

PROJECT

ISA description:

Consider a 16 bit ISA with the following instructions and opcodes, along with the syntax of an assembly language which supports this ISA.

The ISA has 6 encoding types of instructions. The description of the types is given later.

Opcode	Instruction	Semantics	Syntax	Type
00000	Addition	Performs $reg1 = reg2 + reg3$. If the computation overflows, then the overflow flag is set	<code>add reg1 reg2 reg3</code>	A
00001	Subtraction	Performs $reg1 = reg2 - reg3$. In case $reg3 > reg2$, 0 is written to $reg1$ and overflow flag is set.	<code>sub reg1 reg2 reg3</code>	A
00010	Move Immediate	Performs $reg1 = \$Imm$ where Imm is a 8 bit value.	<code>mov reg1 \$Imm</code>	B
00011	Move Register	Performs $reg1 = reg2$.	<code>mov reg1 reg2</code>	C
00100	Load	Loads data from <code>mem_addr</code> into $reg1$.	<code>ld reg1 mem_addr</code>	D
00101	Store	Stores data from $reg1$ to <code>mem_addr</code> .	<code>st reg1 mem_addr</code>	D
00110	Multiply	Performs $reg1 = reg2 \times reg3$. If the computation overflows, then the overflow flag is set.	<code>mul reg1 reg2 reg3</code>	A
00111	Divide	Performs $reg3/reg4$. Stores the quotient in $R0$ and the remainder in $R1$.	<code>div reg3 reg4</code>	C

01000	Right Shift	Right shifts reg1 by \$Imm, where \$Imm is an 8 bit value.	rs reg1 \$Imm	B
01001	Left Shift	Left shifts reg1 by \$Imm, where \$Imm is an 8 bit value.	ls reg1 \$Imm	B
01010	Exclusive OR	Performs bitwise XOR of reg2 and reg3. Stores the result in reg1.	xor reg1 reg2 reg3	A
01011	Or	Performs bitwise OR of reg2 and reg3. Stores the result in reg1.	or reg1 reg2 reg3	A
01100	And	Performs bitwise AND of reg2 and reg3. Stores the result in reg1.	and reg1 reg2 reg3	A
01101	Invert	Performs bitwise NOT of reg2. Stores the result in reg1.	not reg1 reg2	C
01110	Compare	Compares reg1 and reg2 and sets up the FLAGS register.	cmp reg1 reg2	C
01111	Unconditional Jump	Jumps to mem_addr, where mem_addr is a memory address.	jmp mem_addr	E
10000	Jump If Less Than	Jump to mem_addr if the less than flag is set (less than flag = 1), where mem_addr is a memory address.	jlt mem_addr	E

10001	Jump If Greater Than	Jump to mem_addr if the greater than flag is set (greater than flag = 1), where mem_addr is a memory address.	jgt mem_addr	E
10010	Jump If Equal	Jump to mem_addr if the equal flag is set (equal flag = 1), where mem_addr is a memory address.	je mem_addr	E
10011	Halt	Stops the machine from executing until reset	hlt	F

where reg(x) denotes register, mem_addr is a memory address (must be an 8-bit binary number), and Imm denotes a constant value (must be an 8-bit binary number). The ISA has 7 general purpose registers and 1 flag register. The ISA supports an address size of **8 bits**, which is **double byte addressable**. Therefore, each address fetches two bytes of data. This results in a total address space of 512 bytes. **This ISA only supports whole number arithmetic**. If the subtraction results in a negative number; for example “3 - 4”, the reg value will be set to 0 and overflow bit will be set. All the representations of the number are hence unsigned.

The registers in assembly are named as R0, R1, R2, ... , R6 and FLAGS. Each register is 16 bits.

Note: “mov reg \$Imm”: This instruction copies the Imm(8bit) value in the register's lower 8 bits. The upper 8 bits are zeroed out.

Example:

Suppose R0 has 1110_1010_1000_1110 stored, and **mov R0 \$13** is executed.

The final value of R0 will be 0000_0000_0000_1101.

FLAGS semantics

The semantics of the flags register are:

- Overflow (V): This flag is set by `add`, `sub` and `mul`, when the result of the operation overflows. This shows the overflow status for the last executed instruction.
- Less than (L): This flag is set by the "`cmp reg1 reg2`" instruction if `reg1 < reg2`
- Greater than (G): This flag is set by the "`cmp reg1 reg2`" instruction if the value of `reg1 > reg2`
- Equal (E): This flag is set by the "`cmp reg1 reg2`" instruction if `reg1 = reg2`

The default state of the FLAGS register is all zeros. If an instruction does not affect the FLAGS register, then the state of the FLAGS register is reset to 0 upon the execution.

The structure of the FLAGS register is as follows:

Unused 12 bits												V	L	G	E
15												3	2	1	0

The only operation allowed in the FLAGS register is "`mov reg1 FLAGS`", where `reg1` can be any of the registers from R0 to R6. This instruction reads FLAGS register and writes the data into `reg1`. All other operations on the FLAGS register are prohibited.

