First Steps in Z80 Assembly Language

(v1.2.4)

Introduction

Does the world really need another book on Z80 assembly language, in 2020? After all, there are many published texts from the 1980s, available to buy second-hand or downloadable for free from the Internet. And besides, the technology is obsolete, isn't it?

Not quite. The motivation behind this book is the recent release of the Sinclair ZX Spectrum Next – a brand new 8-bit computer, serving as a successor to the original line of Spectrums. I'm excited about this retro-styled machine, as are three-quarters of a million pounds worth of Kickstarter backers, and I want to see a thriving game development scene emerging. That will only happen if more people break away from a dependence on high-level languages like BASIC and embrace the challenge of low-level (machine-level) coding.

I've received a great deal of thanks for a Z80 video tutorial series that I uploaded to YouTube. In just twelve months, the first tutorial accrued 19,000 views, proving that there is still substantial interest in assembly language coding. But more important than that, many viewers have thanked me for being the first person to make Z80 comprehensible to them, after their past efforts with the available texts. I believe this is due to my hands-on approach. Instead of forcing the learner to digest chapter after chapter of technical information before having the opportunity to do anything practical, I flipped that logic backwards and got the learner coding (and seeing on-screen results) right from the beginning. That successful approach to teaching is why I believe a new book is in order.

This will not be a comprehensive text on Z80. Those are already available and there's no need to reinvent the wheel. The purpose of this volume is to help you over the hurdle of how alien assembly language appears, especially for a high-level programmer accustomed to a language that strives to mimic plain English. And the modern-day coder is inevitably a high-level coder. Ironically, a proficiency in high-level coding is something of a stumbling block when learning assembly language, because you can't always find equivalent programming structures. I will attempt to undo this problem by teaching assembly language using comparisons with BASIC. I'm going to assume that you've already dabbled a little in BASIC and have a familiarity with the meanings of common instructions like `PRINT`, `GO TO`, `GO SUB`, `RETURN`, `LET`, `FOR`, `NEXT`, `IF`, `THEN`. When you're done with this book, you'll be able to tackle a more exhaustive Z80 text with confidence.

The lessons in this volume are applicable to the Sinclair ZX Spectrum 48K. With small adjustments, the same teaching can be applied to other computers featuring the Zilog Z80 microprocessor.

1: Your First Z80 Program

"Hello, World!"

Practically every programmer has created a version of the "Hello, World!" program in a high-level language. Usually, it's our very first attempt at coding:

```
10 PRINT "HELLO"
```

Here's how you might program this in Z80 assembly language:

```
LD A, "H"
RST 16
LD A, "E"
RST 16
LD A, "L"
RST 16
LD A, "L"
RST 16
LD A, "O"
RST 16
RET
```

This is not the most efficient way to code this, but it's the least confusing (relatively speaking!), because there are only three instructions to learn. So this is where we'll start.

`LD A, "H"` is similar to the BASIC instruction `LET A$="H"`. A is known as a register. For the time being, you can think of registers as variables. In plain English, this would read as "Load register A with the value 'H'."

`RST 16` is shorthand for "Restart 16". For now, don't worry about why it's called that, or why we're using the number 16. Only concern yourself with what it does. `RST 16` looks at whatever is stored in A and prints it to the screen. This may be confusing to a high-level language programmer, because in high-level languages everything is spelled out in great detail. If `RST 16` does something with A, then you would expect A to be *explicitly* specified somewhere in the instruction. Not so with Z80. As you learn more Z80 instructions, you're going to find that some of them perform quite elaborate tasks *implicitly*, without all of the details of those tasks being visible in the assembly language. You might be wondering how we get `RST 16` to print the contents of a different register, such as B. The simple answer is, it can't. `RST 16` only works with A. Isn't that restrictive? Not really. It just requires a different way thinking than you're accustomed to.

The final instruction is `RET`, which, as you can probably guess, will stop the program execution and return to BASIC. It's a common mistake to forget to include this, because BASIC programs don't require it; there is a BASIC instruction called `STOP`, but a program returns to BASIC regardless of whether `STOP` is present or absent. Not so in machine code. If you fail to end your Z80 program with `RET`, it will crash.

Believe it or not, you can create and execute this assembly language program directly from the Spectrum's BASIC interpreter, without any additional software. All you need is a Z80 reference

guide (see appendix A of the ZX Spectrum BASIC programming manual) and a rudimentary understanding of the BASIC instruction POKE.

Every assembly language instruction has a corresponding number – its *machine code* value. Here are the numbers for the three instructions used in the program above:

Assembly language	*Machine code*
LD A, *n*	62
RST 16	215
RET	201

Throughout this book I will be using the convention *n* to denote any numerical value in the range 0 to 255, and *nn* for larger numbers 0 to 65535.

Note that we don't have separate machine code numbers for "LD", "A", and the comma. The entire instruction `LD A, n` is a single machine code number (62), known as an *opcode*. Whatever we decide to put in *n* is known as the *operand*. Z80 instructions are powerful but simple in structure, designed to take up as little memory as possible. They either consist of a single opcode, or a single opcode followed by a single operand.

The operand in our program's first instruction is "H". Letters on the Spectrum are arranged according to a numbering system known as ASCII (the American Standard Code for Information Interchange). We'll go into this in more detail later. What you need to know for the current program is that the letters A to Z are positioned from 65 to 90 in the ASCII table. So if A is 65, H must be 72.

The first line of our program, stated in machine code, is 62 followed by 72, which is the equivalent of `LD A, n` followed by "H" (opcode followed by operand). How does the computer know when it's an opcode and when it's an operand, since everything is reduced to numbers? `LD A, n` is an opcode where an operand is automatically expected as the next piece of data in sequence. Conversely, `RET` is an opcode that does not have an operand, therefore one is not anticipated. As long as you adhere to the rules of the language, the computer will not become confused by the numbers.

To program this into the Spectrum, you need to decide where to place it in memory. A full discussion of memory will take place in the next chapter. For now, we'll arbitrarily choose memory address 50,000 as the starting point for the program.

Peeking & Poking

Using Spectrum BASIC, take a "peek" at what's in address 50,000 by typing the instruction `PRINT PEEK 50000`. Try that now. It should return the value 0, meaning that the memory location is currently unused.

Type `POKE 50000,62`. This places the Z80 opcode `LD A, n` into address 50,000. Now put 72 into the next available slot using `POKE 50001,72`. You'll notice that you're getting no feedback from the computer that anything is actually taking place, but you can check that you truly are building a program by typing `PRINT PEEK 50000` again. This time, the computer should respond with 62, demonstrating that you've stored the number 62 in memory location 50,000.

You're now ready to continue building the program to its finish. I recommend using BASIC line numbers with appropriate REM (remark) statements, to keep everything organised. REM statements are used by programmers to insert helpful notes into programs. They are ignored by the computer during the execution of a program.

```
 10 REM LD A, "H"
 20 POKE 50000,62: POKE 50001,72
 30 REM RST 16
 40 POKE 50002,215
 50 REM LD A, "E"
 60 POKE 50003,62: POKE 50004,69
 70 REM RST 16
 80 POKE 50005,215
 90 REM LD A, "L"
100 POKE 50006,62: POKE 50007,76
110 REM RST 16
120 POKE 50008,215
130 REM LD A, "L"
140 POKE 50009,62: POKE 50010,76
150 REM RST 16
160 POKE 50011,215
170 REM LD A, "O"
180 POKE 50012,62: POKE 50013, 79
190 REM RST 16
200 POKE 50014,215
210 REM RET
220 POKE 50015,201
```

Type `RUN`. This will not execute the Z80 program. It will only execute the BASIC program, and the function of that program is to copy the data for the Z80 program into memory addresses 50,000 to 50,015. Finally, to execute Z80 program, type `RANDOMIZE USR 50000`.

What?! Nothing happens! Actually, something did happen, but it happened so fast that you missed it. The Spectrum has the quirk of writing "0 OK 0:1" at the bottom of the screen when it finishes an operation – right on top of where it wrote the message "HELLO". You may have expected the message to be written at the top-left, but it happens to use the bottom-left by default. The Spectrum's BASIC interpreter divides the display into the *upper screen* (the main portion where your program listings appear) and the *lower screen* (the horizontal slice at the bottom where you type). Printing in machine code defaults to the lower screen. Later, we'll change that. But for now, to prove that the program really is working, type two commands together, each separated with a colon: `RANDOMIZE USR 50000: PAUSE 0`. When 0 is placed after PAUSE, the computer waits until a key is pressed before returning you to BASIC and splashing its OK message over your work.

Congratulations! You've written your first Z80 program, using 16 bytes of memory. You'll be glad to hear that we won't be relying on POKE to create larger machine code programs. There's a much easier way to code in Z80, using a tool called an assembler. Nevertheless, I felt that it was important to show you this as a first step, so that you're fully aware of what's going on "under the hood" when you rely on an assembler.

Here's your program, in machine code:

62, 72, 215, 62, 69, 215, 62, 76, 215, 62, 76, 215, 62, 79, 215, 201

You've just done something that few programmers today ever do; you've communicated with your computer in its native language: numbers. I could take you one step closer to the computer's native language by converting these decimal numbers to binary, because the computer really only has two digits at its disposal, 0 and 1, corresponding to a flow of electricity through circuits that is either off or on. But that would be overkill for chapter 1.

2: Understanding the Spectrum's Memory

Mapping the Memory

BASIC programs feature line numbers, and so do assembly language programs (in a manner of speaking). BASIC allows you to put any line numbers before instructions. Typically, you would start at 10 and move forward in multiples of ten. Everybody remembers this program:

```
10 PRINT "Hello, world!"
20 GO TO 10
```

When you type a program into the BASIC interpreter, the computer stores it somewhere in memory, but it doesn't tell you where, because it doesn't need to burden you with that information. By contrast, assembly language gives you much more control over the contents of the computer's memory. The programmer is expected to choose precisely where in memory a program should be stored. So, before we can proceed, we need to learn a little bit about how the Spectrum's memory is organised.

The first 16K (kilobytes) of memory is the ROM (Read Only Memory). This contains the operating system (i.e. the BASIC language that launches when you switch the computer on plus other hidden subroutines that the computer needs). The data in the ROM can be looked at (peeked), but it cannot be changed (poked). The next 48K of memory is the RAM (Random Access Memory). You can read from and write to it freely. The contents of the ROM is preserved when you reset the computer (it never changes), whereas the contents of the RAM is deleted.

The entire memory, ROM & RAM, is presented to the programmer as one 64K lump (65536 bytes). You might think the number of bytes ought to be 64000, given that K (for kilo) indicates a multiple of 1000. But there are actually 1024 bytes in a kilobyte. The reason for that will make sense when you learn about binary. For now, just know that 65536 is 64*1024.

Each individual location in memory (called a byte) has a numerical address, organised sequentially. The ROM is addressed from 0 to 16383, the RAM from 16384 to 65535. To help bridge your understanding with BASIC, we can think of these addresses as the line numbers of an assembly language program. So, if you decided to start an assembly language program at "line" 10, you'd be attempting to overwrite a memory location in the ROM. And that's neither wise nor possible. But if you started it at "line" 30000, that would be fine.

Screen Memory

You might think the logical place to start your program would be 16384, but there are several kilobytes at the start of the RAM that are reserved for special purposes. Generally, it's good practice not to start any program prior to 24576.

Locations 16384 to 22527 are the screen memory. This means that any data placed into these addresses will automatically display itself on the screen, i.e. you don't have to insert the data then include a separate instruction to print it. Think of it as print on autopilot.

You're already familiar with peeking and poking into the RAM. Things get particularly interesting when you poke an address that corresponds to the screen memory. Try `POKE 16384, 255`. This places the value 255 into memory address 16384. You should see a short horizontal bar appearing at the top left of the screen, 8 pixels (1 byte) wide; 255 is 11111111 in binary. Now try `POKE 16385,85`. You should see a dotted line next to the bar. 16385 is the next address in sequence, therefore the next location on the screen, and 85 is 01010101 in binary. You can see that the 1's in binary are represented graphically as black pixels and the 0's are white.

That takes care of individual pixels. The Spectrum handles colour in blocks of 8x8 pixels, known as *attributes*. Each attribute contains two colours (ink & paper), a brightness setting (on/off), and a flash setting (on/off). All of that information can be encoded in a single number between 0 and 255.

The attribute data for the whole display is stored right after the pixel data, from 22528 to 23295. Poke any number within that range with a random value from 0 to 255 and see the result. Blocks of colour will appear in various spots.

The following BASIC program will print every possible colour from 0 to 255, to let you see how they are organised.

```
10 LET ADDR=22528
20 FOR I=0 TO 255
30 PRINT "*";
40 POKE ADDR+I,I
50 NEXT I
```

To calculate the numerical value of a desired colour, use the formula colour=ink+(paper*8). The basic colour palette is printed on the keyboard of the Spectrum above the number keys. From 0 to 7, the colours are black, blue, red, magenta, green, cyan, yellow, white. If I want, say, yellow ink on red paper, the calculation I need to perform is **6**+(**2***8)=22. Furthermore, to activate the brightness setting on any colour, add 64 to the result. To activate flash, add 128.

Spectrum games frequently feature loading screens. If you've used an actual Spectrum to load games with a cassette player, you will have seen loading screens undergoing construction on the TV, pixel by pixel. This happens because the data is loading directly into the screen memory from the tape. And screen memory always draws itself on autopilot, without user involvement.

Coding with an Assembler

So, let's do a little coding in Z80. In the old days, you had to use up part of the Spectrum's RAM for an assembler utility, which would allow you to type a program in assembly language and assemble it to a spot in memory. I'm making use of a Spectrum emulator on the PC called ZX Spin, which has an assembler built-in – an assembler which resides in the PC's memory, conveniently outside the virtual Spectrum. This gives me the advantage of being able to program any part of the 48K without interference or compromise. (If you are using a different method, I will assume you're on top of things.)

ZX Spin starts in 48K mode by default, with the familiar "© 1982 Sinclair Research Ltd" text at the bottom of the screen. Click the **Tools** menu and select **Z80 Assembler**. A separate text editor window will appear. This is where you will type in your Z80 code. Try this short example:

```
ORG 30000
LD HL, 22528
LD (HL), 40
RET
```

Running a Z80 Program

At the top of the text editor, click on **File**, then **Assemble**. This copies your code into the virtual Spectrum's memory. Pay attention to the panel at the bottom of the text editor, as it will inform you if there are any errors in your program.

Now head over to the main Spectrum window and type `RANDOMIZE USR 30000` into the BASIC interpreter. This is like the BASIC instruction `RUN`, but instead of executing a BASIC program, it will execute whatever machine code is stored at address 30000 and above.

Don't concern yourself with what each line of code means at this point. Suffice it to say, your program is essentially the Z80 equivalent of `POKE 22528,40`. So you should see the top-left colour attribute on the screen light up as a block of cyan.

Choosing a Program's Location in Memory

Back to the topic of line numbers, all Z80 programs must begin with a statement indicating where the code should be placed in the Spectrum's memory. That's ORG (shorthand for origin). So here's how the program would reside in memory:

```
        ORG 30000
30000   LD HL, 22528
30003   LD (HL), 40
30005   RET
```

I've italicised the line numbers to make it clear that they're not something you should type, as a feature of your program. They already exist, but out of sight. Notice that `ORG 30000` doesn't have its own line number. That's because it would be wasteful putting an instruction in location 30000 that tells the computer to put the instruction in location 30000 – since you're already there. ORG is what's called an *assembler directive* – a piece of code that the assembler makes use of to assemble the program in memory, but which isn't needed in the resulting program. (The program in chapter 1 didn't need an ORG statement, because we were poking the data manually into memory without relying on an assembler to do it for us.)

How a Program is Stored

The numbering of the lines is specific and no deviation is allowed, because these numbers are actual memory addresses. Unlike BASIC, where the 10 that is typed by hand at the beginning of line 10 is a piece of data that occupies a position in memory, the 30000 at the beginning of this program isn't data; it's the actual address where the first byte of the assembled program is stored in memory.

The instruction `LD HL, 22528` takes three bytes of storage space. When you're dealing with a number in the range 0 to 255, it can be stored in a single byte, but larger numbers than this require 2 bytes. They are called 16-bit numbers and they are in the range 0 and 65535.

Byte 1 of the instruction under review is `LD HL,` *nn*. Bytes 2 and 3 store the number 22528. So the whole instruction takes a total of 3 bytes. The next instruction must therefore begin at address 30003.

`LD (HL),` *n* takes 1 byte. And the value 40 takes 1 byte. This line therefore uses 2 bytes. So the next instruction begins at address 30005.

You can test this in BASIC. If what I'm saying is true, then you know that the value 40 ought to be residing in memory location 30004. So, in BASIC, type `PRINT PEEK 30004` to prove me right.

Interestingly, look what happens when you try `PRINT PEEK 30000`. You get the result 33. This means that the instruction `LD HL,` *nn* is instruction number 33. Similarly, `LD (HL),` *n* is instruction 54. And `RET`, as we learned in chapter 1, is 201. All Z80 instructions can be laid out in a table, from 0 to 255. If you wanted to, you could write entire machine code programs without an assembler, using only BASIC's POKE and a lookup table showing the numbers of all the Z80 instructions. It would be laborious, but entirely possible. Let's try it using our tiny example. Reset your Spectrum and type the following in BASIC.

```
POKE 30000, 33
POKE 30001, 0
POKE 30002, 88
POKE 30003, 54
POKE 30004, 40
POKE 30005, 201
```

You may be puzzled about why there are the numbers 0 and 88 where you would expect 22528. The method of converting a 16-bit number into two 8-bit numbers involves dividing the 16-bit number by 256. The result of the division is placed as the second byte of the operand, with the remainder positioned as the first byte – somewhat counterintuitively. The formula is essentially **remainder**+(**result***256). In our example, that means **0**+(**88***256)=22528. Helpfully, the assembler performs this mathematics on your behalf, but if you're working without an assembler, as we are right now, it's up to the programmer to work out the individual byte values.

