5. Computer Architecture: a Software Perspective

Mathematics, rightly viewed, possesses not only truth, but supreme beauty – a beauty cold and austere, like that of sculpture, without appeal to any part of our weaker nature, without the gorgeous trappings of painting or music, yet sublimely pure, and capable of a stern perfection such as only the greatest art can show.

Bertrand Russell (1872 - 1970)

5.1 The Universal Computer

It is interesting to note that the digital computer – the most useful and practical invention of modern times – was born out from an esoteric challenge in pure mathematics. In 1928, a short time before his death, the German mathematician David Hilbert posed a challenge to the scientific community: "prove or disprove that there exists an automatic process which could be applied to any mathematical statement, and which is guaranteed to produce a correct decision as to whether the statement is true." Upon hearing about Hilbert's decidability problem, the British mathematician G.H. Hardy surmised "there is of course no such process, and this is very fortunate, since if there were one, a mechanical set of rules could solve all mathematical problems, and our careers as mathematicians would come to an end."

Alan Turing, a young Ph.D. student at Hardy's Cambridge University, took the bait and started laboring on Hilbert's problem. His first step was to work out a formal definition of "automatic process". He proceeded to describe an abstract machine, equipped with an infinite paper tape that could move back and forth under a read/write head, which he called "scanner." Operating under a pre-defined set of written instructions, the scanner could move the tape, detect a mark on the tape, place a mark on the tape, or remove a mark from the tape. This trivial apparatus comprised the machine's hardware. The machine's software consisted of a new concept called programs – series of elementary instructions that told the scanner what to do in each step. Of course the terms hardware and software did not exist in the 1930's.

In the process of constructing his theoretical machine, Turing came up with three remarkable insights. First, he showed how mathematical statements and proofs could be coded as series of on/off marks written on tape. Second, he showed how the programs of his machine could also be treated as data, i.e. written as series of on/off marks committed to tape. This meant that the machine could process and analyze its own programs – a crucial insight that later became to be known as the stored program concept. Finally, in one of the most visionary and bold statements in the history of science, Turing claimed without proof that his obscure machine was as powerful as any computer that anyone will ever build.

Coming from an eccentric 22 year old mathematician who has never seen a computer (because none existed!) a full decade before the computer industry was actually born, Turing's claim to have invented the "universal computer" seemed highly preposterous, if not megalomanic. But

---

Turing’s conjecture was prophetic. Each and every computer built up to this date, no matter how powerful, fast, and fancy, can be shown to be equivalent to a Turing machine. The proof consists of showing that every elementary operation that the fancy computer is capable of executing can also be performed by a finite set of elementary steps, operating on the Turing machine hardware. It follows that if the processor of the fancy computer can run incredible virtual reality games and highbrow artificial intelligence programs, so does its equivalent Turing machine, moving its clunky tape back and forth.

Turing was the first modern thinker who understood formally what the term "to compute" means. He realized that although we can concoct highly sophisticated computer systems and programming languages, every one of them could be reduced to a (possibly very long) set of elementary symbol-manipulation steps. And these steps, in turn, could be translated to a different but equivalent set of steps that effect precisely the same result on an abstract universal computer. Said otherwise, Turing showed that in a deep theoretical sense, all computers are equivalent to each other. In so doing, he also showed that there was nothing special about his own contraption. Any machine capable of reading, writing, and detecting 0's and 1's (or any other symbols signaling two different states) under program control is as powerful as any other computer.

This was a crucial insight, in part because it said nothing about the speed of the underlying computations. Turing did not care about speed. Indeed, in spite of its historical celebrity, the Turing machine – if one would ever bother to construct it for real -- would have been pathetically slow and downright impractical. Yet in a subtle sense, the fact that Turing ignored performance is what really ignited the field. By arguing that all computers are essentially the same in terms of effectiveness, he has focused the attention of the scientific community on a new race: to build computers that will ever be more powerful in terms of efficiency.

The second great actor who entered the stage was John Von Neumann. Working in Princeton University in 1944, Von Neumann published a seminal paper titled "The Principles of Large-Scale Computing Machines." In it, he described a digital computer framework that was not only "Turing complete", but also practically workable. Remarkably, all the key elements of modern computers were present in the original Von Neumann framework: processor, registers, memory, and control. In addition, Von Neumann gave a concrete and operational meaning to the stored program concept. In particular, he showed how a program could be placed in the memory unit of the machine and manipulated by its processor – just like data.²

It's important to note that the Turing Machine was not only inefficient – it was hopelessly inefficient. That is to say, nothing in the Turing architecture gave a clue on how the machine's performance could possibly be improved. In contrast, the new machine coming from the other side of the ocean was excitingly modern, based on new technologies like electro-mechanics and relay logic. This gave engineers a lot to chew on: practically every hardware and software element in the Von Neumann architecture was (and still is) begging for improvement, and pointing the way on how it could be done. By basing its work on modern technologies, the Von

² Unlike Turing, who was a solitary worker, Von Neumann collaborated with many. The teams that built the first stored program computers included, among others, J. Presper Eckert, John Mauchly, Arthur Burks, and Maurice Wilkes, working at the University of Pennsylvania, Princeton, and Cambridge. Since he was the most prominent scientist in the group, the architecture that emerged from thses efforts is now called "the Von Neumann machine".
Neumann team has lifted the universal computer out of the realm of pure mathematics and placed it squarely in the center of the new fields of computer science and electrical engineering.

Thus, if Turing was the greatest visionary of the computer science field, Von Neumann was its greatest integrator. As of this writing, all the successful computers ever built are designed according to the Von Neumann architecture, and all of them are as powerful as the Turing machine. These commonalities are the two golden threads that run through every important development in computer science to date. For students of the field, the theoretical notion of the universal computer, on the one hand, and the dominance of one practical master architecture, on the other, are extremely important. Put simply, they imply that there is no need to learn how a specific computer works. Rather, it is sufficient to understand how a typical computer works, and, ideally, be able to build one from scratch.

This, in a nutshell, is what we are about to undertake in the next two chapters. Although chapter 5 focuses more on software and chapter 6 on hardware, it should be understood that the deeper you dig into the machine, the harder it is to distinguish between the two. Therefore, the discussion will jump back and forth between hardware and software issues and their various interfaces. In the process, we will unfold a typical digital architecture – one that illustrates all the key concepts and techniques used in the design of modern computers.

Since we will continue to explore and build the hardware of this architecture in the next chapter, we devote the present chapter to a software perspective on low-level computing. In particular, we will introduce assembly and machine languages, and guide you through the construction of an assembler program, designed to translate the former to the latter (project 5). We end the chapter with a description of a typical computer platform, called Hack. In project 6, we describe how to build the Hack platform from the combinational and sequential chips that you built in chapters 3 and 4.

5.2 Hardware: the processing architecture

Consider the computer architecture depicted in Figure 5-1. The processing centerpiece of the architecture is the Arithmetic Logical Unit (ALU) – a combinational chip capable of computing a fixed set of elementary functions, e.g. those listed in Spec. 3-12. The source data of the ALU, as well its output, are stored in short-term storage devices called registers. A minimal computer configuration uses two registers, denoted D for data and A for addressing. The D register is designed to support the ALU's calculations; the A register is designed to support the ALU's calculations and to control the operations of the memory unit. The latter device, also called Random Access Memory, or RAM, is a vector of n registers, each characterized by a unique address that runs from 0 to n-1. At any given point of time, the memory unit emits through its output port the value of its i-th memory register, denoted M, where i is the output of the A register.

The computer's ALU has two data inputs – the present values of D and M – and one control input, denoted f, where f ranges over all the possible functions that the ALU can calculate. Given this notation, the generic operation that this architecture is designed to support can be described as destination=f(D,M), e.g. D=D+M, M=D-1, or A=1 (the right hand-sides of these expressions are instances of the ALU's generic f(D,M) function, drawn from Spec. 3-12) Thus, when we want the computer to effect a certain operation, we issue an instruction that specifies (i) which function f
the ALU is to compute on the present values of \( D \) and \( M \), and (ii) where the resulting value should be stored. This destination can be either one of the \( D \), \( A \), or \( M \) registers, where \( M \) (as always) is the memory register that \( A \) presently points at.

![Diagram of computer architecture](image)

**FIGURE 5-1: the processing logic of a typical computer architecture (first approximation).**

The shown architecture leaves open the question of where the instructions come from, and how the computer knows which instruction to execute next. Instead, the figure focuses on the circuitry that supports the execution of a single given instruction. The instruction specifies the generic operation \( \text{destination} = f(D,M) \), where \( \text{destination} \) is either \( A \), \( D \), or \( M \), the latter being the memory location presently addressed by the \( A \) register. The clock regulates the pace at which instructions are issued and executed.

The state of the computer (an abstract concept) consists of four elements: the contents of the data register \( D \), the contents of the address register \( A \), the contents of the memory unit \( \text{RAM} \), and the next function that the ALU is to compute. A special hardware device, called clock, regulates the state transitions by issuing a steady train 10101010... of alternating signals. Whenever the clock falls from 1 to 0, and only then, the computer changes its state: the \( A \), \( D \) and \( M \) registers commit to their newly computed values (which, most of the time, are the same as their values in the previous state), and the ALU is issued a new instruction. The instructions are formulated according to the syntax rules of a machine language.

The remainder of this chapter takes an unusual track. Normally, machine language is presented in a bottom up fashion, specifying how various hardware devices are manipulated by low-level binary codes. Instead, we wish to start this journey at the other end of the processing hierarchy – that of high level programming. Since most readers of this book have programming experience, we feel that a top-down exploration of machine language may be more comfortable. We begin by observing that high-level languages allow us to express algorithms in a highly flexible and symbolic fashion. In contrast, the target hardware on which these programs are supposed to run
can process only rigidly structured binary codes. The remainder of this chapter seeks to give you a hands-on appreciation of what it takes to translate an abstract high-level program into a binary code that a computer hardware can actually execute. In the process, you will gain an intimate understating of "micro-programming" at the hardware level.

5.3 Software: high-level programming

The max(x-y,0) program: Consider a program that takes two inputs, x and y, and computes the value of max(x-y,0) using the following procedure: get the values of x and y, compute x-y; if the computed value is positive, return the computed value; otherwise, return 0. The returned value is stored in variable z. A high-level implementation of this function is given in Program 5-2.

\[
\begin{align*}
  z &= x - y \\
  \text{if } z > 0 \text{ goto end} \\
  z &= 0 \\
  \text{end:} \\
  \text{stop}
\end{align*}
\]

**PROGRAM 5-2:** High-level version of the max(x-y,0) program, written in a Basic-like programming style.

Let us assume that the program is supposed to run on a computer architecture equipped with a single data register, denoted D. Next, let us rewrite the program in such a way that each command in the new program mentions at most one variable. The translation and the resulting code are shown in Figure 5-3.

\[
\begin{align*}
  D &= x \\
  z &= x - y \\
  \text{if } z > 0 \text{ goto end} \\
  z &= 0 \\
  \text{end:} \\
  \text{stop}
\end{align*}
\]

\[
\begin{align*}
  D &= D - y \\
  z &= D \\
  \text{if } D > 0 \text{ goto end} \\
  z &= 0 \\
  \text{end:} \\
  \text{stop}
\end{align*}
\]

**FIGURE 5-3:** Introducing the one-variable-per-instruction constraint. In the translated program, each instruction is designed to manipulate at most two storage devices: a data register (denoted D) and a single memory location. Said otherwise, each instruction is allowed to mention at most one variable.

Since each individual instruction in the new program is allowed to access the memory unit only once, and since we assume that the hardware architecture employs only one data register, we are forced to do everything one step at a time. Thus, the high level command \( z = x - y \) is translated into
three low level instructions: (i) get the value stored in the memory location referenced by the x symbol and put it in the data register D, (ii) subtract from D the value stored in the memory location referenced by y, and (iii) store the resulting value in the memory location referenced by z.

It should be understood that variables like x, y, and z are high-level artifacts, designed to ease the life of the programmer. The computer architecture does not understand the notion of a variable. Thus, before running a high level program on a computer, the program's variables must be mapped on specific memory locations. Without loss of generality, let us assume that x, y and z are mapped on the first, second, and third locations in the memory unit. Using this mapping, we continue to reduce the program, as shown in Figure 5-4.

![Figure 5-4: Translating symbolic variables to memory locations.](image)

In the new version of the program, all the variable names were replaced with references to specific memory locations. Thus, when the translated program will actually run on the computer, instructions that refer to M[address] locations will need memory access services. In particular, the hardware will have to either fetch data values from these memory locations, or, alternatively, enable them as destinations – depending, respectively, on whether M[address] is referenced in the right hand side or the left hand side of the destination=f(D,M) instruction.

In order to set up for any one of these "fetch" or "enable" operations, the hardware must first locate the individual memory register in the RAM, using the address specified by the instruction. For example, before the ALU can compute the value of D-M[819], the value of RAM register number 819 must first be fetched and routed to the ALU's M input port. In a similar fashion, before an instruction like M[17]=D-1 can be executed, the hardware must first enable RAM register number 17 for a write operation. These fundamental data access operations, known collectively as "addressing", are discussed next.
5.4 Hardware: the data memory

The memory unit, also called RAM, is a vector of $n$ registers, each characterized by a unique address that runs from 0 to $n-1$. The RAM unit is equipped with two input ports, labeled address and data, and one output port. Due to the gate logic that governs its internal behavior, the RAM unit continuously emits the value of the memory register whose address is specified by its address input. Thus, to access a particular memory location – say address 819 – all we have to do is set the RAM's address input to 819. This setting will instantaneously cause the RAM unit to emit the value of $M[819]$ through its output port. The fetched value then becomes the $M$ input of the ALU. The anatomy of this fundamental addressing operation is depicted in Figure 5-5.

**FIGURE 5-5: The data memory unit and data addressing logic.** To access a particular memory location (for either reads or writes), we set the $A$ register to the desired address, say 819. In the next clock cycle, this setting will cause the RAM to emit the value of the RAM[819] register.

The two-stage addressing protocol described above dictates that in addition to the destination=$f(D,M)$ instruction, our language needs another instruction – one that sets the $A$ register to a certain value. Without loss of generality, let us decide that the syntax of this instruction is $@address$, implying the semantics $A=address$.

Taken together, the instructions $@address$ and destination=$f(D,M)$ give us full control over the computer's data memory. Figure 5-6 lists some examples.
\[ @17 \quad D=M \quad \text{effects the result:} \quad D=M[17] \]

\[ @151 \quad M=D+1 \quad \text{effects the result:} \quad M[151]=D+1 \]

\[ @2012 \quad M=0 \quad \text{effects the result:} \quad M[2012]=0 \]

**FIGURE 5-6: Memory manipulation examples.** The instruction syntax \( @\text{address} \) was chosen instead of \( A=\text{address} \) in order to promote the following interpretation: "At memory location 17 – let’s call it \( M \) – execute the command \( D=M^\prime \) (first example); "At memory location 151 – let’s call it \( M \) – execute the command \( M=D+1^\prime \)" (second example), etc.

To sum up, memory access is regulated exclusively by the \( A \) register: the only way to address a certain memory register is to first initialize \( A \) to hold the address of that register, and then to issue a subsequent instruction that involves \( M \) in one way or another. This programming convention leads to another reduction of our \( \text{max}(x-y,0) \) program, as given in Figure 5-7.

**FIGURE 5-7: Generating addressing commands.** Each command that references a memory location \( M[\text{address}] \) is translated into an addressing instruction \( @\text{address} \), followed by a destination=\( f(D,M) \) instruction that operates on the selected memory location. Note: the cryptic instructions marked by arrows effect branching operations, a subject that we explain later in the chapter. For now, they can be ignored.
5.5 Software: assembly programming

The code that was generated in Figure 5-7 is a typical example of an assembly language program. Assembly programs consist of symbolic representations of low-level machine commands. These commands, in turn, can be translated further into a set of binary numbers that express the same program in machine language. The assembly-to-binary translation -- a rather straightforward task -- is carried out by a text manipulation program called assembler, as we elaborate later in the chapter. Figure 5-8 shows the first translation stage of the assembler.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Assembly Code</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>@0</td>
<td>D=M</td>
<td>00 @0 D=M</td>
</tr>
<tr>
<td>@1</td>
<td>D=D-M</td>
<td>01 @1 D=D-M</td>
</tr>
<tr>
<td>@2</td>
<td>M=D</td>
<td>02 @2 M=D</td>
</tr>
<tr>
<td>@end</td>
<td>D; JGT</td>
<td>03 D; JGT</td>
</tr>
<tr>
<td></td>
<td>@2</td>
<td>04 @2</td>
</tr>
<tr>
<td></td>
<td>M=0</td>
<td>05 M=0</td>
</tr>
<tr>
<td>end:</td>
<td></td>
<td>06 @10 →</td>
</tr>
<tr>
<td></td>
<td>@end</td>
<td>07 D; JGT</td>
</tr>
<tr>
<td></td>
<td>jmp</td>
<td>08 @2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>09 M=0</td>
</tr>
<tr>
<td></td>
<td>@10</td>
<td>10 @10 →</td>
</tr>
<tr>
<td></td>
<td>jmp</td>
<td>11 JMP</td>
</tr>
</tbody>
</table>

**FIGURE 5-8:** Assembly code with symbols (left) and without symbols (right). The program that translates from assembly code to machine code is called assembler. The first stage of the translation process generates instruction numbers and replaces all the symbolic references in the code with numbers (the above program uses only one label – a typical assembly program uses many). This is done by processing the entire code twice, as explained in the figure. The instructions marked by arrows are explained in the next section.

Since the assembler can "automatically" replace symbolic references with numbers, the programmer is relieved from the tedium of remembering what various numbers mean. Clearly, it is far easier (and less error-prone) to write instructions like @end instead of @10. That said, note that some of the assembly instructions in Figure 5-8 still refer to data memory locations using physical addresses (e.g. @0 and @1). Toward the end of the chapter we show how an assembly program can be written without making any references to physical addresses whatsoever.
5.6 Hardware: instruction memory

The architecture and instructions that were discussed thus far are designed to manipulate data. We now turn our attention to the notion of control: where computer instructions come from? and how does the computer know what to do next?

As we pointed out at the beginning of the chapter, the seminal stored program concept implies that computer instructions can be coded as numbers and treated like plain data. In particular, instructions can be stored in memory and fetched from memory at will. With that in mind, we will now introduce a new memory unit, called instruction memory. The instruction memory, also called ROM, has precisely the same structure and access circuitry as the data memory (RAM). The only difference is that RAM registers can be read and written to, whereas ROM registers can only be read (ROM stands for "Read Only Memory").

The sealed status of ROM registers has attractive implications. If we could burn the program's instructions into the ROM, the program's code will be 100% protected from accidental or malicious tempering. Given the bug-ridden applications and operating systems software that computer users are routinely forced to use today, it is reassuring to know that at least some of the instructions that our computer operates are safely ensconced in a write-proof ROM chip that no program can touch. Another vivid example of the physical separation between data memory and instruction memory is video game machines. In these computers, the removable game cartridge serves as the system's instruction memory.

As a matter of convention, from now on we will assume that our computer is equipped with two memory units: RAM and ROM. Further, we will assume that the program that the computer executes is stored in the ROM unit, starting at address 0, one instruction per memory register. In order to play by the rules of this game, we have to build the architecture in such a way that when the computer starts running, the hardware will fetch the instruction located in ROM[0] and route it for execution by the ALU.

Which instruction should be executed next? Before we answer this question, we have to introduce the notion of low-level branching commands. Every assembly language features a set of branching commands, designed to effect low-level versions of conditional and unconditional goto operations. In the assembly language presented here, we implement branching commands as optional clauses of the destination=f(D,M) instruction. Specifically, from now on, we change the syntax of this instruction to destination=expression;jump, where =expression is the same as =f(D,M), and jump ranges over 8 possible branching commands. In the new syntax, =expression is a mandatory clause of the instruction, while destination and jump are optional clauses of the instruction.

To illustrate, consider the instruction =D;JGT (line 7 in Figure 5-8), in which the destination clause is missing, the expression clause is D, and the jump clause is JGT (this mnemonic stands for "jump if greater than zero"). This instruction has the following interpretation: if the expression D (ALU result) is greater than zero, fetch and execute the instruction that the A register points at; otherwise, fetch and execute the next instruction in the program. In a similar fashion, the JMP instruction (line 11 in Figure 5-8) codes an unconditional goto operation. This instruction tells the hardware to fetch and execute the instruction that the A register points at, irrespective of any conditions. (In keeping with the destination=expression;jump syntax, we treat
the JMP instruction as shorthand to the instruction =noOp;JMP, where the =noOp clause means that we don't care about the ALU result.)

Thus, by using pairs of addressing and branching instructions, the programmer can cause the hardware to go to, and execute, any desired section in the program's code. Figure 5-9 illustrates how such instructions can implement typical conditional and unconditional branching operations.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Effect the result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td>@10</td>
<td>A=10; if D&gt;0 fetch and execute instruction #10; else fetch and execute instruction #8</td>
</tr>
<tr>
<td>07</td>
<td>=D; JGT</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>@23</td>
<td>A=23; If D-1 ≥ 0 fetch and execute instruction #23; else fetch and execute instruction #57</td>
</tr>
<tr>
<td>56</td>
<td>=D-1; JGE</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>@20</td>
<td>A=20; fetch and execute instruction #20</td>
</tr>
<tr>
<td>31</td>
<td>JMP</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>@10</td>
<td>A=10; fetch and execute instruction #10</td>
</tr>
<tr>
<td>11</td>
<td>JMP</td>
<td>(infinite loop)</td>
</tr>
</tbody>
</table>

**FIGURE 5-9: Branching instructions.** The top two examples implement "conditional goto" operations. The bottom two examples implement "unconditional goto" operations. The last example can be used as a low-level implementation of a high-level STOP command (since a powered computer that finishes to run a program never stops to do something, it might as well do something that we control).

The program counter: We see that in every clock cycle, the control logic must deal with two tasks: (i) determine which instruction to execute next, and (ii) fetch the instruction. The first task is carried out as follows. If the current instruction implies a materialized jump, e.g. the instruction is =D; JGT and the data register happens to be positive, the control logic sets a certain flag bit – let's call it jmp -- to 1. Otherwise jmp is set to 0, signaling a no-jump situation. Clearly, if the instruction involves no jump, e.g. D=D+M the jmp bit is set to 0.

Having taken care of the first task (determining the jump status), the second task (fetching the right instruction) is delegated to a special hardware device called Program Counter. The details are given in Figures 5-10 and 5-11.
FIGURE 5.10: The instruction memory and the Instruction fetching logic. The program counter determines the address of the next instruction to fetch by consulting the jmp bit. This control bit, in turn, is effected by the jump clause of the current instruction and by the current ALU result (not shown in the figure). If jmp=0, the PC causes the ROM to emit the next instruction in the program. If jmp=1, the PC causes the ROM to emit the instruction stored in the address that A points at. This behavior is accomplished by the internal logic of the PC unit, described in Figure 5-11.

FIGURE 5.11: The instruction fetching logic (state chart). When the computer starts running, the PC is initialized to 0. This causes the hardware to fetch and execute ROM[0], the first instruction in the program. As long as jmp=0, the PC acts as a counter, causing the ROM to emit the next instruction in the program in each clock cycle. When a materialized jump is detected (e.g. @gotoAddress followed by =D,JGT, and D happens to be positive), jmp becomes 1. This causes the logic to switch to the jump state, where the PC is reset to the value of the A register. As a result, the ROM will emit the instruction stored in ROM[gotoAddress]. After this instruction is fetched, the logic returns to the no-jump state. This causes the hardware to continue to march through the program's code, beginning at address gotoAddress+1, until the next jump, if any.
Chapter 5: Computer Architecture: a Software Perspective

The PC is a simple counter. Beginning with a base of 0, it increments its output by 1 in every clock cycle. When a jump operation is mandated, the counting base is reset to the value stored in the A register, and the counting resumes from this point onward. Remarkably, this straightforward scheme is all that is takes to implement the flow of control of even the most complex and exotic computer programs, including so called "artificial intelligence" software. It is therefore not surprising that people who have an intimate understanding of digital hardware are rather skeptical about the claims that computers will eventually out-smart the human brain. There must be something more exciting than a program counter humming at the seat of our intelligence.

5.7 Software: the Instruction Set

The design of a new computer platform is not complete until the hardware designer specifies an instruction set for the target platform. For each elementary operation of the machine, the designer specifies and documents a certain binary code, designed to effect the desired operation on the hardware. This code, called a machine instruction, is a structured collection of control bits, organized in fields. Each field is a packet of control bits destined to control a certain hardware device, e.g. the ALU or the RAM. To program the computer hardware at the lowest level possible, one writes a series of such binary instructions and puts them in the instruction memory, starting at address 0. The result is a machine-level program that can then be executed directly by the hardware.

With that in mind, executing an instruction is essentially a matter of (i) unpacking the instruction into the underlying control-bit fields, and (ii) routing the control bits to their intended destinations throughout the architecture, where they end up controlling and enabling the operations of the computer's processing, storage, and control devices. For example, in the Hack computer platform described later in this chapter, the instruction "1111011111001000" contains bits that control the ALU, the D register, the A register, the presently addressed RAM register, and the program counter PC. Taken together, this particular 16-bit pattern happens to code (and, in fact, implement) the following operation: "compute the value of D+1, store the result in the memory location addressed by A, and proceed to fetch the next instruction in the program." In short – "carry out the operation M=D+1"

Clearly, it is much easier to say "execute M=D+1" than "execute 1111011111001000". Therefore, in addition to the instruction set, the computer designer also specifies a set of symbolic codes, also called mnemonics, designed to make low-level programming more accessible to human programmers. Using these mnemonics, the designer can abstract the machine further by formulating an assembly language. The key requirement here is that each assembly command will lend itself to an unambiguous and straightforward translation into one or more machine level instructions that effect the desired operation on the target hardware. Figure 5.12 illustrates this lingual relationship in the Hack platform.
We have already seen several examples of the Hack assembly language in this chapter. To reiterate, the workhorse of this language, called the "compute-store-jump" instruction, has the following syntax:

General: \( \text{destination} = \text{expression} ; \text{jump} \)

Example: \( D = D-1; \text{JNE} \)

The destination and the jump fields are optional, whereas the \( = \text{expression} \) field is mandatory. Taken together, the three fields (in this example) tell the hardware to (i) have the ALU compute the value of \( D-1 \), (ii) store the ALU result in \( D \), and (iii) if the ALU result is not zero, effect a goto operation.

The Hack hardware and instruction set feature 8 possible destination directives, 30 possible ALU directives, and 8 possible jump directives, yielding a total of \( 8 \times 30 \times 8 = 1920 \) possible variants of the \( \text{destination} = \text{expression} ; \text{jump} \) instruction. Four of these variants are illustrated in Table 5-13.
<table>
<thead>
<tr>
<th><strong>machine instruction</strong></th>
<th><strong>assembly command</strong></th>
<th><strong>resulting actions</strong></th>
</tr>
</thead>
</table>
| 1111 0111 1101 0000     | D=D+1                | - ALU computes the value of \(D+1\)  
- ALU result is stored in \(D\)  
- PC is set to 101 (no jump) |
| 1111 1100 0100 1000     | M=M-1                | - ALU computes the value of \(\text{RAM}[500]-1\)  
- ALU result is stored in \(\text{RAM}[500]\)  
- PC is set to 101 (no jump). |
| 1111 0011 0100 0101     | =D-1; JLT            | - ALU computes the value of \(D-1\)  
- ALU result is not stored anywhere  
- If the ALU result is less than 1, the PC is set to 500 (jump); otherwise, the PC is set to 101 (no jump). |
| 1111 0000 0000 0111     | JMP                  | - PC is set to 500 (jump) |

**TABLE 5-13: Machine and assembly instructions on the Hack platform.** We assume that before each instruction is executed, \(\text{PC}=100\) and \(\text{A}=500\). The roles of the addressing register and the program counter in effecting branching operations is explained in section 5.6. The exact relationships between the symbolic commands and the binary codes are defined in section 5.9.

Since this chapter focuses on a software perspective of computer architecture, and since we will explain how binary codes are used to control hardware components in the next chapter, we will not delve here into the cryptic meaning of the binary codes in Table 5-13. For now, suffice it to say that every assembly command can be translated unambiguously into a binary code that effects the desired operation on the target hardware.

This observation brings us to the last station in our descending journey from the high-level terrain of Basic-style programming down to the very bowels of the computer: we are ready to translate the last assembly version of the \(\text{max}(x-y,0)\) program into machine language. The resulting code is shown in Figure 5-14.
5.8 The Assembler

Machine code is "for real". As we will see in the next chapter, an instruction like "1110111100110101" actually causes the hardware to "do something." In contrast, any code above machine language in the software hierarchy – beginning at the assembly language level – is an abstraction.

Specifically, high-level symbolic languages exist in order to make the life of the programmer easy. As far as the computer is concerned, any language other than machine language is plain gibberish. There is nothing in the computer hardware that knows what to do with a command like D=M+1. More to the point, nothing in the architecture is prepared to deal with symbols other than 0 or 1 to begin with. This is true for all computers, including the chips that run cellular telephones and the computers that control the flight of the space shuttle.

Therefore, before an assembly program can be executed on the computer, it must be translated to the instruction set of the target hardware. This is done by a program called assembler. The assembler takes as input a stream of assembly commands, and generates as output a stream of binary instructions. The latter can then be loaded into the computer's instruction memory and executed by the hardware.
Note that the assembler program can be written in *any* programming language. Likewise, the assembler program can run on *any* computer, not necessarily the target hardware. For example, it is a common practice to develop programs for cellular telephones on a personal computer, and then port the generated binary code into the telephone's instruction memory. To make life even easier, one can write a program that emulates the cellular phone's hardware on the personal computer. This way, the programmer can not only develop, but also test, the cellular phone's software on the personal computer. This cross-platform flexibility is precisely what Turing had in mind when he predicted 70 years ago that all computers will be essentially the same.

We see that the assembler is essentially a text-processing program, designed to provide translation services. The programmer who is commissioned to write such an assembler must be given the full specifications of the instruction set of the target hardware, on the one hand, and the agreed-upon syntax of the symbolic assembly commands, on the other. Following this contract, it is not hard to write a program that carries out the following operations (not necessarily in that order):

- Replace all symbolic references with addresses of memory locations (e.g. Figure 5-8)
- Parse the assembly commands into their underlying fields (e.g. Figure 5-12)
- For each field, generate the corresponding bits in the machine language
- Assemble the control bits into machine instructions

Project 5 provides step-by-step guidelines for building an assembler for the Hack computer. We now turn to specify the machine and assembly languages of this platform.

### 5.9 The Hack Assembly and Instruction Set

Up until the previous section, the description of computer architectures and their related low-level languages was given without constraining ourselves to any particular "computer brand." Although we have laid out a particular architecture and described a particular assembly language, similar hardware and software artifacts are used in practically every modern computer today. With that in mind, it is now time to put this general knowledge to the test, roll our sleeves, and build one specific computer from the ground up.

Before we delve into this elaborate construction project, we need clear architectural plans. To begin with, we will now provide a complete specification of the assembly and machine languages of the Hack platform. These specifications provide all the information that you need in order to write the assembler described in Project 5. Having gained a hands-on understanding of how low-level programs work, you will be ready to undertake the construction of the target hardware itself. This will be done in the Project 6.

Hack is a 16-bit computer. All its storage devices (RAM, ROM, and registers) are 16-bit wide, its ALU is designed to carry out 16-bit arithmetic, and its instructions are each 16 bits wide. The Hack platform differs from the examples that we have seen thus far in the chapter in two respects. First, in the Hack platform, the result of the ALU can be stored simultaneously in more than one location (e.g. \( DM=M-1 \) is a legitimate instruction). Second, the \( M \) input of the ALU can be replaced with \( A \). Put simply, for every ALU operation \( f(D,M) \) operating on \( D \) and \( M \) there is an
equivalent \( f(D,A) \) operation operating on \( D \) and \( A \). This way, the \( A \) register doubles as both an address register and a general-purpose 15-bit data register (the register is actually 16-bit, but the MSB is not used). These modifications make the Hack platform more powerful and easier to program.

### 5.9.1 The Compute-Store-and-Jump Command

The *compute-store-jump* command is the workhorse of Hack programming -- the command that gets almost everything done. The command is a specification that answers three questions: (a) which expression should the ALU compute? (b) where to store the computed value? and (c) what to do next? Along with the addressing command, these specifications determine all the possible operations of the computer. The command’s assembly and 16-bit syntax is as follows:

**Assembly:**
\[
\text{dest}=\text{exp};\text{jmp} \quad (=\text{exp \ is \ mandatory; \ dest \ and \ jmp \ optional})
\]

**Binary:**
\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 1 & a & e & e & e & e & d & d & d & j & j & j \\
\end{array}
\]

- \( e \ e \ e \ e \ e \): expression specification \((\text{exp})\)
- \( d \ d \ d \): destination specification \((\text{dest})\)
- \( j \ j \ j \): jump condition specification \((\text{jmp})\)
- \( a \): A/M source flag \((\text{part of exp})\)

**SPEC 5-15:** The *compute-store-and-jump* instruction. The instruction code (MSB=1) instructs the computer to treat the 16-bit word as a *compute-store-jump* instruction consisting of the binary fields documented above. The two bits following the MSB are not used.
Expression specification (exp field): The Hack ALU can calculate a pre-defined set of 30 possible arithmetic/logical operations. To select a particular ALU operation, one sets the eeeeee bits and the A/M flag bit according to Spec. 5-16.

<table>
<thead>
<tr>
<th>exp</th>
<th>e</th>
<th>e</th>
<th>e</th>
<th>e</th>
<th>e</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Not D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Not M</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>-D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-M</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D+1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>M+1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D-1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>M-1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D+M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D-M</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>M-D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D and M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D or M</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D nand M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A/M flag = 0

**SPEC 5-16**: The "compute" specification of the compute-store-jump instruction. The ALU can effect 30 possible calculations. The selected calculation is coded by setting the instruction's eeeeee bits and the A/M flag bit according to the table. The exp columns give the equivalent assembly mnemonics.
Destination specification (dest field): The ddd bits specify where to store the ALU result. The first and second d bits code whether to store the computed value in the A register and in the D register, respectively. The third d bit codes whether to store the computed value in the RAM location currently addressed by the A register. The codes are given in Spec. 5-17.

