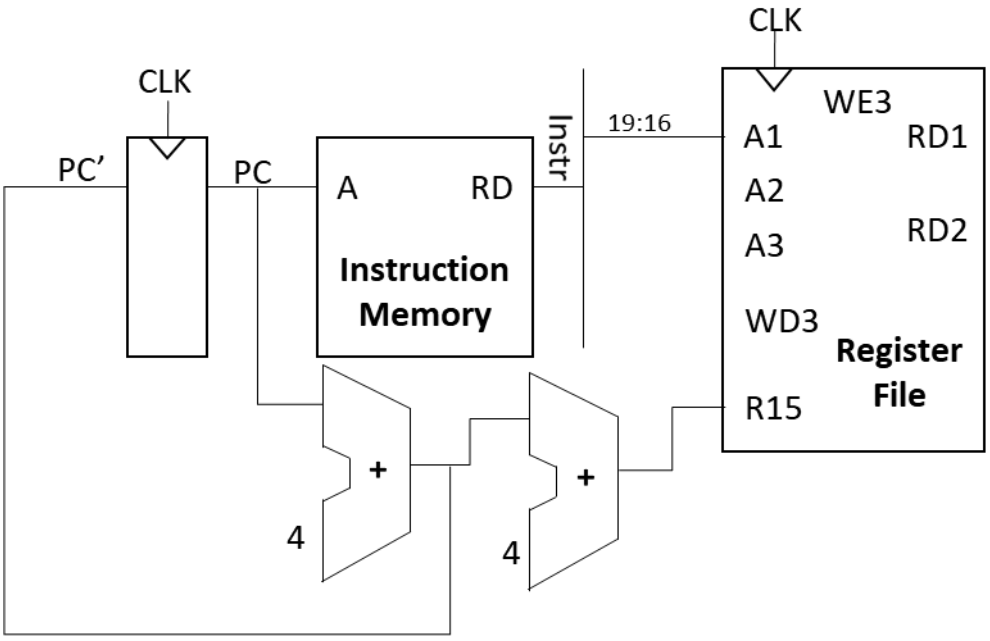


Figure 3: Extended datapath for incrementing the PC, reading the source register and updating the R15 with PC+8.

Extend the Immediate and Read Data

LDR is a memory type instruction. The instruction has immediate value at *Instr[11:0]*. However, our ALU needs 32-bit number as source. Therefore, we need to give extended with zero immediate value to ALU source.

So far, we have Rn + Imm value at *ALUResult*, which is the memory address and [Rn + Imm] value at *ReadData*, which is the data that we want to load Rd.

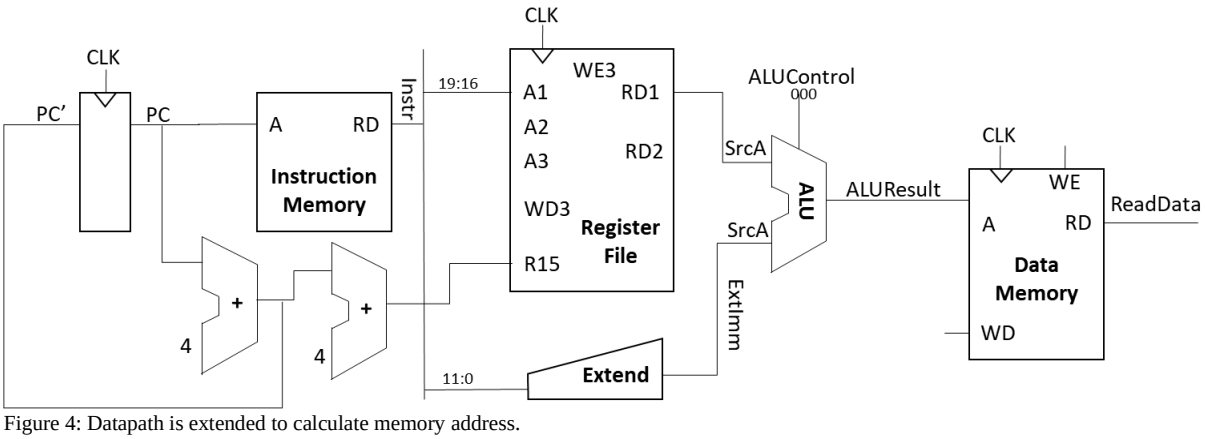


Figure 4: Datapath is extended to calculate memory address.

