Computer Architecture FAQ

Contents

Common Problems

General

`CALL`/`RET`, Subroutines

Interrupts

The CPU Stack

- Why does the CPU support a stack and not some other data structure?
- On the LS-8, why does `POP` need an operand?

Registers

- What are the registers for, and what do they do?
- What is the difference between general-purpose registers and internal, special-purpose registers?
- Is the flags `FL` register one of the `Rx` registers, or is it a special register?
- What about the `IR`, `MAR`, and `MDR` registers?
- How do I move the `PC` to the next instruction without hardcoding the instruction length?
- Why is `R7` set to something other than zero?
- Are the flags on the LS-8 stored on the stack or in a register?

Number Bases and Conversions

- Why is hex base 16? Seems so random.

Questions

How much of the emulator do I need to implement?

As little as possible to get a particular LS-8 program running.

Add features incrementally. Once `print8.ls8` is working, then add a `MULT` instruction to get `mult.ls8` running. And so on.

Of course, you're *allowed* to implement as many instructions are you'd like.

This goes for individual components like registers, as well. Do you need to implement the `FL` register? If you want to use any functionality that depends on it, then yes. The spec will tell you if the thing you're implementing needs the `FL` register to work.

Once we get the `HLT` instruction, what should the emulator do?

You should exit the emulator.

You don't need to worry about any of the LS-8 internals at that point since you're exiting anyway.

How do you return values from subroutines?

Since the `RET` instruction doesn't allow you to specify a return value, you'll have to get the value back by other means.

One of the most common is to set a register (e.g. `R0`) to the return value, and the caller will just know, by convention, that the `R0` register will hold that value once the `CALL` returns.

But you could also push that value on the stack and have the caller pop it off. This would have the advantage of supporting an arbitrary number of return values.

There are no fixed rules when writing code in assembly language. Returning values in registers just happens to be a common convention.

With interrupts, why do we push everything on the stack?

The idea is that if you save the machine state on the stack, then after you service the interrupt you can restore it and seamlessly pick up where you left off.

The CPU might have been in the middle of something important when the interrupt occurred, and it'll want to get back to that once the interrupt handler is complete.

So we push the general purpose registers and internal registers on the stack, then do interrupt stuff, then restore all those registers from the stack so the CPU can carry on with what it was doing before the interrupt occurred.

What is "stack overflow"?

Short answer: it's when the stack grows into some area of memory that something else was using.

In the LS-8, this would mean the stack grew down in RAM to the point that it overwrote some of the instructions in the program.

If the stack grows down to address `0x00` on the LS-8, it wraps around to address `0xff`.

On modern machines with virtual memory, this isn't a practical concern since you'll run out of physical RAM before the stack overflow occurs.

Some interpreted languages like Python track how large their internal stacks have grown and crash out if the stack grows too large. But this is happening within the Python virtual machine, not on the hardware.

What is "stack underflow"?

This means you `POP`ped more times than you `PUSH`ed. Basically you popped an empty stack.

The CPU is more than happy to let you do this, but it's considered an error on the part of the programmer.

If the stack pointer is at address `0xff` on the LS-8, then you `POP`, it will wrap around to address `0x00`.

On the LS-8, why does the stack pointer start at address `F4`, when the first stack element is at `F3`?

Since the first thing a `PUSH` instruction does is decrement the stack pointer, it means that the stack pointer is moved to `F3` first and *then* the value is stored there. Exactly where we wanted it.

How are stacks and subroutines used by higher-level languages like Python?

In Python, when you make a function call, a bunch of space is allocated (pushed) on the stack to hold a number of things:

- The return address to come back to after the function completes
- Space for all the function parameters
- Space for all the other local variables in the function

This allocated chunk of stack is called a stack frame.

When you call any function:

1. A new stack frame is allocated (pushed)
2. Parameter values are copied from the function arguments to their spots on the stack frame

When you return from any function:

1. Any return value is copied from the stack frame into a dedicated register
2. The stack frame is deallocated (popped)

In assembly language, `CALL` doesn't allow any arguments to be passed, and `RET` doesn't allow any values to be returned.

Using stack frames gives `CALL` the power to give parameters to subtroutines.

