Adding the Software

We have seen how sequences of bits can be interpreted in different ways to represent different types of information.

For example,

The 8-bit sequence: 11100000

Could represent the natural number 224,

Or the signed integer -32

Or the character 'à' in the extended ascii character set

Or part of an RGB colour value

Or the fixed-point number -6.0

Or Indeed many other things.

We usually refer to sequences of bits as binary numbers and, of course, we can always see them as such, but for some interpretations that number value is really incidental.

This is because values in a given collection cannot always be ordered in a unique way.

For example

Take the collection of all RGB values. Each value has an R-Component, a G-Component and a B-Component. Together the RGB Components represent a single number, but does it may sense to think of one colour being bigger or smaller than another?

Likewise, just because we use 1 to represent TRUE and 0 to represent FALSE doesn't mean that TRUE > FALSE — only it's representation is!

Sometimes we adopt a standardized ordering among values of a given type — as we do with character sets. This can be vital for information interchange but it can also be used for clever manipulation and processing of values in these types.

For example,

Using alphabetic ordering in the ASCII character set gives rise to efficient algorithms for text manipulation.

A Computer program is a collection of bit sequences to represent actions that a computer can undertake.

The number and type of such actions is dictated by number and type of components that make up the computer and the ways in which these components can interact.

Since we, as programmers, dictate which actions should take place, and when they should take place, we see these actions from our perspective and call them instructions.

Choosing an appropriate set of components and a limited number of ways that these components can interaction — while being able to do the required work is part of the art, science and engineering of computer architecture.

Likewise, designing a set of instructions that can efficiently and comprehensively, exercise the underlying hardware while, at the same time, being efficiently generated from a high-level language is usually the product of a close collaboration between hardware and software engineers.

In this Lecture we will attempt to Create a very Simple Set of Instructions to illustrate how both the Software and Hardware Come together.

Our instruction set will not be Comprehensive — merely illustrative of how things work.

Designing An Instruction Set Architecture for our machine

- The Instruction Register is 8-bits wide.
  Therefore, we will restrict our instructions to be 1-byte In size.
  This is not strictly necessary but since our RAM is so small it doesn't make sense for them to be bigger.

- Addresses Require 4-bits.
  This is important to Know, if we want to reference An address directly as part of an instruction

- Since there are only two data Registers, we only need 1 bit to distinguish them.

  However, we will leave some room for future expansion of our machine and so will use 2-bits, which will allow us to Scale to 4 data registers.

- The most basic instructions that reference memory are **load** and **store** instructions. These typically put a value from memory into a register or put a value from a register into memory.

Recall Samphire: `mov al, [100]` and `mov [100], bl`

Our load/store instructions will therefore use 4-bits for a memory address and 2-bits for a register identification — leaving 2-bits free for the opcode of the instruction.

- If we wish to have arithmetic and logic instructions, then we will typically need to specify the opcode, two operands and perhaps a result location.

One way of referring to operands and results is to put them into data registers and use the register identifications to refer to each.

Recall Samphire `add al, bl`

Potentially leaving 4-bits free for the opcode

We could imagine referring to them directly by using a memory address but that would require more bits than we have available in our 1-byte instruction.

Alternatively, we could put the memory addresses of the operands into registers and refer to them indirectly via these register values.

add [a1], [b1] - not a valid Samphire instruction

Finally, we could put imagine trying to put operand value into an instruction. However, if we wish to express each instruction in 1-byte we would not have the room, since our data values, themselves are 8-bits wide.

Recall Samphire: add a1, 80 However, in Samphire this is a multibyte instruction.

Considering all of the foregoing, let me propose the following instruction set architecture. It is by no means the only possible solution but it is consistent will all of the constraints and considerations above.

Instruction	opCode	Remaining bits of Instr.
Load A	0001	RAM Addr bbbb
Load B	0010	RAM Addr bbbb
Store A	0101	RAM Addr bbbb
Store B	0110	RAM Addr bbbb
ADD	1000	R1 R2
Sub	1001	R1 R2
AND	1010	R1 R2
OR	1011	R1 R2
NOT	1100	R1 xx
END	0000	xxxx

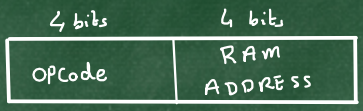
To standardize the number of bits in the opcode to 4, I decided to encode the A and B registers as part of the opcode for Load and store instructions.

I also decided not to explicitly specify the destination register for the Arithmetic and logic instructions
- Similar to way samphire does it.

- The identification of A is 01 and of B is 10
- R_1 R_2 Could be AA, AB, BA, BB i.e., 0101, 0110, 1001, 1010
- X is a 'don't care' bit.
- b is either 0 or 1

Our scheme gives us 3 different instruction formats:

Load / Store Instruction



Example1: Load A 1010

Action: A ← [1010] ; A gets value at Mem. Addr 1010

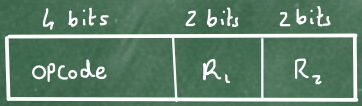
Machine Code: 0001 1010 OR 1A Hex

Example 2: Store B 0011

Action: [0011] ← B ; Value in B put into Mem at Addr 0011

Machine Code 0010 0011 OR 23 Hex

ALU instruction

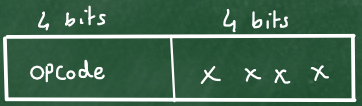


Example: Add A B

Action: A ← A + B

Machine Code: 1000 0110 or 86 Hex

END Instruction



xxxx are Don't care bits

Example: END

Action: Halt execution, Machine code: 0000xxxx OR 0X Hex