In this repository we will create an ARM legV8 CPU by using computer organization and design: The Hardware / Software interface, ARM edition

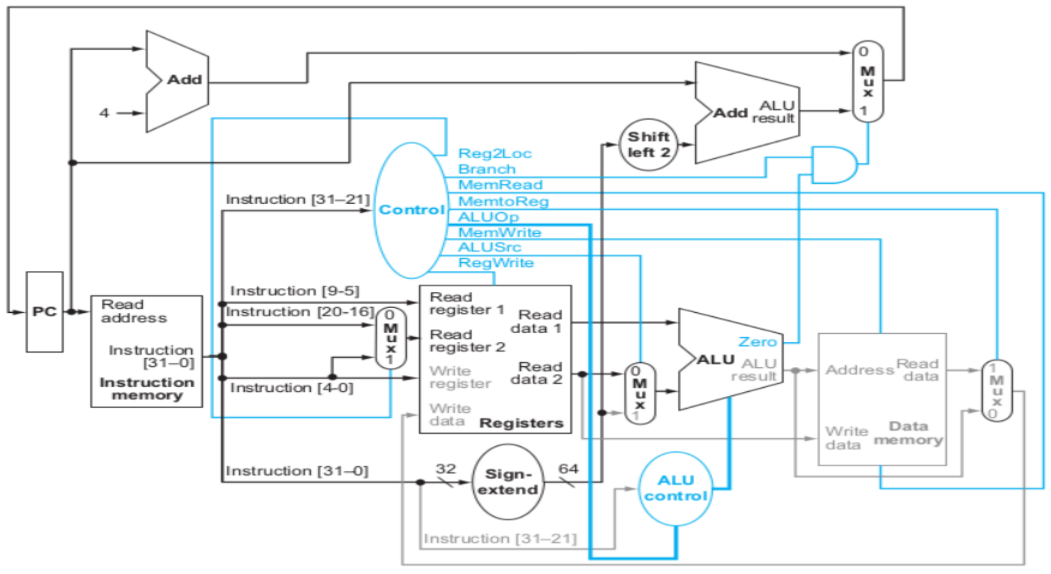


FIGURE 4.21 The datapath in operation for a compare-and-branch-on-zero instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

Figure 1 single cycle ARM CPU

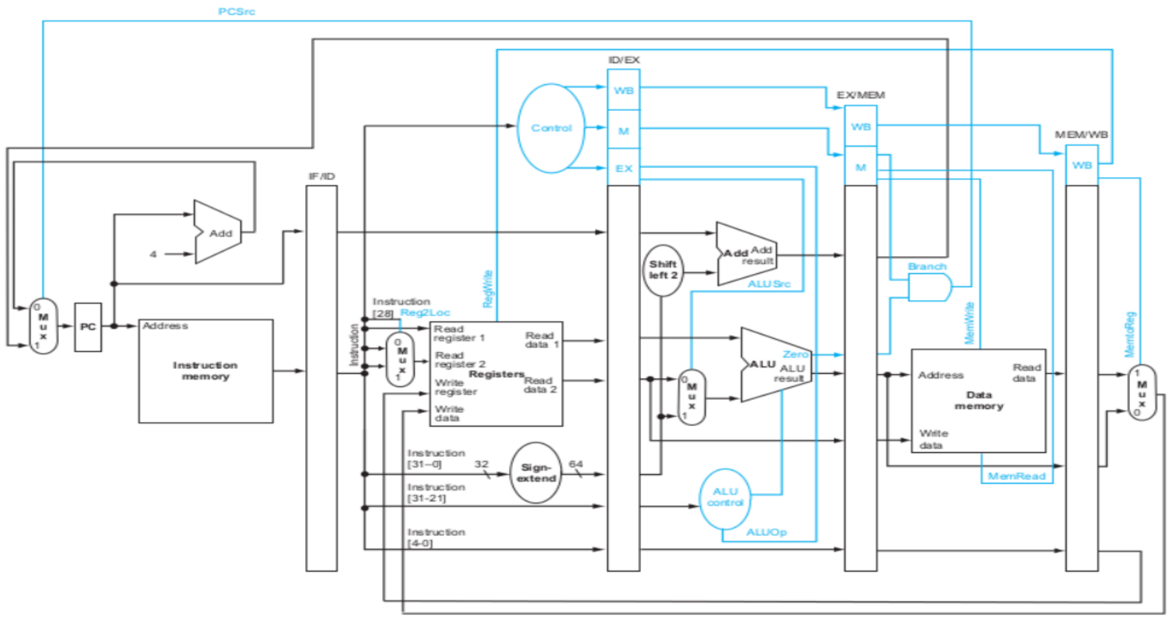
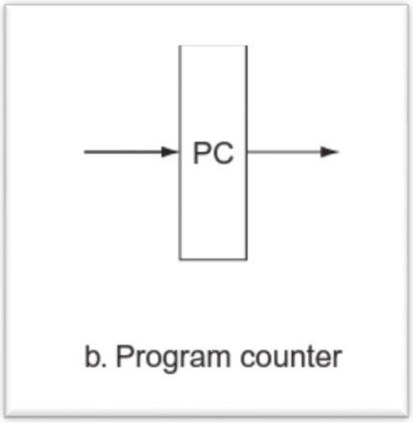


