RISC BASED 8-BIT PIPELINED PROCESSOR

Report

Submitted by

Milin Modi (1190172)

Priyam Sheth (1233198)

EELE-5231 Computer Architecture

Prof. Dr. Ehsan Atoofian

In partial fulfilment for the award if the degree

of

MASTER'S IN SCIENCE

IN

ELECTRICAL AND COMPUTER ENGINEERING

LAKEHEAD UNIVERSITY

THUNDER BAY CAMPUS

Abstract

This report presents the design and implementation of a RISC-based 8-bit pipelined processor that executes a custom instruction set. The processor architecture consists of five pipeline stages: fetch, decode, execute, memory, and write-back. The instruction set supports arithmetic, logical, branch, load, store, and subroutine instructions with two addressing modes: immediate and register-register. The processor design is verified through simulation and synthesis using Verilog and Quartus Prime Lite Edition. The processor is implemented on a DE-10 Lite FPGA board and tested with various input and output scenarios. The report also discusses the challenges faced and the solutions adopted during the design process, such as hazard detection and resolution, performance optimization, and timing analysis. The report demonstrates the feasibility and efficiency of creating a pipelined processor using a RISC-like instruction set.

Table of Contents

List of Figures

List of Tables

1. Introduction

Purpose

The purpose of this project is to design and implement a pipelined processor that executes instructions based on a Reduced Instruction Set Computing (RISC) architecture. The aim is to create a processor capable of handling a specific set of instructions efficiently by breaking down the instruction execution into stages or pipeline phases.

Goals

1. Processor Design: Develop a processor architecture that supports pipelining, enabling parallel execution of instructions.

2. Instruction Set: Define and implement a RISC-like instruction set tailored to the processor's architecture, allowing a range of operations to be performed.

3. Pipeline Implementation: Create a pipeline structure with distinct stages for instruction fetch, decode, execute, memory access, and write-back.

4. Performance Optimization: Aim for maximum throughput and performance by reducing hazards and minimizing stalls or bubbles in the pipeline.

5. Functional Verification: Rigorously test the processor design to ensure correct functionality and adherence to the defined instruction set.

Scope

The project involves the complete design and implementation of a pipelined processor, including defining the instruction set, building the processor architecture, implementing pipelining techniques, optimizing performance, and validating the functionality through extensive testing and verification. The scope also encompasses identifying and addressing potential hazards, ensuring efficient data handling, and achieving desired throughput while meeting the project's requirements and constraints.

2. Design Specifications

The Processor must be strictly based on the RISC (Reduced Instruction Set Computing) Architecture and should have 5-Stage of Pipeline that are Fetch, Decode, Execute, Memory and Write Back. The Processor should have 4 Registers (R0, R1, R2, R3) with 1-byte of memory for data. The Address space of Instruction Memory and Data Memory is 256 bytes and the processor use Little Endian byte ordering. The length of all instructions is same and is of 2-byte. It should support ALU, Branch, Logical, Load and Store Instructions. The processor supports two addressing modes that are Immediate and Register-Register.

3. Architecture Description

The processor is designed for 5-stage pipeline, so we have fetch, decode, execute, memory and write back as our blocks.

