Z80 Assembler – Part 3

Introduction

We find ourselves at part three of these tutorials, so far we have been dealing with printing to the screen and number bases. In this part we will combine these with getting to grips with some of the binary logic and shift/rotate functionality of the Z80.

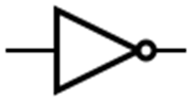
There have been a few requests to put a bit more information for those who have no previous Z80 knowledge. This would make sense, so ill be putting together a Part 0.5 / Pre-Tutorial which will cover the basic z80 instructions used in part 1. Ill also try and include a sort of glossary of instructions used in future parts. Ive got a few idea on cheat sheets ill put together than will be designed to be single page printouts covering the main things you might need to refer to.

Any comments etc, then please email myself at adrian@apbcomputerservices.co.uk or post to the Facebook page. I will endeavour to answer any questions and comment on any posts.

What are Binary Logic Gates

From our previous part you will know that binary is a base 2 number system, it only uses the values 0 and 1 to represent all numbers. At its most basic level everything on a digital computer is built up from these two values, 1 and 0, on and off… (A bit more information on digital computers can be found in the Tech Info section at the end). Logic Gates are devices that have one or more binary inputs and give one binary output, the main gates we will be discussing (as they are the ones we have instructions for in z80) are NOT, AND, OR, XOR.

We will quickly run through the gates and their truth tables before looking at their uses in more detail, so lets look at NOT first, this has only one input.



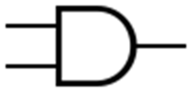
NOT

The truth table (that is the output returned given the inputs are as follows

INPUT	OUTPUT
0	1
1	0

As you can see, this is a very simple logic gate. Whatever binary value you pass in, it returns the opposite.

The next gate is AND



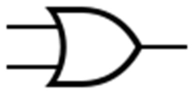
AND

Now as you can see, this has two inputs so our truth table is a little bigger.

INPUT A	INPUT B	OUTPUT
0	0	0
0	1	0
1	0	0
1	1	1

This gate does what it says, that is the output is 1 (true) when input A AND input B are 1. Only when both inputs are 1 does the output go to 1.

The third gate we will look at is OR.

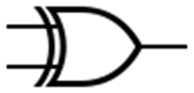

OR

Like AND this has two inputs, but the truth table has a noticeable difference.

INPUT A	INPUT B	OUTPUT
0	0	0
0	1	1
1	0	1
1	1	1

Like AND, this does exactly what it sounds like, the output is 1 if input A OR input B is 1. The only time this gate will return a 0 is when both inputs are 0.

The last gate is a little strange at first, the XOR (eXclusive OR – sometimes called EOR)


XOR

INPUT A	INPUT B	OUTPUT
0	0	0
0	1	1
1	0	1
1	1	0

The output from this gate returns 1 if input A or input B is 1 but not both, that is it must be exclusively A or B, not both. As I say this gate can seem a little odd at first but it has some really useful features. The other interesting feature of the XOR gate is (like many gates) it can be created using other gates, if you want to see how you can combine other gates to produce the XOR gate please refer to the tech info section.

Logic Gates in Z80

Now you are familiar with what the logic gates do, lets look at how the Z80 uses them. As with all arithmetic / logic functions on the Z80 these are performed with the A register and where needed the specified register. Starting with NOT unfortunately complicates things as the Z80 refers to this as CPL (ComPLiment – there is a reason for this when we get onto other instructions in later tutorials, but not for now you just need to know that NOT is CPL on Z80). It still does the same thing, all the bits are inverted. We don't have to specify another register as NOT only had one input and one output. When we looked at the NOT gate, we discussed it with one input bit and one output bit, as you are hopefully aware the A register is made up of 8 bits, so how does that work? It just performs the NOT on all the bits individually, so if A was %10011111 before hand and we did CPL (NOT) A would contain %01100000 afterwards, hopefully that makes sense, each bit is flipped so if the bit was 1 it becomes 0 and likewise if it was 0 it becomes 1. Lets actually see that working in practice, to do this we are going to use the following routine, it will print out the value of A in binary at a given position.

```
;-----------------------------------------------------------
; Inputs:
; A = Value to display
; L = X Position
; H = Y Position

DisplayBinary:
                    push    bc
                    push    hl
                    push    af

                    ; Remember A
                    ld              c, a

                    ; Write out the set position information
                    ld              a, 22
                    rst             &10
                    ld              a, h
                    rst             &10
```

```
                ld          a, l
                rst         &10

                ; Need to do 8 bits
                ld          b, 8
Loop:
                ; Shift the bit off the top
                rl          c

                ; Set A to 0 or 1 as needed
                ld          a, '0'
                jr          nc, BitZero
                ld          a, '1'
BitZero:
                ; Print it
                rst         &10

                ; Do all 8 bits
                djnz    Loop
                pop         af
                pop         hl
                pop         bc
                ret
```

