8 Bits CPU Simulation Documentation

Compter Architecture and Organization

Software Engineering Program,

Department of Computer Engineering,

School of Engineering, KMITL

67011093 Chavit Saritdeechaikul
67011352 Theepakorn Phayonrat

Preface

This project, Design and Implementation of a Minimal 8-Bit CPU, was undertaken as part of the Computer Architecture and Organization course in the second year of Bachelor of Software Engineering at KMITL. It represents a practical exploration of fundamental computer architecture concepts, from logic design to instruction execution. The project allowed us to apply theoretical knowledge of CPU structure, instruction sets, and pipelining into a fully simulated and functioning processor. Using the Digital simulator by Hneemann, we developed and verified a modular CPU that embodies the essence of real processor operation. This report documents the design process, implementation details, and lessons learned throughout the development cycle.

Abstract

This mini project focuses on the design and simulation of a minimal 8-bit pipelined CPU using the Digital circuit simulation software. The CPU incorporates a five-stage pipeline: Fetch, Decode, Execute, Memory, and Write Back—and supports a custom instruction set architecture (ISA) with arithmetic, logical, branching, and I/O operations. The architecture features separate ROM and RAM modules, basic input and output ports, and a status register for flag management. Through simulation and test programs, the system demonstrates instruction execution, branching control, and interrupt handling. The project enhances understanding of CPU internals, digital logic integration, and the challenges of pipelined design, offering a hands-on approach to fundamental architectural concepts.

Contents

Chapter 1

Introduction

1.1 Project Overview

The 8-Bit CPU Simulation project aims to implement a simplified, educational model of a pipelined processor that captures the essential operations of modern CPUs while remaining manageable for simulation. The design follows a five-stage pipeline (Instruction Fetch, Instruction Decode, Execute, Memory, Write Back), allowing concurrent instruction processing for improved efficiency.

The CPU operates with 8-bit data paths and addresses, includes general-purpose and a special register (flag status register), and separates program memory (ROM) from data memory (RAM). A custom instruction set defines all operations, including data transfer (LD, ST, MOV), arithmetic and logic (ADD, SUB, MUL, AND, OR, XOR, NOT), branching (BZ, BNZ, BC, B), and I/O handling (RD, WR). Interrupt support and optional features such as basic caching or branch prediction may extend functionality.

Simulation and verification are performed in Digital, emphasizing modularity, waveform analysis, and clear documentation.

1.2 Background

Central Processing Units (CPUs) are the computational core of digital systems, responsible for executing instructions and managing data flow between memory and peripherals. Understanding how a CPU functions—from fetching an instruction to writing back results—is crucial in computer engineering education.

Traditional classroom learning often focuses on theoretical aspects such as microarchitecture, ISA design, and pipelining, but lacks direct visualization of hardware behavior. This project bridges that gap by using the Digital simulator to implement a CPU from basic logic components. By designing each pipeline stage, students explore how control signals, registers, buses, and memory interact to form a functioning processor.

The project serves as a foundational experience in CPU design, illustrating key topics such as instruction decoding, data hazards, memory access timing, and modular circuit organization.

1.3 Objective

1. **To design and implement** an 8-bit pipelined CPU with a custom instruction set architecture (ISA) using the *Digital* simulation environment.
2. **To apply theoretical concepts** of CPU architecture, including pipeline stages, ALU operations, branching, and flag management, in a practical design.
3. **To develop modular circuit components** (e.g., ALU, control unit, registers, and memory interfaces) that integrate seamlessly within the pipeline.
4. **To simulate and verify** the CPU's operation through test programs demonstrating arithmetic, logic, branching, stack, and interrupt handling.
5. **To enhance understanding** of digital logic design, data path organization, and hardware–software interaction in CPU execution.
6. **To document** the architecture, instruction formats, and verification results clearly and comprehensively for academic evaluation.

