Programming the ZX Spectrum

Alvin Albrecht

So you want to write Spectrum games. You have plenty of company -- retroprogramming, like retrogaming, is taking off, with new software for antique computers arriving every month. It's almost like it's 1984 all over again. Well, maybe 1990. Why this trend is happening, I don't know. Perhaps it's 30-somethings wanting to fulfill that fantasy of being the 16-year-old prodigy with the Ferrari. Perhaps it's part of the nostalgia kick, remembering that feeling that you were on the cutting edge, that new computing discoveries were accessible through that little black box, that new discoveries were made and published in magazines each month.

Whatever the reason, you and I both know that to get anything good out of the little black box will require more than knowing Basic -- it will require a leap into machine code and understanding how the hardware actually works. There are numerous tutorials on the web that can help to make that leap, but they are brief and only scratch the surface. Most of the learning is still up to the individual and this can be an intimidating step requiring much persistence.

This series of articles is intended to fill the void and is aimed at anyone, with or without programming experience, interested in programming the Spectrum and any of its clones but don't know where to start. So we will start from the beginning and end at the end, from the basics to the advanced. Details will be discussed, not glossed over. Examples will be plentiful and there will be much material on the subject that hasn't yet been articulated on the internet. Perhaps we will be lucky enough to have some prolific members of the Spectrum community share some of their wisdom as guest writers.

This first article will provide some background on the brains of the Spectrum -- the z80 cpu -- and some discussion on number systems, a necessary topic that is best covered early on. Once these basics are covered you will be well equipped to dive into the more interesting subject of making your Spectrum jump hoops, in part two of this series. Yes, the basics can be a bit of a slog, but I promise you it will be worth the effort in the end.

1.0 INTRODUCING THE Z80 MICROPROCESSOR

The z80 was introduced in 1976 by Zilog, a company formed by a group of engineers who left Intel after working on the 8080. The z80 was designed to be software compatible with the 8080 but also incorporated many new features. Some of the most important features had to do with the hardware: it was designed to use a single power supply (unlike the original 8080 which required two), it had built-in refresh logic to make interfacing to cheap memory (ie DRAMs) simple, and it had logic to allow a complete microcomputer system to be built with a minimal number of parts.

The z80 was a more powerful replacement of the 8080, featuring twice as many registers and many more instructions. The 8080 already had the lion's share of the microprocessor market so this pretty much guaranteed the z80's success. The ease with which a z80 system could be built coupled with nearly 100% binary compatibility* with the 8080 made it into the dominant 8-bit microprocessor of the 8-bit era, selling more than half a billion** units [1, p7], second only to sales of the less sophisticated but omnipresent 8051.

The z80 has managed to endure even until today in embedded systems. You can still buy new 20 MHz z80s and there have been and still are several 16 and 32-bit microprocessors developed by Zilog based on the z80 (the z800, z8000, z80000, z180, z280, z380, ez80). Rabbit Semiconductor has even entered the business of selling z80 knock-offs. The z80 just doesn't want to die and doesn't look like it will any time soon.

* Although the z80 can run all 8080 programs, the timing of individual instructions is not always identical meaning the few programs that relied on precise timing would not work correctly without modification.

** European readers: 1 billion = 1 thousand million

1.1 THE PHYSICAL

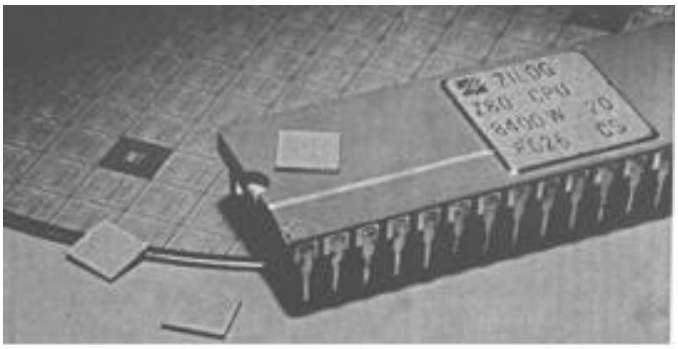


Figure 1. The Z80 chip from wafer to DIP package
(c) 1981 Zilog [4, p6]

The original z80 came in a 40-pin dual-inline package (the DIP), colloquially referred to as a "chip". The z80 circuit (the chip) is actually inside the package, a small square die about the size of your pinky fingernail, and consists of layers of doped silicon (silicon that has been impregnated with ions to change its conductivity), silicon dioxide, and metal arranged to make a working digital circuit. Pads large enough to be visible to the naked eye line the perimeter of the square die. Tiny wires are bound to each pad and run to one of the pins of the package. The whole thing is covered up by the package lid.

Figure 1 is a picture of the z80 chip in a 40-pin DIP, with a small square die (the z80 circuit) resting on the top of the package. The circular array of dies shown is called a "wafer". In the manufacturing process, many dies are made at once on a single silicon wafer which is then chipped into individual pieces. The dark lines that can just about be made out on each die are runs of long metal wires near the surface.

Figure 2 shows a cross sectional view of one die, zoomed in to illustrate a single transistor. The bottom layer is the "substrate" and is the silicon wafer the z80 circuit is built on. The "n+" regions are areas where ions have been shot into the silicon substrate to change the silicon conductivity in that area. The silicon dioxide (SiO2) is grown on the substrate and is there to help electrically isolate the transistor from its neighbours. Aluminum is deposited on the silicon surface to form wires. The original z80's gates would not have been made from polysilicon as shown in the diagram; they would have been metal as well. The diagram being used is a little too modern for our purposes. The original z80 had about 8500 such transistors on each die [2].

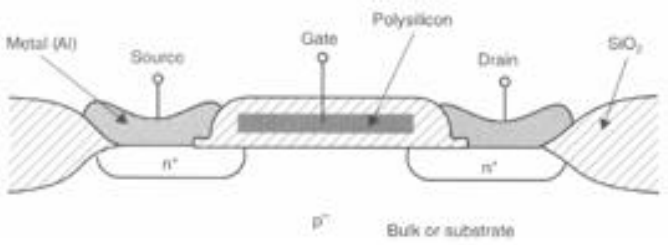


Figure 2. Cross sectional view of N-channel MOS transistor [5, p17]

A diagram showing the purpose of each of the z80's pins can be found in figure 3. Figure 3 shows the pins collected into functional groups and doesn't actually show the physical position of each pin -- rather they are numbered so that the pins can be found on the package easily. These pins are the channels through which the z80 circuit communicates with the outside world.

When dealing with digital circuits, a useful abstraction is to imagine that all signals take on one of two values, called "1"/"high" and "0"/"low". In pins and wires, 1s are represented by a high voltage and 0s are represented by a low voltage. What high voltage and low voltage actually mean depends on the logic family being used ("logic family" meaning "type of electronic circuit used to implement the digital logic"). There are many logic families, some of which you may have heard of, like TTL, CMOS two-phase, NMOS, ECL, etc. Older digital circuits most commonly used 5-volt TTL logic. In this family, anything from 0-0.8 volts is considered "low" or "0" and anything from 2-5 volts is considered "high" or "1". A steady signal in the no-man's land from 0.8 to 2 volts represents a corrupted signal that hasn't decided whether it's a "1" or a "0" and shouldn't happen in a functioning circuit. Naturally when a signal changes from a "1" to a "0", the voltage must travel continuously from a high to a low value, temporarily passing through this no-man's land. The finite time necessary to traverse from a "1" to a "0" and vice versa is what determines how fast a digital circuit can operate.

TTL circuits were implemented using bipolar junction transistors (BJTs) and the voltage ranges are a result of the characteristics of digital circuits made with BJTs. As technology moved along, metal-oxide-semiconductor transistors (MOS) became preferred for implementing digital circuits (figure 2 is a cross-sectional view of an N-channel MOS transistor). However TTL was so entrenched as a means to communicate 1s and 0s between chips that it was common for chips to be implemented using CMOS/NMOS circuits and yet have TTL drivers for signals entering and leaving the package. This was the state of affairs the z80 was born into. The original z80 was an NMOS circuit (made with N-channel MOS transistors) using TTL voltages.