Once again, type `RANDOMIZE USR 30000` to run the program. The result should be identical to last time, because it's the same program, created using a different method.

The terms machine code (or machine language) and assembly language are often used interchangeably. They are almost the same thing because, for instance, 201 and `RET` are the same piece of data, merely represented in two ways. Assembly language is therefore a representation of machine code using mnemonics (abbreviated English terms).

Think of memory addresses as the machine code equivalent of line numbers, but remember that they are not versatile in the way that they are in BASIC. They are, however, much more efficient at optimising the available memory.

3: Opcodes, Operands & Registers

Opcodes & Operands

Let's break down each line of the program from chapter 2, so that we understand what's happening.

```
        ORG 30000
30000   LD HL, 22528
30003   LD (HL), 40
30005   RET
```

LD is shorthand for "load" and the comma signifies "with". The mnemonic `LD HL, nn` means "load HL with the number *nn*". It's very much like the BASIC instruction `LET HL=nn`.

In BASIC, you could write the same program this way:

```
10 LET HL=22528
20 POKE HL,40
30 STOP
```

It should be immediately obvious that the program could be shortened to:

```
10 POKE 22528,40
```

We don't even need `STOP`, because BASIC programs automatically return to the editor when they finish execution. This is not true of machine code programs. If we neglected to include `RET` (return) at the end, the Spectrum's processor would keep right on going, attempting to execute whatever data happened to reside in the memory at 30005 and above, as if that data contained instructions. Inevitably, the program would crash.

`LD (HL), n` means `POKE HL, n`. When you see brackets in assembly language, it means "the memory location indicated by", e.g. "load the memory location indicated by HL with *n*". That's not the same as "load the HL register with *n*". Without the brackets, HL is just a number, not an address. The difference is subtle, but vastly important – as vast as the difference between LET and POKE in BASIC.

You might be wondering why we don't shorten our Z80 instruction to `LD (22528), 40`. Why use HL at all? To answer that, I need to explain that Z80 instructions consist of a maximum of two elements: the opcode and the operand. In the example `LD (HL), n`, the *n* is the operand and everything before it is the opcode. High-level programming might lead you to assume that LD is the opcode, HL is an operand, and n is another operand. Not so. There's no specific opcode called LD, but there are many opcodes that have an LD in them, such as:

`LD HL, nn`	opcode & 2-byte operand (total of 3 bytes)
`LD (HL), n`	opcode & 1-byte operand (total of 2 bytes)
`LD A, n`	opcode & 1-byte operand (total of 2 bytes)
`LD (nn), A`	opcode & 2-byte operand (total of 3 bytes)
`LD B, A`	opcode & no operand (total of 1 byte)

These are just some of the many variations of LD, each of which has its own unique opcode number. Notice that none of the above instructions takes more than 3 bytes of storage space, and some only take a single byte. Think how inefficient this would be if LD took a byte just for itself. The language structure "opcode followed by operand" allows for lots of detail to be packed into a small space. This convention is why something as complex as `LD (22528), 40` is not possible. That would be a case of an instruction containing an opcode plus two distinct operands. You will never construct something as complex as, say, `LET A=B+10` in a single line of Z80 code, only as a calculation worked out over several lines.

Since we can't do `LD (22528), 40`, we write it in two lines, using a "variable" as a go-between:

```
LD HL, 22528
LD (HL), 40
```

Registers & Register Pairs

I'm temporarily calling HL a variable, because that's what it resembles in BASIC, but you couldn't write something like `LD X, 22528`. Z80 features a limited set of "variables" called registers. These are stored outside of the main memory, in the processor itself. They are labelled A, B, C, D, E, H, L. Each of these can hold a single byte of data (an 8-bit number). Some of the registers can be combined (as you may have noticed we've already done), to hold a 16-bit number. These combinations are BC, DE and HL. The basic rule of thumb is, if the register you're using has two letters, it contains a 16-bit number, otherwise it's an 8-bit number. Even if you load HL with 0, the number is still two bytes long: zero followed by zero. And the register combinations are strict; you can't, for instance, combine C and E as CE.

Registers as Short-term Variables

If you've done a bit BASIC programming, you may be wondering how on earth you can write a large, complex program if you're restricted to 7 variables. The answer is, you're not restricted. Registers aren't entirely used like variables. To use an analogy, if you were working out your personal finances on paper, you would never commit every calculation to your brain's long-term memory, because that would have no value. What you're really interested in committing to memory is the final bank balance. In Z80, you would use registers for the calculations, but the final balance would be stored in a different way (which we'll come to later). In human terms, registers are like our short-term memory.

4: Jumps & Loops

Increment & Decrement

Now that we've successfully written and understood a Z80 program (albeit a tiny one), let's expand it and work towards the goal of filling the entire screen with a chequerboard pattern that is cyan and blue.

The first step is to move our focus to the next colour attribute address in sequence: 22529. We could use `LD HL, 22529`, but that's not going to help us in the long-run because we have 32 blocks across the screen and 24 down (768 in total) to cover. We need a way of increasing the value of HL in the manner `LET HL=HL+1`. Well, it turns out there's a useful opcode that adds 1 to the HL register: `INC HL` (short for increment). We can insert this into our program like this:

```
        ORG 30000
30000   LD HL, 22528
30003   LD (HL), 40
30005   INC HL
30006   LD (HL), 8
30008   INC HL
30009   RET
```

It's also worth noting that INC has its opposite. The opcode `DEC HL` (decrement) subtracts 1 from HL.

Test your code before proceeding further. You should see a cyan block followed by a blue block (colours 40 and 8).

Jumps

Now we need some sort of loop to keep this sequence repeating across the screen. The simplest method is to use the Z80 equivalent of GO TO, which is JP (jump). Between 30008 and 30009, insert `JP 30003`. This will naturally push `RET` forward 3 bytes in memory to 30012.

When you run the program, you'll see that it's not a very elegant solution, because the loop never terminates. Unlike BASIC, you can't press the break key to interrupt the execution. The Spectrum has essentially crashed and must be rebooted. You can do this from the pull-down menus in ZX Spin. (Don't make the mistake of closing and restarting ZX Spin itself, or you will lose your assembly language script.)

There's another jump instruction called JR (jump relative), which will save you 1 byte of memory, because it uses only a single byte as its operand. Although you would type `JR 30003`, the assembler will convert this to a value in the range -127 to 128. A value like -30 would mean jump backwards in memory 30 bytes. 50 would mean jump forwards 50 bytes. The range -127 to 128 can be stored as a single byte, because it doesn't exceed 256 possible values. Normally, the range is viewed as 0 to 255, but this is a special circumstance in which the computer treats the number as -128 to 127 (using a convention known as 2's Complement). JR is useful for short-range jumps only. The

maximum forward jump is 128 bytes; the maximum backward jump is 127. Helpfully, the programmer doesn't need to concern himself with the mathematics, as the assembler does the work. The operand of JR should be written as the desired memory address, not the number of bytes to jump. It's the assembler, not the programmer, that converts your 16-bit number into the necessary 8-bit number that facilitates the distance and direction of the jump.

Address Labels

The assembler makes jumping even easier with its ability to attach a text label to a memory address. We will attach the label LOOP to address 30003. Labels can be called anything (without spaces), as long as you avoid using actual Z80 mnemonics, which would only confuse the assembler.

It's helpful to construct your program across two columns, with column 1 containing any memory address labels you've invented and column 2 containing the program itself – much like I've presented it in the example below. Use the tab key to create columns, like this:

```
                ORG 30000
30000           LD HL, 22528
30003 LOOP      LD (HL), 40
30005           INC HL
30006           LD (HL), 8
30008           INC HL
30009           JP LOOP
30011           RET
```

It can be initially difficult to understand that we're not creating a variable called LOOP. After all, in BASIC, if you saw `GO TO LOOP` in a program, that would only make sense if LOOP were a variable that had been previously set to a numerical value, so that GO TO knew where to jump. It's a lot simpler than that in assembly language. LOOP means the same thing as 30003, by virtue of its position. If I happened to place it two lines lower, it would mean 30006. Or, if I changed the first line of the program to `ORG 40000`, LOOP would automatically mean 40003. The ability to represent memory addresses as text labels is a feature of the assembler that removes the tiresome task of working out memory addresses. When you assemble the program, LOOP isn't stored anywhere in it, because loop simply *is* address 30003. And that means "line" 30009 is assembled as if it read `JP 30003`, regardless of the fact that we wrote it as `JP LOOP`. This lifts a great burden from the programmer.

From this point on, I will no longer be showing italicised memory addresses as pseudo line numbers, as they are no longer needed, thanks to text labels. Ironically, I taught you about line numbers so that I could get rid of line numbers. But the lesson was important. This two-column approach to assembly language had me bewildered in the 1980s. I hope I've helped you make sense of it.

Countdown Loops

Now that we've got that out of the way, let's concentrate on terminating this endless loop. We'll work on completing the top row of the display alone, as a first step. The screen is 32 character blocks wide. Our program, in its current state, takes care of 2 blocks, so we need to repeat this 16

times to fill an entire row. In BASIC, when you need to repeat something a specific number of times, you would use FOR and NEXT, like this:

```
10 LET HL=22528
20 FOR I=1 TO 16
30 POKE HL, 40
40 LET HL=HL+1
50 POKE HL, 8
60 LET HL=HL+1
70 NEXT I
```

Assembly language can do something similar to a FOR loop using the instruction DJNZ, which is short for "decrement B, jump if not zero". Quite a mouthful. Here's how it works. We start by using the B register in the same manner that we're using the I variable in the BASIC example. We set it initially to 16: `LD B, 16`. Then we mark the next line with the address label LOOP and insert the block of code that we want repeated. Finally, we mark the end of the section we want to loop with `DJNZ LOOP`.

The program should look like this:

```
        ORG 30000
        LD HL, 22528
        LD B, 16
LOOP    LD (HL), 40
        INC HL
        LD (HL), 8
        INC HL
        DJNZ LOOP
        RET
```

The logic is like BASIC's FOR instruction, except it works in the form of a countdown instead of an upward count. B starts off as 12 on the first pass. When the execution of the program reaches DJNZ, the processor subtracts 1 from B, and asks, "Is B zero?" If no, it performs a relative jump to the address indicated by LOOP, otherwise processing continues with the next line (i.e. the loop terminates).

As a high-level language programmer, it can be a little jarring to come across a Z80 instruction that does so much using a single opcode. In BASIC, this logic would have to be spelled out explicitly in the form `LET B=B-1: IF B<>0 THEN GO TO` *nn*. But in Z80, all you need is `DJNZ` *nn*. Some Z80 opcodes do little; others do a lot. If a question on your mind is "How do I get DJNZ to work with a different register, because I might be using B for another task?" that's the wrong question. You're thinking too much like a high-level programmer, where you can use any number of variables in any number of contexts. DJNZ only works with B, because that's how it has been designed to work. So the right question would be, "How do I free up B, so that I can use it with DJNZ?" And we'll get to that soon. Assemble your program to see it in action before continuing.

This next bit is easy. We need to draw another line of attribute blocks, but we have to reverse the order of the colours, otherwise we'll end up with vertical stripes down the screen instead of a chequerboard. I suggest we simply repeat the relevant section of code, line for line, swapping the 40 and the 8 on the second version. There are undoubtedly more efficient ways to code this, but we're going to learn to crawl before we walk. We also need to rename LOOP, because you can't

logically have two different addresses with the same label, otherwise the processor won't know where to jump. I suggest LOOP1 and LOOP2.

Make these adjustments on your own and test your code. If it produced an unexpected result, the most common oversight here is the failure to reset B back to 16 at the beginning of the second loop.

Nested Loops, Push & Pop

We're almost there. The screen measures 24 characters from top to bottom. We've done two rows already, so we need to repeat everything 12 times to fill the screen exactly. In BASIC, you would surround your two existing FOR loops with another one. It's no problem having a loop within a loop, as long as you close them off properly with all the NEXTs in the right places. Here's a fully working BASIC version, commented with REM statements for clarity.

```
 10 LET ADDR=22528
 20 REM Outer loop
 30 FOR Y=1 TO 12
 40 REM Inner loop 1
 50 FOR X=1 TO 16
 60 POKE ADDR, 40
 70 LET ADDR=ADDR+1
 80 POKE ADDR, 8
 90 LET ADDR=ADDR+1
100 NEXT X
110 REM Inner loop 2
120 FOR X=1 TO 16
130 POKE ADDR, 8
140 LET ADDR=ADDR+1
150 POKE ADDR, 40
160 LET ADDR=ADDR+1
170 NEXT X
180 NEXT Y
```

Notice that line 50 uses X as the FOR counter. It uses it again in line 120. This is fine because the first loop is closed at line 100, so the two uses of X are entirely separate and never interfere with each other. This is not true of the FOR loop beginning at line 30, which doesn't close until line 180. We couldn't possibly use X here, so I opted for Y instead. This presents a huge problem in assembly language, since we're forced to use only the B register as the counter. There is, of course, a solution. It's called pushing and popping, and it's not something you'll have encountered before in BASIC.

Since registers can only be used for short-term storage, it's important to be able to move the data in them around easily. The Z80 processor features a facility called the *stack*, upon which you can place any 16-bit numbers, with a view to retrieving them later. Unfortunately, that means you can't put B alone on the stack, because it contains an 8-bit number, but we can put the register pair BC on the stack, and there's no reason why we shouldn't do so. The data in C is of no interest to us in solving the current problem, but it does no harm to let it sit alongside B in the stack. `PUSH BC` will put the contents of BC onto the stack. Later, when we're ready to retrieve the data, we'll use `POP BC`. The beauty about the stack is that it can contain multiple items. After you've pushed BC, you can go ahead and `PUSH DE` or `PUSH HL`, as long as you pop the data back in the same order you pushed. Retrieving data from the stack is always in a last-in, first-out fashion, just like a stack

of books; you take from the top. As a rule, never push unless you intend to pop one hundred percent of the time. Failure to keep the stack in order will result in a very wonky program. (On a side-note, you may wish to push and pop the data in A, but A is the only register not paired with another. It will work if you use `PUSH AF` and `POP AF`. F is a special register that we'll talk about later.)

In our program, we need to encase LOOP1 and LOOP2 in a wider loop that we'll call LOOP0, which will repeat 12 times. Once we load B with 12 and define the starting point of the loop with a label, we need to immediately push B onto the stack and forget about it until we need it again (which is after LOOP1 and LOOP2 complete their circuits). At that point, pop the data waiting in the stack back into B and complete LOOP0.

Here's the completed program, spaced out for clarity:

```
          ORG 30000
          LD HL, 22528

          LD B, 12
LOOP0     PUSH BC

          LD B, 16
LOOP1     LD (HL), 40
          INC HL
          LD (HL), 8
          INC HL
          DJNZ LOOP1

          LD B, 16
LOOP2     LD (HL), 8
          INC HL
          LD (HL), 40
          INC HL
          DJNZ LOOP2

          POP BC
          DJNZ LOOP0

          RET
```

This time, when you execute the program, use `RANDOMIZE USR 30000: PAUSE 0`. The latter statement will wait for a pause before returning you to the BASIC editor. It's not terribly important, but it prevents the editor from overwriting the bottom two lines of the screen until you're finished looking at the results of your work.

If there's any value in leaving BASIC behind and embracing assembly language, it's the phenomenal difference in processing speed. Compare the two versions of the program and be amazed.

5: Printing Text & Calling Subroutines

We've played with colour; now let's play with text. Try this little program:

```
ORG 30000
LD A, 33
RST 16
RET
```

It prints an exclamation point on the screen. As before, remember to execute this using `RANDOMIZE USR 30000: PAUSE 0`, so that the resulting display won't be overwritten.

We can force the print position to the top-left of the screen by adding two lines to the beginning of the program, right after the `ORG 30000` directive.

```
LD A, 2
CALL 5633
```

In the Spectrum's ROM, at address 5633, is a subroutine that sets the output channel for printing. The default setting is 1, the lower screen (i.e. the area where the user types in BASIC). When it's set to 2, all subsequent printing will be sent to the upper screen, starting at the top left. If it's set to 3, all printing will be sent to the printer (assuming one is attached). The opcode CALL works exactly like the BASIC instruction GO SUB. The subroutine at 5633 is designed to make use of the A register to set the channel, so you need to preload A appropriately before making the call. The subroutine we're calling contains a RET (return), which will return the execution of the program to the address from which the call was made and continue processing from there. RET doesn't just return to BASIC. It returns from the last call made. In fact, when it returns to BASIC, it's really returning from the original call that was `RANDOMIZE USR 30000`.

Assemble the program again. The exclamation point should now appear at the top-left of the screen.

A is an 8-bit register, also called the accumulator because it's primary function is doing sums. But here we're using it just to hold the value 33. All letters, numbers and symbols on the Spectrum are arranged according to a numerical convention known as ASCII (the American Standard Code for Information Interchange). Values 0 to 31 are reserved for special purposes. 32 is a space, 33 an exclamation mark. The number digits 0 to 9 are positioned at 58 to 57, A to Z in upper-case from 65 to 90, A to Z in lower-case from 97 to 122, etc. The following BASIC program will show you a complete list:

```
10 FOR I=32 TO 255
20 PRINT CHR$ I,I
30 NEXT I
```

To be precise, the Spectrum's version of ASCII is slightly modified, as the original (being American) doesn't contain a pound sign. The highest value in ASCII is 127, but the Spectrum makes use of values higher than this for special characters such as graphical blocks, UDGs (user-defined graphics), and BASIC keywords.

Helpfully, the assembler will also allow you to write line 2 as `LD A, "!"`, so you don't have to memorise the entire ASCII table. But don't fall into the trap of thinking that you're assigning a value to a string variable, as you would in BASIC. What the assembler really sees is 33, regardless of how you represent it. In BASIC, the idea doing an equation such as "!"+"A" is preposterous, because you can't perform sums on letters and symbols. But the assembler won't quarrel with you, because what it really sees is 33+65. In assembly language, there's no difference between a single-character string and an integer (whole number). Everything is simply data, and all data is fundamentally numerical, even if you enclose it in quotation marks and type letters and symbols.

