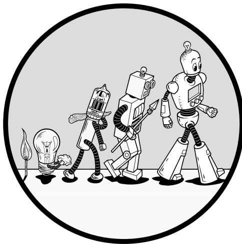
# 3

# BASIC CPU-BASED ARCHITECTURE



Modern CPUs are some of the most complex structures known to humanity, but the basic concepts underlying them, such as executing instructions sequentially or jumping forward or backward to different instructions, are actually quite simple and haven't changed for over 150 years. To ease our way into the study of CPU architecture, this chapter introduces these fundamental concepts by looking at a related but simpler system: a mechanical music player. You'll then see how the same concepts, together with RAM, form the basis of Charles Babbage's Analytical Engine. Studying—and programming—this mechanical system will make it easier to understand what's going on when we turn our attention to electronic systems in Chapter 4.

## A Musical Processing Unit

For a machine to be a computer, it needs to be *general purpose*, meaning it must be able to perform different tasks according to a user specification. One way to arrange for this is to have the user write a sequence of instructions—a program—and have the machine carry them out. A musical score can be viewed as a program, and so we can think of a machine that reads and performs musical scores as a kind of musical computer. We'll call such a device a *musical processing unit*.

In Chapter 1 we looked briefly at musical processing units such as barrel organs and music boxes. After Babbage, musical automata and their programs continued to evolve. Around 1890, "book organs" replaced barrels with continuous, joined decks of punch cards ("book music"), which could accommodate arbitrarily longer compositions without the size limit imposed by a barrel. By 1900 these had evolved to pianolas, or player pianos (Figure 3-1), which used punched paper *piano rolls* instead of cards to drive domestic pianos, rather than church organs. Player pianos are still found today; you might hear one providing background jazz in a mid-range hotel that can afford a piano but not a pianist.

*Figure 3-1: A player piano (1900)*

Let's think about some of the types of instructions found in musical scores that might be playable on these machines. These will be similar to but perhaps more familiar than concepts that we'll need later to make computers. We'll consider only a monophonic instrument here, meaning it can only play one note at a time.

The set of possible instructions that we can give to an automated musical instrument usually contains one instruction per available note. This might be an instruction to "play middle C" or "play the G above middle C," for example. Each row of a player piano's paper roll represents a time and contains one column per musical pitch, which is specified to be either on (punched) or off (not punched) at that time. Modern computer music software such as Ardour 5, released in 2018, continues to use this type of

piano roll notation (turned on its side for human viewers, so time scrolls more intuitively from left to right) to generate electronic music (Figure 3-2).

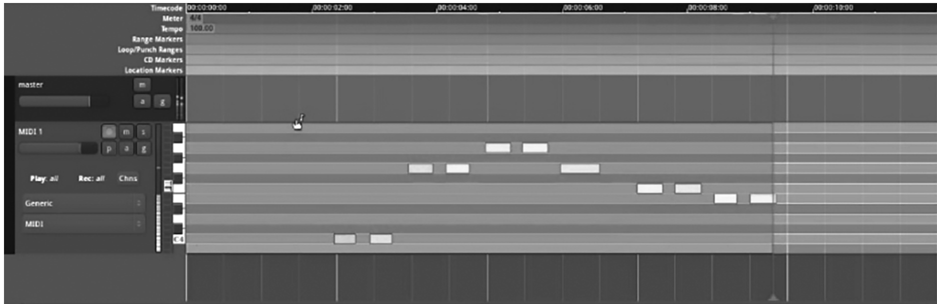



*Figure 3-2: An Ardour 5 piano roll interface (2018)*

When a player piano reads a piano roll, one row at a time is placed into a reader device. Let's call this *fetching* the instruction. The instruction is then *decoded* by some machinery that looks at the punch-hole coding and turns it into a physical activation of some machinery that is going to play the note, such as by opening a tube for air to flow into an organ pipe. Then this machinery actually *executes* the performance of the note.

