



# 8 Bits CPU Simulation Documentation

**Computer Architecture and Organization**

**Software Engineering Program,**

**Department of Computer Engineering,**

**School of Engineering, KMITL**

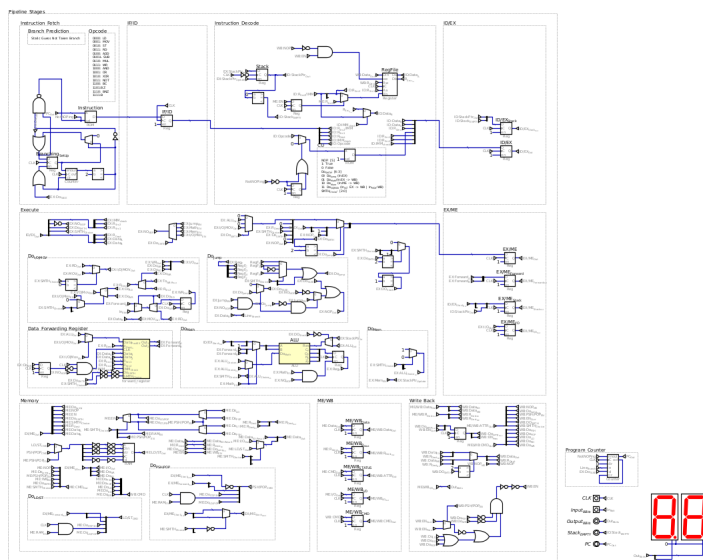
67011093 Chavit Saritdeechaikul

67011352 Theepakorn Phayonrat

# Project Overview

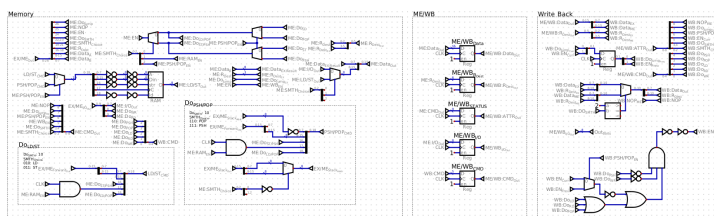
## Hardware Design

### Overall Structure



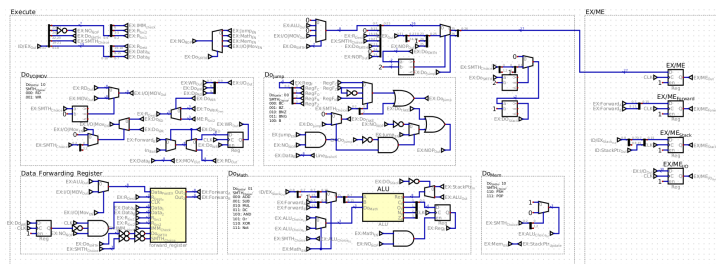
### Structure Dive Through

#### Upper Part



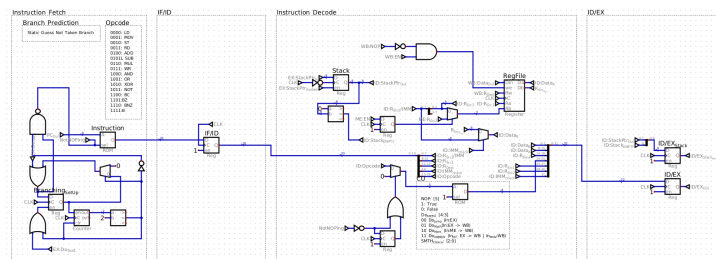
Consist of Instruction Fetch stage and Instruction Decode stage.

## Center Part



Consist of the whole Execute stage.

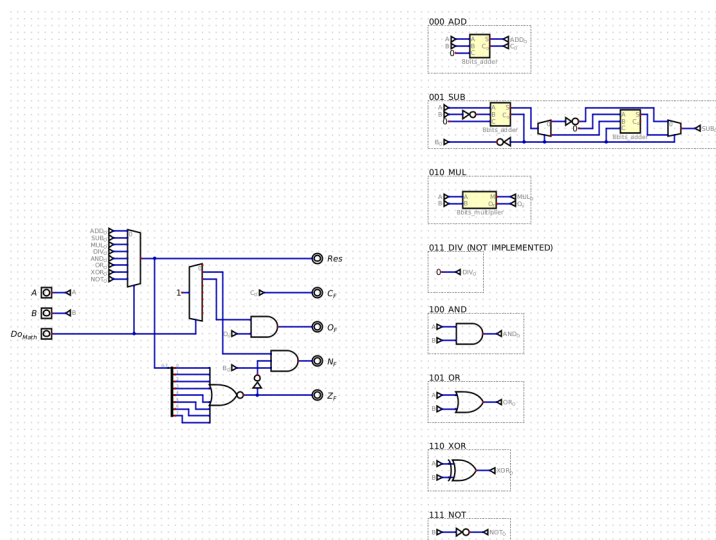
## Lower Part



Consist of Memory stage and Write Back stage.