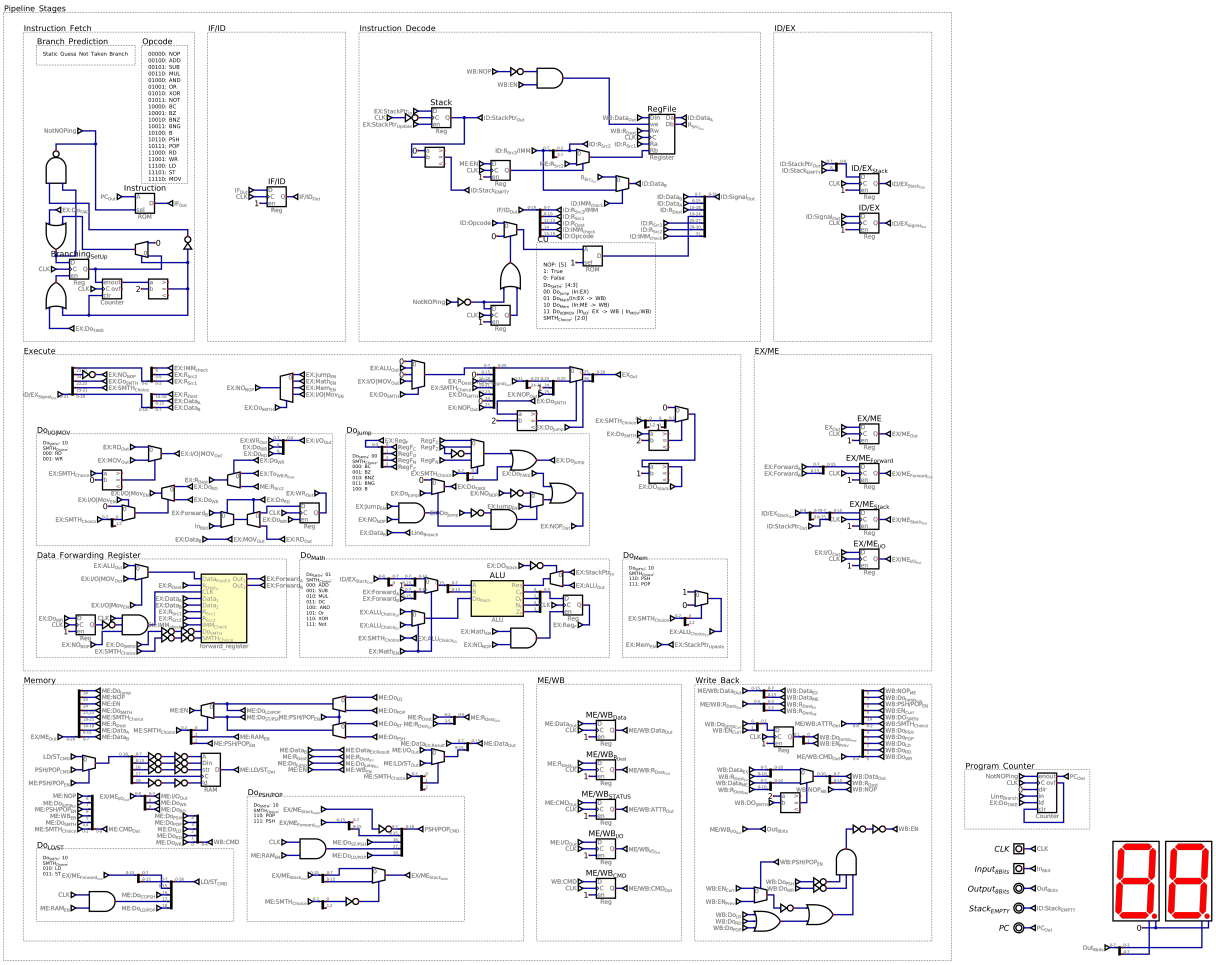
Chapter 2

Project Overview

2.1 Hardware Design

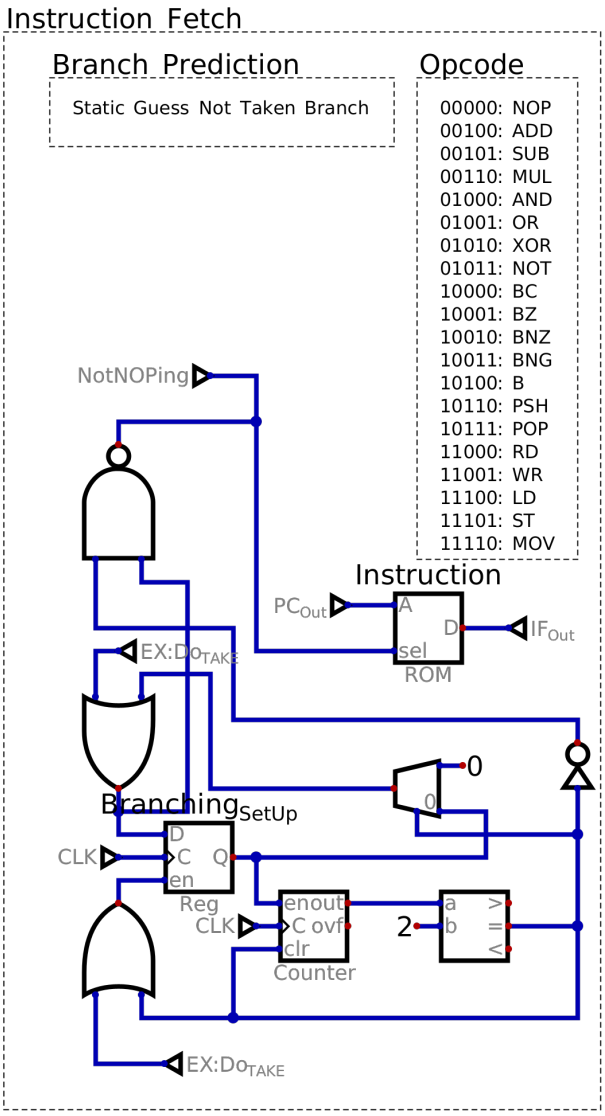
2.1.1 Structure

2.1.2 Overall Structure



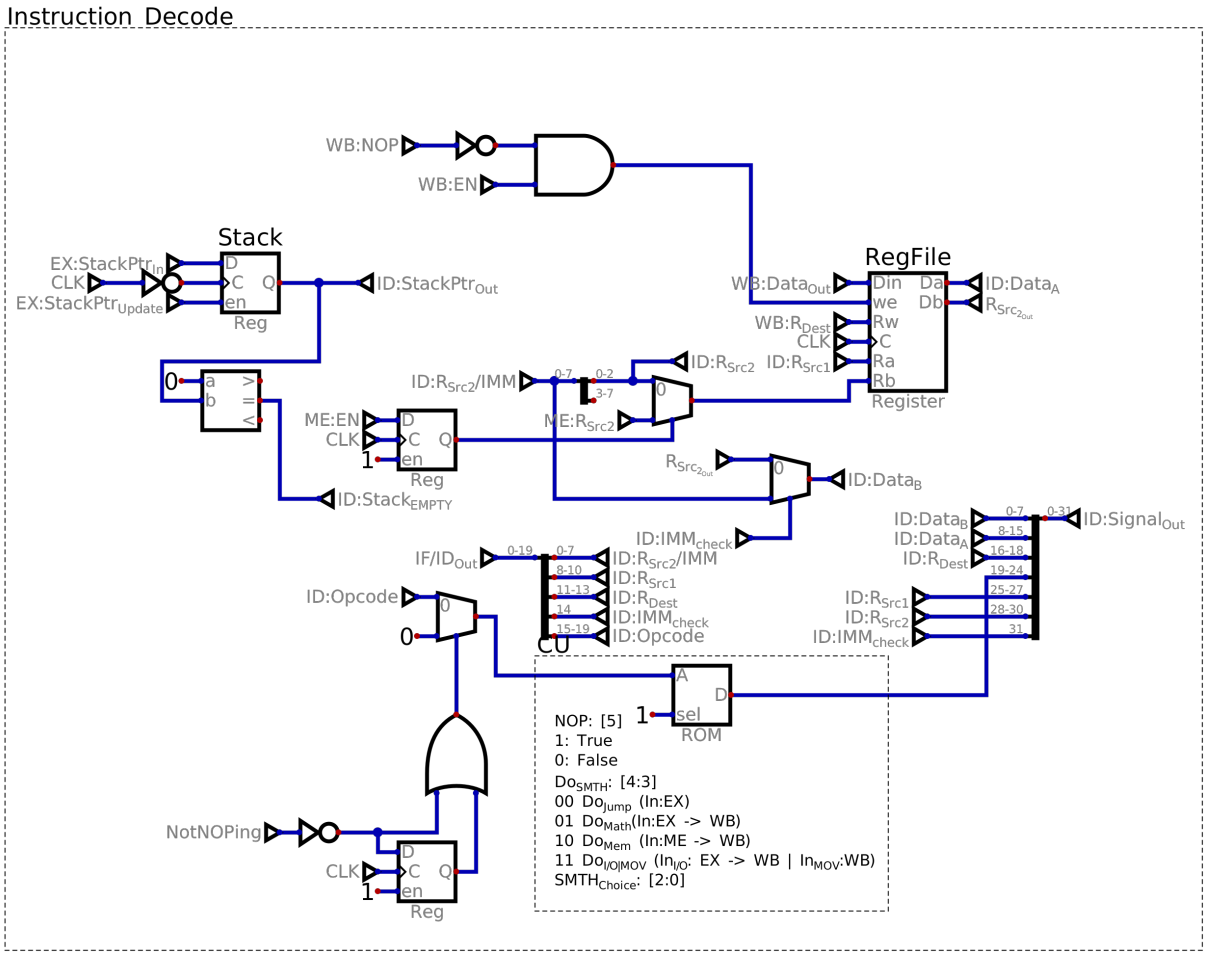
2.1.3 Pipeline Stages Dive Through

Instruction Fetch Stage Part



Fetch instruction to the `IF/ID` pipeline to be used to decode in the next stage.

Instruction Decode Stage Part



Decode the instruction and transfer to the `ID/EX` pipeline to execute in the next stage. In here, the system fetch the immediate value from the R_{Dest} R_{Src1}, if the system detects the IMM_{Check} is 0 it also fetch immediate value from R_{Src2} (The last 3 bits of the 20 bit instruction). Else, the system takes the last 8 bits of the instruction as the immediate value for that instruction executing in the next stage.

In the `Control Unit (CU)` we load the opcode into it and it output the execute signal for each opcode.