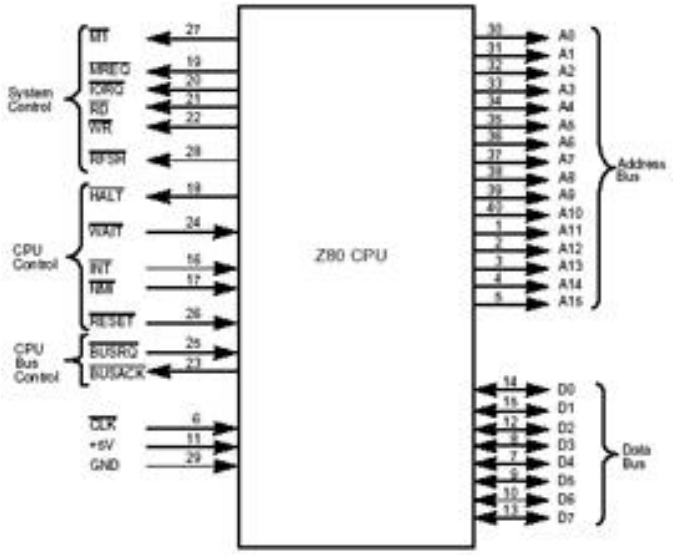


Figure 3. I/O Pin Configuration [6, p7]

1.2 THE Z80 AS TASKMASTER

The z80 controls the entire computer system. It does this by communicating its demands through its pins. There are six kinds of communication, each of them called a "machine cycle". You can see timing diagrams for all of them in figure 4. They are:

A. Instruction Fetch Cycle (or M1 cycle)
Performed whenever the z80 wants to read the next instruction from memory.

B. Memory Read/Write Cycle
Performed whenever the z80 wants to read or write data to memory.

C. Input/Output Cycle
Performed whenever the z80 wants to read or write to peripherals. The z80 makes a distinction between

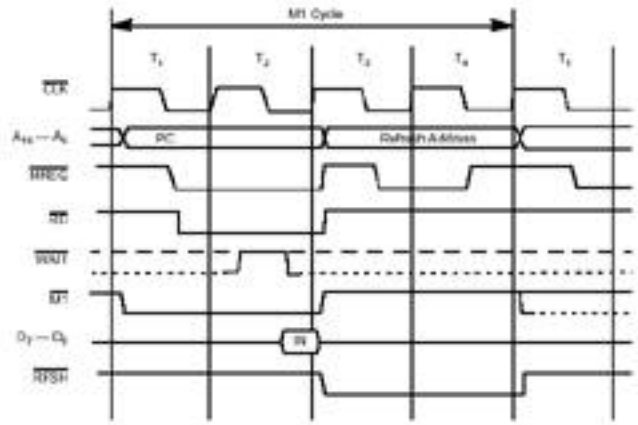


Figure 4a. Instruction Fetch Cycle

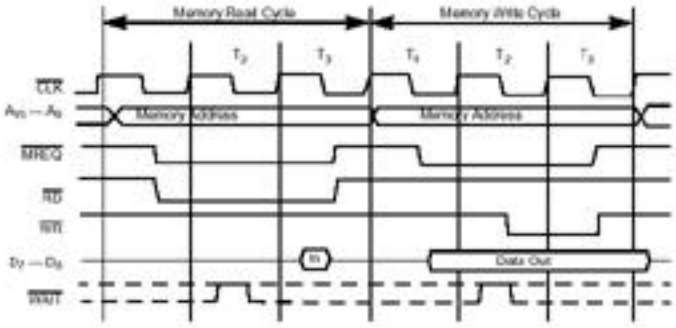


Figure 4b. Memory Read / Write Cycles

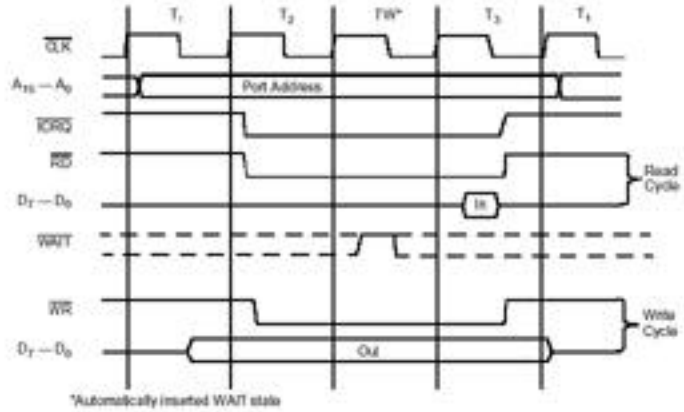


Figure 4c. Input / Output Cycle

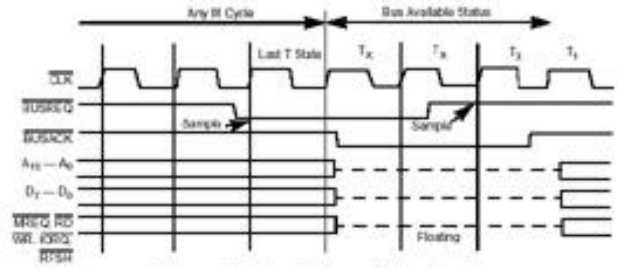


Figure 4d. Bus Acknowledge Cycle

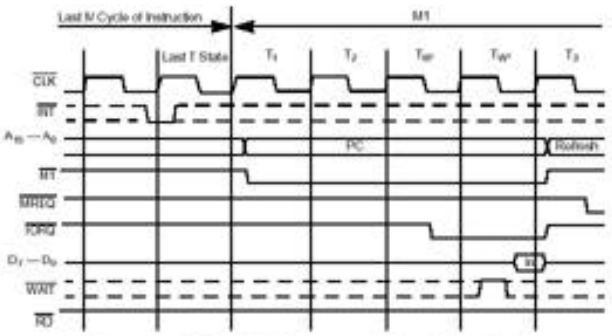


Figure 4e. Maskable Interrupt Acknowledge Cycle

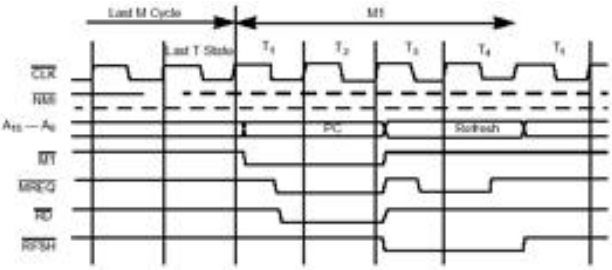


Figure 4f. NMI Acknowledge Cycle

Figure 4. All Six Z80 Machine Cycles [6, p12-18]

memory and peripherals, whereas many other processors do not. This will be explained later on in this series.

D. Bus Request/Acknowledge Cycle
An external circuit can request control of the system by placing a 0 on the "/BUSRQ" pin. If the z80 grants the request, it performs this machine cycle, goes to sleep, and allows the external circuit to control the system. These kinds of circuits rarely existed in inexpensive consumer electronics. An example is the MB02 disk interface for the ZX Spectrum which includes a Z80-DMA chip that takes over the system to perform fast memory copies.

E. Interrupt Request Cycle
An external peripheral can ask for attention from the z80 by placing a 0 on the "/INT" pin. The z80 responds by performing the int request cycle if interrupts are not being ignored.

F. NMI Request Cycle
An external peripheral can demand attention from the z80 by placing a 0 on the "/NMI" pin. The z80 responds by performing the nmi request cycle. An NMI is a second type of interrupt that cannot be ignored by the z80.

To gain a better understanding of what these timing diagrams show, have a look at the memory read cycle in figure 4b. The figure actually shows a complete memory read cycle immediately followed by a complete memory write cycle in order to save some space. We're interested in the first half -- the memory read cycle -- which shows what the z80 does when it wants to read memory.

The first thing to know is that the z80's "CLK" pin receives a clock signal, generated by an external oscillator circuit. The clock signal is a square wave with alternating 0s and 1s -- a diagram of a typical clock signal is shown in figure 5. The length of time of one period of the clock signal is called a "T-state" or a "cycle." The z80 uses the clock signal as a timer. When it needs to do something, like read data from memory, it does it in several steps with each step performed in succession. It is time to move on to the next step when the appropriate amount of time (or T-states) have passed. The faster the clock signal (the shorter the length of the T-state), the faster the z80 performs its operations. The slower the clock signal (the longer the length of the T-state), the slower the z80 performs its operations. So all that needs to be done to speed up the z80 is to attach a faster clock. This is correct, however the clock cannot be made fast without limits. It takes time for logic within the z80 to settle to solid 0s and 1s so this places an upper bound on how fast the clock can be. One also has to consider the speed of external circuits, like memory, as will be seen in a moment.

