

Diving Deeper into the Z80

An adventure in discovery of the Z80 using logic simulation

Brian L. Little



Table of Contents

Introduction.....	1
The Adventure Begins.....	2
Thanks for the memories, the Registers.....	4
Register Latch.....	7
Register File.....	8
Where the action is, the ALU (Arithmetic Logic Unit).....	9
ALU Slice.....	11
PLA (Programmable Logic Array).....	12
Control.....	13
Putting it all together.....	14
Making it run.....	15
A quick dip into FPGA.....	16
Making it real on a FPGA.....	17

Introduction



The first computer I had the Timex Sinclair 1000, the US variant of the ZX81, which had a Z80 CPU. I quickly found out that I could bypass the limitations of Basic by using assembly language. The only limits now were the hardware. Since I learned 6502 assembly in college I quickly picked up on the Z80 assembly. I spent many hours pushing the limits of the Timex Sinclair 1000.

A few years later when I moved onto the PC I was hired as an electrical engineer for a company that made its own Z80 based computer used in quality control for the plastic molding industry. Working with the code rekindled my interest in the Z80. By that time my Timex Sinclair 1000 was long gone but the PC was more than capable of emulating it so I could once again enjoy coding Z80 assembly on the Timex Sinclair 1000.

Around 2010 I was introduced to FPGAs (Field Programmable Gate Arrays). The ability to construct hardware using Hardware Defined Language was intriguing. When I found out that you could also use logic symbols and simulate 74 series TTL chips it made me flash back to the days when I found the article for the Elf 1802 computer. I couldn't afford the 1802 on my lawn mowing profits of my neighbors but my room was littered with logic diagrams for expansion of the long sought after Elf II. I had to get a FPGA development board.

My first project was adapting a ZX81 written in VHDL to my new board. I felt triumphant when I finally got the flashing K in the corner of the screen and was able to play Mazogs on it. After that the next step was to actually design a cpu from scratch, either the 1802 or the Z80. That is where things ended. I didn't have much time to devote to it so from time to time I dabbled with my FPGA over the years, not accomplishing much. I even got another FPGA board with more bells and whistles to motivate me, which didn't work.

Over the last few years through the internet I found sites that had high resolution pictures of the Z80 substrate. Sites that reversed engineered the substrate into schematics and used them to make a Z80 on the FPGA. I also found a net level Z80 simulator and a digital simulator that could analyze a circuit and export hdl code. This sparked my interest again.

One thing I found lacking is more detailed explanation of the logic inside the Z80. So I am writing this to document what I found in my adventure.

So now armed with these resources we are ready to start our adventure by taking the deep dive into the Z80.



The Adventure Begins

The journey begins with the purpose of our adventure of discovery. We will look in detail the logic that drives the Zilog Z80 computer processing unit (cpu) henceforth will be mentioned as the Z80. To do this we will take the registers, arithmetic logic unit, programmable logic array and control systems individually. We will then dive into the heart of each system and look at the building blocks that make up the system.

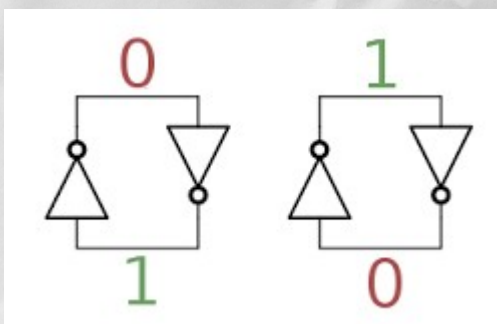
To do this we will be using a very versatile logic simulator [Digital](#). Digital covers all the basic logic gates, many of the 7400 series ics, and many peripherals. It highlights the state of each logic state so you can watch the logic flow from the input to the output. You can also apply tests of expected states to check the validity of the diagram.

The analysis feature produces truth tables, boolean algebra and karnaugh maps. It has probe and oscilloscope peripherals to visually inspect the timing visually. It also adds an adjustable delay in the gates to find potential timing conflicts.

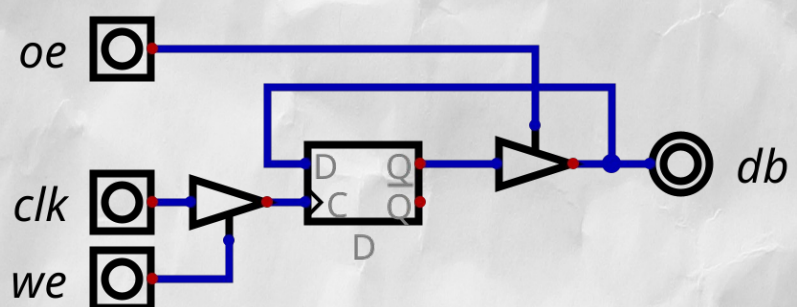
The export feature exports png and svg formatted pictures which are used in this journal. Also Verilog and VHDL Hardware Descriptive Language for Field Programmable Gate Arrays that we will use later on to physically implement our logic design.

