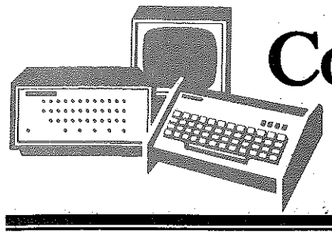


Computer Bits

By Hal Chamberlin

COMPUTER ARITHMETIC

ASK A LAYMAN what a computer does best and he will probably say that it is best at computing complicated mathematical formulas. However, ask the same question of a hobbyist who has obtained a computer for the purpose of mathematical computation and he will probably say that his machine handles text much better than numbers. The truth is that microcomputers have very little "number crunching" ability built-in. All that is normally available is addition and subtraction of 8-bit numbers and some can't even subtract directly! Automatic handling of decimal numbers is frequently provided also, but proper use of decimal arithmetic is far more complex than the normal binary arithmetic. As a result of this limited arithmetic capability, all other mathematical operations must be broken down into addition and subtraction of 8-bit numbers.

Arithmetic with integer numbers larger than the capacity of a computer word is termed "multiple precision arithmetic" with "double precision" used to signify the special case of two-word numbers. The words (bytes in an 8-bit processor) forming the number are simply strung end-to-end. A 24-bit number for example would consist of 3 bytes. The leftmost byte is called the "most significant" or "high" byte, and the rightmost is called the least significant or "low" byte. The bytes in the middle, if any, have no special name.

The most important multiple precision operations for general computation are addition, subtraction, multiplication, division, comparison, negation, incrementing, decrementing, and left and right rotates including the carry flag. Usually the software necessary to do these operations is organized into a subroutine package. It is a common practice in such software packages to define "registers" for multiple precision numbers in *main memory*. At least two registers are typically needed; the *left* operand, and the *right* operand, quite analogous to the equivalent operation done on paper. The package may be written for a specific number length such as 32 bits (4 bytes) or the length may be variable and passed to the subroutines as arguments. Besides the actual arithmetic subroutines, a "number move" routine is needed to conveniently move the multibyte data into and out of the pseudo registers. With a variable length arithmetic package, it becomes easy to do calculations to dozens or even hundreds of "decimal places" of accuracy in assembly language, which is much better than any BASIC language system.

Increment and Decrement. Incrementing and decrementing multiple precision numbers is probably the simplest of this type of operation. Assume for the moment that a 16-bit number, which is stored as two bytes in memory, is to be incremented by one. Although some machines may have an instruction to do this to a pair of registers, let's try to do it directly in memory using a 6502 microprocessor. For incrementing, the first step is to increment the least significant byte directly in memory using the INC instruction of the 6502. Next it is necessary to determine if an "overflow" of that byte occurred. This could normally be determined by looking at the carry flag but on the 6502, INC does not change the carry. Close examination of the overflow situation, which only occurs if the byte was equal to FF_{16} before incrementing, will reveal that after the overflow the byte will *always* be zero! Therefore, the Z-flag can be tested instead. If an overflow did indeed occur, then the most significant byte should be incremented; otherwise the job is done. This procedure can be extended to numbers of any length by continuing to move left, byte-by-byte, incrementing as long as the previous byte overflowed.

Multiple precision decrement is nearly as easy. First the low byte is examined to determine if it is zero. If it is not, it is decremented and the job is finished. If it is zero, a decrement will cause it to underflow. In that case, we decrement it anyway and then move left to the next byte and repeat the sequence. When the most significant byte is reached because of underflow of all previous bytes, it is simply decremented without any testing for zero. Note that these algorithms work equally well for *signed* two's complement multiple precision numbers.

Add and Subtract. Multiple precision addition and subtraction are more interesting. These operations require the left and right operand pseudo registers. The usual convention is to put the answer into the left operand register, much like arithmetic instructions themselves. The first step in double-precision addition is to clear the C flag and then, using the ADC instruction, add the low byte of the left operand to the low byte of the right operand and store the result back into the low byte of the left operand. Now, being careful not to disturb the carry flag, the ADC instruction is used to add the *high* bytes of the two operands together and store the result in the high byte of the left operand which completes the operation. For multiple precision one continues left adding pairs of bytes together with the ADC instruction until the most significant bytes are added.

