Suggested Background Readings for Students Taking

Parallel Computer Architecture and Programming

(Tsinghua Summer 2017)

Parallel Computer Architecture and Programming depends on a number of the key concepts taught in CMU introductory systems programming course: “15-213 -- Intro to Computer Systems”.

The purpose of this document is to give a quick introduction to the key concepts that we expect you to know before beginning *Parallel Computer Architecture and Programming*. Many students will likely already know this material well. However, since sometimes introduction to computer science courses focus on how to write good code, but not how those programs are compiled down to modern computers, it may be useful for you to scan the notes below.

A complete set of **lecture slides and lecture videos** for 15-213 are available online at the following URL:

https://www.cs.cmu.edu/~213/schedule.html

Students unfamiliar with aspects of the material can study on their own time or review background material on their own.

Key topics that students should understand

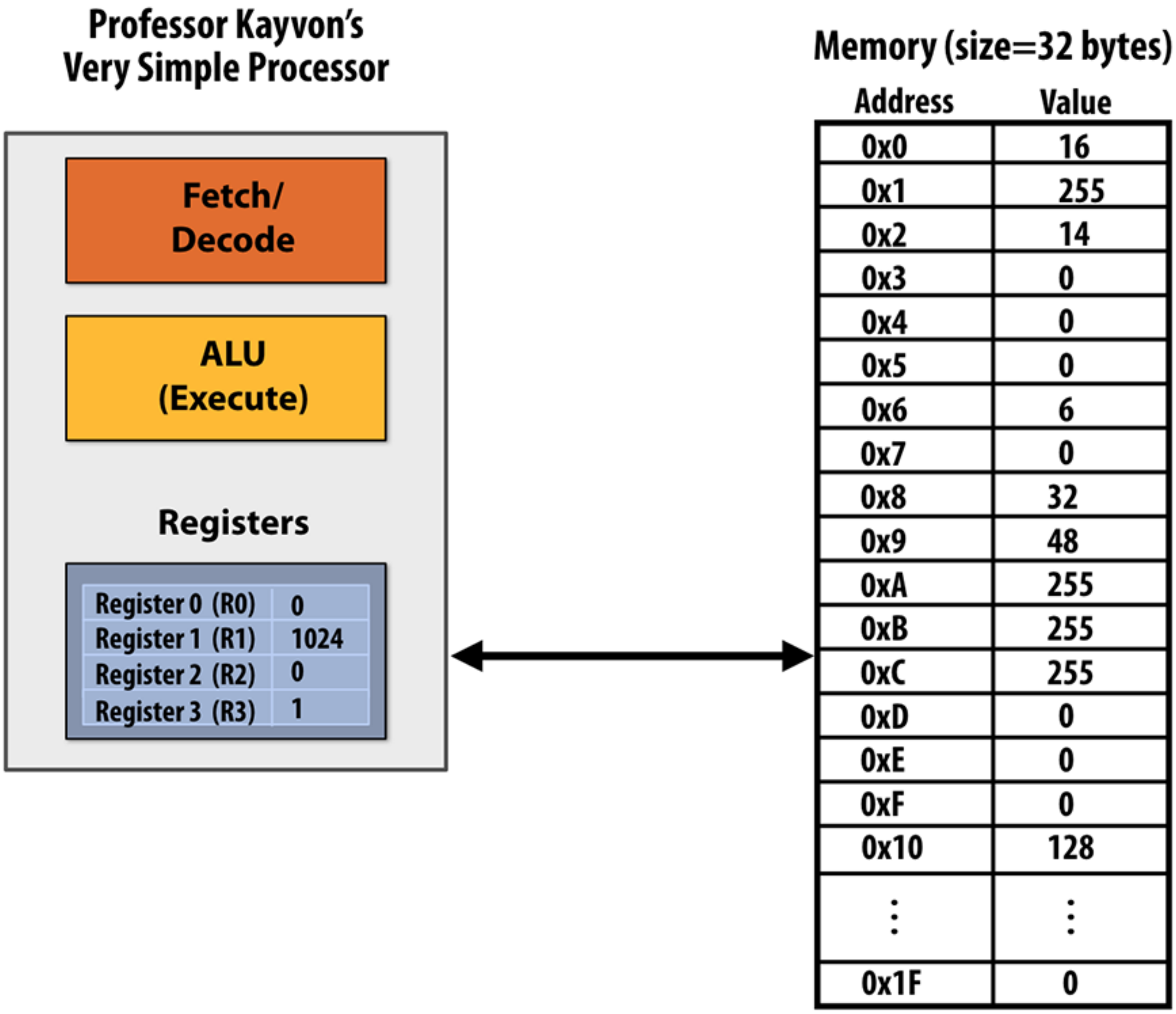
1. What are the basic pieces of computer hardware?