Usually when a human or mechanical music player is following a music program (score), they will execute (play) each instruction (note) and then move on to the next one, advancing their position in the program by one instruction. But sometimes there will also be special additional instructions that tell them to *jump* to another place in the program rather than advancing to the next instruction. For example, *repeats* and *dal segno (D.S.)* are used to jump back to an earlier instruction and continue execution from there, while *codas* are instructions to jump forward to a special ending section. Figure 3-3 shows a musical program.

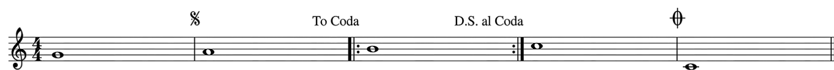



*Figure 3-3: A musical program with notes G, A, B, high C, and low C, as well as jumps shown by repeats, dal segno, and coda*

You can build a barrel organ or player piano that encodes these jump instructions using extra, non-note columns in their punch cards. When one of these is punched, it might be interpreted as an instruction to fast-forward or rewind the barrel or punch cards to a previous or later line. Figure 3-3 could then be coded with punches representing something like:

```
1.  play note: G
2.  play note: A
3.  check if you have been here before
4.  if so, jump to instruction 10
5.  play note: B
6.  check if you haven't been here before
```

```
7.  if so, jump to instruction 5
8.  play note: high C
9.  jump to instruction: 2
10. play note: low C
11. halt
```

If you don't read music, this program explains exactly what the musical score does!

### From Music to Calculation

It's a small conceptual step from this musical processing unit to building a machine that performs arithmetical, rather than musical, operations.

Suppose you've already built several small mechanical devices that each perform some arithmetic operation. For example, Pascal's calculator is a machine that performs integer addition. With some thought, we could similarly construct machines like Pascal's calculator to perform integer multiplication, subtraction, division, and column shifting. We could then write a program, much like a musical score, that would specify the sequence in which we'd like each of these simple machines to be activated.

Assuming that your arithmetic machines all share a single accumulator where the result of each operation is stored, you could describe calculations similarly to sequences of instructions for pressing buttons on a calculator, such as:

```
1. enter 24 into the accumulator
2. add 8
3. multiply by 3
4. subtract 2
5. halt
```

This program would halt with the result 94 in the accumulator. The program could be executed by a human, activating the simple machines in sequence, or we could use a player piano–style roll of punch cards to specify the sequence of instructions, and a Jacquard loom–style mechanical reader to read them and automatically activate the corresponding simple machines in turn.

### From Calculation to Computation

To make a Church computer, it's not enough to run programs of fixed sequences of arithmetic instructions. Computation theory tells us that some functions can only be computed using decisions and jumps, so we need to add similar instructions to those of our musical processing unit, facilitating repeats, codas, and the like. This would enable programs such as:

```
1. enter 24 into the accumulator
2. add 8
3. multiply by 3
```

Computer Architecture (Sample Chapter) © 2/20/24 by Charles Fox

```
4. subtract 2
5. check if the result is less than 100
6. if so, jump to instruction 2
7. halt
```

Computation theory also tells us that some computations require memory to store intermediate results. To distinguish between these results, we'll give each value an *address*, which for now is just an integer identifier. Memory that is addressable in this way is widely called *random-access memory (RAM)*. (This is not quite the correct definition of RAM, but you'll get to that in Chapter 10.)

Having RAM available means that we can add instructions to *load* (read) and *store* (write) to addresses, as in this program:

```
1. store the number 24 into address 1
2. store the number 3 into address 2
3. load the number from address 1 into the accumulator
4. add 8
3. multiply by the number in address 2
4. subtract 2
5. check if the result is less than 100
6. if so, jump to instruction 4
7. halt
```

Computation theory tells us that we can simulate any machine if we have the three kinds of instructions I just demonstrated: those that do the actual work of the arithmetic operations; those that make decisions and jumps; and those that store and load from RAM. This is exactly how Babbage's Analytical Engine was designed.

## Babbage's Central Processing Unit

Despite its age, Babbage's Analytical Engine is a striking modern design: its basic architecture is still used in all modern CPUs. At the same time, it has only the most essential CPU features, so studying it provides a simplified introduction to the basic concepts underlying more modern CPUs. The motion of the Analytical Engine's mechanical parts also makes it easier to visualize how it works compared to today's electronic computers.