Writeback

Now, we will connect the destination register number from *Instr* to *A3* of register file and *ReadData* do *WD3* of register file. *WE3*, write enable, must be 1 for LDR instruction so that we can write the data at *WD3* to register pointed by *A3*.

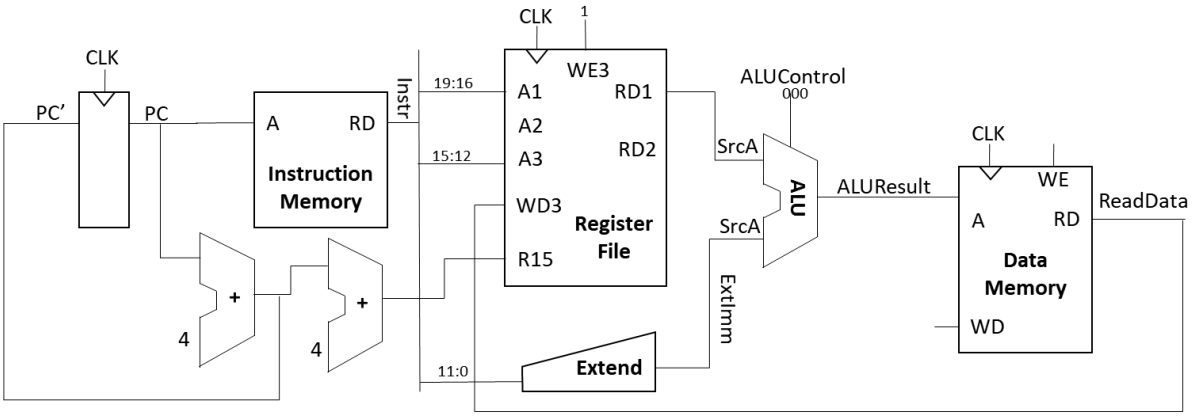


Figure 5: Extended datapath for writeback.

1.1.1.2. STR Instruction

Now, we will extend the datapath so that we can execute STR instruction:

STR Rd, Rn, Imm12

[Rn + Imm12] ← *Rd*

From the previous datapath, Figure 5, we already have content of *Rd* at *RD1*; however, for the STR instruction, we also need the content of *Rn*. Also, we must give the content of *Rd* register to the *WD* pin of the data memory, so that we can write it to the address provided by ALU, *[Rn + Imm12]*.

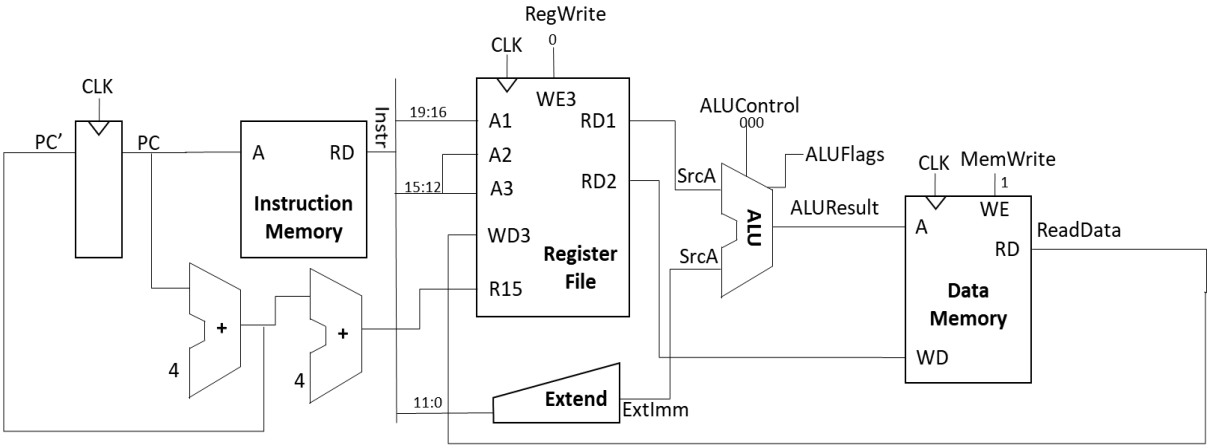


Figure 6: Shows the enhanced datapath for STR operation.

1.1.1.3. Data Processing Instructions

For data processing instructions, we need a path from ALU result to *WD3* of register file so that we can write the result of the operation. Therefore, we need a multiplexer to select between data memory result and ALU result. We can select the operation using ALUControl pin.