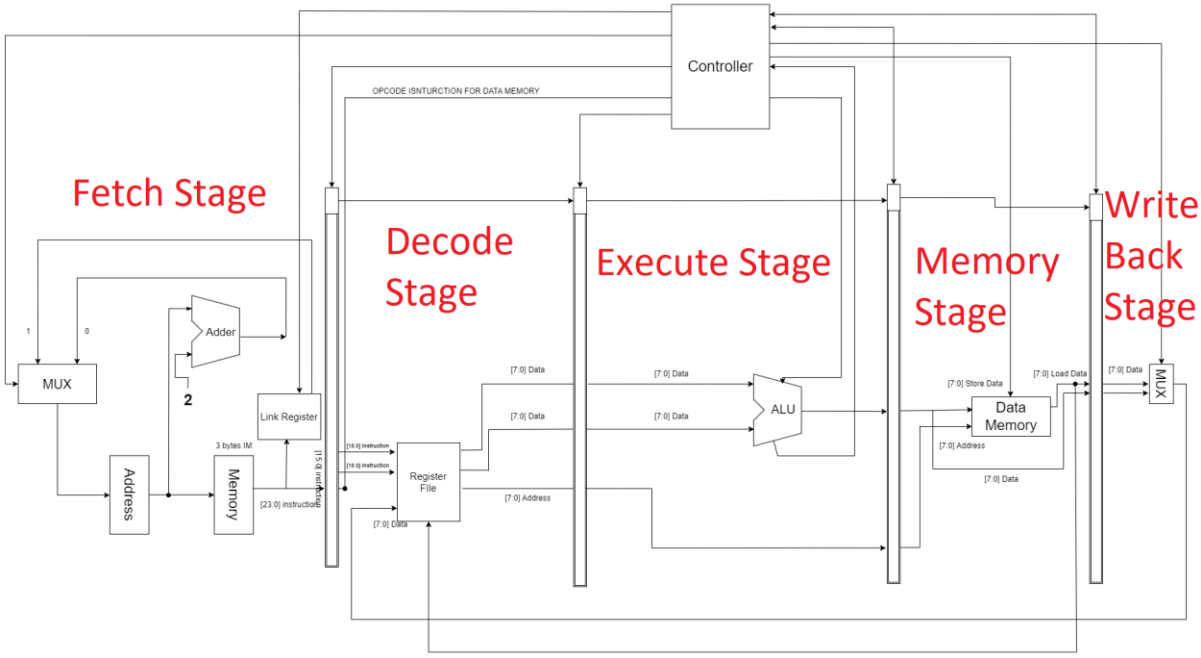

Figure 1.1 Architecture Design

In Fetch block, we have program counter, instruction memory, adder and link register. Initially the program counter is set to 0, from instruction memory the instructions are fetched with respect to the address and then it passed to the decode stage. After that, for next instructions, the program counter is incremented by 2 after every cycle and so, we will be able to fetch the next instruction. We have link register in the fetch stage which is used to store the address of the current instruction during the branch. The branch instruction will jump to another address which is given as the feedback. Whenever the subroutine is called, it will fetch the address from the link register and feed to the program counter, so that we will return to the normal execution of the program.

In decode stage, we have a register file which contains 4 registers of 1 byte in memory. This register decodes the instructions and store the data in registers. In our project, we are additionally passing the address from the decode stage as we will use that address in the memory stage for the storing the data in the data memory

In Execution stage, we execute the data that is present inside the register using ALU (Arithmetic and Logical unit). We perform various arithmetic functions such as add, sub, etc. logical functions such as shift left, shift right, etc. and provide the output in terms of Z (Zero) & N (Negative) flag and ALU Result. Here, different modes of operations for ALU is controlled by controller unit.

In Memory Stage, we have the ALU Result which will be stored in the Data Memory at the specified address whenever the store or load instruction is executed otherwise it will bypass the data memory and will be directed towards the write back stage.
In Write Back stage, we will write back the data that was executed by ALU inside the desired register.

We also have pipelined register which acts as connecting link between stages and also helps to synchronize the whole processor design. These registers help in storing the data and will provide data whenever it is needed. These registers are interconnected between each other so they can transfer data easily.

We have a controller unit, which controls the pipelined register, multiplexers, ALU mode and Data Memory. This unit is very important for the processor as it contains the information about when, which and how the instructions will be executed.

4. Instruction Set Architecture (ISA)

We use a RISC-like instruction set in this project. There are four 1-byte general purpose registers; R0, R2, R2, and R3. A special register which is called Link Register (LR) is used in BR.SUB and RETURN instructions. Instruction memory and data memory are separate, and the address space of each of the two memories is 256. The memories are byte addressable.

For extra credit, there are two optional instructions: BR.SUB and RETURN. BR.SUB instruction is used for subroutine call. RETURN instruction is used at the end of subroutines and changes the flow of program to the main program.

There are 3 different instruction formats:

1) A-Format

Figure 1 depicts an a-format instruction. These instructions are two bytes. The second byte is unused. In the first byte, the Op-code is the high order nibble, and the low order nibble specifies two registers.

