**CMPSCI 630 Systems** **Spring 2023**

Lecture 7

*Lecturer: Emery Berger* *Scribe: Victor L.*

# 7 Hardware/Software Interface

In this lecture we discussed the evolution of ISA and why they continued to grow for so many yeras and also covered CISC and RISC, with some examples from Intel.

## 7.1 Instruction Set Architectures (ISA)

Assembly language is a very high level language with respect to the machine code and hardware. For example: `add $1 $2 $3` is the MIPS x86 representation of the add command. When we get to hardware, things get a lot more complicated. In figure 1, we see the circuit diagram of a 1-bit adder, the hardware that actually executes the instruction. To be able to operate on larger numbers (more than 1-bit), we need many of these in parallel, with the carry-out wire $C_{out}$ tied to the next adders carry-in wire $C_{in}$.

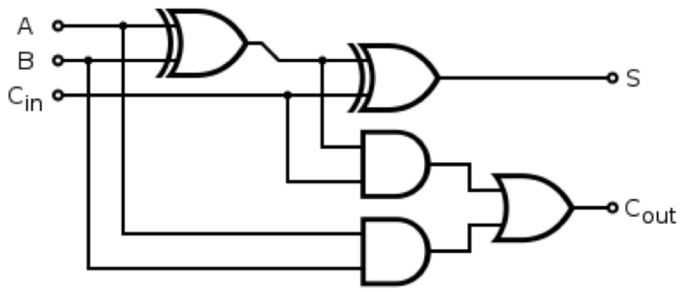

Figure 1: Full Adder [C4]

There are various levels from abstraction, from high level programming languages (e.g. C++), which are compiled down to assembly, then assembled into machine code. Operating at a lower level gives us more control over the computer and allows us to do optimizations not possible at higher levels.

The minimum for an assembly language to be Turing complete is simple; any assembly language capable of: `read, store, if $1==0 -> goto $2` are Turing complete. Being Turing complete however is not an indicator of a good assembly language, as this example would only allow us to perform operations on data 1-bit at a time. There are some trade-offs that must be considered when designing an ISA such as the abstraction level, the number of instructions, and the complexity of decoding.

Many companies, especially Intel, started added an abundance of instructions into the ISA hoping for better performance, support of more programming languages, and easier compilation of programs. In the '70s and '80s, many specialized CPUs were created. These CPUs would be entirely single purpose (like embedded systems) and would have ISAs that only supported certain applications, like database operations, running JVM, and even one that only ran LISP instructions (called the LISP-machine). This belief that dedicated hardware circuits would always outperform software is true for the most part. Take for example implementing the square-root function. The Newton-Raphson algorithm takes `log(n)` operations as it makes a guess, averages the guess with the quotient and "guesses" again, performing a sort of binary search. This is a lot of instructions as this will run recursively. In hardware, much of the overhead of executing an instruction like this could be removed as there's no fetch, decode, or execute cycles; you would just pass the circuit the operand and get the answer. However, dedicated circuits take up space on the silicon. This was however not a problem at the time as Moore's law was in full effect, giving computer architects double the chip space to work with every year or so.

With high level programming languages, there are often a lot of small things that are abstracted away to make programming easier. In C++, if we have for example a class variable (e.g. `vector<int>`) and call a function of the class (`vector.push_back()`), the address of the instructions to execute this function are stored in a virtual dispatch table (vtable). There is a table for every class so we need to check the class "tag" of the variable to find it's vtable. So for every class function call, there is quite a bit of extra work that must be done.

## 7.2 Complex Instruction Set Computer (CISC)

A lot of instructions added to ISAs are inspired by the actual code found in programs. Intel added an instruction, `REP`, to work with `MOV`. Together, these instructions would allow you to move characters of a string (or any other data) from one location to another. A lot of these instructions that were added required the use of special hardware, and so were extremely complex and often of various lengths. This required that the processor have a more complex decoder to decipher what each instruction was, and this stage could take a long time as the decode would have to on the fly figure out the length of the instruction as well. With the introduction of so many instructions were new security risks. Many of these instructions were unchecked and were just blindly executed by the CPU. An attacker could for example perform code injection through a buffer overflow attack to a region of memory that stores frequently used code (e.g. destructor), and know their code would eventually be run.