And we can use a dedicated register, like `R0`, to pass returned values back to the caller over a `RET` instruction.

Since all the local variables for a function are stored in the stack frame, they all vaporize as soon as the stack is popped when the function returned. This is why local variables are not persistent from call to call.

Furthermore, using the stack to hold frames allows us to call functions to an arbitrary nesting level. Indeed, it is what allows for recursion at all.

Is the flags `FL` register one of the `Rx` registers, or is it a special register?

It's a special purpose register that can be added separately to the `class CPU` similar to how `PC` works.

In `class CPU`, it's convenient to have an array to store `R0` through `R7`, but the other registers are just fields in the `class`.

What about the `IR`, `MAR`, and `MDR` registers?

You can store those special-purpose registers similar to how `PC` and `FL` are stored in the `class`.

...Or, if you're not using them in any place except a single function, maybe they can be locals or function parameters.

It's a matter of which way you think produces more readable code.

What are the registers for, and what do they do?

You can think of the registers as the CPU's variables. They hold numbers. You use them like you would variable in another langauge.

In a high-level language, you can make all the variables you need. But in a CPU, there are a fixed number of them, and they have fixed names, and they only hold numbers. You cannot make more.

(The reason you can't make more is because registers are literally built out of the hardware--you can't make more without changing the hardware.)

Most operations (like math) in the CPU work on registers.

But if we have RAM, why do we need registers?

While some CPUs like the x86 can use either values in RAM or registers to do work, RAM is far, far slower to access. Nothing is faster to access in the CPU than a register. For that reason, assembly language programs use registers whenever possible to keep speed up.

If RAM is faster than an SSD, why not just store everything in RAM?

Cost. 1 TB SSD is orders of magnitude cheaper than 1 TB of RAM. And finding a motherboard that supports 1 TB of RAM is a challenge.

Also the SSD continues to store data even if power is removed, unlike RAM.

Someday someone will discover RAM that is cheap, fast, and will permanently store data, and when that happens, SSDs will vanish.

Do CPUs get hot because of the power constantly running through them?

Yup. When you run current through any regular conductor, heat is generated.

In that regard, a CPU is like a tiny, expensive electric blanket that is capable of arbitrary computation but really bad at giving you a good night's sleep.

Why is hex base 16? Seems so random.

Conveniently, one hex digit represents exactly 4 bits (AKA a *nibble*).

This means a byte can be represented by exactly 2 hex digits (assuming you put a leading zero on numbers less than `0x10`). And the biggest byte's value roundly ends at `0xff`.

It's compact, and easy to convert to and from binary.

Compare to decimal, where one decimal digit represents somewhere between 3 and 4 bits. And a byte is represented by 3 digits, isn't easily convertible to binary, and ends quite unroundly on `255` for the largest value.

How do I move the `PC` to the next instruction without hardcoding the instruction length?

Check out the spec where it talks about instruction layout.

The two high bits of the instruction tell you how many operands the instruction has. The value of those two bits plus one is the number of bytes you have to move the `PC`.

Use `>>` and an `&` mask to extract those two bits, then add one to the result, then add that to the `PC` to get to the next instruction.

> Note that some instructions (like `CALL`, `RET`, and all the `JMP` variants) move the `PC` to a specific destination. In those cases, you *do not* want to advance the PC to the next instruction.

Why are the ALU and the RAM read/write functions broken out? Can we just code the lines to do the work directly?

Because the ALU is a separate component on the CPU, and the RAM is a separate component off the CPU, it makes logical sense from a learning perspective to have different pieces of code handle the work.

Plus having the RAM access function there makes the code easier to read, and easier to change if the structure of RAM were to change somehow in the future.

Do you have some handy code for helping trace what the CPU is doing?

You'll find a `trace()` function in `cpu.py` that you can call each iteration of your main loop to get a dump of registers, etc.

Why is `R7` set to something other than zero?

`R7` has additional meaning: it is the *stack pointer*. So it needs to start just past the top of the stack so that the `PUSH` and `POP` (and `CALL` and `RET`) functions operate normally.

Why do opcodes have the numeric values that they do?