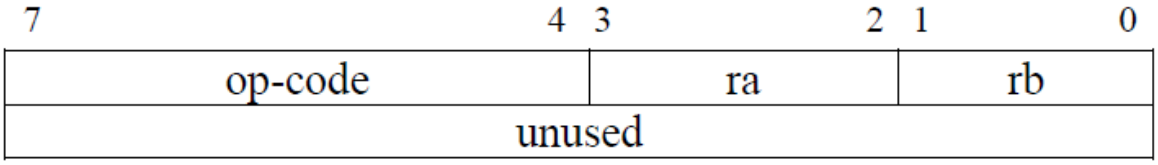

Figure 1.2 A – Format Instruction

Table I shows op-code values for a-format instructions and explains their functionality. R[Ra] and R[Rb] indicate values for registers Ra and Rb. There are two flags: zero flag(Z) and negative flag (N). Arithmetic instructions affect the z-flag and the n-flag. If the result of an arithmetic operation is zero/negative, the Z/N flag is set to one; otherwise, the flags are set to zero. The processor

has 1-byte input port and 1-byte output port. The ports are connected to the external pins. IN and OUT instructions transfer values between the processor ports and the internal registers.

Example

For *ADD r2, r1* instruction, the op-code, Ra, and Rb are as follows:

op-code = 1
Ra = 2
Rb = 1

The bit stream for the instruction is: 00011001. Hence, the hexadecimal format of the instruction in is: 0x19

Table 1.1 A – Format Instruction

Mnemonic	Op-code	Function
NOP	0	Nothing
ADD	1	R[ra] - R[ra] + R[rb]; (R[a] + R[rb]) = 0) => Z - 1; else Z - 0; (R[a] + R[rb]) <0) => N - 1; else => N - 0;
SUB	2	R[ra] - R[ra]-R[rb]; ((R[ra] - R[rb]) = 0) Z - 1; else Z - 0; (R[a] - R[rb]) <0) => N - 1; else =N - 0;
NAND	3	R[ra] - R[ra] NAND R[rb]; (R[a] NAND R[rb]) =0)=> - 1; else=>z-0: (R[a] NAND R[rb]) <0)=N-1;else=>N- 0;
SHL	4	Z - R[ra]<7>; R[ra] - (R[ra]<6:0>&0):
SHR	5	Z - R[ra]<0>; R[ra] - (0&R[ra]<7:1>):
OUT	6	OUT PORT - R[ra]:
IN	7	R[ra] - IN.PORT:
MOV	8	R[ra] - R[rb]:

2) B-Format

B-format instructions are two bytes and are used for branch instructions. As figure 2 shows, a b-format instruction has an op-code, brx, and effective address (ea) fields. The op-code of BR.Z and BR.N is the same, and brx field is used to distinguish the two instructions. The ea holds destination address in B-format instructions. The three low order bits of the first byte is unused

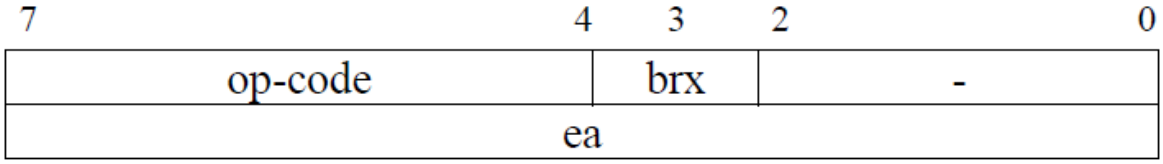

Figure 1.3 B – Format Instructions

Table II shows more details for a b-format instruction. BR instruction jumps into the destination address (ea). BR.Z is a conditional branch. If the Z flag is one, it jumps into the destination. Similarly, BR.N jumps into destination address if the N flag is one.

Optional Instructions

BR.SUB is used for the subroutine call. It saves the address of the next instruction in Link Register (LR) and jumps to the subroutine address. The second byte of instruction holds subroutine address. At the end of the subroutine, the RETURN instruction writes LR into PC. Note that LR is a special register and is separate from general purpose registers (R0 to R3).

Table 1.2 B – Format Instruction

Mnemonic	Op-code	Function
BR	9	PC - ea;
BR.Z	10	(brx=0 n Z=1) PC - ea ; (brx=0 n Z=0) PC - PC + 2;

BR.N	10	(brx=1 n N=1) => PC + ea; (brx=1 n N=0) => PC + PC+2
BR.SUB	11	LR - PC+2; PC - ea
RETURN	12	PC LR;

3) L-Format

L-format instructions are two bytes and are used for load/store instructions. The first byte holds the op-code and ra, and the second byte holds the address of the memory location or an immediate value. Figure 3 shows an L-format instruction. Note that in an L-format instruction, the first two low order bits of the first byte are unused.

