ASSUMPTION UNIVERSITY

FACULTY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

SC 6231 COMPUTER ARCHITECTURE

TERM PROJECT

Submitted To:	Dr. Anilkumar K.G.
Submitted By:	Avanish Shrestha (5919364)

Instruction Set Architecture

What is an ISA?

An Instruction Set, or Instruction Set Architecture, is a part of the computer architecture related to computer programming, including native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification for the set of opcodes (machine language), and the native commands implemented by a particular processor.

Instruction set architecture is distinguished from the micro architecture, which is the set of processor design techniques used to implement the instruction set. Computers with different micro architectures can share a common instruction set. For example, the Intel Pentium and the AMD Athlon implement nearly the same identical version of the x86 instruction set, but have radically different internal designs.

Some virtual machines that support byte code for Smalltalk, the Java virtual machine, and Microsoft's Common Language Runtime virtual machine as their ISA implement it by translating the byte code for commonly used code paths into native machine code, and executing less-frequently-used code paths by interpretation.

Instruction Set Architecture of a machine fills the semantic gap between the user and the machine. It is a way to talk to the computer.

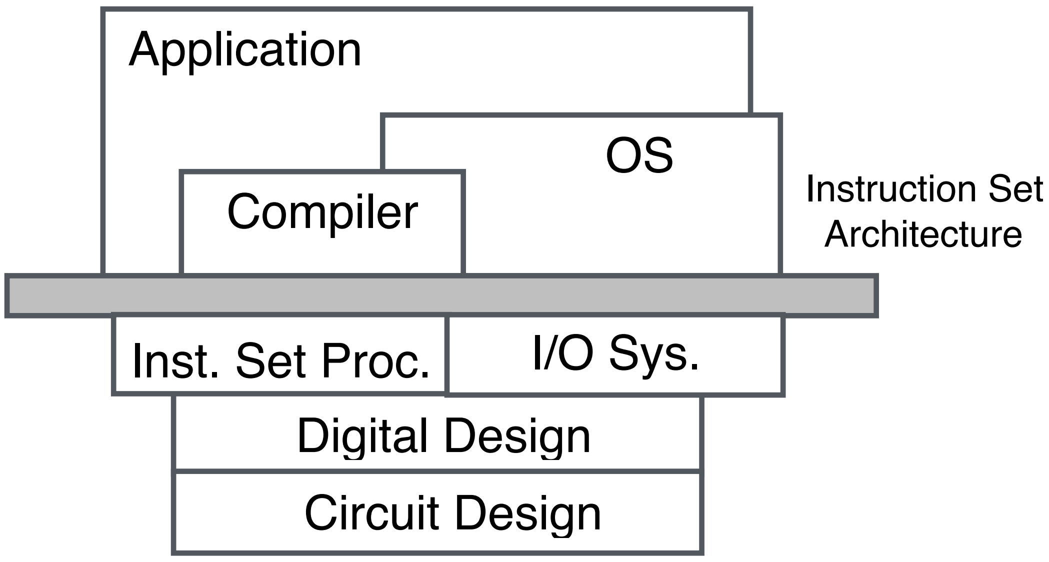


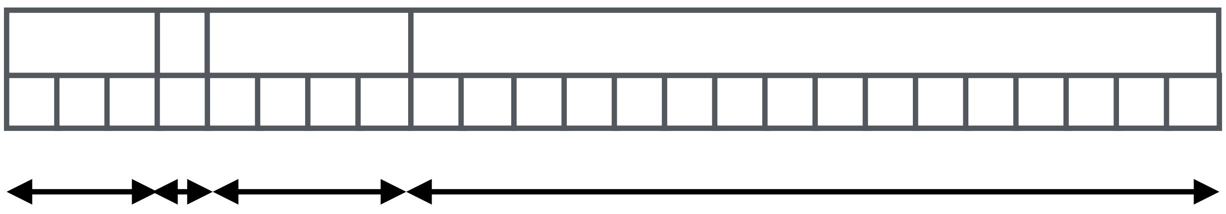
Figure 1. Instruction Set Architecture

ISA Design

The designed Instruction Set Architecture is a 24-bit architecture. It is able to handle any 16-bit architecture operations.

