LS-8 Microcomputer Spec v4.0

Registers

8 general-purpose 8-bit numeric registers R0-R7.

- R5 is reserved as the interrupt mask (IM)
- R6 is reserved as the interrupt status (IS)
- R7 is reserved as the stack pointer (SP)

> These registers only hold values between 0-255. After performing math on registers in the emulator, bitwise-AND the result with 0xFF (255) to keep the register values in that range.

Internal Registers

- `PC`: Program Counter, address of the currently executing instruction
- `IR`: Instruction Register, contains a copy of the currently executing instruction
- `MAR`: Memory Address Register, holds the memory address we're reading or writing
- `MDR`: Memory Data Register, holds the value to write or the value just read
- `FL`: Flags, see below

Flags

The flags register `FL` holds the current flags status. These flags can change based on the operands given to the `CMP` opcode.

The register is made up of 8 bits. If a particular bit is set, that flag is "true".

`FL` bits: `00000LGE`

- `L` Less-than: during a `CMP`, set to 1 if registerA is less than registerB, zero otherwise.
- `G` Greater-than: during a `CMP`, set to 1 if registerA is greater than registerB, zero otherwise.
- `E` Equal: during a `CMP`, set to 1 if registerA is equal to registerB, zero otherwise.

Memory

The LS-8 has 8-bit addressing, so can address 256 bytes of RAM total.

Memory map:

```
      top of RAM
+----------------------+
| FF  I7 vector        |    Interrupt vector table
| FE  I6 vector        |
| FD  I5 vector        |
| FC  I4 vector        |
| FB  I3 vector        |
| FA  I2 vector        |
| F9  I1 vector        |
```

```
  | F8  I0 vector         |
  | F7  Reserved          |
  | F6  Reserved          |
  | F5  Reserved          |
  | F4  Key pressed       |    Holds the most recent key pressed on the
keyboard
  | F3  Start of Stack    |
  | F2  [more stack]      |    Stack grows down
  | ...                   |
  | 01  [more program]    |
  | 00  Program entry     |    Program loaded upward in memory starting at 0
  +-----------------------+
      bottom of RAM
```

Stack

The SP points at the value at the top of the stack (most recently pushed), or at address `F4` if the stack is empty.

Interrupts

There are 8 interrupts, I0-I7.

When an interrupt occurs from an external source or from an `INT` instruction, the appropriate bit in the IS register will be set.

Prior to instruction fetch, the following steps occur:

1. The IM register is bitwise AND-ed with the IS register and the results stored as `maskedInterrupts`.
2. Each bit of `maskedInterrupts` is checked, starting from 0 and going up to the 7th bit, one for each interrupt.
3. If a bit is found to be set, follow the next sequence of steps. Stop further checking of `maskedInterrupts`.

If a bit is set:

1. Disable further interrupts.
2. Clear the bit in the IS register.
3. The `PC` register is pushed on the stack.
4. The `FL` register is pushed on the stack.
5. Registers R0-R6 are pushed on the stack in that order.
6. The address (*vector* in interrupt terminology) of the appropriate handler is looked up from the interrupt vector table.
7. Set the PC is set to the handler address.

While an interrupt is being serviced (between the handler being called and the `IRET`), further interrupts are disabled.

See `IRET`, below, for returning from an interrupt.

Interrupt numbers

- 0: Timer interrupt. This interrupt triggers once per second.
- 1: Keyboard interrupt. This interrupt triggers when a key is pressed. The value of the key pressed is stored in address `0xF4`.

Power on State

When the LS-8 is booted, the following steps occur:

- `R0`-`R6` are cleared to `0`.
- `R7` is set to `0xF4`.
- `PC` and `FL` registers are cleared to `0`.
- RAM is cleared to `0`.

Subsequently, the program can be loaded into RAM starting at address `0x00`.

Execution Sequence

1. The instruction pointed to by the `PC` is fetched from RAM, decoded, and executed.
2. If the instruction does *not* set the `PC` itself, the `PC` is advanced to point to the subsequent instruction.
3. If the CPU is not halted by a `HLT` instruction, go to step 1.