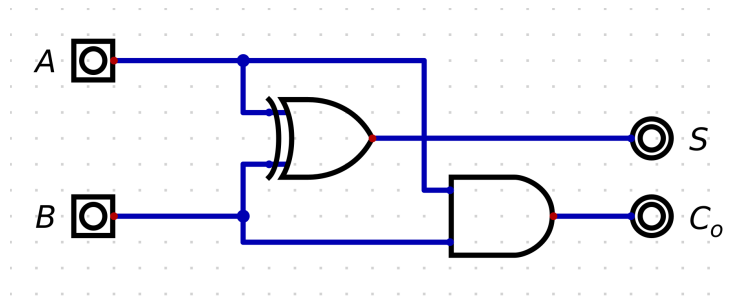
## Components Dive Through

### ALU



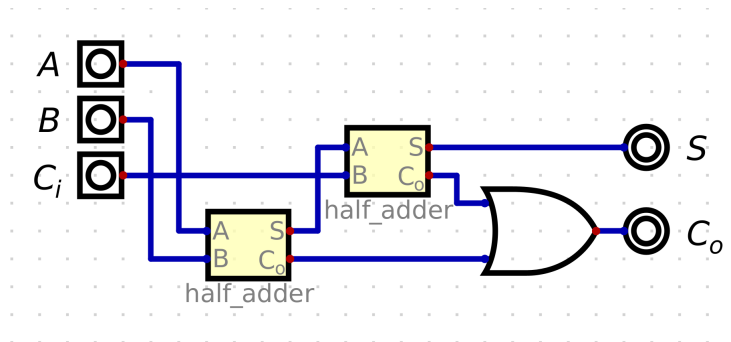
Get the Decision choice value in 3 bits and 2, 8 bits data to calculate. The output result and flags are depended on the calculation. E.g. Using MUL may cause Overflow flag while SUB may cause Negative Flag.

### Half Adder



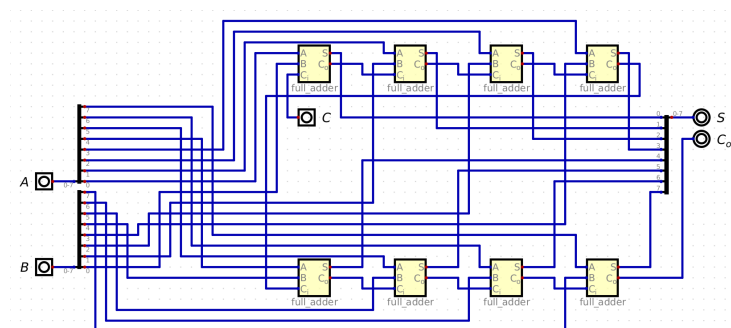
Simplest form of 1 bit adder.

### Full Adder



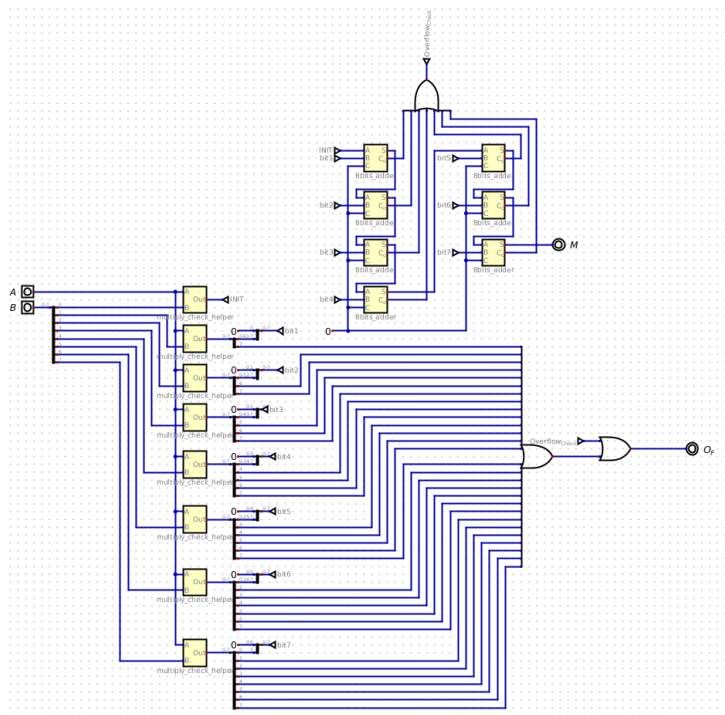
1 bit adder that can have carry input.