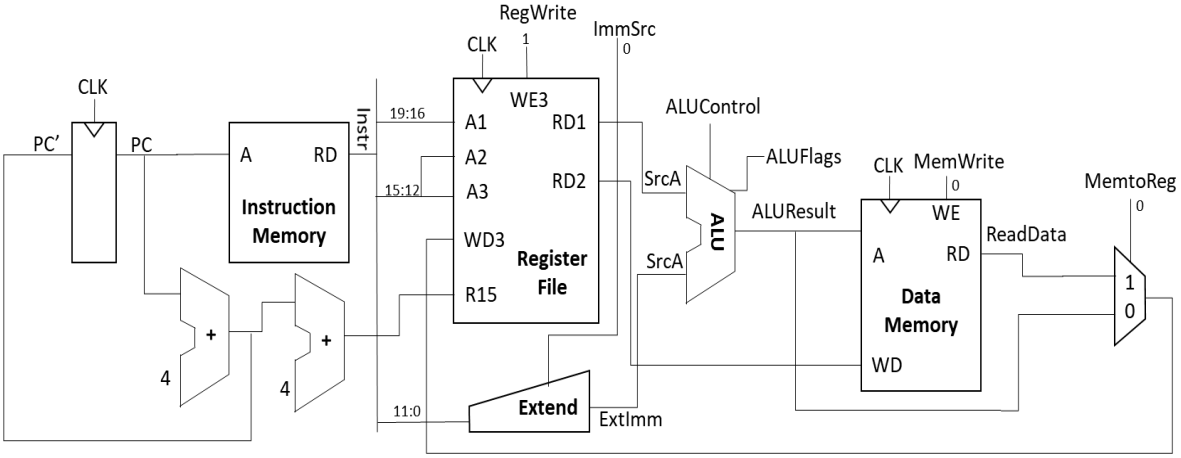


Figure 7: Shows the enhanced datapath for data processing operations.

1.1.1.4. Register Addressed and Shift Instructions

Current datapath in Figure 7 can only execute immediate data processing instructions like:
ADD Rd, Rn, #5
However, our ISA requires register addressed data processing instructions like:
ADD Rd, Rn, Rm
Also, we need to improve our datapath so that it can execute shift operations.

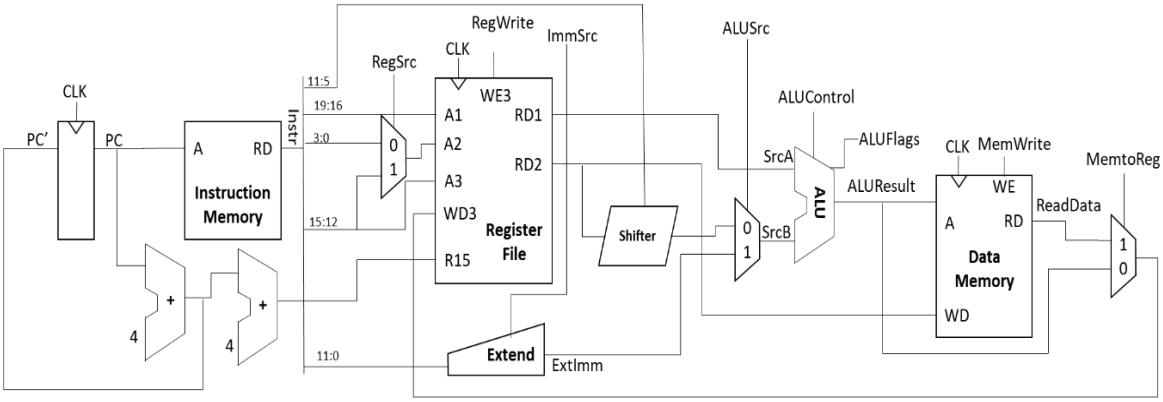


Figure 8: Shows the final datapath.