CU_ROM.hex

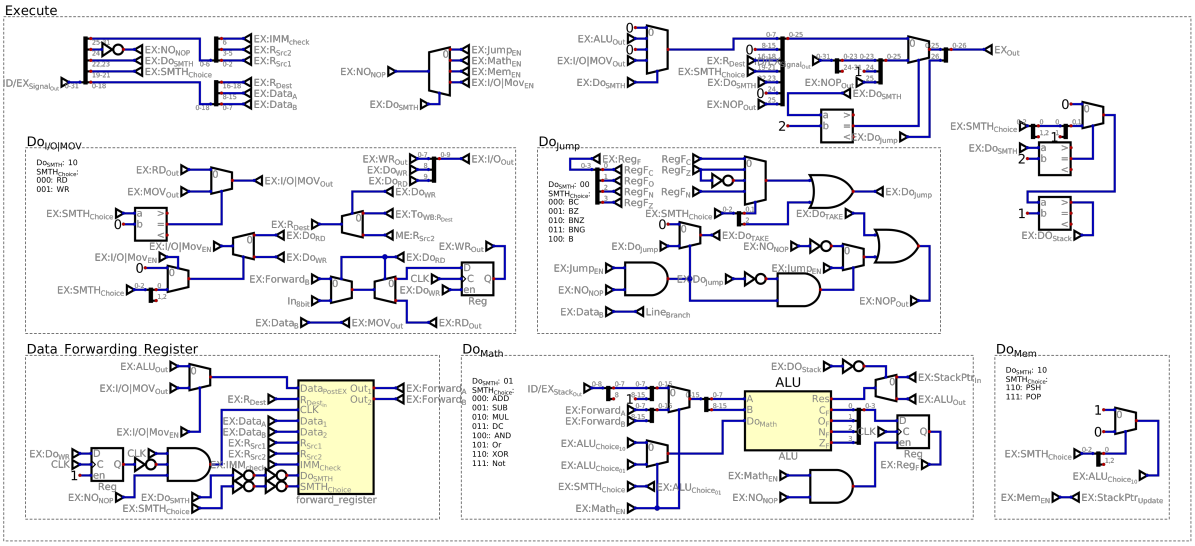
```
v2.0 raw
20
20
20
20
8
9
A
20
C
D
E
F
20
20
20
20
0
1
2
3
4
20
1A
1B
12
13
20
20
17
16
18
20
```

Control Unit Signal Format (Length: 6 bits):

$$NOP[5] \mid Do_{SMTH}[4:3] \mid SMTH_{Choice}[2:0]$$

Format	**Signal**	**Description**
Do_{Jump} Signal		
Do_{BC}	0 00 000	BC in $EX : Do_{Jump}$
Do_{BZ}	0 00 001	BZ in $EX : Do_{Jump}$
Do_{BNZ}	0 00 010	BNZ in $EX : Do_{Jump}$
Do_{BNG}	0 00 011	BNG in $EX : Do_{Jump}$
Do_{B}	0 00 100	B in $EX : Do_{Jump}$
Do_{Math} Signal		
Do_{ADD}	0 01 000	ADD in $EX : Do_{Math}$
Do_{SUB}	0 01 001	SUB in $EX : Do_{Math}$
Do_{MUL}	0 01 010	MUL in $EX : Do_{Math}$
Do_{AND}	0 01 100	AND in $EX : Do_{Math}$
Do_{OR}	0 01 101	OR in $EX : Do_{Math}$
Do_{XOR}	0 01 110	XOR in $EX : Do_{Math}$
Do_{NOT}	0 01 111	NOT in $EX : Do_{Math}$
Do_{Mem} Signal		
Do_{LD}	0 10 010	XOR in $ME : Do_{RAM}$
Do_{ST}	0 10 011	NOT in $ME : Do_{RAM}$
Do_{POP}	0 10 110	XOR in $ME : Do_{Stack}$
Do_{PSH}	0 10 111	NOT in $ME : Do_{Stack}$
$Do_{I/O\|MOV}$ Signal		
Do_{RD}	0 10 010	RD in $EX : Do_{I/O\|MOV}$
Do_{WR}	0 10 011	WR in $EX : Do_{I/O\|MOV}$
Do_{MOV}	0 10 000	MOV in $ME : Do_{I/O\|MOV}$, ready to write back in WB
NOP Signal		
NOP	1 | $DC[4:0]$	No Operation

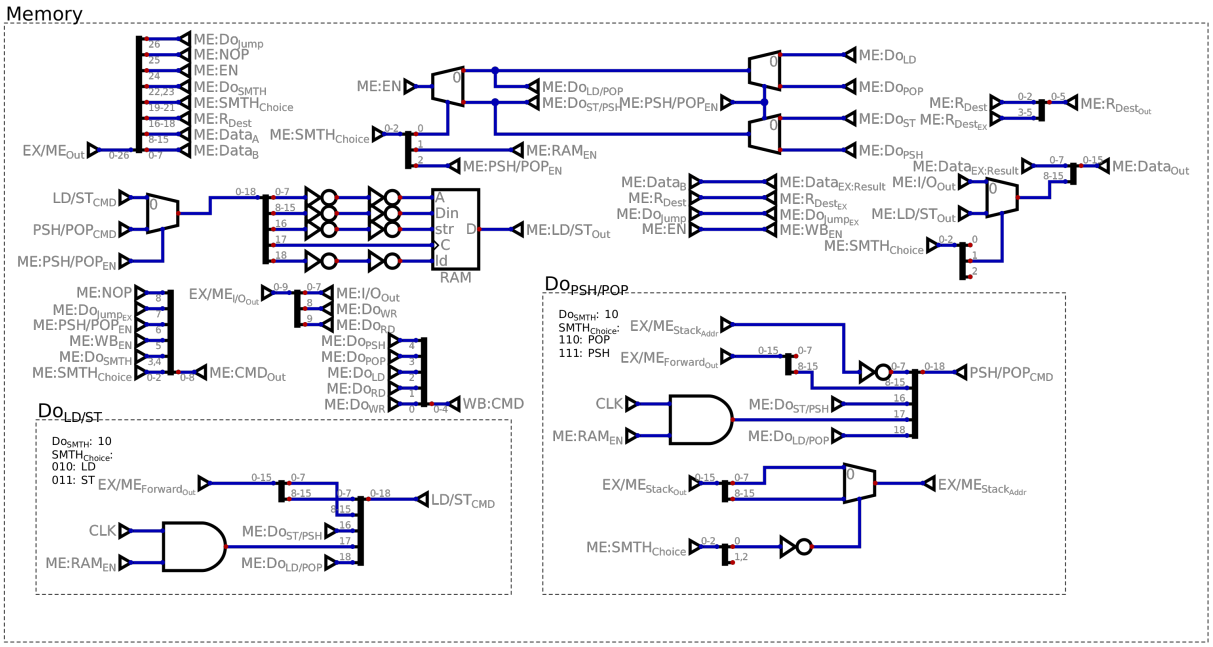
Execute Stage Part



In this stage, each instruction has 4 choices, whether to go do the calculation in the `ALU` or do the branch taking or do the `I/O` process or to pass through without doing anything.

If the instruction is a jump instruction (`BC` or `BZ` or `BNZ` or `BNG` or `B`), the system will start the pipeline flushing process by flushing the 2 instructions coming later in `IF` and `ID` stages and fill with `NOP` instruction to cleanup, then the system will fetch the instruction in the `Instruction Register` where address is equal to `Program Counter` value given in the instruction.