Quadruple precision addition operation is shown below. The C-flag is used

Addition of two 4-byte numbers.

```
02791256      02  79   12  56
+0FE534B3     0F  E5   34  B3
125E4709     1↷  0↷   1↷  0←C=0
C=0← 012 ↶15E ↶047 ↶109
```

to transfer carry information from lesser significant bytes to more significant ones. One can actually think of it as adding multiple digit numbers together where each "digit" is a byte between 0 and 255. The carries are transferred from digit to digit just like decimal addition on paper. If care is taken on the return sequence from the add subroutine, except for Z, the status flags will correctly indicate the result of the operation just performed.

For subtraction, one could write a similar subroutine using the SBC instruction on each byte in the numbers. Before starting, however, it is necessary to *set* the C-flag for proper operation. Another way to do subtraction is to *complement* the right operand and then *add* it to the left operand using the add subroutine just described. Usually a complement routine is needed anyway, so this scheme can also save some memory.

A two's-complement operation consists of merely inverting the bits of the number and then incrementing the result. Thus a multiple-precision complement routine would invert the bits of each byte of the right operand and then call the multiple precision increment routine described earlier. An exclusive-OR of a byte with all ones, using an "EOR #$FF" instruction, is all that is necessary to invert it.

Comparison. Not all needed multiple-precision functions are involved with computing answers, some comparison operations are necessary also. Probably the most important of these is a comparison with zero since the Z-flag is not meaningful after a multiple-precision add or subtract. Such a subroutine could be used to logic-OR all of the bytes in the number together using the accumulator. If the result of the OR'ing is zero, then each byte in the number must have been zero.

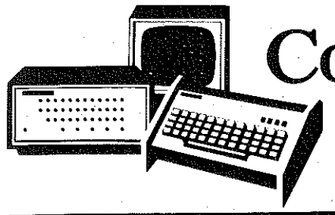
One way to do a signed comparison between two signed numbers is to subtract them and then see if the result is negative (right operand is larger), zero, or positive (left operand is bigger). Besides destroying one of the numbers being compared, this method suffers from a subtle pitfall. If the left operand is a large positive number and the right operand is a large negative number, the subtraction can overflow and the comparison result will be invalid. The converse case, right operand positive and left operand negative, creates the same problem. There is never any possibility of overflow when the numbers are of like sign however.

A dedicated comparison subroutine overcomes both problems. Unlike previous routines where the operation started at the right, comparison should start with the most significant bytes. The first step is to look at the sign bits. If they differ, the comparison result is clear already and any overflow problems are avoided. If the sign bits are the same, then the most significant bytes are subtracted but the result is not stored. If the result of the subtraction is nonzero, then the outcome of the comparison is known (negative means right operand larger, positive means left operand larger) and a return can be taken. If the result is zero, then lesser significant bytes must be subtracted until either a nonzero result is obtained or the entire number has been processed. In the latter case, equality between the two is the conclusion.

Rotation. Multiple precision rotation is much like addition in that the C-flag is used to transfer bits from one byte to the next. Although a more comprehensive set is easily written, subroutines to rotate left and right including C are usually sufficient. The rotate is effectively changed to a shift if the calling program clears C before calling multiple rotate. Rotates by several bit positions are accomplished by repeated calls to a single bit position rotate routine.

For a multiple rotate left, start at the rightmost byte. The byte is rotated left with carry by use of the ROL instruction which puts the old C-flag into bit-zero, shifts the whole byte left by one, and puts bit-7 into the C-flag. Following this, the next byte to the left is rotated in a similar manner. The C-flag serves to transfer bit-7 of the low byte into bit-zero of the next higher byte. This process is repeated until the entire number has been done. Note that in the 6502, all of the manipulation can be performed directly in the pseudo register.

