The main components of a computer (CPU, memory, I/O devices, buses) as viewed from the user perspective is called the **architecture** of the computer.

In CS240, we were focused on the hardware details of how a computer works (which is referred to as computer **organization**). In Lab, we actually implemented a predefined computer architecture. In CS340, we get a chance to see how to come up with the specification of the architecture itself.

For Lab 9, we will use the design of our CS240 project (I told you we'd return to it!) as a basis for a more complex architecture. The resulting machine will serve as a vehicle for showing the principles on which a CPU architecture might be designed and implemented.

There are both software and hardware aspects of a computer's architecture. We will begin here by discussing the software.

**Software Aspects**

To define an architecture for a system, you must consider what type of tasks you must perform with the available instruction set. You then add the logic signals that are required for communication between the CPU and memory and I/O devices.

To begin, define essential programming tasks (including I/O, assignment, and simple control of program flow) to decide which instructions are needed.

The following are a list of such essential tasks (some of which we solved with our CS240 project instruction set):

1. I/O - need a way to read from an input device such as a keyboard - similarly, need a way to write to an output device, such as a monitor, printer, disk drive, etc. We did this with memory-mapped I/O in CS240 project; define separate instructions for reading and writing in this architecture, instead.

2. Assignment - all values are read into or written from the accumulator. We already have instruction which allow us to do assignment; specifically, LOAD and STORE as previously defined.

3. Negation - the simplest arithmetic operation, necessary for many tasks. NEG, as previously defined.

4. Addition - ADD, as previously defined.

5. Subtraction - can be performed with a combination of ADD and NEG.
6. Two-byte arithmetic - need ADDC, add-with-carry instruction. Will need to add a carry flag, in addition to previously defined N and Z bits.

7. Constants - implemented in previous design by storing values in reserved memory locations. More convenient to allow "immediate addressing mode", so immediate value can be specified in operand. This will expand our instruction set, and necessitate the use of an extension word to store the immediate value of the operand.

8. Multiplication and division - although possible in our previous design through repeated addition, awkward and inefficient for large products. Better to use addition combined with shifting. New instructions, ASR, arithmetic shift right, and SL, logical shift left, are required.

9. Logic operations - very handy for masking purposes and tasks such as code conversion, not implemented in previous design. New instructions AND and OR will be very useful for this purpose.

10. Conditional execution - implemented previously with JMP, JMPN, and JMPZ instructions. These are adequate in their present form.

11. Interrupt I/O processing - we have been learning in CS340 that interrupt processing is key for efficient processing of I/O in many computer systems. So, we'd like to add this to our design. There are a number of details to consider, but the primary software aspect is the need for an RTE (return from the exception) instruction. When an interrupt is generated, we will want to push the PC on the stack, and put the interrupt service routine address into the PC. When the routine is complete, we will allow the CPU to return control to the main program from the interrupt handling routine, by restoring the PC from the stack.

Other Issues
Several aspects of the CS240 project architecture make it fairly awkward to program. For example, having a single general-purpose register (the accumulator) makes some operations cumbersome. It would be very nice to have at least two additional registers where values can be stored temporarily (let's call these A and B).

This adds to the complexity of our instruction set, because A and B are then potential operands for many instructions (rather than simply an absolute address, as was the case for the CS240 project). We now have 3 possible addressing modes; implicit (accumulator as source/target), immediate (immediate value specified in extension word), and absolute (absolute address, register, or I/O device specified as part of instruction or in extension word).
We also need to define a system "stack", and a register to hold its current value (SP), where we can store values temporarily (specifically, the PC for interrupt processing, but also any other data values we wish to store temporarily).

To really make our CS240 project design practical, we definitely need a larger memory space (if you recall, the 5-bit bus gave us a 32 address space, which we used for all program instructions, data, and I/O). This severely limited the size of the programs we could write. We also need a larger space if we want to reserve part of it as a system stack, as indicated above.

It would also be a good idea to define an area of EPROM memory, where we could potentially store monitor and bootstrap programs (primitive operating system). This will require some address decoding, to select the correct device for a given address.

Therefore, we will expand our address bus to 12 bits. However, we will stay with an 8 bit data bus. That means that the data and address registers must be 12 bits. With a larger instruction set, our instruction register will also need to be larger than 3 bits (as it was for the CS240 project).

The following are the 8 new instructions and their function:

1. Signed addition (ACC + OP + carry -> ACC):
   OP = A,B,SP, immediate value, or memory address whose contents will be used,
2. Arithmetic shift right (ASR ACC -> ACC),
3. Logical shift left (SL ACC -> ACC),
4. AND (ACC AND OP -> ACC):
   OP = A,B,SP, immediate value, or memory address whose contents will be used,
5. OR (ACC OR OP -> ACC),
   OP = A,B,SP, immediate value, or memory address whose contents will be used,
6. INPUT (input device specified by 2-bit operand -> ACC):
   2 bits will encode 4 input and 4 output devices.
7. OUTPUT (ACC -> output device specified by 2-bit operand),
8. RTE (return from exception, restore PC from stack).

We'll also need to change the definition of some of the previous instructions, to incorporate the new addressing modes:

1. Unsigned addition (ACC + OP -> ACC):
   OP = A,B,SP, immediate value, or memory address whose contents will be used,
2. LOAD (OP -> ACC):
   OP = A,B,SP, immediate value, or memory address whose contents will be used,
3. STORE (ACC -> OP):
   OP = A,B,SP, or memory address whose contents will be used.
**Hardware Aspects**

Our machine will need the following data paths and signals to access memory and communicate with I/O devices:

1. As mentioned above, an 8-bit data bus to memory and I/O devices.

2. A 12-bit address bus shared between I/O devices (which only use 2 bits of the address bus) and memory. You will have to think about the details of how to manage the 12-bit registers with an 8-bit data path.

3. A way to distinguish between memory and I/O, called IO/MEM'. When high, CPU accessing memory; when low, accessing I/O. We did this simply with memory-mapped logic in our CS240 project. Now that we have specific IN and OUT instructions, it will be generated differently.

4. A signal, ENM, set high on a memory or I/O access, and low otherwise. It should work as designed for CS240, so we do not have to change it.

4. A signal to indicate whether a read or write is occurring on a data transfer (R/W', already implemented in CS240 project).

5. An INT signal, which interrupts the CPU after the current instruction executes.

6. To separate successive internal CPU states and to control memory access, we need a clock signal, divided into 2 phases. On phase 1, the CPU must put all the correct information on the bus (address, data, R/W', IO/MEM'). On phase 2, when all these signals are stable, the actual memory or I/O device should be enabled (ENM), and the transfer should be allowed to occur. The CS240 project already uses such a timing scheme, and should be adequate for our purposes.

**Your Tasks**

You may perform the following tasks by marking up and adding pages where necessary to the specification for the CS240 project. Please make all the modifications to the specification that are needed to implement the architecture discussed above.

- Increase size of address bus. Add EPROM and specify stack space. Add address decoding. What will your computer do on a reset? How will it start a program? How does it execute from EPROM?
Devise instruction format to encode new instructions and addressing modes. Don’t forget the carry bit, and define the effect of each instruction on it. How will you produce the carry bit?

Add/modify control signals/logic, with the appropriate timing, to bus (remember, I/O is no longer memory-mapped).

Add control signals/devices to bus for interrupt I/O processing (come up with a scheme for how the CPU handles interrupts). Assume only a single level of interrupt is used. What kind of additional hardware do you think you might need?

Modify CPU/microcode to incorporate above changes.

Modify specification wherever required to incorporate above changes.