In this section I use modern terminology to describe the Analytical Engine's parts and functions. These aren't the terms Babbage used, but they'll help later when I transfer the concepts to modern machines. (Some of Babbage's original terms are included in parentheses in case they're of interest.) Babbage and Lovelace never left documentation for their instruction set, but it's been largely inferred or fantasized from other documents. I assume the instruction set and assembly language notation used by the Fourmilab emulator, an online re-creation of the Analytical Engine (*https://www.fourmilab.ch/babbage/*).

Both my presentation and the Fourmilab emulator take some liberties with the historical truth. This is easy to do because the original source documents are messy and often contradictory. There was never a single definitive design, so we can pick the versions that best suit our story. Our purpose here is really to understand *modern* CPU concepts, so I sometimes simplify, modernize, or outright lie about some of the engine's details to make this study easier.

## High-Level Architecture

The Analytical Engine consists of three things: a CPU, which executes programs; RAM, which stores data and allows the CPU to read and write it; and a bus that connects them. If that sounds similar to the overall architecture of a modern, single-core computer, that's because it is! This isn't a coincidence: the Analytical Engine's architecture was explicitly used in ENIAC (after translating its mechanics into electronics), and ENIAC then became the template for our modern electronic machines.

Physically, the Analytical Engine is made of 50 copies of the slice (what Babbage called a "cage") shown in Figure 3-4, stacked vertically, one on top of the other, as in Figure 1-14.

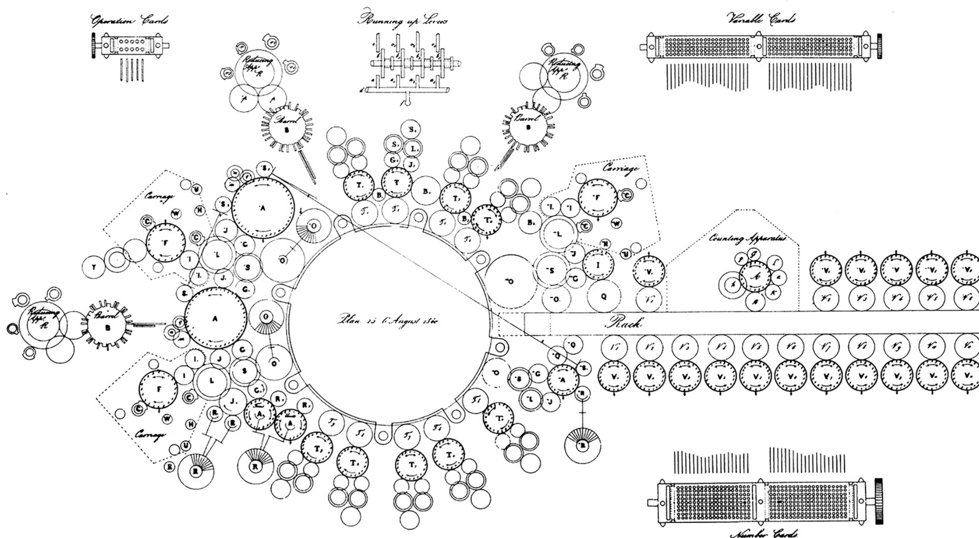



*Figure 3-4: Babbage's Analytical Engine architecture (1836)*

The circles are mechanical gears. The CPU, RAM, and bus each extend through all slices, and we can see each of them in Figure 3-4. For each number represented in each structure of the machine, the slice shows and handles one of its many digits. The stack of all the slices together handles all digits.

The RAM ("store axes") consists of 100 stacks of gears, with each stack representing one 50-digit decimal integer number. It appears on the slice as

the large homogeneous area on the right side of Figure 3-4. Each of these locations in the RAM has an address, numbered from 0 to 99 inclusive; this address distinguishes the location from the other locations and is used to identify it.