We can now use a simple program to show the CPL instruction working as expected.

```
Start:
                ; Clear the screen
                call    0x0DAF

                ld      a, %10011111

                ; Display A
                ld      h, 0
                ld      l, 0
                call    DisplayBinary

                ; Perform the NOT
                cpl

                ; Display the new value below the first
                ld      h, 1
                ld      l, 0
```

```
        call    DisplayBinary

        ; Return to BASIC
        Ret
```

If you assemble and run this program you will get the output

```
10011111
01100000
```

As we would have expected. We can use this same base program to check the logic gate we investigated, the AND gate. You will recall this needed two inputs, one will always be the A register and the other you can specify. So AND B will perform and AND between the 8 bits of the A register and the 8 bits of the B register, leaving the result in A. The AND gate sets the output to 1 if and only if both the input bits are 1, so if we have

A = 10110011
B = 10001010

We should get the answer

A = 10000010

(Each bit is AND'd with the corresponding bit in both registers, that is BIT 7 if A with BIT 7 of B and the result is put in BIT 7 of A.)

```
Start:
        ; Clear the screen
        call    0x0DAF

        ld              a, %10110011
        ld              b, %10001010

        ; Display A
        ld              h, 0
        ld              l, 0
```

```
call    DisplayBinary

; Display B
ld      h, 1
ld      l, 0

; We need to put the value of B into A to display
push    af
ld      a, b
call    DisplayBinary
pop     af

; Perform the AND
and     b

; Display the new value below the first
ld      h, 2
ld      l, 0
call    DisplayBinary

; Return to BASIC
ret
```

If you assemble and run this program you will get the output

```
10110011
10001010

10000010
```

I will leave the first two challenges to you to change this program to perform and check the OR and XOR functionality.

Flags (F) register

If you know what the flags register is on the Z80, feel free to skip over this section, but we will start to make use of it so I thought it best we discuss it a little.

On the Z80, along with the registers you are probably familiar with A,B,C,D,E,H and L etc there is another register called F (flags). You will most probably have noticed it paired with A for things such as PUSH AF and POP AFas well as EX AF, AF' (Don't worry about these instructions if you haven't seen them before). You cannot access the F register like the others, its primary use is to store information about what has happened. The 8 bits in the F register each have a different meaning as below, don't worry if you don't understand all of these at the moment, we will return to them as needed (Before those that know say the unused bits are actually used, I am aware of the undocumented features of the Z80 but for now lets keep it simple, in the future we may return to undocumented features 😊).

Bit	Flag	Description
7	Sign	If the A register most significant bit (bit 7) is 1 then this is 1 to indicate A is negative in twos-compliment notation
6	Zero	Indicates certain operations that have completed had a zero result
5	Unused	
4	Half Carry	Used in mathematically operations to indicate a bit was carried between bits 3 and 4.
3	Unused	
2	Parity / Overflow	Used to indicate an overflow on some functions also parity of a byte normally for I/O
1	Add / Subtract	Indicates whether the last mathematically operation was an addition or subtraction
0	Carry	Indicates a bit was carried over or borrowed to complete the operation.

You can jump/call and return from programs based on the flags register, so if you wanted to CALL a routine if the zero flag was set you could use CALL Z,MyRoutine, if however you wanted to call it if the zero flag wasn't set you would use CALL NZ,MyRoutine (Z = Zero, NZ = Not Zero).

Not all instructions will change the flags, you will see tables that indicate which alter which. Knowing which do change the flags will be important in the future, for now we are only worrying about the zero flag.

Uses of binary logic.

