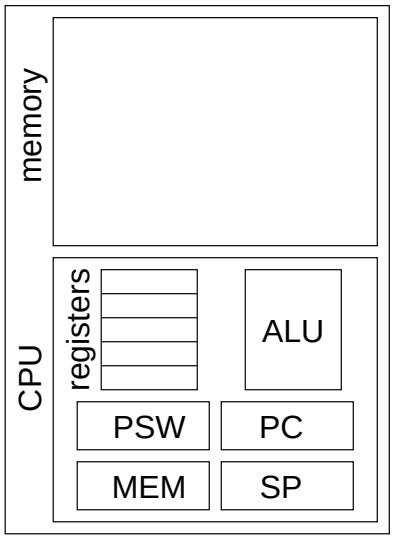
Appendix ***A***

Background on Computer Architecture

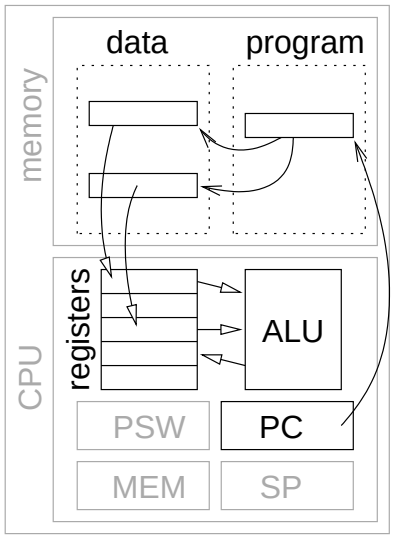
Operating systems are tightly coupled with the architecture of the computer on which they are running. Some background on how the hardware works is therefore required. This appendix summarizes the main points. Note, however, that this is only a high-level simplified description, and does not correspond directly to any specific real-life architecture.



At a very schematic level, we will consider the computer hardware as containing two main components: the memory and the CPU (central processing unit). The memory is where programs and data are stored. The CPU does the actual computation. It contains general-purpose registers, an ALU (arithmetic logic unit), and some special purpose registers. The general-purpose registers are simply very fast memory; the compiler typically uses them to store those variables that are the most heavily used in each subroutine. The special purpose registers have specific control functions, some of which will be described here.

The CPU operates according to a hardware clock. This defines the computer's "speed": when you buy a 3GHz machine, this means that the clock dictates 3,000,000,000 cycles each second. In our simplistic view, we'll assume that an instruction is executed in every such cycle. In modern CPUs each instruction takes more than a single cycle, as instruction execution is done in a pipelined manner. To compensate for this, real CPUs are superscalar, meaning they try to execute more than one instruction per cycle, and employ various other sophisticated optimizations.

One of the CPU's special registers is the *program counter* (PC). This register points to the next instruction that will be executed. At each cycle, the CPU loads this instruction and executes it. Executing it may include the copying of the instruction's operands from memory to the CPU's registers, using the ALU to perform some operation on these values, and storing the result in another register. The details depend on the architecture, i.e. what the hardware is capable of. Some architectures require operands to be in registers, while others allow operands in memory.



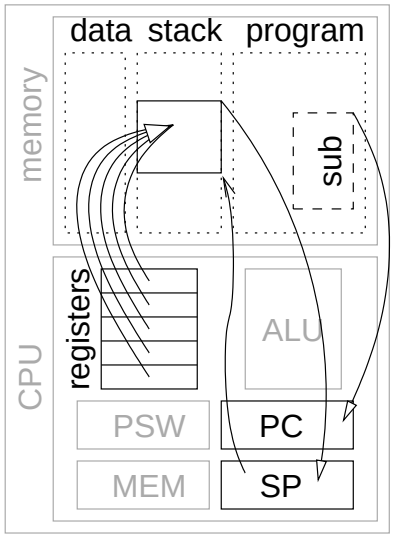
Exercise 13 *Is it possible to load a value into the PC?*