FIGURE 4.50 The pipelined datapath of Figure 4.45, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Figure 2 ARM CPU with pipeline

Introduction modules that we use in our CPU

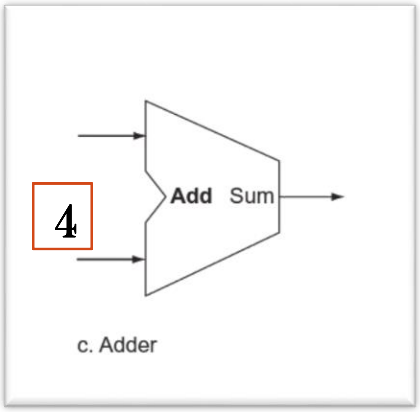


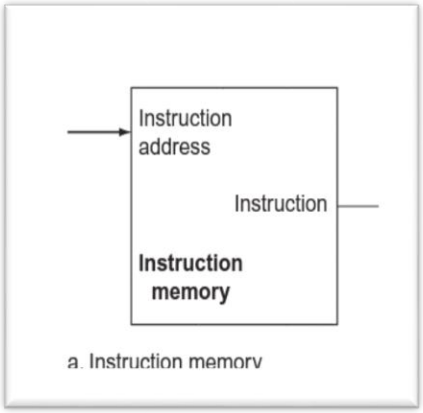
Program Counter (PC):

Program counter: The register containing the address of the instruction in the program being executed.
Inputs : NewPC, rst(Reset), w(Write), clk(Clock)
Output: OldPC

Adder :

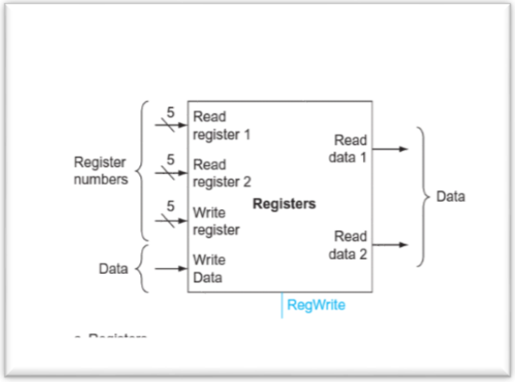
we will need an adder to increment the PC to the address of the next instruction.
We have to connect the output of the PC to the input of the adder
Inputs : a , b ,cin (Carry in)
Outputs: cout(Carry out) ,s(sum)





Instruction Memory :

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later
Input : readAddress (OldPC)
Output : instruction



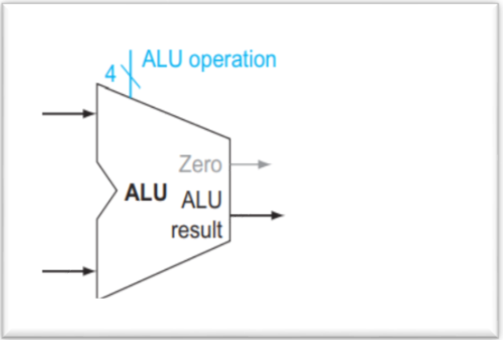
Registers:

The processor's 32 general-purpose registers are stored in a structure called a register file. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer.