1.1.1.5. Black Box Diagram of Datapath

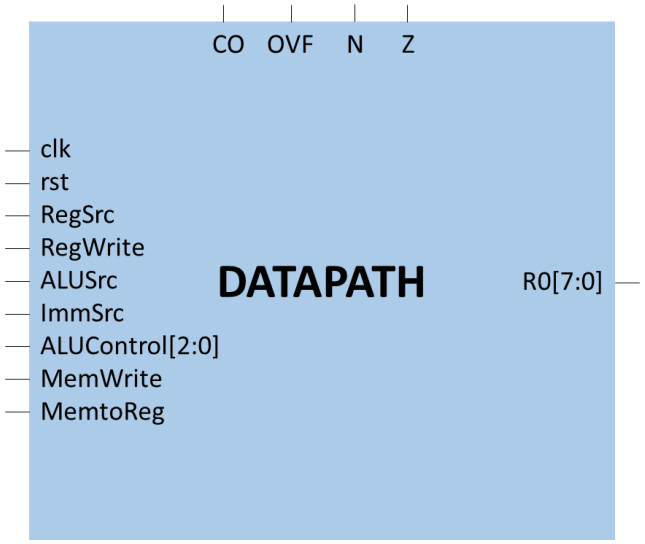


Figure 9: Shows the black box diagram of overall datapath design.

Instructions	Control Signals					
	RegSrc	RegWrite	ALUSrc	ALUCtrl	MemWrite	MemtoReg
ADD	0	1	0	000	0	0
SUB	0	1	0	001	0	0
AND	0	1	0	100	0	0
ORR	0	1	0	101	0	0
LSR	0	1	0	000	0	0
LSL	0	1	0	000	0	0
CMP	x	0	0	001	0	x
STR	1	0	1	000	1	0
LDR	x	1	1	000	0	1

We do not need *ImmSrc* signal because all the subset of ARM instructions that we consider in design are register referenced, not immediate, as seen in Figure 1.

1.1.2. Control Unit Design

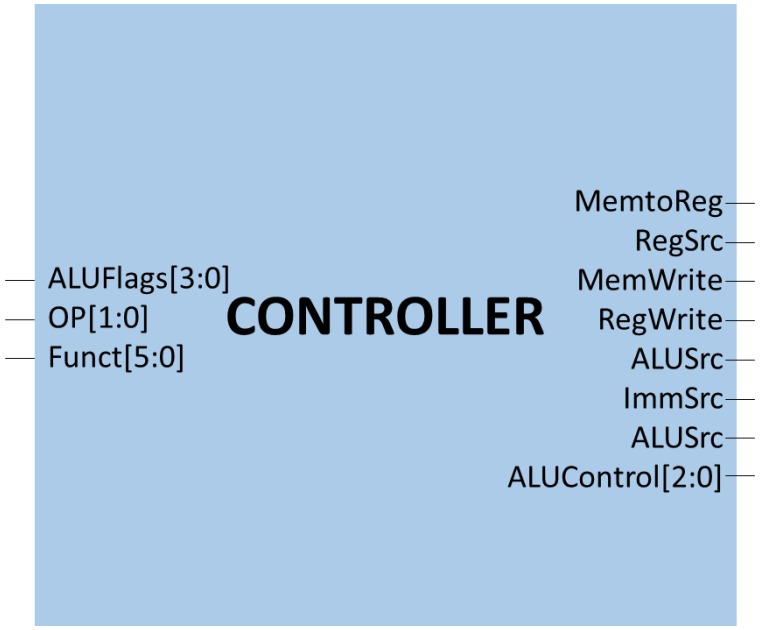


Figure 10: Shows the black box diagram of the controller unit.

As stated in the laboratory manual, the controller unit design should be similar to the design in the book. Therefore, inside the black box we have two units as Decoder and Conditional Logic. Decoder decodes the instruction and sends the control signals to Conditional Logic. Conditional Logic can mask these signals depending on whether the condition in the instruction is satisfied or not.