RST (short for restart) is like CALL, but it only works with a particular set of addresses, all of which are very close to the beginning of the Spectrum's ROM. In this example, address 16 calls a subroutine that prints the ASCII value of whatever happens to reside in the A register. It is up to the programmer to load that register appropriately before invoking `RST 16`.

The A register can only hold a single byte, which means it can only be used to print one character at a time. The simple task of printing the word "Hello" would involve a program like this:

```
ORG 30000
LD A, "H"
RST 16
LD A, "e"
RST 16
LD A, "l"
RST 16
LD A, "l"
RST 16
LD A, "o"
RST 16
RET
```

Imagine writing a paragraph! Thankfully, there's another way. Type in, assemble, and execute the following program:

```
        ORG 30000
        LD A, 2
        CALL 5633
        LD DE, GREET
        LD BC, 5
        CALL 8252
        RET
GREET   DEFB "Hello"
```

Line 8 looks suspiciously like a variable called GREET containing a string. And that's exactly how we're going to treat it. However, bear in mind, that this line could also have been written as:

```
GREET   DEFB 72, 101, 108, 108, 111
```

Those numbers are the ASCII values of "H", "e", "l", "l" and "o". GREET is just a memory address containing 72. It doesn't even contain the other numbers, because a single address can only hold a single byte. 101 ("e") is held in the address GREET+1, 108 ("l") is held in address GREET+2, and so on. You can refer to addresses in this manner in your own programs, if you ever need to.

DEFB (define byte) is an assembler directive that tells the assembler that what follows is not a Z80 instruction, but data. As a directive, it doesn't take up space in the assembled code.

In line 6, we're calling another subroutine in the ROM (at address 8252) – one that prints data. It expects the DE register to be preloaded with the address of the first character to be printed (an address labelled GREET), and the BC register to be preloaded with the total number of characters to print (5).

It's perhaps a little counterintuitive that we're defining a variable after the end of the program, when BASIC would insist upon defining everything at the beginning. Remember, these aren't truly variables. They're just locations in memory that can be used in a manner similar to variables. We're not saying `LET G$="Hello"`. We're storing the data "Hello" in a series of 5 memory locations, knowing that we can fetch that data when needed because we've labelled the address of the first character ("H") as GREET.

We're now going to make a small modification to our print routine that will take away the burden of having to manually count the length of the message we're printing. Take a new line at the bottom and type:

```
EOGREET EQU $
```

That's an address label EOGREET (my shorthand for "end of greeting"), followed by a new assembler directive. `EQU $` does absolutely nothing, but as an assembler directive it has the advantage of not taking up any space in the assembled program; not a single byte wasted. In our example, it allows us to label an empty space so that the computer can count how many bytes are between GREET and EOGREET (i.e. the length of the "string"). You can see that 5 bytes (one for each letter) are occupied between the addresses GREET and EOGREET. Therefore the equation EOGREET-GREET equates to 5. So, modify `LD BC, 5` to read `LD BC, EOGREET-GREET`. On the surface, this looks like something far too complex for a single Z80 instruction, but it's allowed because it's the assembler doing the work. The assembler will pick up on the fact that EOGREET-GREET is 5, so it will assemble it as 5. In precisely the same way, there's nothing wrong with writing an instruction like `LD A, 32+((3*10)/15)-5`. It's merely `LD A, 42` written in an insanely convoluted way, after all. This means that you can change "Hello" to "Hello, world!" and the program will execute just fine, because the assembler can immediately tell that line 5 is now `LD BC, 13` under the surface of the address label equation.

It can be frustrating to try and remember addresses in the ROM that you call frequently. The EQU (equates to) directive can be used to attach labels to address you use frequently. At the top of the program, beneath the origin statement, insert the line:

```
PRINT   EQU 8252
```

This means the label PRINT equates to 8252. Now you can type `CALL PRINT` instead of `CALL 8252` any time you want to program a print statement.

This is the complete print routine:

```
        ORG 30000
PRINT   EQU 8252
        LD A, 2
        CALL 5633
        LD DE, GREET
        LD BC, EOGREET-GREET
        CALL PRINT
        RET
GREET   DEFB "Hello, world!"
EOGREET EQU $
```

The PRINT instruction in BASIC is quite powerful. You can position your message on the screen at a particular spot using Y and X coordinates. You also can choose the desired ink and paper values, among other things. Here's an example:

```
PRINT AT 5,10; INK 6; PAPER 2; BRIGHT 1; FLASH 1; "Hello, world!"
```

The ROM's print routine can function just as powerfully when called using assembly language. We do this by making use of the special ASCII codes below 32.

AT is achieved using code 22 followed by the Y coordinate (vertical position), followed by the X coordinate (horizontal position). All we have to do is insert the numbers directly before the data we wish to print, like this:

```
GREET   DEFB 22, 5, 10, "Hello, world!"
```

It may look ugly, but it works. When the print routine encounters code 22, it anticipates two further numbers and it knows what to do with them – in our case, positioning the print position to row #5, column #10 (with 0, 0 being the top-left). Ink, paper, etc., are encoded the same way. Here's a list of useful codes:

13: Carriage return (takes a new line)
16: Ink (0-7)
17: Paper (0-7)
18: Flash (0/1, off or on)
19: Bright (0/1)
20: Inverse video (0/1)
21: Over (0/1)
22: At Y, X
23: Tab

It doesn't matter what order you stack the codes in, as long as logic is preserved. The Z80 equivalent of our BASIC example above is:

```
GREET   DEFB 22, 5, 10, 16, 6, 17, 2, 19, 1, 18, 1, "Hello, world!"
EOGREET EQU $
```

If that mass of numbers looks deeply confusing, look more carefully. See the "16, 6" in the middle? That simply means "set ink to yellow" (code 16 is ink, and value 6 is yellow).

We could also type the above data like this:

```
GREET   DEFB 16, 6, 17, 2, 18, 1, 19, 1
        DEFB 22, 5, 10
        DEFB "Hello, world!"
EOGREET EQU $
```

This flexibility is allowed because the assembled code isn't organised in lines, as it is on the screen for the programmer. It's just one byte after another.

6: Binary, Hexadecimal & UDGs

Binary to Decimal Conversion

Within the computer's architecture, beneath the surface of all these decimal numbers we've been peeking and poking, there are only 1's and 0's. That's because electricity flowing through circuits is either on or off. Humans count using ten digits (because we have ten fingers), whereas computers only have two. Insane as it sounds, two is actually enough.

It's useful to learn how to convert decimal numbers to binary, and vice versa. The fun way to figure this out is to create some graphics. Start with a piece of graph paper and mark out a box that is 8 squares by 8. Draw a graphic within this box, by colouring in some of the squares. Now redraw the graphic using 1's to represent the coloured squares and 0's to represent the uncoloured squares. Here's my effort:

```
00011000
00011000
11111111
10111101
10111101
00111100
00100100
01100110
```

You might already be able to tell that it's a rather crude looking little man. Each horizontal line is a single byte of data, and we'll work out the decimal value of each, one by one. We start by constructing a table with 8 columns. The headings at the top are the numbers, 1, 2, 4, 8, 16, 32, 64, 128 (that's really 2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7), written in reverse order. Underneath, we fill in the 1's and 0's of our graphic, starting with the top row.

```
128  64  32  16   8   4   2   1
-------------------------------
  0   0   0   1   1   0   0   0
```

To work out the decimal value, we ignore the 0's and add up the instances of 1's, not by counting the 1's themselves, but by counting the values in the column headings above the 1's. The above example is 16+8, which equals 24.

The complete graphic looks like this:

```
128  64  32  16   8   4   2   1
-------------------------------
  0   0   0   1   1   0   0   0 = 16+8                  = 24
  0   0   0   1   1   0   0   0 = 16+8                  = 24
  1   1   1   1   1   1   1   1 = 128+64+32+16+8+4+2+1 = 255
  1   0   1   1   1   1   0   1 = 128+32+16+8+4+1      = 189
  1   0   1   1   1   1   0   1 = 128+32+16+8+4+1      = 189
  0   0   1   1   1   1   0   0 = 32+16+8+4            = 60
  0   0   1   0   0   1   0   0 = 32+4                  = 36
  0   1   1   0   0   1   1   0 = 64+32+4+2            = 102
```

Look carefully at the third row of binary digits. If you've been wondering why the maximum value you can store in a byte is 255, you can now see that it's because the byte contains the maximum number of 1's, namely 8. This also explains why we call these 8-bit numbers; there are 8 bits (binary digits) in a byte. 16-bit numbers are two 8-bit numbers side-by-side. They can store values up to 65535 because the powers of two are extended from 2^7 to 2^{15}.

User-defined Graphics

This sequence of 8 bytes should be placed into a labelled memory address in the same way that we placed our "Hello" message (typically at the bottom of a program, so that it isn't mixed up with the instructions).

```
GFX     DEFB 24, 24, 255, 189, 189, 60, 36, 102
```

The ZX Spin assembler will also allow you to write the data in binary. If you precede a byte of data with the symbol %, the assembler treats it as binary.

```
GFX     DEFB %00011000, %00011000, %11111111, %10111101
        DEFB %10111101, %00111100, %00100100, %01100110
```

But I don't want you to take the lazy way out. It's important to understand the relationship between binary and decimal numbers.

Creating UDGs (user-defined graphics) in BASIC is a topic covered in the Spectrum 48K user manual. In assembly language, this is a rare instance where the procedure of converting 8 bytes of data into a graphic is simpler than it is in BASIC.

When you switch the Spectrum on, memory addresses 23675 and 23676 contain two 8-bit numbers which together form the 16-bit number: 65368. This number is the start address where the UDG characters are stored. Address 23675(&6) is what's known as a system variable. Notice that it resides in the early portion of RAM that I said was reserved for special purposes. There are other system variables that we'll make use of in future lessons. The system variable 23675(&6) contains a pointer to the default location of the UDGs in memory (65368).

We could poke our little man into locations 65368 to 65375, and that would work fine. But it's a lot easier to simply change where the system variable 23675(&6) is pointing. We can point it to the address labelled GFX. All we have to do is poke that value into 23675(&6).

First, don't forget to start your program with an origin statement:

```
ORG 30000
```

Begin by loading register HL with the address label GFX (my shorthand for "graphics"):

```
LD HL, GFX
```

Then load the memory address 23675 with HL:

```
LD (23675), HL
```

If you've been paying attention, you'll know that you can only load an 8-bit number into a memory location, so the above opcode might be a little confusing. You can put the H into 23675, or you can put the L into it, but how can you put HL into it? Well, the opcode `LD (nn), HL` automatically knows to use address nn and nn+1 to store H & L. In our example, 23675 and 23676 will both be overwritten with the data in HL.

Believe it or not, our work is pretty much done. Assemble and execute the following code, as a test. I've included comments in the listing, for clarity. The semi-colon is the assembler equivalent of BASIC's REM instruction. Anything after a semi-colon on the same line will be ignored during assembly. You obviously don't need to type in these comments, but when it comes time to write your own programs, I advise commenting heavily, otherwise you will be lost when you need to go back and make modifications.

```
          ORG 30000
          LD A, 2           ; set print channel to upper screen
          CALL 5633
          LD HL, GFX        ; set up UDGs
          LD (23675), HL
          LD A, 144         ; print first UDG
          RST 16
          RET               ; return to BASIC
GFX       DEFB 24, 24, 255, 189, 189, 60, 36, 102    ; UDG data
```

UDGs are accessed using characters 144 to 164 of the ASCII table. There are a maximum of 21. Making more than one is just a matter of continuing the data under GFX. The additional values you type will automatically be loaded into characters 145, 146, and so on.

Decimal to Binary Conversion

There's one final lesson before we finish with binary. I've showed you how to convert binary numbers to decimal, but what about converting decimal numbers to binary? The method is easy. Once again, we start with our table header, marked out in powers of 2.

```
128  64  32  16   8   4   2   1
-------------------------------
```

As an example, let's say I asked you to convert the number 92 to binary. We start at the left-hand side of the table and ask, "Does 128 fit inside 92?" The answer is no, so put a 0 under 128.

Then we move to the next column and ask, "Does 64 fit inside 92?" The answer is yes, so we put a 1 under 64. Now that we've used up the 64 slot, we subtract 64 from 92, leaving 28 remaining. Moving forward:

"Does 32 fit inside 28?" No, therefore 0.
"Does 16 fit inside 28?" Yes, therefore 1 (28-16 leaves 12).
"Does 8 fit inside 12?" Yes, therefore 1 (12-8 leaves 4).
"Does 4 fit inside 4?" Yes, therefore 1 (4-4 leaves 0).
"Does 2 fit inside 0?" No, therefore 0.
"Does 1 fit inside 0?" No, therefore 0.

So 92 is 01011100 in binary.

The Benefit of Hexadecimal

So far, we've dealt with numbers represented in decimal (base 10) and binary (base 2). Numbers can also be represented in hexadecimal (base 16). It's entirely optional whether you want to work with numbers in hexadecimal, but hexadecimal (hex for short) is useful because it's more closely related to binary than decimal (16 is a power of 2, whereas 10 isn't). There's nothing especially elegant about decimal. It's just what we're used to. We would all be counting in hex naturally if humans had evolved with 8 fingers on each hand.

In decimal, a single digit has a range of 10 possible values (0-9). In hexadecimal, there are 16. We use the letters A to F to symbolically represent the additional digits, i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (that's equivalent to 0 to 15 in decimal). All 8-bit numbers can be represented with 2 hex digits. Decimal value 255 (the maximum allowed) is FF. The minimum, 0, is typically written as 00. The ZX Spin assembler will treat any numbers preceded by a $ symbol as hex values. LD A, $FF is the same as LD A, 255. I will use the same convention in this book.

Some important addresses in the Spectrum's memory are a lot easier to memorise in hex than they are in decimal. 16384, the beginning of the RAM (and the beginning of the screen memory) is $4000. The top address in RAM (65535) is $FFFF. The total amount of colour attributes (768) is $0300.

Hex is also useful when you want to examine the contents of 16-bit register pairs individually. If you type `LD HL, 16384`, you're clueless about exactly what values go into H and L – until you get the calculator out. But 16-bit hex numbers are always 4 digits, neatly splittable down the centre. `LD HL, $4000` places $40 into H and $00 into L.

Converting binary to hex, and vice versa, is remarkably easier than dealing with decimal. The largest single digit of hex is F (15), which is 1111 in binary. So you can think of each grouping of 4 digits in binary as a single digit of hex. If you have a long 16-bit binary number that you want to convert to hex, just divide it all up into groupings of 4 digits and convert those slices individually. Here's an example: 1011011000111101.

```
  8 4 2 1    8 4 2 1    8 4 2 1    8 4 2 1
  -------    -------    -------    -------
  1 0 1 1    0 1 1 0    0 0 1 1    1 1 0 1

= 11 ($B)    6 ($6)     3 ($3)     13 ($D)

= $B63D
```

You'll probably never have occasion to convert a hex number to decimal, but in case you do, here's how it's done. Mark out columns in the same manner you did when converting binary to decimal in an earlier chapter, only use base 16 instead of base 2: 16^0, 16^1, 16^2, 16^3 (1, 16, 256, 4096) in reverse order:

```
   4096        256         16          1
   ---------------------------------------
     B          6           3          D

4096*11      256*6       16*3       1*13
  45056   +   1536    +    48    +    13   = 46653
```

With the hex number written underneath the heading, you can see that we multiply the numbers in the heading with the corresponding hex digits, one by one, then total the results into a single figure. B63D in hex is 46653 in decimal.

7: Decision-making & Bit-rotation

Spicing Up the System Font

Spectrum games frequently feature a replacement of the system font with one created by the programmer, much like programming UDGs. The system font, including upper-case letters, lower-case letters, number digits, and symbols, is a total of 96 characters. The graphical design of each character takes up 8 bytes of memory. That means a new font is going to take up 96*8 (768) bytes of space in the RAM. That would be quite a bit of typing for you, if I wanted to show you a font I had made. Instead, I'd like you to assemble and execute the following brief program. (This is a modified version of a routine originally written by Jonathan Cauldwell.)

```
        ORG 30000
        LD HL, 15616          ; address of system font
        LD DE, 60000          ; address of new font
        LD BC, 768            ; 96 chars (8 bytes per char)
LOOP    LD A, (HL)            ; get current byte of graphic
        RRA                   ; rotate all bits right by 1 bit
        OR (HL)               ; combine both images
        LD (DE), A            ; write new image to new font
        INC HL                ; move to next byte of system font
        INC DE                ; move to next byte of new font
        DEC BC                ; decrement counter
        LD A, B               ; but high byte of BC into A
        OR C                  ; compare B & C (using A).
        JR NZ, LOOP           ; repeat until BC=0
        LD HL, 60000-256      ; font minus 32*8
        LD (23606), HL        ; point to new font
        RET
```

When the program has finished executing, have a play around in the BASIC interpreter, paying attention to the Spectrum's font. Every character now looks different. How on earth did this tiny program accomplish all that? Time to explain.

The start address of the system font is 15616, where we find the graphic data for the ASCII character 32 (space) followed by the rest. Unfortunately, 15616 is in the ROM, so we can't overwrite this data directly. We can, however, read it and copy it to somewhere in the RAM. 60000 has been chosen as the target address. There are 96 characters to copy, each one containing 8 bytes of graphic data; that makes a total of 768 bytes.

Looping More Than 255 Times

The copy process is being done one byte at a time using a loop called, appropriately enough, LOOP. We can't use DJNZ this time, because that only works with values up to 255 stored in the B register, whereas we need 768 iterations. We're going to have to resort to 16-bit registers to accomplish this.