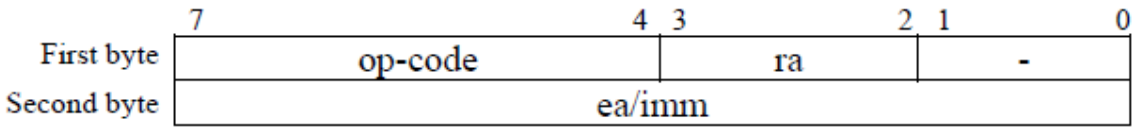

Figure 1.4 L – Format Instructions

Table III shows more details for L-format instructions. LOAD and STORE instructions write/read register ra into/from address ea. LOADIMM writes a constant value (imm) into register ra.

Table 1.3 L – Format Instruction

Mnemonic	Op-code	Function
LOAD	13	R[ra] - M[ea];
STORE	14	M[ea] - R[ra];

LOADIMM	15	R[ra] - imm;

5. Design Methodology

For the processor design, we have used bottom-up design approach, where we have connected the smaller blocks and designed the different blocks for fetch, decode, execute, memory and write back. We have used Verilog language for building these blocks. We have used structural and behavioral modeling for creating these blocks and designed the processor.

Tools: Quartus Prime Lite Edition (v.18.0.0.1)
Language: HDL Language (Verilog)
FPGA Board: DE-10 Lite Board (MAX 10 10M50DAF484C7G)
Board Specifications:

- On Board USB Blaster
- 64 MB SDRAAM, x16 Bits Data Bus
- Accelerometer
- One 2X20 GPIO Connector
- Arduino Uno R3 Connector
- 4-Resistor VGA
- 10 LEDs, 10 Switches, 2 Buttons
- 6 – Seven Segment Displays

6. Implementation Details

For implementation of processor, we will be having two blocks; first will be the counter block and second will be the CPU block.

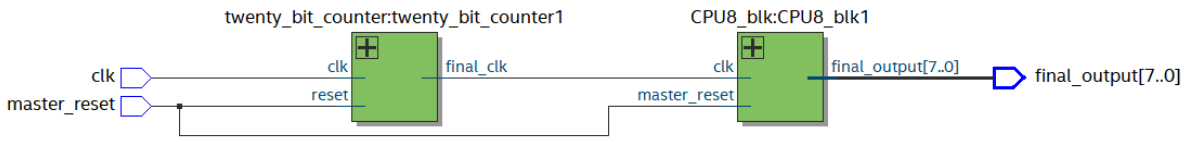

Figure 1.5 RTL View of Processor

The Counter block is used to reduce the reduce the frequency. As the FPGA board has 50MHz clock frequency which cannot be directly used as it is too fast for humans to observe, so we will reduce the frequency to 50 Hz so it can be observed. We will implement the 20-bit counter for this processor, so we can easily observe our output.

The CPU block contains various blocks such as fetch block, fetch-decode register, decode register, decode-execute register, execute block, controller block, execute-memory register, memory block, memory-writeback register and write back block. These blocks are connected with each other with the wires and thus executes after every clock cycle. Thus, the pipeline processor design is achieved.

Separate Verilog modules have been implemented for every block and thus interconnected with each other. Some modules are completely combinational and some of them are completely sequential. Based on input of clock sequential blocks will be activated while combinational blocks are independent of clock so it will work independently.

As an input we will be giving 0X0F from switches, clock and reset. We will obtain an 8-bit data stream as an output in the form of LED pattern.

7. Simulation and Verification

For, simulation of our project we have used Model-Sim to see the visual output. We have designed a testbench where we were giving clock and reset values as our input and getting the desired output. For testing purpose, we have given input from the fetch stage as 0X0F which is directly assigned to