The `cmp` instruction can implicitly write to the FLAGS register. Similarly, conditional jump instructions can implicitly read the FLAGS register.

Example:

R0 has 5, R1 has 10

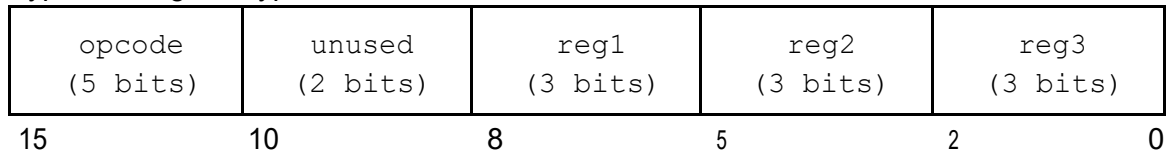
Implicit write: `cmp R0 R1` will set the L (less than) flag in the FLAGS register.

Implicit read: `jlt 10001001` will read the FLAGS register and figure out that the L flag was set, and then jump to address `10001001`.

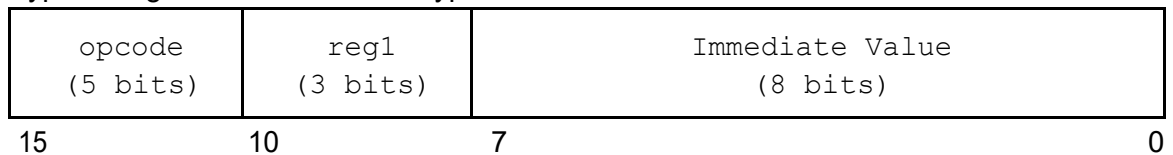
Binary Encoding

The ISA has 6 types of instructions with distinct encoding styles. However, each instruction is of 16 bits, regardless of the type.

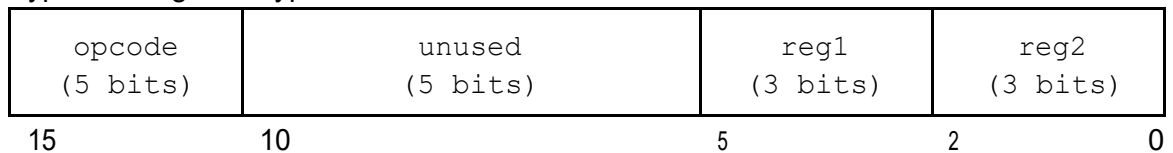
- Type A: 3 register type



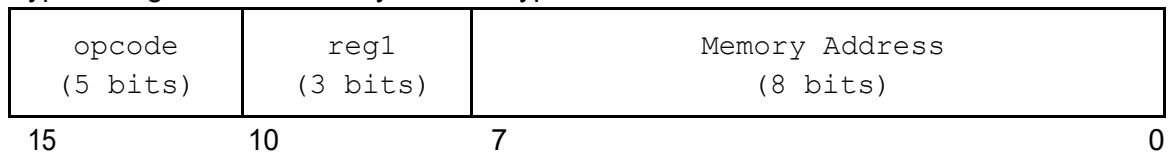
- Type B: register and immediate type



- Type C: 2 registers type



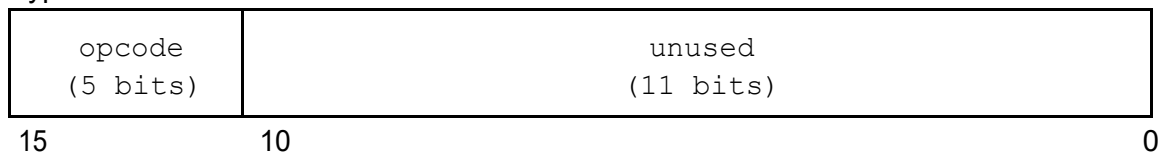
- Type D: register and memory address type



- Type E: memory address type



- Type F: halt



Binary representation for the register are given as follows:-

Register	Address
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
FLAGS	111

Executable binary syntax

The machine exposed by the ISA starts executing the code provided to it in the following format, until it reaches `hlt` instruction. There can only be one `hlt` instruction in the whole program, and it must be the last instruction. The execution starts from the 0th address. The ISA follows von-neumann architecture with a unified code and data memory. The variables must be allocated in the binary in the program order.

code
(last instruction) halt
variables

```
var X
```

```
mov R1 $10
mov R2 $100
mul R3 R1 R2
st R3 X
hlt
```

0001000100001010

```
0001000100001010
0001001001100100
0011000011001010
00101011100000101
1001100000000000
```

```

initialize(MEM);           // Load memory from stdin
PC = 0;                    // Start from the first instruction
halted = false;

while(not halted)
{
    Instruction = MEM.getData(PC);           // Get current instruction
    halted, new_PC = EE.execute(Instruction); // Update RF compute new_PC
    PC.dump();                               // Print PC
    RF.dump();                               // Print RF state
    PC.update(new_PC);                       // Update PC
}
MEM.dump()                               // Print memory state

```