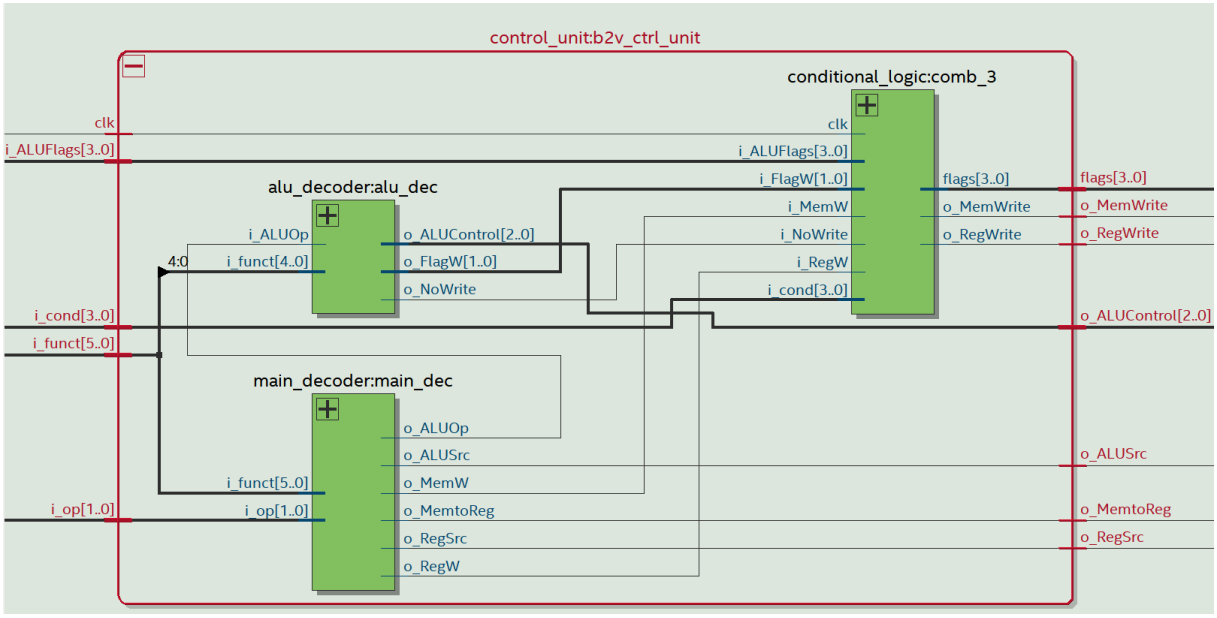


Figure 11: Shows my controller design on RTL viewer.