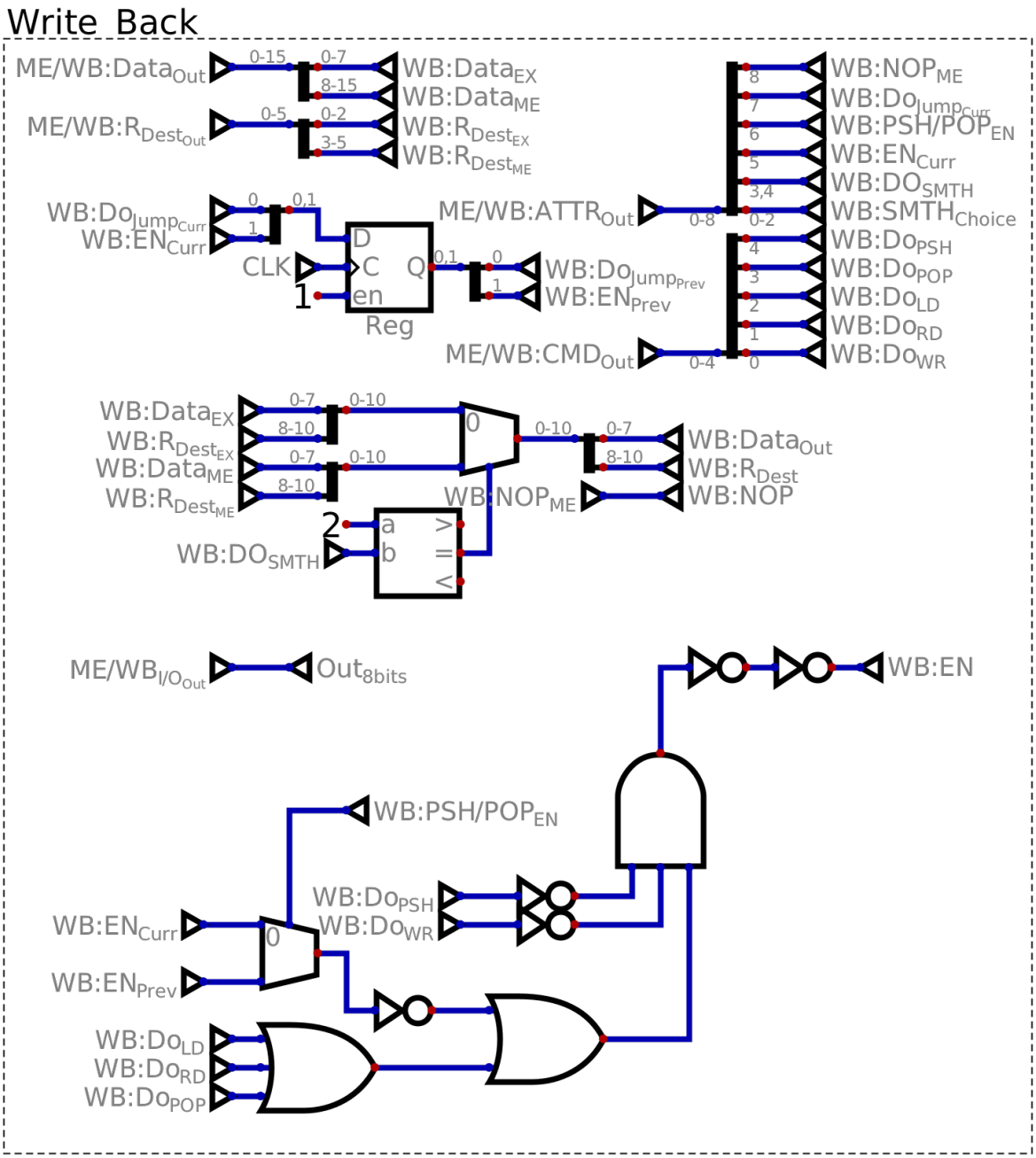
Memory Stage Part



In this stage each instruction has 3 choices, whether to pass through without doing anything, to load data into the `RAM` or to fetch data out from the `RAM`. If the instruction is either `LD` or `POP` the `RAM` outputs the data to the next stage. Else if the instruction is `ST` or `PSH` the `RAM` input data into the memory address given as the immediate or register to fetch address from.

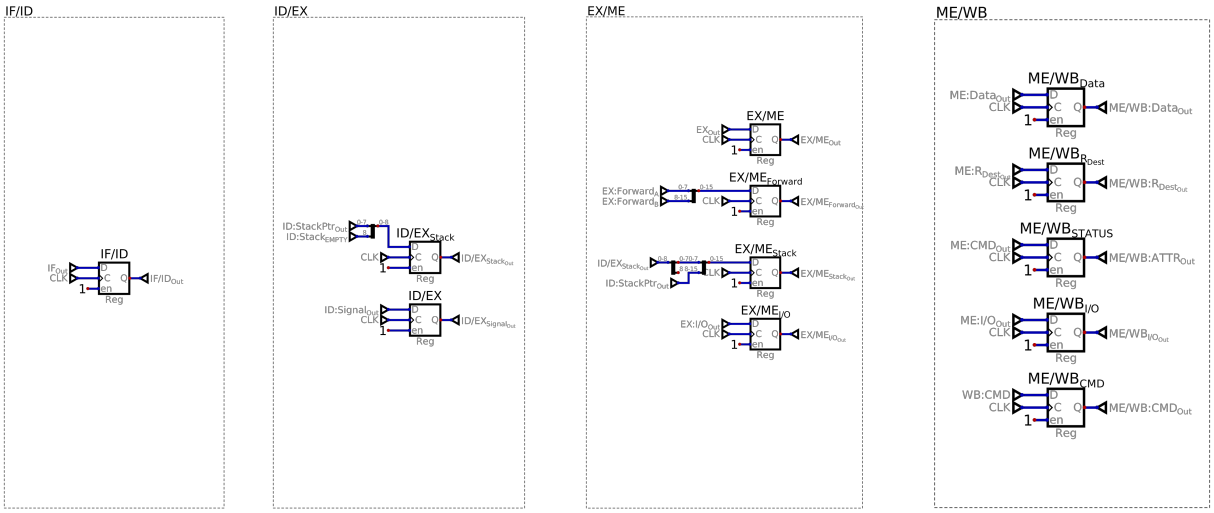
If the instruction is a stack instruction (`PSH` or `POP`), the data will load and fetch from the back most of the `RAM` and will increment or decrement the `Stack Pointer` base on the instruction.

Write Back Stage Part



Once every data are processed in each pipeline stage, in `Wb` stage, they are now ready to be written back into the `Register File`. For the condition to enable the write back, the instruction needs to be either arithmetic or logic or `RD` or `LD` or `POP`. All of these instruction all give back value to be loaded into the register in the `Register File`.

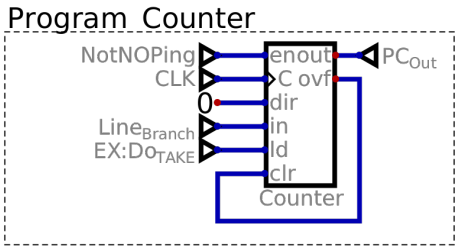
2.1.4 Pipeline Buffers Dive Through

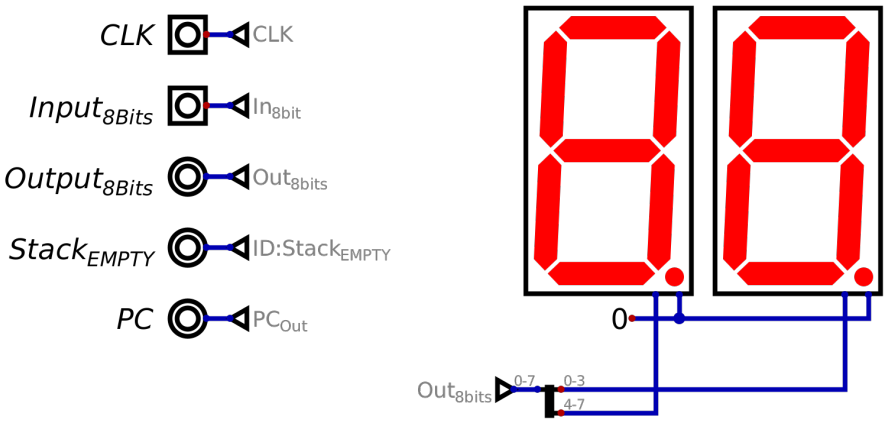


Pipeline buffers are placed between stages to prevent data and control signals to overlap in each stage.