Rotate right is the exact opposite of rotate left. Start with the leftmost byte and use the ROR instruction on each lower byte in sequence. Unfortunately, some early production 6502's were manufactured without the ROR instruction. For these, an ROR can be simulated by doing 8 ROL's in a row instead. ◇



Computer Bits

By Hal Chamberlin

MICROCOMPUTER MEMORY

ALMOST from the start, memory has played a major, if not dominant, role in the practicality and cost of a hobbyist computer system. In the very early days (before 1972), the small "home brew" group of computer hobbyists used any type of memory that was available and relatively low in cost. Telephone relays, magnetic-tape loops, delay lines, salvaged memory drums, and of course core memory stacks were used. However, all of these devices were either extremely slow in speed, or were complex.

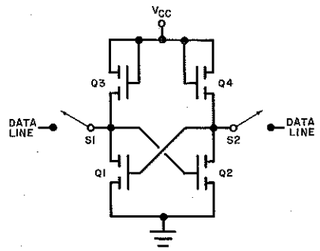
Along about 1972, semiconductor memories became low enough in price to be used by someone brave enough to tackle building a home computer. These memory devices were as easy to use as the rest of the logic within the system, thus adding to their appeal.

So, when the home computer revolution began in 1975 with the introduction of a low-cost microcomputer kit, semiconductor memories were right there along with the microprocessor chips. Even now, the main memory (as distinguished from external mass storage such as a cassette), is the dominant cost and performance factor in a home computer. The speed and sophistication of the MPU mean nothing if the main memory does not have ample capacity. Although some MPU's have more efficient storage of programs, when it comes to raw data storage, all systems are equal.

Types of Memory. Classical memory devices are divided into two distinct groups: random access and nonrandom access. A random access memory requires essentially the same amount of time to read or write a particular memory cell regardless of which cell is addressed, or the order of consecutive addresses. It literally means that a *random* sequence of addresses is handled just as fast as an ordered sequence.

The very earliest memory IC's were long shift registers that were serial access rather than random-access devices. When presented with a random address, a shift-register memory requires a variable access time depending on where the data is within the register. Today's systems use random-access memory exclusively for main memory.

Random-access semiconductor memories can be further broken down into *read-only, read-mostly*, and *read/write* classes. A read-only memory (ROM) can only be read. The information in the memory is placed there during manufacture and can never be changed. Read-only memories are typically used for unchanging system programs such as a monitor or BASIC interpreter. The advantage of permanent memory is that loss of operating power does not destroy the memory contents.



Six-transistor static memory.

The read-mostly kind of memory IC normally behaves just like a read-only memory but it is possible, using specialized equipment and a procedure called programming, to change the memory contents. Such memory devices are called Programmable Read-Only Memories or PROM's and the equipment is called a PROM programmer. One type of PROM is manufactured with all memory cells containing "0's". The programming procedure can change selected cells to "1's" to get the desired memory contents. The PROM can be programmed again later to write additional "1's" but once set, a cell can never return to a "0".

Another type of read-mostly memory IC can be *erased* to its all-"0" state and then completely reprogrammed as often as desired. These are called EPROM's for Erasable PROM. The erasure is usually accomplished with intense short-wave ultraviolet light, although a couple of types exist that can be erased with voltage pulses. The EPROM costs more than any other type of memory IC.

A read/write memory, which is usually called just a RAM, can be written into as quickly and easily as it can be read from. Most of the memory in a typical system is of this type because such a memory does not have to be dedicated to any single program or data table as ROM and PROM are. User programs and data are always stored in RAM and frequently many of the system programs such as the assembler and text editor are also stored there. Of course, the very flexibility, and ease of writing, makes RAM contents easily destroyed by errant programs or operating power failures.

Inside RAM. Since plain RAM is the most popular kind of memory, let's take a closer look at RAM operation and terminology. Two basic storage circuits are used in modern RAM's. The first type is a conventional flip-flop (as shown in the diagram) made from MOS transistors *Q1* and *Q2*. Transistors *Q3* and *Q4* function as high-value load resistors and are used because they are physically smaller than an equivalent resistor would be. Switches *S1* and *S2* connect the memory cell to the outside world and provide the read and write data path.

When the cell is unaddressed, both switches are open and the cell is isolated. To read, both switches are closed and the state of the flip-flop can be determined by sensing the voltage level on the data lines. To write, the switches remain closed and other circuitry forces the data lines to voltage levels that will cause the flip-flop to change state.