The companion Github site [Diving Deeper into the Z80](#) has a resource folder that contains the Digital files (.dig), Verilog, VHDL, and png files for every subcircuit covered in the journal. Digital has a hierarchy design that permits one circuit to appear as blocks in circuits higher up in the hierarchy,

Since we are looking at the logic functionality of the Z80 we are using a representation of the logic not the actual circuitry of the Z80, for example the register in the Z80 is made up of two inverters that produce a stable state while the logic simulator uses a D flip flop with tri-state buffers both perform the same logic.



Simple inverter register



D Flip flop register

If you want to dive deeper into the actual circuitry and logic there are a few exceptional sites listed below

Visual6502.

Z80 Die shots

This site is hosted by a group that has decapped the Z80 and made high resolution shots of the dies which others have used for reverse engineering of the logic. It might be worth a look to see the other cpus and peripherals they have made die shots for.

Ken Shirriff's blog

Index

Z80 Notes A collection of technical notes that Ken has written about the inner workings and specifications of the Z80

Ken reverse engineers critical parts of the Z80 dies. He explains in detail the logic of the parts listed above. He has also reverse engineered various other cpus and shares their highlights as well.

Baltazar Studios

Baltazar Studio

Goran Devic hosts this site and he has many articles where he has reversed engineer the Z80. The site mentions his Github site that has the A-Z80 a FPGA soft core made up of the logic he reversed engineered from the die. He has broken down the logic into the schematics that we use in this journal.

Also Goran has written a program that gives you an in depth analysis of the actual Z80 architecture, the Z80 Explorer . The Explorer will load and run Z80 compiled programs and highlight the activity on a map of the substrate of the Z80 die. You can also click on a certain location and get the net list of the selected area. You can also label the net lists for easier identification.

Once we have covered the logic of the Z80 we will use the logic simulation in a computer simulation in Digital. The system will have a 80's retro feel with a 4x4 key pad, 7 segment leds displaying address and data, status leds, serial port, and a vga monitor. We will write a simple operating system that will permit the loading and saving files into the simulated computer.

Finally we will port our simulation into a FPGA development board, the Intel DE10-Lite which has all the above features with the addition of a membrane 4x4 keypad.

Throughout this journal you will see definition details as follows

Additional details for the subject

Now its time to dive deeper into the Z80.



