Computer Architecture

Sunday, February 20, 2022 5:34 PM

"Make everything as simple as possible, but not simpler."
- Albert Einstein (1879 - 1955)

Computer Architecture

Chapters 1-3 taught us about the hardware we will use to construct the logic within the language we learned in chapter 4. We will now create the instruction set for the CPU and thus the CPU itself to then use Assembly language (Hack language) to write programs.

Hack will be built using the *von Neumann architecture - a computer architecture based on a 1945 description by John von Neumann and others in the First Draft of a report on the EDVAC. A stored-program digital computer is one that keeps its program instructions, as well as its data, in read-write, random-access memory.*

Computer Architecture Fundamentals

5.1.1 The stored Program Concept

The stored program concept is one of the most profound, if now the very foundation of modern Computer Science.

These computers have unlimited versatility, whether it is to calculate complex arithmetic, drive a car, perform data analysis on millions of points, etc. We often take these simple yet versatile computers for granted.

The concept is fairly simple, a finite set of instructions are stored in the computers memory and they can be combines in an infinite number of ways to run an infinite number of programs. Up until the 1930's the idea of temporarily storing a "programs" code in the machine *like data* was unheard of. This is where the idea of software was born.

5.1.2 The *von Neumann Architecture*

The turning machine (1936) - an abstract artifact describing a deceptively simple computer - is used mainly in theoretical computer science for analyzing the logical foundations of computation.

The *von Neumann Machine (1945)* - a practical model that informs the construction of almost all of the computer platforms today (also described above). This machine is also CPU centered.

5.1.3 Memory

Physically - Memory is a linear sequence of addressable, fixed-size registers that have a unique address and value.

Logically - This address has two purposes, to store data and to store instructions. Both instruction and data are stored identically (as a sequence of bits).

In some variants of the von Neumann design, *memory* and *instruction* data are stored in the same memory unit, while some are stored in two separate units (each having its own pros and cons discussed later).

Data Memory:

Higher level languages use data types to perform big instructions in one line of code. The use of variables, arrays, and objects are all higher level concepts which get broken down into read/write instructions in the CPU.

Instruction Memory:

Before a high-level program can be run it must first be compiled into machine code. A file containing just 0's and 1's (machine code) is called the binary or executable, version of the program.

Before running the binary version we must first pull it from a mass storage device, and serialize its instruction into the computer's instruction memory.

How the binary version of the file gets to memory is considered an external issue so when we discuss the use of the CPU we are assuming that the binary version of the program is already in the computers memory.

5.1.3 Central Processing Unit (CPU)

The CPU is the centerpiece of the computer's architecture. It is in charge of executing the current instructions of the currently running program. Each instruction tells the CPU which computation to perform, which registers to access and which instruction to load and execute next.

Arithmetic Logic Unit (ALU):

The ALU chip was built in chapter 3. It is used to perform the low level arithmetic and logical operations featured by the computer.

How many different operations your ALU can perform is a design decision. You should also note that any operation that your ALU cannot perform can be realized later through software.

Note that hardware Implementation is more efficient but is more expensive but software implementation is less efficient and less expensive. This is where you should decide whether it will be worth it or not to actually implement the operation in hardware or software.

Registers:

We use designated registers within the CPU to reduce the condition known as starvation. The point where your CPU is waiting on data to perform the next operation.

In theory we could store temporary values in memory but the RAM and CPU (which are two separate chips) are far away from each other. We store these temporary value within designated registers within the CPU.

Starvation - When a fast processor depends on a sluggish data store for supplying its inputs and consuming its outputs.

While RAM is considered direct-access (instantaneous) memory, there is still a reasonable distance that the data must travel between from the RAM to the CPU. Especially when you are performing two billion operations (most modern day CPU's) per second!!

A typical CPU like the Intel i7-9750H (64-bit machines) have 8 registers in 32-bit mode and 16 registers in 64-bit mode. The Hack Computer utilizes only 3...

Control:

Each instruction will be comprised of 1-n agreed upon bits depending on your devices size. Before an instruction can be run, it must first be broken down into its micro-codes (the C instructions from chapter 4). After this, each micro-code tells specific parts of the CPU what to do during this instruction.

Fetch/Execute

During each step (cycle) the CPU must also discern which instruction to fetch and execute next.

Input and Output

We do not concern ourselves with the various I/O devices that could be connected to a computer. This would make the engineering process close to impossible as we would then have to device sets of instructions for each specific I/O device.