See the "Instruction Layout" part of the LS-8 spec for what the specific bits mean in any particular instruction.

In a real CPU, these bits correspond to wires that will have voltage or no-voltage on them depending on whether or not the bit in the instruction is `0` or `1`.

So the instruction bits are close to the metal, literally. Their exact meanings are closely tied with how the CPU will be physically constructed.

What is a "cache hit" or "cache miss"?

If a program accesses a byte of RAM at some address that's in the cache already, that's a *cache hit*. The byte is returned immediately.

If a program accesses a byte of RAM at some address that's not in the cache, that's a *cache miss*, and the cache must be updated by going out to RAM to get that data.

The cache is fast memory that sits between main RAM and the CPU.

It's common that if you access a byte of RAM, that you will soon access subsequent bytes in RAM. (E.g. like when printing a string, or doing a `strlen()`.) The cache makes use of this assumption.

The cache figures, if you're going to spend the time making a relatively slow RAM request for a single byte, why not go ahead and transfer the next, say 128 bytes at the same time into the faster cache. If the user then goes on to access the subsequent bytes, like they probably will, the data will already be in cache ready to use.

How are logic gates built?

They're made out of transistors. Details are getting into the realm of materials science and is beyond the scope of the course.

How does the CPU use logic gates?

Logic gates can be composed into circuits that can do far more than Boolean logical operations.

You can build an ALU, for example, that does arithmetic and comparisons using only logic gates.

You can even build circuits that store data.

The fantastic book *The Elements of Computing Systems* talks about this in great detail from the ground up.

Why is half a byte called a *nibble*?

It's a pun, playing off byte/bite. Sometimes it's spelled *nybble*.

What are the << and >> shift operators useful for?

Most commonly, they're used to get or set individual bits within a number.

This is useful if multiple values are packed into a single byte. Bytes hold numbers from 0 to 255, but parts of a byte can hold smaller numbers. For example, if you have 4 values that you know only go from 0-3 each, you can pack that into a byte as four 2-bit numbers.

Packing the numbers 3, 0, 2, and 1 into a single byte:

```
  Three
  ||
  ||  Two
  vv  vv
0b11001001
    ^^  ^^
    ||  One
    ||
    Zero
```

This technique is normally only used in high-performance situations where you absolutely must save space or bandwidth.

For example, if we wanted to extract these 3 bits from this number:

```
    vvv
0b10110101
```

We'd get `110`, which is 6 decimal. But the whole number is 181 decimal. How to extract the 6?

First, we can shift right by 3:

```
       vvv
0b00010110
```

Then we can bitwise-AND with the mask `0b111` to filter out just the bits we want:

```
         vvv
  0b00010110   <-- Right-shifted original number
& 0b00000111   <-- AND mask
------------
         110
```

And there's our 6!

On the flip side, what if we wanted to set these bits to the value 2 (`0b010`)? Right now the three bits have the value 7 (`0b111`):

```
    vvv
0b10111101
```

First let's take our 2:

```
0b010
```

and left shift it by 3:

```
0b010000
```

Secondly, let's use a bitwise-AND on the original number to mask out those bits and set them all to zero:

```
      vvv
  0b10111101   <-- original number
& 0b11000111   <-- AND mask
------------
```

```
  0b10000101
      ^^^
  These three bits set to 0, others unchanged
```

Lastly, let's bitwise-OR the shifted value with the result from the previous step:

```
      vvv
  0b10000101   <-- masked-out original number from previous step
| 0b00010000   <-- our left-shifted 2
-------------
  0b10010101
      ^^^
  Now these three bits set to 2, others unchanged
```

And there we have it. The three bits in the middle of the number have been changed from the value 7 to the value 2.

What is the difference between general-purpose registers and internal, special-purpose registers?

The general-purpose registers are `R0` through `R7`.

Special-purpose registers are things like `PC`, `FL`, and maybe `IR`, `MAR`, and `MDR`.

The main difference is this: general-purpose registers can be used directly by instructions. Special-purpose registers cannot.

```
LDI R0,4   ; Valid
LDI PC,5   ; INVALID--PC is not a general-purpose register

ADD R0,R1  ; Valid
ADD FL,R0  ; INVALID--FL is not a general-purpose register
```

In `class CPU`, it's convenient to represent the general purpose registers with an array for easy indexing from `0` to `7`.