In the 48K Spectrum, the clock signal is derived from the 14MHz video oscillator used to generate the TV picture. The z80 clock is driven at 14MHz/4 = 3.5MHz making each T-state 1/3.5MHz = 285.7ns (285.7 nanoseconds = 285.7 x 10^-9 seconds) in length. The TS2068 uses a 14.11MHz video oscillator and drives the z80 clock at 14.11MHz/4 = 3.5275MHz making each T-state 283.5ns. The 128K Spectrums are slightly different as well.

Everything the z80 does is synchronized to the clock, which is why it appears prominently at the top of all the timing diagrams. Let's return to the memory read diagram which shows what the z80 does when it wants to read memory. From the left we see there is a rising edge on the clock, marking the beginning of T1 - the first T state of the machine cycle. Shortly afterward the z80 places a memory address on the address pins A15-A0. Then there is a falling edge on the clock. The z80 puts 0s on the /MREQ and /RD pins. External memory will see this and know that it is expected to respond. A rising edge on the clock occurs to finish the first T-state and start the second (called T2). Nothing much happens in the second T-state. The external memory circuitry might place a 0 on the /WAIT pin to indicate to the z80 that it needs more time. If it does, then another T-state identical to the second will be inserted by the z80 to give the external memory more time to respond. This is called "inserting a wait state" and can happen if the z80 cpu is clocked faster than external memory can handle. This does not occur in the ZX Spectrum. The timing diagram shows that the z80 samples the /WAIT pin at the falling edge of the clock in the second T-state. In the diagram, the /WAIT pin was found to be a 1, so the z80 proceeds to the third T-state. The external memory is expected to have placed a single byte of 0s and 1s on the data bus (pins D7-D0) corresponding to the memory address supplied by the z80. The z80 reads in that data at the falling edge of the clock in the third T-state. The z80 then releases the /MREQ and /RD pins to indicate the read memory cycle is finished and the external memory should let go of the data bus pins D7-D0.

So to read a single byte from memory, the z80 executes the memory-read machine cycle, which takes 3 T-states of time in total as long as no wait-states are inserted. On a 48K Spectrum, those 3 T-states amount to 3*285.7ns = 857.1ns of time.

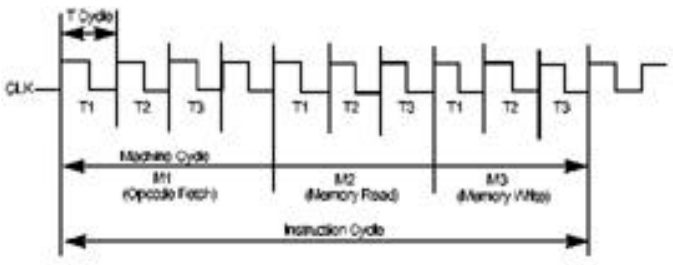


Figure 5. Typical Clock with Machine Cycles Superimposed [6, p12]

1.3 AN ASIDE ON RAM MEMORY

The z80 memory-read machine cycle was actually designed to match the requirements of existing external memories. In fact, the read cycle timing diagram just examined looks a lot like the timing diagrams found in static RAM data books. Because of this, it is possible to attach static RAM memory to the z80 without any extra logic -- remember the earlier claim that the z80 was easy to use? You just need to make sure that the external RAM has enough time to respond to the z80's requests, which places another upper bound on the z80's clock speed.

Static RAMs (SRAMs) use six transistors to remember a single bit of memory (either a 0 or a 1). It is called static (as in "not changing") because once a 1 or 0 is stored in the memory cell, it will be remembered as long as power is maintained to the RAM chip.

Dynamic RAMs (DRAMs) use a single transistor and a small capacitor for each memory bit. A capacitor is an electrical device that can hold a charge. When a 1 is written to the bit, the capacitor is charged high (in early days, to 5V). When a 0 is written, the capacitor is drained to 0V. The trouble with capacitors is that they are not perfect. Once a 1 is written, it begins to drain charge so that the voltage falls from 5V toward 0V. If nothing is done, the capacitor will discharge completely and there won't be a 1 stored there any longer. The memory is short-term, lasting on the order of tens of milliseconds. In order for DRAMs to be useful they need to be periodically refreshed. This means that every single bit must be read before those tens of milliseconds are up and rewritten with the full 5V wherever a 1 is stored in order to "top up" the charge on the capacitors.

This seems like a lot of work but there is a payoff. Because DRAMs only use one transistor per bit, DRAM chips can store six times as much memory as the same size SRAM chip, making them at least six times cheaper. To help with the necessary refresh operation, DRAM chip packages are also made smaller by eliminating half the address pins.

DRAM chips used with a z80 system, which has 16 address pins, might have 8 address pins on their package. The full size 16-bit address A15-A0 of the z80 is split in half, with A15-A8 called the "row" and A7-A0 called the "column". When the z80 performs a memory access, presenting the full address on A15-A0, extra external logic must split that into a row and column part, present the row part on the 8 pins of the DRAM chip, activate the DRAM chip's /RAS pin (row address strobe), then present the column part on the 8 pins of the DRAM chip, and finally activate the DRAM chip's /CAS pin (column address strobe) before data comes out of the DRAM chip. Sound complicated? You bet. That is one of the reasons why SRAMs are popular.

So how does this help with refresh? DRAM chips contain internal logic that allows entire rows to be refreshed at a time. The hypothetical DRAM chip being discussed here has a row size of 8 bits, meaning 256 refreshes need to be performed every 10+ milliseconds or so. A refresh is performed by placing a row address on the DRAM's 8 address pins and activating the chip's /RAS signal. In order to make sure all memory cells are refreshed, rows 0,1,2,3,..,255 should be refreshed in a complete sequence every 10+ milliseconds.

With other cpus of the day, extra logic had to be added to count the refresh rows (0..255) and pause the cpu each 10+ms/256 = 0.04+ milliseconds to perform a single refresh operation. The z80 came with built-in refresh logic. It contains a refresh register that automatically keeps track of the next row needing refresh and adds a refresh cycle at the end of the instruction fetch machine cycle (see figure 4a) so that memory can be refreshed without pausing the cpu. This is another reason the z80 was so easy to use.

1.4 HARDWARE - A FINAL NOTE

This series won't be dwelling too much on hardware issues. This is a series aimed at programmers, not engineers. We will come back to timing diagrams on occasion, but the main points you should take from the preceding are the following:

* The z80 circuit controls the entire computer system by communicating its demands using six basic machine cycles.

* Each machine cycle is a sequence of signals (0s and 1s) impressed on the z80's pins and timed by the clock.

* The basic measurement of time is the T-state, which is the length of the clock signal's period in seconds. Machine cycles take a number of T-states to complete.

* External devices (or interface circuits) like memory, joysticks, keyboards, sound ICs, etc. monitor the z80 pins to know when it is time to interact with the cpu.

* The program, which is a sequence of instructions read from memory by the z80, determines what machine cycles the z80 executes and in what order.

2.0 NUMBER SYSTEMS: BINARY, HEXADECIMAL, OCTAL AND DECIMAL

It is inevitable that any introductory tutorial on microprocessors eventually turns to a discussion on number systems. There is good reason for that. It should be clear now that all digital microprocessors, the z80 included, function by manipulating 0s and 1s. 0s and 1s by themselves mean nothing until humans attribute meaning to them and the most fundamental meaning is numbers.

Let's begin with some definitions:

* BIT
A single 0 or 1 is called a bit, short for "binary digit".

* NIBBLE
A collection of four bits is called a nibble. The leftmost bit in a nibble is "most significant" and the rightmost bit is "least significant". The meaning of "significance" will become clear later.

* BYTE
A collection of eight bits is called a byte. A byte consists of two nibbles. The leftmost nibble is the "most significant" nibble and the rightmost nibble is the "least significant". In data communication circles the term "octet" is sometimes preferred when referring to 8 bits that are part of a larger group of bits.