The RAM locations are all physically close to, but not usually touching, a mechanical bus ("rack"). The bus is a rack gear—exactly like the one found in modern car steering racks and LEGO Technic sets (Figure 3-5).

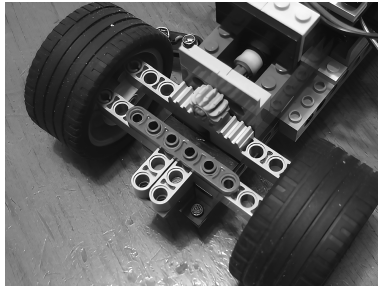


*Figure 3-5: A rack (linear gear) and pinion (rotating gear)*

The rack gear can *physically* shift left and right. Each of the RAM locations can be brought into contact with the rack by levers. The gears in that RAM location then act as pinions so that giving off the number from the location makes the bus physically shift to the left by that amount. Or, acting in the opposite direction, shifting the bus to the right from elsewhere adds numbers into the memory location.

The CPU ("mill") is the active part of the machine. It requests data from and sends data to the RAM on the bus, and then processes it in various ways.

## Programmer Interface

Unlike the Difference Engine, the Analytical Engine was designed as a general-purpose computer. This means we can ask it to perform different operations in different orders. To do this, we need a way to specify what these operations and orders are.

Let's clarify some terms I've been using loosely. An ordered list of *instructions* to perform operations is called a *program*. The act of carrying out a program is called *execution* or a *run*. The set of all available instructions is the *instruction set*.

Programs are stored as codes on punched cards, like those of the Jacquard loom seen previously in Figure 1-11. Each card contains one row of holes and non-holes, which together code for one instruction. Usually the instructions are executed in order, with the cards advancing in sequence, but some instructions make the cards rewind or fast-forward to jump around in the program. Let's look at what particular instructions are available.

### Constants

One basic instruction is to set one of the RAM addresses to a given integer. For example, "Put the integer 534 into RAM address 27." This will move the gears in the 27th RAM location's column to the (decimal) digits 534, with zeros on the gears for the thousands place and higher. Let's first denote this using a human-readable notation:

```
N27 534
```

Here, `N` (for *number*) tells us that this is a RAM integer-setting instruction. The following number (`27`) tells us which RAM location is to be set, and the final number (`534`) is the value we're setting it to. A typical program begins by setting many RAM addresses to specific values in this manner. For example:

```
N27 534
N15 123
N99 58993254235
N0  10
N2  5387
```

Once we have some starting values, we can then use further instructions to compute with them, as in the next sections.

### Load and Store

To process values from RAM, they must be moved into the CPU. To load a value from RAM into the CPU, we write `L` for *load*, followed by the RAM address where the value is stored. For example, this program sets the 27th RAM location to the value 534, then loads the value from this location into the CPU:

```
N27 534
L27
```

To store the CPU's latest result to RAM address 35, we write `S` for *store* followed by the desired address:

```
S35
```

Storing (`S`) is different from setting RAM to a constant (`N`) because it involves the CPU's accumulator. It transfers whatever value is in the accumulator to the RAM, rather than putting a fixed constant into RAM.

Now that we can move data around, we would like to perform calculations in the form of arithmetic on it.

### Arithmetic

The Analytical Engine is able to perform elementary arithmetical operations: addition, subtraction, multiplication, and division, all on integers. These are denoted by the instructions `+`, `-`, `*`, and `/`.

To do arithmetic, you first have to set the *mode*, which tells the engine which of these operations you want to do. For example, to add two numbers, you put it into adding mode and then load the two arguments in order into the CPU. Consider the following program:

```
N0 7
N1 3
+
L0
L1
S2
```

This program first puts the integers 7 and 3 into addresses 0 and 1, respectively. It then puts the CPU into adding mode with the + instruction and loads the number from these addresses. It finally stores the result of the addition into address 2.

Now that we have arithmetic, we finally need to move from calculation to computation by adding jumps and branches.