2.1.5 Other Parts Dive Through

Program Counter and I/O Parts

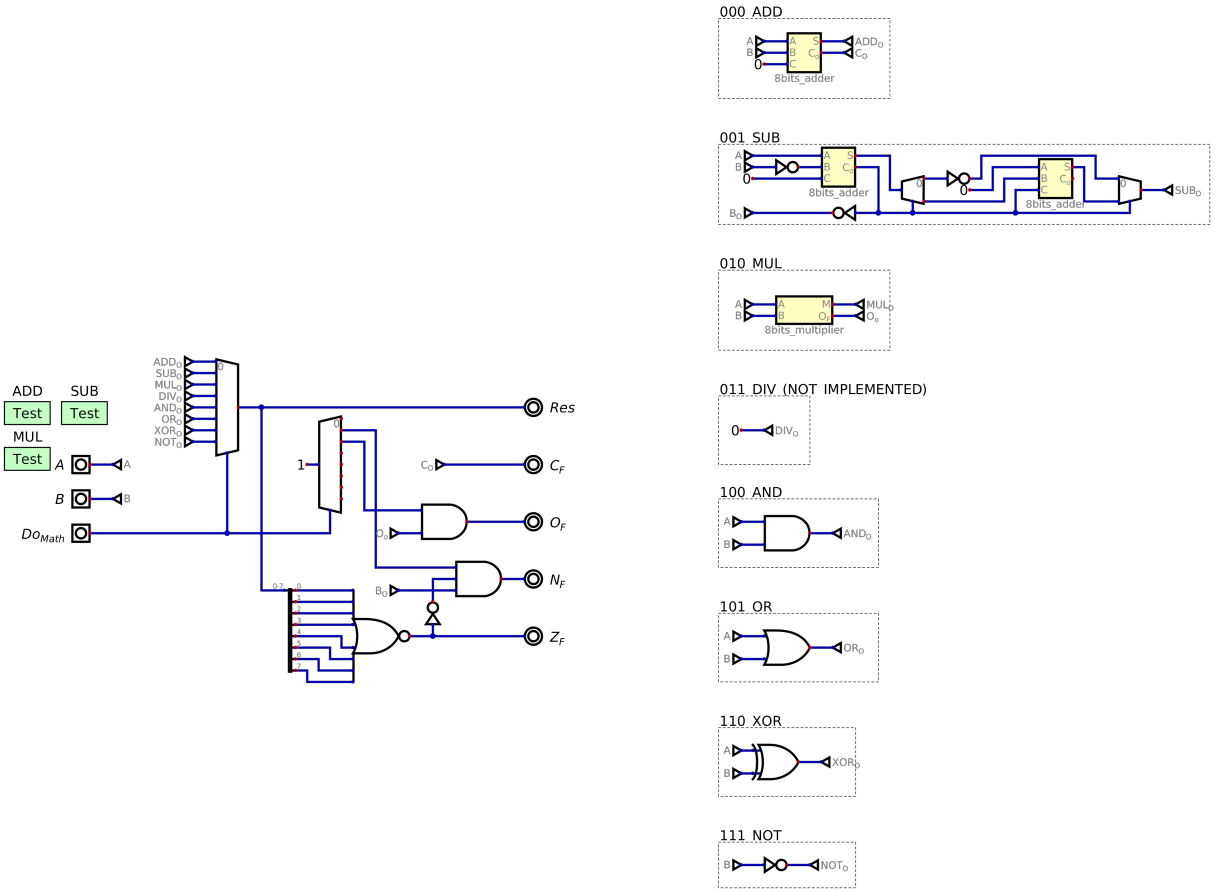




For input and output value for the system and for control the instruction to be done in each pipeline stages via `Program Counter (PC)`.

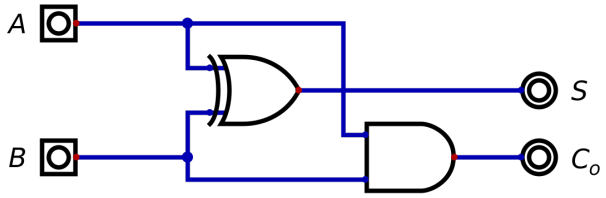
2.1.6 Inner Components Dive Through

ALU



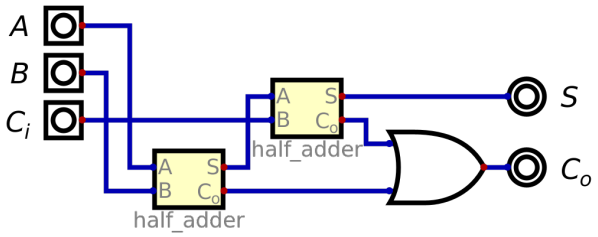
Get the Decision choice value in 3 bits and 2, 8 bits data to calculate. The output result and flags are depended on the calculation. E.g. Using MUL may cause Overflow flag while SUB may cause Negative Flag.

Half Adder



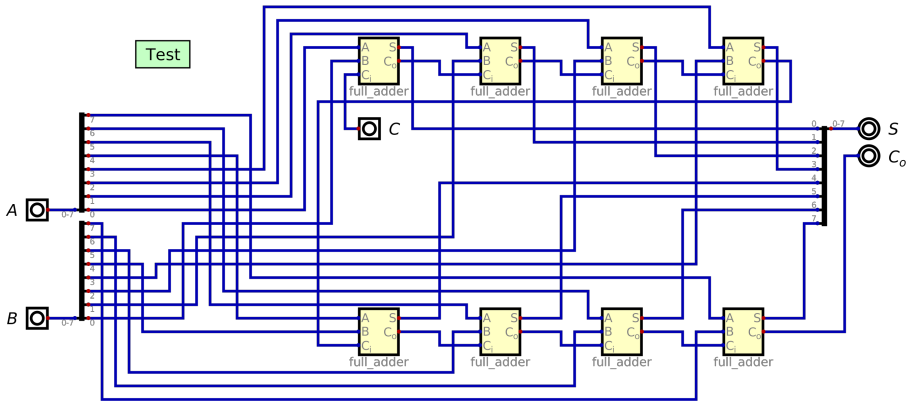
Simplest form of 1 bit adder.

Full Adder



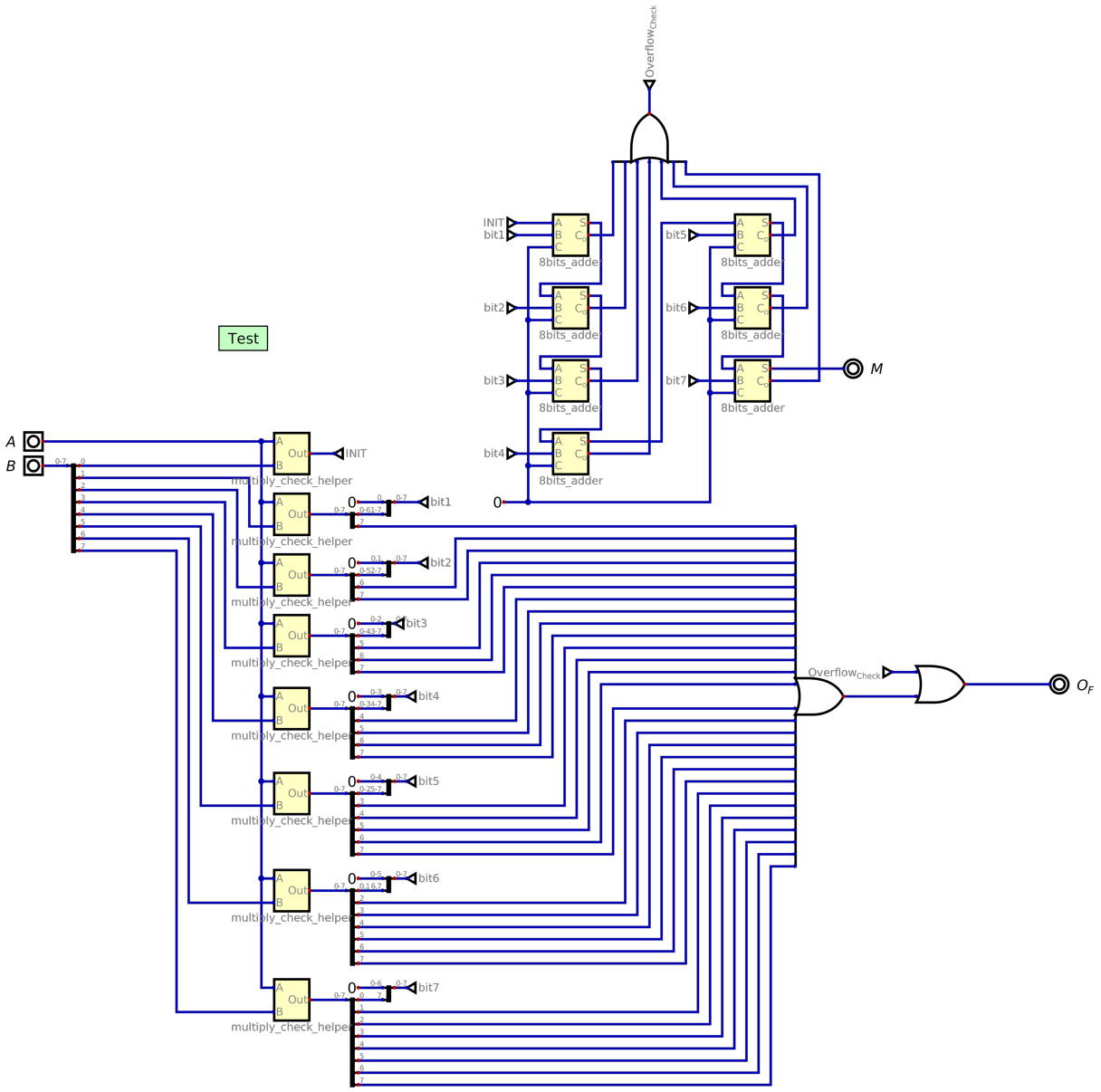
1 bit adder that can have carry input.