* WORD
Here things become a little murky. A word, in the world of 8-bit cpus, is a collection of 16 bits. In the modern computing world, a word is a collection of 32 bits. In actuality, a word is the same size as the address bus of the cpu in question. Since this tutorial is about the z80 and the z80 has 16 address pins, a word is henceforth defined as a collection of 16 bits. Such a word contains two bytes and four nibbles. Following the same pattern, the leftmost items in a word are "most significant" and the rightmost are "least significant".

Here is an example of a word:

```
0100 0010 1101 1001
```

By convention long binary strings are broken up by spaces to make them more readable.

This word contains the four nibbles "0100", "0010", "1101" and "1001". Of these, "0100" is the most significant nibble and "1001" is the least significant nibble. It can be said that "0010" is more significant than "1101".

This word also contains two bytes: "0100 0010" and "1101 1001". "0100 0010" is the most significant byte whereas "1101 1001" is the least significant.

The most significant bit in the word is the leftmost "0". The least significant bit is the rightmost "1". The most significant three bits are "010" and the least significant five bits are "1 1001".

Summary:

BIT = a single 1 or 0
NIBBLE = four bits
BYTE = eight bits
WORD = sixteen bits
MOST SIGNIFICANT = leftmost items
LEAST SIGNIFICANT = rightmost items

2.1 THE DECIMAL NUMBER SYSTEM

The decimal number system is taken for granted in the modern world, but what we have today is actually the product of thousands of years of human development. I am neither a historian nor am I particularly interested in doing research on the subject so I won't bother detailing the history here. I leave that as an exercise to the reader. Suffice it to say that the major contributors to the modern decimal system were the Romans, Indians, and Arabs.

What we have today is a positional weighted system of representing quantities based on the ten symbols 0-9. It is positional because the order of symbols makes a difference in the total value. "456" is not the same as "654". It is weighted because symbols have different values in different positions. The "2" in "200" has a different weight than the "2" in "20".

The essential innovation was this. The first nine objects are counted with the symbols 1-9. There is no symbol to represent ten objects and, rather than inventing a new one for the purpose, ten objects is denoted by "10" with the "1" in the second position meaning one group of ten and the "0" in the first position meaning zero singles. Ten to nineteen objects can be represented with "10" through "19", then we have "20" representing two groups of ten and no singles. This is repeated until "99" is reached. There is no way to represent one hundred objects without introducing a third position so "100" is arrived at, meaning one group of one hundred, zero groups of ten and no singles.

There is a mathematical pattern to all of this. Given the string of decimal digits dn...d1d0 the numerical value is dn*10^n+...+d1*10^1+d0*10^0.

An example:

```
12345 = 1*10^4  + 2*10^3 + 3*10^2 + 4*10^1 + 5*10^0
      = 1*10000 + 2*1000 + 3*100  + 4*10   + 5*1
```

You will note that the leftmost digit has the largest weight and contributes the largest chunk toward the quantity represented. One could say it is "most significant". The

rightmost digit has the smallest weight and contributes just 5 to the total quantity. One could say it is "least significant".

2.2 BINARY NUMBERS

The decimal system is appropriately named as it is based on the ten symbols 0-9. "Deci" means ten. The binary system is based on two symbols, hence the "bi" in its name.

In computers, those two symbols are "1" and "0", represented physically in a circuit by high and low voltages. In the same way that large quantities were represented by strings of decimal digits in the decimal system, binary digits can be strung together to represent large quantities as well. Because there are only two symbols, each digit position has a weight that is a power of 2.

Given the binary string bn,...,b1b0 the quantity represented is bn*2^n+...+b1*2^1+b1*2^0.

An example:

```
1001 1011 = 1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 +
            1*2^3 + 0*2^2 + 1*2^1 + 1*2^0
          = 1*128 + 0*64  + 0*32  + 1*16  +
            1*8   + 0*4   + 1*2   + 1*1
          = 128 + 16 + 8 + 2 + 1 = 155
```

Specific bits in a binary number can be referred to by their "bit position". The bit position is the particular power of 2 that acts as the weight for the binary digit. So bit 0 is the rightmost least significant bit, having weight 2^0. Bit 1 is the second bit from the right with weight 2^1, and so on. In this example, bit 6 = 0, bit 3 = 1, and bit 5 = 0 and have weights 2^6, 2^3 and 2^5 respectively.

Because binary numbers have the same mathematical foundation as decimal numbers, you can apply the same operations to them such as addition, subtraction, multiplication and division using the same algorithms you learned in elementary school for decimal numbers.

Try adding two binary numbers together:

```
   ,,,
  10011
+ 00101
  -----
  11000
```

Starting from the right 1+1 = 2 right? Not quite. The binary system has just two symbols and can only represent 0 and 1, so what we get is "10" (1 group of two plus 0 singles) or 0, carry the 1. The carry is shown by the "," in the problem. Proceeding in this manner, the sum turns out to be "11000". Let's check this result.

The first number "10011" is (starting with the least significant bit) 1*2^0+1*2^1+1*2^4 = 19. The second number "00101" is 1*2^0+1*2^2 = 5. The answer is "11000" = 1*2^3+1*2^4 = 24. 19+5=24, correct!

Subtraction can also be done the old fashioned way. Instead of borrowing ten from the position to the left, one borrows two in binary. In binary there is a better way to perform subtraction by using addition so we won't dwell on subtraction here.

Here is an example of a multiplication:

```
    1101    (13)
x    101    ( 5)
    ----
    1101
   0000
+ 1101
 -------
 1000001    (65)
```

Binary multiplication is particularly easy because in each step you are multiplying the multiplicand ("1101" in this case) by either 0 or 1 so the partial result is either zero or the multiplicand itself.

The multiplication algorithm depends on a peculiar property of weighted positional number systems, and that is that multiplication by the base of the number system is the same as "left shifting" the digits or, equivalently, adding a "0" to the end of the number.

In decimal the base is 10 because there are ten symbols. Multiplying a decimal number by 10 can be accomplished by adding a "0" to the end of the number. Eg: 45*10 = 450. The same thing happens in binary. Multiplying a binary number by 2 can be accomplished by left shifting the binary number or, in other words, by appending a "0" to the end. Eg: 101 (five) x two = 1010 (ten).

Why does this work? Recall that any binary number can be expressed as: bn*2^n+...+b1*2^1+b0*2^0. Multiply this by two and you get: bn*2^(n+1)+...+b1*2^2+b0*2^1+0*2^0. Ie: the binary digits have been shifted left with 0 placed in the least significant bit.

Since a binary number like "101" can be expressed as (2^2 + 2^0), multiplication by "101" is equivalent to adding 2^0=1 times the number plus 2^2=2*2 times the number, or adding a copy of the number plus a copy of the number shifted left two positions. Have another look at the multiplication example to see this in action. The old elementary school multiplication algorithm is based on this insight.

Division works along the same lines, but we'll save this for an example program later on in the series. Division

by two in binary is the same as shifting the binary number to the right. So 110 (2+4=6) divided by two = 11 (1+2=3).

Dividing by two is the same as multiplying by 2^(-1). If we begin with a generic binary number bn*2^n+...+b1*2^1+b0*2^0 and multiply it by 2^(-1) we get bn*2^(n-1)+...+b1*2^0+b0*2^(-1). You will note that this is precisely the binary number shifted right one position UNLESS the least significant digit (b0) is not a 0. Then we get a peculiar 1*2^(-1)=0.5 added to the result. We have stumbled across a binary fraction!

The same thing happens with decimal numbers. 450/10 = 45, no problem, but 452/10 = 45.2 with the "2" representing 2*10^(-1). The decimal point notation separates a left whole number part of the decimal number from a right fractional part of the decimal number.

In binary, the binary point serves the same purpose, so we get results like 1011 (eleven) / two = 101.1 (1*2^2+1*2^0+1*2^(-1)=5.5). Just as with decimal numbers where the fractional positions represent one tenth (10^(-1)), one hundredth (10^(-2)), one thousandth (10^(-3)) etc., binary numbers have fractional positions representing one half (2^(-1)), one quarter (2^(-2)), one eighth (2^(-3)), and so on.

In general, a binary number has both whole and fractional parts and can be expressed as: bn*2^n+...+b1*2^1+b0*2^0+ f1*2^(-1)+f2*2^(-2)+...+fm*2^(-m).

