INSTRUCTION SET ARCHITECTURE(ISA)

THANATHAS CHAWENGVORAKUL 6014586

WHAT IS ISA?

An instruction set architecture (ISA) is an abstract model of a computer. It is also referred to as architecture or computer architecture. A realization of an ISA is called an implementation. An ISA permits multiple implementations that may vary in performance, physical size, and monetary cost (among other things); because the ISA serves as the interface between software and hardware. Software that has been written for an ISA can run on different implementations of the same ISA. This has enabled binary compatibility between different generations of computers to be easily achieved, and the development of computer families. Both of these developments have helped to lower the cost of computers and to increase their applicability. For these reasons, the ISA is one of the most important abstractions in computing today.

```
$000          rjmp    RESET           ; Reset Handler
$001          rjmp    EXT_INT0        ; IRQ0 Handler
$002          rjmp    EXT_INT1        ; IRQ1 Handler
$003          rjmp    TIM2_COMP       ; Timer2 Compare Handler
$004          rjmp    TIM2_OVF        ; Timer2 Overflow Handler
$005          rjmp    TIM1_CAPT       ; Timer1 Capture Handler
$006          rjmp    TIM1_COMPA      ; Timer1 CompareA Handler
$007          rjmp    TIM1_COMPB      ; Timer1 CompareB Handler
$008          rjmp    TIM1_OVF        ; Timer1 Overflow Handler
$009          rjmp    TIM0_OVF        ; Timer0 Overflow Handler
$00a          rjmp    SPI_STC         ; SPI Transfer Complete Handler
$00b          rjmp    USART_RXC       ; USART RX Complete Handler
$00c          rjmp    USART_UDRE      ; UDR Empty Handler
$00d          rjmp    USART_TXC       ; USART TX Complete Handler
$00e          rjmp    ADC             ; ADC Conversion Complete Handler
$00f          rjmp    EE_RDY          ; EEPROM Ready Handler
$010          rjmp    ANA_COMP        ; Analog Comparator Handler
$011          rjmp    TWSI            ; Two-wire Serial Interface
Handler
$012          rjmp    SPM_RDY         ; Store Program Memory Ready
Handler
```

An ISA defines everything a machine language programmer needs to know in order to program a computer. What an ISA defines differs between ISAs; in general, ISAs define the supported data types, what state there is (such as the main memory and registers) and their semantics (such as the memory consistency and addressing modes), the instruction set (the set of machine instructions that comprises a computer's machine language), and the input/output model.

ADDRESSING OF OPERAND AND OPERATION

To perform instruction user need to enter 3 operands.

1). Basic Operations (mov, add, sub, mul, div)
2). Destination Register (r0-r7)
3). Date (can be either registers or decimal number)

```
Operand : r0-r7
Operation : mov add sub mul div
end 0 0 to quit the program
mov r0 5
add r0 8
mul r0 2
div r0 2
mov r1 r0
add r5 r1
end 0 0
```

```
val identifiedIndex = hashMapOf("r0" to 0, "r1" to 1, "r2" to 2, "r3" to 3, "r4" to 4, "r5" to 5
        , "r6" to 6, "r7" to 7, "mov" to 0, "add" to 1, "sub" to 2, "mul" to 3, "div" to 4)
val operand = arrayOf(0, 0, 0, 0, 0, 0, 0, 0)
val address = arrayOf("000", "001", "010", "011", "100", "101", "110", "111")
```

I store both register operands and operations as index of address array, so when I access register and operation. I can know both where is data location in binary and in array.

```
[PROGRAM STEP]
=============================================================
PC          DECODE          ENCODE
PC[0]       mov r0 5        000 000 000000000000000000000101
PC[1]       add r0 8        001 000 000000000000000000001000
PC[2]       mul r0 2        011 000 000000000000000000000010
PC[3]       div r0 2        100 000 000000000000000000000010
PC[4]       mov r1 r0       000 001 000000000000000000000000
PC[5]       add r5 r1       001 101 000000000000000000000001

[REGISTER VALUE]
================================
r0 = 000000000000000000001101
r1 = 000000000000000000001101
r2 = 000000000000000000000000
r3 = 000000000000000000000000
r4 = 000000000000000000000000
r5 = 000000000000000000001101
r6 = 000000000000000000000000
r7 = 000000000000000000000000

CPI = 2.6666666666666665
```