Inputs : a(read register1),b(readregister2),

clk(clock),w(write),datac(Write Data)

Outputs : dataA(read data1),dataB(readdata2)



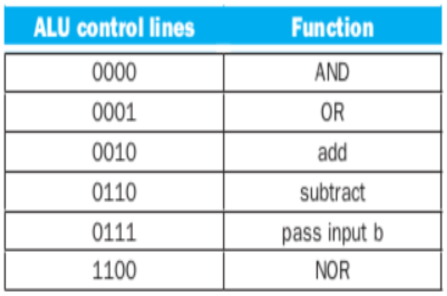
ALU:

ALU takes two 64-bit inputs and produces a 64-bit result, as well as a 1-bit signal if the result is 0

ALU connect with outputs of register.

Inputs :A ,B , Op(operation)

Output: f(ALU result),z(zero)



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

Figure 3 ALU operation functions

Clock:

Create clock cycle for CPU to create synchronization between components

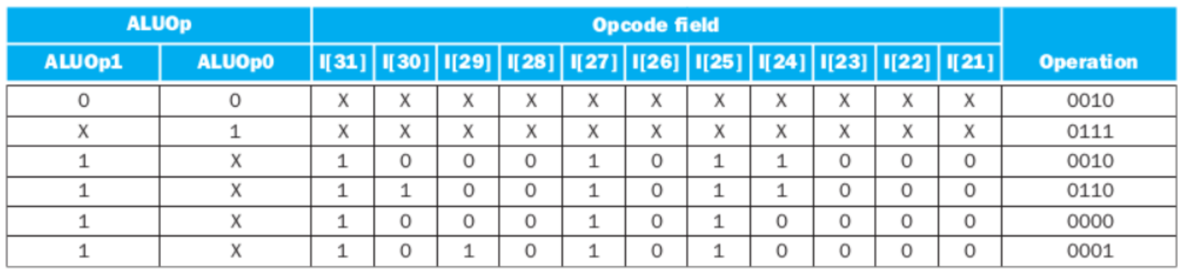
Output: clk

Mux:

We need some Multipelexer to select which data shuld enter to module and it control by control unit

Input: selector_multipelexer, data1, data2

Output: mux_out



ALUOp		Opcode field											
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[24]	I[23]	I[22]	I[21]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	X	0111
1	X	1	0	0	0	1	0	1	1	0	0	0	0010
1	X	1	1	0	0	1	0	1	1	0	0	0	0110
1	X	1	0	0	0	1	0	1	0	0	0	0	0000
1	X	1	0	1	0	1	0	1	0	0	0	0	0001

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and opcode field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. While we show all 11 bits of the opcode, note that the only bits with different values for the four R-format instructions are bits 30, 29, and 24. Thus, we only need these three opcode bits as input for ALU control instead of all 11.