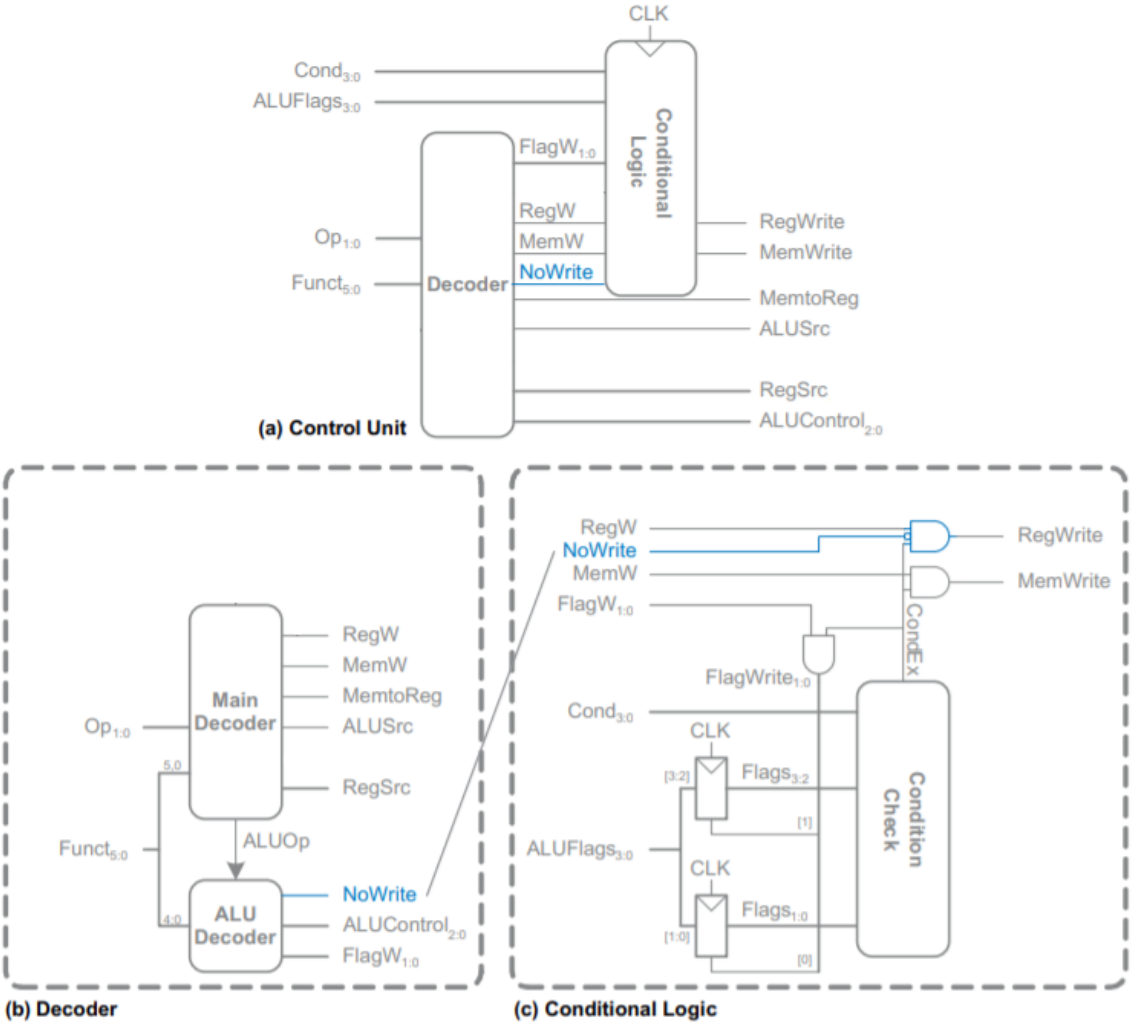


Figure 12: Control unit internal design in the book.

1.1.2.1. LDR Instruction

LDR Rd, Rn, Imm; Rd ←[Rn+Imm]

Table 1: Control signals for LDR instruction.

Decoder Outputs		
RegW	1	
MemW	0	
NoWrite	0	
ALUOp	0	; not a data processing operation
FlagW[1:0]	00	; do not write flags
Final Outputs (Condition satisfied)		
MemtoReg	1	; choose memory output as WD3 port of ALU
MemWrite	0	; do not write the memory
ALUSrc	1	; choose extended immediate as SrcB of ALU
RegWrite	1	; write to destination register, Rd
RegSrc	x	; do not care
ALUControl[2:0]	000	; ALUResult = SrcA + SrcB

1.1.2.2. STR Instruction

STR Rd, Rn, Imm; [Rn + Imm12] ← *Rd*

Table 2: Control signals for STR instruction.

Decoder Outputs		
RegW	0	
MemW	1	
NoWrite	0	
ALUOp	0	; not a data processing operation
FlagW[1:0]	00	; do not write flags
Final Outputs (Condition satisfied)		
MemtoReg	x	; do not care
MemWrite	1	; write to the data memory
ALUSrc	1	; choose extended immediate as SrcB of ALU
RegWrite	0	; not write to register file
RegSrc	1	; choose [Rd] as RD2 of register file
ALUControl[2:0]	000	; ALUResult = Rn + Imm

1.1.2.3. Data Processing Instructions

Table 3: Control signals for data processing instruction.

Decoder Outputs		
RegW	1	
MemW	0	
NoWrite	0	
ALUOp	1	; data processing operation
FlagW[1:0] S=0 S=1	 00 If add or sub:11 Else:10	 ; do not write flags ; write all flags ; write only N and Z flags
Final Outputs (Condition satisfied)		
MemtoReg	0	; ALU result to register file
MemWrite	0	; do not write the memory
ALUSrc	0	; choose Rm as SrcB of ALU
RegWrite	1	; write to destination register, Rd
RegSrc	0	; choose shifter unit output as RD2 of register file
ALUControl[2:0]	000 001 100 101	; ALUResult = Rn + Rm ; ALUResult = Rn – Rm ; ALUResult = Rn & Rm ; ALUResult = Rn | Rm

For LSR and LSL, *ALUControl[2:0] = 000, Rn = 0.* Shifter unit will give the shifted version of *Rm* and ALU will add with 0; therefore, we will have shifted unit as ALU result.

1.1.2.4. CMP Instruction

Decoder Outputs		
RegW	0	
MemW	0	
NoWrite	1	
ALUOp	1	; data processing operation
FlagW[1:0]	11	; write to flags

Final Outputs		
MemtoReg	x	; ALU result to register file
MemWrite	0	; do not write the memory
ALUSrc	0	; choose Rm as SrcB of ALU
RegWrite	0	; do not write to register file
RegSrc	0	; choose shifter unit output as RD2 of register file
ALUControl[2:0]	010	; ALUResult = SrcB

