There are many approaches to building a general-purpose computer that can be easily re-programmed for new problems.

Almost all modern computers are based on the “stored program” computer architecture developed by John von Neumann in 1945, which is now commonly referred to as the “von Neumann model”.

The von Neumann model has three components.

There’s a central processing unit (aka the CPU) that contains a datapath and control FSM as described previously.

The CPU is connected to a read/write memory that holds some number W of words, each with N bits.

Nowadays, even small memories have a billion words and the width of each location is at least 32 bits (usually more).

This memory is often referred to as “main memory” to distinguish it from other memories in the system.

You can think of it as an array: when the CPU wishes to operate on values in memory , it sends the memory an array index, which we call the address, and, after a short delay (currently 10’s of nanoseconds) the memory will return the N-bit value stored at that address.

Writes to main memory follow the same protocol except, of course, the data flows in the opposite direction.

We’ll talk about memory technologies a couple of lectures from now.

And, finally, there are input/output devices that enable the computer system to communicate with the outside world or to access data storage that, unlike main memory, will remember values even when turned off.

The key idea is to use main memory to hold the instructions for the CPU as well as data.

Both instructions and data are, of course, just binary data stored in main memory.

Interpreted as an instruction, a value in memory can be thought of as a set of fields containing one or bits encoding information about the actions to be performed by the CPU.

The opcode field indicates the operation to be performed (e.g., ADD, XOR, COMPARE).

Subsequent fields specify which registers supply the source operands and the destination register where the result is stored.

The CPU interprets the information in the instruction fields and performs the requested operation.

It would then move on to the next instruction in memory, executing the stored program step-by-step.

The goal of this chapter is to discuss the details of what operations we want the CPU to perform, how many registers we should have, and so on.

Of course, some values in memory are not instructions!

They might be binary data representing numeric values, strings of characters, and so on.

The CPU will read these values into its temporary registers when it needs to operate on them and write newly computed values back into memory.

Mr. Blue is asking a good question: how do we know which words in memory are instructions and which are data?

After all, they're both binary values!

The answer is that we can't tell by looking at the values — it's how they are used by the CPU that distinguishes instructions from data.

If a value is loaded into the datapath, it's being used as data.

If a value is loaded by the control logic, it's being used an instruction.

So this is the digital system we'll build to perform computations.

We'll start with a datapath that contains some number of registers to hold data values.

We'll be able to select which registers will supply operands for the arithmetic and logic unit that will perform an operation.

The ALU produces a result and other status signals.

The ALU result can be written back to one of the registers for later use.

We'll provide the datapath with means to move data to and from main memory.

There will be a control unit that provides the necessary control signals to the datapath.

In the example datapath shown here, the control unit would provide ASEL and BSEL to select two register values as operands and DEST to select the register where the ALU result will be written.

If the datapath had, say, 32 internal registers, ASEL, BSEL and DEST would be 5-bit values, each specifying a particular register number in the range 0 to 31.

The control unit also provides the FN function code that controls the operation performed by the ALU.

The ALU we designed in Part 1 of the course requires a 6-bit function code to select between a variety of arithmetic, boolean and shift operations.

The control unit would load values from main memory to be interpreted as instructions.

The control unit contains a register, called the “program counter”, that keeps track of the address in main memory of the next instruction to be executed.

The control unit also contains a (hopefully small) amount of logic to translate the instruction fields into the necessary control signals.

Note the control unit receives status signals from the datapath that will enable programs to execute different sequences of instructions if, for example, a particular data value was zero.

The datapath serves as the brawn of our digital system and is responsible for storing and manipulating data values.

The control unit serves as the brain of our system, interpreting the program stored in main memory and generating the necessary sequence of control signals for the datapath.

Instructions are the fundamental unit of work.

They’re fetched by the control unit and executed one after another in the order they are fetched.

Each instruction specifies the operation to be performed along with the registers to supply the source operands and destination register where the result will be stored.

In a von Neumann machine, instruction execution involves the steps shown here:

the instruction is loaded from the memory location whose address is specified by the program counter.

When the requested data is returned by the memory, the instruction fields are converted to the appropriate control signals for the datapath,

selecting the source operands from the specified registers, directing the ALU to perform the specified operation, and storing the result in the specified destination register.

The final step in executing an instruction is updating the value of the program counter to be the address of the next instruction.

This execution loop is performed again and again.

Modern machines can execute up more than a billion instructions per second!

The discussion so far has been a bit abstract.

Now it’s time to roll up our sleeves and figure out what instructions we want our system to support.

The specification of instruction fields and their meaning along with the details of the datapath design are collectively called the instruction set architecture (ISA) of the system.

The ISA is a detailed functional specification of the operations and storage mechanisms and serves as a contract between the designers of the digital hardware and the programmers who will write the programs.

Since the programs are stored in main memory and can hence be changed, we’ll call them software, to distinguish them from the digital logic which, once implemented, doesn’t change.

It’s the combination of hardware and software that determine the behavior of our system.

The ISA is a new layer of abstraction: we can write programs for the system without knowing the implementation details of the hardware.

As hardware technology improves we can build faster systems without having to change the software.

You can see here that over a fifteen year timespan, the hardware for executing the Intel x86 instruction set went from executing 300,000 instructions per second to executing 5 billion instructions per second.

Same software as before, we’ve just taken advantage of smaller and faster MOSFETs to build more complex circuits and faster execution engines.

But a word of caution is in order!

It’s tempting to make choices in the ISA that reflect the constraints of current technologies, e.g., the number of internal registers, the width of the operands, or the maximum size of main memory.

But it will be hard to change the ISA when technology improves since there’s a powerful economic incentive to ensure that old software can run on new machines, which means that a particular ISA can live for decades and span many generations of technology.

If your ISA is successful, you’ll have to live with any bad choices you made for a very long time.

Designing an ISA is hard!

What are the operations that should be supported?

How many internal registers?

How much main memory?

Should we design the instruction encoding to minimize program size or to keep the logic in the control unit as simple as possible?

Looking into our crystal ball, what can we say about the computation and storage capabilities of future technologies?

We’ll answer these questions by taking a quantitative approach.

First we’ll choose a set of benchmark programs, chosen as representative of the many types of programs we expect to run on our system.

So some of benchmark programs will perform scientific and engineering computations, some will manipulate large data sets or perform database operations, some will require specialized computations for graphics or communications, and so on.

Happily, after many decades of computer use, several standardized benchmark suites are available for us to use.

We’ll then implement the benchmark programs using our instruction set and simulate their execution on our proposed datapath.

We’ll evaluate the results to measure how well the system performs.

But what do we mean by “well”?

That’s where it gets interesting: “well” could refer to execution speed, energy consumption, circuit size, system cost, etc.

If you’re designing a smart watch, you’ll make different choices than if you’re designing a high-performance graphics card or a data-center server.

Whatever metric you choose to evaluate your proposed system, there’s an important design principle we can follow:

identify the common operations and focus on them as you optimize your design.

For example, in general-purpose computing, almost all programs spend a lot of their time on simple arithmetic operations and accessing values in main memory.

So those operations should be made as fast and energy efficient as possible.

Now, let’s get to work designing our own instruction set and execution engine, a system we’ll call the Beta.