Lecture 2 : Architecture and Program Development

Data Encoding

When encoding data, we take information in one form and translate that information into an entirely different form. Often, but not always, the eventual goal is that we can translate back to the first form when it becomes convenient or necessary. Why not just use the data in its original form? We translate information to the second form because the second form is easier to deal with in some way.

Let's examine the phone example again. Imagine you are in your home town and you are talking on the phone to someone in another state. If you did not have a phone, and just stood in the middle of your yard and yelled, the person across the country is not going to hear you, no matter how loud you yell. The sound generated by your voice can be carried through the air to the people standing near you, but that sound cannot be carried by the air across the country.

Having a telephone, however, changes this. When you speak, the telephone you are holding encodes the sounds of your voice into electrical signals. While sound waves cannot easily be trans-ported long distances through air, electrical signals can be transported long distances, though high-conductivity wires. Of course, this idea is useless unless your friend across the country also has a working phone, since your friend cannot understand the encoded electrical signals. But, assuming there is a phone on the other end of the line to decode the electrical signals back into sound waves, then your friend can hear your voice in their location.

The concept here was that sounds were not convenient for sending across long distances, so we encode the sounds (our voices) into a form that is more convenient to us, electrical signals, and then decode the information back to the first form, sound waves, once the transport is done. If we think of vocal sounds as information and our goal is creating or hearing that information with our bodies, then sound waves are the most useful form. Ultimately, we want the information in "sound wave" form. But when our goal is transporting the information long distances quickly, then electrical signals are the more useful form, and thus we want the information in "electrical signal" form for cross-country portion of the information journey. The ability to convert between "sound wave" form and "electrical signal" form (using phone hardware) is critical for our species to communicate freely across great distances.

Encoding data as bits

Data encoding is of paramount importance in the design of today's computers. The reason is that a computer is nothing more than a collection of electronic circuitry – effectively, a pile of electrical switches connected together by wires. Each of the electrical switches is called a transistor, and each of them has only two settings or positions: "on", and "off". Likewise, any of the wires in the computer either has electrical current flowing through it at that particular moment (current), or else it doesn't have electrical current flowing through it at that particular moment (nocurrent). Since most of the time, we don't want to have to think about the particular circuitry layout on a computer chip, and what position switches are in and where current is flowing, we instead represent the on/off state, or the current/nocurrent state of a single transistor or the flow of electricity on a single wire, with the idea of a ***bit***.

We represent a bit as a single digit that is always either 1 or 0. We implement a bit in hardware with a transistor set to either "on" or "off", respectively, or with an electrical wire that either does (in the case of a 1), or doesn't (in the case of a 0), have electrical current flowing through it. Thinking in terms of transistors and wires is more detail than we really care about in many situations, so commonly, when thinking about having a collection of transistors, or a row of wires, we will instead view it as a collection of bits.

We use the concept of a bit to _______________ away the details of transistor state and wire current state so that we can focus on bigger picture processes. (answer: abstract)

A bit sequence or bit string is then a collection of bits. For example, the following:

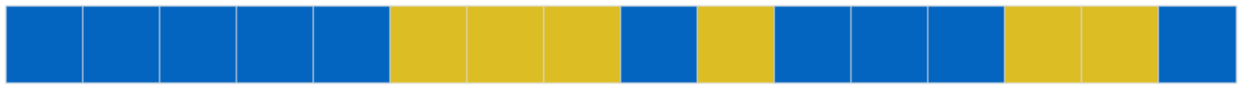
```
  1111100010111001
```

is a bit string that is 16 bits long. In the actual computer hardware, it would be represented by 16 transistors in a row set as follows:

```
   on on on on on off off off on off on on on off off on
```

or by a row of wires, set up as follows (with CUR meaning there is current on that wire, and NCUR meaning there is no current on that wire):

```
   CUR CUR CUR CUR CUR NCUR NCUR NCUR CUR NCUR CUR CUR CUR NCUR NCUR CUR
```



In this abstract image representation of the bit string, what color represents "on"? What color represents "off"?