input. On board implementation, we will be implementing these using switches, which will be used as input. We have pre-defined loaded instruction memory which is shown as below:

```
00000000:00000000       nop
                        00000001:00000000
                        00000010:00000000       nop
                        00000011:00000000
 start:                 00000100:01110000       in              r0   //set the
switches on the board to 4'hF
                        00000101:00000000
                        00000110:11100000       store           r0,add_nand
                        00000111:11111111
                        00001000:11110000       loadimm         r0,7
                        00001001:00000111
                        00001010:11100000       store           r0,counter
                        00001011:00011111
                        00001100:11110000       loadimm         r0,FF
                        00001101:11111111
                        00001110:11110100       loadimm         r1,FF
                        00001111:11111111
loop:                   00010000:01010000       shr             r0
                        00010001:00000000
                        00010010:01000100       shl             r1
                        00010011:00000000
                        00010100:10001100       mov             r3,r0
                        00010101:00000000
                        00010110:11010000       load            r0,add_nand
                        00010111:11111111
                        00011000:01010000       shr             r0
                        00011001:00000000
                        00011010:11100000       store           r0,add_nand
                        00011011:11111111
                        00011100:10000011       mov             r0,r3
                        00011101:00000000
                        00011110:10100000       brz             nand:
                        00011111:00100100
                        00100000:00010001       add             r0,r1
                        00100001:00000000
                        00100010:10010000       br              out_add_nand
                        00100011:00100110
nand:                   00100100:00110001       nand            r0,r1
                        00100101:00000000
out_add_nand:           00100110:01100000       out             r0
                        00100111:00000000
                        00101000:10110000       br.sub
count_decrement_subroutine
                        00101001:00110100
                        00101010:10000011       mov             r0,r3
                        00101011:00000000
                        00101100:10100100       brn             out
                        00101101:00110000
                        00101110:10010000       br              loop
                        00101111:00010000
out:                    00110000:10010000       br              start
                        00110001:00000100
                        00110010:00000000       noop
                        00110011:00000000
```

```
count_decrement_subroutine:

                        00110100:11010000      load            r0,counter
                        00110101:00011111
                        00110110:10001001      mov             r2,r1
                        00110111:00000000
                        00111000:11110100      loadimm         r1,1
                        00111001:00000001
                        00111010:00100001      sub             r0,r1
                        00111011:00000000
                        00111100:11100000      store           r0,counter
                        00111101:00011111
                        00111110:10000110      mov             r1,r2
                        00111111:00000000
                        01000000:11000000       return
                        01000001:00000000
```

As per the assembly code we will get the final output as follows:

0X81
0XC3
0XE7
0XFF
0XE7
0XC3
0X81

The above output goes on continuously for infinite times. These will generate a pattern for LED on board.

By resolving all the issues, we were getting the below output as shown in figure.

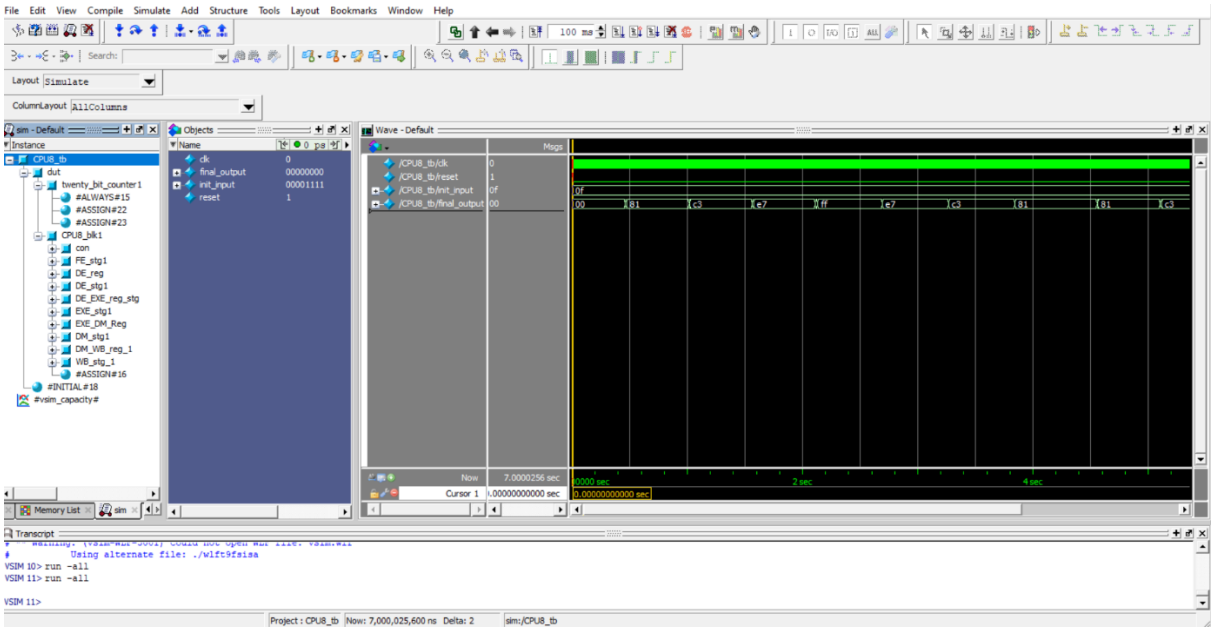

Figure 1.6 Simulation

For verification, we were checking our output stage by stage and resolving issues during undesired output. This led us to have a proper output on simulation.

