Some CS3410 (or equivalent) topics you might have forgotten

RVR

August 27, 2019

This section presents a simplified view of a computer, with terminology and concepts that all CS4410 students are required to know.

As shown in Figure 1, a computer consists of

- one or more *Central Processing Units* (CPUs), each with one or more *cores*,
- memory (RAM, ROM, ...),
- a collection of *peripherals* (aka *devices*),
- an *address bus*, a *data bus*, and a *control bus*, each having a certain number of *lines*.

Memory is organized in *bytes*, each of which has 8 *bits*. Each byte (*not* each bit) has its own *address*. If an address has x bits, a core can address up to 2^x bytes. For most modern computers, $x = 64$. (We will ignore that typically only 48 or so are actually used.) For simplicity, we assume that an address bus has x lines.

Cores load and store *words* of memory. On a y-bit computer, a word has y bits. For most modern computers, $y = 64$, that is, a word on such a computer consists of 64 bits and the data bus has y lines. The address of a word is the same as the address of its first byte. (Depending on the CPU architecture, this may either be the least or most significant byte of the word.)

Addresses are typically specified in *hexadecimal* numbers. Each digit in a hexadecimal number corresponds to 4 bits and is therefore a number between 0 and F. An address of x bits can be specified with $x/4$ hexadecimal digits (including leading zeroes). We often prepend '0x' to distinguish a hexadecimal number. For example, a 32-bit address needs $32/4 = 8$ hexadecimal digits. Such addresses range from 0x00000000 to 0xFFFFFFFF. We call that range the *address space* (see left side of Figure 2—note that an address space is usually shown with the lowest address at the bottom.)

Memory stores words. Assuming there is only one CPU, a core can load a word by placing the address of the word on the address bus and activating the "read" line on the control bus. The memory will place the value of the word on

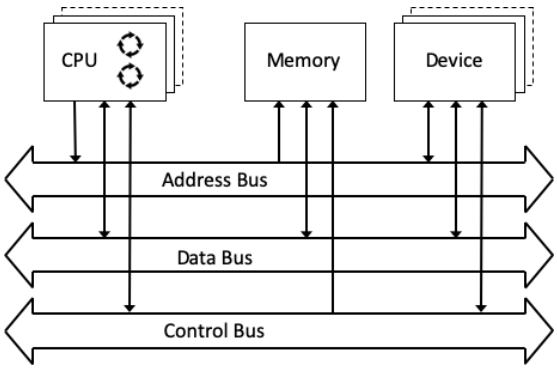

Figure 1: Schematic diagram of computer

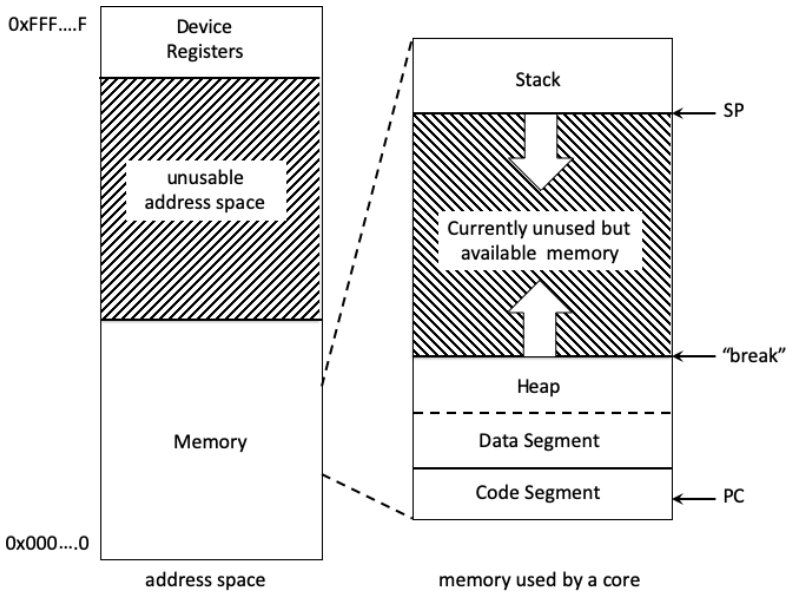

Figure 2: An address space and how a core may use part of the memory

common name	abbr.	int. std. name	#bytes	approximation
kilobyte	KB	kibibyte	2^{10}	$\sim 10^3$ bytes
megabyte	MB	mebibyte	2^{20}	$\sim 10^6$ bytes
gigabyte	GB	gibibyte	2^{30}	$\sim 10^9$ bytes
terabyte	TB	tebibyte	2^{40}	$\sim 10^{12}$ bytes
petabyte	PB	pebibyte	2^{50}	$\sim 10^{15}$ bytes

Table 1: Names and sizes of common units of memory

the data bus. A core can store a word in memory by placing the address of the word on the address bus, the value of the word on the data bus, and activating the "write" line on the control bus.