When you start writing longer and more complex programs you will be amazed how much you come to use, one such use is to store flags in a single byte (Don't confuse the use of flags here with the F – flags register on the Z80, the use is similar but we are talking about our own flags, we will come onto the flags register in more detail at another point). Flags are generally used to signal something say that an enemy is moving left or right or that its animation loops back to the start rather than stopping at the end. You could use an entire byte for these things, so if move_left variable is not 0 then you move left, if it is zero you move right, but that's a lot of wasted space on a computer where space is limited. You only need to use 1 bit to store the left/right flag as its either set (1) for move left or reset (0) for move right. There are a few ways we could do this.

```
                    ld      a, (Flags)
                    bit     0, a
                    jr      z, MoveRight
MoveLeft:           …

                    ret


MoveRight:          …

                    Ret
```

This uses the BIT instruction which checks if a given bit in a register is 0 or 1. If its 0 then the ZERO flag is set to 1, otherwise the flag is set to 0. The routine above checks bit 0 or A, if its zero it would jump to the MoveRight routine, else it would carry on with the MoveLeft routine. You may be asking yourself how you could change the value that bit so we could say flip from left to right, that's where the SET/RES instructions come in.

```
                    ld      a, (Flags)
                    bit     0, a
                    jr      z, MoveRight
MoveLeft:           res     0, a
                    ld      (Flags), a
                    ret


MoveLeft:           set     0, a
                    ld      (Flags), a
                    ret
```

SET as you would expect sets the given bit to 1 while RES (Reset) clears the given bit to 0. This program would now change the bit over, so if its 0 it would become 1 while if it was 1 it would become 0. Now there is nothing technically wrong with using BIT/RES/SET in this way but its not the fastest instruction, this is where our binary logic comes into play.

We can replace the BIT instruction with one of the binary logic instructions we have talked about, the AND instruction. We can use AND to isolate given bits within a byte, so if we want just bit 0 like in the above example we could AND with the binary value %00000001. If you recall how the AND works we would get none of the top 7 bits because whatever value they were would return 0 (because we have masked them out using a 0 in our value), the only bit we would get is the bottom bit where we have used a bit value of 1. So AND %00000001 is the same as performing BIT 0, A. The AND instruction sets the ZERO flag if the entire of the A register is 0 else its resets it, remember we have masked out the top 7 bits so the ZERO flag will reflect only the bottom bit.

The one thing to remember when using AND in this way is that it changes all the other bits of the A register, as mentioned it resets them all to 0. The BIT instruction doesn't do this. So if we were going to use AND in the above program we would have to do the following

```
                    ld      a, (Flags)
                    and     %00000001
                    jr      z, MoveRight
```

```
MoveLeft:
                ld      a, (Flags)
                res     0, a
                ld      (Flags), a
                ret


MoveLeft:
                ld      a, (Flags)
                set     0, a
                ld      (Flags), a
                ret
```

We have to get the flags value into A again as the AND will have destroyed all the other flags in the variable. You may recall when discussing the flags register I said that not all instructions change the flags, well LD is one instruction that doesn’t change any flags, so whatever the flags register was before an LD it will be the same afterwards. In the above example we can use this to our favour to save repeating the ld a, (Flags) so much..

```
                ld      a, (Flags)
                and     %00000001
                ld      a, (Flags)
                jr      z, MoveRight
MoveLeft:       res     0, a
                ld      (Flags), a
                ret


MoveLeft:       set     0, a
                ld      (Flags), a
                ret
```

This time we reload A with the flags value after we have performed the AND, since the LD doesn’t alter the flags we can still jump based on the result of the AND.

Lets see if we can replace those SET/RES instructions next. What do we need to do, well logically for SET we don't care what the bit is already we always want the answer to be 1, the OR operation will do this if we OR with the value %00000001, why you ask, well if you think of the logic tables, using a 0 will leave the bit as it was before because if it was a 0 before and we OR with 0 – 0 OR 0 = 0, if it was 1 before and we OR with 0 we get 1 OR 0 = 1. The only bit we are going to alter is bit 0 because we have specified a 1. It doesn't matter if it was 0 or 1 before as 0 OR 1 = 1 and 1 OR 1 = 1, so no matter what happens the bottom bit will always be 1 and the rest will be left as they were.

What about the RES instruction, well that's similar to the BIT instruction except in this case we want to leave all the top 7 bits as they were and make the bottom bit 0, that's the AND instruction with the value %11111110 as where we have specified 1, 1 AND 1 is 1 but 0 AND 1 is 0 – that's fine as the answer is always the same as the value that was already there. For the bottom bit we specified 0 so 1 AND 0 = 0 and 0 AND 0 = 1. So it doesn't matter what the value was, because we passed in a 0 for that bit we will always get a zero out.