Why does the CPU allow for stack overflow or underflow?

It takes time for the CPU to check to see if either condition has occurred. And most of the time it won't have.

CPUs are interested in running instructions as quickly as possible.

Also, you'd need additional hardware in place to make those checks, and that costs money.

Because assemnbly language is so low-level, the CPU is already putting basically ultimate trust in the developer to not do something they shouldn't do.

> If you didn't want me to overflow the stack, why did you tell me to overflow the stack?
>
> --The CPU

Why does the CPU support a stack and not some other data structure?

Turns out a stack is a really useful data structure for a number of reasons:

- It's a great place to temporarily store data.
- It's useful for holding a return address for a subroutine/function.
- It's a place to pass arguments to subroutines.
- It's a good place to hold a subroutine's local variables.
- It can hold all the information that needs to be saved while the CPU is servicing an interrupt.

Additionally, it's pretty cheap to implement. All CPUs already come with this functionality:

- Memory (for the stack data)
- Registers (for the stack pointer)
- A way to decrement and increment registers (to move the stack pointer)
- A way to read and write data to and from RAM (to retrieve and store data on the stack)

Since the CPU was doing all that anyway, adding `PUSH` and `POP` instructions is a pretty low-hanging fruit.

On a multicore CPU, is there some kind of overseer that coordinates between the cores?

Not really, and from a programmer perspective, no.

Cores have their own registers, own PCs, and generally run autonomously on their own.

What they *do* share is RAM (and usually at least some cache) and peripherals.

The real "overseer" is the operating system, which decides which programs run on which core at any particular time.

On a multicore CPU, do cores share registers or do they have their own sets?

They have their own.

Cores generally run autonomously on their own.

What they *do* share is RAM (and usually at least some cache) and peripherals.

Are the flags on the LS-8 stored on the stack or in a register?

Flags (the `FL` register) are their own special-purpose register, similar to the `PC`.

Each bit of the `FL` register has special meaning as laid out in the LS-8 spec.

Does the ALU handle conditionals/`CMP`?

Yes.

The compare instruction `CMP` will set the flags register appropriately indicating whether or not the values of the registers compared are less-than, greater-than, or equal.

This is actually quite similar to a subtraction, which the ALU can already do.

If I give you two numbers, `a` and `b`, and you compute the difference `b - a`, you can look at the result and determine if the values are equal, or if one is greater than the other.

If `b - a` is a positive number, it means that `a` is less than `b`.

If `b - a` is a negative number, it means that `a` is greater than `b`.

If `b - a` is zero, it means that `a` equals `b`.

So the ALU can use its subtraction circuitry to do a `CMP`, saving money and construction complexity.

On the LS-8, why does `POP` need an operand?

Because you probably want to know what the value was you popped off the stack, rather than just throwing it away.

Basically, `POP R0` is saying "pop the value from the top of the stack and store it in `R0`."

How are floating point numbers represented in binary?

There is a standard binary format for storing floating point numbers called IEEE 754.

It basically breaks a number into three parts:

- **Sign**--indicating positive or negative, 1 bit
- **Mantissa** (AKA "Significand")--the actual binary digits of the number, unsigned, e.g. 22 bits
- **Exponent**--signed binary exponent to apply to the mantissa, e.g. 8 bits

A simpler-to-comprehend example might be in base 10, decimal.

For example, the components that make up the decimal number `-98.273` are:

- **Sign**: `-1` (because it's -98, not 98)
- **Mantissa**: `98273` (all the digits)
- **Exponent**: `-3` (tells us where the decimal place is)

The result (again for base 10) is:

```
sign * mantissa * 10 ^ exponent
```

or:

```
-1 * 98273 * 10 ^ -3 == -98.273
```

Basically the exponent tells us how to left (if it's negative) or right (if it's positive) to shift the decimal point.