8Bit Adder



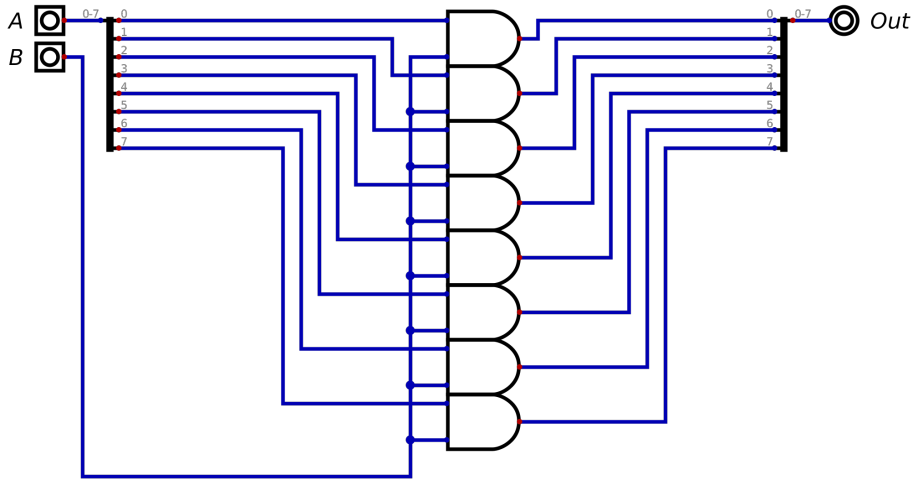
8 Full Adders combine to make system available for 8 bits binary addition.

8Bit Multiplier



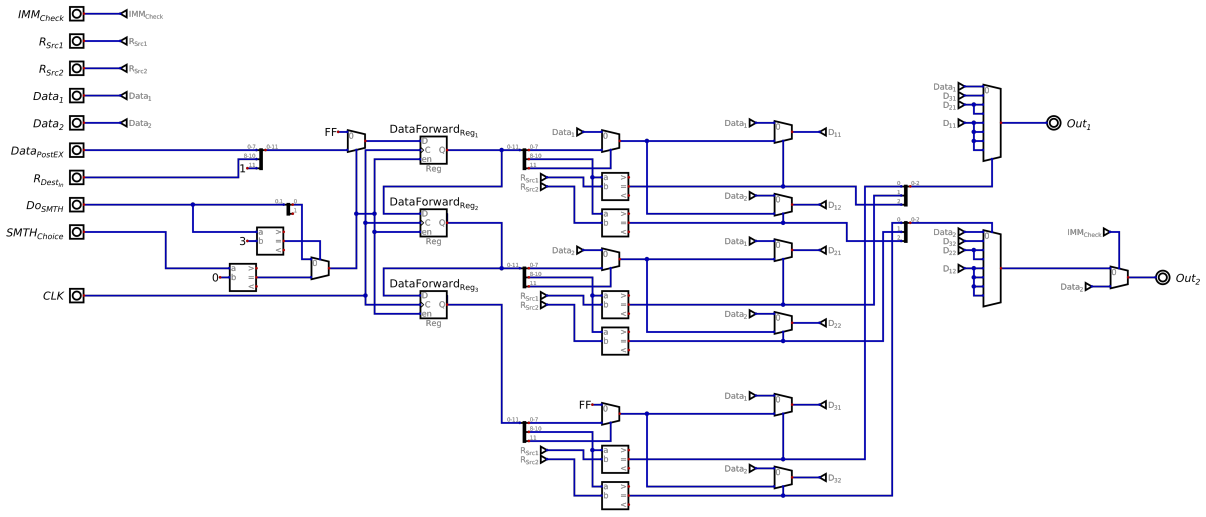
Bit-shifting each line and add up together. If the value exceed 8 bits, it toggles Overflow Flag in the ALU.

Multiplier Helper



Help to set the binary ready before bit-shifting

Data Forwarding Register



Help to prevent Data Hazard if the earlier instructions have the same register used in the later instructions. It can be called from here. Every instruction result executed in EX and its destination register are stored here for a while then if in no used it drops.

Pipeline States

1. **Instruction Fetch** (`IF`): The CPU fetch the instruction line by line from the ROM.
2. **Instruction Decode** (`ID`): The `CPU` decode the instruction line we fetch from `IF` and decode to extract each part of the instruction (e.g. R_{Dest}, R_{Src1}, R_{Src2}, $\#IMM$) to be used in the later stages of pipeline.
3. **Execute** (`EX`): Every instruction which does not use the `RAM` are done here. The data can be forward for the later instructions and can transfer to the later pipeline stages too.
4. **Memory** (`ME`): The data that use `RAM` are done here (e.g. `PSH`, `POP`, `LD`, `ST`). For the instructions with data as output, the output will combine with the R_{Dest} transferred from earlier pipeline stages and transfer to `WB` stage.
5. **Write Back** (`WB`): The data transferred from earlier stages are now ready to be written back to `Register File`.