What we do instead is we use a concept called *memory-mapped I/O* which was seen in the last chapter when we output to the Hacks screen and keyboard (@SCREEN and @KBD).

Memory-mapping is just taking a specific chunk of memory and dedicating it to that piece of hardware.

Mapping these I/O devices to memory means that when, for example, you press a key on a keyboard. That binary code representing that key will also appear to change in memory. Memory acts with the device. Also remember in the last chapter when we were changing the color of the screen we would change the individual binary code for each pixel to represent the screen's pixel's new color.

I/O devices are refreshed many times per second. So it seems as though (to the user) that these changes are happening instantaneously.

This implication is that low-level computer programs can access any I/O device by manipulating its designated memory map.

The memory map convention is based on several agreed-upon contracts:
1. The data that drives each device must be serialized, or mapped, onto the computer's memory, hence the name *memory map*
2. Each I/O device is required to support an agreed-upon interaction protocol, so that its programs will be able to access it in a predictable manner.

These agreed-upon industry-wide standards are a crucial role in realizing these low-level interaction contraptions.

Device Drivers are programs that bridge the gaps between the I/O devices and memory map data and the way this data is actually rendered on, or generated by, the physical I/O Device.

5.2 The Hack Hardware Platform: Specification

A 16-bit von Neumann machine that is designed to execute Hack language. To do so the computer will utilize a CPU and two separate memory modules serving as instruction and data memory.

Instructions for the CPU will be stored within ROM and data will be stored in RAM

The CPU consists of an ALU (chapter 3), and 3 registers - D and A registers as well as the PC register.

A is used for one of three things, address to a specific place in memory, or data from an address in data memory (RAM) or data from an address in instruction memory (ROM).

5.2.2 Central Processing Unit

```
Chip name: CPU
Input:
    Instruction[16]  // Instruction to execute
    inM[16]          // The instruction's M input ( contents of RAM[A] )
    Reset            // Signals whether to restart the program ( if reset==1 )
                     // or continue executing the program ( if reset==0 )
Output:
    outM[16]         // Writing to RAM[addressM], the instruction's M output
    addressM[15]     // At which address to write?
    writeM           // Write to memory?
    pc[15]           // Address of the next instruction
```

5.2.3 Instruction Memory

```
Chip name: ROM32K
Input:     address[15]
Output:    out[16]
Function:  Emits the 16-bit value stored in the address selected by the
           address input. It is assumed that the chip is preloaded
           with a program written in the Hack machine language.
```

5.2.4 Input / Output

To see the specifics of the Hacks screen and keyboard see the Chapter notes for Chapter 4 - Machine Language.

```
Chip name: Screen // Screen memory map
Input:
    in[16]        // What to write
    address[13]   // Where to read/write
    Load          // write-enabled bit
Output:
    Out[16]       // Screen value at the given address
Function:
    Exactly like a 16-bit, 8K RAM plus a screen refresh side effect.

Chip name: Keyboard // Keyboard memory map
Output:    out[16]
Function:  Emits the 16-bit character code of the currently pressed
           key on the keyboard or 0 if no key is pressed.
```

5.2.5 Data Memory

```
Chip name: Memory         // Data memory
Input:     in[16]         // What to write.
           address[15]    // Where to read/write
           load           // Write-enabled bit
Output:    out[16]        // Value at the given address
Function:  The complete address space of the Hack computer's data memory
           Only the top 16K+8K+1 words of the address space are used.
           Accessing an address in the range 0 - 16383 results in accessing RAM16K
           Accessing an address in the range 16384 - 24575 results in accessing SCREEN.
           Accessing the address 24576 results in accessing Keyboard.
           Accessing any other address is invalid.
```

5.2.6 Computer

The computer is a chip named `Computer` that is connected to a screen and keyboard. The user has access to the screen, keyboard, and a reset button. If the user sets the reset bit to 1 then to 0 the computer will run whatever program is currently loaded. At this point the user is at the will of the software that is implemented.

This reset button is also known as on/off, or just the "boot" button. When it is pressed the computer will start up the CPU which will load set instruction (BIOS/Firmware in ROM) which will load the kernel (software) into the RAM. From which the Kernel will then execute a process (another software), that will listen to the input devices. At which the user will eventually do something to run other applications and use the computer.

```
Chip name: Computer
Input:     reset
Function:
    When reset==0, the program stored in the computers executes.
    When reset==1, the execution of the program restarts.
    To start the program's execution, set reset to 1 and then to 0.
    (It is assumed that the computer's instruction memory is loaded
    with a program written in the Hack Language).
```