It works exactly the same way in binary (base 2):

The components that make up the binary number `0b101.11` are:

- **Sign**: `0b1` (because it's 101, not -101)
- **Mantissa**: `0b10111` (all the digits)
- **Exponent**: `-2` (tells us where the decimal place is)

Then the formula is:

```
sign * mantissa * 2 ^ exponent
```

or:

```
0b1 * 0b10111 * 2 ^ -2
```

Again, the exponent tells us how far to shift the decimal; in this case, shift it 2 bits left for `0b101.11`.

Printing out binary floating point numbers in decimal is a bit weird because you have to think in fractions of two instead of 10.

Decimal example:

`12.34` is written as:

- `1` 10s ($10 == 10^1$)
- `2` 1s ($1 == 10^0$)
- `3` 1/10ths ($1/10 == 10^{-1}$)
- `4` 1/100ths ($1/100 = 10^{-2}$)

Of course you see powers of 10 all over because it's base 10.

With base two, binary:

`11.01` is written as:

- `1` 2s ($2 == 2^1$)
- `1` 1s ($1 == 2^0$)
- `0` 1/2s ($1/2 == 2^{-1}$)
- `1` 1/4s ($1/4 == 2^{-2}$)

Which would give us (in decimal): `2 + 1 + 1/4` or `3.25`.

`11.01` binary is `3.25` decimal.

Luckily `printf()` handles that with `%f` for us.

How are signed integers represented in binary?

It's a format known as *two's complement*.

Basically, about half of the bit patterns are used for negative numbers.

The following is an example with 3-bit integers, though it applies equally well to integers of any number of bits.

Unsigned, binary on the left, decimal on the right:

```
111   7
110   6
101   5
100   4
010   2
011   3
001   1
000   0
```

Signed (same on the right, but in sorted numeric order):

```
111  -1          010   2
110  -2          011   3
101  -3          001   1
100  -4          000   0
011   3          111  -1
010   2          110  -2
001   1          101  -3
000   0          100  -4
```

Notice how the bit pattern for `7` (unsigned) is `111`, which is the same bit pattern for `-1` (signed).

So is `111` `-1` or is it `7`? It depends only on whether or not *you* are looking at those bits as signed or unsigned.

Another thing to notice is that if the high (leftmost) bit of a signed number is `1`, it means that number is negative.

Also notice how the positive signed numbers have the same bit patterns as their unsigned counterparts.

If you have a signed number (either sign) and you want to find it's two's complement opposite, first you subtract one from it and then take the bitwise-NOT of that result.

- `2` is `0b010` binary.
- Subtract `1` to get `0b001`.
- Then bitwise-NOT to get `0b110`.
- `0b110` is `-2`.

Using two's complement to represent signed numbers has one great advantage: the exact same circuitry in the ALU can be used to add or subtract numbers regardless of whether they are signed or unsigned. The ALU doesn't have to care.

How does the CPU cache work? What is L1, L2, L3 and so on?

You don't have to implement any kind of emulated cache on the LS-8.

Generally speaking, the cache is some fast memory that's closer to the CPU than RAM.

The assumption is that if a program wants a byte at address `x`, it's likely to very soon thereafter want the byte at address `x + 1`.

(Imagine printing a string, for example. You got the first byte, printed it, and now you're going to print the next one.)

So CPU first looks in the cache to see if the byte is there. If it's not, it goes out and requests a block of memory from RAM be copied into the cache. The block includes the byte in question, but also the subsequent bytes.

Then when you go to read the subsequent bytes, they're in the super fast cache and you don't have to go to RAM again, thus increasing the speed of your run.

In a modern CPU, cache is arranged as a hierarchy. Closest to the CPU is L1 (*Level 1*), which is fast and doesn't hold much. If the data isn't in L1, L1 looks to see if it's in L2. If it's not there L2 looks to see if it's in L3. If it's not there, it looks in the next level again for however many levels of cache your CPU has. Eventually if it can't be found in any level, it is obtained from RAM.

Level	Capacity	Lookup Time (nanoseconds)
L1	2-8 KB	~1 ns
L2	256-512 KB	~3 ns
L3	1024-8192 KB	~12 ns
RAM	8-32 **GB**	~100 ns

> Capacities and speeds are examples only. Actual number vary.