Pipeline Hazards Management

1. **Structural Hazards**: Fixed my 2 concepts
   (a) Calculation must be done before accessing RAM or WB stage
   (b) Accessing RAM with LD instruction will return value therefore it must be done before WB stage.
2. **Data Hazards**: Fixed by creating `Data Forwarding Register`.
3. **Control Hazards**: If prediction is wrong the system will the flush pipeline with `NOP`

2.2 Software Design

2.2.1 Instructions

Instruction Format (Length: 20 bits):

$$Opcode[19:15] \mid IMM_{Flag}[14] \mid R_{Dest}\ [13:11] \mid R_{Src1} \mid R_{Src2}[2:0]\ or\ IMM[7:0]$$

Instruction	**Opcode + Fields**	**Description**
Do in IF		
NOP	00000 | $IMM_{Flag}[14]$ | $DC[13:0]$	No Operation
Do in EX_{ALU}		
ADD	00100 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $R_{Src1}[10:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Addition
SUB	00101 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $R_{Src1}[10:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Subtraction
MUL	00110 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $R_{Src1}[10:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Multiplication
AND	01000 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $R_{Src1}[10:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Bitwise AND
OR	01001 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $R_{Src1}[10:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Bitwise OR
XOR	01010 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $R_{Src1}[10:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Bitwise XOR
NOT	01011 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $DC[10:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Bitwise NOT
Do in EX_{JUMP}		
BC	10000 | $IMM_{Flag}[14]$ | $DC[13:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Branch if Carry
BZ	10001 | $IMM_{Flag}[14]$ | $DC[13:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Branch if Zero
BNZ	10010 | $IMM_{Flag}[14]$ | $DC[13:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Branch if Not Zero
BNG	10011 | $IMM_{Flag}[14]$ | $DC[13:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Branch if Negative
B	10100 | $IMM_{Flag}[14]$ | $DC[13:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Unconditional Branch
Do in $EX_{I/O}$		
RD	10110 | $DC[14:3]$ | $R_{Src2}[2:0]$	Read input value to register
WR	10111 | $DC[14:3]$ | $R_{Src2}[2:0]$	Write value from register to output
Do in ME_{RAM}		
LD	11000 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $DC[10:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Load value from memory to register
ST	11001 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $DC[10:8]$ | $R_{Src2}[2:0]$ *or* $\#IMM[7:0]$	Store register to memory
Do in ME_{Stack}		
PSH	11010 | $DC[14:3]$ | $R_{Src2}[2:0]$	Push R_{Src2} value into Stack
POP	11011 | $DC[14:3]$ | $R_{Src2}[2:0]$	Pop the top register value from the Stack
Do in WB		
MOV	11110 | $IMM_{Flag}[14]$ | $R_{Dest}[13:11]$ | $R_s rc1[2:0]$ *or* $IMM[7:0]$	Move value fromR_{Src1} or $\#IMM$ into R_{Src1}

2.2.2 Assembler

We had written an assembler in Python.

compiler.py

```
# ---------------- Assembly to 20-bit Hex Converter ----------------

import sys

OPCODES = {
    "NOP": "00000", "ADD": "00100", "SUB": "00101", "MUL": "00110",
    "AND": "01000", "OR":  "01001", "XOR": "01010", "NOT": "01011",
    "BC":  "10000", "BZ":  "10001", "BNZ": "10010", "BNG": "10011",
    "B":   "10100", "RD": "10110", "WR": "10111", "LD":  "11000",
    "ST":  "11001", "PSH":  "11010", "POP":  "11011", "MOV": "11110"
}



def reg_3b(reg: str) -> str:
    """Convert rX to 3-bit binary (r0{r7)."""
    return format(int(reg.replace("r", "")), "03b")



def imm_8b(value: int) -> str:
    """Convert immediate number to 8-bit binary."""
    return format(value & 0xFF, "08b")



def assemble_binary(line: str) -> str | None:
    """Convert assembly line to 19-bit binary string."""

    # -- Remove comments while compiling ------------------------
    if ";" in line:
        line = line.split(';')[0]
    # -----------------------------------------------------------

    # -- Decode Opcode to Binary --------------------------------
    line = line.strip()
    if not line or line.startswith("#"):
        return ""
    parts = line.replace(",", "").split()
    instr = parts[0].upper()
    opcode = OPCODES.get(instr, "????")
    # -----------------------------------------------------------

    # -- Set default binary -------------------------------------
    imm_flag = "0"
    r_dest = "000"
    r_src1 = "000"
    imm_or_r_src2 = "00000000"
    operands = parts[1:]
    # -----------------------------------------------------------

    def get_operand_bits(opnd) -> str:
        """Get operands binary bits"""
        nonlocal imm_flag
        # -- Check if second operand is immediate ------------
        if opnd.startswith("#"):
            # True: Set IsImm flag to 1 ------------------
            imm_flag = "1"
            return imm_8b(int(opnd[1:]))
        else:
            # False: Set IsImm flag to 0 -----------------
            return "00000" + reg_3b(opnd)
        # ------------------------------------------------

    # Assemble binary for each commmand ----------------------------
    if instr == "NOP":
```

```
        return "0000000000000000000"
    elif instr == "PSH" or instr == "WR":
        # -- We found instruction that require only ----------
        # -- a register                                ----------
        imm_flag = "0"
        r_dest = "000"
        r_src1 = "000"
        imm_or_r_src2 = f"00000{reg_3b(operands[0])}"
    elif instr == "BC" or instr == "BZ" or instr == "BNZ" or \
        instr == "BNG" or instr == "B":
        # -- We found instruction that require only ----------
        # --  a register or an immediate            ----------
        imm_flag = "1"
        r_dest = "000"
        r_src1 = "000"
        temp = 0
        if operands[0].startswith('#'):
            temp = int(operands[0][1:])
        imm_or_r_src2 = imm_8b(temp)
    elif instr == "NOT":
        # -- We found NOT -----------------------------------
        r_dest = reg_3b(operands[0])
        r_src1 = "000"
        imm_or_r_src2 = "00000" + reg_3b(operands[1])
    elif instr == "ST":
        r_src1 = reg_3b(operands[0])
        op2 = operands[1]
        if op2.startswith("#"):
            imm_flag = "1"
            imm_or_r_src2 = imm_8b(int(op2[1:]))
        else:
            imm_flag = "0"
    elif instr == "RD" or instr == "POP":
        r_dest = reg_3b(operands[0])
        imm_flag = "0"
        r_src1 = "000"
        imm_or_r_src2 = "00000000"
    elif len(operands) == 2:
        # -- We found instruction which need 2 operands ------
        r_dest = reg_3b(operands[0])
        op2 = operands[1]
        if op2.startswith("#"):
            imm_flag = "1"
            imm_or_r_src2 = imm_8b(int(op2[1:]))
        else:
            imm_flag = "0"
            imm_or_r_src2 = "00000" + reg_3b(op2)
    elif len(operands) == 3:
        # -- We found instruction which need 3 operands ------
        r_dest = reg_3b(operands[0])
        r_src1 = reg_3b(operands[1])
        imm_or_r_src2 = get_operand_bits(operands[2])
    else:
        # -- We found empty line ----------------------------
        return None
    # ------------------------------------------------------------

    return f"{opcode}{imm_flag}{r_dest}{r_src1}{imm_or_r_src2}"


def binary_to_hex(bin_str: str | None) -> str | None:
    """Convert binary string to hex (uppercase, no prefix)."""
    # -- Check if we need to convert to Hex or not ---------------
    if not bin_str:
        # -- If it is a empty line --------------------------
        return None
    # ------------------------------------------------------------
    val = int(bin_str, 2)
    hex_str = format(val, "05X")
    return hex_str


def cli_args_collect() -> list[str]:
```

```
    cli_args = sys.argv
    if len(cli_args) == 1:
        print("Please at least insert a file to convert")
        sys.exit(1)
    elif 2 <= len(cli_args) <= 3:
        input_file = cli_args[1]
        try:
            input_file_part = input_file.split('.')
            try:
                if input_file_part[1] != "ass":
                    print("We need .ass file extension to convert.")
                    sys.exit(1)
            except IndexError:
                print("We need .ass file extension to convert.")
                sys.exit(1)
        except IndexError:
            output_file = f"{input_file}.hex"
        output_file = f"{input_file.split('.')[0]}.hex"

        if len(cli_args) == 3:
            output_file = cli_args[2]
            try:
                _ = output_file.split('.')[1]
            except IndexError:
                output_file = f"{output_file}.hex"
                print("Do not forget to add file extension .hex")
            if output_file.split('.')[1] != "hex":
                output_file = f"{output_file.split('.')[0]}.hex"
                print("Do not forget to change file extension to .hex")
    else:
        print("Too many arguments")
        sys.exit(1)
    return [input_file, output_file]



def main():
    [input_file, output_file] = cli_args_collect()

    with open(input_file, "r") as fin, open(output_file, "w") as fout:
        fout.write("v2.0 raw\n")
        for line in fin:
            hex_val = binary_to_hex(assemble_binary(line))
            if hex_val == None:
                continue
            fout.write(hex_val + "\n")

    print(f"Conversion complete. Hex output saved to '{output_file}'.")



if __name__ == "__main__":
    main()
```

Chapter 3

Installation and Execution Guide

3.1 Prerequisites

- Have git install in your system
- Have Hneemann's Digital installed in your system

3.2 Git Clone from the Remote Repository

```
git clone https://github.com/Pottarr/8Bit-CPU
```

After that you can open the `CPU.dig` in through your Digital.