An example:

```
1010.011 = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 +
           0*2^(-1) + 1*2^(-2) + 1*2^(-3)
         = 1*8 + 1*2 + 1*0.25 + 1*0.125
         = 10.375
```

Multiplying and dividing binary numbers by two involves shifting the string of binary digits left or right with respect to the binary point.

One frequently needs to know what the largest number that can be represented by N binary digits is.

The maximum four-bit binary number is 1111, which is 15. So using four bits, one can represent the numbers 0 through 15. The maximum five-bit binary number is 11111, which is 31.

It is tedious but not impossible to answer the question this way. There is a better way, though. Note that if one is added to the largest four-bit binary string, we get:

1 + 1111 = 10000 = 2^4

Rearranging the equation somewhat, the largest number representable with four bits is:

2^4 - 1 = 1111 (binary) = 15

This result generalizes. The largest number representable with N binary digits is "2^N - 1". With eight bits, the largest number is 2^8-1=255 and with 16 bits, it's 2^16-1 = 65535.

This formula can also come in handy to find out how many bits are required to represent a specific quantity.

Q:
How many bits are needed to represent the number 7360?

A:
Solve "2^N-1=7360" for N, N = 12.8 or at least 13 bits.

2.3 OCTAL & HEXADECIMAL

The base 2 number system is called "binary" and the base 10 number system is called "decimal". These are so called because they are based on two and ten unique symbols respectively. "Hexadecimal" and "octal", to a lesser extent, are also in common use in connection with computing. Hexadecimal is base 16 and uses 16 unique symbols: the digits 0-9 followed by the letters A-F. Octal is base 8 and uses the digits 0-7 as its eight symbols.

Here is a brief table showing the numbers 0-15 represented in various bases:

Decimal	Binary	Hexadecimal	Octal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Hexadecimal and octal are in use because they can easily represent a binary bit pattern compactly. This is because they have bases that are exact powers of two. You will see why this matters in a moment.

Any hexadecimal number hn...h1h0 represents the value hn*16^n+..+h1*16^1+h0*16^1.

An example:

```
DFE7 = 13*16^3 + 15*16^2 + 14*16^1 + 7*16^0
     = 13*4096 + 15*256  + 14*16   + 7*1
     = 57319
```

Likewise any octal number cn..c1c0 represents the value cn*8^n+..+c1*8^1+c0*8^0.

An example:

```
3407 = 3*8^3 + 4*8^2 + 0*8^1 + 7*8^0
     = 3*512 + 4*64          + 7*1
     = 1799
```

It is sometimes not clear which number system a particular string of digits belongs to (eg the "3407" in the last example could be a decimal number; the only way you know it is octal is that it is clear from the context). The convention is to append a subscript showing the base the number belongs to.

Eg: 3407_8 means this is an octal number.

There are other conventions as well. Binary numbers frequently have the letter "b" appended to them as in "10011b". Hexadecimal numbers can have a letter "h" appended or there may be a "0x","#","$" preceding the digits as in "0ah","0x0a","$0a","#0a". The "0x" is made standard by the C language and the "$" and "#" show up frequently in z80 assemblers.

The main reason that hexadecimal and octal see use is that they can represent binary numbers compactly. Let's perform some operations on a long binary number to see why.

```
0100 1101 1001
= (0100)*2^8     + (1101)*2^4     + (1001)*2^0
= (0100)*(2^4)^2 + (1101)*(2^4)^1 + (1001)*(2^4)^0
= (0100)*16^2    + (1101)*16^1    + (1001)*16^0
= $4D9
```

Recall that multiplying a binary number by 2 is the same as adding a "0" to the end of the number. Multiplying by 2^n is the same as multiplying by 2 n times, adding n "0"s to the end of the number. If you find the first line difficult to follow, just expand the 4-bit binary numbers by appending the "0"s demanded by the multiplications, add the result, and see that the expansion is valid.

The same number operated on in a different way:

```
010 011 011 001
= (010)*2^9 + (011)*2^6 + (011)*2^3 + (001)*2^0
= (010)*(2^3)^3 + (011)*(2^3)^2 +
  (011)*(2^3)^1 + (001)*(2^3)^0
= (010)*8^3 + (011)*8^2 + (011)*8^1 + (001)*8^0
= 2331 (octal)
```

In the first case, the binary number is broken up into nibbles (groups of four bits) and expanded as a shifted sum of 4-bit binary numbers. Then 16 (2^4) is factored from each term and we wind up with something that looks like the definition of a hexadecimal number (base 16, a number expressed as a weighted summation of powers of 16). Effectively, this is a shortcut for converting binary numbers to hexadecimal numbers, and the procedure is as follows:

CONVERT BINARY TO HEX:
1. Divide the binary number into nibbles, starting with the least significant bits.
2. Rewrite each nibble with the hexadecimal digit representing its value. The table of values at the beginning of this section can help.

You can also go in the opposite direction by reversing the process.

CONVERT HEX TO BINARY:
1. Replace each hex digit with the 4-bit binary string representing the same numerical value. The table of values at the beginning of this section can help.

The second case above shows that it is also easy to convert between binary and octal. This time the binary number is broken into groups of three bits, each of which can be represented by an octal digit. Again, the process can be reversed, starting with an octal number and converting it to binary by replacing each octal digit by a string of three bits representing the same value. Octal has fallen into disuse and will only be mentioned sparingly in this series.

See if you can perform these conversions and get the same results:

```
10001000111010 = $223a
$4ead = 0100111010101101
```

2.4 CONVERTING BETWEEN BASES

We have seen a number of shortcuts for converting between certain bases already. After a brief review, we'll have a look at a general method for converting between bases that doesn't rely on tricks.

* Converting Binary to Decimal

The easiest method is based on the definition of a binary number, eg:

```
1010.11
= 1*2^3+ 0*2^2 + 1*2^1 + 0*2^0 + 1*2^(-1) + 1*2^(-2)
= 1*8  + 0*4   + 1*2   + 0*1   + 1*0.5    + 1*0.25
= 10.75
```

But instead of actually performing the expansion and multiplying by powers of two, we take note that each bit's

weight doubles as one moves to the left and halves as one moves to the right.

So we say that bit position 0 has weight 1, bit 1 has weight 2, bit 2 has weight 4, etc. Where there's a "1" in a particular bit position, we add the weight to the final total.

Eg: 10001110101 = 1 + 4 + 16 + 32 + 64 + 1024 = 1141

Here we start from bit 0 and know its worth is one and add that to the total. Bit 1 is worth two but it's a "0" in this example so that is not added to the total. Bit 2 is worth four; it's a "1" so four gets added to the total, etc.

With the fractional part of a binary number we do something similar. We start with the first fractional digit, having weight 0.5. The next digit to the right has half the weight, or 0.25, etc.

Eg: 0.10011 = 0.5 + 0.0625 + 0.03125 = 0.59375

* Converting Binary to Hexadecimal

The binary number is grouped into nibbles, starting with the least significant bits. Then each nibble is replaced by a single hexadecimal digit that represents its value.

Eg: 11101011011 = 111 0101 1011 = $75B

* Converting Hexadecimal to Binary

Each hexadecimal digit is replaced by a four-bit binary number representing the same value.

Eg: $6AF7 = 0110 1010 1111 0111

* General Conversion Method

To illustrate the general conversion method, we will look at converting a decimal number to binary. We start with the whole part, postponing dealing with the fractional part.

We start with the source number:

```
123 (decimal)
```

We know it has a binary representation:

```
B6b5b4b3b2b1b0
```

If we take this binary number and divide by two, we get:

```
B6b5b4b3b2b1.b0
```

The fraction part of this number will be b0*2^(-1). In other words, "0.5" if b0=1 or "0" if b0=0.

Since our decimal number is this same number, we can divide it by two and determine what b0 is:

```
123 / 2 = 61.5
```

Our result has fraction part "0.5" which means b0=1.

We strip off this fractional part and have the numbers: `61 (decimal)` and `b6b5b4b3b2b1` which also represent the same quantity.