We can use this to change our program to

```
                        ld      a, (Flags)
                        and     %00000001
                        ld      a, (Flags)
                        jr      z, MoveRight
MoveLeft:               and     %11111110
                        ld      (Flags), a
                        ret

MoveLeft:               or      %00000001
                        ld      (Flags), a
                        ret
```

This is still far from the best way to do this, again it comes down to understanding logic which unfortunately is a case of 'the more you do it the better you get'. Lets look at what we want at a bit level. The input of BIT 0 to this function can be 0 or 1, we want the output to be the opposite, great you make thing, the NOT gate. Unfortunately the NOT on the Z80 works on all 8 bits.. DAMN… but there is still a way, a table always helps

STARTING FLAG VALUE	MASK VALUE	RESULT
0	0	0
1	0	1
0	1	0
1	1	1

Ive laid it out this way on purpose, but look carefully at the values, there is a logic gate that works for this, that is it will leave the starting value unchanged if you specify a 0 in the mask and will flip the flag if you specify a mask of 1. Look back at the truth tables and see if you can find it. Ill leave the last challenge as to rewrite this routine, its possible to do it in 3 instruction (without the RET at the end).

What else…

Binary Logic is good for other things, its mainly for masking things and combining things, for instance if we wanted to store a value in the top 4 bits and a different value in the bottom 4 bits of a byte we could use AND to separate them..

```
ld      a, %11011101
and     %11110000
```

The A register now holds only the TOP 4 bits, so in this case %11010000. If we change the AND to AND %00001111 we would be left with the bottom 4 bits (%00001101). So how do we use that, well imagine this function

```
ld      a, (Combined)
ld      b, a
and     %11110000
ld      c, a
ld      a, b
and     %00001111
ld      b, a
```

After this routine C would hold the top 4 bits and B would hold the bottom 4 bits. How about if we wanted to change the bottom 4 bits of Combined to be whatever was in the C register already.

```
ld      a, (Combined)
and     %11110000
or      c
ld      (Combined), a
```

The AND here clears the bottom 4 bits to 0 while leaving the top 4 bits as they are, the OR will mix the value in C into A before storing it back into combined.

This part has been going on a while and we have covered some complicated things, lets leave it here, you may want to re-read it a few times and play around with some examples. You can use the DisplayBinary routine we had at the start to check what happens if you like, ill leave a few challenges, but next time we shall get onto moving the bits around.. then the fun really starts and we can get to drawing our own graphics on the screen!

Challenges

1) Alter the Logic Gates in Z80 program to perform and check the OR instruction
2) Alter the Logic Gates in Z80 program to perform and check the XOR instruction

3) Look at the logic gates and write the program in Uses of binary logic to perform the same thing using only 3 instructions.

Instructions

Below is a bit of information on the main instructions we have used in this tutorial

Instruction	Description
CPL	Performs a NOT on all the bits of the A register
AND	Performs a bitwise AND between the A register and the value presented in this instruction (that's is either a specific value eg. AND %10101111, another register AND C, or from memory AND (HL)
OR	Performs a bitwise OR between the A register and the value presented in this instruction (that's is either a specific value eg. OR %10101111, another register OR C, or from memory OR (HL)
XOR	Performs a bitwise XOR between the A register and the value presented in this instruction (that's is either a specific value eg. XOR %10101111, another register XOR C, or from memory XOR (HL)
BIT	Sets the Z flag if the specified bit is 0 else resets the Z flag eg. BIT 0, A
SET	Sets the given bit of the register or memory to 1 eg. SET 0, A or SET 0, (HL)
RES	Resets the given bit of the register or memory to 0 eg. RES 0, A or RES 0, (HL)

Tech Info.

So for a little more fun and games lets look how a digital computers CPU can perform things such as addition when its only working with 1's and 0's. Adding two numbers can actually be broken down very easily into bits. Lets say we want to add 4 and 5, first lets look at the numbers in binary..

4 = %0100
5 = %0101

Now you know the answer to 4 + 5 is 9 which gives us

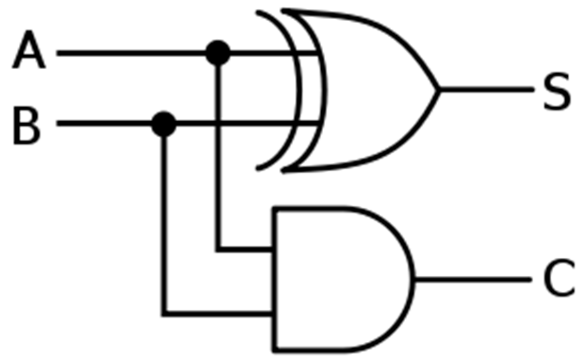
9 = %1001

But how does the computer do that, well it works on each column in turn, the same way as if I asked you to add 245 and 193 you would probably add the 5+3 to get 8, then 4+9 to give 13 (you would write down the 3 and carry the 1 into the next addition) and finally the 2+1, and the 1 you carried from the 4+9 to give 4, so 245+193 is 438. The computer does it the same way..