When dealing with computers everything gets encoded using bit strings. The example on the left below shows how we might encode a set of 4 integers using 2 bits. If we want to represent the number 3 in that case, we can do it using two bits, both set to 1 (or in other words, two transistors, both set to "on", or two wires, both with current). The example on the right below shows how we might encode 8 integers using 3 bits. The more bits we have, the larger the set of integers we can encode. Given a bit string length of N, we have 2^N different bit strings of that length, and so we can encode 2^N different values (N is 2 in the example on the left below, and 3 in the example on the right below).

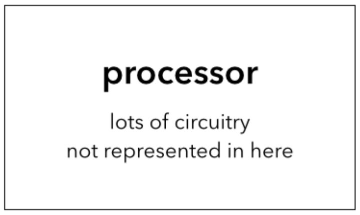
integer		bit string encoding
0	⟷	**00**
1	⟷	**01**
2	⟷	**10**
3	⟷	**11**

integer		bit string encoding
0	⟷	**000**
1	⟷	**001**
2	⟷	**010**
3	⟷	**011**
4	⟷	**100**
5	⟷	**101**
6	⟷	**110**
7	⟷	**111**

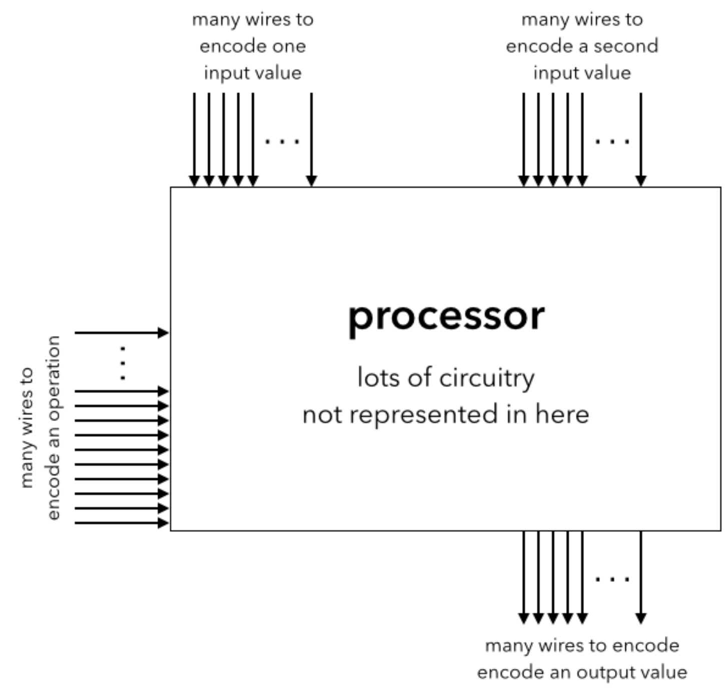
Basic computer architecture

Computer architecture is much more complicated than we are describing here, but the simplified view of things we are presenting is good enough for our purposes. You'll learn more details in other courses.

The two components of the computer that we are concerned with understanding are the ***processor*** and ***memory***. The processor is where the circuitry for performing additions, subtractions, and so on, resides.



We need the ability to send data into the processor, and receive data out of it. For example, if we want to add 2 and 3, we need a way to send the encoded values of 2 and 3 into the processor, and we need a way to receive the result - the encoded value of 5. In addition, we need to be able to tell the processor what operation it should be doing (e.g., this is an addition operation, or this is a multiplication operation). Therefore, the following is a more accurate model for a processor:


many wires to encode one input value
many wires to encode a second input value
processor
lots of circuitry not represented in here
many wires to encode an operation
many wires to encode encode an output value

That's basically all a processor is – a bunch of wires carrying input, a bunch of wires carrying output, and circuitry between the input and output wires which manipulates the input signals in the desired way, to produce the desired output signals. The interesting thing about processors is how you design that circuitry – how it is that you can wire transistors together to perform addition, subtraction, etc. on encoded data values. Unfortunately, how to design a processor is beyond the scope of this course. (You will learn the beginning ideas in CS 231 and CS 232 [or ECE 290 and ECE 291]).