Now we repeat the procedure until all the binary digits are determined. Doing so, you will find:

```
FACT: Decimal Number 123 has binary representation
      b6b5b4b3b2b1b0

123/2 = 61.5  ; b6b5b4b3b2b1b0 / 2 = b6b5b4b3b2b1.b0
  ==> b0 = 1, strip off fraction of result
 61/2 = 30.5  ;   b6b5b4b3b2b1 / 2 = b6b5b4b3b2.b1
  ==> b1 = 1, strip off fraction of result
 30/2 = 15.0  ;     b6b5b4b3b2 / 2 = b6b5b4b3.b2
  ==> b2 = 0, strip off fraction result
 15/2 =  7.5  ;       b6b5b4b3 / 2 = b6b5b4.b3
  ==> b3 = 1, strip off fraction result
  7/2 =  3.5  ;         b6b5b4 / 2 = b6b5.b4
  ==> b4 = 1, strip off fraction result
  3/2 =  1.5  ;           b6b5 / 2 = b6.b5
  ==> b5 = 1, strip off fraction result
  1/2 =  0.5  ;             b6 / 2 = 0.b6
  ==> b6 = 1
```

The conclusion: 123 = 1111011

It is also possible to determine what a specific bit in the final binary number would be. In this example, with 123 as the source number and b6b5b4b3b2b1b0 as its binary representation, we could have determined what bit 5 (b5) was as follows:

```
123 / 2^(5+1) = b6b5b4b3b2b1b0 / 2^(5+1)
     1.921875 = b6.b5b4b3b2b1b0
```

If b5 is a "1", then the fraction part of the result of this division must be at least a half. If b5 is a "0", then the fraction part of the result of this division must be less than half. In this case, 0.921875 >= 0.5 so we know that b5 is a "1".

Let's now illustrate how to convert a decimal fraction to a binary fraction.

We begin with the source number 0.0973 which we know has a binary representation 0.f1f2f3f4f5...

If both numbers are multiplied by two, we find that:

```
2*0.0973 = 2*0.f1f2f3f4f5...
  0.1946 = f1.f2f3f4f5....
```

Since the whole part of the result is 0, we know that f1=0.

Repeating, equating the fractional result:

```
2*0.1946 = 2*0.f2f3f4f5...
  0.3892 = f2.f3f4f5...
  ==> f2 = 0, equate fractional results

2*0.3892 = 2*0.f3f4f5...
  0.7784 = f3.f4f5f6f7...
  ==> f3 = 0, equate fractional results

2*0.7784 = 2*0.f4f5f6f7...
  1.5568 = f4.f5f6f7...
  ==> f4 = 1, equate fractional results

2*0.5568 = 2*0.f5f6f7...
  1.1136 = f5.f6f7f8...
  ==> f5 = 1, equate fractional results.
```

I'll stop there and say that 0.0973 is approximately equal to 0.00011 in binary. In actuality, the binary number 0.00011 equals 2^(-4)+2^(-5) = 0.09375. Since the last binary digit has weight 2^-5 = 0.03125, I know that the result will have an error no larger than 0.03125. The actual error is 0.0973-0.09375=0.00355.

This method can be generalized to convert between the numbers of any two bases.

Q:
How many binary fraction places are needed to represent a decimal fraction accurate to six places?

A:
The sixth digit of a decimal fraction has weight 10^-6. The Nth binary fraction place has weight 2^-N. If the binary fraction is truncated to the Nth place, we know the error will be no larger than 2^-N. Therefore, to represent a number with accuracy 10^-6, we can solve "2^-N=10^-6" for N to find how many binary positions are needed. Doing so, N=19.9 or at least 20 binary fraction digits.

2.5 TWO'S COMPLEMENT

So far discussion has managed to avoid the subject of negative numbers, except for a hint that binary subtraction can be done using addition(!).

In the decimal number system, negative numbers have a "-" sign prefixed to them. This is known as sign-magnitude representation. The "-" sign says this is a negative number and the number itself gives a magnitude.

One could do the same in binary. In fact, feel free to do so in your programs, but you will find out shortly that there are better ways to go about things most of the time. In computers, there is no "-" sign, just 0 and 1. So it is common to take "0" as meaning positive (no negative sign) and "1" as meaning negative (presence of a negative sign). The sign is typically the leftmost bit in a signed binary number with the remaining bits forming the magnitude. Here are some examples of sign-magnitude in binary:

```
1 10011 = -19
0 10111 = 23
```

If your program were given two numbers to add, you'd have to consider several cases. If the first bit of both numbers was "1" then both numbers are negative and you would add the two magnitudes and place a "1" in front of the sum to indicate the sum was also negative. If the first bit of the first number was "1" and the first bit of the second number was "0" then you would determine which of them had a larger magnitude, subtract the smaller from the larger, and place either a "1" or "0" in front of the sum depending on whether the larger magnitude was negative or positive. There are two more cases to consider, which I leave to you to ponder.

In short, this is painful. Circuitry could be implemented to do all this work but this is also painful to do in hardware and would add to the transistor count, increasing the cost of the final product.

Instead, two's complement representation of numbers was invented. Two's complement is a specific case of the more general "radix complement" representation. "Radix" is a fancy term for "base" and, as you may have guessed, radix complement notation can also be used to generate ten's complement numbers in decimal or sixteen's complement numbers in hexadecimal. We will only bother with the specific two's complement case for binary as nobody uses ten's complement or sixteen's complement.

In two's complement, binary numbers must be a fixed number of bits in size. Negative numbers are defined as follows:

Negative binary number = 2^N - B = (2^N - 1) - B + 1

Where "N" is the number of bits in the binary number and "B" is the magnitude of the negative number. Positive numbers are represented as before. Pay attention to this formula in the following discussion.

Recall from the end of section 2.2 that "2^N-1" is a binary string of "N" 1s. Eg: 2^8-1 = 1111 1111. Subtracting an N-bit binary number "B" bit by bit from a string of N 1s is equivalent to complementing each bit. If a particular bit in B is 0, then 1-0 will make it into a 1. If a particular bit in B is 1, then 1-1 will make it a 0. According to the formula, the two's complement representation of a negative number can be had by first complementing the positive representation of the magnitude of the number and then by adding 1.

Here are some examples of both positive and negative numbers in two's complement form, assuming 8 bit binary (N=8).

```
   116  = 0111 0100
  -116  = Complement(0111 0100)+1
        = 1000 1011 + 1
        = 1000 1100
-(-116) = Complement(1000 1100)+1
        = 0111 0011 + 1
        = 0111 0100 = 116
   -13  = Complement(0000 1101)+1
        = 1111 0010 + 1
        = 1111 0011
```

The third example above shows that taking the negative of a negative number returns the absolute value of the number, as expected.

Since two's complement requires binary numbers to be confined to N binary digits, we can only represent a finite range of numbers determined by the size of N. If we confine the range to 2^(N-1)-1 for most positive number to -2^(N-1) for most negative number, then negative numbers will always start with a "1" and positive numbers will always start with a "0". That way you can easily spot whether a number is positive or negative. Have a second look at the few examples above to see this is the case.

If our numbers are 8 bit (N=8), we can represent a range of -128 through 127 using two's complement. With 16 bits, we have a range of -32768 through 32767. Zero has the usual value (all 0 bits).

What makes two's complement so wonderful is that you can add, multiply and divide numbers exactly as before without any changes to the familiar algorithms. When adding or multiplying numbers, excess bits are simply stripped off. Here are some examples, assuming 8-bit two's complement:

```
  ,,   ,,
  0100 0011 ( 67)
+ 1110 0011 (-29)
  ---------
1 0010 0110 --> 0010 0110 (lose the extra bit) = 48

    , ,,,
  0000 1111 ( 15)
+ 1010 1111 (-81)
  ---------
  1011 1110 --> -(Complement(1011 1110)+1) = -66
```

Above, the leading "1" indicates the result is negative, so we take the negative of that to find the magnitude of the negative number, which is 66.

```
  0111 0010 (114)
+ 0110 0001 ( 97)
  ---------
  1101 0011 --> -(Complement(1101 0011)+1) = -45???
```

Something is wrong in the final example because 114+97 is supposed to be 211 not -45. Overflow has occurred: the result is larger than 127, which is the largest positive number that can be represented in 8-bit two's complement. Underflow can also occur if the result is too negative to be represented. There are rules for detecting overflow and underflow that can determine if a result is valid or not. These rules are typically implemented in hardware so that programs can test the validity of results, if needed.