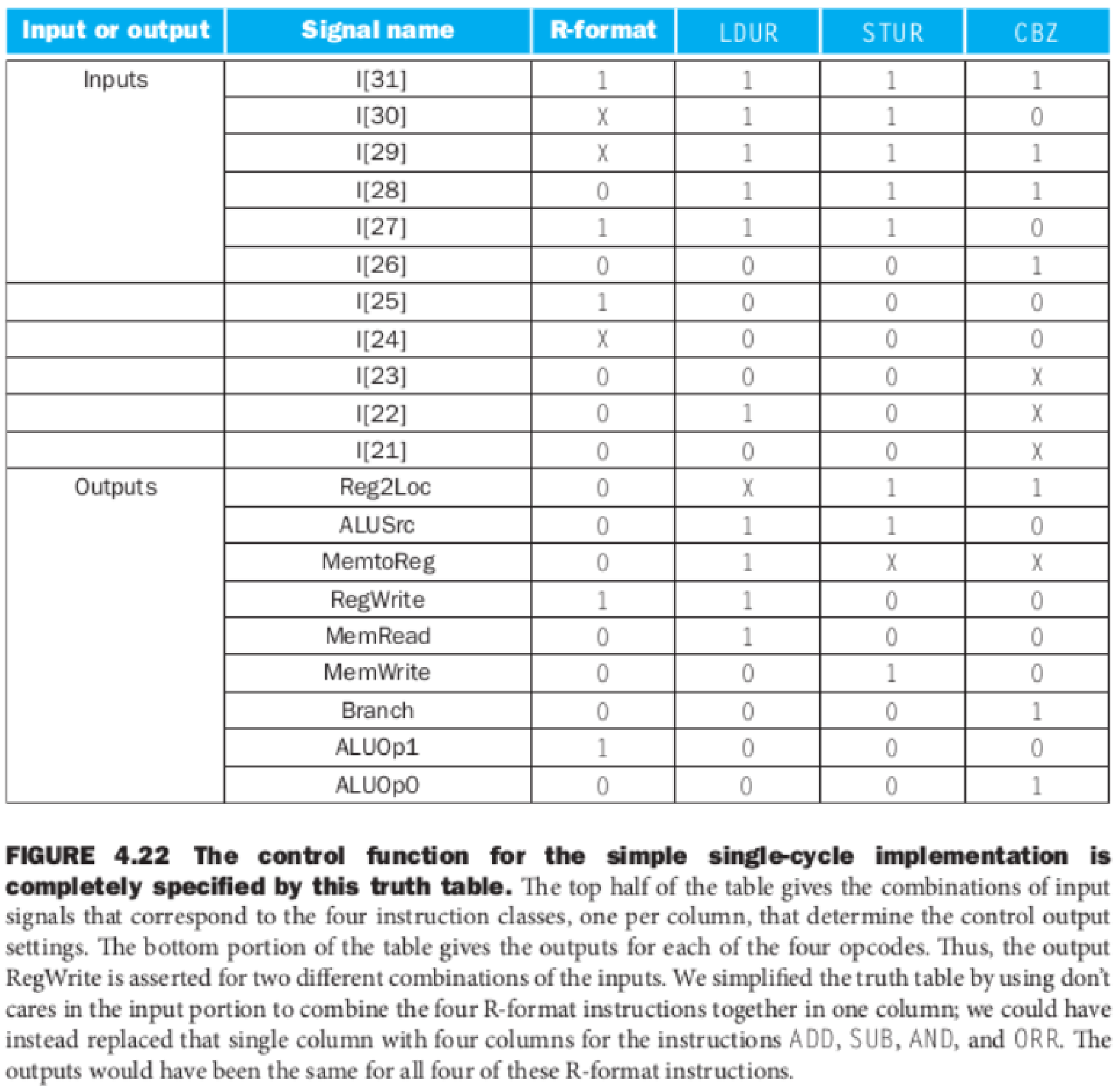
Figure 4 fetch ALU opcode from instruction opcode

ALU control:

It fetch ALU operation function from instruction

Input: ALUop, instruction

Output: opAlu



Input or output	Signal name	R-format	LDUR	STUR	CBZ
Inputs	I[31]	1	1	1	1
	I[30]	X	1	1	0
	I[29]	X	1	1	1
	I[28]	0	1	1	1
	I[27]	1	1	1	0
	I[26]	0	0	0	1
	I[25]	1	0	0	0
	I[24]	X	0	0	0
	I[23]	0	0	0	X
	I[22]	0	1	0	X
	I[21]	0	0	0	X
Outputs	Reg2Loc	0	X	1	1
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURE 4.22 The control function for the simple single-cycle implementation is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four instruction classes, one per column, that determine the control output settings. The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. We simplified the truth table by using don't cares in the input portion to combine the four R-format instructions together in one column; we could have instead replaced that single column with four columns for the instructions ADD, SUB, AND, and ORR. The outputs would have been the same for all four of these R-format instructions.

Figure 5 fetch control unit outputs from instruction opcode

Control Unit:

It control all module's performance by get instruction opcode

Input: instruction[31:21]

Output: regMux_selector, aluMUX_selector,

Mem_selector, wRegbank, readMem,

To add pipeline to our CPU we just need to add some register that can save data of each state and in next clock pass it to next state we can design it especially but we can use PC to do this job.

So for pipeline CPU as you see in file:(ARMCPU_pipeline.v) we just add some PC to save our data and pass them to next state by clock.

This project done by fatemeh mohammadi and Amirhossein Andarabi .