Having a complex decoder allowed more specialized instructions to be supported by the ISA, but it came at the cost of increased latency. The second stage of most processor pipelines is the decode stage which usually only takes one cycle. However, now that the instructions are more complicated, either cycle time needs to increase, decreasing the throughput, or split the decode stage into multiple stages lengthening the pipeline.

Having such a large multitude of instructions also caused other problems. There were often instructions that were very similar, and differed ever so slightly that it was extremely hard to differentiate. The problem of instruction selection for compilers became harder as it there were too many options and it wasn't immediately apparent how they differed. As a compiler writer, it is hard to determine when to use each instruction, and even harder to write a compiler that can make the differentiation.

## 7.3 Reduced Instruction Set Computer

RISC's ISA offers a more limited instruction set as well as uniform instruction size. This comes with many benefits such as being easier to write compilers and optimize code. Having uniform instruction sizes allows for pre-fetching of instructions as well as easier and faster decoding. Caching of instructions is also easier as

we can store a set number of instructions in one cache line. All this makes it really easy to pipeline a RISC processor as instruction fetch and decode are so straightforward. The simplicity of RISC allows processors to have lower depth circuits, leading to better energy efficiency, and less heat produced. As a result, nearly every mobile processor is based on RISC.
John L Hennessy and David A Patterson, the author of *The Case for the Reduced Instruction Set Computer* (and later both authors of *Computer Architecture: A Quantitative Approach*) banked on RISC being the future of computer architecture. Hennessy and Patterson both lead a group of companies (MIPS Computer and Sun Microsystems respectively) to work on the development of a RISC processor. Sun Microsystems today is a part of Oracle.

## 7.4 Intel Chips

It is doubtful that today, there are actually any ”pure” CISC architectures left (n.b. CISC and RISC have no formal definition). Intel’s chips claim to be based on a CISC architecture, however there is a RISC core on chip. The x86 ISA has an on-chip compiler which compiles instructions into microcode (a.k.a. $\mu$-code, between assembly and machine code), which is then turned into RISC instructions and fed to the RISC core. There are however still non-RISC instructions as part of the ISA (Since Intel almost never removes an instruction once it has been added) and other processing units to execute the non-RISC instructions. These other processing units are marketed to us as accelerators (e.g. NPU). Other units now included with Intel chips include a true RNG (based on quantum states) and support for cryptocurrency operations.

The Intel 8088 ISA had hardware support for integers and performed floating point operations in software; floating point instructions were prefixed with an ’F’. IBM designed a computer with two CPU slots, one for the Intel 8088 and the other for the Intel 8087. The 8087 was just a floating point unit and any floating point instruction would be sent from the 8088 to the 8087, which would then return the result back to the 8088. Eventually, these became unified into a single chip.

There are often instructions that we want to be performed on hardware as it is significantly faster. Field programmable gate arrays (FPGA) (exactly what it sounds like) are a good solution to this problem. It can be thought of as Lego blocks (logic-gates) that we put together at run-time to form any circuit we need. As a result, they are now found in many computers such as Microsoft Azure servers. An example of the speedup that can be realized from an FPGA is with AES encryption. The Xilinx Zynq 7000 processor has a 667MHz clock and a 17,000 LUT FPGA. On the processor, it is able to perform 16-bit encryption at 3Mbps maximum. Implementing an AES core on the FPGA, we are able to get 50Mbps, a nearly 20x speedup.

Intel computers for many years were akin to Porsches. They were designed with only performance in mind without regard for power consumption and they were very expensive. ARM, a company that designs RISC based processors and licenses them out, started designing chips but they weren’t fighting Intel for marketshare. ARMs primary sector was embedded systems as they were often resource constrained, so they were incapable of supporting an Intel processor. Fast forward a few years and Intel’s chips are now as fast as they can get and mobile computing platforms (cellphones) are now the most popular computing platform. Intel has none of this marketshare as there is not a phone in the world capable of powering the chip and dissipating the heat given off. ARM chips are relatively low power and don’t have a heating issue so they end up taking over the market. Intel tries to develop an energy efficient chip (the first being the Atom); the Atom turned out to not be so energy efficient and it was extremely slow.