First of all, BC is loaded with 768. The starting point of the loop is then defined as LOOP. For the time being, forget about the next few lines, where everything we want to do with the current

byte is being processed. Concentrate on the tail end of the loop, where the processor has to make a decision about whether to jump back to the beginning of the loop or continue. An ingenious combination of instructions is employed here to ensure that the program counter jumps back to LOOP precisely 768 times:

```
DEC BC
LD A, B
OR C
JR NZ, LOOP
```

1 is decremented manually from BC, because we're not using DJNZ (which decrements from B as part of its internal logic). You already know what `LD A, B` does, but `OR C` will take some explaining.

And, Or & Xor

In BASIC, the terms OR and AND are used in the following manner:

```
IF LIVES=0 OR SCORE>10000 THEN GO TO 1000
IF X=5 AND Y=5 THEN GO TO 2000
```

The logic of the above is easily understandable. But what on earth does `OR C` mean, especially when there's only one piece of data under scrutiny?

Implicit in every OR instruction is the A register. `OR C` makes a comparison between C and A in binary, using OR logic. It will make sense when we lay it out in the fashion of a table. Let's use an arbitrary example of having preloaded A with 85 and C with 104.

```
A =  85 = 0 1 0 1 0 1 0 1
C = 104 = 0 1 1 0 1 0 0 0
          ---------------
OR C      0 1 1 1 1 1 0 1 = 125
```

Each bit/column is calculated individually. The question that the processor asks is, "If the first bit in A *or* the first bit in C is 1, then the first bit in the result is 1, otherwise 0." You can see that 0 applies here. Then the processor asks, "If the second bit in A or the second bit in C is 1, then the second bit in the result is 1, otherwise 0." And so on, until the byte is completed. The A register is then overwritten with the result.

`OR B` does the same thing, except it uses the A and B registers. `OR n` uses the A register and an 8-bit number of your choosing. There are many opcode combinations built around OR.

And there are just as many for AND. The logic of AND differs marginally from OR. While OR will mark the result as 1 if either of the two bits are 1, AND will only mark the result as 1 when both bits are 1. See how the result in our example differs:

```
A =  85 = 0 1 0 1 0 1 0 1
C = 104 = 0 1 1 0 1 0 0 0
          ---------------
AND C     0 1 0 0 0 0 0 0 = 64
```

A third related instruction is XOR (exclusive OR). This one will only mark the result as 1 when precisely 1 of the two bits is 1, never both:

```
A =  85 = 0 1 0 1 0 1 0 1
C = 104 = 0 1 1 0 1 0 0 0
          ---------------
XOR C     0 0 1 1 1 1 0 1 = 61
```

One of the most common uses of XOR is as a trick method of setting the A register to 0. You could of course use `LD A, 0`, which would take up two bytes of memory. But `XOR A` has no operand and always results in 0, no matter what value was previously in A:

```
A =  85 = 0 1 0 1 0 1 0 1
A =  85 = 0 1 0 1 0 1 0 1
          ---------------
XOR A     0 0 0 0 0 0 0 0 = 0
```

The intention of our program is to compare B with C using OR logic, but we can't do that directly, because `OR C` implicitly works with A. We get around that by loading B into A first, so that A contains the same data as B when executing `OR C`.

The reason why we're doing this at all reveals itself when we ask the question, "When will `OR C` result in 0?" The answer is: "Only when B is 0 and C is 0." In other words, "Only when BC has reached 0." And when it that? After 768 passes through LOOP.

Jumping (and the Zero Flag)

Lastly, JR NZ means "jump relative if not zero". If what isn't zero? When any calculation is performed, the Z80 makes use of a feature called flags. Flags are stored in the unused F register. There are several of them, and they each do different tasks, but the only one we're concerned with at the moment is the zero flag. Each flag can be in one of only two states: 0 or 1. When a calculation is performed using registers and the result turns out to be 0, the zero flag is automatically set (1). When a calculations results in anything other than 0, the zero flag is reset (0).

JR NZ consults the present state of the zero flag. If it's 1, the program jumps to the address specified. If it's 0, the program continues to the next instruction. (By the way, since JR NZ means "jump relative if not zero", can you guess what JR Z does?)

Bit-rotation

In the BASIC editor, look carefully at the font. It's the same one as before, but it's bold. This was the letter A:

originally	*now*
00000000	00000000
00111100	00111110
01000010	01100011
01000010	01100011
01111110	01111111
01000010	01100011
01000010	01100011
00000000	00000000

The bold effect was achieved by taking the original graphic, shifting all the 1's and 0's right by one bit, then superimposing the two images on top of each other. This is the code that accomplishes it:

```
LD A, (HL)          ; get current byte of graphic
RRA                 ; rotate all bits right by 1 bit
OR (HL)             ; combine both images
LD (DE), A          ; write new image to new font
```

`RRA` (rotate right A) moves all the bits that are in the A register to the right by 1 (any 1's that fall off the edge are forgotten): 00111100 in the second line of the graphic becomes 00011110. (Can you guess what `RLA` does?)

`OR (HL)` compares the byte stored in the address pointed to by HL with the byte stored in A, using OR logic. A bold effect occurs. When 00111100 is OR'd with 00011110, the result is 00111110.

Lastly, after the loop has completed its 768 iterations and all the data has been copied, we tweak a system variable, like we did with the UDGs. 23606(&7) contains the address where the system fetches the graphics for the font. We want to point that system variable to 60000, where our new font is stored. Actually, we want to point it to 60000-256. We started our font at ASCII value 32 (the space), but the system variable points to value 0 on the ASCII table. So we need to rewind 32*8 (256) bytes from the start position (remembering that each character takes up 8 bytes graphically).

```
LD HL, 60000-256    ; font minus 32*8
LD (23606), HL      ; point to new font
```

8: Arithmetic & Long-term Variables

8-bit Addition

We won't get far in writing a program without tackling some rudimentary mathematics. So far, we've met INC (increment) and DEC (decrement). Let's now look at ADD and SUB.

Adding two numbers together is done using the ADD instruction. For 8-bit numbers, adding is always done using the A register (the accumulator), e.g. `ADD A, 10` adds 10 to the existing value of A and overwrites A with the new result. It's equivalent to `LET A=A+10` in BASIC. You can't, however, perform `ADD B, 10`. If you want to ADD 10 to B, you have to first load B into A, add 10, then load the result back into B. The following example shows how you would add 10 to B, after setting B initially to 30:

```
LD B, 30
LD A, B
ADD A, 10
LD B, A
```

16-bit Addition

For larger 16-bit numbers, the HL register must be used for addition. There is an additional restriction for 16-bit calculations: you can't add numbers directly, e.g. `LD HL, 1000` is not permissible. You can only do addition using the contents of other 16-bit registers.

```
LD HL, 3000
LD BC, 1000
ADD HL, BC
```

This results in 4,000 stored in HL.

8-bit Subtraction

8-bit subtraction is done using the SUB instruction. The following example sets A to 30 and subtracts 10, leaving A as 20.

```
LD A, 30
SUB 10
```

It's little jarring that the bottom instruction isn't written as `SUB A, 10`. A is omitted explicitly (but is always present implicitly), because there is no subtraction opcode for 16-bit numbers using HL; subtraction can only be done using A. If you need to perform subtraction on 16-bit numbers, you must to create a longer workaround.

16-bit Subtraction

Although there is no SUB opcode for HL, `DEC HL` is permitted, to subtract 1 from HL. If, for example, you wanted to subtract an 8-bit number, such as 100, from the 16-bit register HL, you could do it in a long-winded way using DEC and a loop with 100 iterations:

```
        LD HL, 3000
        LD B, 100
LOOP    DEC HL
        DJNZ LOOP
```

Alternatively, there is a more efficient method, allowing for 16-bit subtraction from a 16-bit register:

```
OR A
LD HL, 3000
LD BC, 100
SBC HL, BC
```

Line 1, comparing A with A using OR logic, leaves A unchanged, but has the effect of resetting the carry flag, which is necessary for successfully executing SBC (subtract with carry) in line 4. More on the carry flag later in the chapter; suffice it to say, if the carry flag had been accidentally set beforehand, subtracting BC from HL would not give the correct answer. Naturally, the result of the calculation is stored in HL.

Combining 8-bit and 16-bit Addition

Another hurdle you might encounter in your programming is that you want to add an 8-bit number (stored in A) to the contents of a 16-bit register (HL). Simply move the 8-bit number to a free 16-bit register (BC or DE) first, before performing the addition:

```
LD HL, 3000
LD A, 100
LD C, A
LD B, 0
ADD HL, BC
```

In the above example, take great care that you copy A into C and not B (or E and not D). HL consists of a high byte and a low byte, in the order H followed by L (hence its name). The same ordering applies to BC and DE.

The low byte is like the units column of a decimal number, only in this context units means any number between 0 and 255 (instead of 0 and 9). The high byte is like the 10's column of a decimal number, only in this context it's the 256's column. Confusingly, these are placed in memory in the order: opcode followed by low byte of operand followed by high byte of operand. It's a bit like representing the decimal value 25 as 52, but still thinking of it as 25. So, if 200 were placed into B (the high byte), with C set to 0, these values will be positioned in memory as 0 followed by 200, and the actual 16-number number in BC will equate to **0**+(**200***256)=51200. If we store the numbers the other way around, setting B to 0 and C to 200, then BC equates to **200**+(**0***256)=200, which is what we were aiming for.

If you're confused, note that you really do use the same formula in your brain when working out decimal numbers, only you do it subconsciously. The number 58 is really 8+(5*10). The 10 in the formula corresponds to the 10 fingers on your hands, which is the reason humans calculate in base 10 (decimal) in the first place. It might help to imagine the Z80 microprocessor as a creature with 256 fingers!

Long-term Variable Storage

We can't store numbers in registers indefinitely, or we'll quickly run out of "variables" to play with. Numbers can be moved in and out of memory locations with ease using the opcode `LD A, (nn)`. The brackets convention indicates we're referring to an address containing data rather than a piece of data itself. The following program provides a practical example that you can test:

```
          ORG 30000
          LD A, (NUM1)
          LD B, A
          LD A, (NUM2)
          ADD A, B
          LD (RESULT), A
          RET
NUM1      DEFB 20
NUM2      DEFB 5
RESULT    DEFB 0
```

Can you tell what it does? It examines the data stored in the memory addresses labelled NUM1 and NUM2, adds the two numbers together, and stores the result in the memory location labelled RESULT. Earlier, I referred to registers as short-term memory. Well, the data in memory addresses is the long-term memory, the proper machine code equivalent of variables. This data differs from variables only in regard to the processor's inability to perform calculations on them directly. There's no `LET RESULT=NUM1+NUM2`. You have to move the data into suitable registers, perform your calculations, then move the data back again.

The above program does nothing on-screen, but I can demonstrate that it's working in BASIC because I can manually count up the bytes occupied by the program listing and work out that that the address behind the label RESULT is 30014.

After assembling the code and executing it with `RANDOMIZE USR 30000`, type `PRINT PEEK 30014`. The Spectrum should print the number 25 (20+5). In your assembler, try changing the values in NUM1 and NUM2, reassemble the program, and execute it again. This time, when you peek into 30014, you should get a different result.

8-bit Multiplication

There are no opcodes for multiplication or division, but it is possible to write your own routine to accomplish these ends. This can be a little complicated, when the numbers you want to deal with are large, but I can often get by with the following little multiplication routine, which works fine as long as the numbers (and the desired result) are in the range 0 to 255. The following example calculates out 5*3 and stores the result in A:

```
        LD A, 0
        LD B, 3
LOOP    ADD A, 5
        DJNZ LOOP
```

8-bit Division (and the Carry Flag)

Here's a simple routine for division. The example works out 15/6 and stores the result in A (the remainder is ignored):

```
        LD A, 15
        LD B, 6
        LD D, 0
LOOP    INC D
        SUB B
        JR Z, FINISH
        JR NC, LOOP
        DEC D
FINISH  LD A, D
```

JR NC, in line 7 above, is new to you. The C in NC should not be confused with register C. In this context, it is shorthand for the carry flag, just like Z is an abbreviation for the zero flag. C is normally 0, but is automatically set to 1 when a calculation results in a value outside the bounds of 0 to 255. C indicates that something has been lost. JR NC means "jump (relative) if carry flag not set".

Using our test data, on the first pass through LOOP, 6 is subtracted from 15, leaving 11. On the second pass, 6 is subtracted from 11, leaving 5. On the third pass, 6 is subtracted from 5, leaving -1, which the register can't handle. This automatically triggers the setting of the carry flag, which terminates the loop. The answer to the equation is the total passes through LOOP minus 1. The result is 2. If other test data happens to result in a clean answer without any remainder (which won't trigger the carry flag), it *will* trigger the zero flag, and the answer to the equation becomes the total passes.

There's a lot to remember here, but it all becomes second nature with a bit of practice.

9: Player Input & Sprite Movement

Detecting a Key-press

Address 23560 is system variable that contains the ASCII value of the last key pressed on the keyboard. Address 23560 works behind the scenes, always keeping track of key-presses, regardless of what you are doing. In other words, if you press A in the BASIC editor right now, address 23560 will contain 97 (the ASCII value of a lower-case A). Nothing will happen visually to tell you that's happening, but trust me it's happening. Type `PRINT PEEK 23560` in BASIC. 13 will appear on screen, because the last key you tapped was enter, and 13 is the ASCII code for enter (carriage return).

In BASIC, detecting a key-press is achieved with an instruction like `IF INKEY$="a" THEN GOTO nn`. In assembly language, we use the same logic, but it looks like: "If address 23560 contains 97 ("a"), then jump to …"

If … Then …

IF is the cornerstone of high-level language programming. Without it, there's practically nothing useful you can do, because it's the facility that sends the program execution down one path or another. No decision-making is possible without it. Assembly language was initially very confusing for me, because it didn't seem to feature an IF instruction. Well, it's there all right, but it's craftily hidden. In Z80, the majority of decision-making takes place by examining flags: C (carry) and Z (zero). JR Z, which you've encountered, means "if the zero flag is set, then jump to …"

Another powerful IF-type instruction that uses the zero flag is CP (compare). It behaves like SUB, only it doesn't store the result anywhere. It does, however, adjust the zero flag according to the result. These four examples will illustrate this:

`LD A, 20` `SUB 10`	`LD A, 20` `CP 10`	`LD A, 20` `SUB 20`	`LD A, 20` `CP 20`
A becomes 10 Z is reset (0)	A remains 20 Z is reset (0)	A becomes 0 Z is set (1)	A remains 20 Z is set (1)

What follows is the assembly language equivalent of `IF INKEY$="a" THEN GOTO nn`:

```
LD (23560), A      ; load last key-press into A
CP 97              ; compare A register with 97 ("a")
JR Z, WHOOP        ; if Z is set (if A is identical to "a"), jump
```

Obviously the above routine won't work unless the address label that I randomly called WHOOP has been defined elsewhere. This is just to let you see the logic. Incidentally, `CP 97` can also be coded as `CP "a"`.

Controlling a Sprite

We're going to make use of 23560 to detect key-presses and move a UDG across the screen. Since we'll be using this system variable several times, it makes sense to attach a label to it. Traditionally, this is named LASTK. The following should be placed under your origin statement along with another other system variables that you wish to label:

```
LASTK   EQU 23560
```

Arcade games are usually coded in the manner of a main loop that keeps circling round and round, waiting for the player to press a key. The following program uses that logic:

```
        ORG 30000

LASTK   EQU 23560
PRINT   EQU 8252

        LD A, 2             ; set print channel to screen
        CALL 5633
        LD HL, GFX          ; set up UDGs
        LD (23675), HL

MAINLP  CALL PRTPLAY        ; print player sprite
        LD A, (LASTK)       ; read last key-press
        CP "o"              ; was it "o"?
        JR Z, GOLEFT        ; if so, jump to GOLEFT
        CP "p"              ; was it "p"?
        JR Z, GORIGHT       ; if so, jump to GORIGHT
        JR MAINLP           ; loop back to scan again

GOLEFT  LD A, " "           ; change graphic to empty space
        LD (PLAYER+3), A    ; store it
        CALL PRTPLAY        ; undraw graphic from screen
        LD A, 144           ; change graphic back to normal
        LD (PLAYER+3), A    ; store it
        LD A, (PLAYER+2)    ; get player's X coordinate
        DEC A               ; subtract 1
        LD (PLAYER+2), A    ; store new X coordinate
        LD A, 0             ; load A with 0 (meaning no key-press)
        LD (LASTK), A       ; clear last key-press
        JR MAINLP           ; jump to start of main loop

GORIGHT EQU $               ; (to be completed)

PRTPLAY LD DE, PLAYER       ; print graphic
        LD BC, EOPLAYR-PLAYER
        CALL PRINT
        RET

PLAYER  DEFB 22, 21, 15, 144
EOPLAYR EQU $

GFX     DEFB 24, 24, 255, 189, 189, 60, 36, 102
```

In BASIC variables would be employed to store a sprite's position on the screen, typically called Y and X. In our current context, we can do without them, for an ingenious reason. The sprite's start position is Y=21 and X=15. These values are defined within the print data for PLAYER:

```
PLAYER  DEFB 22, 21, 15, 144
```

The address labelled PLAYER contains the ASCII code for AT (22); the address implicitly known as PLAYER+1 is the Y coordinate, and PLAYER+2 is the X. In the main part of the program, we can simply…

load this data into A:	`LD A, (PLAYER+2)`
change it:	`DEC A`
and put it back:	`LD (PLAYER+2), A`

Notice that we're actually modifying part of the assembled program from within the program as it's running – something that would never be allowed in BASIC.

I've said previously that the Spectrum's screen is 24 characters in height (Y coordinates 0 to 23), but I haven't set the player's Y coordinate to 23 (the final row). Instead, I set it to 21. This is due to a limitation of the ROM's print subroutine, which exists primarily to serve BASIC programs. The BASIC interpreter reserves the bottom two rows of the display for the parser (that is, the user's input facility) at all times. So the maximum Y value we can print at is 21. Alternative graphics-handling routines can be programmed, which don't rely on the ROM's pre-made method. You can even create a routine that enables larger sprites to move smoothly, one pixel at a time, with animation, but that's a much more complicated lesson for a future date.