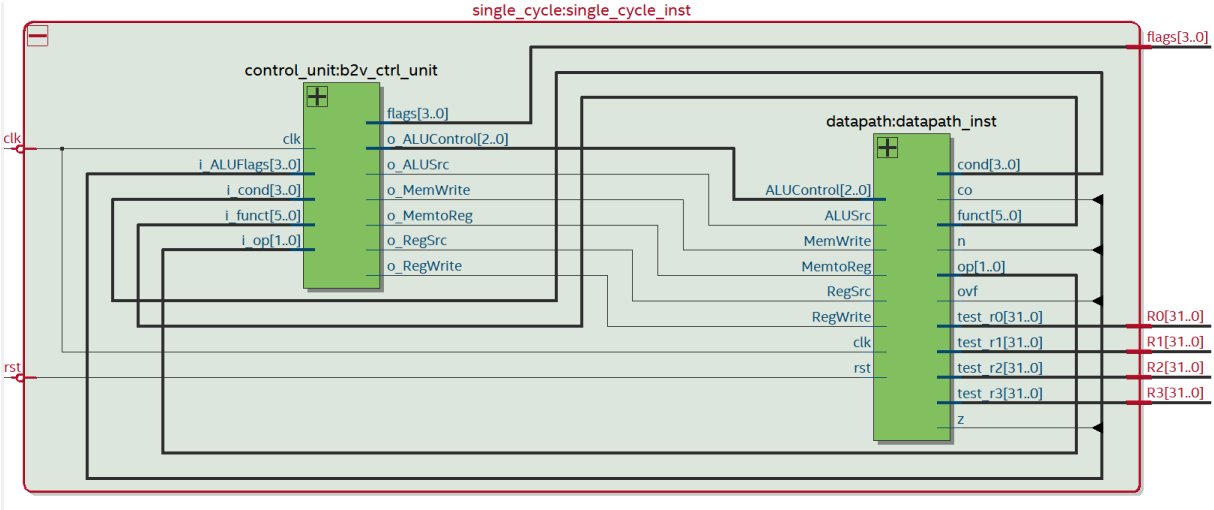


Figure 13: Shows the overall design on RTL viewer.

2. Experimental Work

Table 4: Shows the instruction memory. The module can be found in module_library.v

```
// Instruction memory
module instruction_memory #(parameter W=32)(
                                    input [W-1:0] i_A,
                                    output [W-1:0] o_RD
);
// RAM memory 4096x32-bit
reg [W-1:0] memory [0:255];

    integer i;
    initial begin       // test the architecture
        memory[0]  = 32'b1110_01_011001_0001_0000_000000000000;     // LDR R0,[R1,#0]    (R0
<- M[0]=15)
        memory[1]  = 32'b1110_01_011001_0010_0001_000000000001;     // LDR R1,[R2,#1]    (R1
<- M[1]=5)
        memory[2]  = 32'b1110_01_011001_0001_0010_000000000010;     // LDR R2,[R1,#2]    (R2
<- M[7]=7)
        memory[3]  = 32'b1110_00_010100_0110_0000_00000000_0111;        // CMP R6, R7
(0==0: EQ FLAG)
        memory[4]  = 32'b0001_00_000100_0000_0000_00000000_0001;        // SUBNE R0,R0,R1
(NOT EXECUTE)
        memory[5]  = 32'b0000_00_000101_0000_0000_00000000_0001;        // SUBSEQ R0, R0, R1
(R0 <- 15-5=10, UPD. FLAGS)
        memory[6]  = 32'b1011_00_001000_0000_0000_00000000_0001;        // ADDLT R0, R0, R1
(NOT EXECUTE)
        memory[7]  = 32'b1100_00_001000_0000_0000_00000000_0010;        // ADDGT R0, R0, R2
(R0 <- 10+7=17)
        memory[8]  = 32'b1110_01_011000_0010_0000_000000000100;     // STR R0,[R2, #4]
(M[11] <- 17
        memory[9]  = 32'b1110_00_001000_0010_0000_00000000_0001;        // ADD R0, R2, R1
(R0 <- 7+5=12)
        memory[10] = 32'b1110_01_011001_0010_0011_000000000100;     // LDR R3,[R2,#4]    (R3
<- M[11])
        memory[11] = 32'b1110_00_011010_0000_0011_00011_00_0_0001;  // LSL R3, R1, #3    (R3
<- 40)
        memory[12] = 32'b1110_00_011010_0000_0010_00001_01_0_0011;  // LSR R2, R3, #1    (R2
<- 20)
```

```
        memory[13] = 32'b1110_01_011001_0001_0010_000000000101;     // LDR R2,[R1, #5]   (R2
<- M[10]=0xAA)
        memory[14] = 32'b1110_00_000001_0010_0011_00000000_0001;        // ANDS R3, R2, R1
        memory[15] = 32'b1110_00_011000_0010_0010_00000000_0000;        // ORR R2, R2, R0
        memory[16] = 32'b1110_00_000101_0011_0010_00000000_0010;        // SUBS R2, R3, R2
        memory[17] = 32'b0000_00_000000_0010_0011_00000000_0001;        // ANDEQ R3, R2, R1
        memory[18] = 32'b1011_00_000000_0010_0011_00000000_0001;        // ANDLT R3, R2, R1

        for(i=19; i<255; i= i+1) begin
            memory[i] = 32'b0;
        end
    end

// read the instruction, pointed by 'i_A'
assign o_RD = memory[i_A[7:0]];

endmodule
```