Some instructions set the PC directly. These are:

- CALL
- INT
- IRET
- JMP
- JNE
- JEQ
- JGT
- JGE
- JLT
- JLE
- RET

In these cases, the `PC` does not automatically advance to the next instruction, since it was set explicitly.

Instruction Layout

Meanings of the bits in the first byte of each instruction: `AABCDDDD`

- `AA` Number of operands for this opcode, 0-2
- `B` 1 if this is an ALU operation
- `C` 1 if this instruction sets the PC
- `DDDD` Instruction identifier

The number of operands `AA` is useful to know because the total number of bytes in any instruction is the number of operands + 1 (for the opcode). This allows you to know how far to advance the `PC` with each instruction.

It might also be useful to check the other bits in an emulator implementation, but there are other ways to code it that don't do these checks.

Instruction Set

Glossary:

- **immediate**: takes a constant integer value as an argument
- **register**: takes a register number as an argument
- `iiiiiiii`: 8-bit immediate value
- `00000rrr`: Register number
- `00000aaa`: Register number
- `00000bbb`: Register number

Machine code values shown in both binary and hexadecimal.

ADD

This is an instruction handled by the ALU.

`ADD registerA registerB`

Add the value in two registers and store the result in registerA.

Machine code:

```
10100000 00000aaa 00000bbb
A0 0a 0b
```

AND

This is an instruction handled by the ALU.

`AND registerA registerB`

Bitwise-AND the values in registerA and registerB, then store the result in registerA.

Machine code:

```
10101000 00000aaa 00000bbb
A8 0a 0b
```

CALL register

`CALL register`

Calls a subroutine (function) at the address stored in the register.

1. The address of the ***instruction*** *directly after* `CALL` is pushed onto the stack. This allows us to return to where we left off when the subroutine finishes executing.
2. The PC is set to the address stored in the given register. We jump to that location in RAM and execute the first instruction in the subroutine. The PC can move forward or backwards from its current location.

Machine code:

```
01010000 00000rrr
50 0r
```

CMP

This is an instruction handled by the ALU.

`CMP registerA registerB`

Compare the values in two registers.

- If they are equal, set the Equal `E` flag to 1, otherwise set it to 0.
- If registerA is less than registerB, set the Less-than `L` flag to 1, otherwise set it to 0.
- If registerA is greater than registerB, set the Greater-than `G` flag to 1, otherwise set it to 0.

Machine code:

```
10100111 00000aaa 00000bbb
A7 0a 0b
```

DEC

This is an instruction handled by the ALU.

`DEC register`

Decrement (subtract 1 from) the value in the given register.

Machine code:

```
01100110 00000rrr
66 0r
```

DIV

This is an instruction handled by the ALU.

`DIV registerA registerB`

Divide the value in the first register by the value in the second, storing the result in registerA.

If the value in the second register is 0, the system should print an error message and halt.

Machine code:

```
10100011 00000aaa 00000bbb
A3 0a 0b
```

HLT

`HLT`

Halt the CPU (and exit the emulator).

Machine code:

```
00000001
01
```

INC

This is an instruction handled by the ALU.

`INC register`

Increment (add 1 to) the value in the given register.

Machine code:

```
01100101 00000rrr
65 0r
```

INT

`INT register`

Issue the interrupt number stored in the given register.

This will set the _n_th bit in the `IS` register to the value in the given register.

Machine code:

```
01010010 00000rrr
52 0r
```

IRET

`IRET`

Return from an interrupt handler.

The following steps are executed:

1. Registers R6-R0 are popped off the stack in that order.
2. The `FL` register is popped off the stack.
3. The return address is popped off the stack and stored in `PC`.
4. Interrupts are re-enabled

Machine code:

```
00010011
13
```

JEQ

`JEQ register`

If `equal` flag is set (true), jump to the address stored in the given register.

Machine code:

```
01010101 00000rrr
55 0r
```

JGE

`JGE register`

If `greater-than` flag or `equal` flag is set (true), jump to the address stored in the given register.

```
01011010 00000rrr
5A 0r
```

JGT

`JGT register`

If `greater-than` flag is set (true), jump to the address stored in the given register.