When you test the program, movement to the left will work when the O key is tapped, but you can see that I've omitted the GORIGHT routine. Have a go at constructing the missing section (hint: it's very similar to GOLEFT).

When testing your program, try moving the sprite off the edge of the screen. The error "B Integer out of range 0:1" appears. We need to insert some additional code into the GOLEFT and GORIGHT sections to prevent attempts at movement beyond the perimeter. See if you can use a combination of CP with JR Z (or a related JR opcode) to accomplish this.

9: Player Input (Arcade-style)

Reading Hardware Ports

The previous chapter's method of reading input from the keyboard isn't really suitable for arcade games; it designed more for typing text. It doesn't allow you to hold a key in for an extended duration to achieve a smooth movement, and it can't handle two keys held down simultaneously, such as P and Q to move a sprite diagonally up and right. Helpfully, there's an entirely different method of reading the keys that solves all this.

The Z80 instructions IN and OUT are used to read and write to a computer's hardware ports. A port is like a point of communication between the processor and the external world. The edge connector at the back of the Spectrum, where a printer or joystick interface can be attached, is one such point of communication. The sound buzzer is another. But the port we're really interested in is the one relating to the keyboard.

Actually it's ports, plural. 8 ports are employed in the service of enabling the processor to watch the keyboard for presses. The port with the address $FEFE scans the keys shift, Z, X, C and V. The following table shows all eight:

Port	bit0	bit1	bit2	bit3	bit4
$FEFE	Shift	Z	X	C	V
$FDFE	A	S	D	F	G
$FBFE	Q	W	E	R	T
$F7FE	1	2	3	4	5
$EFFE	0	9	8	7	6
$DFFE	P	O	I	U	Y
$BFFE	Enter	L	K	J	H
$7FFE	Space	Symbol	M	N	B

Here's how this works. If I wanted to detect whether the X key was currently being pressed, I would examine binary bit 2 from the byte stored at port $FEFE. The 8 bits in a byte are labelled from right to left, as 0 to 7. When X is being pressed, bit 2 of the byte will be 0. The other bits will be 1 or 0, depending on what other keys are being pressed simultaneously. Counterintuitively, 0 means pressed while 1 means not pressed.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
?	?	?	?	?	0	?	?

To check whether X has been pressed in assembly language, code the following:

```
LD BC, $FEFE      ; load port address into BC
IN A, (C)         ; load port data into A
AND %00000100     ; isolate the key we're scanning for
JR Z, XPRESS      ; if X is pressed (if 0), jump
```

This is quite complex, so I'll break it down fully. First, you might be wondering why the IN instruction looks at C and not BC, since the full port address is $FEFE. While port addresses are

16-bit in nature, they all have $FE as their low byte (refer to the keyboard table above), so BC is implicit in `IN A, (C)`.

The AND instruction is best explained with some test data. Let's say that some random keys were being pressed simultaneously (excluding X) and this resulted in port $FEFE containing the data %01101101. The IN instruction loads this data into the A register, so the AND instruction will compare %00011101 with its operand %00000100 using AND logic.

```
A       = 0 0 0 1 1 1 0 1
operand = 0 0 0 0 0 1 0 0
          ---------------
result  = 0 0 0 0 0 1 0 0
```

The result is not zero (meaning that X has not been pressed). Since the zero flag has not been set, the jump to the address labelled XPRESS won't occur.

Now let's re-test the code with X pressed (bit 2 as 0). We'll also hold down some other random keys at the same time, changing the byte in port $FEFE to %0010010.

```
A       = 0 0 0 1 0 0 1 0
operand = 0 0 0 0 0 1 0 0
          ---------------
result  = 0 0 0 0 0 0 0 0
```

Now the result is 0, which will set the zero flag and enable the jump to XPRESS.

Test the following program:

```
        ORG 30000

PRINT   EQU 8252

        LD A, 2             ; set print channel to screen
        CALL 5633
        LD HL, GFX          ; set up UDGs
        LD (23675), HL

MAINLP  CALL PRTPLAY        ; print player sprite
        HALT
        HALT
        HALT

        ; scan for left ("o")
        LD BC, $DFFE        ; load port address into BC
        IN A, (C)           ; load port data into A
        AND %00000010       ; isolate the key we're scanning for
        JR Z, GOLEFT        ; if X is pressed (if 0), jump

        ; scan for right ("p")
        LD BC, $DFFE
        IN A, (C)
        AND %0000001
        JR Z, GORIGHT

        JR MAINLP           ; loop back to continue scanning
```

```
GOLEFT  LD A, (PLAYER+2)    ; if player is at left edge, don't continue
        CP 0
        JR Z, MAINLP
        CALL UNDRAW
        LD A, (PLAYER+2)    ; get player's X coordinate
        DEC A               ; subtract 1
        LD (PLAYER+2), A    ; store new X coordinate
        JR MAINLP           ; jump to start of main loop

GORIGHT LD A, (PLAYER+2)    ; if player is at right edge, don't continue
        CP 31
        JR Z, MAINLP
        CALL UNDRAW
        LD A, (PLAYER+2)    ; get player's X coordinate
        INC A               ; add 1
        LD (PLAYER+2), A
        JR MAINLP

PRTPLAY LD DE, PLAYER       ; print graphic
        LD BC, EOPLAYR-PLAYER
        CALL PRINT
        RET

UNDRAW  LD A, " "           ; change graphic to empty space
        LD (PLAYER+3), A    ; store it
        CALL PRTPLAY        ; undraw graphic from screen
        LD A, 144           ; change graphic back to normal
        LD (PLAYER+3), A    ; store it
        RET

PLAYER  DEFB 22, 21, 15, 144
EOPLAYR EQU $

GFX     DEFB 24, 24, 255, 189, 189, 60, 36, 102
```

Slowing a Game Down

With a little careful examination, you should be able to comprehend everything above, except one new opcode that I snuck in: HALT. Try running the code without the three HALT instructions at the beginning of MAINLP. Unlike games written in BASIC, which tend to run slugglishly by default, machine code is lightning quick. So what we're using HALT to do is slow things down. HALT functions like the BASIC instruction PAUSE. It doesn't have an operand, so you can't type `HALT 100` for a longer pause (HALT's actual meaning is "suspend processor operation until an interrupt or reset is received," but that level of detail is irrelevant in our current context). For longer pauses, you can use HALT in conjunction with DJNZ.

Optimising Repeated Code Using a Subroutine

Notice that I've created a new subroutine called UNDRAW. When you completed the two movement sections in the previous chapter, you may have noticed that a lot of code was being repeated for GOLEFT and GORIGHT. The code that undraws the sprite was identical in both

routines, so I moved it to a separate subroutine, which could then be called from anywhere, anytime. This made the program smaller and saved memory.

Pressing Two Keys at Once

Reading key-presses via hardware port addresses works even if more than one key is held down. This makes the method much more powerful than the LASTK approach. You might want to design a game that features a sprite which can move diagonally, using two keys held simultaneously. Or you might want the player to be able to tap repeatedly on a key that fires a laser while maintaining movement in a direction with another key. To achieve this, you only have to read the correct binary bits from the correct port addresses (singular or plural), and program the intended consequences.

10: Rapid Screen Drawing

Copying blocks of data

There's a handy instruction in Z80 that enables you to copy a block of data from one position in memory to another: LDIR. There's no operand for this opcode. It knows what to do based on whatever 16-bit values are pre-loaded into registers BC, DE and HL. HL should be set to the starting address of the data to be copied. BC should be set to the size of the data block (in bytes). DE should be set to the destination address of the data.

In the context of games programming, this opcode is particularly useful for rapidly drawing graphics – even an entire screen. All you need to do is copy a large chunk of data into the screen memory, which begins at address 16384. To illustrate this, let's start by copying the screen memory to elsewhere. Type in and assemble the following program:

```
ORG 50000
LD HL, 16384      ; start address
LD BC, 6912       ; number of bytes to copy
LD DE, 30000      ; destination address
LDIR
RET
```

When we execute this from within BASIC, it will copy everything that's on the screen into address 30000 and above. However, the screen is empty at present. So, let's quickly fill it with something. Type the following into BASIC:

```
10 PRINT INK INT (RND*8); PAPER INT (RND*8); "Hello";
20 GO TO 10
```

Run the program. When the screen fills up, press the break key, and with the multicoloured "Hello" messages still visible, type `RANDOMIZE USR 50000`. Then press enter to clear the display.

Now let's tweak the Z80 program so that it has the opposite effect of copying the data from address 30000 (and above) back to 16384. Change line 2 to `LD HL, 30000` and line 4 to `LD DE, 16384`. Assemble the program once again, and in BASIC type `RANDOMIZE USR 50000`.

Presto! Your seemingly lost screen full of graphics instantly re-appears.

Conclusion

Z80 assembly language is a rabbit hole that goes much deeper than the level we've explored so far. But I trust it's not quite as alien a place as it used to feel. There are many teaching resources available that can take you further down. My mission was only to get you started. In closing, I want to show you that even the little we've learned so far is sufficient to code a game of substance. It's all about how you put the pieces together.

The puzzle game *Knights* was developed in 2017 by Arzola (https://heisarzola.itch.io/) for Windows, Mac OSX and Linux. While playing it on my PC, I realised that it would translate very well to an 8-bit platform like the ZX Spectrum, and at the time I was looking for a project to test my newly acquired Z80 coding skills. I contacted the developer, who kindly gave me permission to create a free version of the game for the Sinclair Spectrum community.

Here it is, fully playable, containing all the levels of the original and a passcode system to save your progress. The script below may look daunting, but all the opcodes have been covered in preceding chapters. The text has also been heavily commented to help you understand what each section does. It can be awkward to extract a script from a PDF file, so I've made it available on GitHub: https://gist.github.com/darrylsloan/614adec99090dde6f16253e5c85c58b9

An enhanced version of the game, featuring pixel art by the talented Andy Green, sound effects, and music composed using *Wham! The Music Box*, can be downloaded from: https://spectrumcomputing.co.uk/index.php?cat=96&id=32308

This ebook is freely distributable. If you found it useful and wish to express your gratitude to the author, please consider making a small donation: https://www.paypal.me/darrylsloan