Lets look at the first column of 4+5 in binary 0 + 1, well that's easy its 1. Next column is 0+0, again easy its 0. Last we have 1+1, now in binary we know that's 10 (one zero, not ten!), using the same principle as before we carry that 1 over and keep the 0, so the answer is 0 but we now have a value for the next column 0+0+1 (which we carried over) gives us 1. If you look back that gives %1001. We can look at this as a series of truth tables, each column has two input bits from the sum which give the following table.

Value 1	Value 2	Answer	Bit carried over
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

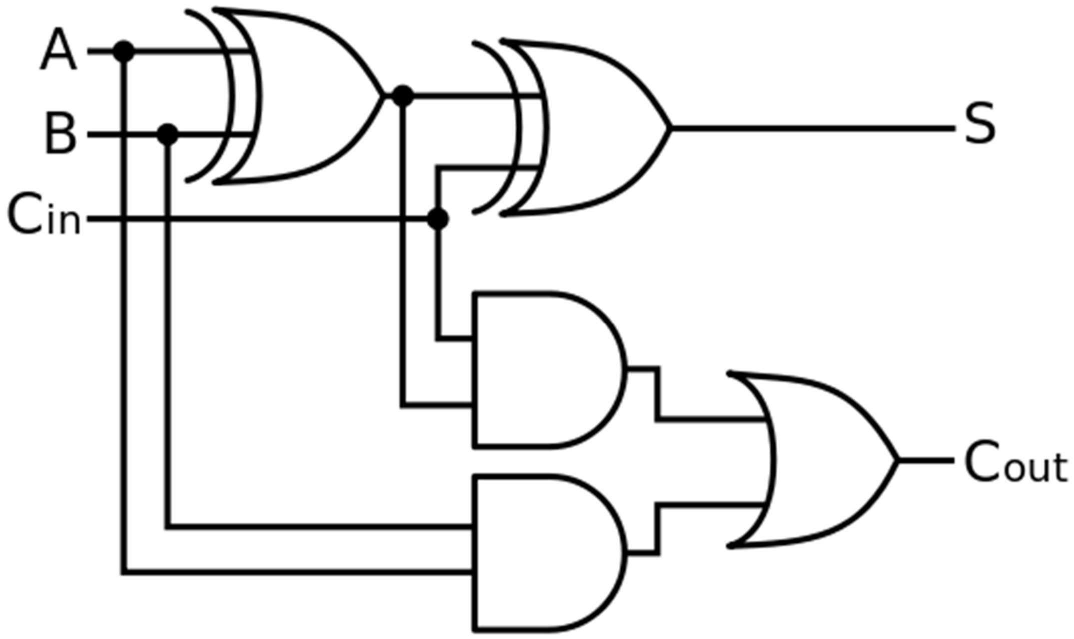
I don't know about you, but that Answer bit looks a lot like an XOR of the value 1 and value 2 bits while the Bit carried over look a lot like an AND gate.



Where A is Value 1, B is Value 2, the Answer is S and the Bit carried is C. This logic circuit is called the HALF ADDER. Its adding two bits together BUT as you may have noticed, it doesn't care about any carry that may have passed in.

Value 1	Value 2	Bit Carried In	Answer	Bit carried over
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

At first this looks a lot more complicated, but half of this we already know as it's the Half Adder, this will be the bit where the carry was 0 (i.e. there wasn't a carry forward). If you look at the final answer where the carry in was 1 you may notice it looks familiar, it's the inverted version of the answer when the carry in was 0. SOOO, if carry in is 1 we need the opposite value to if carry in is 0… Hang on that's an XOR!!! SOOO take the answer from the half carry and XOR it with the Carry In value gives us our final answer bit. Well that's half the battle, finally we need to sort that final carry over. This is a bit more tricky, we know when carry in is 0 we need to just AND the two input values, but what about when carry in is 1. There are a few options, the general way is to AND the answer from the half carry with the carry in – this gives us the second part which we can then OR with the first part from the half carry, hopefully this diagram will make it clearer. This is a fairly complex circuit but as you can see with a few simple logic gates we can make something that can count 1 bit, in the ALU of a CPU you would have several of these FULL ADDERS linked together with the answer going to the destination and the carry out from one bit going to the carry in for the next bit.


A
B
C_{in}
S
C_{out}