Table 5: Shows the data memory. The module can be found in module_library.v

```
// Data memory
module data_memory #(parameter W=32)(
                                     input clk,
                                     input i_WE,
                                     input [W-1:0] i_WD,
                                     input [W-1:0] i_A,
                                     output [W-1:0] o_RD
);
// RAM memory 4096x32-bit
reg [W-1:0] memory [0:255];

// initially, read the instructions to our memory

    integer i;
    initial begin       // initializing memory for debug purposes
        memory[0] = 32'd15;
        memory[1] = 32'd5;
        memory[2] = 32'd0;
        memory[2] = 32'd0;
        memory[3] = 32'd0;
        memory[4] = 32'd0;
        memory[5] = 32'd0;
        memory[6] = 32'd0;
        memory[7] = 32'd7;
        memory[8] = 32'd0;
        memory[9] = 32'd0;
        memory[10] = 32'h000000AA; // ..1010_1010
        for(i=11; i<255; i= i+1) begin
            memory[i] = 32'b0;
        end
    end

// read the instruction, pointed by 'i_A'
assign o_RD = memory[i_A[7:0]];

always@(posedge clk)
    if(i_WE)
        memory[i_A[7:0]] <= i_WD;
endmodule
```

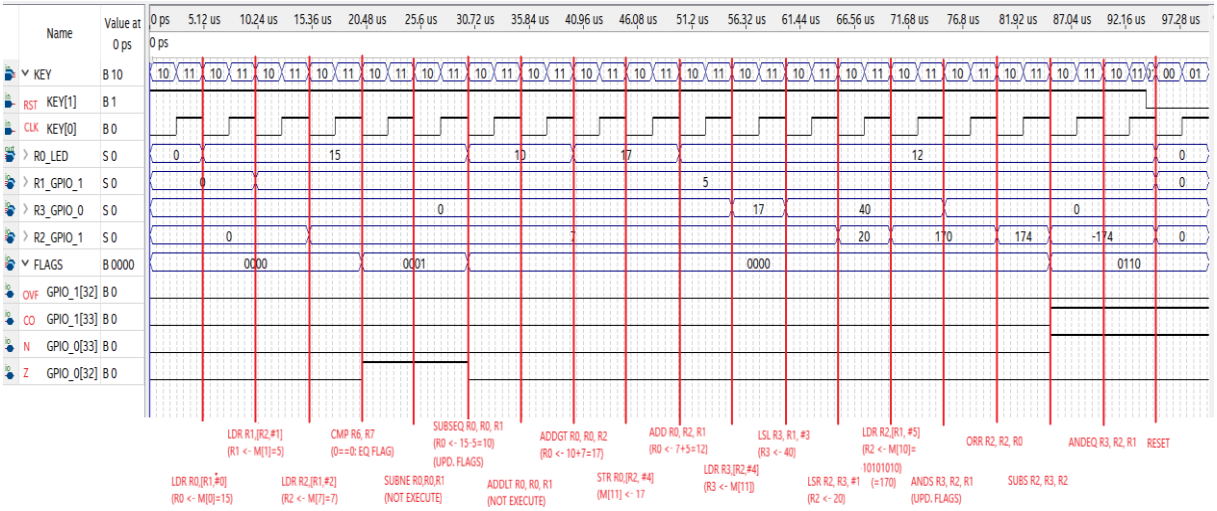


Figure 14: Shows the simulation result. Simulation file can be found as Waveform1.vwf