Good luck on your Z80 coding adventure!

```
; KNIGHTS by Darryl Sloan, 16 May 2018

        ORG 43000
LASTK   EQU 23560

START   LD A, 0                 ; border black
        CALL 8859
        LD A, 2                 ; select upper screen
        CALL 5633
        LD A, 8                 ; caps lock on
        LD (23658), A

; fill screen with paper black
        LD HL, 22528
        LD BC, 768
CLSLP   LD (HL), 0
        INC HL
        DEC BC
        LD A, B
        OR C
        JR NZ, CLSLP
```

```
; set up udgs
        LD HL, UDGS
        LD (23675), HL

; print intro messages
        LD DE, INTRO1
        LD BC, EOINTR1-INTRO1
        CALL PRTMSG
        LD DE, INTRO2
        LD BC, EOINTR2-INTRO2
        CALL PRTMSG
        LD DE, INTRO3
        LD BC, EOINTR3-INTRO3
        CALL PRTMSG
CODEENT LD DE, INTRO4
        LD BC, EOINTR4-INTRO4
        CALL 8252

; player inputs passcode
        LD DE, CURSPOS                ; set print position
        LD BC, EOCURSP-CURSPOS
        CALL 8252
; loop 4 times for 4 digits
        LD B, 4
INPLOOP PUSH BC
        LD DE, CURFLAS                ; print cursor
        LD BC, EOCURFL-CURFLAS
        CALL 8252
        LD DE, CURSPOS                ; retrace 1 step (to allow cursor
overwrite)
        LD BC, EOCURSP-CURSPOS
        CALL 8252
; scan for keypress
        LD HL, LASTK          ; clear last keypress
        LD (HL), 0
PAUSE   XOR A                 ; clear carry flag
        LD A, (LASTK)
        CP 65                 ; if keypress < "A", repeat
        JR C, PAUSE
        CP 91                 ; if keypress > "Z", repeat
        JR NC, PAUSE
; print keypress
        RST 16
; store keypress in pcode
        LD HL, PCODE
KSLOOP  LD A, (HL)         ; scan through pcode until empty slot found
        CP 0
        JR Z, KSTORE
        INC HL
        JR KSLOOP
KSTORE  LD A, (LASTK)
        LD (HL), A
; shift cursor position right by 1
        LD A, (CURSPOS+4)
        INC A
        LD (CURSPOS+4), A
        POP BC
        DJNZ INPLOOP         ; repeat until 4 digits entered
```

```
; locate desired level
; copy currently scanned data into lcode
CPCODE  LD HL, (LEVRES)
        INC HL
        LD DE, LCODE
        LD BC, 4
        LDIR
; compare pcode with lcode
        LD A, (LCODE)
        LD D, A
        LD A, (PCODE)
        CP D
        JR NZ, NEXTLEV
        LD A, (LCODE+1)
        LD D, A
        LD A, (PCODE+1)
        CP D
        JR NZ, NEXTLEV
        LD A, (LCODE+2)
        LD D, A
        LD A, (PCODE+2)
        CP D
        JR NZ, NEXTLEV
        LD A, (LCODE+3)
        LD D, A
        LD A, (PCODE+3)
        CP D
        JR NZ, NEXTLEV
        JR SETUPLV              ; code accepted, jump to setupbg
        DJNZ CPCODE
; scan forward until next 0 marker in data
NEXTLEV LD HL, (LEVRES)
SCANFWD INC HL
        LD A, (HL)
        CP 255      ; end of data marker met, error in player's code entry
        JR Z, CODEERR
        CP 0
        JR NZ, SCANFWD
        LD (LEVRES), HL         ; store new data restore point
        LD A, (LEVEL)           ; increase level count
        INC A
        LD (LEVEL), A
        JR CPCODE               ; jump back to scan new level code
; when user enters a wrong code
CODEERR LD A, 1                 ; reset level variables
        LD (LEVEL), A
        LD HL, LEVDAT
        LD (LEVRES), HL
        LD A, 28
        LD (CURSPOS+4), A
        LD A, 0
        LD (PCODE), A
        LD (PCODE+1), A
        LD (PCODE+2), A
        LD (PCODE+3), A
        LD DE, BADCODE
        LD BC, EOBADCO-BADCODE
```

```
        CALL PRTMSG
        JP CODEENT
; set data restore point to start of level data
SETUPLV LD (LEVRES), HL
; if level>9 then stage=2, width=4, height=3
        XOR A               ; reset flags
        LD A, (LEVEL)
        CP 9
        JR Z, SETUPBG       ; if level=9, exit
        JR C, SETUPBG       ; if level<9, exit
        LD A, 2             ; else make changes
        LD (STAGE), A
        LD A, 4
        LD (WIDTH), A
        LD A, 10
        LD (LSTART), A
        LD A, 21
        LD (LFIN), A
; if level>21 then stage=3, width=4, height=4
        XOR A               ; reset flags
        LD A, (LEVEL)
        CP 21
        JR Z, SETUPBG       ; if level=21, exit
        JR C, SETUPBG       ; if level<21, exit
        LD A, (STAGE)       ; else make changes
        LD A, 3
        LD (STAGE), A
        LD A, 4
        LD (HEIGHT), A
        LD A, 22
        LD (LSTART), A
        LD A, 37
        LD (LFIN), A
; if level>37 then stage=4, width=5, height=4
        XOR A               ; reset flags
        LD A, (LEVEL)
        CP 37
        JR Z, SETUPBG       ; if level=37, exit
        JR C, SETUPBG       ; if level<37, exit
        LD A, (STAGE)       ; else make changes
        LD A, 4
        LD (STAGE), A
        LD A, 5
        LD (WIDTH), A
        LD A, 38
        LD (LSTART), A
        LD A, 57
        LD (LFIN), A
; if level>57 then stage=5, width=5, height=5
        XOR A               ; reset flags
        LD A, (LEVEL)
        CP 57
        JR Z, SETUPBG       ; if level=57, exit
        JR C, SETUPBG       ; if level<57, exit
        LD A, (STAGE)       ; else make changes
        LD A, 5
        LD (STAGE), A
        LD A, 5
```

```
        LD (HEIGHT), A
        LD A, 58
        LD (LSTART), A
        LD A, 82
        LD (LFIN), A

; print stage & level
SETUPBG LD A, (STAGE)           ; insert current level in banner
        ADD A, 48
        LD (LEVDISP+9), A
; insert level code on banner
        LD A, (LCODE)
        LD (LEVDISP+41), A
        LD A, (LCODE+1)
        LD (LEVDISP+42), A
        LD A, (LCODE+2)
        LD (LEVDISP+43), A
        LD A, (LCODE+3)
        LD (LEVDISP+44), A
; fill in blank spaces on banner
        LD A, (LFIN)            ; lfin-lstart+1 = number of levels in stage
        LD HL, LSTART
        SUB (HL)
        INC A
        LD B, A
        LD HL, LEVDISP+13
LVEMPTY LD A, 157               ; insert empty banner udg
        LD (HL), A
        INC HL
        DJNZ LVEMPTY
        LD (HL), 158            ; insert close banner udg
; fill in completed levels on banner
        LD A, (LEVEL)
        LD HL, LSTART
        SUB (HL)
        CP 0
        JR Z, PRTBANN
        LD B, A
        LD HL, LEVDISP+13
LVCOMPL LD A, 156               ; insert crown banner udg
        LD (HL), A
        INC HL
        DJNZ LVCOMPL
; print level banner
PRTBANN LD DE, LEVDISP
        LD BC, EOLEVDI-LEVDISP
        CALL 8252

; store levres in tlevres to facilitate retry of level, if player wishes
        LD HL, (LEVRES)
        LD (TLEVRES), HL

; print borders of board
; print top-left cell
        LD A, 2                 ; reset y position to top
        LD (BORD1+5), A
        LD DE, BORD1
        LD BC, EOBORD1-BORD1
```

```
        CALL 8252
; work out width in amount of attr blocks
        LD A, (WIDTH)
        LD B, A
        LD A, 0
BWIDMUL ADD A, 3
        DJNZ BWIDMUL
        LD (WIDTHX3), A
; print top row of cells
        LD B, A
BTOP    LD A, 160
        RST 16
        DJNZ BTOP
        LD A, 159
        RST 16
; work out height in amount of attr blocks
        LD A, (HEIGHT)
        LD B, A
        LD A, 0
BHEIMUL ADD A, 3
        DJNZ BHEIMUL
        LD (HEIGHX3), A
; print all remaining rows except bottom
        LD A, (WIDTHX3)                 ; adjust width
        ADD A, 3
        LD (BORD2+6), A
        LD A, 3                         ; reset rows
        LD (BORD2+1), A
        LD (BORD2+5), A
        LD A, (HEIGHX3)                 ; loop by number of rows
        LD B, A
BMAIN   PUSH BC
        LD DE, BORD2
        LD BC, EOBORD2-BORD2
        CALL 8252
        LD A, (BORD2+1)
        INC A
        LD (BORD2+1), A
        LD (BORD2+5), A
        POP BC
        DJNZ BMAIN
; print bottom-left cell
        LD A, (HEIGHX3)         ; work out y position of bottom
        ADD A, 3
        LD (BORD1+5), A
        LD DE, BORD1
        LD BC, EOBORD1-BORD1
        CALL 8252
; print bottom row of cells
        LD A, (WIDTHX3)
        LD B, A
BBOTTOM LD A, 160
        RST 16
        DJNZ BBOTTOM
        LD A, 159
        RST 16

; print board tiles
```

```
; y, x loop within loop
        LD A, (HEIGHT)
        LD B, A
BHLOOP  PUSH BC
        LD A, (WIDTH)
        LD B, A
BWLOOP  PUSH BC
; print current square
        LD HL, (LEVRES)
        LD A, (HL)
        CP 1                        ; print empty square subroutine
        JR Z, BEMPTY
        CP 2                        ; print blocked square subroutine
        JP Z, BBLOCK
; print square featuring destination circle
        LD (GDEST+1), A             ; put colour (cyan or magenta in ink)
        LD (GDEST+12), A
        LD (GDEST+23), A
        LD (GDEST+32), A
        LD A, (BCOL)
        LD (GDEST+3), A
        LD A, (PY)
        LD (GDEST+5), A
        INC A
        LD (GDEST+17), A
        INC A
        LD (GDEST+25), A
        LD A, (PX)
        LD (GDEST+6), A
        LD (GDEST+18), A
        LD (GDEST+26), A
        LD DE, GDEST
        LD BC, EOGDEST-GDEST
        CALL 8252
; shift restore point forward to next square
NEXTSQU LD HL, (LEVRES)
        INC HL
        LD (LEVRES), HL
        LD A, (PX)                  ; increase x
        INC A
        INC A
        INC A
        LD (PX), A
        CALL SWAPCOL                ; swap board colour
        POP BC
        DJNZ BWLOOP
; new row
        LD A, 3                     ; reset x
        LD (PX), A
        LD A, (PY)                  ; increase y
        INC A
        INC A
        INC A
        LD (PY), A
        LD A, (WIDTH)               ; if width=4 (even), swap BCOL again
        CP 4
        JR NZ, NOSWAP
        CALL SWAPCOL
```

```
NOSWAP  POP BC
        DJNZ BHLOOP
        JR PRINTP               ; end of printing board, go to print pieces
; print empty square (subroutine)
BEMPTY  LD A, (BCOL)
        LD (GEMPTY+3), A
        LD A, (PY)
        LD (GEMPTY+5), A
        INC A
        LD (GEMPTY+11), A
        INC A
        LD (GEMPTY+17), A
        LD A, (PX)
        LD (GEMPTY+6), A
        LD (GEMPTY+12), A
        LD (GEMPTY+18), A
        LD DE, GEMPTY
        LD BC, EOGEMPT-GEMPTY
        CALL 8252
        JR NEXTSQU
; print blocked square (subroutine)
BBLOCK  LD A, (PY)
        LD (GBLOCK+5), A
        INC A
        LD (GBLOCK+11), A
        INC A
        LD (GBLOCK+17), A
        LD A, (PX)
        LD (GBLOCK+6), A
        LD (GBLOCK+12), A
        LD (GBLOCK+18), A
        LD DE, GBLOCK
        LD BC, EOGBLOC-GBLOCK
        CALL 8252
        JP NEXTSQU

; print knight pieces
PRINTP  LD A, 3                         ; reset x & y
        LD (PY), A
        INC A
        LD (PX), A
        LD A, 7                         ; reset board colour
        LD (BCOL), A
; y, x loop within loop
        LD A, (HEIGHT)
        LD B, A
PHLOOP  PUSH BC
        LD A, (WIDTH)
        LD B, A
PWLOOP  PUSH BC
; print current piece
        LD HL, (LEVRES)
        LD A, (HL)
        CP 1                        ; skip empty piece
        JR Z, NEXTPIE
; print knight piece
        LD (GKNIGHT+1), A           ; put colour (cyan/green/magenta) in ink
        LD A, (BCOL)
```

```
        LD (GKNIGHT+3), A
        LD A, (PY)
        LD (GKNIGHT+7), A
        INC A
        LD (GKNIGHT+11), A
        INC A
        LD (GKNIGHT+15), A
        LD A, (PX)
        LD (GKNIGHT+8), A
        LD (GKNIGHT+12), A
        LD (GKNIGHT+16), A
        LD DE, GKNIGHT
        LD BC, EOGKNIG-GKNIGHT
        CALL 8252
; shift restore point forward to next piece
NEXTPIE LD HL, (LEVRES)
        INC HL
        LD (LEVRES), HL
        LD A, (PX)              ; increase x
        INC A
        INC A
        INC A
        LD (PX), A
        CALL SWAPCOL            ; swap board colour
        POP BC
        DJNZ PWLOOP
; new row
        LD A, 4                 ; reset x
        LD (PX), A
        LD A, (PY)              ; increase y
        INC A
        INC A
        INC A
        LD (PY), A
        LD A, (WIDTH)           ; if width=4 (even), swap bcol again
        CP 4
        JR NZ, NOSWAP2
        CALL SWAPCOL
NOSWAP2 POP BC
        DJNZ PHLOOP
        LD A, 7          ; reset bcol to white for top-left start position
        LD (BCOL), A

; cursor movement
        LD A, 0                 ; caps lock off
        LD (23658), A
; initialise cursor to top left (3, 3)
        LD A, 3
        LD (PX), A
        LD (PY), A
        CALL GETATTR
        CALL PTATTR1
; northeast/northwest keypress
GAMELP  LD BC, $FBFE              ; north check
        IN A, (C)
        AND %00000001
        JR NZ, SKIPNEW
        LD BC, $DFFE              ; east check
```

```
        IN A, (C)
        AND %00000001
        JP Z, MOVENE
        LD BC, $DFFE                ; west check
        IN A, (C)
        AND %00000010
        JP Z,  MOVENW
SKIPNEW EQU $
; southeast/southwest keypress
        LD BC, $FDFE                ; south check
        IN A, (C)
        AND %00000001
        JR NZ, SKIPSEW
        LD BC, $DFFE                ; east check
        IN A, (C)
        AND %00000001
        JP Z, MOVESE
        LD BC, $DFFE                ; west check
        IN A, (C)
        AND %00000010
        JP Z,  MOVESW
SKIPSEW EQU $
; north keypress
        LD BC, $FBFE
        IN A, (C)
        AND %00000001
        JP Z, MOVEN
; east keypress
        LD BC, $DFFE
        IN A, (C)
        AND %00000001
        JP Z, MOVEE
; south keypress
        LD BC, $FDFE
        IN A, (C)
        AND %00000001
        JP Z, MOVES
; west keypress
        LD BC, $DFFE
        IN A, (C)
        AND %00000010
        JP Z,  MOVEW
; select keypress
        LD A, (LASTK)
        CP 32
        JP Z, SELECT
; reload level keypress
        LD BC, $FBFE
        IN A, (C)
        AND %00001000
        JP Z, RELOAD
        JP GAMELP

; move north
; if already at north end of board, skip move
MOVEN   LD A, (PY)
        CP 3
        JP Z, GAMELP
```

```
; execute move
        LD B, 3
N3SQ    PUSH BC
        CALL PTATTR0
        LD A, (PY)
        DEC A
        LD (PY), A
        CALL GETATTR
        CALL PTATTR1
        HALT
        HALT
        HALT
        POP BC
        DJNZ N3SQ
        CALL SWAPCOL
        JP GAMELP

; move northeast
; if already at north end of board, skip move
MOVENE  LD A, (PY)
        CP 3
        JP Z, GAMELP
; if already at east end of board, skip move
        LD A, (WIDTH)
        LD C, A
        XOR A
        LD B, 3
WIDX3NE ADD A, c
        DJNZ WIDX3NE
        LD B, A
        LD A, (PX)
        CP B
        JP Z, GAMELP
; execute move
        LD B, 3
NE3SQ   PUSH BC
        CALL PTATTR0
        LD A, (PX)
        INC A
        LD (PX), A
        LD A, (PY)
        DEC A
        LD (PY), A
        CALL GETATTR
        CALL PTATTR1
        HALT
        HALT
        HALT
        POP BC
        DJNZ NE3SQ
        JP GAMELP

; move east
; if already at east end of board, skip move
MOVEE   LD A, (WIDTH)
        LD C, A
        XOR A
        LD B, 3
```