The ISA uses sixteen 16-bit General Purpose Registers (R0 - R15), and two additional registers; a 32-bit register 'RM' to store the result of multiplication operation and a 16-bit register 'RE' to store the remainder after division operation.

<u>24-bit Instruction</u>



03-bit	-	*Opcode*
01-bit	-	*Control bit*
04-bit	-	*Operand 1*
16-bit	-	*Operand 2*

The ISA is designed using custom C++ structures for instructions, binary instructions, as well as registers. All the custom structures are stored in vectors. Therefore, the ISA can handle any number of instructions. Since the registers are 16-bit, it can only hold 16-bit integers (-32767 to 32767).

```
struct Register {
    int index;              // 0 - 7
    string name;            // R0 - R7
    int value;              // Initialized with 0
};

struct Instruction {
    string opcode;          // [MOV, ADD, SUB, MUL, DIV]
    string first_operand;   // [R0 - R7]
    string second_operand;  // [R0 - R7] || Integer value (-32767) to (+32767)
    int clock_cycle;        // [MOV = 1, ADD = 2, SUB = 2, MUL = 3, DIV = 5]
};

struct BinaryInst {
    string opcode;          // 04-bit: opcode; 2^4 = 16 Opcodes
    string optype;          // 01-bit: 2nd operand type; 0:register; 1: value;
    string first_operand;   // 03-bit: binary first operand ( 2^3 = 8 Registers)
    string value;           // 16-bit: value (-32767) to (+32767)
};
```

Opcode and its Meaning

Opcode	Control Bit	Operation	Details
000	0	MOV	Register to Register
	1		Mem/Value to Register
001	0	ADD	Register to Register
	1		Value to Register
010	0	SUB	Register to Register
	1		Value to Register
011	0	MUL	Register to Register
	1		Value to Register
100	0	DIV	Register to Register
	1		Value to Register

1. **MOV** - This opcode is used to move an integer value, or a value stored in another register, to a register. The opcode is also used for loading the value from a memory location, indicated by [R0].

    Example,

    MOV R0 R1 (moves the value stored in R1 to register R0)

    MOV R0 10 (moves 10 to R0)

    MOV R1 [R0] (moves the value in memory location denoted by R0)

2. **ADD** - This opcode is used to add two values stored in two registers, or a value stored in a register with an integer value. It will store the result in first register of the instruction.

    Example,

    ADD R0 R1

    ADD R0 10

3. **SUB** - This opcode is used to subtract two values stored in two registers, or an integer value from a value stored in the register.

    Example,

    SUB R0 R1

    SUB R0 10

4. **MUL** - This opcode is used to multiply two values stored in two registers, or a value stored in a register with an integer value. The result of this instruction will be stored in a 32-bit register 'RM'.

    Example,

    MUL R2 R3

    MUL R2 10

5. **<u>DIV</u>** - This opcode is used to divide two values stored in two registers, or an integer value from a value stored in the register. The remainder of this instruction will be stored in a 16-bit register 'RE'.

    Example,

    DIV R5 R1

    DIV R6 2

6. **<u>END</u>** - This opcode is used to end the program. To use this opcode, you only need to type *'END'* without the registers or integer value.

Steps for using the ISA

1. Select one of the six opcodes available *[MOV, ADD, SUB, MUL, DIV, END]*.

2. Select the first operand. The first operand must always be one of the available registers *(R0 - R15)*.

3. Select the second operand. The second operand can either be a register or an integer value. Second operand can also be a memory location, denoted by [R0].

<u>Note</u>: The ISA is **<u>not</u>** case-sensitive in the case of opcode as well as operands, i.e. it can accept the opcodes & operands in either lowercase or uppercase.