### Jumps

If you want part of a program to repeat forever, a simple method is to glue the end of the last punch card to the top of the first one to create a physical loop, as in Figure 1-15. However, this doesn't generalize well, so it's useful instead to have an instruction to rewind or fast-forward the cards to jump to any other line of the program when needed. Call this C for *control*. We'll then say whether we want to go backward (B) or forward (F) in the cards, and by how many. We'll also include the symbol + before the number (for reasons you'll see in the next section). Putting it all together, CB+4, for example, is a control instruction to go backward by four cards.

The following program uses CB+4 to loop forever:

```
N46 0
N37 1
+
L46
L37
S46
CB+4
```

Here we use address 46 as a counter, adding 1 to its value every time we go around the loop.

### Branches

Looping forever often isn't very useful; we usually want to loop *until* something has happened, then stop looping and move on to the next part of the program. This is done with a conditional *branch*, which asks whether a condition holds and jumps only if it does.

We'll use the same CF and CB notation we used for jumps, but with the symbol ? replacing the + to denote that the jump is conditional. For example, CB?4 is the control instruction to go backward by four cards only if some condition is true.

The following program uses a conditional branch and an unconditional jump together to compute the absolute value (always positive) of the sum of two numbers.

```
N1 -2
N2 -3
N99 0
+
L1
L2
S3
+
L99
L3
CF?1
CF+4
-
L99
L3
S3
```

This program uses the + instruction to add the two numbers in RAM locations 1 and 2, storing the result at location 3. It then adds zero (loaded from address 99) to that result, loaded back from location 3. Behind the scenes this addition operation also sets a special *status flag* to a 1 if the sign of the result differs from the sign of the first input (zero is considered positive). The conditional instruction (CF?1) then uses this status flag to decide what to do. If the flag is a 1, we skip over the next instruction, and so we arrive at the - instruction and perform a subtraction of the result from 0 to swap its sign. If the status flag is a 0, the conditional jump doesn't occur, so we simply move on to the next instruction (CF+4). This is an unconditional jump that skips over the four lines of subtraction code so as not to swap the sign. The final result is stored in address 3.

Branching completes the instruction set of the Analytical Engine and (assuming enough memory is always available) makes it into a Church computer. You can try tackling the end-of-chapter exercises and programming the Analytical Engine now—or, if you're interested to see how the machine works on the inside, read on.

## Internal Subcomponents

Let's look at the subcomponents *within* the CPU that are needed to execute these programs. This section describes their static structure; we'll bring the

subcomponents to life in the next section when we cover how they move and interact with one another.

A CPU is formed from many independent simple machines, each made from several number representations and the machinery that acts upon them. The simple machines are grouped into three types: registers, an arithmetic logic unit, and a control unit.

As shown in Figure 3-4, all of these simple machines are arranged in a circle around a single large gear called the central wheel. Like the bus, the central wheel makes and breaks arbitrary data connections between components, in this case between the simple machines inside the CPU. These connections are made and removed by levers that put small additional gears into contact between the central wheel and the various machines.

### Registers

*Registers* (what Babbage called "axes") are small units of memory location inside the CPU itself, rather than in the main RAM. There are only a few registers in the CPU, while there are many RAM addresses.

Recall from Chapter 2 that integers are represented in the Analytical Engine by digital, decimal gears. A digit $d$ is read off a gear by rotating a shaft by a full circle, which results in the gear rotating by $d$ tenths of a circle. To represent an $N$-digit integer, we simply stack $N$ of these gears vertically, spanning the $N$ cages of the machine. A register is one of these stacks.

The input register ("ingress axle") receives incoming data from RAM. The output register ("egress axle") temporarily stores (or *buffers*) results from the CPU's work, which are then transferred out to RAM. Other registers are used during computations for other purposes.