3.3 Test Assembly Code

3.3.1 Test 1: Overall Instructions

`test1.ass`

```
MOV r0, #3
MOV r1, #3
MOV r2, #3
MOV r3, #3
MOV r4, #3
MOV r5, #3
SUB r0, r0, #1
SUB r1, r1, #1
SUB r2, r2, #1
SUB r3, r3, #1
SUB r4, r4, #1
SUB r5, r5, #1
BNZ #6          ; WILL RUN UNTIL REACHING 0
BC #16          ; WILL NEVER RUN
BZ #15          ; Will skip to the next line
B #16           ; This also skip to the next line
MOV r6, #175
MOV r7, #255
```

3.3.2 Test 2: Data Hazard Check

test2.ass

```
MOV r0, #103    ; 0x67
MOV r1, #67     ; 0x43
MOV r3, #32     ; 0x20
MOV r2, #123    ; 0x7B, Check if r2 WAW with next line
ADD r2, r0, r1  ; 0xAA, Check if r2 WAW with previous line and RAW with next line
ADD r4, r3, r2  ; 0xCA, Check if r2 RAW with next line
SUB r0, r1, r0  ; 0x24
NOT r1, r3      ; 0xDF, Check if r3 WAR with next line
SUB r3, r2, r0  ; 0x86, Check if r3 WAR with previous line
```

3.3.3 Test 3: Structure Hazard and Control Hazard Check

test3.ass

```
MOV r0, #17         ; 0x11 @ 0x05 in WB, Test for BNG and Hazard in EX, ME and WB stages
MOV r1, #47         ; 0x2F @ 0x06 in WB, PS: After @ are value of PC while observing
ADD r2, r1, r0      ; 0x40 @ 0x07 in WB
SUB r3, r1, r0      ; 0x1E @ 0x08 in WB
ST  r3, #100        ; 0x1E @ 0x08 in ME
LD  r1, #100        ; 0x1E @ 0x09 in WB
MUL r4, r1, #5      ; 0xEB @ 0x0B in WB
NOT r5, r4          ; 0x14 @ 0x0C in WB
AND r6, r5, r4      ; 0x00 @ 0x0D in WB (Remain 0x00 as earlier)
SUB r4, r4, #100    ; 0x87 @ 0x00 in WB, IR Addr: 0x09
BNG #12             ; Taken @ 3rd/0x0C, Jump to IR 0x0C
B #9                ; Taken @ 1st/0x0D 2nd/0x0D, Jump to IR 0x09
MOV r7, #255        ; 0xFF @ 0x11, IR Addr: 0x0C
```

3.3.4 Test 4: Stack Warning Check

test4.ass

```
MOV r0, #67     ; 0x43
MOV r1, #5      ; 0x05
MOV r2, #9      ; 0x05
B #8            ; Jump to 0x08 to start PSH Loop
POP r3          ; Stack will Empty @ 4th/___ (Hypothesis)
SUB r2, r2, #1  ; After Stack is empty
BNZ #4          ; It is set pop out value to be the value of 0xFF address in RAM
B #14           ; Jump to 0x0E to finish
PSH r0          ; Stack will start Warning about size @ 4th/___ (Hypothesis)
PSH r1          ; By that time Stack: 0x43, 0x04, 0x45, 0x03, 0x47, 0x02, 0x49, 0x01
ADD r0, r0, #2  ; By that tim Stack: 0x43, 0x04, 0x45, 0x03, 0x47, 0x02, 0x49, 0x01, 0x4B, 0x00
SUB r1, r1, #1  ; But will continue until loops end
BNZ #2          ; Loop continue if NZ is still on
B #4            ; Jump to 0x04 to start POP Loop
PSH r0
POP r4
MOV r7, #255    ; End at PC: 0x0E
```

3.3.5 Test 5: I/O Test

test5.ass

```
MOV r0, #67     ; Game name: Guess 67
MOV r3, #238    ; Set Default Output
MOV r4, #78     ; Set Answer Output
RD r1           ; INPUT READ @ 0x06 Read input
WR r3           ; Default Output: 0x00
SUB r2, r1, r0  ; Use ZF in ALU to compare guess and answer
BZ #8           ; Jump to Answer Reveal
B #3            ; Jump back to Input Guess
WR r4           ; Output Answer: 0x67
```

Chapter 4

Summary

4.1 Project Summary

The *Design and Implementation of a Minimal 8-Bit CPU* project focuses on creating a functional pipelined processor using the *Digital* simulation software. This project integrates theoretical and practical knowledge of computer architecture by designing a five-stage pipelined CPU that executes a custom instruction set architecture (ISA). It emphasizes modular design, digital logic integration, and simulation-based verification of CPU functionalities such as arithmetic operations, branching, and interrupt handling.

4.2 Challenges faced with Solutions

Challenge	**Solution**
CU was too complex using MUX.	Use ROM in CU as signal provider instead.
Manually converting Assembly code into Binary code and insert line by line into IR was difficult.	Create a compiler to compile Assembly code to Hex code and load into the IR.
There were too many tangled cables in the circuit.	Use tunnel component to make the circuit more organize.
RD instruction needed very precise timing set input a value.	Make the input receive the input value available in the input port in that moment of execution. (Therefore, users can pre-input the value and wait for the instruction to execute.)
We fetch wrong data when we want to fetch data before editing.	Try falling edge signal.

4.3 Learning Outcomes

- Successfully implemented a fully functional 8-bit pipelined CPU with distinct stages: Fetch, Decode, Execute, Memory, and Write Back.
- Developed a custom instruction set architecture (ISA) supporting arithmetic, logic, branching, and I/O operations.
- Designed and simulated essential CPU modules including the ALU, control unit, registers, and memory units (ROM and RAM).
- Implemented flag registers (Zero, Non-Zero, Carry) to support conditional branching and status tracking.
- Demonstrated interrupt handling and basic I/O communication using input and output ports.
- Verified CPU functionality using test programs, simulation waveforms, and step-by-step execution tracing.

4.4 Accomplishments

- Gained hands-on experience in digital logic and CPU design principles through simulation.
- Applied theoretical knowledge of pipelining and instruction execution to a practical, working model.
- Improved understanding of hardware design challenges such as data hazards and control flow management.
- Strengthened teamwork, modular circuit design, and documentation skills.
- Completed a comprehensive report and video demonstration showcasing CPU functionality and performance.

4.5 Further Development

We have planned to implement the SWI (Software Interrupt) into our CPU. But due to project time period was not enough, therefore we might implement it in the future.

Chapter 5

References

- Digital Design and Computer Architecture ARM Edition
- How to build a computer TUTORIAL

Chapter 6

Appendix

6.1 Github Repositories

- This project repository: `https://github.com/Pottarr/8Bit-CPU`
- Helmut Neemann's Digital repository: `https://github.com/hneemann/Digital`