Machine code:

```
01010111 00000rrr
57 0r
```

JLE

`JLE register`

If `less-than` flag or `equal` flag is set (true), jump to the address stored in the given register.

```
01011001 00000rrr
59 0r
```

JLT

`JLT register`

If `less-than` flag is set (true), jump to the address stored in the given register.

Machine code:

```
01011000 00000rrr
58 0r
```

JMP

`JMP register`

Jump to the address stored in the given register.

Set the `PC` to the address stored in the given register.

Machine code:

```
01010100 00000rrr
54 0r
```

JNE

`JNE register`

If `E` flag is clear (false, 0), jump to the address stored in the given register.

Machine code:

```
```

```
01010110 00000rrr
56 0r
```

LD

`LD registerA registerB`

Loads registerA with the value at the memory address stored in registerB.

This opcode reads from memory.

Machine code:

```
10000011 00000aaa 00000bbb
83 0a 0b
```

LDI

`LDI register immediate`

Set the value of a register to an integer.

Machine code:

```
10000010 00000rrr iiiiiiii
82 0r ii
```

MOD

This is an instruction handled by the ALU.

`MOD registerA registerB`

Divide the value in the first register by the value in the second, storing the *remainder* of the result in registerA.

If the value in the second register is 0, the system should print an error message and halt.

Machine code:

```
10100100 00000aaa 00000bbb
A4 0a 0b
```

MUL

This is an instruction handled by the ALU.

`MUL registerA registerB`

Multiply the values in two registers together and store the result in registerA.

Machine code:

```
10100010 00000aaa 00000bbb
A2 0a 0b
```

NOP

`NOP`

No operation. Do nothing for this instruction.

Machine code:

```
00000000
00
```

NOT

This is an instruction handled by the ALU.

`NOT register`

Perform a bitwise-NOT on the value in a register, storing the result in the register.

Machine code:

```
01101001 00000rrr
69 0r
```

OR

This is an instruction handled by the ALU.

`OR registerA registerB`

Perform a bitwise-OR between the values in registerA and registerB, storing the result in registerA.

Machine code:

```
10101010 00000aaa 00000bbb
AA 0a 0b
```

POP

`POP register`

Pop the value at the top of the stack into the given register.

1. Copy the value from the address pointed to by `SP` to the given register.
2. Increment `SP`.

Machine code:

```
01000110 00000rrr
46 0r
```

PRA

`PRA register` pseudo-instruction

Print alpha character value stored in the given register.

Print to the console the ASCII character corresponding to the value in the register.

Machine code:

```
01001000 00000rrr
48 0r
```

PRN

`PRN register` pseudo-instruction

Print numeric value stored in the given register.

Print to the console the decimal integer value that is stored in the given register.

Machine code:

```
01000111 00000rrr
47 0r
```

PUSH

`PUSH register`

Push the value in the given register on the stack.

1. Decrement the `SP`.

2. Copy the value in the given register to the address pointed to by `SP`.

Machine code:

```
01000101 00000rrr
45 0r
```

RET

`RET`

Return from subroutine.

Pop the value from the top of the stack and store it in the `PC`.

Machine Code:

```
00010001
11
```

SHL

This is an instruction handled by the ALU.

Shift the value in registerA left by the number of bits specified in registerB, filling the low bits with 0.

```
10101100 00000aaa 00000bbb
AC 0a 0b
```

SHR

This is an instruction handled by the ALU.

Shift the value in registerA right by the number of bits specified in registerB, filling the high bits with 0.

```
10101101 00000aaa 00000bbb
AD 0a 0b
```

ST

`ST registerA registerB`

Store value in registerB in the address stored in registerA.

This opcode writes to memory.

Machine code:

```
10000100 00000aaa 00000bbb
84 0a 0b
```

SUB

This is an instruction handled by the ALU.

`SUB registerA registerB`

Subtract the value in the second register from the first, storing the result in registerA.

Machine code:

```
10100001 00000aaa 00000bbb
A1 0a 0b
```

XOR

This is an instruction handled by the ALU.

`XOR registerA registerB`

Perform a bitwise-XOR between the values in registerA and registerB, storing the result in registerA.

Machine code:

```
10101011 00000aaa 00000bbb
AB 0a 0b
```