### Arithmetic Logic Unit

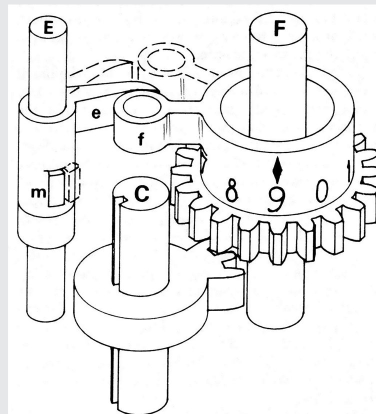
The *arithmetic logic unit (ALU)* is a collection of independent simple machines that each perform a single arithmetic operation. For example, a simple machine similar to Pascal's calculator is used to do addition. Multiplying by $m$ can be done by a machine that triggers $m$ repetitions of this adder. Multiplying or dividing by the $n$th power of 10 can be done by an especially simple machine that shifts all of its digits by $n$ columns, the mechanical equivalent of "putting a zero on the end."

In addition to sending the result to an output register, some ALU operations can also set a single *status flag* as an extra, side-effect output. The status flag in the Analytical Engine is a single mechanical lever that is in either the up (1) or down (0) position. It might have had an actual red fabric flag on it to visually alert human as well as mechanical observers that "something interesting just happened" in the ALU.

### ALU MECHANISMS

A digit *d* is given off from a gear *D* when it's read by physically rotating the gear by *d* tenths of a full circle. This digit can be added to another digit *a* stored on gear *A* by placing the gears next to one another so that their teeth mesh together, then giving off from *D*. As gear *D* rotates *d* tenths of a circle, gear *A* will be caused to rotate by the same amount, so gear *A* will end up storing the digit *a* + *d*. We say that *A* acts as an *accumulator* because we can go on adding many digits into it, and it accumulates their sum—that is, until the total goes above 9.

Integers larger than 9 are represented on stacks of gears, such as in registers. Adding them together is done similarly to adding in columns with pen and paper: the two digits in each column need to be added, but we also need to keep track of carrying when a digit goes above 9 by passing a 1 to the next column. Pascal had already developed a basic mechanical ripple carry system in his calculator, which allowed numbers to be added into an accumulator, and Babbage's carries are based on this. The following figure shows part of Babbage's design.




When a gear reaches the number 9 and is rotated by one more position in an addition, such as by an incoming carry (c), a tappet (f) connects to another tappet (e). The latter connects to a rod (m) that transfers the carry "upstairs" to the next cage, where it appears as (c) for the next column. Getting the timing right for long ripples of carries is very difficult, and this is where Babbage spent most of his design time.

#### Control Unit

The *control unit (CU)* reads instructions from the program in memory, decodes them, and passes control to the ALU or elsewhere to carry the instructions out. Then it updates the position in the program according to either normal sequential execution or a jump. The CU is like the conductor of an orchestra, coordinating the actions of all the other components at the right times. Babbage's CU is shown in Figure 3-6.

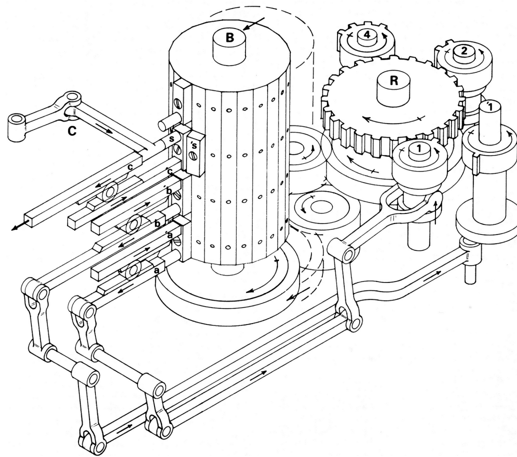



*Figure 3-6: The Analytical Engine control unit*

A mechanical barrel, just like that of a barrel organ, rotates over time, and each column of the barrel has several sockets for pins that may or may not be present. The pins trigger tappets that activate the other simple machines in the CPU through a complex system of mechanical levers. This enables each phase of the control unit's work to be triggered in sequence, much like a barrel organ playing a sequence of notes. The speed of rotation of the barrel can be controlled by feedback mechanisms, so the next step doesn't commence until the current step has been completed.