Before leaving the topic of two's complement, I will also mention that there is such a thing as one's complement. In two's complement, negative numbers are represented by complementing the bits representing the magnitude of the negative number and then adding one. In one's complement, negative numbers are represented by complementing the bits representing the magnitude of the negative number but one is not added. This results in the existence of two zeroes: +0 has all zero bits while -0 has all one bits. The rules of arithmetic are also slightly different. Numbers are added and multiplied in the normal way, but the excess bits are later added to the result rather than being stripped off as in two's complement.

SUMMARY:

* Two's complement allows negative and positive numbers to be represented in such a way that all the normal algorithms for addition, subtraction, multiplication, etc. can be applied to generate correct results.

* A two's complement number has a fixed length in bits. If the result of an addition or multiplication results in more bits, the excess bits are just stripped off.

* Overflow occurs when the result of a mathematical operation is a number too positive to be represented with the finite number of bits used in the two's complement number. Likewise, underflow occurs when the result is too negative to be represented. Hardware identifies when overflow or underflow occurs so that programs can determine if results are valid, if necessary.

* The range of numbers that can be represented by an N-bit two's complement number is -2^(N-1) from the negative side through 2^(N-1)-1 on the positive side. Zero always has a representation that is all 0 bits.

* Positive two's complement numbers have the same representation as the positive binary numbers looked at in a previous section. A negative two's complement number is formed by adding one to the complement of the absolute value of the negative number.

* The sign of a two's complement number can be determined by examining the most significant bit of the

number. A one indicates a negative number while a zero indicates a positive one.

2.5.1 Sign Extension

Two's complement relies on the fact that a fixed number of bits are used to represent each number. From time to time, you will find the need to add differently sized two's complement numbers. For example, you may want to add an 8-bit two's complement number to a 4-bit two's complement number.

A correctly represented two's complement number (by "correctly represented" I mean a two's complement number in the allowable range) will have its most significant bit "1" if the number is negative and "0" if it is positive. It turns out that negative two's complement numbers will have a whole string of leading 1s and positive two's complement numbers will likewise have a whole string of leading 0s. Here are some examples using 8 bits:

```
 -13 = 1111 0011
 -46 = 1101 0010
-100 = 1001 1100
  13 = 0000 1101
  46 = 0010 1110
 100 = 0110 0100
```

Notice how -13 has a string of four leading 1s. We could lose three of these leading 1s and still know the number is negative. It turns out that we can remove these three leading 1s and be left with "1 0011" which is the correct 5-bit two's complement representation of -13. We could also remove just two of the leading 1s leaving "11 0011" which also happens to be the correct 6-bit two's complement representation of -13.

If we investigate the 8-bit two's complement representation of +46, we see that there are two leading 0s. We could lose one of them to get the valid 7-bit two's complement representation of +46. If we lose both 0s, the first bit would be a "1", misleading us into thinking the resultant two's complement number was negative. The conclusion is that there is no 6-bit two's complement representation of +46 because it is out of the allowable range for 6-bit two's complement numbers (can you confirm this from the discussion in the last section?).

The binary representation of 100 has just a single leading 0, so it must be represented with at least 8 bits using two's complement.

The result of this discussion is that a two's complement number can be made into a larger two's complement number by sign extending it. That is, by adding bits on the left equal to the sign bit of the original number. For example, the 4-bit two's complement number "1101" can be made into an 8-bit two's complement number by adding four more bits on the left equal to the sign bit "1": "1111 1101". This process is called "sign extension".

Similarly, an 8-bit two's complement number can be reduced in size by removing redundant sign bits. For example, the 8-bit two's complement number "0001 0010" can be represent by the 6-bit two's complement number "01 0010" by eliminating two redundant sign bits.

2.6 FIXED POINT

Fixed point numbers are used to represent numbers with a whole part and a fractional part. Fractional numbers were discussed in section 2.2. What is new is that a fixed point number format has a predetermined number of bits and "fixes" the position of the binary point within those bits. Fixed point numbers are often described as having an "A.B" format meaning there are A bits for the whole part of the number and B bits for the fractional part. Fixed point numbers of this format always have exactly A+B bits.

For example, the 5.3 unsigned fixed point number "1011 0101" is understood to mean "10110.101" which represents 22.625. Such numbers use 8 bits in total, the first 5 bits representing the whole part (for a range of 0 to 2^5-1=31) and the remaining 3 bits representing the fractional bit (for a precision of 2^-3=0.125). You can apply any mathematical operation to fixed point numbers of the same A.B form as if they were normal unsigned binary numbers.

Adding and subtracting fixed point numbers of different formats involves shifting one of the numbers so that the binary point aligns before performing the operation. For example, to add an unsigned 8.3 number to an unsigned 5.2 number, one could add a '0' to the end of the 5.2 number to make it 5.3, then add three leading 0s to make it 8.3. Then both numbers have the same fixed point format and can be added.

To multiply two fixed point numbers, first multiply each by a power of two to eliminate the binary point, then multiply as usual, and then divide by the same powers of two to get the result.

Thus far, I have been careful to call these numbers "unsigned". Fixed point numbers can also be "signed", representing negative quantities as well as positive. "Signed" numbers in binary always implies two's complement. To generate a negative quantity in fixed point, first represent the magnitude of the negative number as a positive quantity, then convert it to two's complement, ignoring the presence of the binary point (ie, complement it then add 1).

The previous unsigned 5.3 fixed point number could represent values from 0 through (2^8-1)/2^3=31.875 (there are 8 bits with range 0 to 2^8-1; divide by 2^3 to

get the three fraction bits). A signed 5.3 fixed point number can represent values from -2^(8-1)/2^3=-16 through (2^(8-1)-1)/2^3=15.875. Again, the range of two's complement values for 8 bits is found and then divided by 2^3 to account for the three fraction bits.

Q:
Assuming the following 8-bit binary numbers are signed 5.3 fixed point, what are the decimal equivalents?

1011 1001, 0011 1111, 0101 0101

A:
10111.001 = (1011 1001)/8 = -71/8 = -8.875, 7.875, 10.625

Q:
Assuming the following 8-bit binary numbers are signed 6.2 fixed point, what are the decimal equivalents?

1011 1001, 0011 1111, 0101 0101

A:
101110.01 = (1011 1001)/4 = -71/4 = -17.75, 15.75, 21.25

Q:
Assuming the following 8-bit binary numbers are unsigned 5.3 fixed point, what are the decimal equivalents?

1011 1001, 0011 1111, 0101 0101

A:
10111.001 = (1011 1001)/8 = 185/8 = 23.125, 7.875, 10.625

Q:
Represent the following decimal numbers as a 5.2 signed fixed point number.

-9.2, 4.1, -0.8

A:
(i) -9.2. Multiply this by four to account for the two fractional positions of the 5.2 format and try to represent the integer result using a two's complement 7-bit number. We will divide by four later to change the 7-bit number to 5.2 format.

-9.2*4 = -36.8, round to -37. In seven bits, 37 = 010 0101. Negate it to form the 7-bit two's complement number 101 1011 (-37). Finally, divide by four to get the 5.2 fixed point result: 10110.11, or 101 1011 in 5.2 format.

(ii) 4.1 = 001 0000 (approximately)

(iii) -0.8 = 111 1101 (approximately)

2.7 FLOATING POINT

An important feature of fixed point numbers is that adding and subtracting them is very quick on the z80. On later generation microprocessors (including z80 derivatives like the ez80), hardware is also present for multiplying and dividing integers, which can also be used to multiply and divide fixed point numbers.

However, fixed point numbers have a couple of drawbacks. One is precision. A 5.3 format fixed point number always has three binary fraction digits, meaning the finest granularity fractional number that can be represented is 2^-3 = 0.125. If granularity smaller than that is needed, more bits are required. Sometimes this can be a lot of bits -- an example problem at the end of section 2.4 showed that 20 binary fraction bits are needed to represent a number accurate to six decimal places.

The other drawback is range. By choosing a 5.3 format, you are confining all numbers represented and the results of all mathematical operations to a fixed range of representable numbers.