Example,

mov R0 10

add r1 R0

end

<u>Clock Per Instruction (CPI)</u>

CPI = No. of CPU clock cycles / Instruction Count (IC)

The ISA will calculate the CPI based on the following clock cycles for each opcode.

MOV	= 1 clock cycle	**ADD**	= 2 clock cycles	**SUB**	= 3 clock cycles
MUL	= 4 clock cycles	**DIV**	= 5 clock cycles		

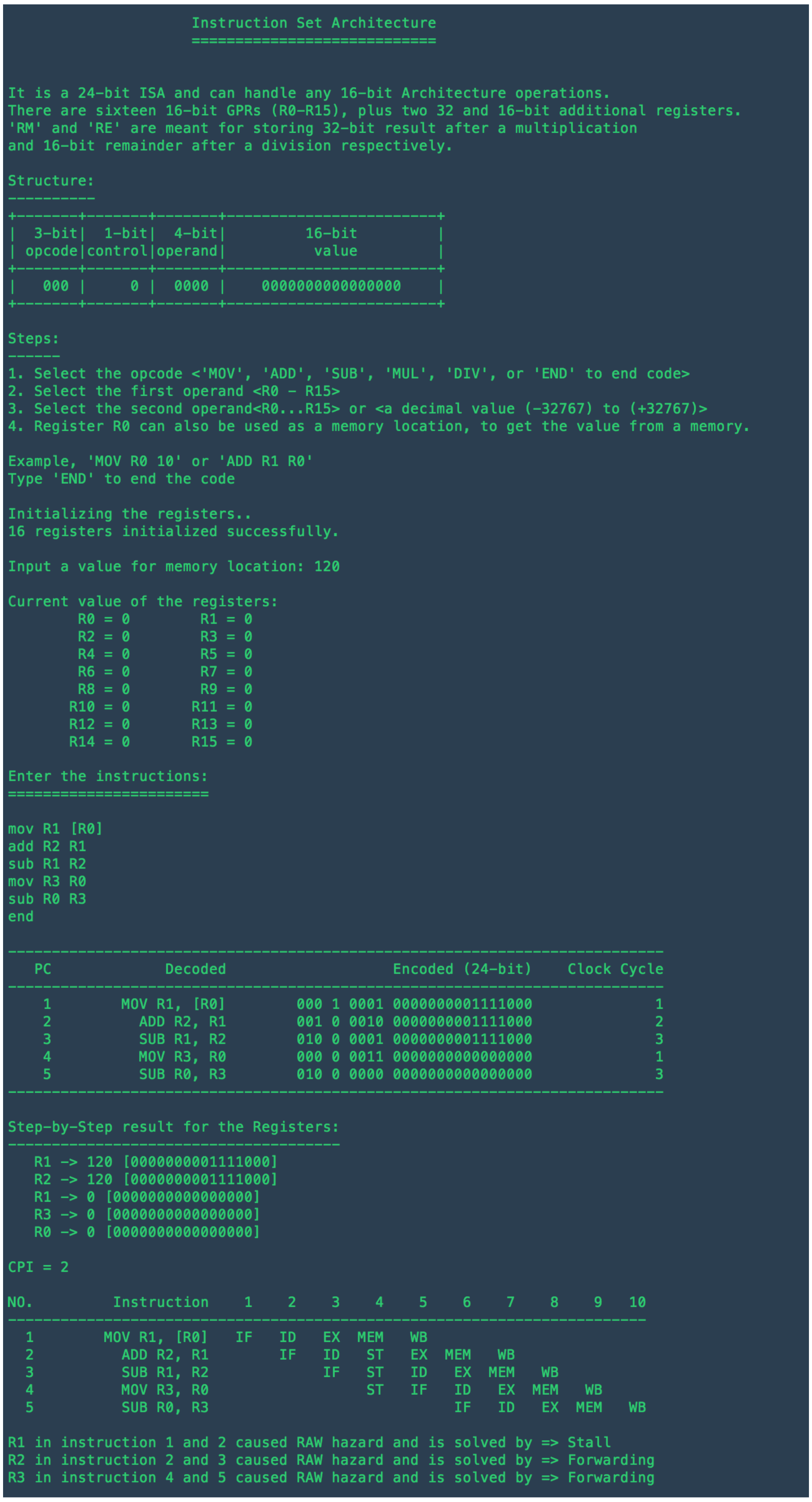
Pipeline

The application will also able to simulate a 5-stage pipelined execution of the instructions.