Perhaps the most simple definition of a computer is that a computer maintains **state** (data) and **executes operations** that manipulate that state. Given this simple description, it is reasonable to think about a simple computer as having the following components:

- **Memory**, which holds program data.

- A **processor**, which contains several components:
    - **Registers**, which are components that store data values to be manipulated by the processor’s execution unit.
    - An **execution unit**, which executes operations that manipulate the contents of registers, or move data from registers to memory or from memory to registers. For example, an execution unit might be able to add two numbers, copy a value from one register to another, etc.
    - A unit **(called fetch/decode in the figure below)** responsible for deciding what operation the processor should execute next in a program

Below is an illustration of this simple computer. To be clear, I've shown a processor with **four registers named R0, R1, R2, and R3 (that currently store the values 0, 1024, 0, and 1)**, and a memory that holds 32 bytes of data.



Registers	
Register 0 (R0)	0
Register 1 (R1)	1024
Register 2 (R2)	0
Register 3 (R3)	1

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

What is a program, and how does a computer execute instructions?

Simply put, a **computer program** is a sequence of processor **instructions**. If you have to perform a complex task, such as cooking food, you will likely follow a set of instructions. (In cooking that set of instructions is called a recipe!). A computer program is similar: it describes a sequence of operations that the computer must perform to complete a task.

When talk about processor **instructions**, we are referring to the fundamental operations that can be performed by a processor. Examples of instructions are "add the values in these two registers and put the result in this register”, “move this value from one register to another”, “move this value from memory to a register”.

A computer runs a program by executing the sequence of instructions in a program. I suggest you watch the CMU 15-213 lecture called “Machine Prog: Basics” to learn more. (It is the 6th lecture on this list.)
https://www.cs.cmu.edu/~213/schedule.html

If you’re curious about what types of instructions a modern processor can execute, take a look at this basic summary of x86-64 instructions written up by students at Stanford. It is a good list of common instructions executed by a modern Intel processor.
https://web.stanford.edu/class/cs107/guide_x86-64.html

I write programs in a language like C (I don’t write programs by specifying a list of instructions), so where do the instructions executed by a processor come from?

It is the job of the **compiler** to convert a program written in a high level programming language (that humans can read and understand) into an equivalent sequence of instructions that a processor can understand. The executable file output by a compiler is just a binary representation of a sequence of instructions. In this class, you don’t need to know too much about how a compiler translates a program into machine instructions, but if you’re interested, you should take Tsinghua’s compilers class!

What is memory?

For reasons that we will soon discuss in the class, any modern processor only has a small number of registers to hold data values. But we write programs all the time that manipulate millions (MBs) or billions (GBs) of data.

So where is all this data stored?

It is stored in the computer’s main memory.

Memory stores data organized as an array of bytes. Each byte is memory is identified by its **address**. For example, the figure above illustrates a memory holds 32 bytes of data. The byte of data add address 0x0 has the value 16. The byte of data at address 0xC (address 12) in memory has the value 255. Although in this illustration the size of memory is only 32 bytes, modern computers have high memory capacity. For example, Prof. Kayvon’s laptop has 16 GB of memory.

There are two important types of processor instructions that move data from a location in memory into registers, and from registers into a location in memory.

Memory load: Given an address, a memory load instruction will retrieve the data at the specified memory address and return the data to the processor.

Memory store: Given an address and a piece of data, memory will store the data at the specified address in memory.

You might be interested in this introduction to virtual memory from the University of Wisconsin. You can ignore the word “virtual” in the context of our class, although you may be aware of the difference between virtual memory and a machine’s physical memory if you have taken an operating systems course.
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf