CHAPTER 4
Machines

Peter J. Denning
Craig H. Martell

DRAFT v10-- 7/17/14

A person with paper, pencil, eraser, and subject to strict discipline, is a universal machine.
--Alan Turing

Machines may be the true humanizing influence. They do the work that makes life possible; human beings do the things that make life worthwhile.
--Isaac Asimov

Computer scientists are fond of abstractions. An abstraction is a mental model that captures the essential features of a thing and suppresses all other features. Computer scientists often describe programming as designing a hierarchy of abstractions represented as "abstract objects" operated on by designated functions. This notion has become so popular that computer science is often touted as the field that has learned best how to manage abstractions.

Computing abstractions differ in an important way from the mathematical abstractions common in other fields: computing abstractions perform actions. The terminology of abstractions often obscures the *principle of stuff*: the reality that computational actions are implemented as physical processes controlled by programs.

Consider for instance a musical song. On a computer, a song is represented by an MP3 file, which contains a digitized version of music from the publisher. To listen to the song, we activate a program "play" on the file. The "play" program encodes the millions of bits from the file in disk storage as signals that travel to the earphones, where sound-generator circuits vibrate diaphragms. At the abstract (user) level, the play program and the MP3 file appear as single objects: apply "play" to "song" and you hear music. The implementation is quite complicated, involving many steps, each of which depends on a physical process.

In this chapter, we examine how to organize physical machines that compute functions. The allowable moves of the machine are expressed as single instructions, such as adding two numbers. A program is a series of instructions arranged in a precise way to cause the machine to evaluate the desired function. Instructions and data are encoded as binary patterns stored in a memory. When fetched into a processor, instructions cause the hardware to transform input data into output data.

In the earliest days of electronic computing, programmers wrote programs directly as binary codes arranged in sequences on paper tapes or cards. Programming languages quickly superseded binary coding because they were much less error prone. A special program called a compiler automatically translated statements of the language into binary machine code. In the next chapter, on programming, we discuss how a compiler does this.

The organization of a computing machine is often called its architecture.[1] An architectural specification covers the central processing unit (CPU), which executes instructions; the random-access memory (RAM), which contains the program code and the data,[2] and the data structures used to organize program components in memory.

Machines

A machine is an apparatus for using or applying energy to perform a particular task. Machines are usually powered by mechanical, chemical, thermal, or electrical means. Electronic machines are powered by electricity with no moving parts -- for example, radio, television, mobile phones, and tablet computers.

An automaton is a self-operating machine. The cuckoo on a clock was once considered an automaton. So was The Turk chess player of the late 1700s (Standage 2003) (see figure 1). From the 1940s, computer scientists have thought of automata as abstract mathematical models of computers, and from the 1950s they believed automata embodied into software or robots have the potential for self-conscious thought.

Machines to aid calculation date back thousands of years. From 2700 BCE onward, merchants in Mesopotamia, Egypt, Persia, Greece, Rome, and China used the abacus to calculate sums. The Greeks showed how to measure the height of a tree by measuring its shadow and taking ratios with the shadow of a stick of known height; the stick and its operating procedure were a simple computing device. Another measuring stick, the slide rule, was invented around 1620 after John Napier published the concept of a logarithm; often called a slip-stick, the slide rule was a standard computing machine used by engineers until the 1970s, when the electronic calculator displaced it. In 1642 Blaise Pascal built a computing machine that added and subtracted numbers, and he presented algorithms for multiplication and division as repeated additions or subtractions. Charles Babbage designed the Difference Engine (1822-1842) to compute numerical tables of arithmetic functions; existing tables, calculated tediously by hand, were riddled with errors and posed great risks to navigators and other users. In 1911 the Marchant Company began selling mechanical calculators built from gears, pulleys, and levers that could add, subtract, multiply, and divide. In 1922, the German engineer Arthur Scherbius invented the Engima machine for generating ciphers; the Poles broke the code in 1932 and passed the information to the British, who used it to build the Bletchley code-breaking machine in the

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

early 1940s. In the late 1920s Vannevar Bush built the differential analyzer to solve differential equations by mechanical integration.

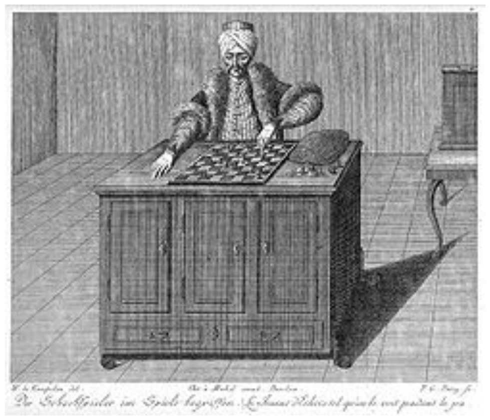


Figure 1. In his 1784 book *Inanimate Reason*, Karl Gottlieb van Windisch described The Turk, a chess-playing machine. Beginning in 1770 for the next 84 years, its various owners promoted it as an automaton that would play chess with anyone, winning most matches. It was an elaborate hoax. An expert chess player hid inside the cabinet, observed the pieces with mirrors, and used levers to move his pieces on the board. The illusion appealed to a deep human belief, perhaps a fear, that the human brain is a machine and most intelligent acts are actually mechanical moves. In 1997 the chess computer IBM Big Blue beat grandmaster Garry Kasparov. The reaction was not that the machine had become intelligent but rather that the machine searched faster than Kasparov.

In World War II the US Army commissioned teams of women at Aberdeen Proving Grounds to calculate ballistic tables for artillery. Gunners used the tables to determine the best gun direction and angle given the wind and range of the target. Following programs written on paper, the women operated mechanical calculators (such as the Marchant machines) to prepare these ballistic tables. Because the teams were error-prone and could not keep up with the volume of ballistic tables needed for the growing inventory of ordnance, the Army decided to replace the human calculators with electronic machines. They commissioned the first computing machine project, the ENIAC, at the University of Pennsylvania in 1943. The ENIAC could compute ballistic tables a thousand times faster than the human teams. Although the machine was not ready until 1946, after the war ended, the military made heavy use of computers after that.

It is interesting to note that in the 1920s, the term "computer" meant a person who calculated numbers. Thus, to distinguish them from actual, human computers, the first electronic computing machines were billed as "automatic

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

computers". The acronyms of the first electronic computers in the 1940s ended in "-AC" to signify this.

In 1937 Alan Turing defined a computer as a machine capable of calculating a mathematical function, and he discovered functions that cannot be calculated by any computer. He used the term *computable* for functions that could be calculated by computers. A function is computable if there exists a finite set of instructions that can generate its output value for any given input value (see figure 2). For example, addition is computable because we can specify a finite set of instructions that produce the sum $x + y$ given any numbers x and y. An unanswered mathematical question in Turing's time was how we could describe the set of computable functions. We examine this question more deeply in Chapter 6 on computation.

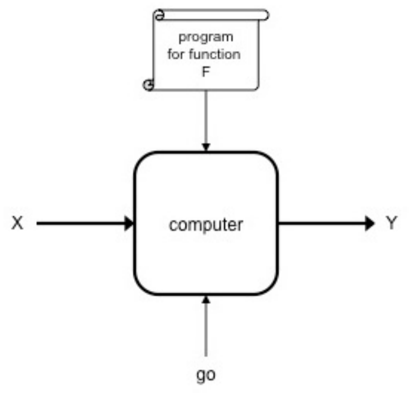


Figure 2. A computer is a machine that takes an input binary pattern *X* and calculates an output binary pattern *Y*. The computer is controlled by instructions from a program designed to calculate a specific function *F*. When a signal arrives on the "go" input, the computer starts to work and after a while stops with output *Y* = *F(X)*. The time required before the computer stops depends on the function and the program. Some programs may contain infinite loops, in which cases the computer will never stop. We can define a function *H(F,X)* that yields value 1 if program *F* halts for input *X* and 0 if it does not halt. Alan Turing proved that *H* cannot be implemented by any computer.

Turing argued that every computational method to calculate any computable function was based on the very simple operations of reading symbols, setting a control state depending what has been read, and writing symbols. He created an abstract machine, now called a Turing machine, which consisted of a control unit moving along an infinite tape, reading and writing symbols in the squares of the

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

tape. The specification of the control unit was the machine's program. Programs used loops to repeat steps as many times as needed. He also described a universal machine capable of simulating any other Turing machine given its program. And finally he showed the existence of functions that are well defined but not computable, such as the problem of determining whether a Turing machine will halt (come to a stop without going into an infinite loop). Although several others in his day also produced designs for computational machines and demonstrated them equivalent to Turing machines, Turing's design became the reference model because it most closely resembled the functions of real electronic computers, particularly the processor (control unit) and memory (tape).

The definition of computer as a machine that transforms an input pattern into an output pattern and then stops is not the only mode for using computers. Interaction is common. An interactive machine receives numerous inputs and generates numerous outputs and never stops. We noted in chapter 3 that an interactive machine, in cooperation with a human, could compute functions that a stopping computer could not.

Computing Machines

A computer is a machine controlled by a program that computes an output value from a given input value. Now we take a closer look at how we can build a machine that works this way.

A *stored-program computer* is electronic hardware that implements an instruction set. An *instruction* is a single arithmetic or logical operation carried out by the machine. An *operation* is a very simple, elementary function. Typical operations take two inputs and produce one output. For example ADD sums two numbers and EQ compares whether two numbers are equal; thus ADD(3,5)=8 and EQ(3,5)=0 (false). Instruction sets also contain branch instructions that control which instruction is next after the current one.

A *program* is a set of instructions arranged in a pattern that causes the desired function to be calculated. *Programming* is the art of designing a program and providing convincing evidence that the program computes its function correctly.

A *computing system* is a combination of program and machine. The program causes the machine to calculate a function. We can also say that the computing system calculates a function.

To make all this work, our computing system needs:

1. Precise specification of the set of instructions implemented by the hardware.
2. Precise method to represent a program as a series of instructions.
3. A memory that stores the program and the data on which it operates.
4. A control unit that reads and executes instructions of a program in the order prescribed by the program.

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

A CPU (central processing unit) is a hardware device that reads instructions from a program and executes them, one at a time, in the order prescribed by the program.

A RAM (random access memory) is a hardware device that holds data values in locations that can be read or written by the CPU. RAM is organized as a linear array of locations. Each location holds an elemental quantity of data, typically an 8-bit byte or a 32-bit word. The locations are numbered 0,1,...,2^n-1, where n is the number of bits in an address. RAM is called "random access" because it can access any random location in the same amount of time. Locations only hold only binary patterns (of 8 or 32 bits). The RAM does not attempt to interpret patterns; it simply stores and retrieves them reliably. The time required for the memory to respond to a CPU read or write request is the *memory cycle time*, today typically just a few nanoseconds. A block diagram of the CPU and RAM is shown in figure 3, and one of the interface between CPU and RAM in figure 4.

Real computers have memory other than RAM, for example a disk. Disk access times are random variables, depending on seek and rotation delays of moving magnetic media. The additional problems of moving data among multiple types of memory are considered in chapter 7 on memory.

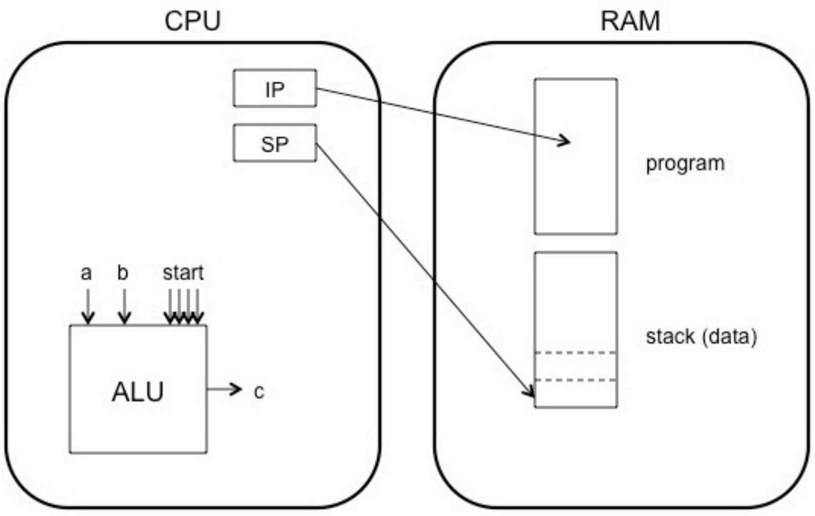


Figure 3. The hardware of a computing system consists of a central processing unit (CPU) and a random access memory (RAM). The program and its data are in the RAM. The data are arranged as a stack, which means that new values are added only to the top of the stack and values are retrieved only from the top. The CPU contains two special registers. The instruction pointer (IP) is the

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

RAM address (in the program) of the next instruction to be executed. The stack pointer is the RAM address of the top of the stack. The CPU also contains an arithmetic-logic unit (ALU), which takes two input numbers (*a* and *b*) and produces one output number (*c*). A series of start lines signals the ALU which operation to perform, for example, add, multiply, or test equality.

The CPU uses the instruction pointer (IP) register to keep track of which instruction is next to execute. It executes instructions of a program by repeating the following cycle until it comes to an exit instruction in the program:

1. **fetch** instruction is addressed by IP and set IP=IP+1
2. **decode** by reading the operation code contained in the instruction
3. **execute** by carrying out the operation
4. **check** for interrupts: error conditions that might have arisen during the previous steps

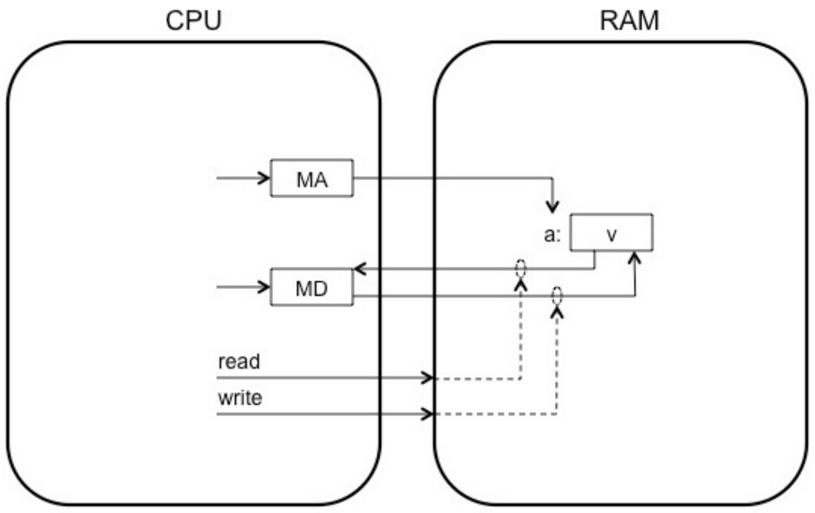
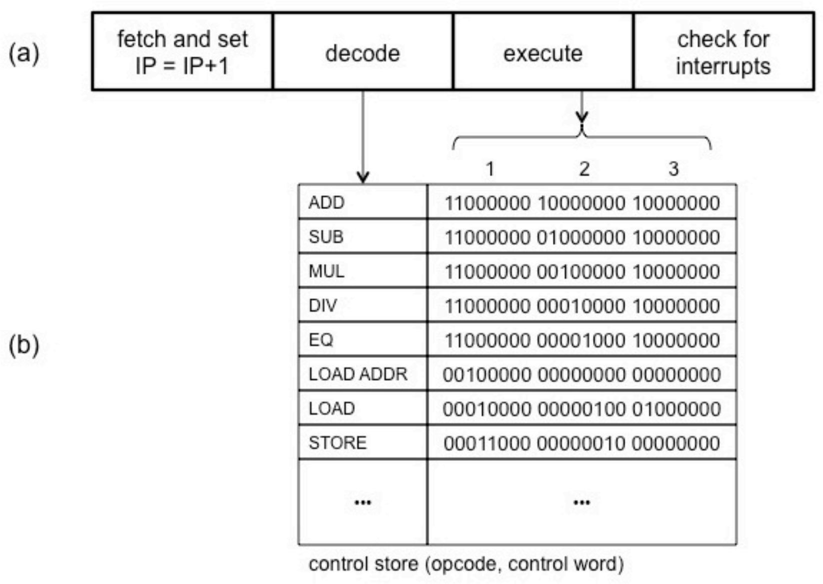


Figure 4. The CPU-RAM interface consists of several components. The objective is to read or write a particular location (*a*) in the RAM; a read operation transmits the value *v* in the selected location to the CPU, and a write operation transmits a new value from the CPU to the selected location. The memory address register (MA) tells which location is selected. The memory data register (MD) holds the value. The read signal line tells the memory hardware to select an address (in MA) and copy its value to the CPU (in MD). The write signal line tells the memory hardware to copy the value from the CPU (in MD) to the location selected (by MA). The time required to do these operations is called the memory cycle time, under 10 nanoseconds in modern RAMs.

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

The CPU contains a clock that issues a signal once every clock tick. The clock signal propagates through the CPU and activates selected circuits. A typical clock tick interval is around 0.5 nanosecond. It takes four ticks to sequence the CPU through the four steps of an instruction cycle. The length of the clock tick interval is chosen to allow all the circuits involved in a step of the instruction cycle to settle into a new state. If the clock tick is too short, some circuits will not have had time to settle and the CPU will malfunction. Figure 5 illustrates how the CPU decodes and executes instructions, and figure 6 illustrates how the ADD component of the CPU's Arithmetic-Logic Unit (ALU) works.

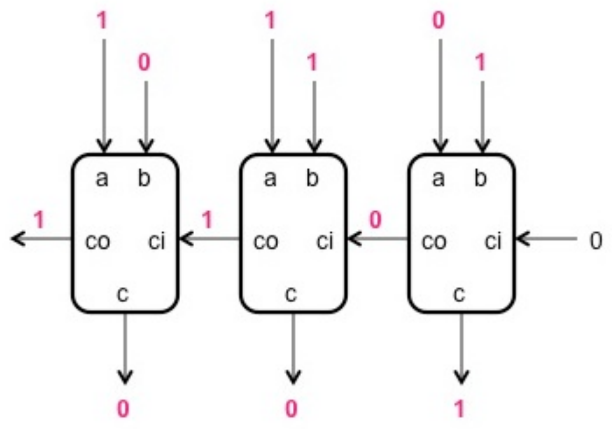


ADD	11000000 10000000 10000000
SUB	11000000 01000000 10000000
MUL	11000000 00100000 10000000
DIV	11000000 00010000 10000000
EQ	11000000 00001000 10000000
LOAD ADDR	00100000 00000000 00000000
LOAD	00010000 00000100 01000000
STORE	00011000 00000010 00000000
...	...

Figure 5. CPU instruction cycle consists of four phases (a). At each clock tick, the CPU advances to the next phase. The first phase gets a copy of the current instruction from address IP (instruction pointer) in RAM and sets IP to the next instruction. The second phase takes the operation code bits from the current instruction and interrogates a local control memory to get a control word for that instruction. In this example the control word is broken in three eight-bit blocks, corresponding to three subticks that occur between two regular clock ticks. We designate bits by their block and position; thus, bit 1.1 is bit 1 of the first block. At a subtick, each of the eight bits in a control word block activates a logic circuit; up to eight things can happen in parallel. The first five example instructions assume that the two operands are in registers R1 and R2. Bit 1.1 copies R1 to the "a" input of the ALU, and bit 1.2 copies R2 to the "b" input of the ALU. The first five bits of the second block send an appropriate trigger signal to the ALU telling it to add (2.1), subtract (2.2), multiply (2.3), divide (2.4), or test-for-equal (2.5). The first bit of the third block (3.1) copies the ALU output to register R1. The other three instructions activate different paths. LOAD ADDR activates 1.3, which says "copy the address bits from the instruction word to R1". LOAD activates 1.4, 2.6, and 3.2,

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

which say: "copy R1 to MA (memory address register)", "activate memory read", and "copy the MD (memory data) to R1". Finally, the STORE instruction assumes R1 contains an address and R2 a value; it activates 1.4 and 1.5 in parallel and then 2.7, meaning: "copy R1 to MA", "copy R2 to MD", "activate memory write".



ci, a, b	d, co
0 0 0	0 0
0 0 1	1 0
0 1 0	1 0
0 1 1	0 1
1 0 0	1 0
1 0 1	0 1
1 1 0	0 1
1 1 1	1 1

Figure 6. To add numbers, we follow a simple algorithm that sums the 1s digits, then the 10s digits, then the 100s digits, and so on, and occasionally transfers a carry of 1 to the next higher stage if a digit sum is bigger than 9. For example, to add 17 and 26, we start by adding the 1s digits 7 and 6, giving a sum digit 3 with carry 1. Then we add the 10s digits 1 and 2 plus the carry, giving a sum digit 4 with carry 0. The answer is 43. The same algorithm simplifies for the binary number system because sums can be only 0 or 1. The figure depicts a three-bit binary adder (*left*). The inputs are numbers *a* and *b*, and output is number *c*. The carry transfers (*co* = carry out, *ci* = carry in) from one stage to the next higher. The right stage sums the 1s bit, the next stage the 2s bits, and the third stage the 4s bits. Some bit combinations produce a carry; for example 1+1 = 0 with carry 1. The table (*right*) shows the output combinations of a stage for all possible input combinations. We use the carry from the leftmost stage (*co*) as a fourth bit of the output because some sums are greater than 7 (the largest number that can be represented in three bits). The largest sum would be 111 (=7) + 111 (=7) or 1110 (=14). Each stage is implemented with a few transistors. The sum is available as soon as all the stages settle; the worst-case settling time occurs when a carry propagates the entire length of the chain. The same structure is used for larger numbers; thus, a 32-bit machine represents numbers with 32 bits and uses 32-stage adders. In most computers the adder is a component of a larger arithmetic logic unit (ALU) that performs add, subtract, multiply, divide, and logical test operations such as equal, not equal, or less than.

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

The conclusion from this brief introduction is that we can design electronic circuits that will cause a machine to calculate a function by executing a sequence of instructions. The design outlined here was created for the first electronic computing machines in the 1940s -- at the University of Pennsylvania, MIT, Princeton, and Cambridge. John von Neumann, a mathematician working with some of the engineers, wrote up descriptions of the design. Because of his writings, the design is often called "von Neumann architecture", although the design was actually invented by the engineers J Presper Eckert, John Mauchly, Hermann Goldstein, Arthur Burks, and others.

Many other architectures are possible for computers. The common feature is that they completely automate the process of following programs of instructions to calculate functions.

Programs and Their Representations

The previous discussion might give the impression that a program is any sequence of instructions from the machine's instruction set. That is not so. Programs have to obey precise rules of structure. There can be absolutely no ambiguity about what each individual instruction does and what the whole pattern does. Otherwise, we could not attain reliable computers that give the same answer for the same input every time.

Let us outline a design for programs. Generally when we calculate numbers we do three kinds of things:[3]

1. Perform instructions in a strict sequential order (sequencing);
2. Make a choice between two alternative calculations based on the outcome, true or false, of a test (choice).
3. Repeat a calculation many times until a test says to stop (iteration).

Notice that the iteration pattern opens the possibility of an infinite loop because the test might never be satisfied.

A *programming language* is a set of syntax rules describing a precise notation for each of the above structures. There are thousands of programming languages. Despite the diversity of possible computer languages, they all have a single purpose: to describe how a computing machine can be made to evaluate a specific function.

When we design programs, we think of a pointer moving through the program steps and the machine doing each designated instruction, one at a time. The pointer is called the instruction pointer (IP). The CPU implements the IP as a register containing the address of the RAM location of the next instruction to be executed (see figure 3). When we are done with an instruction, we normally go to the next instruction in sequence (IP+1) unless a control instruction redirects IP. For example, an instruction "GO 17" sets the IP to 17 so that the CPU next executes the instruction at memory location 17.

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

The next step in our story about machines that execute programs is to show how to design an instruction set that supports any program conforming to the three-part structure above. That is the subject of the next section.

Stack Machine: A Simple Model of a Computer System

For a thousand years students of algebra have been told that arithmetic operators have orders of precedence: all multiplications and divisions are done before additions and subtractions. A series of operators of the same order are evaluated left to right. These rules ensure that all expressions evaluate the same, no matter who does the evaluation. For example, 1+2*6/4-2 would be evaluated by applying operators one at a time starting with the highest precedence:

1+2*6/4-2

1+12/4-2

1+3-2

4-2

2

More advanced students also learned that there is a third precedence level, exponents and logarithms, which are performed before multiplications and divisions.

Students of algebra are also taught that algebraic terms can be grouped within parentheses to force groupings not implied by the rules of precedence. For example, grouping the last two terms in the previous expression results in a different outcome:

1+2*6/(4-2)

1+12/(4-2)

1+12/2

1+6

7

In 1920 Polish logician Jan Lukasiewicz invented a new notation, now called Reverse Polish notation (RPN), which followed the rules of precedence and avoided parentheses.[4] The idea was to follow two numbers by the operator that combines them in the expression. In his notation the two expressions above respectively become

1 2 6 * 4 / + 2 –

1 2 6 * 4 2 - / +

Early in the days of computer science someone noticed that Polish notation expressions could be evaluated on a stack. A stack is a last-in-first-out memory structure. You read the Polish expression from left to right, pushing numbers on the stack as you encounter them; and you perform operators on the top two numbers, replacing them with the result. For example, the series of stack

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

configurations for the first expression is as follows (with top of stack on the right):

1	(Push 1 onto the stack.)
1 2	(Push 2 onto the stack.)
1 2 6	(Push 6 onto the stack.)
1 12	(Pop 2 and 6, multiply them, then push product 12.)
1 12 4	(Push 4 onto the stack.)
1 3	(Pop 12 and 4, divide 12 by 4, then push quotient 3.)
4	(Pop 1 and 3, add them, then push sum 4.)
4 2	(Push 2 onto the stack.)
2	(Pop 4 and 2, subtract, then push difference 2.)

The Burroughs B5000 machine (1961) organized its memory around a stack and achieved a highly efficient method of evaluating expressions (Organick 1973). The English Electric KDF9 (1963) used a stack structure. The Hewlett Packard scientific calculator HP-67 (1972) used the same structure because it reduced keystrokes and errors when evaluating complicated expressions. Modern HP calculators continue to use the stack structure. Numerous programming languages, beginning with Algol (1958), were designed on the assumption that the underlying machine had a stack memory. Modern multicore computing chips use stack memory for subroutine calls. Modern compilers use CPU machine registers to simulate pushdown stacks for evaluating expressions. The stack memory structure is ubiquitous.

Table 1 is an instruction set for a CPU-RAM configuration as depicted earlier in figure 3. The "Op Code" is an abbreviation for the name of the instruction. The effect of executing the instruction is shown in the "Before" and "After" columns, which show the stack configuration just before and just after the instruction is executed. The letter "S" represents the state of the stack prior to the current instruction. Mem[*a*] means the contents stored in memory location *a*. Essential side effects of changing the instruction pointer and changing the contents of a memory location are shown in the "Memory Effects" column.

Table 1: Instruction Set of Stack Machine

Type	Op Code	Name	Before	After	Memory Effects
Arithmetic and logical operators	ADD	Add	S a b	S c	
	SUB	Subtract			
	MUL	Multiply			
	DIV	Divide			
	EQ	Test for equal			
	NE	Test for not equal			

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

Memory interface	LA a L ST	Load address a Load Store	S S a S a v	S a S v S	 v = Mem[a] leaving Mem[a]=v
Sequencing	GO GOF	Go Go on false	S a S a v	S S	leaving IP=a leaving IP=a if v=0
Completion	EXIT	Exit	empty	empty	

Figure 7 is an example of a program in this instruction set evaluating an assignment statement that sets a variable *X* to the value of an expression.

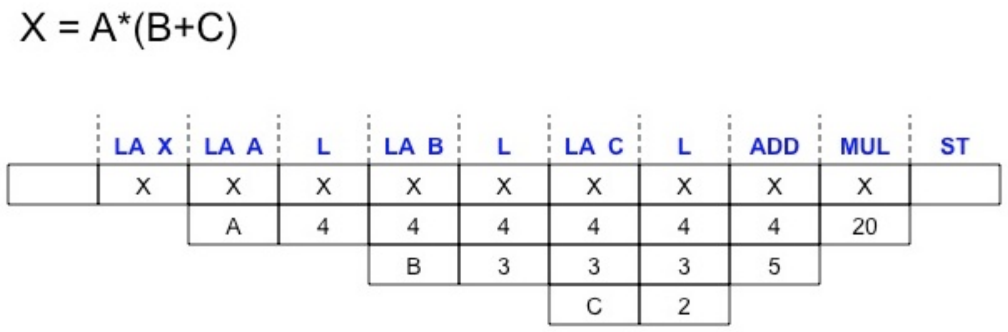


Figure 7. This series of snapshots shows the stack as it executes a program implementing the statement X=A*(B+C) when A=4, B=3, and C=2; when the program is done, memory location X=20 and the stack is empty.

Procedures and Exceptions

The machine's instruction set contains instructions that control the sequencing of the CPU as it moves through a program. Programs in higher-level languages require more sophisticated sequencing control because they allow programmers to write their own functions beyond those in the instruction set and to write functions that deal with errors and other events requiring special attention. The basic structure for both cases is the procedure call and return mechanism. The purpose of the procedure mechanism is to transfer the CPU to the first instruction of another program and, when the called program is done, to return the CPU to its calling point.

The designers of the first stored program computers realized that programmers would want to add functions of their own design, implemented as new subprograms that can be invoked with the same ease as machine instructions. A subprogram mechanism allows a programmer to call a

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

subprogram from wherever it is needed rather than rewrite its code at that point in the program. It also allows experts to create libraries of standard functions, such as trigonometry or algebra, which can be used reliably by anyone else.

Originally, in the 1950s, reusable subprograms were called *subroutines*. That name eventually gave way to "procedure" in the 1960s under the influence of the Algol language. A *procedure* is a subprogram that implements a single, usually simple, function.

The key idea of procedures is that a procedure is "active" only between the moment it is called and the moment it returns, and the data it needs while active are in a private segment of memory called an *activation record* (AR). A call allocates memory for the procedure's activation record, and a return reclaims it. When procedure calls are nested -- meaning that an active procedure can call another procedure, including itself -- there will be multiple activation records, one for each call. They will be linked together in the order of call, so that when one returns, its caller can resume from where it made the call (see figure 8.) Because returns occur in reverse order of calls, activation records are pushed on the normal stack on calls and deleted on returns (see figure 9.)

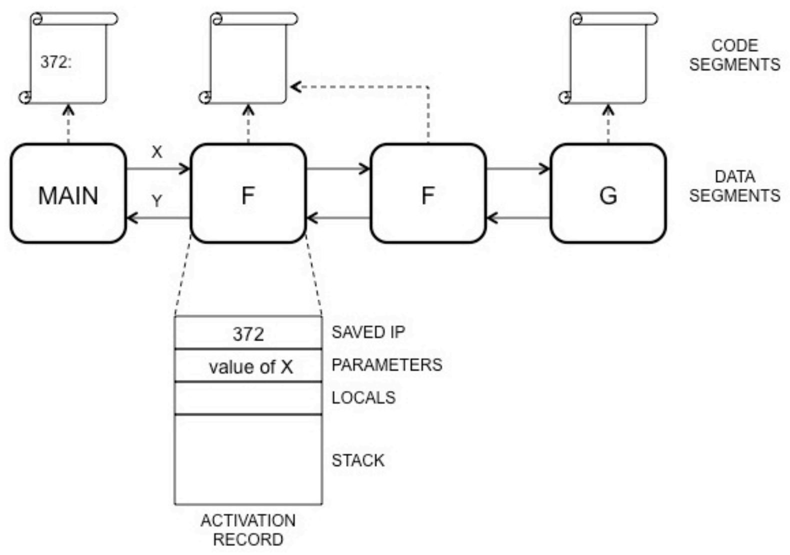


Figure 8. Programming languages accommodate procedures (separate subprograms) implementing functions. The *MAIN* program is treated as a procedure called by the operating system. In this example the *MAIN* program has called procedure *F*, then *F* called itself, and then the second *F* called *G*. While procedure *G* is active and executing, procedures *MAIN* and both calls to *F* are active but suspended. The right arrows represent the call actions and passing of parameters; for example, *MAIN* called *F* with parameter *X*. The left arrows represent the returns of values; for example the value *Y=F(X)*. Each procedure is implemented with a code segment and a data segment. The dashed arrows represent links to the procedure's code. When a procedure *F* calls itself, each instance gets its own data segment, and all instances link to the same code segment for *F*. The data segment is implemented as an

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

activation record that contains the saved instruction pointer (IP), the parameters of the call, local variables used only by the procedure, and a stack area used only by the procedure. The saved IP belongs to the caller; for example, the instruction at address 372 in the *MAIN* code segment called *F*, and when *F* is done the CPU instruction pointer is restored to 372. Because procedure activations are not known until a program is executed, the storage for activation records must be handled dynamically. A stack can be used for this purpose because deactivations (returns) occur in reverse order from activations (calls).

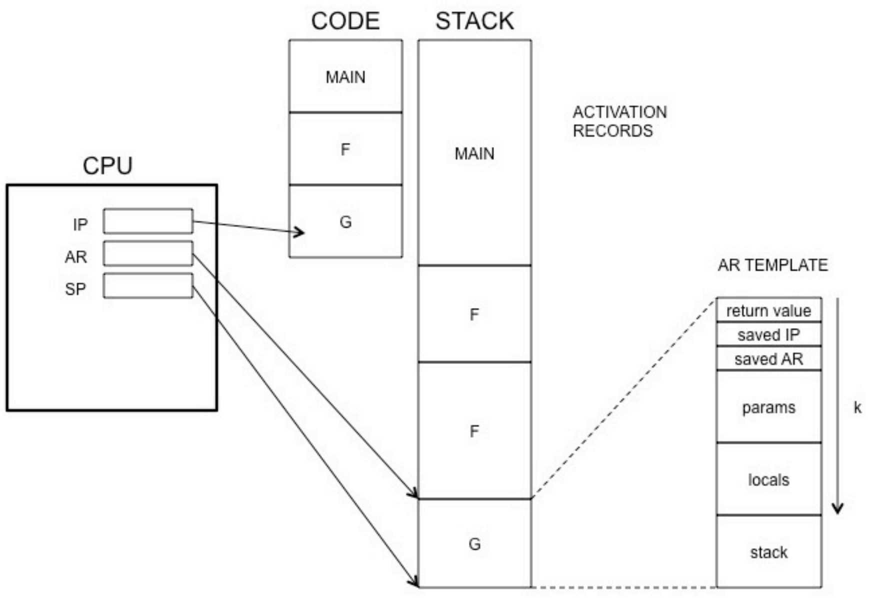


Figure 9. With a few modifications of the CPU, the activation records of an executing program (as in figure 8) can be stored on the program's stack. Procedure call pushes an activation record for the called procedure on top of the stack, and return pops it from the stack. The register AR points to the beginning of the current activation record. Just prior to the call, the caller code builds the new AR on the stack by loading values of parameters and local variables on the stack. At the call, the IP and AR registers are diverted to the called procedure, and their former values are restored at the return. Inside the called procedure, parameters and local values are found relative to the base of the activation record; for example, the first parameter is at address AR+3. The caller resumes with the value computed by the called procedure on top of the stack.

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

Consider an example of a call on the function LOG(Y). The purpose of the procedure call is to execute the code that computes $\log_2 Y$ and leave the result on top of the stack. To do this, the CALL instruction diverts the CPU to the code for LOG. The LOG code computes the result and places it in the reserved slot at the start of the AR. When the LOG code finishes, it executes a RET (return) instruction, which resumes the CPU at the next instruction after its CALL. These five steps give more details of how this happens:

1. The caller builds a new activation record for LOG in accordance with the LOG AR template. The template reserves one slot for the parameter (Y) and two slots for internal local variables. This is accomplished by a series of k=6 load operations to fill in those slots.
2. The caller places the target address for LOG on top of the stack. At this point, the base of the new activation record is precisely k slots below the stack pointer; in other words the new AR base is to be SP-k.
3. The caller executes the instruction CALL k, which does all of the following: save IP and AR registers in their reserved slots (at addresses SP-k+1 and SP-k+2, respectively), set register AR = SP-k, and pop the top of stack into register IP.
4. Now the CPU executes the code of the LOG function. That code will find the value of the parameter Y at location AR+3, and the two internal variables at AR+4 and AR+5. The code saves the computed value of LOG(Y) into the slot served for the return value, which is at address AR.
5. The called procedure executes the RET instruction, which sets SP to AR and restores the values of IP and AR from their saved locations. Now the caller resumes executing instructions after its call and the value of LOG(Y) is on top of the stack.

The procedure architecture described above allows recursive procedures, which are programs that can call themselves.[5] Lisp and Algol, first specified in 1958, were the first programming languages to incorporate recursion. Their designers did this because they wanted a language capable of expressing and executing any algorithm. Lisp expressed algorithms using Church's lambda calculus, and Algol expressed them with procedure notation consistent with recursive functions. Around 1960, Edsger Dijkstra proposed organizing the memory as a stack and built the first working Algol compiler. By comparison, Lisp compilers were much more difficult; it was not until the 1970s that efficient ones were available. Many programming languages since that time have provided for recursive procedures.

The procedure architecture turned out to be immensely useful not just for programming functions but also for dealing with errors during computations. For instance, what happens if a program attempts to divide by zero? Mathematically the result is undefined, and the program cannot give an answer. Rather than allow an undefined value to propagate through the program, CPU designers built the arithmetic-logic unit (ALU) to signal when this error condition occurs. They modified the CPU instruction cycle to check for this

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

signal; it is the fourth step of the instruction cycle described a few pages back. If the "divide by zero" error condition was set, the CPU used a procedure call to divert to a special subprogram that would either remove the error or abort the program. The action of diverting the CPU to the error handler was called an "interrupt." Elliott Organick (1973) characterized an interrupt as an "unexpected procedure call."

A divide-by-zero error is not the only reason for interrupting the CPU. Designers used the words *exceptional condition* for any event that requires immediate attention from the CPU. Exceptional conditions can be of two kinds: errors and external signals. An error is a condition in a program that would cause incorrect or undefined behavior. Examples of errors that can be detected by sensors in the CPU are divide-by-zero, arithmetic overflow and underflow, page fault, protection violation, or array reference out of bounds. An external signal indicates that a high-priority event has occurred. Examples of external signals are timer alarm, disk completion, mouse click, or network packet arrival. For any exception, the CPU is interrupted from whatever it was doing and put to work on dealing with the error or responding to the external signal. With the addition of an "interrupt vector", the CPU can automatically and rapidly invoke interrupt handler procedure *k* when the sensors report that exception *k* has occurred (see figure 10.)

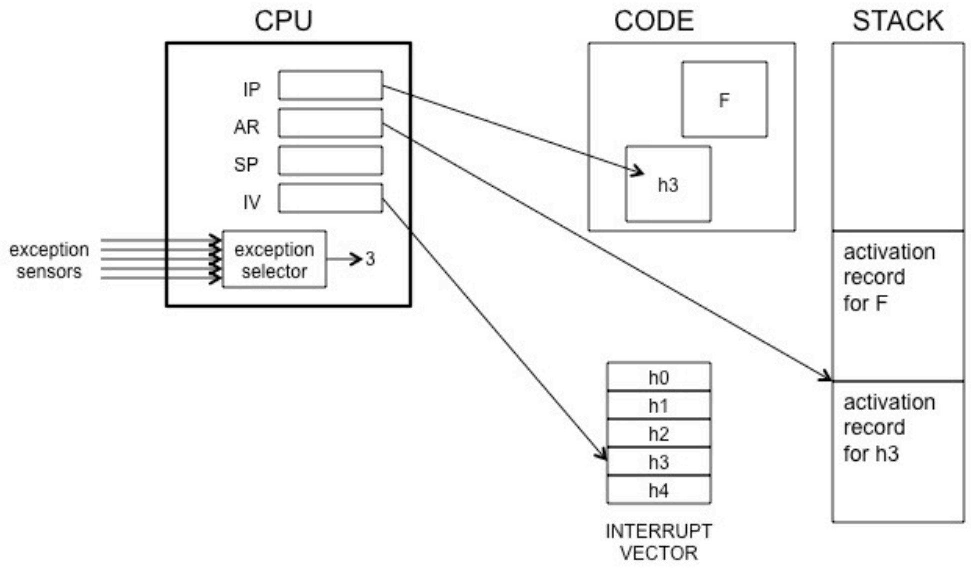


Figure 10. An interrupt mechanism enables the CPU to interrupt the current task and execute a procedure (handler) that resolves an error or responds to a high-priority external signal. Sensor circuits in the CPU and elsewhere in the computer system detect when any exceptions exist. A selector circuit selects the exception of highest priority and outputs its number (or 0 if there is no exception). At the end of each instruction cycle the CPU checks for exceptions, and, if there is one, the CPU suspends normal instruction execution. Instead it uses the exception number (here number 3) to index an interrupt vector, which

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

is a list of entry-point addresses for each of the handlers, and makes a procedure call (here on h3). The AR (here for h3) is pushed on the stack and the normal instruction cycle resumes. When the handler is done, its return instruction restores the interrupted program, which continues from where it was interrupted.

Choice Uncertainty

The interrupt mechanism opens the door to *metastability*, a subtle and potentially devastating error. What happens when an exception signal occurs at the same time the CPU is trying to read the flipflop that records the signal? The clock controls when the CPU looks for interrupts, but not when the external signal arrives.

Here is what happens. Suppose the circuits use 3 volts to represent 0 and 5 volts to represent 1. The arrival of the external signal triggers the interrupt flipflop to transition from the 0 to the 1 state, meaning that the flipflop's output voltage changes from 3 to 5 volts. Because that transition takes time, there is a small interval where the voltage is in between 3 and 5 volts but not close enough to either be reliably counted as 0 or 1. Electronics engineers call such an output a "half signal". A half-signal input can cause a flipflop to enter a *metastable state* with the output voltage poised midway between the two stable states. That midpoint is like a ball poised perfectly on the peak of a roof: it can sit there for an unknown amount of time until air molecules or roof vibrations cause it to lose its balance.

Metastability creates a risk of malfunction of any circuit that reads the flipflop's output. If the half signal persists beyond the next clock tick, the next circuit will receive an input that cannot be interpreted as 0 or 1, and its behavior may be unpredictable.

The metastability problem was well known to hardware engineers. Chaney and Molnor (1973) and Kinniment and Woods (1976) describe experiments to measure the likelihood and duration of metastable events. By synchronizing clock frequency with external signal frequency, they attempted to induce a metastable event on every external signal change. They saw frequent metastable events on their oscilloscope, some of which persisted 5, 10, or even 20 clock intervals (see figure 11.) Other engineers had known for a long time that chooser circuits, also known as *arbiters* because their job was to arbitrarily choose one of two simultaneous signals, were hard to build (Seitz 1980, Denning 1985, Ginosar 2003).

Since that time, chip makers have been concerned about the chances of metastable states in their circuits. Sutherland and Ebergen (2002) reported that contemporary flipflops switched in about 100 picoseconds (100 x 10^{-12} sec) and that a metastable state lasting 400 picoseconds or more occurred once every 10 hours of operation. Xilinx.com, a chip maker, reported that its modern flipflops had essentially no chance of showing a metastable state when clock frequencies

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

were 200 MHz or less, but faster clocks incurred metastable events (Alfke 2005). In experiments with interrupt signals arriving 50 million times a second, they observed a metastable state about once a minute at clock frequency 300 MHz and about once every 2 milliseconds at clock frequency 400 MHz. In a computer system generating 500 interrupts per second, about 1/100,000 of their experimental rate, these extrapolate to one interrupt-caused metastable state about every 2 weeks at 300 MHz, and about every 3 minutes at 400 MHz.

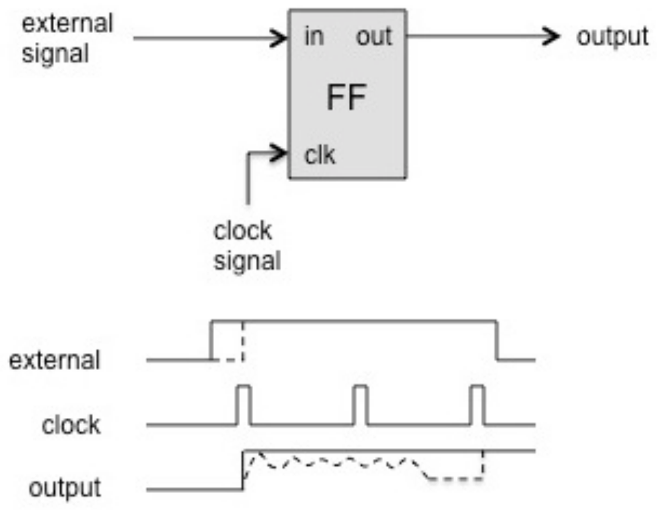


Figure 11. An experimental setup enables observing flipflop (FF) metastability. Each clock pulse signal triggers the FF state to match the input signal. If the input signal is changing when the clock pulse arrives (dashed external line), FF may enter an indefinite state that lasts several clock intervals (dashed output lines). In a digital computer, the indefinite output becomes the input of other logic circuits at the next clock pulse, causing half-signal malfunctions.

Now you can see the problem. There is a chance that the interrupt flipflop is metastable at the time when the CPU asks for the state, and that throws the next bank of flipflops controlling the CPU cycle into a metastable state. If those flipflops have not settled down by the next clock tick, the behavior of the CPU is unpredictable. The experimental results show that there is a good chance this can happen.

This problem plagued many early computer systems. Before engineers understood it, all they would see was that at random times the CPU would stop. They described these mysterious freezes as "cosmic ray crashes" because they seemed to be random disruptions of transistor function. Only a full-power-off

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

reboot would restart the CPU. Because they could occur every few hours or days, these freezes could be quite troublesome.

Around 1970 David Wheeler, a hardware engineer at the Computing Laboratory of the University of Cambridge, UK, discovered the reason for these mysterious freezes: half signals appearing at the output of the interrupt flipflop. He designed a new kind of flipflop, which he called a threshold flipflop, and a protocol for using it that eliminated the danger of CPU freeze on interrogating the interrupt flipflop (see figure 12.)

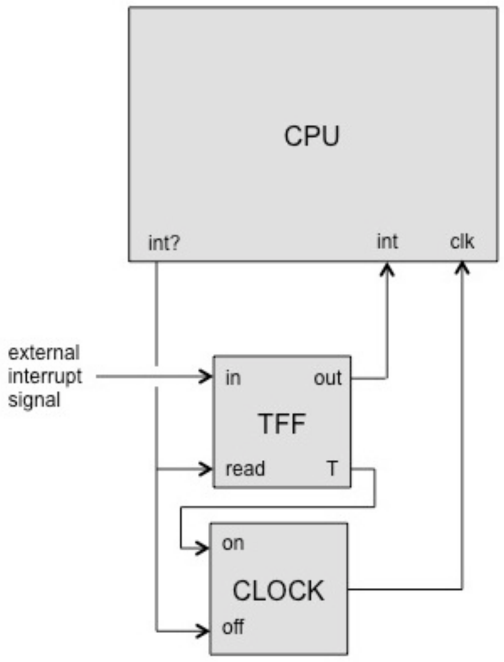


Figure 12. The threshold flipflop (TFF) guarantees that the CPU's interrupt input ("int") will be stable when interrogated at the end of an instruction cycle. When the CPU asks for the value of the external interrupt signal ("int?"), it triggers the TFF to record the current external signal in its state, and it turns the clock off. As soon as its state returns to "0" or "1", the TFF sends a pulse on the *T* output, which turns the clock back on. The clock suspension is only as long as necessary to assure that TFF is again stable. David Wheeler proposed the idea of temporarily shutting off the clock when checking interrupts in the Cambridge CAP computer in the 1970s.

Situations like the interrupt flipflop confront hardware engineers in many other parts of a computer. Circuits that must choose between near-simultaneous

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

events are everywhere. For example, at the same time, two CPUs access the same memory location, two transactions lock the same record of a database, two computers broadcast on Ethernet, two packets arrive together at the network card, an autonomous agent receives two requests, or a robot subsystem perceives two alternatives at the same time. In all these cases if we demand that a decision be made between the choices by the next clock tick, there is still a chance that the chooser circuits have not settled. If we want to wait for the circuits to settle, we need to stop the clock.

We can summarize these findings about chooser circuits with the choice uncertainty principle: "No choice between near-simultaneous events can be made unambiguously within a preset deadline"[6] (Lamport 1984, Denning 2007). The source of the uncertainty is the metastable state that can be induced in the chooser by conflicting forces generated when two distinct signals change at the same time.[7]

Choice uncertainty is not about how a system reacts to an observer but how an observer reacts to a system. It also applies to choices we humans make. What happens when the options are presented together and we are given a short time to choose? Sometimes we are still in indecision when the deadline comes, and we lose the opportunity presented to us. We do not have the option, as did Wheeler, to turn off the clock until we can decide. Individuals and groups can persist in an indecisive state for seconds, hours, days, months, or even years.

The possibility of indefinite indecision is often attributed to the fourteenth-century philosopher Jean Buridan, who described the paradox of the hungry dog that, being placed midway between two equal portions of food, starved (Lamport 1984, Denning 1985). If he were discussing this today with cognitive scientists, Buridan might say that the brain can be immobilized in a metastable state when presented with equally attractive alternatives.

Conclusions

Our purpose has been to show, in convincing detail, that it is possible to build an electronic machine that will calculate any function for which someone can find a computational method. The machine consists of a processor (CPU) and memory (RAM). In a repeating instruction cycle, the processor executes a sequence of machine instructions stored in RAM, operating on data also stored in RAM. The machine instructions implement simple operations including arithmetic, memory read and write, and control sequencing. Each instruction is implemented by a circuit in the CPU. We showed how to design a simple instruction set for the case where the data part of RAM is organized as a stack. Instructions for basic operations, choices, and iterations give the machine universal computing power.

The procedure-calling mechanism permits separately written programs to be invoked as procedure calls at any point within any program. On detecting exceptional conditions, operating systems use the procedure mechanism to interrupt programs.

We concluded with the choice uncertainty problem, which is that chooser circuits may be thrown into a metastable state by simultaneous inputs and be

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

unable to make a choice by a deadline such as the next clock tick. The problem arises from the physics of circuits and can be avoided if the clock is turned off until the circuits settle.

The study of machines reemphasizes the central importance of physical “stuff” in computation. All the instructions and data of a machine are recorded as patterns of 0 and 1 in physical circuits and media. The 0 and 1 are the names of states of the media. Instructions manipulate these stored states in precise, prescribed ways. Programs record the steps of computational methods as series of instructions arranged in precise patterns. The machine reads the program instructions and carries them out on the data. All this is done automatically. The circuits simply obey laws of electricity and physics; they have no understanding of the meanings of the signals passing through them.

The stack structure cited here is only one of several models for executing programs. Each model has its own rules and machine structures. But they all do the same thing: control electronic circuits that calculate output values from input values.

Bibliography

Author’s note: We have drawn heavily on seventy years of accumulated knowledge of computer architecture and compilers and have not attempted to provide a detailed historical account of the development of the ideas. The bibliography below lists a few items that have been most helpful to us. For readers who wish to dig deeper, we recommend searches of Wikipedia or the Internet.

Alfke, Peter. 2005. Metastable recovery in Virtex-II Pro FPGAs. Technical Report xapp094 (February), available from Xilinx.com website.

Bohm, Corrado, and Giuseppe Jacopini. 1966. Flow diagrams, Turing machines, and languages with only two formation rules. *Communications of the ACM 9*, 5 (May), 366-371.

Burks, Arthur, Don Warren, and Jesse Wright. 1954. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation 8* (1954), 53–57.

Chaney, T. J., and C. E. Molnor. 1973. “Anomalous behavior of synchronizer and arbiter circuits.” IEEE *Transactions on Computers 22* (April), 421-422.

Denning, Peter. 1985. “The arbitration problem.” *American Scientist 73* (Nov-Dec), 516-518.

Denning, Peter. 2007. The choice uncertainty principle. *Communications of the ACM 50*, 11 (November), 9-14.

Ginosar, R. 2003. “Fourteen Ways to Fool Your Synchronizer”. *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, IEEE (Feb), 8pp.

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

Available at
http://www.ee.technion.ac.il/~ran/papers/Sync_Errors_Feb03.pdf .

Harel, David. 1980. On folk theorems. *Communications of the ACM 23*, 7 (July), 379-389.

Hennessey, John, and David Patterson. 2011. *Computer Architecture: A Quantitative Approach.* (5th Ed.) Morgan Kaufman.

Kinniment, D J, and J V Woods. 1976. "Synchronization and arbitration circuits in digital systems." IEEE *Proceedings* (October), 961-966.

Kleene, Stephen. 1936. General recursive functions of natural numbers. *Mathematische Annalen 112*, 727-742.

Lamport, L. 1984. Buridan's Principle. Technical report available from http://research.microsoft.com/users/lamport/pubs/buridan.pdf .

Lukasiewicz, Jan. 1957. Aristotle's Syllogistic from the Standpoint of Modern Formal Logic. Oxford University Press.

Organick, Elliott. 1973. *Computer System Organization: B5700-B6700 Series.* Academic Press.

Seitz, C. L. 1980. "System Timing". In *Introduction to VLSI Systems* (C. Mead and L. Conway), Addison-Wesley, 218-262.

Standage, Tom. 2003. *The Turk: The Life and Times of the Famous Eighteenth-Century Chess-Playing Machine.* Berkeley Trade.

Sutherland, I. and J. Ebergen. 2002. "Computers without clocks." *Scientific American* (August), 62-69. Available from Sun Microsystems, available as http://research.sun.com/async/Publications/KPDisclosed/SciAm/SciAm.pdf.

Turing, Alan. 1937. On computable numbers with an application to the Entscheidungs problem. *Proc. London Mathematical Society 2,* 230-265.

von Neumann, John. 1993. *First Draft of a Report on the EDVAC.* In IEEE *Annals of the History of Computing 15*, 4 (1993), 27-43. (Original work published as a technical report from the US Army project at the University of Pennsylvania, 1945.)

End Notes

[1] The book by Hennessey and Patterson (2011) has excellent coverage of all aspects of computer architecture. The original architecture of the stored program computer is frequently attributed to John von Neumann

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.

(1993), who published notes of his meetings with Eckert, Mauchly, Burks, and Goldstine. Most of that architecture came from Eckert and Mauchly, not from von Neumann.

[2] IBM may have been the first to describe memory access as "random" with its new disk storage system RAMAC (Random Access Memory Accounting Machine) in 1956. Random meant that the time to complete an access was a random variable composed of seek time (arm positioning) and latency (rotational positioning). Today RAM refers to the main memory of a computer chip, but random means that the access time for any randomly chosen address is fixed, a different use of the word "random".

[3] Corrado Bohm and Giuseppe Jacopini (1966) proved that any computable function can be computed by a program limited to these three structures. This theorem was used as the basis of "structured programming", a movement to make programs easier to understand and prove correct. Some years later, David Harel (1980) traced this claim all the way back to the design of the von Neumann architecture itself and to a normal-form theorem proved by Stephen Kleene in the 1930s.

[4] Arthur Burks, Don Warren, and Jesse Wright (1954) are credited with being the first to notice that reverse Polish notation simplified mechanical evaluation of expressions. Fritz Bauer and Edsger Dijkstra are credited with independently discovering this in the early 1960s (Wikipedia).

[5] Recursion can lead to simpler programs. For example, it is possible to write a sort routine in the form SORT(list) = {SORT(left half of list); SORT(right half of list); MERGE(left half, right half)}, with the boundary condition SORT(empty list)=empty list. Each inner call to SORT must have a smaller input than the outer call.

[6] There is a superficial similarity with the Heisenberg uncertainty principle of quantum physics. That principle says that product of the standard deviations of position and momentum is at least 10^{-34} joule-seconds. Trying to reduce the uncertainty of one forces greater uncertainty of the other. Part of the reason for the Heisenberg principle is that the very act of observing either adds or removes energy from the particle being observed. But this only holds only at atomic scales of electrons and not at the macro scales of currents in wires. The choice uncertainty principle is not an instance of Heisenberg's principle.

[7] Asynchronous circuits (see chapter 8, Parallelism) are made of modules that interact with ready-acknowledge signals. They can be designed so that they will not generate ready or acknowledge signals while in a metastable state. They need no clocks. They are often faster than clocked circuits because modules "fire when ready" and do not have to wait for a next clock tick.

© 2012-2014, Peter J. Denning and Craig Martell. Permission needed to reproduce in any form.