The configuration of the barrel's pins is *not* the user's program, but rather a lower-level *microprogram* that defines the sequencing of the CPU itself: the fetch-decode-execute cycle that we'll discuss next. As the microprogram runs, it causes individual commands from the user's higher-level program to be read into registers from punched cards, then causes those commands to be executed via the simple machines in the rest of the CPU.

## Internal Operation

The CU—in Babbage's case the rotating barrel—triggers a regular cycle of activities. These are usually grouped into three main stages: fetch, decode, and execute. All of the CU's operations must be carefully timed to occur in the right order. Let's look at these three stages in turn.

### Fetch

*Fetching* means reading the machine code for the next instruction into the CPU. Recall that the human-readable assembly language instructions such as `N37 1` and `CB+4` are actually represented as binary machine code on the punched cards. For the Analytical Engine, fetching could be done exactly as on the Jacquard loom, by attempting to insert a set of physical pins into the locations on the card. Where there's a punched hole, the pin can pass through, but where there isn't a hole the pin gets stuck on the card and

doesn't move as far. The physical positions of these pins can then be amplified and transmitted into the CPU by metal levers.

The card reader is a physical device, rather like a typewriter, in which there's a current line accessible to the pins. To read from any other line, it's necessary to pull the string of punch cards through this reader until the desired line is positioned in it. The current physical state of the punch cards—which one is currently in the reader—thus acts as a form of memory. We'll call this physical state the *program counter*.

The physical positions of the metal levers can also be considered as a form of memory that contains a copy of the current instruction inside the CPU. We'll call this the *instruction memory*.

### Decode

It isn't immediately obvious what the binary encodings on the punch cards mean, either to a human or a machine: at this stage, they're just patterns of 0s and 1s. *Decoding* means figuring out what this code means. The card-reading levers traveling into the CPU can activate different pieces of machinery there, depending on what *combinations* of levers are up or down. For example, if the load instruction (`L`) is represented as binary 010, a machine could be set to respond only if three fetch levers are down, up, and down, respectively. Similarly, numerical addresses included in instructions need to be decoded, from decimal codes to mechanical activations of the addresses they represent. The decoder is a bank of machines that each look for a specific pattern in the fetched signal and activate something when they see it.

### Execute: Load and Store

*Execution* means carrying out the decoded instruction. How this is done will depend on what type of instruction it is. Each form of execution is implemented by a different simple machine, and the decoder will select and activate the appropriate one.

Values can be *loaded* into the CPU registers from RAM when the CPU needs to use them—for example, as part of a calculation. The results of the CPU's work are also placed in registers, whose values can then be *stored* by copying them out to RAM addresses.

To load a value, the CU makes mechanical connections between the gears at the RAM address and the bus, and between the bus and input register at the CPU end. It then triggers a giving off at the RAM address, spinning the gears by a full circle so that they make the bus physically shift toward the CPU by *n* steps, where *n* is the digit represented. This occurs in parallel, with each column of the number having its own RAM gear, bus, and input register gear.

When a value is to be stored, the CU triggers the opposite set of steps. Storing assumes that the value to be stored is already in the output register. First, it clears the RAM at the target address by rotating all the digits to zero. Then it makes mechanical connections from the output register to the bus, and from the bus to the required address in RAM. Then it spins the output register by a full circle, which physically shifts the bus by *n* steps toward the

RAM, which in turn rotates the RAM gear by *n* steps so that the number is stored there.

### Execute: Arithmetic Instructions

When an arithmetic instruction, such as an addition, is required, the appropriate simple machine, such as an adder, is brought into mechanical contact with the input and output registers and activated. In the Analytical Engine this is done mechanically by inserting gears (cogs) that physically link the registers to the simple machine, then transmitting power to the simple machine to make it run. Babbage's adder was similar to a Pascal calculator, loading in the first argument, adding the second argument to, and then transferring the result to the output register. When the calculation is done, these gears are pulled away to disable the simple machine.

In addition to affecting the output register, the ALU's simple machines may also raise or lower the status flag if something interesting happens during the arithmetic. The different simple machines in the ALU each have their own definition of "interesting" and can each set the flag according to these interests: `+` and `-` set the status flag to true if and only if the sign of their result differs from the sign of their first input, while `/` sets the status flag to true if a division by zero was attempted.