<table>
<thead>
<tr>
<th>d</th>
<th>d</th>
<th>d</th>
<th>mnemonic</th>
<th>destination (of the ALU result)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>=</td>
<td>the result is not stored anywhere</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>=M</td>
<td>RAM register addressed by the A register</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>=D</td>
<td>D register</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>=DM</td>
<td>D register and RAM register addressed by A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>=A</td>
<td>A register</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>=AM</td>
<td>A register and RAM register addressed by A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>=AD</td>
<td>A register and D register</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>=ADM</td>
<td>A register, D register, and RAM register addressed by A</td>
</tr>
</tbody>
</table>

**SPEC 5-17**: The “store” specification of the compute-store-jump instruction. The ALU result can be simultaneously stored in one, two, three, or no registers.

Jump specification (jmp field): The question of which instruction should be executed next is determined by inspecting the current instruction and the current ALU result. If the current instruction includes a jump specification, and if the ALU result satisfies the jump condition, the hardware will fetch and execute the instruction located in the ROM register addressed by the A register. Spec. 5-18 lists the jump specifications featured by Hack.

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
<th>j</th>
<th>mnemonic</th>
<th>meaning (“jump” = “jump to the instruction addressed by the A register”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(null)</td>
<td>no jump</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>JGT</td>
<td>if the ALU output is greater than 0, jump</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>JEQ</td>
<td>if the ALU output is 0, jump</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>JGE</td>
<td>if the ALU output is greater than or equal to 0, jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>JLT</td>
<td>if the ALU output is less than 0, jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>JNE</td>
<td>if the ALU output is not 0, jump</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>JLE</td>
<td>if the ALU output is less than or equal to 0, jump</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>JMP</td>
<td>unconditional jump</td>
</tr>
</tbody>
</table>

**SPEC 5-18**: The “jump” specification of the compute-store-jump instruction. The control logic examines the jjj bits of the current instruction and the current ALU result. This information, along with the ALU result, determines the behavior of the program counter, thus effecting which instruction will be fetched next.
5.9.2 The Address Instruction

The address instruction \( \text{@address} \) is used to set the \( A \) register to a given value. The instruction serves three purposes. First, it provides the only way to enter a constant into the computer without using the memory. Second, it is used as a preamble to a subsequent \textit{compute-store-jump} instruction that mentions a memory location \( M \). Third, it is used as a preamble to a subsequent \textit{compute-store-jump} instruction that includes a jump directive. The instruction syntax is given in Spec. 5-19.

**Assembly:**

\[ \text{@address} \text{(constant between 0 and } 32767 = 2^{15} - 1) \]

**Binary:**

\[
\begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\
\end{array}
\]

**SPEC 5-19:** The address command. The instruction code (MSB=0) causes the architecture to store the 15-bit value xxxxxxxxxxxxxxx in the \( A \) register. The instruction effects the operation \( A=xxxxxxxxxxxxxxxxx \).

**Symbolic addressing:** One of the most useful features that assembly language offers (in contrast to machine language) is the ability to refer to memory locations through symbolic labels rather than physical addresses. Thus, in addition to the \text{@address} instruction where \text{address} is a numeric constant, the assembly language also features the instruction \text{@label}, where \text{label} is a textual symbol. In the latter case, the contract is such that somewhere in the same program there is another assembly instruction of the form "\text{label": that gives meaning to the \text{label} symbol. Program 5-20 gives an example of symbolic references in action.

**FIGURE 5-20:** Assembly code with symbols (left) and without symbols (right). This code segment effects the following logic: IF \( D<1 \) set \( \text{RAM[100]=1} \) else set \( \text{RAM[100]=0} \). The example shows how the assembler converts symbolic addresses to actual instruction numbers.
Instruction set restrictions: In the Hack platform, a single addressing register is used to address both the data memory unit and the instruction memory unit. In some occasions the programmer uses A to select a specific RAM register for a subsequent ALU operation involving M, whereas in other occasions A is used to select a specific ROM register for a subsequent jump instruction. Hence, to prevent conflicting usage of the A register, a compute-store-jump instruction that specifies a jump should not mention M in its the expression or destination parts, and vice versa: a compute-store-jump instruction that mentions M in its the expression or destination parts should not have a jump directive. Violations of this rule will cause the computer to do something unexpected. This restriction cuts down the Hack instruction set from 1920 possible variants to 352 instruction variants that make sense.

5.10 Examples

One limitation of all the assembly programs that we have seen thus far is that they refer to data memory locations using physical addresses, e.g. @0, @1, etc. It would have been far better if assembly programmers could refer to variables using symbols, e.g. @x, @y, etc., and have the assembler map these references on actual memory locations. As it turns out, this is very simple to implement. First, we have to do decide on a base address b in which we want to start storing variables in the RAM. Next, we have to write the assembler in such a way that whenever it encounters an "@label" instruction that does not have a corresponding "label:" instruction elsewhere in the code, it concludes that label must be a variable name and thus it maps this variable on the next available RAM slot, starting at address b. Thus, if the program mentions the unresolved labels x, y, and z (in that order), the assembler will translate these references to RAM addresses b, b+1, and b+2, respectively.

Without loss of generality, we can assume that b=0. However, it should be noted that the assembly programmer could not care less. As far as the programmer is concerned, it doesn't matter if her variable x is mapped on RAM address 0, 17, or 34287. The only relevant requirement is that whichever is the allocated address of x, it will always be translated to that address. Once again, this service can be easily incorporated into the assembler.

The max(x-y,0) program: Assuming that our assembler knows how to resolve symbolic variable names, Program 5-21 gives the last and most refined assembly version of the max(x-y,0) program.
PROGRAM 5-21: The max(x-y,0) program. This version of the program is more powerful than previous versions in two respects. First, the code is easier to read and maintain. Second, since it makes no references to physical memory addresses, the program is relocatable – it can be loaded into, and ran off from, any base address in memory, having the assembler make all the necessary translations. This is a necessary feature for computers that multi-task between several programs that co-exist in memory simultaneously.