```
WIDX3E  ADD A, c
        DJNZ WIDX3E
        LD B, A
        LD A, (PX)
        CP B
        JP Z, GAMELP
; execute move
        LD B, 3
E3SQ    PUSH BC
        CALL PTATTR0
        LD A, (PX)
        INC A
        LD (PX), A
        CALL GETATTR
        CALL PTATTR1
        HALT
        HALT
        HALT
        POP BC
        DJNZ E3SQ
        CALL SWAPCOL
        JP GAMELP

; move southeast
; if already at south end of board, skip move
MOVESE  LD A, (HEIGHT)
        LD C, A
        XOR A
        LD B, 3
HIGX3SE ADD A, c
        DJNZ HIGX3SE
        LD B, A
        LD A, (PY)
        CP B
        JP Z, GAMELP
; if already at east end of board, skip move
        LD A, (WIDTH)
        LD C, A
        XOR A
        LD B, 3
WIDX3SE ADD A, c
        DJNZ WIDX3SE
        LD B, A
        LD A, (PX)
        CP B
        JP Z, GAMELP
; execute move
        LD B, 3
SE3SQ   PUSH BC
        CALL PTATTR0
        LD A, (PX)
        INC A
        LD (PX), A
        LD A, (PY)
        INC A
        LD (PY), A
        CALL GETATTR
        CALL PTATTR1
```

```
        HALT
        HALT
        HALT
        POP BC
        DJNZ SE3SQ
        JP GAMELP

; move south
; if already at south end of board, skip move
MOVES   LD A, (HEIGHT)
        LD C, A
        XOR A
        LD B, 3
HIGX3S  ADD A, c
        DJNZ HIGX3S
        LD B, A
        LD A, (PY)
        CP B
        JP Z, GAMELP
; execute move
        LD B, 3
S3SQ    PUSH BC
        CALL PTATTR0
        LD A, (PY)
        INC A
        LD (PY), A
        CALL GETATTR
        CALL PTATTR1
        HALT
        HALT
        HALT
        POP BC
        DJNZ S3SQ
        CALL SWAPCOL
        JP GAMELP

; move southwest
; if already at south end of board, skip move
MOVESW  LD A, (HEIGHT)
        LD C, A
        XOR A
        LD B, 3
HIGX3SW ADD A, c
        DJNZ HIGX3SW
        LD B, A
        LD A, (PY)
        CP B
        JP Z, GAMELP
; if already at west end of board, skip move
        LD A, (PX)
        CP 3
        JP Z, GAMELP
; execute move
        LD B, 3
SW3SQ   PUSH BC
        CALL PTATTR0
        LD A, (PX)
        DEC A
```

```
        LD (PX), A
        LD A, (PY)
        INC A
        LD (PY), A
        CALL GETATTR
        CALL PTATTR1
        HALT
        HALT
        HALT
        POP BC
        DJNZ SW3SQ
        JP GAMELP

; move west
; if already at west end of board, skip move
MOVEW   LD A, (PX)
        CP 3
        JP Z, GAMELP
; execute move
        LD B, 3
W3SQ    PUSH BC
        CALL PTATTR0
        LD A, (PX)
        DEC A
        LD (PX), A
        CALL GETATTR
        CALL PTATTR1
        HALT
        HALT
        HALT
        POP BC
        DJNZ W3SQ
        CALL SWAPCOL
        JP GAMELP

; move northwest
; if already at north end of board, skip move
MOVENW  LD A, (PY)
        CP 3
        JP Z, GAMELP
; if already at west end of board, skip move
        LD A, (PX)
        CP 3
        JP Z, GAMELP
; execute move
        LD B, 3
NW3SQ   PUSH BC
        CALL PTATTR0
        LD A, (PX)
        DEC A
        LD (PX), A
        LD A, (PY)
        DEC A
        LD (PY), A
        CALL GETATTR
        CALL PTATTR1
        HALT
        HALT
```

```
        HALT
        POP BC
        DJNZ NW3SQ
        JP GAMELP

; player selects piece
; check if a knight exists under cursor
SELECT  CALL GETATTR
        LD DE, 33     ; focus on dead centre, avoiding destination corners
        ADD HL,de
        LD A, (HL)
        CP 120                  ; if empty square (white), check move
        JP Z, CHECKMV
        CP 72                   ; if empty square (blue), check move
        JP Z, CHECKMV
        CP 121                  ; if blocked square, deselect current
        JP Z, DESONLY
; deselect previous selection (if any)
        LD A, (TX)              ; if no current selection, skip ahead
        CP 255
        JR Z, NEWSEL
        LD B, A
        LD A, (PX)
        CP B
        JR NZ, DESELCT
        LD A, (TY)
        LD B, A
        LD A, (PY)
        CP B
        JR Z, NEWSEL
DESELCT CALL DESPREV
; select this piece
NEWSEL  LD A, (BCOL)
        LD (TBCOL), A             ; remember bcol for later
        CP 7
        JR Z, WHITDEC
        LD A, (HL)
        SUB 72
        JR NOTWHIT
WHITDEC LD A, (HL)
        SUB 120
NOTWHIT LD (TKCOL), A             ; match ink to original knight
        LD (GKFUZZY+1), A
        LD A, (BCOL)              ; match paper to bcol
        LD (GKFUZZY+3), A
        LD A, (PY)
        LD (GKFUZZY+7), A
        INC A
        LD (GKFUZZY+11), A
        INC A
        LD (GKFUZZY+15), A
        LD A, (PX)
        INC A
        LD (GKFUZZY+8), A
        LD (GKFUZZY+12), A
        LD (GKFUZZY+16), A
        LD DE, GKFUZZY             ; print fuzzy (selected) knight
        LD BC, EOGKFUZ-GKFUZZY
```

```
        CALL 8252
; clear lastk & remember x, y for later
        LD A, 0
        LD (LASTK), A
        LD A, (PY)
        LD (TY), A
        LD A, (PX)
        LD (TX), A
; loop back to player accepting input
        JP GAMELP

; deselect only (no new selection)
DESONLY CALL DESPREV
        JP GAMELP

; check if intended move is a valid chess-style move
CHECKMV LD A, (TX)
        ADD A, 3
        LD (TTX), A
        LD A, (TY)
        SUB 6
        LD (TTY), A
        LD A, (TTX)
        LD B, A
        LD A, (PX)
        CP B
        JR NZ, CHECK2
        LD A, (TTY)
        LD B, A
        LD A, (PY)
        CP B
        JR NZ, CHECK2
        JP EXECMOV
CHECK2  LD A, (TX)
        ADD A, 6
        LD (TTX), A
        LD A, (TY)
        SUB 3
        LD (TTY), A
        LD A, (TTX)
        LD B, A
        LD A, (PX)
        CP B
        JR NZ, CHECK3
        LD A, (TTY)
        LD B, A
        LD A, (PY)
        CP B
        JR NZ, CHECK3
        JP EXECMOV
CHECK3  LD A, (TX)
        ADD A, 6
        LD (TTX), A
        LD A, (TY)
        ADD A, 3
        LD (TTY), A
        LD A, (TTX)
        LD B, A
```

```
        LD A, (PX)
        CP B
        JR NZ, CHECK4
        LD A, (TTY)
        LD B, A
        LD A, (PY)
        CP B
        JR NZ, CHECK4
        JP EXECMOV
CHECK4  LD A, (TX)
        ADD A, 3
        LD (TTX), A
        LD A, (TY)
        ADD A, 6
        LD (TTY), A
        LD A, (TTX)
        LD B, A
        LD A, (PX)
        CP B
        JR NZ, CHECK5
        LD A, (TTY)
        LD B, A
        LD A, (PY)
        CP B
        JR NZ, CHECK5
        JP EXECMOV
CHECK5  LD A, (TX)
        SUB 3
        LD (TTX), A
        LD A, (TY)
        ADD A, 6
        LD (TTY), A
        LD A, (TTX)
        LD B, A
        LD A, (PX)
        CP B
        JR NZ, CHECK6
        LD A, (TTY)
        LD B, A
        LD A, (PY)
        CP B
        JR NZ, CHECK6
        JP EXECMOV
CHECK6  LD A, (TX)
        SUB 6
        LD (TTX), A
        LD A, (TY)
        ADD A, 3
        LD (TTY), A
        LD A, (TTX)
        LD B, A
        LD A, (PX)
        CP B
        JR NZ, CHECK7
        LD A, (TTY)
        LD B, A
        LD A, (PY)
        CP B
```

```
        JR NZ, CHECK7
        JP EXECMOV
CHECK7  LD A, (TX)
        SUB 6
        LD (TTX), A
        LD A, (TY)
        SUB 3
        LD (TTY), A
        LD A, (TTX)
        LD B, A
        LD A, (PX)
        CP B
        JR NZ, CHECK8
        LD A, (TTY)
        LD B, A
        LD A, (PY)
        CP B
        JR NZ, CHECK8
        JP EXECMOV
CHECK8  LD A, (TX)
        SUB 3
        LD (TTX), A
        LD A, (TY)
        SUB 6
        LD (TTY), A
        LD A, (TTX)
        LD B, A
        LD A, (PX)
        CP B
        JR NZ, CHKOVER
        LD A, (TTY)
        LD B, A
        LD A, (PY)
        CP B
        JR NZ, CHKOVER
        JP EXECMOV
CHKOVER CALL DESPREV
        JP GAMELP

; execute valid move
EXECMOV EQU $
; undraw previous piece
        LD A, 0                         ; ink black to previous knight
selection
        LD (GKWIPE+1), A
        LD A, (TBCOL)                   ; match paper to tbcol
        LD (GKWIPE+3), A
        LD A, (TY)
        LD (GKWIPE+7), A
        INC A
        LD (GKWIPE+11), A
        INC A
        LD (GKWIPE+15), A
        LD A, (TX)
        INC A
        LD (GKWIPE+8), A
        LD (GKWIPE+12), A
        LD (GKWIPE+16), A
```

```
        LD DE, GKWIPE                ; print space over knight
        LD BC, EOGKWIP-GKWIPE
        CALL 8252
; draw new knight
        LD A, (TKCOL)
        LD (TKCOL), A                ; match ink to original knight
        LD (GKFUZZY+1), A
        LD A, (BCOL)                 ; match paper to bcol
        LD (GKFUZZY+3), A
        LD A, (PY)
        LD (GKFUZZY+7), A
        INC A
        LD (GKFUZZY+11), A
        INC A
        LD (GKFUZZY+15), A
        LD A, (PX)
        INC A
        LD (GKFUZZY+8), A
        LD (GKFUZZY+12), A
        LD (GKFUZZY+16), A
        LD DE, GKFUZZY                ; print fuzzy (selected) knight
        LD BC, EOGKFUZ-GKFUZZY
        CALL 8252
        LD A, (PY)                    ; remember x & y for later
        LD (TY), A
        LD A, (PX)
        LD (TX), A

; check if level complete (all knights in destinations)
        CALL PTATTR0            ; remove cursor from screen
; y, x loop within loop
        LD HL, 22627            ; target first attr on top-left of board
        LD A, 1                 ; set win flag (temporarily)
        LD (WIN), A
        LD A, (HEIGHT)
        LD B, A
CHLOOP  PUSH BC
        LD A, (WIDTH)
        LD B, A
CWLOOP  PUSH BC
        LD A, (HL)
        CP 8                  ; skip if blank square (blue)
        JR Z, CWFIN
        CP 56                 ; skip if blank square (white)
        JR Z, CWFIN
        CP 57                 ; skip if blocked square
        JR Z, CWFIN
        LD B, A               ; put attr colour in b
        INC HL                ; scan right 1 attr
        LD A, (HL)            ; put attr colour in a
        DEC HL                ; retrace step on screen for later
        CP B                  ; if colours don't match, no win
        JR Z, CWFIN
        LD A, 0
        LD (WIN), A
CWFIN   INC HL                ; move right to next square
        INC HL
        INC HL
```

```
        POP BC
        DJNZ CWLOOP
; new row
        LD DE, 64               ; add two rows
        ADD HL, DE
        LD A, (WIDTHX3)         ; calculate width of board
        LD B, A
        LD A, 32
        SUB b
        LD D, 0                 ; move forward extra amount
        LD E, A
        ADD HL, DE
        POP BC
        DJNZ CHLOOP
        LD A, (WIN)
        CP 1                    ; check if win still acive
        JR Z, NEWLEV
        CALL PTATTR1
        JP GAMELP

; load new level after a win
NEWLEV  EQU $
; brief pause
        LD B, 40
NLPAUSE HALT
        DJNZ NLPAUSE
; clear lastk & increase level count
        LD A, 0
        LD (LASTK), A
        LD A, (LEVEL)
        INC A
        LD (LEVEL), A
; has player completed final level of current stage?
        LD B, A
        DEC b
        LD A, (LFIN)
        CP B
        JR NZ, FINLEV
; insert final crown into banner
        LD HL, LEVDISP+13
        LD A, (LSTART)                ; calculate number of levels in stage
        LD B, A
        LD A, (LFIN)
        SUB b
        LD B, A                       ; move forward number of levels
LASTCRO INC HL
        DJNZ LASTCRO
        LD A, 156                     ; crown graphic
        LD (HL), A
        LD DE, LEVDISP                ; print modified banner
        LD BC, EOLEVDI-LEVDISP
        CALL 8252
; has player completed final level of game?
FINLEV  LD HL, (LEVRES)
        LD A, (HL)
        CP 255                         ; end of data marker reached?
        JR Z, GAMEWON
; grab next level code
```

```
        INC HL
        LD DE, LCODE
        LD BC, 4
        LDIR
        LD (LEVRES), HL                 ; store new data restore point
LOADLV  LD A, 3                         ; reset x & y to top-left of board
        LD (PY), A
        LD (PX), A
        LD A, 255
        LD (TY), A
        LD (TX), A
        LD A, 7                         ; reset bcol
        LD (BCOL), A
        JP SETUPLV                      ; jump back to scan new level code

; reload current level
RELOAD  LD HL, (TLEVRES)
        JR LOADLV

; player completes game
GAMEWON LD HL, CONGRAT+1
        LD A, 0
        LD B, 8
CONGLP0 PUSH BC
CONGLP1 LD (HL), A
        PUSH AF
        LD DE, CONGRAT
        LD BC, EOCONGR-CONGRAT
        CALL 8252
        HALT
        HALT
        HALT
        POP AF
        INC A
        CP 7
        JR NZ, CONGLP1
CONGLP2 LD (HL), A
        PUSH AF
        LD DE, CONGRAT
        LD BC, EOCONGR-CONGRAT
        CALL 8252
        HALT
        HALT
        HALT
        POP AF
        DEC A
        CP 0
        JR NZ, CONGLP2
        POP BC
        DJNZ CONGLP0
; reset relevant variables and screen positions
        LD A, 0
        LD (PCODE), A
        LD (PCODE+1), A
        LD (PCODE+2), A
        LD (PCODE+3), A
        LD A, 1
        LD (LEVEL), A
```

```
        LD (STAGE), A
        LD (LSTART), A
        LD A, 9
        LD (LFIN), A
        LD HL, LEVDAT
        LD (LEVRES), HL
        LD A, 3
        LD (WIDTH), A
        LD (HEIGHT), A
        LD (PY), A
        LD (PX), A
        LD A, 255
        LD (TY), A
        LD (TX), A
        LD A, 7
        LD (BCOL), A
        LD A, 28                ; reset code-entry cursor position
        LD (CURSPOS+4), A
        LD HL, LEVDISP+23       ; reset banner graphic
        LD A, " "
        LD B, 16
FILL    LD (HL), A
        INC HL
        DJNZ FILL
        JP START                ; restart game

; subroutine: print message and pause (keypress to continue)
PRTMSG  CALL 8252
        LD HL, LASTK
        LD (HL), 0
        LD B, 250
PRTWAIT HALT
        LD A, (LASTK)
        CP 0
        JR NZ, PRTEND
        DJNZ PRTWAIT
PRTEND  RET

; subroutine: swap board colour (blue/white)
SWAPCOL LD A, (BCOL)
        CP 7
        JR Z, TURNBLU
        LD A, 7
        LD (BCOL), A
        JR NOTBLUE
TURNBLU LD A, 1
        LD (BCOL), A
NOTBLUE RET

; subroutine: work out curattr from x & y
GETATTR LD HL, 22528            ; start of attributes
        LD BC, 0                ; x offset
        LD A, (PX)
        LD C, A
        ADD HL, BC
        LD A, (PY)              ; y offset
        LD B, A
DOWNROW PUSH BC
```

```
        LD BC, 32
        ADD HL, BC
        POP BC
        DJNZ DOWNROW
        LD (CURATTR), HL
        RET

; subroutine: print attribute block
PTATTR1 LD DE, 30               ; for next line, retracing 2 steps from 32
        LD HL, (CURATTR)        ; top left attr in block
        LD B, 3
PTALOO1 LD A, (HL)              ; get current value
        ADD A, 64               ; add 64 to make bright
        LD (HL), A              ; put on screen
        INC HL                  ; go to next block on right
        LD A, (HL)              ; repeat 3 times...
        ADD A, 64
        LD (HL), A
        INC HL
        LD A, (HL)
        ADD A, 64
        LD (HL), A
        ADD HL, DE
        DJNZ PTALOO1
        RET

; subroutine: unprint attribute block
PTATTR0 LD DE, 30               ; for next line, retracing 2 steps from 32
        LD HL, (CURATTR)        ; top left attr in block
        LD B, 3
PTALOO2 LD A, (HL)              ; get current value
        SUB 64                  ; subtract 64 to turn off brightness
        LD (HL), A              ; put on screen
        INC HL                  ; go to next block on right
        LD A, (HL)              ; repeat 3 times...
        SUB 64
        LD (HL), A
        INC HL
        LD A, (HL)
        SUB 64
        LD (HL), A
        ADD HL, DE
        DJNZ PTALOO2
        RET

; subroutine: deselect previous selection
DESPREV LD A, (TX)
        CP 255
        RET z
        LD A, (TKCOL)               ; match ink to previous knight
selection
        LD (GKNIGHT+1), A
        LD A, (TBCOL)               ; match paper to bcol
        LD (GKNIGHT+3), A
        LD A, (TY)
        LD (GKNIGHT+7), A
        INC A
        LD (GKNIGHT+11), A
```

```
        INC A
        LD (GKNIGHT+15), A
        LD A, (TX)
        INC A
        LD (GKNIGHT+8), A
        LD (GKNIGHT+12), A
        LD (GKNIGHT+16), A
        LD DE, GKNIGHT                ; print normal knight over fuzzy
        LD BC, EOGKNIG-GKNIGHT
        CALL 8252
        LD A, 255
        LD (TX), A
        RET

; strings for graphics
GEMPTY  DEFB 16,0,17,7
        DEFB 22,3,3,"   "
        DEFB 22,4,3,"   "
        DEFB 22,5,3,"   "
EOGEMPT EQU $
GBLOCK  DEFB 16,1,17,7
        DEFB 22,3,3,154,154,154
        DEFB 22,4,3,154,154,154
        DEFB 22,5,3,154,154,154
EOGBLOC EQU $
GDEST   DEFB 16,5,17,1
        DEFB 22,3,3,150,16,0," ",16,5,151
        DEFB 16,0
        DEFB 22,4,3,"   "
        DEFB 16,5
        DEFB 22,5,3,152,16,0," ",16,5,153
EOGDEST EQU $
GKNIGHT DEFB 16,5,17,7,19,0
        DEFB 22,3,4,144
        DEFB 22,4,4,145
        DEFB 22,5,4,146
EOGKNIG EQU $
GKFUZZY DEFB 16,5,17,7,19,1
        DEFB 22,3,4,147
        DEFB 22,4,4,148
        DEFB 22,5,4,149
EOGKFUZ EQU $
GKWIPE  DEFB 16,0,17,7,19,0
        DEFB 22,3,4," "
        DEFB 22,4,4," "
        DEFB 22,5,4," "
EOGKWIP EQU $
BORD1   DEFB 16,2,17,0
        DEFB 22,2,2,159
EOBORD1 EQU $
BORD2   DEFB 22,3,2,161
        DEFB 22,3,11,161
EOBORD2 EQU $

; strings for text
INTRO1  DEFB 22,0,1,16,7,17,0,"Original PC game ",$7F," 2017 Arzola"
EOINTR1 EQU $
INTRO2  DEFB 22,0,0,"Spectrum version by Darryl Sloan"
```

```
EOINTR2 EQU $
INTRO3  DEFB 22,0,0,"Keys: Q, A, O, P, SPACE, R=Retry"
EOINTR3 EQU $
INTRO4  DEFB 22,0,0,"Level code (AAAA to begin):     "
EOINTR4 EQU $
CURSPOS DEFB 18,0,22,0,28        ; turn off flash, place cursor in y, x
EOCURSP EQU $
CURFLAS DEFB 18,1," "            ; print flashing cursor
EOCURFL EQU $
BADCODE DEFB 22,0,0,"Code not recognised!       "
EOBADCO EQU $
LEVDISP DEFB 16,7,17,0,19,0,22,0,0,"*"
        DEFB 16,6,155,"+++++++++]                ",16,7,"****"
EOLEVDI EQU $
CONGRAT DEFB 16,0,17,0,19,1,22,0,0,"  You have completed the game!  "
EOCONGR EQU $

; variables
PCODE   DEFB 0,0,0,0
LCODE   DEFB 0,0,0,0
WIN     DEFB 0
LEVEL   DEFB 1
LEVRES  DEFW LEVDAT
TLEVRES DEFW LEVDAT           ; temporary store to enable retry of level
STAGE   DEFB 1
WIDTH   DEFB 3
HEIGHT  DEFB 3
WIDTHX3 DEFB 0
HEIGHX3 DEFB 0
LSTART  DEFB 1
LFIN    DEFB 9
PY      DEFB 3                ; for cursor
PX      DEFB 3                ; for cursor
TY      DEFB 255              ; for selected piece
TX      DEFB 255              ; for selected piece (255=no current
selection)
TTY     DEFB 0                ; used when determining if a move is valid
TTX     DEFB 0                ; used when determining if a move is valid
BCOL    DEFB 7                ; for cursor
TBCOL   DEFB 7                ; for selected board tile
TKCOL   DEFB 5                ; for selected knight piece
CURATTR DEFW 22528

; udg data
; knight
UDGS    DEFB 0,0,0,0,16,28,46,126
        DEFB 255,79,15,31,63,127,255, 255
        DEFB 60,126,255,255,0,0,0,0
; knight fuzzy (selected)
        DEFB 0,0,0,0,16,24,52,106
        DEFB 245,74,13,26,53,106,213,234
        DEFB 52,106,213,255,0,0,0,0
; destination circle
        DEFB 255,252,240,224,192,192,128,128        ; top-left
        DEFB 255,63,15,7,3,3,1,1                    ; top-right
        DEFB 128,128,192,192,224,240,252,255        ; bottom-left
        DEFB 1,1,3,3,7,15,63,255                    ; bottom-right
; blocked tile
```

```
        DEFB 51,51,204,204,51,51,204,204
; progress bar
        DEFB 1,1,1,1,1,1,1,1                    ; start
        DEFB 255,0,73,107,127,127,0,255         ; filled cell
        DEFB 255,0,0,0,0,0,0,255                ; empty cell
        DEFB 192,64,64,64,64,64,64,192          ; finish
; board borders
        DEFB 0,126,84,106,84,106,84,0           ; corner
        DEFB 0,255,85,170,85,170,85,0           ; horizontal
        DEFB 84,106,84,106,84,106,84,106        ; vertical

; level data
; stage 1 (levels 1-9)
LEVDAT  DEFB 0, "AAAA"          ; passcode
        DEFB 1,1,1,2,2,1,5,2,1 ; board (1=empty, 2=block, 3/5=mag/cyn dest)
        DEFB 5,1,1,1,1,1,1,1,1 ; pieces (1=empty, 3/4/5=mag/grn/cyn knight)
        DEFB 0, "KVFG"
        DEFB 1,1,5,2,2,1,1,1,1
        DEFB 1,1,1,1,1,1,1,1,5
        DEFB 0, "BCPT"
        DEFB 1,1,1,5,2,5,1,1,1
        DEFB 5,1,1,1,1,1,1,1,5
        DEFB 0, "ZJFV"
        DEFB 1,5,1,1,2,5,1,1,5
        DEFB 1,5,1,5,1,1,1,1,5
        DEFB 0, "YRSA"
        DEFB 2,5,1,1,2,5,1,1,1
        DEFB 1,5,1,5,1,1,1,1,1
        DEFB 0, "DYJE"
        DEFB 2,1,1,1,2,5,5,1,1
        DEFB 1,1,1,5,1,1,1,5,1
        DEFB 0, "RQXE"
        DEFB 1,2,1,3,2,5,1,2,1
        DEFB 5,1,1,1,1,1,1,1,3
        DEFB 0, "KJOT"
        DEFB 3,1,1,1,2,1,1,1,5
        DEFB 5,1,1,1,1,1,1,1,3
        DEFB 0, "GSFK"
        DEFB 1,1,1,3,2,1,5,1,1
        DEFB 3,1,1,1,1,1,1,1,5
; stage 2
        DEFB 0, "BCTD"
        DEFB 1,1,3,1,2,5,1,5,1,1,3,1
        DEFB 1,3,1,5,1,1,1,1,1,3,1,5
        DEFB 0, "SHSP"
        DEFB 1,3,1,1,2,3,1,1,5,1,1,1
        DEFB 1,1,1,1,1,1,5,3,3,1,1,1
        DEFB 0, "HFSX"
        DEFB 1,5,1,1,1,2,3,3,1,5,1,1
        DEFB 1,5,1,5,1,1,1,1,1,3,1,3
        DEFB 0, "CECU"
        DEFB 5,1,3,1,1,2,1,2,3,1,5,1
        DEFB 1,3,1,5,1,1,1,1,1,3,1,5
        DEFB 0, "RYGE"
        DEFB 1,1,3,1,2,5,2,1,2,5,1,1
        DEFB 1,1,1,1,1,3,1,1,1,5,1,5
        DEFB 0, "ZSDO"
        DEFB 1,1,5,5,2,1,1,2,1,1,1,3
```

```
        DEFB 1,1,1,1,1,1,3,1,5,5,1,1
        DEFB 0, "BZEF"
        DEFB 3,1,1,5,1,1,1,1,1,5,3,1
        DEFB 5,1,1,3,1,1,1,1,5,1,1,3
        DEFB 0, "NWGL"
        DEFB 1,3,1,1,1,5,2,1,1,5,1,3
        DEFB 3,1,3,1,5,1,1,1,5,1,1,1
        DEFB 0, "JORX"
        DEFB 3,1,2,5,1,1,1,1,1,1,3,1
        DEFB 1,1,1,1,1,3,5,3,1,1,1,1
        DEFB 0, "PKGM"
        DEFB 3,1,1,3,1,2,1,2,5,1,1,5
        DEFB 5,1,1,5,1,1,1,1,3,1,1,3
        DEFB 0, "QBJM"
        DEFB 2,1,1,3,1,1,5,2,1,5,3,1
        DEFB 1,5,1,1,1,1,3,1,1,5,1,3
        DEFB 0, "VBEJ"
        DEFB 1,5,1,2,5,2,2,3,2,1,3,1
        DEFB 3,1,1,1,5,1,1,3,1,1,1,5
; stage 3
        DEFB 0, "PRJX"
        DEFB 1,1,5,5,1,1,2,5,3,2,2,1,3,3,1,1
        DEFB 1,1,3,3,1,1,1,3,5,1,1,1,5,5,1,1
        DEFB 0, "HJCW"
        DEFB 1,5,1,3,1,1,2,1,2,3,1,5,5,2,1,1
        DEFB 3,1,1,5,5,1,1,1,1,1,1,1,1,1,5,3
        DEFB 0, "NDLE"
        DEFB 2,1,1,2,3,2,5,5,3,3,2,5,2,1,1,2
        DEFB 1,1,1,1,5,1,3,3,5,5,1,3,1,1,1,1
        DEFB 0, "TKCW"
        DEFB 1,1,5,1,1,1,2,5,3,2,1,3,1,1,1,1
        DEFB 1,1,1,1,3,1,1,1,1,1,3,1,5,5,1,1
        DEFB 0, "KEBY"
        DEFB 5,2,1,5,3,1,1,3,1,1,2,2,2,3,1,5
        DEFB 1,1,1,1,5,1,3,5,1,3,1,1,1,1,5,3
        DEFB 0, "SYDE"
        DEFB 3,1,1,2,2,5,3,1,1,5,5,2,2,1,1,3
        DEFB 1,3,1,1,1,3,3,1,1,5,5,1,1,1,5,1
        DEFB 0, "TOYG"
        DEFB 3,1,1,5,1,3,5,1,1,5,3,1,5,1,1,3
        DEFB 1,3,3,1,5,1,1,5,5,1,1,5,1,3,3,1
        DEFB 0, "KDWG"
        DEFB 1,3,1,3,1,2,5,1,5,2,1,2,3,1,1,5
        DEFB 1,3,3,1,5,1,1,5,1,1,1,1,1,5,3,1
        DEFB 0, "YKUF"
        DEFB 1,1,1,5,5,1,3,2,3,5,1,1,1,2,1,3
        DEFB 1,3,5,1,1,5,1,1,1,1,3,5,3,1,1,1
        DEFB 0, "RJXK"
        DEFB 1,5,3,2,1,3,1,2,2,1,1,5,5,1,3,2
        DEFB 3,1,1,1,5,1,5,1,1,5,1,1,3,3,1,1
        DEFB 0, "WIMD"
        DEFB 5,3,5,3,1,1,1,1,1,1,1,1,3,5,3,5
        DEFB 1,1,1,1,3,5,3,5,5,3,5,3,1,1,1,1
        DEFB 0, "ECID"
        DEFB 5,2,1,3,3,3,1,5,1,2,1,2,1,5,1,2
        DEFB 1,1,1,5,1,5,3,3,5,1,1,1,1,1,3,1
        DEFB 0, "YDKM"
        DEFB 3,1,2,5,1,2,1,1,3,1,3,3,1,3,1,2
```

```
        DEFB 5,3,1,1,3,1,3,3,1,3,1,1,1,1,1,1
        DEFB 0, "RVYT"
        DEFB 1,3,1,2,1,5,1,3,1,1,2,2,5,1,3,5
        DEFB 1,1,1,1,3,1,3,5,5,3,1,1,1,1,5,1
        DEFB 0, "GSJT"
        DEFB 1,1,1,1,5,3,5,3,3,5,3,5,1,1,1,1
        DEFB 1,1,1,1,3,5,3,5,5,3,5,3,1,1,1,1
        DEFB 0, "LRBR"
        DEFB 3,1,5,1,2,1,3,5,5,3,1,2,1,2,2,1
        DEFB 5,1,1,3,1,3,5,1,1,5,3,1,1,1,1,1
; stage 4
        DEFB 0, "TMID"
        DEFB 3,2,1,1,5,3,2,1,2,5,3,2,1,2,5,3,1,1,2,5
        DEFB 5,1,1,1,3,5,1,1,1,3,5,1,1,1,3,5,1,1,1,3
        DEFB 0, "XHTS"
        DEFB 3,3,3,3,1,2,2,1,1,2,2,1,1,2,2,1,5,5,5,5
        DEFB 5,5,5,5,1,1,1,1,1,1,1,1,1,1,1,1,3,3,3,3
        DEFB 0, "KEIF"
        DEFB 1,1,1,3,3,2,2,1,2,2,2,2,1,2,2,5,5,1,1,1
        DEFB 5,5,1,1,1,1,1,4,1,1,1,1,4,1,1,1,1,1,3,3
        DEFB 0, "KFBT"
        DEFB 5,2,1,1,2,5,1,1,1,3,1,1,1,3,3,5,2,1,1,2
        DEFB 3,1,1,1,1,3,1,4,4,5,1,4,4,5,5,3,1,1,1,1
        DEFB 0, "ILYV"
        DEFB 5,1,2,1,5,1,1,3,1,2,2,1,1,1,1,3,1,5,1,3
        DEFB 3,1,1,3,4,4,4,5,4,1,1,5,3,4,1,4,1,4,1,5
        DEFB 0, "OMKF"
        DEFB 1,5,2,3,1,2,1,3,1,2,2,1,5,1,2,1,3,2,5,1
        DEFB 1,3,1,4,1,1,3,4,3,1,1,5,4,5,1,1,5,1,4,1
        DEFB 0, "JFIL"
        DEFB 5,2,1,1,3,1,3,1,2,1,1,2,1,1,1,3,5,1,2,5
        DEFB 4,1,3,1,5,3,4,4,1,1,5,1,1,4,4,4,4,5,1,3
        DEFB 0, "VNOG"
        DEFB 1,1,1,1,1,5,2,3,2,3,1,5,1,1,1,3,1,2,1,5
        DEFB 1,4,4,3,1,4,1,5,1,4,1,3,4,5,4,5,4,1,1,3
        DEFB 0, "QUJL"
        DEFB 1,1,1,5,1,2,2,1,2,3,2,2,3,2,1,2,2,1,5,1
        DEFB 5,4,1,1,4,1,1,3,1,5,1,1,4,1,4,1,1,3,1,1
        DEFB 0, "GDHT"
        DEFB 2,1,1,5,2,1,1,1,1,3,5,1,1,1,2,3,2,1,3,5
        DEFB 1,4,1,3,1,1,4,4,5,4,4,3,4,1,1,5,1,4,5,3
        DEFB 0, "MDUC"
        DEFB 1,5,3,1,2,5,2,1,1,1,1,1,3,2,1,1,5,2,3,1
        DEFB 1,3,4,4,1,4,1,4,3,5,4,5,4,1,1,1,3,1,5,4
        DEFB 0, "DHUK"
        DEFB 1,1,1,2,3,5,2,1,1,5,1,3,1,1,2,5,2,1,1,3
        DEFB 1,5,1,1,4,3,1,5,5,3,1,4,4,4,1,3,1,4,4,4
        DEFB 0, "AKFB"
        DEFB 2,2,2,5,3,1,1,1,1,2,1,1,1,1,2,2,2,2,3,5
        DEFB 1,1,1,3,5,4,1,4,4,1,4,1,4,4,1,1,1,1,5,3
        DEFB 0, "WHDP"
        DEFB 3,1,1,2,5,2,3,1,5,1,5,1,1,1,2,1,2,2,1,3
        DEFB 5,4,5,1,4,1,4,4,4,3,3,1,3,4,1,1,1,1,1,5
        DEFB 0, "SKFI"
        DEFB 2,1,5,1,1,1,3,2,1,5,1,1,3,1,1,1,5,1,2,3
        DEFB 1,3,3,1,3,1,5,1,4,4,4,4,5,4,4,1,4,4,1,5
        DEFB 0, "RJCE"
        DEFB 5,1,1,3,2,1,2,1,5,1,1,3,5,2,1,2,1,1,3,1
```

```
        DEFB 4,3,5,4,1,4,1,1,4,1,1,4,3,1,5,1,5,1,4,3
        DEFB 0, "IFNI"
        DEFB 1,1,1,2,1,5,3,2,5,1,2,1,1,1,3,3,1,5,1,1
        DEFB 4,1,3,1,1,4,4,1,3,4,1,3,5,4,4,4,5,4,5,1
        DEFB 0, "DKFX"
        DEFB 1,5,2,1,1,2,1,3,1,1,1,1,1,5,2,5,3,2,1,3
        DEFB 4,4,1,5,3,1,3,5,4,1,5,1,4,4,1,4,4,1,3,1
        DEFB 0, "JFBD"
        DEFB 1,1,1,5,5,1,1,1,3,2,3,2,1,2,1,3,1,5,1,1
        DEFB 1,5,3,4,4,5,4,1,4,1,4,1,4,1,3,4,3,4,1,5
        DEFB 0, "OHDY"
        DEFB 5,3,5,1,1,1,1,1,1,1,3,3,2,2,2,1,5,2,2,2
        DEFB 3,1,1,1,5,5,1,5,1,3,4,4,1,1,1,4,3,1,1,1
;stage 5
        DEFB 0, "KGKU"
        DEFB 3,1,1,1,1,1,1,5,2,1,1,5,2,3,1,1,2,3,1,1,1,1,1,1,5
        DEFB 4,4,4,4,3,1,5,4,1,1,3,4,1,4,5,1,1,4,5,1,3,4,4,4,4
        DEFB 0, "HIJF"
        DEFB 1,1,1,1,1,1,2,5,2,1,1,3,1,3,1,2,5,2,5,1,1,1,3,1,1
        DEFB 4,4,1,4,4,4,1,3,1,4,5,4,4,4,5,1,3,1,3,1,1,4,5,4,1
        DEFB 0, "FJYC"
        DEFB 1,5,1,5,1,1,2,5,2,1,1,1,1,1,1,1,1,1,1,3,2,3,2,3
        DEFB 5,3,5,3,5,1,1,3,1,1,1,4,1,4,1,4,4,4,4,4,4,1,4,1,4
        DEFB 0, "TGKD"
        DEFB 1,1,1,1,3,2,5,1,5,2,3,1,2,1,3,2,5,1,5,2,1,1,1,1,3
        DEFB 1,4,5,4,4,1,3,4,3,1,5,1,1,1,5,1,3,4,3,1,1,4,5,4,4
        DEFB 0, "LGFB"
        DEFB 1,3,1,1,2,1,2,3,1,1,1,5,2,3,1,1,1,5,2,1,2,1,1,5,1
        DEFB 1,5,4,1,1,4,1,5,4,1,4,3,1,5,4,1,4,3,1,4,1,1,4,3,1
        DEFB 0, "KYLR"
        DEFB 1,1,3,1,1,1,2,3,2,1,5,5,2,5,5,1,2,3,2,1,1,1,3,1,1
        DEFB 1,4,5,4,1,4,1,5,1,4,3,3,1,3,3,4,1,5,1,4,1,4,5,4,1
        DEFB 0, "UGFT"
        DEFB 5,2,1,1,1,1,1,5,1,1,3,1,3,2,3,1,1,5,1,1,5,2,1,1,1
        DEFB 4,1,5,1,4,4,3,4,4,1,5,5,3,1,4,4,3,4,4,1,4,1,5,1,4
        DEFB 0, "YTFV"
        DEFB 1,1,1,1,3,1,1,3,2,1,5,2,1,5,1,1,3,2,1,1,1,5,1,1,1
        DEFB 1,1,4,4,5,4,4,5,1,4,3,1,4,3,4,4,5,1,4,4,4,3,4,1,1
        DEFB 0, "TVLC"
        DEFB 1,1,2,1,1,1,1,2,1,1,5,1,1,1,5,3,2,5,2,3,2,3,5,3,2
        DEFB 4,3,1,3,4,1,4,1,4,1,3,1,4,1,3,5,1,4,1,5,1,5,4,5,1
        DEFB 0, "KTIV"
        DEFB 2,3,5,3,2,1,1,1,1,1,1,5,2,5,1,1,1,5,1,1,3,2,1,2,3
        DEFB 1,5,3,5,1,1,4,4,4,1,1,3,1,3,1,5,4,3,4,5,4,1,4,1,4
        DEFB 0, "CIKS"
        DEFB 1,1,3,1,1,3,2,1,2,3,1,2,3,2,1,5,5,1,5,5,1,1,1,1,1
        DEFB 1,4,4,4,1,4,1,4,1,4,3,1,4,1,3,1,3,4,3,1,5,5,1,5,5
        DEFB 0, "FYLV"
        DEFB 3,1,1,1,1,2,3,1,5,2,1,1,5,1,1,1,5,2,3,1,1,1,2,1,3
        DEFB 4,1,4,1,5,1,4,4,3,1,3,4,5,4,3,1,3,1,4,1,5,4,1,4,4
        DEFB 0, "EFZG"
        DEFB 2,1,5,1,2,1,3,1,3,1,1,2,5,2,1,1,1,1,1,1,1,5,3,5,1
        DEFB 1,4,3,4,1,4,4,1,4,4,5,1,4,1,5,1,5,1,5,1,4,3,4,3,4
        DEFB 0, "BIOF"
        DEFB 3,2,1,2,3,1,5,1,1,3,2,1,1,1,2,1,3,1,1,5,5,2,1,2,5
        DEFB 4,1,1,1,4,3,3,4,5,5,1,4,4,1,1,5,5,4,3,3,4,1,1,1,4
        DEFB 0, "WQHK"
        DEFB 5,1,1,1,5,2,1,2,1,2,1,1,1,1,1,1,2,3,2,1,5,3,1,3,5
```

```
DEFB 4,1,3,1,4,1,4,1,4,1,3,1,5,1,3,5,1,4,1,5,4,4,5,4,4
DEFB 0, "TRNO"
DEFB 3,2,1,5,1,2,1,3,2,5,1,3,1,3,1,5,2,3,1,2,1,5,1,2,3
DEFB 5,1,1,4,1,1,4,3,1,3,1,3,3,5,4,4,1,5,3,1,1,3,4,1,5
DEFB 0, "GLRO"
DEFB 1,1,3,1,1,3,2,1,2,3,2,5,1,5,2,5,2,1,2,5,1,1,3,1,1
DEFB 4,1,3,1,3,5,1,1,1,5,1,4,4,4,1,5,1,1,1,5,3,1,3,1,4
DEFB 0, "AZJR"
DEFB 2,1,5,1,1,1,1,3,2,1,5,3,1,3,5,1,2,3,1,1,1,1,5,1,2
DEFB 1,5,3,3,1,1,5,4,1,4,4,4,4,4,4,4,1,4,5,1,1,3,3,5,1
DEFB 0, "PTBL"
DEFB 1,1,1,1,1,1,3,5,2,1,5,2,2,3,5,1,3,5,2,1,1,1,1,1,1
DEFB 1,5,4,4,1,3,4,4,1,4,3,1,1,5,5,3,4,4,1,4,1,5,4,4,1
DEFB 0, "YZRL"
DEFB 5,2,5,1,1,2,1,5,1,2,5,5,3,2,3,1,1,2,1,1,1,2,3,1,2
DEFB 3,1,3,4,1,1,4,4,1,1,3,4,5,1,5,4,1,1,4,5,1,1,5,5,1
DEFB 0, "WRHS"
DEFB 3,2,1,3,1,1,5,2,1,1,2,1,5,1,1,1,5,2,1,1,3,2,1,3,1
DEFB 4,1,5,1,3,4,3,1,4,1,1,4,4,4,5,4,3,1,4,1,4,1,5,1,3
DEFB 0, "HRJA"
DEFB 3,1,1,1,2,5,3,1,5,1,1,2,1,2,1,1,5,1,3,3,2,1,1,1,5
DEFB 5,1,4,1,1,4,5,4,4,3,4,1,3,1,4,3,5,4,5,4,1,1,4,1,3
DEFB 0, "CRJS"
DEFB 1,1,3,1,1,1,2,1,2,1,5,3,5,3,5,1,2,1,2,1,1,1,3,1,1
DEFB 1,4,5,4,1,3,1,4,1,3,3,4,4,4,3,4,1,5,1,4,1,4,5,4,1
DEFB 0, "YSKR"
DEFB 1,1,3,1,2,1,1,3,1,1,5,5,2,5,5,1,1,3,1,1,2,1,3,1,1
DEFB 4,4,5,4,1,1,5,4,3,1,3,4,1,4,3,1,3,4,5,1,1,4,5,4,4
DEFB 0, "NSOR"
DEFB 1,1,3,1,1,1,3,2,3,1,5,1,2,1,5,1,2,1,2,1,1,5,3,5,1
DEFB 4,1,5,1,4,4,4,1,4,4,4,3,1,3,4,5,1,1,1,5,3,4,5,4,3
DEFB 255                        ; end of data marker
```