8. Synthesis and Physical Implementation

- For synthesizable code, we need to remove all the initial values that were given during the testing phase.
- Also, we need to remove the clock where the clock is not required.
- Remove the unnecessary inputs and outputs from the design.
- Try to implement sequential and combinational blocks separately.
- Keep a track that combinational circuit does not contain clock and reset signal, while sequential signal will have clock and reset signal.
- Timing analysis is very important aspect to find about time to execute each block.
- For physical implementation, we need to keep track on the frequency of the clock.
- See that the device is compatible with the programmer and has the latest driver install so that code can be uploaded easily.

9. Challenges Faced

We have faced many challenges during implementation of our processor design. Some of them are listed below:

- Output Z and N flags from ALU came later. After the next instruction is executed (i.e. it is in decode stage), Z or N flag is reflected.
- SHR command works in 2 ways. One of the shift right instructions in the assembly code has the number 1's shifting to right until they are 0. Another SHR commands requires to add 1 in the MSB after all the bits are 0.
- IN and OUT took longer to execute.
- Data hazards.
- Executing branch format in any other stage than fetch stage caused multiple problems.
- Getting Z and N values from ALU.
- Stalling issues.
- Link Register storing address (causing to add inter-path address)
- Load immediate takes longer time to execute for loading data into register.
- Load and Store instructions takes too many clock cycles to execute as these instructions are performed till write back stage.

Solution to the above issues are as follows:

- For, Z or N Flag, we storing the values of Z and N in the register and is sent directly to fetch stage through controller to have proper execution for B - Format instructions.
- For SHR problem we have implemented that by comparing the value of R0 with zero, if it zero then we will send 1 bit for shift right and if it is not zero then we will send 0 bit for shift right.
- To remove data hazard hazards, we have implemented stall between the instructions by inserting NOP.
- We have solved the IN and OUT execution problem by resolving it DE stage.

- Branch was not able to jump in execute stage, so we have implemented it in the fetch stage where we were storing the current address into the link register and jump address is directly loaded into the current address.
- During branch we were not getting the values of Z and N flag due to which we were not getting the desired output. So, we have extended the wire of Z and N flag to fetch stage,
- To reduce the collision of instructions in branch, we have installed the stall between the instructions by adding NOP between the two instructions.
- We have resolved load immediate instruction in decode stage as it can save 1 clock cycle and also requirement of stall can be resolved.
- Load and Store takes longer time to resolve, in order to reduce the collision between the data and instructions, we have implemented stall between the instructions by inserting NOP.

10. Conclusion

This report presented the design and implementation of a RISC-based 8-bit pipelined processor that executes a custom instruction set and supports pipelining techniques, performance optimization, and functional verification. The processor was designed using Verilog language and synthesized using Quartus Prime Lite Edition. The processor was tested and verified using Model-Sim and DE-10 Lite FPGA board. The processor achieved a throughput and performance by reducing hazards, minimizing stalls, and implementing branch and subroutine instructions. The processor demonstrated the advantages of RISC architecture and pipelining, such as simplicity, efficiency, parallelism, and scalability. The project provided valuable learning experiences in computer architecture, processor design, Verilog coding, simulation and verification, synthesis and physical implementation, and debugging and troubleshooting.

11. References

1. Verilog Tutorial
   https://www.chipverify.com/tutorials/verilog
2. DE Lite – 10 FPGA Board
   https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=China&No=1021&FID=a13a2782811152b477e60203d34b1baa
3. Prof. Dr. Ehsan Atoofian Lecture Slides (Lakehead University – Electrical and Computer Engineering Department)
4. The Morgan Kaufmann Series in Computer Architecture and Design] John L. Hennessy, David A. Patterson - Computer Architecture, Sixth Edition_ A Quantitative Approach (2017, Morgan Kaufmann