Chapter 3
The CPU and Memory

Figure 3-1
John von Neumann
Los Alamos National Laboratory

The defining component of any computer system is the **central processing unit**, or **CPU**. The design and architecture of the CPU determine what instructions are available for programming it, how fast it will be able to run, and how data are organized and addressed. The CPU receives data and instructions from memory, performs arithmetic or logical operations on the data, and returns the data to memory. The CPU can perform billions of such operations every second. Even though each operation is relatively simple, the CPU can perform complex actions very quickly by combining those relatively simple operations.

The first commercially available CPU, that of the UNIVAC I computer delivered to the Census Bureau in 1951,was about 14 feet long, about eight feet wide, and over eight feet tall. By 1970, the progress of Moore's Law reduced the CPUs of some computers to a single circuit board. In 1971, Intel introduced the first CPU on a chip, intended for use in electronic calculators and incorporating about 2,250 transistors. Modern CPU chips have tens of billions of transistors. In the late 20th century, chips with multiple processing units were introduced. When a chip has more than one processing unit, the processing units are called **cores**.

Also in the late 20th century, supporting circuitry like memory and bus controllers began to be added to CPU chips. These evolved to become systems on a chip (SOCs). Today, CPU chips with external supporting circuitry are used where performance is the major factor, such as in servers and workstation computers. SOCs are used where small size and low power consumption are driving forces, such as in phones and tablet computers.

3.1 The von Neumann Architecture

John von Neumann was a Hungarian-born American mathematician and mathematical physicist. He made numerous contributions to mathematics, physics, and game theory. His most important contribution was a design for a programmable electronic computer.

Von Neumann suggested the use of binary numbers in computers, but his contributions in the 1945 *First Draft of a Report on the EDVAC* [36] went much further. The *First Draft* described a binary, sequential, stored program computer with a memory that held both instructions and data. Although other computer architectures have been tried, every commercially successful computer has used this design. That is why the design has come to be called the von Neumann computer architecture.

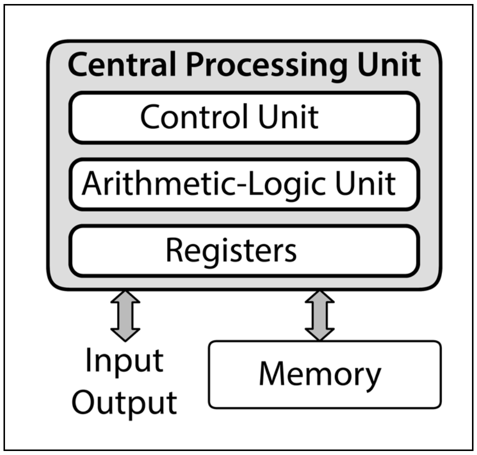

Figure 3-2
Organization of the von Neumann architecture

Figure 3-2 shows the organization of a von Neumann architecture computer. The registers are fast memory that are a part of the CPU. Typically the registers are the same size as the word size of the computer. Word size is discussed in Section 3.5.2. The arithmetic-logic unit performs, as you might expect, arithmetic and logical operations on data. It is made of combinational logic circuits like those discussed in Chapter 2. The control unit generates the digital logic signals the cause the CPU to carry out its operations. Not shown in the diagram, but present in the *First Draft*, is a system clock that generates a regular stream of pulses that synchronize the operation of the CPU.

36 "Electronic Discrete Variable Automatic Computer." Although only von Neumann's name appeared on the report, several others contributed to the ideas it contained.

The instructions that a computer can perform are stored in the computer's memory as numbers. A part of the number, called the **operation code** or op code, tells the control unit what operation is to be performed. Other parts, called **operands**, tell what data should be used in the operation. At first these were written by hand. It wasn't long before a notation more easily used by humans was developed, and the translation to numbers was performed by a computer program. Those numbers are the machine language of a computer. The idea of translation by a program led to the high-level programming languages of today.

The **program counter** is a fundamental concept in the von Neumann architecture. The program counter does not "count" anything. Since both instructions and data are held in the same memory, there must be a way to keep track of the flow of instructions. *The program counter holds the address of the next instruction.* Since the instructions are executed in sequence, often the next instruction will be the one at the next sequential memory location. The execution of an instruction can modify the program counter when it is necessary to change the flow of the program.

3.1.1 Instruction Set Architecture

You are no doubt aware of different types of CPUs. A desktop or laptop computer might have an Intel CPU while a phone or tablet might have an ARM CPU. In 2020 Apple Computer announced that they would begin using Apple-designed CPUs in Mac computers. These different types of CPU cannot run the each other's programs because they use different machine languages.

Every computer design has some number of basic machine language operations it can perform. The repertoire of basic operations for a computer is called its **instruction set**. The number of instructions in an instruction set can range from about a dozen to several hundred, depending on the complexity of the CPU.

In the early history of electronic computers, every new computer had a new instruction set. Buying a new computer meant rewriting software. That was tolerable when the number of computers and their collections of software were small. By 1960, government, business, and universities has amassed a large number of software applications, and the thought of rewriting all that software was daunting. Computer upgrades were postponed because of the cost of converting software.

In 1964, IBM made an extraordinary promise with the announcement of its System/360 line of computers. They promised potential buyers that, if the software were rewritten one more time, for the System/360 instruction set architecture, it would never have to be rewritten again. [37] Although it seems obvious to us now, it was IBM's innovation that led to "families" of CPUs that are compatible at the level of binary machine instructions. The newest Intel CPU chips of the 21st century will run programs written for the Intel 80386, announced in 1985.

A family of computers has a similar instruction set, although newer members of such a family may have instructions not available in older implementations. This is called **backward compatibility**. A newer member of a family will run anything that older members of the same family will run, but the reverse is not true. Older members of a family of computers eventually become obsolescent, but the software written for them does not.

3.1.2 Microarchitecture

Newer computers can run the same software as their older counterparts because they share the same instruction set architecture. They often provide improved performance because the underlying hardware is faster. The extra speed comes not only from faster transistors in the CPU, but from improvements in the CPU design. The design of a CPU is called its **microarchitecture**. Although the functions of a particular computer family may be constrained by the instruction set architecture of that family, the chip designer is free to use the newest innovations in CPU design. Such innovations might include instruction pipelining or the inclusion of more functional units. These are discussed in Section 3.6. We

37 Not only was IBM's promise extraordinary, they did an extraordinary job of keeping that promise. IBM zEnterprise mainframes of the 21st century will run programs written for the System/360 of 1964.

will shortly examine a design for a control unit that is fast but relatively expensive because of the number of gates needed, and remark that there is a way to build a slower but less expensive device that will do the same work.

3.2 The "Little Man" Computer

The "Little Man Computer," or LMC, devised by Stuart Madnick [38] in 1965, imagines a computer as a room with mailboxes representing memory and a calculator to perform arithmetic. The calculator and mailboxes are manipulated by a hypothetical "little man" who can perform only a very few, very limited operations. This model allows teaching the principles of a von Neumann architecture computer while abstracting away most of the details. Several people have developed LMC emulators that allow students to write and run programs in the LMC's language.

LMC has 100 mailboxes (memory locations), numbered zero to 99, each of which can hold three decimal digits. a calculator that can hold a single result, and a two-digit counter, the **program counter**, that can be advanced by one or set to an arbitrary value by the little man. The program counter can also be reset to zero from the outside the computer; this action starts or re-starts the computer. The little man of LMC performs the actions that result in computation.

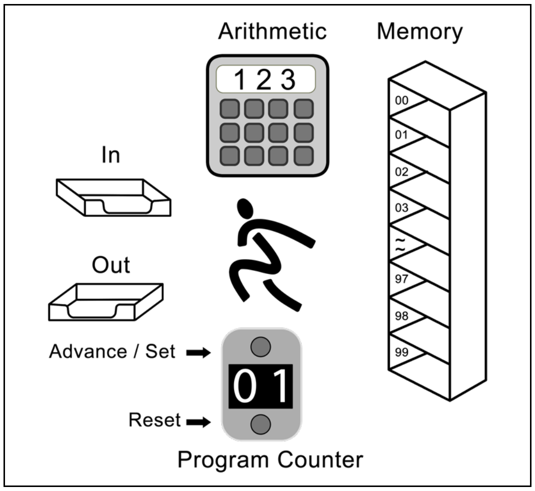

Figure 3-3
The Little Man Computer

38 Madnick is a professor at the MIT Sloan School of Management.

Each memory location can hold either an instruction or a three-digit decimal number. Instructions consist of a one digit operation code and a two digit operand address No more than ten instructions are possible.

Although it was designed primarily as a tool to teach concepts, interesting programs can be written for the LMC.

3.2.1 The Instruction Cycle

A fundamental feature of the von Neumann architecture is that *the program counter holds the address of the next instruction*. The importance of that concept will be clear from what follows.

The little man does the same things repeatedly.

1. Read the number in the program counter. (It will be zero when the LMC is first started.)
2. Read the three-digit number at the memory location given by the program counter. The little man assumes that the first digit is an operation code and the remaining two digits are a memory address.
3. Advance the program counter by one. The program counter now has the address of the instruction which will (usually) be executed next.
4. Perform the operation indicated by the first digit of the value read in step 2. Often this will require another trip to memory to read or store a value. Reads are non-destructive, but storing a value to memory replaces whatever was there before. If the instruction is a branch, the next instruction will be read from a non-consecutive location; the little man sets the program counter to the last two digits of the value obtained in step 2.
5. Go back to step 1.

This process is called the **instruction cycle**, or the fetch, decode, execute cycle. It repeats continuously in LMC and in every von Neumann architecture computer for as long as the CPU is running.

The little man could remember the address of the memory location needed and the value to be entered into the calculator. In a real computer, **registers** are used to store such data temporarily.

3.2.2 Programming the Little Man Computer

The Little Man Computer is programmed using abbreviations for the operations the computer can perform. These abbreviations are called **instruction mnemonics**. The instruction mnemonics must be translated to their numeric equivalents before the program can be executed.

The program in Figure 3-4 reads two numbers using the INP instruction, stores them in memory, performs the subtraction, and outputs the result. The halt (HLT) instruction stops the program.

```
            INP
            STA   minuend
            INP
            STA   subtrahend
            LDA   minuend
            SUB   subtrahend
            OUT
            HLT
minuend     DAT
subtrahend  DAT
```

Figure 3-4
LMC program to subtract two numbers

The program in Figure 3-4 is written in LMC's **assembly language**. Assembly language instructions generally have a one-to-one correspondence with machine instructions of a particular computer architecture and so are architecture-dependent. Programs written in a computer family's assembly language are translated to the target machines hardware instructions by a program called an **assembler**.

Details of programming the Little Man Computer are in Appendix A.

3.2.3 Problems with the Little Man computer

The essential problem with LMC is that it is a decimal computer. That obscures some fundamental relationships between binary representation and computer architecture. For example, LMC has 100 memory locations. We need seven bits to enumerate 100 locations, but with seven bits, we can actually enumerate 128 locations. Such a computer would be designed to support up to 128 memory locations. Similarly, four bits are needed to enumerate up to ten instruction operation codes, but four bits allow enumeration of 16 instruction codes, so the design need not be limited to ten.

The concept of a "man" is problematic because there is nothing volitional about the execution of instructions in a CPU.

3.3 Let's Build a CPU – the Tiny Binary Computer

The hypothetical little man of the Little Man Computer can perform addition and subtraction using the calculator, can enter numbers into the calculator, and can copy the results. The little man is also able to increment the program counter. In this section, we'll design an arithmetic logic unit (ALU) that can perform the functions required of the LMC. We are undoing the abstraction of the LMC to see how a real, but very simple, computer might work.

A CPU consists of an **arithmetic logic unit** (ALU), some number of **registers**, which are small, fast storage within the CPU, the electrical connections, called **buses**, between them, and a **control unit**. Although all four components are essential, it is the ALU that determines what a particular CPU can do, so we look first at the ALU.

3.3.1 Architecture of the TBC

Our Tiny Binary Computer should be able to execute the same instructions as the Little Man Computer.[39] That means the instruction operation code must be four bits to allow for nine operation codes. Seven bits are needed to address 100 storage locations, but can actually address 128 locations. The change from decimal to binary increases the number of possible instructions from ten to 16 and increases the maximum addressable memory from 100 locations to 128. The magnitude of the largest representable data item is also increased. That is a concrete example of the use of binary numbers increasing the expressive power of binary circuits.

39 Although the Tiny Binary Computer executes the instruction set of the Little Man Computer, its architecture and design were heavily influenced by Andrew Tanenbaum's MIC-1 design. (Tanenbaum, 1990)

Four bits of operation code and seven bits of address makes a total word size of 11 bits, distinctly odd for even a tiny, hypothetical, computer. If eight bits are used for addresses, one gets a maximum addressable memory of 256 locations. That makes a data word twelve bits, allowing signed numbers from −2048 to +2047 in twos complement form. Twelve bit words are unusual, but the hugely-successful PDP-8 computer of the 1960s used twelve bit words.

The TBC's data registers must be twelve bits wide and address registers must be eight bits wide. The instruction format is shown in Figure 3-5, with four bits of operation code and twelve bits of operand address.

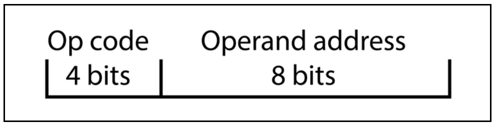

Figure 3-5
TBC Instruction format

3.3.2 A One Bit ALU

We saw in Section 2.5.2 that an AND gate can implement an *enable* function, either propagating an input value to the output or holding the output to zero. If one input to an AND gate is zero, the output will always be zero. If one input, which we call *Ena* or *Enable,* is one, the output will be equal to the other input. We also saw that the XOR gate can be used as a controlled inverter. When *Inv* is zero, the value of *In* appears at the output. When *Inv* is one, the value of the output is the inverse of the input and the circuit behaves like a NOT gate. Used in this way, the XOR gate can be used to choose whether to invert a signal or not. These concepts are shown as truth tables in Figure 2-7 and Figure 2-8.

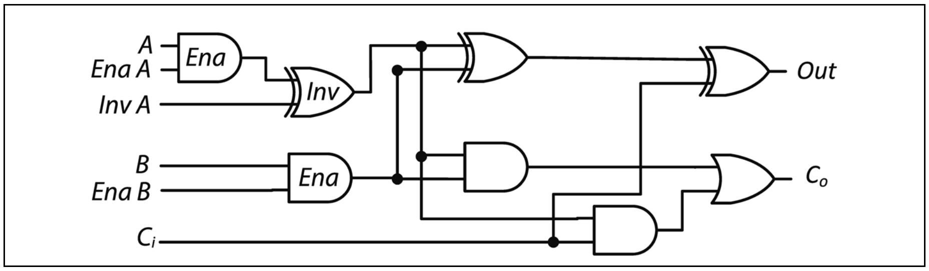

Figure 3-6
One bit arithmetic logic unit

It is important to remember that these are just ordinary AND and XOR gates. The difference is that they are put to a specific use, namely using one input to control how the signal on the other input is propagated through the gate.

We can now examine the arithmetic logic unit of Figure 3-6. In the following discussion, saying that a signal is **asserted** means it is set to a logic one to cause the circuit to act.

The gates on the right side of the circuit are a full adder. The arrangement of the gates has been changed slightly to minimize line crossings, but the full adder is logically equivalent to the circuit in Figure 2-10c.

The two AND gates labeled *Ena* perform the *enable* function for the two inputs, *A* and *B*. The XOR gate labeled *Inv* is a controlled inverter for the *A* input.

If input signals *Ena A* and *Ena B* are asserted and *Inv A* is false, the circuit functions as a full adder, adding *A*, *B*, and C_i to produce the sum at *Out* and the carry out at C_o.

If *Ena A*, *Ena B*, and *InvA* are false, the adder computes 0+0 and emits a constant zero. This very simple ALU can perform a number of other operations. They are listed in Figure 3-8.

3.3.3 Functions of the ALU

To see what other functions this ALU can perform, it is necessary to extend the ALU to operate on more than a single bit. That can be done in a manner similar to the ripple carry adder of Figure 2-11. Such an arrangement is shown in Figure 3-7. (TBC will need twelve bits in its ALU, but that only requires adding eight more one-bit ALUs.)

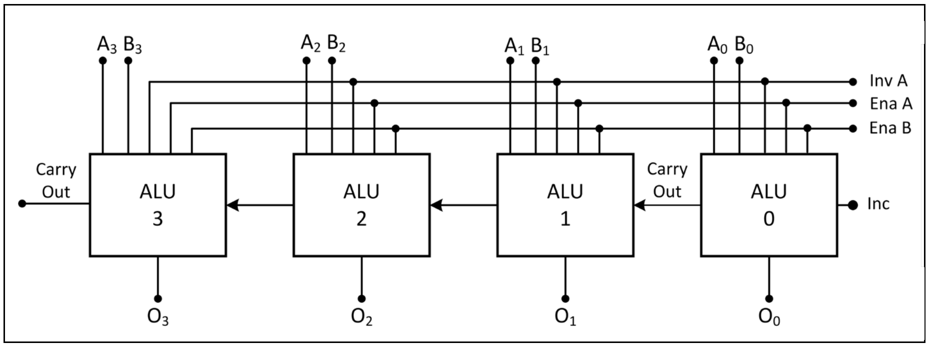

Figure 3-7
Four bit "ripple carry" ALU

Carry in to the rightmost bit (ALU 0) has been relabeled *Inc*, increment. The other controls, *Inv A*, *Ena A*, and *Ena B*, are connected to all of the ALU units.

There are 16 possible combinations of four control bits, but not all of them are useful. Everything necessary to execute the LMC instruction set can be accomplished with four control combinations as shown in Figure 3-8. A bullet in a cell means that control signal is true, or one; a blank cell means that signal is false, or zero

Function	Inv A	Ena A	Ena B	Inc
Add (A+B)		•	•	
Subtract (B–A)	•	•	•	•
Copy A		•		
Increment A		•		•

Figure 3-8
Functions and control signals for the ALU

The subtract function applies the subtraction rule for two's complement numbers explained in Chapter 1, namely take the two's complement of the subtrahend and add it to the minuend. The ALU forms the two's complement of A by inverting the A bits using *Inv A* and adding one by setting *Inc*. The adder completes the operation.

The copy A function actually computes A+0 by turning off *Ena B*. The increment operation enables A and asserts the *Inc* control, adding one to the A input.

Those are the operation needed to execute the LMC instruction set. This ALU can perform several more useful operations, including Copy B, Increment B, Constant zero, Constant one, and Constant minus one. Those operations aren't needed for the LMC instruction set.

3.3.4 Testing the ALU Output

We aren't quite finished. Both LMC and real computers have instructions to test the result of calculations performed by the arithmetic logic unit. For example, LMC and TBC have a BRP, branch on positive, instruction that goes to a different location in the program if the result of the last calculation was positive and a BRZ, branch on zero instruction. We need to be able to test the output for positive and zero. Because the ALU performs finite-precision arithmetic on twelve-bit operands, it is possible to compute a result that is too large to be stored. That condition is called **overflow** and is an error for which the ALU must check.

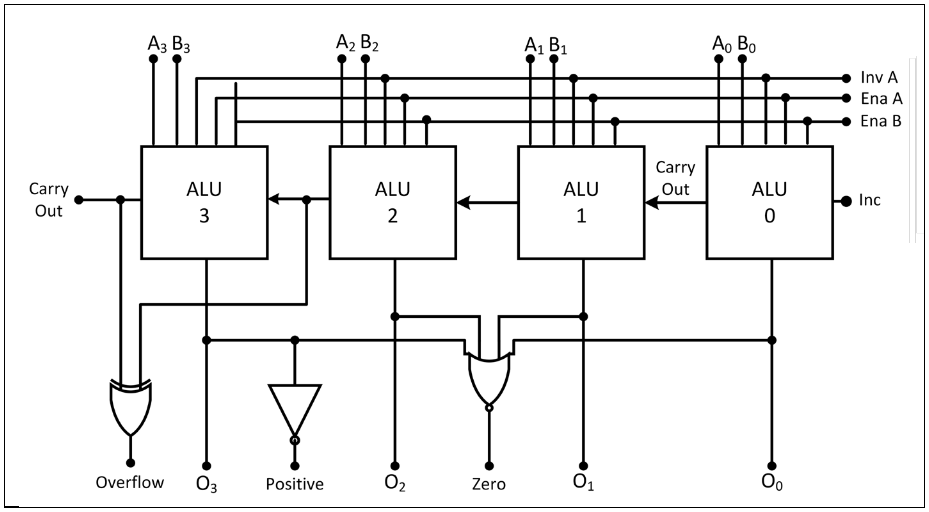

Figure 3-9
The ALU with checks for output

The zero condition is checked with an *n*-input NOR gate. If all inputs to that gate are zero, the gate produces a one or true; otherwise, it is zero or false.

The positive condition depends on the fact that the leftmost bit of a two's complement number is the sign bit. If the leftmost bit is a one, indicating a negative number, the inverter produces a zero, or false; otherwise, it produces a one, or true indicating a positive result.

The **overflow rule** for two's complement addition is: *overflow has occurred if two numbers of the same sign are added and the result is of the opposite sign.*

The overflow condition is checked by comparing the carry in to the leftmost bit with the carry out from the leftmost bit. Recall that an XOR gate produces a one, or true, when the input bits are equal and a zero, or false when they're different. There are two ways that can happen.

Case 1: If the carry in to the leftmost bit is zero, the only way the carry out can be one is if both A and B of the leftmost bit were one, *i.e.* both negative. The sum of 1+1 (plus zero for the carry in) will be zero with a carry out. The two operands were negative, but the leftmost bit of the result will be zero, or be positive, and overflow has occurred.

Case 2: If the carry in to the leftmost bit is one, the only way carry out can be zero is if both A and B of the leftmost bit were zero, *i.e.* positive. The sum of 0+0+1 will be one. The two operands were positive, but the carry out will be zero and leftmost bit of the result will be one, or negative, and overflow has occurred.

TBC processes only signed numbers. The mechanism above does not work for unsigned integers. In the case of unsigned integers, a carry out from the leftmost bit indicates overflow.

Overflow is an error, and neither LMC nor TBC has an instruction like BRV, branch on overflow, to test for such an error. The error must be handled by having the TBC emulator print an error message and halt. In a real computer, overflow might be handled by a **trap**. In the event of overflow, the CPU would force a branch to a specific address, which would be expected to be the operating system's overflow handler.

The zero and positive values must be saved so they can be tested in subsequent instructions. A real computer with a less-limited instruction set would also save the overflow and carry out bits. Conceptually, these would be stored in a four-bit **flags register** at the same time the result of the operation was stored. The flags register always holds the result of the last arithmetic operation. In TCB, the flags register is called the P/Z latch.

3.3.5 The Registers and Datapath

A **register** is a small, fast memory component that is a part of the CPU or similar device. Generally, registers store a single word of data. A storage device smaller than one word is often called a **latch**. The Little Man Computer has two

Register	Function
Accumulator	Receives the results of ALU operations; the accumulator is represented by the calculator display in LMC.
Program Counter	Holds the address of the next instruction.
Instruction Register	Holds the operation code and operand address of the current instruction.
Memory Address Register	Holds the address to be read from or written to memory.
Memory Data Register	Holds data to be sent to memory; receives data read from memory.
P/Z Latch	Holds the positive and zero flags from the last ALU result to be stored in the accumulator.

Figure 3-10
Registers of Tiny Binary Computer and their functions.

visible registers, the calculator display and the program counter. The little man remembered the memory location to be read or written, the contents when reading from memory, and the instruction being performed. In a real computer, however simple, that remembered data is held in registers.

To do those things the Little Man Computer can do, TBC needs five registers and a latch. They are listed in Figure 3-10.

The **datapath** of a computer is the ALU and possibly other functional units like multipliers, shifters, or incrementers, the registers, and the buses that connect them. The datapath and control unit together comprise the CPU.

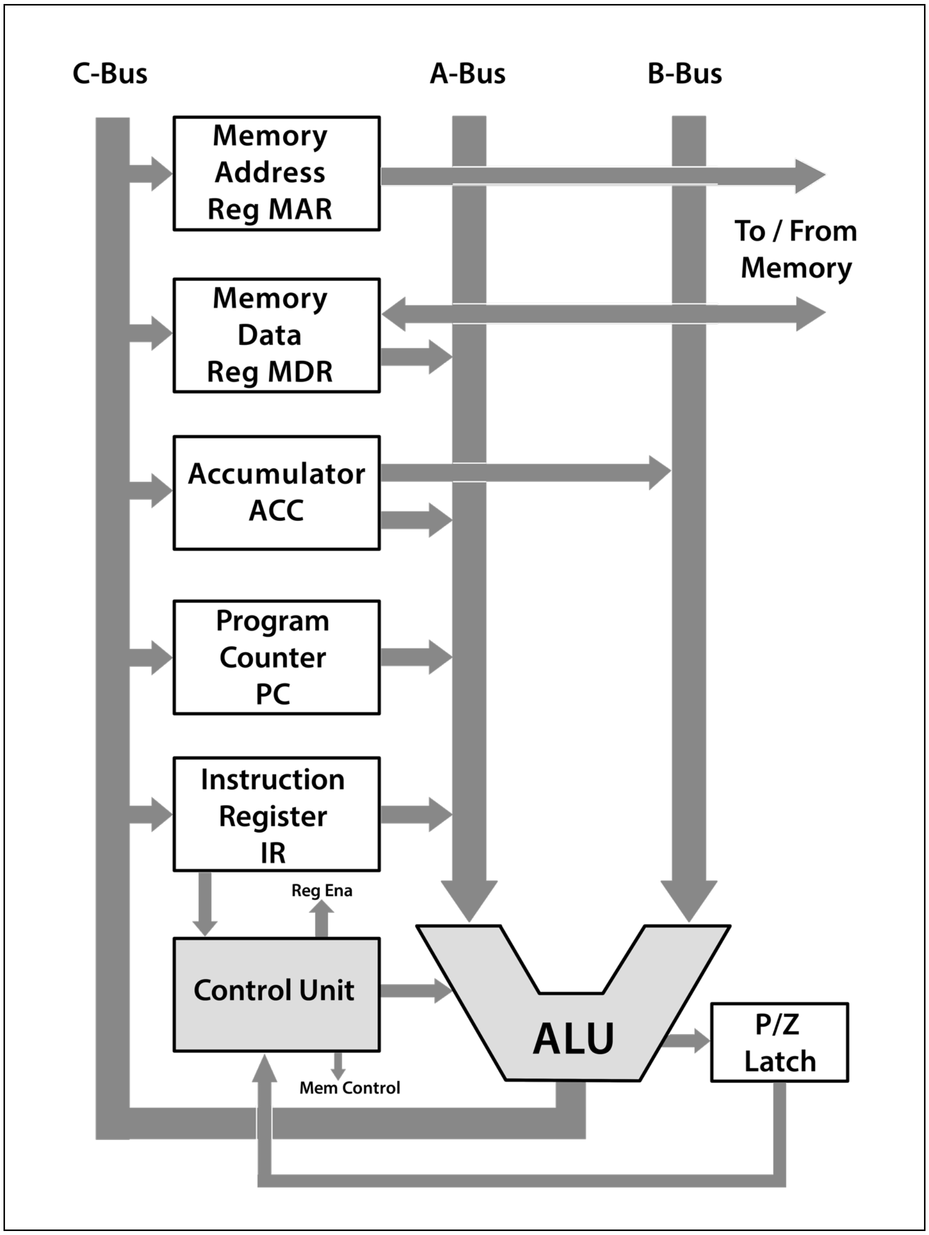

Figure 3-11
The CPU of the Tiny Binary Computer

Figure 3-11 is a diagram of the CPU of the TBC. Units with light gray background are combinational circuits; those with white background are registers. The symbol for the ALU is a truncated V, shown at the bottom right. Using that symbol abstracts away the details of the ALU. The buses that connect the units are darker gray with arrows to show direction of data flow. Buses that cross are not connected.

Registers and Buses

The A, B, and C buses and the memory data bus are twelve bits wide; they have twelve electric conductors and can transfer twelve bits at a time. The memory address bus is eight bits wide.

The memory address register and program counter hold addresses, and so are eight bits wide. When the program counter's output is enabled, the eight bits appear on the rightmost eight bits of the A bus and the remaining four bits are zero. When the memory address register is loaded from the C bus, the rightmost eight bits are copied into the MAR.

The remaining registers are twelve bits wide. The connections from the instruction register are described below.

3.3.6 The TBC Datapath Cycle

The clock cycle of TBC is that of Figure 2-18. On the falling edge of the clock pulse the control unit sets up the necessary signals. After a delay determined by the signal propagation time and the characteristics of the register memory, the selected registers place their contents on the A and B buses. The ALU is combinational logic, so it is computing continuously but the output is not valid until some time after the data on the A and B buses is valid. The period of the clock is set so that the output of the ALU will be valid and propagated back to the registers on the C bus. The clock period has a small amount of additional time to allow for reliable operation.

The process of sending data from the registers through the ALU and back to the registers is called a **datapath cycle**. TBC performs one datapath cycle per clock period.

The operations necessary to execute an instruction in TBC's CPU are described using a register transfer language similar to the notation of Davidson and Fraser (1980). This line:

MAR ← PC; read

means that the contents of the program counter are copied to the memory address register and the memory read control line is asserted. The control unit accomplishes this by doing the following:

- Enable the PC register, placing its contents on the A bus.
- Send a "copy A" command to the ALU by asserting the *Ena A* control.
- Load the MAR from the C bus by asserting a write signal to the MAR; the register will be loaded on the rising edge of the clock.
- The read control line to memory is asserted, also at the rising edge of the clock cycle.

Since the program counter holds the address of the next instruction, that sequence causes the next instruction to be read from memory and delivered to the memory data register. However, the memory is slower than the CPU. The memory data register will not contain the instruction until one full clock cycle after the memory read control line is asserted. Real computers take 50 or more clock cycles to read memory, although cache memory, described in Section 3.5.4, mitigates this speed mismatch somewhat.

The control unit can generate a wait operation by not asserting any of the control lines. A clock cycle goes by with nothing being stored.

The CPU performs an ADD instruction

The LMC / TBC ADD instruction fetches a value from memory at the address given in the operand field and adds that value to the accumulator. It takes six datapath cycles, and so six clock cycles, to perform an add instruction. Here is the register transfer language description of the ADD instruction.

MAR ← PC; read	Start a memory operation
PC ← PC + 1	Increment the program counter
IR ← MDR	Copy the instruction to instruction register
MAR ← IR[addr]; read	Start fetch of operand value

```
wait                  Wait for memory to complete
ACC ← ACC + MDR       Perform the addition
```

The first three datapath cycles are the same for every instruction. The first operation is described above. The second one increments the program counter so that it again points to the next instruction. During the same datapath cycle that updates the PC, memory is completing the read from step one and delivering the instruction word to the memory data register. That completes the fetch part of the fetch-decode-execute instruction cycle.

In the third datapath cycle, the instruction word is copied from the MDR to the instruction register. The control unit enables the MDR onto the A bus, sends a copy command to the ALU, and a write signal to the IR. The signal that enables the write to the IR also signifies that an operation code is available. The control unit is fast enough to determine the next steps during the high part of the clock cycle. That is the decode part of the fetch-decode-execute instruction cycle.

The last three register transfer operations are the execute part of the cycle. The address part of the instruction register is loaded into the memory address register and memory is commanded to read from that address. There is nothing to do until the memory operation completes, so the next datapath cycle is a wait cycle. The last step performs the addition. The MDR is enabled on the A bus, the accumulator is enabled on the B bus, and the ALU is sent the signals for addition. At the end of the clock cycle, the accumulator is loaded from the C bus, replacing the prior contents with the sum from the addition.

The other instructions that are part of the TBC instruction set work similarly. There is a complete list in Appendix B.

How branch instructions work

If you have written programs in a high level language like Java or Python, you are used to writing procedures or methods, then using them as though they were like any other feature of the language. The procedure gets executed, then the statement following the invocation of the procedure. At the level of machine language, which is where things really happen, a procedure call involves saving the program counter, then branching to the first instruction in the procedure.

Code in the procedure restores that saved program counter when the procedure has finished. We will discuss how a procedure completes the return in Section 3.5.5.

The mechanism for calling a procedure, performing a loop, or doing anything other than executing the next instruction in sequence is a **branch instruction**. Computer architectures implement both unconditional and conditional branch instructions. An unconditional branch always transfers control to another location within the program. A conditional branch only transfers control if some specific condition has been met. LMC and TBC implement two conditional branches, branch on positive and branch on zero. The branch is taken only if the last arithmetic operation resulted in a positive or zero value, respectively.

Branch instructions take advantage of the fact that the program counter holds the address of the next instruction. In TCB, the program counter is updated while the CPU is waiting for the instruction word to be delivered by the memory subsystem. The process is slightly more complex in computer architectures with variable-length instructions, but the program counter is always updated before the execute phase of the fetch-decode-execute instruction cycle.

To execute a branch, the datapath stores the branch target address, which is given in the branch instruction, into the program counter. Here is the register transfer language for a branch on positive.

MAR ← PC; read	Start a memory operation
PC ← PC + 1	Increment the program counter
IR ← MDR	Copy the instruction to instruction register
IFP; PC ← IR[addr]	Branch target address stored in PC only if the P flag is set

The first three clock cycles are the same for every TBC instruction. In the fourth cycles, the branch target address from the instruction is copied into the program counter, but only if the *P* flag is set. For an unconditional branch, the IF part is removed.

The branch target address is available from the instruction register, without another memory access.

Now it should be clear why the program counter must be incremented before the execute phase of the instruction. If it were incremented after the execute step, then incrementing the program counter would incorrectly change the branch target.

The control unit

The TBC control unit generates five sets of control signals. It generates ALU control signals as shown in Figure 3-8 to perform the operation needed for the instruction. It generates signals to the registers to enable their contents onto a bus and also to load a result from the C bus. Registers can send their contents to the A bus, or, in the case of the accumulator, to either the A or B bus. Only one register can be enabled to the A bus and only one to the B bus during any datapath cycle. The C bus can load more than one register simultaneously, although that capability is not used in TBC.

When the accumulator stores data from the C bus, the P/Z latch is also signaled to store the positive and overflow bits from the ALU. That means the P/Z latch values always correspond to the value currently held in the accumulator, even though the output from the ALU may have changed.

The instruction register is twelve bits wide like the others, but is wired so that only the low-order eight bits, the operand address, will be transferred to the A bus if the instruction register is enabled. The four bits of operation code are connected directly to the instruction decoder and control unit by a dedicated bus. (The instruction format is shown in Figure 3-5.)

Finally, when a memory operation is needed, the control unit sends either a read command or a write command to memory.

Because the TBC instructions take more than one datapath cycle the control unit needs a counter to keep up with the current step in the instruction. Such a counter is fundamentally similar to the adder we have explored in detail. It will also have a RESET input that returns the counter to zero. No TBC instruction needs more than six datapath cycles, so a three-bit counter is needed. The counter is input to a three-to-eight decoder. (See Figure 2-13 for a two to four decoder.) It's not necessary to decode values six or seven, so we can design the decoder with only six outputs for values zero to five.

In the fourth datapath cycle, the operation code is available through the connection from the instruction register to the control unit. The operation code is four bits, and so needs a four-to-16 decoder. The branch instructions need two bits from the P/Z latch. That gives a total of 24 bits, six from the counter, 16 from the instruction decoder, and two from the P/Z latch. Those 24 bits are input to a combinational logic circuit that generates the proper signals for each datapath cycle. The combinational logic circuit also generates the RESET signal to the counter after the last datapath cycle of each instruction, which causes the operations to repeat for the next instruction.

We can build a combinational logic circuit for any function for which we can write a truth table. Before decoding, we have nine bits of variable, three from the counter, four from the instruction register, and two from the P/Z latch. Such a truth table would have $2^9 = 512$ rows, but there are automated tools to help with designing such circuits. It isn't necessary to produce a complete design to see that we can build a control unit that will do what is needed. One can get a good conceptual understanding of how the CPU works without going further into the details.

A control unit that uses combinational logic and perhaps a few registers to generate the necessary control signals is called a hardwired control unit. The alternative is a microprogrammed control unit, in which the steps for the datapath cycles are stored in a fast read only memory. We won't go into the details here, but for the curious, a microprogrammed implementation of a control unit for TBC is described in Appendix B. Hardwired control units are generally faster, but also more complex and hence more expensive than microprogrammed control units.

It is important to recognize that TBC is a very simple computer, and so has a very simple control unit. Real computer control units are substantially more complex.

3.4 Instruction Formats and Addressing Modes

The TBC instruction format, shown in Figure 3-5 in about as simple as one could get. Even so, it is worth a more detailed look. There is a four-bit operation code, which allows up to 16 different operations. The eight bits of **operand address** allow for addressing any of the possible 256 twelve-bit words. The TBC addresses are **direct addresses**, meaning that no other information is needed to specify a memory location. Most computer operations involve two operands. In TBC, the second is an **implicit operand**. For example, the ADD instruction has an operation code and address. The second, implicit operand is the accumulator, which supplied one of the addends and receives the result of the addition. The accumulator is an implicit operand for LDA, SUB, and STO. The INP and OUT instructions are a bit different. They have the same operation code and use the address field to distinguish between input and output.

3.4.1 Instruction Formats

Most computers have more than one register that can receive the results of an arithmetic operation or participate in load and store operations. That means that, in most cases, implicit operands can't be used and both operands must be explicitly specified in the instruction. A general format for an instruction with two operands is shown in Figure 3-12. Some computers specify the destination operand, the operand where the result will be stored as the first operand and some specify it as the second operand. The operand that will be overwritten is called the **destination operand**.

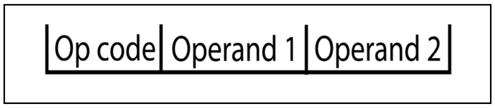

Figure 3-12
General form of an instruction

The size of an operand field depends on how many possibilities must be enumerated. For a computer with 16 registers, four bits are enough to specify one of them. A computer with four gibibytes of memory needs 32 bits to specify a direct memory address. Both of these considerations have influenced the way CPU designers format instructions. Memory was initially very expensive and the designers of CPUs used as little as possible. An IBM mainframe instruction with two register operands is 16 bits long, eight bits for an operation code and four bits each for the register numbers. The mainframe instruction set has over 20 different instruction formats and three different instruction lengths, making the instruction decoder very complex. The trade-off is that programs for those early mainframes, and for today's mainframes, used less memory than otherwise. The Intel x86 instruction set also has multiple formats and instruction lengths.

The other consideration is the number of bits needed for direct addressing of large memories. Instruction formats that must be retained for backward compatibility do not have room for large direct address fields. As a result, CPU designers provided a number of ways of generating the address actually used by the memory controller to address a word of memory.

There are instruction formats other than the two-address format shown here. Some architectures have three-address instructions. That allows two operands to participate in an operation and the result stored in a third location instead of overwriting one of the operands. Some instructions use implicit operands, particularly when referring to specialized registers.

The Intel instruction set, especially, provides for instruction modifier bytes. These modify the meaning of operation codes and further complicate the control unit design.

3.4.2 Addressing Modes

There are three general forms of addresses. A register-to-register instruction format specifies two registers as the operands. The number of bits required depends on the number of registers, but will generally be small.

The register-to-memory format needs one register address and one main memory address. The main memory address is necessarily larger than a register address.

Some architectures include memory-to-memory instructions, in which both operands are main memory addresses.

An **orthogonal** instruction set would allow any addressing mode for any operand. Although that was a design goal fifty years ago, modern instruction sets are not orthogonal. The Intel instruction sets can be said to be somewhat orthogonal due to design decisions made early in the design of the CPU family.

The **effective address** of an instruction operand is the address actually used to address the operand held in a register or main memory, and is often the result of arithmetic performed in an address generation unit or its equivalent. An effective address in main memory is also used for the target address of branch instructions.

Addressing modes for data and for flow of control (branching) are slightly different in many computer architectures.

Addressing modes for data

Direct addressing requires that the full memory address of data be stored in the address field of the instruction. Often the only way to do this is to make instructions more than one word long.

Displacement addressing forms an effective address by adding the address constant from the instruction to a **base register**. That requires fewer bits in the address field of the instruction provided the base register is wide enough to address all of memory. The size of the address field in the instruction determines the amount of memory that can be addressed with a single base register.

With **register indirect** addressing, the address of the data is held in a register and the address field of the instruction need only be large enough to enumerate the registers. **Memory indirect** addressing loads the effective address from a location in main memory. That requires both address arithmetic to locate the address in main memory and a memory access to retrieve the address before data can be accessed.

Indexed addressing adds an **index register** to the address arithmetic process. For example, a base register may hold the address of an array in main memory, and the contents of the index register selects one element of the array. A program can step through an array by manipulating only the index register.

Immediate mode addressing does not actually use a register or memory address at all. Instead, a constant value is encoded directly in the instruction. It is "immediate" because no main memory access is needed to retrieve a value; the value is available in the address field of the instruction register. Andrew Tanenbaum (1978) examined over 10,000 lines of program text and found that over 81% of all the constants in a program are zero, one, or two. Even if only a few bits are available to hold immediate data, a substantial performance increase can be achieved by avoiding memory accesses for small constants.

Addressing modes for flow of control

Sequence is the natural operating mode of von Neumann architecture computers. In the absence of a specific instruction to change the flow of control, the next instruction in sequence is executed.

With **direct addressing**, a branch or call instruction has the full memory address of the branch target. As with addressing modes for data, the operand field must be large enough to address all of memory.

An **indirect branch** uses a table of effective addresses stored in main memory. Address arithmetic first selects an entry in the table, than places that address in the program counter. Indirect branching is used in interrupt handling, where an interrupt number determines which branch table entry should be used.

Relative addressing forms an address from the program counter and an address constant in the instruction. The principal advantage is that a few bits of address constant allow branching to "nearby" addresses. An eight-bit signed address constant is enough to allow jumping forward 127 bytes or backward 128 bytes. Each additional bit doubles the range. In well-modularized code, branches are likely to be to nearby addresses. Returns from a module are likely to use a stack indirect or register indirect branch. Another advantage is that such code is position-independent; it can be loaded anywhere in memory without need to adjust addresses.

In **register indirect** addressing, the branch address is taken from the register specified in the instruction's address field.

Stack indirect addressing is used for returns from procedure calls. The branch target address is taken from the memory location specified by the stack pointer register and the stack pointer is adjusted to remove that item from the stack. The use of a stack in this way is described in Section 3.5.5.

Some architectures implement a **SKIP** instruction. This is a conditional instruction which, if the condition is true, causes the *next* instruction to be skipped. The following instruction is often an unconditional branch. The value of using SKIP rather than a conditional branch is that the CPU pipeline need not be flushed when a branch is taken. Pipelined execution id discussed in Section 3.6.3.

3.5 Memory

Memory is the digital circuitry that holds programs and data immediately available to the CPU. Memory is also called RAM (random access memory), primary memory, or main memory. It is sometimes called main storage, although this book reserves the word "storage" for semi-permanent storage on devices like disks or flash drives. The term *random access memory* means that any location in the memory system can be read or written in about the same amount of time as any other location.

The main memory system is connected to the CPU with data, address, and control buses. The control bus can signal the memory to read or write. The address bus holds the address of the memory location to be read or written, and the data bus transfers data between the CPU and memory. A memory management unit (MMU) handles the details of communication between the CPU and memory.

3.5.1 Memory Technology

Memory requires a technology capable of maintaining and distinguishing two states that will represent bit values zero or one. A device that can maintain either of two states is called **bistable**. Some of the earliest commercial memories used tiny ferrite rings called *cores* [40] that could be magnetized either clockwise

40 The ferrite cores used for memory are not the same as the multiple cores in some CPUs. The latter are complete processors.

or counter-clockwise. Such core memory was mostly assembled by hand and used in early commercial mainframe computers. The first core memories cost about a dollar per bit. By the 1960s, prices had come down to about one cent per bit, which would have made a gibibyte of memory cost over 86 million dollars.

Memory can be either volatile or non-volatile. **Volatile** memory depends on electrical power to maintain storage and loses its contents when power is removed. **Non-volatile** memory retains its contents even when power is removed.

Four memory technologies, used for distinctly different purposes, predominate in modern general purpose computers.

Dynamic RAM

The main program and data storage memory of modern computers is synchronous double data rate dynamic RAM (DDR SDRAM). In dynamic RAM, each bit of storage comprises one transistor and one capacitor. A capacitor is an electronic device for temporarily holding an electrical charge. Operation of synchronous memory is controlled by a train of clock pulses, and double data rate means the memory can transfer data on both the rising edge and falling edge of a clock pulse.

Progress in the development of dynamic RAM has been in favor of higher densities, that is, more memory, and lower cost. Speed of dynamic RAM has increased as well, but it remains tens of times slower than typical CPU speeds.

Because the capacitor in each bit position can hold a charge only temporarily, dynamic RAM must be refreshed periodically. A refresh cycle reads and rewrites each bit, renewing the charge on the capacitor. Refresh times for modern memory are in the tens of milliseconds. Refreshing is handled by the memory controller, which is often in the same chip package as the CPU. The requirement to refresh memory periodically imposes a small performance penalty on dynamic RAM.

Improvements in dynamic RAM are identified by numbered generations, standardized by the JEDEC Solid State Technology Association, an independent trade and standards association. Differences in generations include chip density, operating voltage, and clock speed. DDR4 memory modules began supplanting DDR3 modules in 2017. The DDR5 specification was published in July 2020, but as of this writing, no DDR5 memory modules are available.

Dynamic RAM can be buffered or unbuffered. Buffered memory provides a transistor buffer between the memory module and the memory bus. This reduces the load on the bus and so allows more memory to be installed. There can be a performance penalty to using buffered memory.

Memory configurations are generally not interchangeable. For example, a computer designed for DDR3 memory will not accept DDR4 memory. A computer designed for buffered memory generally will not accept unbuffered memory. It is important to know the system specifications when adding or upgrading dynamic RAM.

Static RAM

Static RAM uses six or more transistors per bit of storage, similar to the clocked D-latch of Figure 2-17. That makes a static RAM module physically larger than a dynamic RAM module of similar capacity. There is a dramatic difference in the time / cost trade-off. Static RAM can operate at CPU speeds, but costs 100 or more times as much as the equivalent amount of dynamic RAM.

Static RAM is used for registers within the CPU and for cache memory. Cache memory is discussed in Section 3.5.4.

Mask ROM

ROM is read-only memory. It can be read, but not written or changed. The name **mask ROM** comes from the fact that both the pattern of the memory and also the contents are set when a chip is manufactured. Mask ROM is very fast; it can operate faster than the CPU clock, which makes it suitable for holding a chip's microprogram. Some controller chips are built with their start-up code in mask ROM. A disadvantage of mask ROM, and any other read-only memory, is that it cannot be changed.

NOR Flash

NOR flash memory is used to hold executable code used to start a computer system. The name comes from the fact that the storage elements are similar to, but not identical to, the NOR gates described in Chapter 2. NOR flash memory can be read a word at a time in much the same way as dynamic or static RAM. That makes it suitable for holding executable code, such as the code used to start an operating system. It is this memory that is updated when one "flashes the BIOS" [41] of a computer. NOR flash is a different technology from the NAND flash used for storage.

3.5.2 Memory Organization and Addressing

Modern general purpose computers almost universally organize memories into addressable units of eight bits, called **bytes** or **octets**. Octets are further organized into **words**, often of 64 or 32 bits, that is, eight or four octets. Other memory organizations are possible; for example, some digital signal processing chips organize memories into 16-bit units. Section 3.3.1 describes a memory organized into twelve bit words.

The amount of memory a computer can accommodate physically depends on several factors, some as simple as the number of connectors available for installing physical memory. A fundamental limitation is that the maximum physical memory is limited to 2^m addressable cells, where m is the number of physical address lines. A computer with 32 address lines cannot address more than 2^{32} memory cells. It the case of byte-addressable memory that means 2^{32} bytes or four gibibytes.

The amount of memory a program can logically address depends on the number of address bits available to the programmer, which may be different from

41 BIOS, pronounced BY-oss, is "basic input output system" and is the executable code that loads and starts the operating system's loader. The term BIOS is obsolescent. Most modern computers use the unified extensible firmware interface (UEFI) to start the operating system.

the number of physical address bits. A program can address at most 2^n cells where n is the size of the logical address available to the programmer.

Computers with 32-bit addresses became common with the introduction of the Intel 80386 chip in 1985. At the time, memory was measured in megabytes, and a computer for use by an individual might have one or two megabytes of memory installed. Operating systems used the features of the CPU to implement virtual memory, simulating a four gibibyte address space in a few megabytes of memory. Performance was poor for programs that required more than the available physical memory. Virtual memory is explored in Chapter 6.

The Pentium Pro, introduced in 1995, provides an interesting case study. It was a 32-bit processor capable of addressing 64 GB of memory. Intel accomplished this by providing 36 physical address lines through a feature called physical address extension. These larger physical addresses were managed by the operating system's virtual memory facility. User programs were still limited to four gibibytes each because the internal registers used for addressing were only 32 bits. Similar techniques, with names like bank switching or extended memory management, were used as far back as the 1980s to work around the addressing limitations of processors. Current Intel processors provide 64-bit registers, but "only" 48 address lines. "Only" is in quotes because 2^{48} is an extremely large number, equivalent to 256 tebibytes. It is unlikely that even very large computers will have that much memory in the near future.

Address decoding

Physically activating a given memory cell presents another kind of problem, namely decoding addresses. Decoding a 32-bit address in the straightforward way would require a 32 to 4,294,967,296 decoder. Four billion wires is unwieldy to the point of impossibility even with modern semiconductor technology. A 48-bit address is correspondingly worse. The solution is to think of memory not as a line, but as a plane with rows and columns. One can address a four gibibyte memory with 16 bits of row address and 16 bits of column address, quantities that are much easier to deal with. A memory with 36 bits of address would use 18 bits for row address and 18 bits for column address.

Word boundary alignment

Computer hardware generally transfers data between CPU and memory in units that are multiples of the size of the computer's word. For modern computers, that is often 64 bits, or eight bytes. Unless complex circuitry has been added to memory and the memory control unit, these transfers take place on a **word boundary**. For a computer with 64-bit words, that means an address divisible by eight. Compilers automatically insert padding as necessary to make sure that data items longer than one byte and less than or equal to one word are aligned on a word boundary.

Byte ordering

The way bytes of a word are numbered is a choice made when the architecture of a computer family is designed. The two choices are **big endian**, in which the most significant byte of a word is stored at the lowest address, and **little endian**, in which the least significant byte is stored at the lowest address. [42] IBM mainframes are big endian.

The Datapoint 2200 computer of the 1970s did bit-serial arithmetic and, as a consequence, needed the low order bit at the low address to handle carries. When Intel designed the 8008 chip to Datapoint's specifications, it necessarily had a little endian architecture for compatibility. Current x86 and x64 processors from Intel and compatible processors from AMD are little endian.

As long as a computer architecture is internally consistent, it makes no difference what byte ordering is chosen. Problems can arise when data are transferred between machines with different byte ordering and the unit of transfer is larger than one byte. That case arises with 16-bit Unicode UTF-16. Unicode defines a **byte order mark**, the hexadecimal characters FEFF. If data are transferred between systems of different endianness, the two bytes will be swapped, and all following byte pairs must also be exchanged. The byte order mark is optional, and if not present at the beginning of a file or transmission, big endian encoding is to be assumed.

42 The terms big endian and little endian come from Jonathan Swift's *Gulliver's Travels*, and refer to the war over which end of an egg to open. They were applied to computer byte ordering by D. Cohen (1981).

Byte order mismatches also cause problems when transferring binary data such as binary integers and floating point numbers. For that reason, data exchanges are often based on values encoded as streams of characters.

3.5.3 Error Detection and Correction

The technology used in the memory systems of general-purpose computers is such that it's possible for one or more bits to change state erroneously, often because a cosmic ray passes through the storage element. However, current DRAM technology is very stable and errors are extremely rare. This book was written on a computer without memory error detection or correction.

The simplest error detection mechanism is **parity**. One additional bit is stored for each memory cell. When a write to the cell occurs, the additional bit is set so that the total number of one-bits is even. [43] When the cell is read, the one-bits are counted. If they are no longer an even number, an error has occurred and the memory subsystem can signal the error to the memory control unit and the CPU. Often the only action that an operating system can take in the case of such an error is to halt the operation of the computer system, possibly displaying an error message.

A parity system can detect only errors in an odd number of bits. [44] If two bits are changed erroneously, the parity will be correct even though the contents have been corrupted.

Error correction with Hamming codes

Error detection with parity depends on redundancy. An additional bit is added to each cell of memory to be checked. By adding more than one bit, it becomes possible to detect errors of more than a single bit, end even to correct errors.

43 Requiring an even number of one-bits is called even parity. Odd parity, requiring an odd number of one-bits, works equally well. The only requirement is consistency.

44 Detecting errors in an odd number of bits is often expressed as detecting only single-bit errors.

In 1950, Richard Hamming of Bell Laboratories devised such a system, now called a **Hamming code**. The principle of the Hamming code is that every data bit is checked by more than one parity bit.

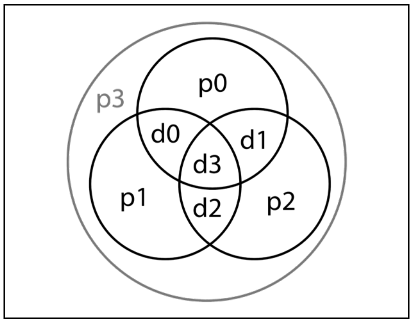

Figure 3-13
Error correction using a Hamming code

Figure 3-13 shows four data bits being checked be three parity bits. If data bit *d0* were corrupted, parity bits *p0* and *p1* would both indicate errors. The only data bit checked by both *p0* and *p1* is *d0*. If we know which single bit is incorrect, it can be corrected by complementing it, *i.e.* flipping it back to the other possible value.

A fourth parity bit, *p3*, checks the parity of all the other parity and data bits. A Hamming code like this can correct single-bit errors and detect double-bit errors.

The example uses four parity bits to check four data bits, an overhead of 100%. However, the overhead diminishes as the number of data bits to be checked by a Hamming code increases. A 32-bit word can be checked with seven parity bits, six of which check individual data bits and one of which checks the overall parity.

Error correction with triple module redundancy

The calculations needed to validate Hamming codes take time, even when performed in hardware. Some triple-bit errors can be incorrectly recognized as single-bit errors and wrongly "corrected."

Another approach is to use three separate memory modules. When a write to memory occurs, data are sent to all three modules. When a read occurs, the data are sent through a bitwise majority function. For example, if a bit from each of the three memories is 111, a one is returned. If the three memories return 110 a one is still returned, on the assumption that the third memory is in error. If the three memories return 100, a zero is returned. Thus, the majority function performs "voting" among the three memories. The majority function can be computed with two gate delay times. This technique is called **triple module redundancy**.

3.5.4 Cache Memory

Cache [45] **memory** is small, fast memory that is between main memory and the CPU and physically and logically close to the CPU. Cache memory helps to compensate for the fact that main memory is so much slower than the CPU.

Cache memory is effective because the nature of the von Neumann architecture induces **locality of reference** for both data and instructions. Locality of reference means that when particular instructions or data are needed by the CPU, nearby instructions or data will likely also be needed. That is particularly easy to see for instructions. Sequential execution, part of the nature of the von Neumann architecture, means that the next instruction in memory is likely to be the next one executed. Similarly, data are arranged in structures so that nearby data are likely to be used soon after any data item. The probability that nearby instructions or data will be needed is **spatial locality of reference**.

Temporal locality of reference is the circumstance that when in instruction or item of data is used, it will likely be used again in the near future. This is also easily observed for instructions. A program loop involves executing the same set of instructions repeatedly. It is also true that data are frequently reused during the execution of many programs.

Locality of reference is part of the nature of the von Neumann architecture; programmers need not do anything special to take advantage of this nature. It is, however, possible to create a circumstance for which locality of reference fails. For example, if an array is stored column-wise, then accessed row-wise, successive references are likely to be far apart.

In a system with a single cache, when a word of data is needed by the CPU, the cache control unit first checks the cache memory. If the needed word is present, called a **cache hit**, it is delivered to the CPU and no access to main memory is necessary. If the needed word is not in the cache, called a **cache miss**, it and

45 Pronounced "cash."

nearby data are fetched from main memory and stored in the cache. The needed word is delivered to the CPU. Cache memories are organized into blocks, and when a reference to main memory is needed, an entire block is transferred to the cache.

Modern caches are usually organized as multi-way associative caches. "Multi-way" means there are two more places in the cache where a block of data from main memory can be stored. **Tag bits** indicate the main memory address from which a particular block was loaded. "Associative" means the tag bits are checked in parallel to see whether a particular main memory address is in the cache. Associative checking is very fast, but also expensive in terms of the number of transistors needed.

Multi-level caches

The discussion above assumes one cache memory and one main memory. Modern computers implement multiple levels of cache memory. The cache physically and logically nearest the CPU is called the level 1 cache or L1 cache. The L1 cache is likely to be implemented on the same silicon chip as the CPU. L2 and sometimes L3 caches can be on the same silicon as the CPU, or in the same package.

There is a diminishing return with additional levels of cache. Intel has built systems with four levels of cache, but architectures with three levels of cache are more usual.

The L1 cache might be 32 or 64 kibibytes for a modern CPU. The L2 cache could be eight or more times larger, perhaps 256 or 512 kibibytes. The L3 cache might be four or eight mebibytes. Each succeeding level is many times larger than its predecessor, but also much slower. However, even the L3 cache is much faster than main memory. Some 21st century CPUs use embedded dynamic RAM, called eDRAM, for the L3 cache.

In multi-core chips, it is usual for each core to have its own L1 cache, and sometimes its own L2 cache.

Instructions and data exhibit different locality of reference behavior. For that reason, the L1 cache is often a **split cache**, with one cache for instructions and another for data.

Cache and memory coherence

With multiple levels of cache and separate cache memories for each of several cores, no item of data has a canonical home. The same word may exist in main memory, in an L3 cache, and in two or more L1 or L2 caches. Since multiple cores may update the same word, it is possible for two copies of a word to differ. This is called the **cache coherence problem**. It is solved by circuitry through which the various caches communicate, so that only one copy of a particular word is considered the valid one if their values differ. [46]

Even with a cache coherence protocol mediating among the caches, if a CPU updates a word, the contents of the cache can differ from the same word in main memory. That is problematic if an output operation from main memory takes place when cache and memory do not match. That is partially addressed by the cache **write policy** of a CPU architecture. A **write through policy** updates memory every time a cache entry is modified ad a cost of using extra memory bandwidth. A **write back policy** flags a cache block as changed but defers writing until the block must be replaced or the corresponding entry in main memory is accessed. That saves memory bandwidth at a cost of additional circuitry in the cache controller.

3.5.5 Stacks and Stack Pointer

Writing a method, procedure, or function in a high-level language is like adding a new instruction to the language. When a procedure is called, control passes to the first instruction in the procedure. When the procedure completes, control returns to the instruction immediately following the procedure call. We saw in Section 3.3.6 how a branch instruction can pass control to the first instruction in a procedure by storing a new value in the program counter. Procedures

46 Cache coherence protocols are beyond the scope of this book, but interesting. For further reading, look up the MOESI protocol

have the same difficulty that Hansel and Gretel did; getting there is easy, the problem is how to get back.

That difficulty is solved easily with a data structure called a **stack**. A stack is a last-in first-out (LIFO) structure, functionally like the stack of trays in a cafeteria.

Figure 3-14 shows the use of a stack to store return addresses. Numbers in gray are example addresses. The main program has a CALL A instruction at location 1026; procedure A starts at location 2048. The CALL instruction first **pushes** the program counter at the top of the stack. Since the program counter holds the address of the next instruction, and has already been adjusted by the time the execute phase of the CALL instruction is reached, the value 1027 is stored as shown in Figure 3-14*(a)*. Once the return address has been stored, the CALL instruction executes a branch to the procedure's address, 2048 in the example, by storing the branch address in the program counter.

The real value is the stack becomes clear in Figure 3-14*(b)* when procedure A calls procedure B. The CALL B instruction pushes the program counter, now 2050, onto the stack, the branches to procedure B at location 4096. Placing a value on the stack is called pushing because prior values are pushed down, like the stack of trays in the cafeteria.

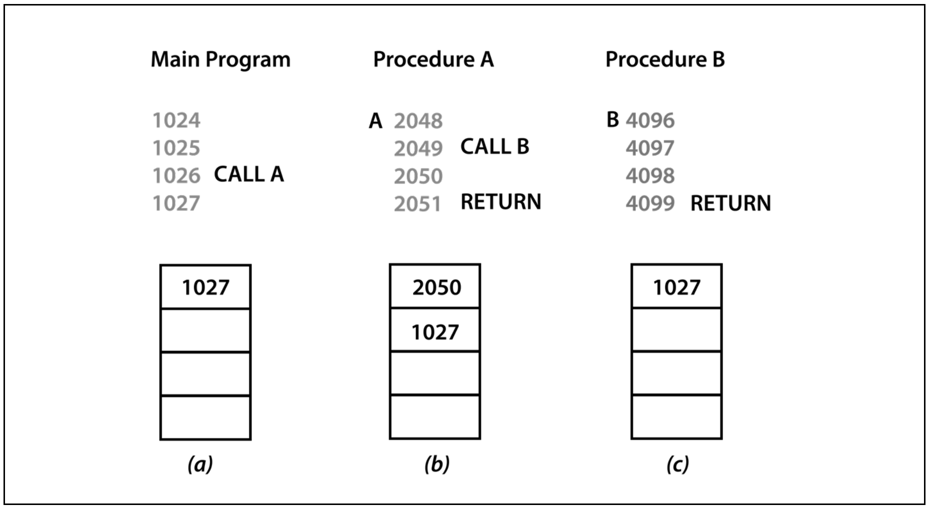

Figure 3-14
Procedure call and return using a stack

Procedure B could call yet another procedure because of the flexibility of using a stack to hold return addresses; it's unnecessary for this example.

When Procedure B executes the RETURN instruction at location 4099, the address at the top of the stack, 2050, is popped off the stack and stored in the program counter. The next instruction to be executed will come from location 2050, immediately following the CALL B instruction of Procedure A. Figure 3-14*(c)* shows the stack *after* the return instruction has been executed. Similarly, the return instruction in Procedure A will transfer control to location 1027, in the main program and immediately after the CALL A instruction.

Many computer architectures, including the Intel x86 and x64, include a **stack pointer** register to allow manipulation of the stack address at register speeds. Architectures that support stacks generally include POP and PUSH instructions in addition to CALL and RETURN. The presence of POP and PUSH allow arbitrary data, not just return addresses, to be stored on the stack. In that way, arguments for procedure calls can be passed on the stack and results returned.

The data passed to a procedure or method is called a **stack frame**, and consists of more than the return address. Operating systems prescribe a set of **calling conventions** that tell how arguments, return values, and return addresses are to be managed, and so describe the stack frame for that OS. High-level language processors are written to follow the calling conventions of the OS on which they're running.

Because the stack grows and shrinks as it is used, it is generally placed at the highest address in a program's memory space and grows downward.

3.6 Modern CPU Architecture

The von Neumann architecture of Figure 3-2 is how computers were designed and built in the middle of the 20th century. It still serves as a useful abstraction for thinking about the function of modern computers: a memory a CPU, and sequential execution of instructions. Modern CPUs behave that way from the outside looking in, but there are a lot of moving parts in the actual implementation. We've already seen that, instead of one main memory, there is a hierarchy of main memory and multiple levels of cache with complex protocols implemented in hardware to control the flow of data. CPU architecture has acquired an even greater level of complexity over three quarters of a century.

A "better" CPU would be smaller, more powerful, faster, and consume less electric power to make batteries last longer. Each of these presents challenges to the designers of CPU chips.

Making chips smaller and more powerful means making the transistors on the chip smaller. There are two difficulties with making transistors smaller. One is the photolithography process by which chips are made. The wavelength of visible light is too great to make the tiny transistors of modern chips. Chip makers are using extreme ultraviolet light, x-radiation, and immersion lithography in the chip making process to get features smaller than 20nm. [47] The other difficulty is that, as transistors get smaller, quantum effects overwhelm classical

47 A nanometer (nm) is a billionth of a meter.

semiconductor physics. The development of the FinFET [48] by Chenming Hu [49] and others overcomes quantum leakage and extends Moore's law for perhaps another decade. (Perry, 2020)

CPU designers' challenges

Making a chip run faster poses three challenges. The first two are power consumption and heat. If the CPU clock runs faster, more power is consumed, which is counter to the desire to use less power to extend battery life. Even worse, consuming more power produces more heat. Heat sinks and fans can remove some of the heat from desktop computers and to some extent even from laptops. The process is much harder in phones and wearable devices.

Figure 3-15
A nanosecond is this long

The third barrier is the speed of light. Light in a vacuum can travel a little under a foot in one nanosecond. [50] Electrical signals travel somewhat more slowly in the conductors of a chip. Each digital logic gate contributes its propagation delay to the time required to transfer a signal. Complex circuits have a large number of gates chained together. The CPU's clock must run "slowly" enough to be sure that the signals within the CPU have settled into valid states before a clock pulse commands storage of a result. In a CPU with a four GHz clock, that could be as little as 0.25 nanosecond.

Logical equivalence of hardware and software

Computing hardware and software are logically equivalent. (Tanenbaum, 1990) Specialized hardware can be built, if you have enough transistors, to perform

48 A metal oxide semiconductor field-effect transistor with a three-dimensional "fin" that helps overcome quantum effects in very small transistors.

49 Dr. Hu and others at UC Berkeley developed the FinFET under a DARPA grant around the turn of the century. Hu received the IEEE Medal of Honor in 2020 "for a distinguished career of developing and putting into practice semiconductor models, particularly 3-D device structures, that have helped keep Moore's Law going over many decades,"

50 Grace Murray Hopper used to demonstrate "nanosecond" with pieces of copper wire a bit less than a foot long.

any computation that can be performed in software. Conversely, software can be written to do anything that can be done with hardware. Software has been written to simulate entire computers. Such a software system is known as a **virtual machine**. The limitation, of course, is that there must be some actual hardware in order to run software.

It is the equivalence of hardware and software that allows chip designers to move functions formerly performed by software into the hardware. The CPUs of the 1970s and 1980s were microprogrammed. That meant a sequence of steps controlled by the CPU clock issued the register transfer commands that interpreted the "machine" instruction set. Microprogramming made it relatively easy to add new machine instructions. Some IBM mainframes of the 1970s had writable control stores, the storage for the microprogram; that meant new instructions could be added to existing machines already deployed to customers. CPU designers added machine instructions to each generation of CPUs to do more things that were needed by high-level languages. That meant a high-level language statement could be completed in one or a few machine instructions rather than several. The trade-off was that CPUs became increasingly complex.

3.6.1 RISC Computers

Beginning about 1980 a number of researchers, among them David Patterson at the University of California, Berkeley developed the concept of reduced instruction set (RISC) computers. They observed that the most complex instructions of current computers were used infrequently if at all, and reasoned that it made no difference if five machine instructions were needed to accomplish what formerly needed only one if the new machine could run ten times as fast.

RISC machines did run much faster and there were several commercially successful RISC designs, among them the MIPS architecture, the ARM architecture, the Sun SPARC, and the IBM POWER architecture.

Mainstream processors, including IBM mainframes, the Motorola 68000 line, and the Intel x86 family were trapped by the need for backward compatibility. They could not jettison the complex instructions already built into their CPUs without sacrificing backward compatibility.

The RISC principles did influence the evolution of even those systems trapped by the need to maintain compatibility. The RISC principles included:

- Simple and regular instruction formats, which made possible…
- Fast hardwired (*i.e.* digital logic) control units in place of microprogrammed control.
- Many general purpose registers, which made possible…
- Fewer memory references.
- Aggressive pipelining to allow…
- Completion of one instruction per clock cycle.

Designers of complex instruction computers (CISC) were able to incorporate many of the lessons learned from RISC designs to improve performance.

3.6.2 Hardwired Control Units

The complex instructions required for backward compatibility in some computer families were possible because of microprogrammed control units. With more transistors available because of Moore's Law, it is often possible to design hardwired control units that will execute those instructions. It is possible to profile a large sample of programs to determine which instructions are executed most frequently. If the full instruction set cannot be implemented in hardwired control, designers can implement those frequently-used instructions in hardwired control and the remainder with microprogrammed control. The instructions needed for backward compatibly will run more slowly, but they will be available, and the most frequently-used instructions can run at hardwired speeds.

3.6.3 Pipelined execution

Pipelining in a CPU is similar in operation to an automobile assembly line. Although it can take 18 hours or more to build an automobile, an assembly line can produce a finished car every one to two minutes because many cars are in different stages of assembly at the same time.

In computer architecture, pipelining means starting a new instruction before the current instruction finishes to provide **instruction level parallelism**. The simplest approach to pipelining, and one of the earliest to be adopted, is instruction pre-fetching. Because of the nature of the von Neumann architecture, almost always the next instruction needed will be the one following the current instruction. Even in the case of an unconditional branch, the next instruction address is known as soon as the current instruction is decoded.

The obvious next step is a separate decode stage, so that a three stage pipeline might have one instruction in the execute phase, the next one being decoded, and the one after that fetched. More pipeline stages tend to make each state less complex, and so potentially faster. Ten-stage pipelines are not uncommon, and one Intel chip had an astonishing 31-stage pipeline. Instructions may still take several clock cycles, but the goal is to finish one instruction every clock cycle, much like an automobile assembly line finishes one car every minute or two.

Pipelined computers introduce the problem of **hazards**, conditions which would cause incorrect results if the pipeline kept running. There are three general types of hazards. **Data hazards** occur when data from an earlier instruction, not yet complete, are needed, and when completing an instruction would overwrite data still needed. **Structural hazards** occur when a functional unit of the CPU is needed, but is already in use. Structural hazards are also called resource hazards. **Branch hazards** occur when incorrect instructions are brought into the pipeline as the result of a branch. Branch hazards are also called control hazards. For data and structural hazards, sometimes the only option is to **stall** the pipeline until the needed data or resource becomes available. In the case of a branch hazard, if the wrong instruction stream has been fetched, the entire pipeline must be **flushed** and restarted at the correct address.

Branch hazards occur because the next address is not known until the execute phase of the fetch-decode-execute cycle. Sometimes the instruction fetch unit will have fetched the wrong instruction, and the pipeline would have to be flushed and the instruction could be fetched. That led to attempts at **branch prediction**. A simplistic approach is to predict that a backward branch will be taken and a forward branch will not. The rationale is that a backward branch is part of a loop and a forward branch is an exception handler. Branch prediction became more sophisticated with the implementation in silicon of branch history tables.

Starting the instruction at the predicted branch target is called **speculative execution**. If the branch prediction was incorrect, then the work done in speculative execution must be undone and the pipeline flushed. If enough resources are available, the hardware could start execution of both possibilities. Once the branch target is known, the work done on that branch is kept and the work done on the other branch is discarded. Executing both branches and selecting which one to keep is called **eager execution**.

3.6.4 Many Registers

Some types of data hazards can be avoided if there are more registers for storing data. Adding more registers seems like an easy thing to do until one considers that current applications were written with a particular set of registers in mind. The Intel x86 family of computers had eight 32-bit registers, and software was written to use those registers. A change in architecture does provide an opportunity. The Intel x64 architecture added eight more registers and doubled the size of each to 64 bits. Those registers went unused until software was rewritten for the 64-bit CPU. The Intel Itanium processor had 128 64-bit registers. Because it could not run software written for the x86 processor family, it was never a commercial success, and Intel stopped taking orders for the processor in January, 2020.

However, there's a way to provide more registers, and thus reduce memory accesses without rewriting software. It is called **register renaming**. Consider the following snippet of TBC code:

```
LDA   A
ADD   B
STO   A
LDA   C
```

That code adds B to A, then stores the result back into memory location A. If TBC were pipelined, the load from location C could not start until the store completed, and the pipeline would have to stall. If TBC had more than one general purpose register, a clever programmer could have used a different register for that second load, allowing it to start before the store completed. But it doesn't.

In a CPU with register renaming, the hardware would detect that there the LDA C instruction has no **dependence** on the previous contents of the accumulator and use a **shadow register** to start the load. When the store completes, the hardware would **rename** the shadow register as the accumulator so that subsequent instructions get the newly loaded value.

The AMD Zen processor executes the x64 instruction set, and so appears to the programmer to have 16 64-bit registers. The hardware has 168 64-bit integer shadow registers and 160 128-bit vector floating point shadow registers.

3.6.5 Superscalar Architecture

Even with pipelining, branch prediction, shadow registers, and speculative execution, a CPU can complete at most one instruction per clock cycle because there is only one arithmetic logic unit. Needing a computational resource that's not available results in a structural hazard, at which the pipeline must stall. A CPU with superscalar architecture might have not a single ALU, but two integer units, a floating point unit, and a load/store unit. Given the right instruction mix, such a CPU could complete four instructions on every clock cycle. Hardware that can check for dependencies can execute instructions **out of order** provided that no instruction depends on the results of an instruction to be executed later.

3.6.6 Multiple Cores

Clock speeds on the fastest production processors are around five GHz, with three to four GHz being more usual. Power use, heat, and propagation time make higher speeds very difficult, but Moore's Law has given designers a lot of transistors. With limits to clock speed, CPU designers did those things discussed above to try to complete one instruction every clock cycle. With a four GHz clock, that meant four billion instructions every second.

To go even faster, designers placed more than one complete CPU in the silicon. When a processor has more than one CPU, they are called **cores**, and the entire package is called a multi-core CPU. Each core has its own L1 and perhaps L2 cache and can complete instructions at the full clock speed.

In theory, if one has four cores, each running at four GHz, one has a CPU capable of completing 16 billion instructions every second. In practice, it's not that neat. In order to get 16 billion instructions per second, the computational job must be completely partitionable, in other words, 16 separate jobs that can run independent of one another. Not all tasks are partitionable. In the context of software development, Frederick Books (1975) pointed out that bearing a child takes nine months no matter how many women one assigns to the task.

Taking advantage of the parallelism offered by multiple cores requires attention at the level of programming. The hundreds of thousands of applications written before multi-core CPUs became available did not do that because there was no need. Some, particularly multimedia applications and scientific applications, have been rewritten to take advantage of multiple cores. The average application, if there is such a thing, will not run any faster on an eight-core CPU than on a single-core CPU of the same clock rate.

3.7 Summary

The von Neumann computer architecture was designed in 1945 by John von Neumann and others. It is characterized by a central processing unit consisting of an arithmetic and logic unit, a control unit, and registers. The CPU communicates with a memory and with input and output devices. Instructions are executed sequentially unless changed by a branch instruction. Modern computer architectures incorporate a great deal of internal parallelism, but the von Neumann architecture still serves as a useful abstraction of the 21st century digital computer.

There are several families of computer architecture, including CPU chips from Intel, the ARM architecture, and IBM's z/Series mainframes. Each architecture is characterized by an instruction set, which defines the programmer's view of the architecture. The microarchitecture of a CPU is the physical design.

The Little Man Computer is a hypothetical computer used as an instructional aid, particularly in understanding the instruction cycle. The fact that the Little

Man Computer is a decimal computer and all real computers are binary obscures some important points about the relationship between computers and binary numbers.

The Tiny Binary Computer (TBC) is a binary computer that can execute the same instructions as the Little Man Computer. It has a very simple arithmetic and logic unit and five registers. The arithmetic and logic unit is constructed of digital logic.

Real computers have more complex instruction formats and addressing modes beyond the direct addressing of TBC.

Modern computers use different memory technologies for different purposes. Memory speeds for the dynamic RAM used in main memory have not kept up the speed of CPUS. Multiple levels of higher-speed cache memory between main memory and the CPU attempt to compensate for the difference.

Most modern general purpose computers have byte-addressable memories with addresses supplied as row and column addresses. Some memory systems have error detection and correction technology.

The CPU of the 21st century has many registers, even though most of them may not be accessible to the programmer or compiler writer. It has hardwired control for the frequently executed instructions, but may keep microprogrammed control for particularly complex or particularly infrequent instructions. It has an aggressive pipeline with branch prediction and speculative execution. It has multiple functional units, permitting multiple instructions to be executed in parallel. Perhaps most important from the user's viewpoint, it has multiple cores.

3.8 References

Brooks, F. P. (1975). *The Mythical Man Month.* Boston, MA: Addison-Wesley.

Cohen, D. (1981). On Holy Wars and a Plea for Peace. *IEEE Computer*, 14(10), 48-54.

Davidson, J., & Fraser, C. (1980, April). The Design and Application of a Retargetable Peephole Optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2), 191-202.

Perry, T. S. (2020, May). How the Father of FinFETs Helped Save Moore's Law. *IEEE Spectrum*, 57(5), pp. 46-52.

Tanenbaum, A. (1990). *Structured Computer Organization Third Edition.* Upper Saddle River, NJ: Prentice-Hall.

Tanenbaum, A. S. (1978, March). Implications of structured programming for machine architecture. *Communications of the ACM*, 21(3), 237-246.