The processor cannot store information, though – it can only process the information it is given. Where does that information come from, that we ultimately give to the processor? That's where memory comes in.

You can think of memory as a set of shelves. Imagine there is a room with some shelving in it, and each each shelf is numbered, starting with zero at the top.

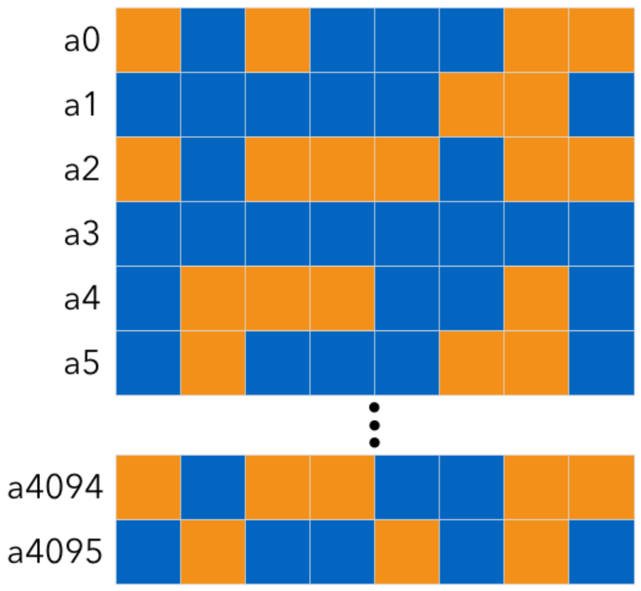
0	top shelf
1	second shelf from the top
2	third shelf from the top
3	fourth shelf from the top
4	fifth shelf from the top
5	sixth shelf from the top
6	seventh shelf from the top

I could tell you to do things such as "take the item from shelf 3 and move it to shelf 6", or "take the item from shelf 4 and throw it away". Anytime I want you to do something to one of the items on a shelf, I first tell you what shelf to go to by telling you the shelf number. It would not do you any good for me to say, "take the item from the shelf and throw it away", since you wouldn't know if I was talking about some item on shelf 0, or shelf 4, or some other shelf.

You can think of computer memory as if it were a shelving unit, except the "items" we store on these "shelves" are bit strings – one per "shelf". The "shelves" are often called ***memory cells*** or ***memory locations***, and the numerical labels on these memory locations are usually called ***memory addresses***. The addresses of our memory locations start at 0, just as the shelves in our above example did. When talking about memory, we'll use the convention of putting a letter 'a' in front of the number, just to remind ourselves that it's an address. (In reality, the address is represented as a bit string in the machine, just as the rest of our data is.)

a0	row of 8 switches
a1	row of 8 switches
a2	row of 8 switches
a3	row of 8 switches
a4	row of 8 switches
…	…
a4094	row of 8 switches
a4095	row of 8 switches

Or, an example loaded up with bits all decked out in Illini Orange and Blue, some on, some off, might look like this:



In our example above, we have 4096 memory locations (think: 4096 numbered shelves), with addresses 0 through 4095. Each memory location holds a row of 8 switches – i.e., it holds one 8-bit-long bit string. The purpose of memory – much like the purpose of a shelving unit – is to be a storage unit where data can be stored until it is directly needed. At that time, it can be read or written by the processor. That is, memory basically supports two operations:

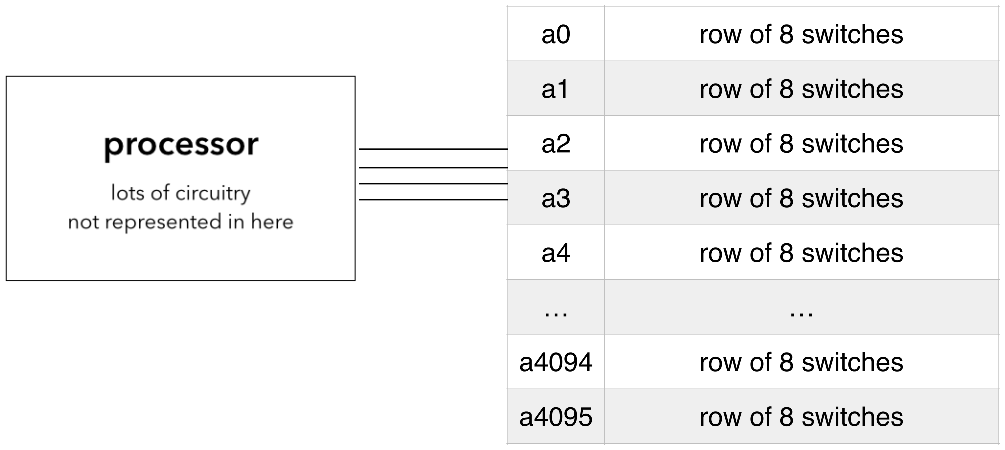
1. Given an address, obtain the 8-bit bit string stored at that given address, and send it back to whatever part of the hardware requested that information.

2. Given an address and an 8-bit bit string, write the given 8-bit bit string into the memory cell at that given address – meaning that the 8-bit bit string is now stored at that address for as long as we need it to be.

It may be that some data value we are trying to store takes up more than 8 bits. For example, we might decide to encode 2^{32} different integers as 32-bit bit strings. If we did that, we could not fit our 32-bit bit string into one memory cell, since each memory cell only holds 8 bits. So, in such cases, we break our larger bit string into 8-bit pieces and store them in consecutive memory cells. In the above example, your piece of information needs 32 bits to be represented. Assuming the storage of this information started at the memory cell with address *a104*, then you need 4 memory cells to store the information, since 32 bits will take four 8-bit cells. Since the information storage starts at *a104*, your information not only takes up the memory cell at *a104*, but also must include the memory cells at *a105*, *a106*, and *a107* as well. Those four consecutive memory cells, located at addresses *a104*, *a105*, *a106*, and *a107*, would together hold the 32 bits that our bit string needs. In general, that is how we store larger chunks of information – we break it up into many consecutive 8-bit memory cells, which collectively store our larger bit string.

Question: If we need to store a number represented by 64 bits, and the starting memory address location is *a56*, what memory addresses will be used in the storage of this information?

So, the image below is the model of a computer that we will deal with in this class. The assorted input signals to the processor come (mostly) from the memory itself, and the output signals get written back into the memory.



a0	row of 8 switches
a1	row of 8 switches
a2	row of 8 switches
a3	row of 8 switches
a4	row of 8 switches
…	…
a4094	row of 8 switches
a4095	row of 8 switches

The Stored Program Computer

Since bit strings are all a computer understands, we need to make sure the following two things are true in order for computers to be able to accomplish anything:

1. all our data is encoded as bit strings and stored in memory
2. all our instructions to the processor, are encoded as bit strings and stored in memory

That is, not only is our data stored in bit string form, but our instructions to the computer – our "computer programs" – are also stored in bit string form. This is the essential idea behind a ***stored-program computer*** – that the same hardware that is used to store our data, could also be used to store our programs, because our programs are encoded into bits, as is our data.

There are two key concepts here, one related to instructions, and one related to data. The first important idea is that whatever task we want a computer to do, we need to tell it to do that task by supplying it with some long sequence of bits that encodes our instructions to the machine. That means that writing a program is a matter of coming up with some proper series of instructions that the processor should run through, and then encoding those instructions into a form such that the program can be stored in the computer's memory. Our large sequence of bits specifying our program is broken into 8-bit pieces and stored in consecutive memory cells.

The second important idea is that any given data value is also stored as a sequence of bits. If our data value is only 8 bits long, it can be stored in one memory cell (since each cell can hold up to 8 bits). Larger bit strings get broken up into 8-bit pieces and stored in consecutive memory cells, just like larger bit strings representing programs were broken up into 8-bit pieces and stored in consecutive memory cells.