### Execute: Program Flow

At the end of each instruction, the CU must complete the fetch-decode-execute cycle and prepare for the start of the next one. How this is done differs depending on whether we have a normal instruction (such as load and store or ALU instructions) or one whose purpose is to alter the program flow—that is, jumps and branches.

In *normal execution*, when an instruction completes, we want to advance to the next instruction in the program, which for Babbage is the one on the punch card whose top is attached by string to the bottom of the current instruction's punch card. This will prepare the system for the next fetch, which will be on the new instruction. To do this, the CU needs to trigger and increment the program counter. For the Analytical Engine, this is done by making mechanical connections that supply power to the punch card reader to perform a line feed, pulling the card deck through the reader by one card.

*Jump instructions* mean fast-forwarding or rewinding the program as requested. Consider the instruction `CF+4`, which means forward by four lines. When the CU sees this instruction, it will again modify the program counter, but rather than simply incrementing it, it will advance or rewind it by the number of lines requested. In the Analytical Engine, this is done by sending power to the line feeder for a longer time than a single line advancement, and also by mechanically switching the direction of line feed between forward and backward.

*Branch instructions* such as `CB?4` are executed differently, depending on the state of the status flag. This instruction, for example, tells the CU to jump, decreasing the program counter by four, if and only if the status flag is up. Otherwise, the instruction has no effect, and normal execution is used

to increment the program counter and move to the next instruction. This branching is the important difference that separates the Analytical Engine from previous barrel and punch card program machines such as music players and the Jacquard loom. Unless historians discover any previous machines that could do it, this engine marked the first time that a machine was designed to modify the execution of its own program rather than always follow it in the same sequence. This ability to look at the state of things and make decisions based on it is a key requirement of a Church computer.

## Summary

We've studied Babbage's Analytical Engine in this chapter because it was and still is the blueprint for all computers that came after it, including modern PCs. Its high-level architecture includes a CPU, RAM, and a bus connecting them. Inside the CPU is an ALU, registers, and a CU that conducts a fetch-decode-execute cycle. The instruction set includes load and store, arithmetic, and jump and branch instructions. There's a program counter storing the current program line number, and a status flag that gets set if something interesting happened in the latest arithmetic operation. All of these features are found essentially unchanged in a modern PC.

As a mechanical system, the Analytical Engine can be much more concrete to visualize and understand than electronics. But electronic computers are based on simply translating each of Babbage's components into a faster and smaller implementation based on electronic switches grouped into logic gates. In the second part of this book, you'll see how this is done by building up the modern electronic hierarchy from switches to CPUs. Now that you've seen what a CPU needs to do, you should have a clearer picture of where this electronic hierarchy is heading.

## Exercises

### Programming the Analytical Engine

1. Install the Fourmilab Analytical Engine emulator from *https://www.fourmilab.ch/babbage*, or use its web interface.
2. Enter and run the Analytical Engine programs discussed in this chapter. If you run the programs using the **`java aes -t test.card`** command, then the `-t` option will print out a trace of changes to the machine state at each step.

### Lovelace's Factorial Function

Write a factorial function for the Analytical Engine. Ada Lovelace wrote one of these, and it has since become the standard "Hello, world!" exercise to try whenever you meet a new architecture. (Actually printing "Hello, world!" tends to be more complicated, as it requires ASCII and screen output—you'll see how to do this in Chapter 11.)

Computer Architecture (Sample Chapter) © 2/20/24 by Charles Fox

## Further Reading

- For a more historically accurate description of the Analytical Engine, see A. Bromley, "Charles Babbage's Analytical Engine, 1838," *Annals of the History of Computing* 4, no. 3 (1982): 196–217.
- For a more fictional version, see William Gibson and Bruce Sterling, *The Difference Engine* (London: Victor Gollancz, 1990). This is the original steampunk novel, featuring Babbage, Lovelace, and a working Analytical Engine.