Memory sizes are usually specified in units such as kilobyte, megabytes, etc. Table 1 shows the most common ones.

A core uses (at least) three sections of memory: the code (aka *text*), the data, and the stack (see right side of Figure 2). The code section holds CPU instructions and is usually at or near the start of the address space. The PC points (or at least should point) into the code section. The data section, right after the code section, holds global data such as variables and arrays. It grows "up" when new data is allocated. We call that dynamic part of the data section the *heap*. The top of the heap is sometimes referred to as the *break*, but there is no dedicated address register to hold that value. The stack section starts high in the address space. It grows "down" as needed. Perhaps confusingly, the top of the stack, pointed to by SP, is the lowest part of the stack section. Note that the heap and the stack grow toward one another—ideally they never meet.

Each core contains a set of *registers*. Some are used to hold addresses, while others may hold words of data. The *program counter* (PC), aka *instruction pointer* (IP), is a register that holds the address of the current instruction stored in memory. The *stack pointer* (SP) holds the address of the top of the stack.

When a core executes a "call function" instruction, it saves the *context* consisting of the PC and the registers that the function may use by pushing all this context onto the stack. On function return, the context is restored by popping those registers from the stack. By restoring the PC, the execution returns to where the function was invoked. When there are multiple cores, each core should have its own stack section, but cores can share code and data sections.

Besides individual load and store operations, a core also has some combined load and store operations that execute atomically (indivisibly). These are particularly useful when multiple cores are executing simultaneously and sharing memory. The simplest example is the "test-and-set" instruction. This instruction loads a memory location and stores a specified value. Note that it returns the old contents of the memory location.

There are many types of peripherals or devices. The most common ones include screens, keyboards, mice, disks, clocks, audio interfaces, network interfaces, as well as generic device interfaces such as USB. A device is typically

memory-mapped, which means that it pretends to be memory and occupies part of the address space of a computer. For example, each pixel on the screen may be represented by a word that contains its RGB color value, so a core can simply store such values just like it would with ordinary memory.

Most devices, however, require a less straightforward way of interfacing. For example, it is not possible to read to or write from disk one word at a time—instead the disk is read or written a *block* at a time, where a block is typically at least 512 bytes. A disk device has a set of so-called *control registers*. that a core can read and write like memory:

- *block number*: the address of a block on disk;
- *memory address*: where in memory to load or store the content of a block;
- *command register*: read, write, and other commands;
- *status register*: contains the status of an outstanding command.

To read a block, a core stores the address of the block in the block number register, stores the memory address of where to load the contents of the block in the memory address register, and stores the "read command" in the command register. While the disk device retrieves the block and writes its contents to the specified memory, the core can continue executing other instructions. On completion (or error), the disk device *interrupts* the core by activating the interrupt line on the control bus. The core may then read the status register to see if the disk read was completed successfully or encountered an error.

Signal Handling

Signals are an important part of a core's execution. There are essentially two kinds of signals: *synchronous* and *asynchronous* signals.

Synchronous signals have various names such as *exceptions* or *faults* and happen deterministically during execution. Examples include include divide by zero, accessing an invalid or restricted memory address, executing an unknown or restricted instruction code, and so on. A system call instruction or breakpoint instruction also causes a synchronous signals.

Asynchronous signals are known as interupts, and are typically sent by a device to a core upon completion of an operation or when an error occurs in the device. Interrupts are *maskable*: a core can enable or disable interrupts. When disabled, an interrupt is not dropped—it is delayed until the time the core enables interrupts. A core has to explicitly drop an interrupt by writing to a device register. By contrast, exceptions or faults cannot be masked.

Signals, whether synchronous or asynchronous, may be *trapped*. The arrival of a signal causes the core to automatically do the following:

- if maskable, disable interrupts (at least for that specific device);
- push the current PC onto the stack;

- set the PC to the *trap handler*: a predefined location in the code segment.

In the trap handler, the core typically starts with saving the current context, in particular its registers. It then executes the code to handle the specific signal. Often (but not always) it is desirable to resume the original code that was executing. In that case the trap handler restores the registers and then executes a "return-from-trap" instruction. This instruction atomically pops the PC from the stack and re-enables interrupts, causing the original code to resume execution in essentially the same state it was in before the signal. Because trap handlers run with interrupts disabled, it is important that they are short and finish quickly.

When a core wants to execute some code that should not be interrupted, it should first disable interrupts, then execute the code, and then re-enable interrupts. This is useful when some data is accessed both by "normal" code and an interrupt handler. For example, the software for a keyboard device may use a queue. The trap handler is invoked when a key is pressed, and causes a character to be pushed onto the queue. Normal code may try to pop characters from the queue. When such code runs unmediated, this can lead to so-called *race conditions* that could corrupt the queue data structure.