IF	- Instruction Fetch	**ID**	- Instruction Decode	**EX**	- Execution
MEM	- Memory	**WB**	- Write Back		

The ISA is able to detect RAW data hazard, and solve it using *forwarding* or *stall* (**ST**).

Output



```
                     Instruction Set Architecture
                     ============================


It is a 24-bit ISA and can handle any 16-bit Architecture operations.
There are sixteen 16-bit GPRs (R0-R15), plus two 32 and 16-bit additional registers.
'RM' and 'RE' are meant for storing 32-bit result after a multiplication
and 16-bit remainder after a division respectively.

Structure:
----------
+-------+-------+-------+------------------------+
|  3-bit|  1-bit|  4-bit|         16-bit         |
| opcode|control|operand|          value         |
+-------+-------+-------+------------------------+
|   000 |     0 |  0000 |    0000000000000000    |
+-------+-------+-------+------------------------+

Steps:
------
1. Select the opcode <'MOV', 'ADD', 'SUB', 'MUL', 'DIV', or 'END' to end code>
2. Select the first operand <R0 - R15>
3. Select the second operand<R0...R15> or <a decimal value (-32767) to (+32767)>
4. Register R0 can also be used as a memory location, to get the value from a memory.

Example, 'MOV R0 10' or 'ADD R1 R0'
Type 'END' to end the code

Initializing the registers..
16 registers initialized successfully.

Input a value for memory location: 120

Current value of the registers:
        R0 = 0        R1 = 0
        R2 = 0        R3 = 0
        R4 = 0        R5 = 0
        R6 = 0        R7 = 0
        R8 = 0        R9 = 0
       R10 = 0       R11 = 0
       R12 = 0       R13 = 0
       R14 = 0       R15 = 0

Enter the instructions:
=======================

mov R1 [R0]
add R2 R1
sub R1 R2
mov R3 R0
sub R0 R3
end

---------------------------------------------------------------------------
   PC             Decoded                   Encoded (24-bit)    Clock Cycle
---------------------------------------------------------------------------
    1        MOV R1, [R0]        000 1 0001 0000000001111000              1
    2          ADD R2, R1        001 0 0010 0000000001111000              2
    3          SUB R1, R2        010 0 0001 0000000001111000              3
    4          MOV R3, R0        000 0 0011 0000000000000000              1
    5          SUB R0, R3        010 0 0000 0000000000000000              3
---------------------------------------------------------------------------

Step-by-Step result for the Registers:
--------------------------------------
   R1 -> 120 [0000000001111000]
   R2 -> 120 [0000000001111000]
   R1 -> 0 [0000000000000000]
   R3 -> 0 [0000000000000000]
   R0 -> 0 [0000000000000000]

CPI = 2

NO.         Instruction    1    2    3    4    5    6    7    8    9   10
--------------------------------------------------------------------------
  1        MOV R1, [R0]   IF   ID   EX  MEM   WB
  2          ADD R2, R1        IF   ID   ST   EX  MEM   WB
  3          SUB R1, R2             IF   ST   ID   EX  MEM   WB
  4          MOV R3, R0                  ST   IF   ID   EX  MEM   WB
  5          SUB R0, R3                            IF   ID   EX  MEM   WB

R1 in instruction 1 and 2 caused RAW hazard and is solved by => Stall
R2 in instruction 2 and 3 caused RAW hazard and is solved by => Forwarding
R3 in instruction 4 and 5 caused RAW hazard and is solved by => Forwarding
```