After user end the program, user will be able to see following information

1). Program Step
   - Program Counter
   - Decode
   - Encode
2). Final value of all register in 24-bits
3). CPI

4). PipeLine Model

5). Detect Hazard

```
[PIPELINE]
=============================================================

                  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 |
mov r0 5         IF | ID | EX | WB |
add r0 8            IF | ID | EX | WB |
mul r0 2               IF | ID | EX | WB |
div r0 2                  IF | ID | EX | WB |
mov r1 r0                    IF | ID | EX | WB |
add r5 r1                       IF | ID | EX | WB |

r0 in instruction 4 and 5 caused RAW hazard and is solved by forwarding.
r1 in instruction 5 and 6 caused RAW hazard and is solved by forwarding.
```

```
Thanathas Chawengvorakul 6014586
*/
val identifiedIndex = hashMapOf("r0" to 0, "r1" to 1, "r2" to 2, "r3" to 3, "r4" to 4, "r5" to 5
        , "r6" to 6, "r7" to 7, "mov" to 0, "add" to 1, "sub" to 2, "mul" to 3, "div" to 4)
val operand = arrayOf(0, 0, 0, 0, 0, 0, 0, 0)
val address = arrayOf("000", "001", "010", "011", "100", "101", "110", "111")
var clockSum = 0.0
var inputSum = 0.0
val decodeHistory = arrayListOf<String>()

fun main(args: Array<String>) {
    println("Operand : r0-r7\nOperation : mov add sub mul div\nend 0 0 to quit the program")

    do {
        val input = readLine()!!
        val split = input.split( ...delimiters: " ")
        var setIndex: Int

        if(split[0] != "end") {
            decodeHistory.add(input)
        }
```

```
try {
    setIndex = identifiedIndex[split[1]]!!.toInt()
} catch (i: NullPointerException) {
    setIndex = 0
}

val getValue = if (identifiedIndex.containsKey(split[2])) operand[identifiedIndex[split[2]]!!.toInt()] else split[2].toInt()

when (split[0]) {
    "mov" -> {
        operand[setIndex] = getValue
        clockSum += 1
        inputSum++
    }
    "add" -> {
        operand[setIndex] += getValue
        clockSum += 3
        inputSum++
    }
    "sub" -> {
        operand[setIndex] -= getValue
        clockSum += 3
        inputSum++
    }
    "mul" -> {
        operand[setIndex] *= getValue
        clockSum += 4
        inputSum++
    }
```

```
            "div" -> {
                operand[setIndex] /= getValue
                clockSum += 4
                inputSum++
            }
        }
    } while (split[0] != "end")

    println("\n[PROGRAM STEP]\n==========================================================")

    println("%-10s".format( ...args: "PC") + "%-15s".format( ...args: "ENCODE") + "DECODE")

    for (i in 0..decodeHistory.size - 1) {
        println("%-10s".format( ...args: "PC[$i]") + "%-15s".format(decodeHistory[i]) + encode(decodeHistory[i]))
    }

    println("\n[REGISTER VALUE]\n================================")

    for (j in 0..operand.size - 1) {
        println("r$j = ${to24Binary(operand[j])}")
    }

    println("\nCPI = ${(clockSum / inputSum)}")
}
```

```
fun to24Binary(dec: Int): String {
    if (dec < 0) {
        return Integer.toBinaryString(dec).substring( startIndex: 8)
    } else {
        return "%024d".format(Integer.toBinaryString(dec).toInt())
    }
}

fun printPipeLine(decode: ArrayList<String>){

    val step = arrayOf("IF","ID","EX","WB")
    print("%-15s".format( ...args: ""))
    for (a in 1..decode.size+3){
        print("%-5s".format( ...args: " $a |"))
    }
    println()
    for(i in 1..decode.size){
        print("%-15s".format(decode[i-1]))

        for (i in 0..i-2){
            print("%-5s".format( ...args: ""))
        }

        for (j in 0..3){
            print("%-5s".format( ...args: "${step[j]} |"))
        }
        println()
    }
    println()

    for (b in 0..decode.size-2){
        val first = decode[b].split( ...delimiters: " ")[1]
        val second = decode[b+1].split( ...delimiters: " ")[2]
        if (first == second){
            println("$first in instruction ${b+1} and ${b+2} caused RAW hazard and is solved by forwarding.")
        }
    }
}
```