### 8Bit Adder



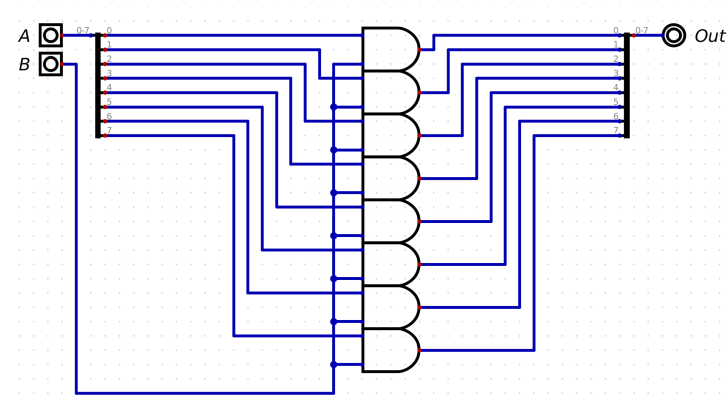
8 Full Adders combine to make system available for 8 bits binary addition.

## 8Bit Multiplier



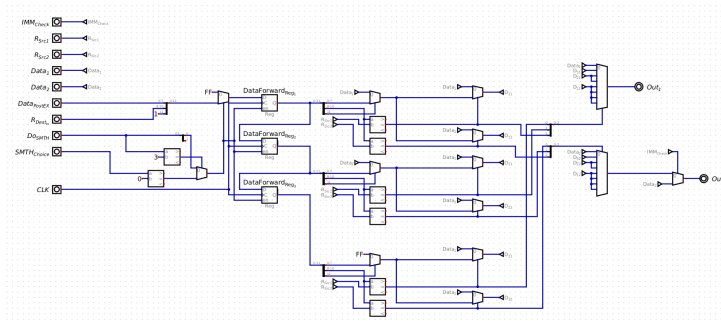
Bit-shifting each line and add up together. If the value exceed 8 bits, it toggles Overflow Flag in the ALU.

## 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

# Software Design

## Instructions

Instruction Format (Length: 20 bits):

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

Instruction	Opcode + Fields	Description
Do in $IF$ <i>NOP</i>	00000   $IMM_{Flag}[14]$   $DC[13 : 0]$	No Operation
Do in $EX_{ALU}$ <i>ADD</i>	00100   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $R_{Src1}[10 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Addition
<i>SUB</i>	00101   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $R_{Src1}[10 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Subtraction
<i>MUL</i>	00110   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $R_{Src1}[10 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Multiplication
<i>AND</i>	01000   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $R_{Src1}[10 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Bitwise AND
<i>OR</i>	01001   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $R_{Src1}[10 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Bitwise OR
<i>XOR</i>	01010   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $R_{Src1}[10 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Bitwise XOR
<i>NOT</i>	01011   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $DC[10 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Bitwise NOT
Do in $EX_{JUMP}$ <i>BC</i>	10000   $IMM_{Flag}[14]$   $DC[13 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Branch if Carry
<i>BZ</i>	10001   $IMM_{Flag}[14]$   $DC[13 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Branch if Zero
<i>BNZ</i>	10010   $IMM_{Flag}[14]$   $DC[13 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Branch if Not Zero
<i>BNG</i>	10011   $IMM_{Flag}[14]$   $DC[13 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Branch if Negative
<i>B</i>	10100   $IMM_{Flag}[14]$   $DC[13 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Unconditional Branch
Do in $ME_{Stack}$ <i>PSH</i>	10110   $DC[14 : 3]$   $R_{Src2}[2 : 0]$	Push $R_{Src2}$ value into Stack
<i>POP</i>	10111   $DC[14 : 3]$   $R_{Src2}[2 : 0]$	Pop the top register value from the Stack
Do in $ME_{I/O}$ <i>RD</i>	11000   $DC[14 : 3]$   $R_{Src2}[2 : 0]$	Read input value to register
<i>WR</i>	11001   $DC[14 : 3]$   $R_{Src2}[2 : 0]$	Write value from register to output
Do in $ME_{RAM}$ <i>LD</i>	11100   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $DC[10 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Load value from memory to register
<i>ST</i>	11101   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $DC[10 : 8]$   $R_{Src2}[2 : 0]$ or $\#IMM[7 : 0]$	Store register to memory
Do in $WB$ <i>MOV</i>	11110   $IMM_{Flag}[14]$   $R_{Dest}[13 : 11]$   $R_{Src1}[2 : 0]$ or $IMM[7 : 0]$	Move value from $R_{Src1}$ or $\#IMM$ into $R_{Src1}$

# Assembler

We had written an assembler in Python.

## Test Assembly Code

### 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
```

### 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
```

### Test 3: Structural 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
```

## 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
```

## 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
```

# Summary

## 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.

## 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.