Exercise 14 *What happens if an arbitrary value is loaded into the PC?*

In addition to providing basic instructions such as add, subtract, and multiply, the hardware also provides specific support for running applications. One of the main examples is support for calling subroutines and returning from them, using the instructions `call` and `ret`. The reason for supporting this in hardware is that several things need to be done at once. As the called subroutine does not know the context from which it was called, it cannot know what is currently stored in the registers. Therefore we need to store these values in a safe place before the call, allow the called subroutine to operate in a "clean" environment, and then restore the register values when the subroutine terminates. To enable this, we define a special area in memory to be used as a call stack. When each subroutine is called, it's data is saved on top of this stack.

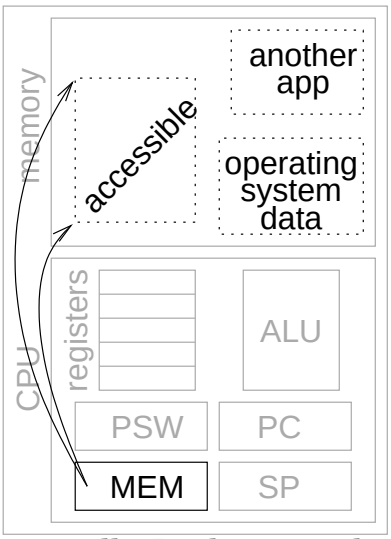
The `call` instruction does the first part:

1. It stores the register values on the stack, at the location pointed to by the stack pointer (another special register, abbreviated SP).
2. It also stores the return address (i.e. the address after the `call` instruction) on the stack.
3. It loads the PC with the address of the entry-point of the called subroutine.
4. It increments the stack pointer to point to the new top of the stack, in anticipation of additional subroutine calls.



After the subroutine runs, the `ret` instruction restores the previous state:

1. It restores the register values from the stack.
2. It loads the PC with the return address that was also stored on the stack.
3. It decrements the stack pointer to point to the previous stack frame.



The hardware also provides special support for the operating system. One type of support is the mapping of memory. This means that at any given time, the CPU cannot access all of the physical memory. Instead, there is a part of memory that is accessible, and other parts that are not. This is useful to allow the operating system to prevent one application from modifying the memory of another, and also to protect the operating system itself. The simplest implementation of this idea is to have a pair of special registers that bound the accessible memory range. Real machines nowadays support more sophisticated mapping, as described in Chapter 4.

A special case of calling a subroutine is making a system call. In this case the caller is a user application, but the callee is the operating system. The problem is that the operating system should run in privileged mode, or kernel mode. Thus we cannot just use the `call` instruction. Instead, we need the `trap` instruction. This does all what `call` does, and in addition sets the mode bit in the processor status word (PSW) register. Importantly, when `trap` sets this bit, it loads the PC with the predefined address of the operating system entry point (as opposed to `call` which loads it with the address of a user function). Thus after issuing a `trap`, the CPU will start executing operating system code in kernel mode. Returning from the system call resets the mode bit in the PSW, so that user code will not run in kernel mode.

There are other ways to enter the operating system in addition to system calls, but technically they are all very similar. In all cases the effect is just like that of a trap: to pass control to an operating system subroutine, and at the same time change the CPU mode to kernel mode. The only difference is the trigger. For system calls, the trigger is a `trap` instruction called explicitly by an application. Another type of trigger is when the current instruction cannot be completed (e.g. division by zero), a condition known as an exception. A third is interrupts — a notification from an external device (such as a timer or disk) that some event has happened and needs handling by the operating system.

The reason for having a kernel mode is also an example of hardware support for the operating system. The point is that various control functions need to be reserved to the operating system, while user applications are prevented from performing them. For example, if any user application could set the memory mapping registers, they would be able to allow themselves access to the memory of other applications. Therefore the setting of these special control registers is only allowed in kernel mode. If a

user-mode application tries to set these registers, it will suffer an illegal instruction exception.