An array processing program: Program 5-22 illustrates a typical step-wise manipulation of an array of numbers, implemented as a continuous block of RAM cells. The goal is to set all the cells of the array to a certain constant. Thus, the program’s inputs are the array’s base address b, the number of array elements n, and the constant value y. The program implements the equivalence of the C code \( \text{WHILE} \ (\neg n > 0) \ x[n] = y \).
// data: b is a 3-cells array, starting at RAM address 9000
@9000    // b=9000
D=A
@b
M=D
@3     // n=3
D=A
@n
M=D
@1     // y=1
D=A
@y
M=D

// program: set all the cells of the array b to the value of y.
loop:
    @n     // let n=n-1 and D=n
    DM=M-1
@end    // if n<0 goto end
=D; JLT
@b     // make p point to b[n],
D=M     // by setting p=b+n
@n
D=D+M
@p
M=D
@y     // D=y
D=M
@p     // b[n]=D
A=M
M=D
@loop  // loop to process the previous b[n]
jmp
end:    // stop
@end
jmp

PROGRAM 6-11: An array processing program, implementing the equivalent C
code WHILE(--n>=0) x[n]=y.
5.11 Postscript: the Halting Problem

We began this chapter by observing that the modern computer was invented as a side effect of Turing's work on Hilbert's decidability problem: "prove or disprove that there exists an automatic process which could be applied to any mathematical statement, and which is guaranteed to produce a correct decision as to whether the statement is true." For the sake of history, it is fitting to end this chapter with the result of Turing's work on this problem.

To make a long story short, Turing established that such a process does not exist. His line of attack was sophisticated and indirect. After describing the "hardware" of his machine and noting that it could be used to read its own programs as data, Turing proceeded to describe the following Halting Detector program:

The Halting Detector program: "When given any program \( P \) as input, the Halting Detector program \( H \) analyzes and establishes one aspect of \( P \)'s expected runtime behavior, as follows. If \( H \) concludes that \( P \) contains an infinite loop, \( H \) prints the message "\( P \) never terminates"; otherwise, \( H \) prints the message "\( P \) terminates".

Two critical points are in order here. Clearly, it is quite easy to determine weather some programs have a finite running time. The trick is to be able to do so for any given program. Second, it is not allowed to simply run the given program \( P \) on the machine and wait to see what happens. Of course if the program will terminate its execution after a while, we could announce with certitude that indeed it terminates. Yet if the program will continue to spin its wheels throughout our lifetime, we will not be able to determine that it won't stop at some point in the future. Therefore, "proving by trying" will not work here.

As it turns out, determining the runtime behavior of certain programs is one of the deepest and most profound questions in computer science. For example, consider the following program \( P \):

\[
\text{while not(x=1) do \{if x is odd let x=x/2 else let x=3*x+1\}}
\]

Over the years, people tried to run \( P \) on many positive integers \( x \), and the while loop always terminated, pushing \( x \) through all sorts of long, convoluted and unexpected sets of values. Yet – and that's the crux of the argument -- no one succeeded to prove that the \( P \) stops on any given input \( x \).

In principle, this lack of proof is quite troubling. If \( P \) were part of a mission-critical system that regulates the operations of a nuclear plant, where \( x \) can assume arbitrarily small or large values, we would never be able to say with confidence that the plant will always function properly. We could only make an educated guess. Given that programs like \( P \) lurk in the background of almost every aspect of modern life today, the thought that no one can guarantee their proper behavior under all circumstances is rather chilling. Of course if Turing's run-time detector program \( H \) were available, we could simply feed \( P \) into it, sit back, and wait to see what's the verdict on \( P \)'s runtime behavior. Yet Turing showed that a program like \( H \) could not be written. More accurately, he showed that the existence of this program is inconsistent with the notion of a computer designed to run such programs.

How is Turing's halting detector program \( H \) related to proving mathematical conjectures? A mathematical conjecture is an unproved claim believed to be true because up to this point many people have tried and failed to show that it's false. In 1742, for example, Christian Goldbach claimed without proof that every even number greater than 2 is the sum of two prime numbers
(e.g., 8=5+3, 32=19+13, 96=89+7, etc.). For over two and a half centuries, many people (and computers) have tested Goldbach’s conjecture on an astronomical number of even numbers, and every one of them had the Goldbach property. Yet in spite of all these empirical experiments, no one managed to prove the general rule using logical arguments. Goldbach’s conjecture remains one of mathematics’ most celebrated open problems.

It turns out that if Turing’s run-time detector program $H$ were available, we could automatically prove or disprove the Goldbach conjecture, as follows. Let $G$ be a program that starts by setting $x$ to 4, and then entering the following loop. In each iteration, $G$ inspects all the pairs of numbers $a,b$ for which $a+b=x$. For each such pair, $G$ checks if both $a$ and $b$ are primes. If a prime pair is found, the program rests the case for $x$ and loops back to inspect the next even number $x=x+2$. Otherwise, the program prints the message “Goldbach’s conjecture is false: $x$ is a counter example” and terminates the loop. Now, if Turing’s run-time detector program $H$ were available, we could feed it with $G$ as input, sit back, and wait for the verdict. If $H$ would print “$G$ never terminates”, this would amount to proving Goldbach’s conjecture. If $H$ would print “$G$ terminates”, this would amount to disproving Goldbach’s conjecture.

In addition to paving the way to the invention of modern computers, the Halting Problem had profound implications on many fields. First, Turing showed that computers cannot outsmart the human mind in at least one area in which we take special pride: proving mathematical assertions. This finding led to a collective sigh of relief in the quarters of mathematics, philosophy, theology, and metaphysics (the Halting Problem may well have been the only thing that ever united these fields). Second, Turing showed that computers cannot be used to determine with 100% confidence that software programs will always “behave well” under all circumstances. This gave rise to a thriving area of research and practice known as program verification, in which computer scientists investigate the types of programs whose behavior can be analyzed and predicted with certainty or close to certainty, using logical methods.

In closing, it is tantalizing to see how much mileage has been gained from one simple problem, clearly stated and analyzed by a great mind of Turing’s caliber. As the saying goes, “invent a better mousetrap, and the world will beat a path to your door”. To this date, Turing’s Halting Problem remains one of the most effective mousetraps in the history of science.