Floating point was invented to transcend these limitations. As its name implies, the binary point is not fixed at a specific bit position in the binary representation -- it is allowed to float. Floating point numbers are very similar to scientific notation numbers in the decimal system. With scientific notation, there are some number of significant digits used to represent the number to some fixed precision and this is multiplied by a power of 10 to get the number's magnitude (and therefore place the "floating" decimal point among the digits present). Floating point separates the binary digits used to represent the number to some precision into a "mantissa" and the power of 2 used to get the magnitude into an "exponent". Numbers are represented in sign-magnitude form so there is a single sign bit as well.

Until the late '70s and early '80s there wasn't really a standard bit format for floating point numbers and different math software could very well use different and incompatible floating point representations. That changed with the advent of math co-processors in the early '80s, with the hardware adopting IEEE-754 as the standard representation for floating point. IEEE-754 describes a single precision and a double precision floating point representation, with the latter having twice as many mantissa bits and more exponent bits so that numbers could be represented more precisely and the range of numbers was larger. Needless to say double precision numbers take longer to multiply and divide in the hardware.

The Spectrum (and the z80) does not feature hardware support for floating point so all mathematical operations on floating point numbers must be done through software. Thankfully the Spectrum ROM contains a very handy and sophisticated reverse Polish notation

calculator that can be used from machine code to apply a whole list of mathematical operations to floating point numbers. This is something we will revisit later in the series. The Spectrum uses its own unique floating point number format and has just one precision. A floating point number is five bytes long and is formatted thus:

```
EEEE EEEE SMMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM
```

The most significant eight bits are the exponent, followed by a single sign bit (with a "1" meaning negative) and followed finally by 31 mantissa bits to represent the binary digits of the number. In actuality, there are 32 mantissa bits as a leading "1" is understood to be there but is not stored. All binary numbers in floating point start with a leading "1" which is why it doesn't need to be stored (can you see why? -- answer below. What is the only number that does not have at least a single "1" bit?) The exponent has a bias of 128 added to it. If the actual exponent is 5, 133 would be stored. If the exponent was -10, 118 would be stored. The bias simplifies the floating point software.

An example will demonstrate how decimal numbers can be converted to the Spectrum's floating point format. Let us represent "-3.141592653".

First we note that the number is negative so the sign bit will be "1". Then, with the help of a calculator, we can use the general conversion method described in section 2.4 to quickly convert the positive number "3.141592653" to unsigned binary. The result is "11.001001000011111101101010100001", where I have stopped at 32 bits since the Spectrum's mantissa is only 32 bits long including the understood leading "1".

The mantissa is understood to be at least a half but less than 1 so we rewrite this number such that there is no whole number part and a one following the binary point: "0.11001001000011111101101010100001 * 2^2". The exponent is 2 and after adding the bias is 130. We remove the first 1 in the 32-bit mantissa to get the remaining 31 bits stored in the Spectrum's floating point format. Using these results we have:

```
-3.141592653
  EEEE EEEE SMMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM
= 1000 0010 1100 1001 0000 1111 1101 1010 1010 0001
= $ 82 C9 0F DA A1
```

We can also go the other way around, starting with some number in Spectrum floating point and finding its value in decimal. Let's try with a random bit pattern:

```
EEEE EEEE SMMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM
0110 1011 0011 0100 1110 1111 0101 0000 0110 1110
```

The eight exponent bits represent 107 in decimal. We subtract the 128 bias to get -21. The sign bit is 0, indicating a positive number. This leaves us with a 32-bit mantissa which represents the digits in the number. Inserting the leading "1" understood to be present in the mantissa we have:

```
1MMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM
1011 0100 1110 1111 0101 0000 0110 1110
```

Convert this to hex: $B4EF506E. Then convert this hex number to decimal:

```
$B4 EF 50 6E
= 11*16^7 + 4*16^6 + 14*16^5 +
  15*16^4 + 5*16^3 + 6*16^1 + 14*16^0
= 3035582574
```

The result is that this mystery number in decimal is:

```
+3035582574*2^-21 = 1447.478568
```

The floating point number format assumes that there is at least a single "1" bit in the binary number being represented. The number is multiplied by a power of 2 so that the leading "1" bit appears just after the binary point. It is then ignored (assumed to be there) when creating the mantissa. Not actually storing that "1" bit allows an extra bit of precision to be stored. But there is one number that has no "1" bits in it, and that is 0. To represent 0, we need to have an exception. In Spectrum floating point, an all-zero floating point number is understood to represent the number 0. The IEEE-754 standard defines several other exceptions, including representations for negative infinity, positive infinity and not-a-number, but these are not supported by the Spectrum's math routines.

The Spectrum calculator routines also support a special integer type for representing integers from -65535 to +65535. Five bytes are also used with the first byte being 0, the next three bytes being the two's complement representation of a number in the range +-65535, and the final fifth byte is also set to 0. These integers and regular floating point numbers can be mixed at will, with the calculator routines identifying integers by the 0 in their exponents. Mathematics on pure integers is faster. Integers are automatically promoted to floating point whenever necessary.

Spectrum BASIC is one of a few early BASICs that actually had separate floating point and integer types. Most BASIC programmers were probably unaware of this as this was all transparent to the user ("LET a=1" created an integer variable whereas "LET a=1.0" created a floating point variable), but it became an issue for those who tried BASIC compilers, since not all supported floating point numbers and were sold as integer-BASIC compilers only.

2.8 OTHER INTERPRETATIONS OF 0s AND 1s: BCD AND ASCII

It's easy to get lost in all the previous discussion and forget that there are other interpretations of bits than simply as binary numbers. In fact, you can attribute whatever meaning you want to any bit pattern and write software to manipulate those representations in a meaningful way. What makes the binary number types described previously "special" is that there is support for manipulating them in either the z80 circuitry or the software in the Spectrum ROM. For example, addition and subtraction of two's complement numbers is supported by z80 circuitry through "ADD" and "SUB" instructions and multiplication and division of floating point numbers is supported by the calculator in the Spectrum ROM.

Two other types relevant to the z80 and the Spectrum are BCD and ASCII. BCD stands for "binary coded decimal". It's a means of representing decimal numbers directly in binary. Each decimal digit is represented by the corresponding 4-bit binary number. If a single 8-bit byte contains two decimal digits side-by-side, the number is called packed BCD. The z80 supports packed BCD numbers with special instructions that allow one to add, subtract, and shift them right or left. BCD is next to useless so we won't spend a lot of time on it. Here are some examples of two-byte packed BCD numbers:

```
0000 0001 0101 0110 = 0156
0010 0011 0100 0101 = 2345
```

ASCII maps the English alphabet, punctuation, numerals and data communication control codes to a 7-bit binary word. The Spectrum ROM supports a variation of ASCII in a full byte. The data communication control codes are replaced by special characters the ROM interprets as colour control bytes and cursor movement bytes. User-defined graphics characters are represented as well as all the tokens used in Spectrum BASIC. The list of byte to character mappings can be found at the back of the Spectrum's user manual [3, Appendix A].

2.9 NUMBER SYSTEMS - A FINAL NOTE

If you managed to persevere this far, give yourself a pat on the back. We have covered a lot of ground in a short period. We have seen unsigned binary numbers, binary fractions, two's complement, fixed point and floating point representations. We have seen a convenient way for representing bit patterns using hexadecimal and we have seen means for converting between decimal and all of the binary number representations discussed. You will use this knowledge frequently when writing your machine code programs and if you have managed to absorb it all, you are well on your way to becoming a competent assembly language programmer.

References

[1] Bobby Brey, "Programming the 80286, 80386, 80486 and Pentium-Based Personal Computer," (c) 1996 Prentice-Hall Inc.

[2] http://www.ahinc.com/chips.htm. There is a lot of inaccurate information on the internet concerning the history of microprocessors and the actual specs of early microprocessors. This page mentioned transistor count and seemed to have the fewest errors of the pages I found. The transistor count looks reasonable given the consistent numbers widely reported for the 8080.

[3] "ZX Spectrum User Manual."
http://www.madhippy.com/8-bit/sinclair/zxspecman/index.html

[4] Philip Robinson, "The Programming Guide to the Z80 Chip," (c) 1984 TAB Books.

[5] David Johns & Ken Martin, "Analog Integrated Circuit Design," (c) 1997 John Wiley and Sons.

[6] "Zilog Z80 User Manual."
http://www.z80.info/zip/z80cpu_um.pdf