This type of cell is called *static* because once the flip-flop is set to a particular state, it will remain in that state until instructed to change, or the power supply voltage drops. Switches *S1* and *S2*, are in reality, MOS transistors and so the memory cell in Fig. 1 is a 6-transistor static memory cell.

Another common data storage circuit is just a capacitor and a switch which again is really a transistor. When the cell is unaddressed, the switch is open and the voltage level on the capacitor determines the cell's state. To read the cell, the switch (transistor) is closed thus discharging the capacitor into a sensing circuit connected to the data line. If a surge of current from the discharging capacitor is sensed, then a 1 was stored—no surge represents a 0. The data is then restored to the cell by applying a high voltage to the data line if a 1 had been previously sensed. When writing new data, the initially sensed data is ignored.

This cell is called *dynamic* because the charge will leak away from the capacitors if they are not written or read often enough. At room temperature, the charge remains for a second or so; but at the top end of the rated temperature range, the period may be only a few milliseconds. Actually, in a dynamic memory IC, the capacitor is really just stray capacitance. Thus the entire memory cell consists of just one transistor.

The small size of such memory cells allows as many as 16,384 of them to be placed on one chip, whereas only 4096 of the 6-transistor type are diffused on one chip. Even the lowest cost and most popular dynamic RAM's pack 4096 bits in a chip, whereas static types contain only 1024 cells.

Another advantage of the dynamic cell is that power consumption is very low. Cells just idling don't consume any power. The dynamic RAM consumes power only when being accessed while a static RAM constantly draws current to keep the thousands of internal flip-flops powered. It is not unusual to see a 32k static memory system require over 8 amperes while an equivalent dynamic memory system might require less than one ampere.

As previously mentioned, a dynamic memory system must read or write every cell occasionally to recharge the storage capacitors. Since this does not always happen during normal operation of the system, a separate *refresh* operation is usually performed. Refreshing is quite simple and amounts to nothing more than sequentially reading through a portion of memory using a counter to generate addresses. Due to the internal organization of the memory IC's, only 64 addresses really have to be read to refresh all the cells.

Early memory board designs using dynamic RAM's would periodically stop the MPU while refresh was being performed. Modern designs look at the state of the MPU and during those times when memory access is not required, a refresh cycle is slipped in.

Memory Boards. There are certainly more different kinds of memory boards for hobbyist systems on the market than any other type of board. For Altair (S-100) bus systems, the earliest memory boards contained only 1024 bytes of static RAM. Later, MITS introduced its 4k dynamic memory board using 22-pin 4k RAM's. It was quite power conservative but priced fairly high. Refreshing was done by halting the MPU periodically. This opened the door for competing brands of memory boards which, in the interests of quick development and marketing strategy, used 1k-bit static RAM IC's and had a total capacity of 4k bytes. While they worked well and were low in cost due to intense competition, the much greater power consumption of static RAM's strained the computer's power supply and cooling system. Computer manufacturers responded with massive power supplies to satisfy customers who plugged 32k and more of memory into a system.

Later improvements in printed circuit board fabrication allowed as much as 8k on a board using the same 1k memory components. Finally, the IC manufacturers figured out how to put 4096 bits of static memory onto one chip and also cut the power consumption per bit by a factor of two to four. Thus, 16k static memory boards became available. Although more expensive per bit than the 4k or 8k boards, these larger units are becoming popular.

Finally, memory board designers have found ways to "hide" the refresh cycles of a dynamic memory in the MPU's idle time periods. Also, at the same time that 4k static-memory IC's became available, 16k dynamic RAM's also became available. Using these, it is now possible for 64k bytes to be put on a single board. The 16k IC's are still quite expensive, but an 8k version is being used in very cost-effective memory boards with a capacity of 32k bytes.

However the least expensive memory boards on today's market still use the same 22-pin 4k RAM's that were used in the original MITS 4k board. These have a 16k capacity, utilize hidden refresh, and are $300 to $400.

Memory costs are constantly decreasing. In two years it will be possible to buy a 64k memory board for what a 16k board costs now. Already the computer manufacturers are introducing systems that can address more than 64k of memory to make room for the never-ending memory capacity spiral. ◇