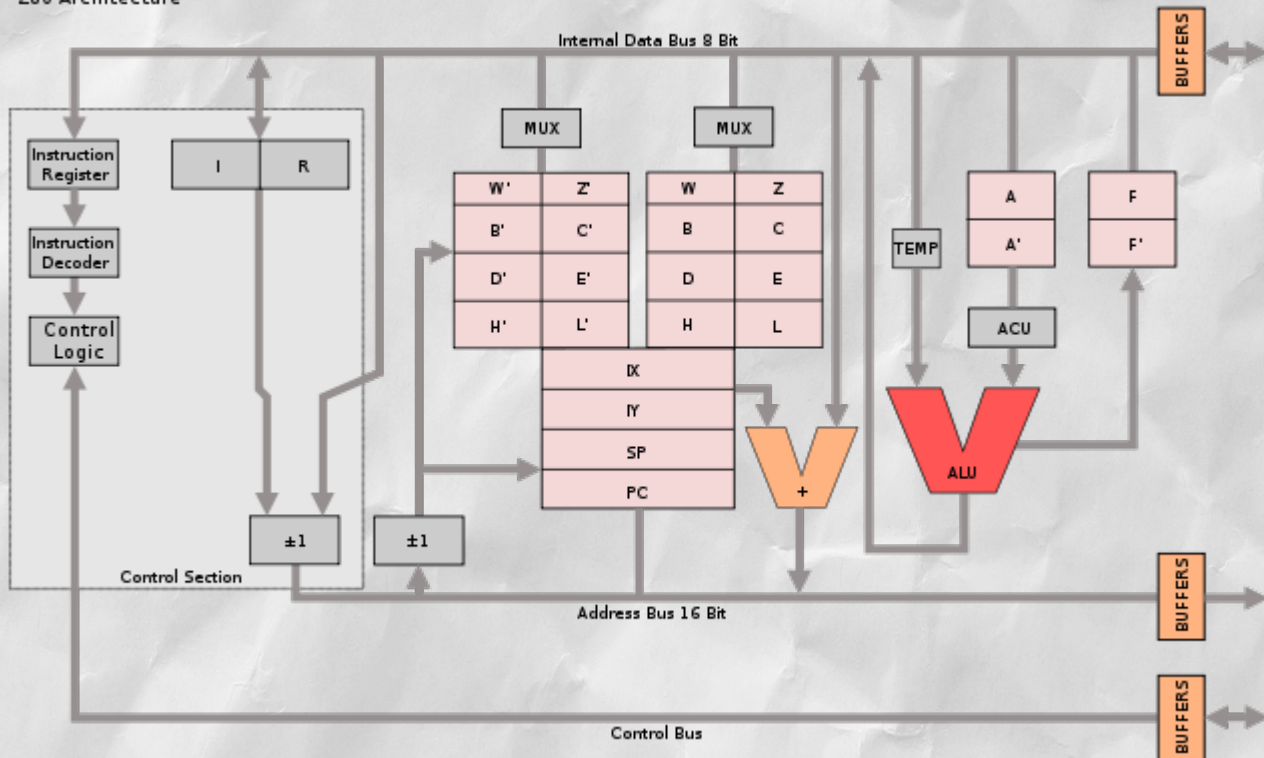
Thanks for the memories, the Registers

The first stop in our adventure is the registers. The registers are the scratch pad of the Z80 retaining data for the ALU to use, program flow and indexing. These registers are made up of latches which are sequential logic.

A **sequential digital circuit** is a logic circuit that is dependent on the previous values of inputs and is usually triggered by a clock signal. Latches, multiplexers, and flip flops are examples of sequential logic.

These registers are either 8 or 16 bit and the logical representation of the bank of registers is shown below.

Z80 Architecture



General

These are 8 bit registers that are paired together as AF, BC, DE and HL. There are also the alternate registers, A'F', B'C', D'E' and H'L' to these registers which can be switched back and forth from general to alternate.

The AF register register also has the purpose of being the accumulator and flags used in ALU processes. When an instruction requires data directly from memory the data is put into the accumulator first before doing the operation. Then the result of the operation is then placed



back into the accumulator. The F or flag register holds flags set by the operation in the ALU. These flags are:

Bit	Flag	Label	Description
0	Carry	C	Set if the adding process in the ALU produces a carry
1	Negate	N	Set if the previous operation was subtraction (2s complement)
2	Parity/Overflow	P/V	Set if the accumulator has an even number of set bits. If the 2s complement produces a carry it sets this bit
3	X	X	Not used
4	Half Carry	H	Takes the 3 rd bit bit value and moves it to the 4 th bit in the accumulator. See the next section on the ALU.
5	X	X	Not used
6	Zero	Z	Set if the accumulator has a zero value
7	Sign	S	Set for the use of 2s complement math which is used to make the add function do subtraction.

The b register is used as a decrementing counter for looping that continues until it goes to zero.

The BC, DE, and HL registers can be used as 8 bit registers if separated as B, C, D, E, H and L or as 16 bit registers BC, DE and HL.

Program Flow

The PC (Program Counter) is a 16 bit register that holds the pointer for the next instruction in memory to be processed. Once the operation is completed it increments the correct number of bytes for that operation. The PC can be altered by a jump or branch command. The call command puts the PC on the stack (see below) then puts the address in the call command. Once the return command is encountered the PC pulls from the stack so it can resume its operations. This process is used in interrupts.

The SP (Stack Pointer) is a 16 bit register that uses memory as a storage space. The LD SP command can load the start of the stack with a memory address. It decrements from that since it is a first in first out (fifo) stack. It can be used to preserve a 16 bit register (i.e. DE) before a call. Interrupts cause the PC and general register pairs to be pushed (placed) on the stack then

popped (taken) from the stack when the interrupt is finished. This preserves the state of the CPU to when an interrupt occurs.

Index Registers

The registers IX and IY are 16 bit registers. They hold a base address that a positive or negative index is added to the index can be incremented or decremented. This allows quick indexing of data tables.

Hardware Control

The I and R registers are 8 bit registers used in hardware control. The I register holds the upper 8 bits of an interrupt routine while the device doing the interrupting provides the lower 8 bit. This allows dynamic mapping of interrupts in memory.

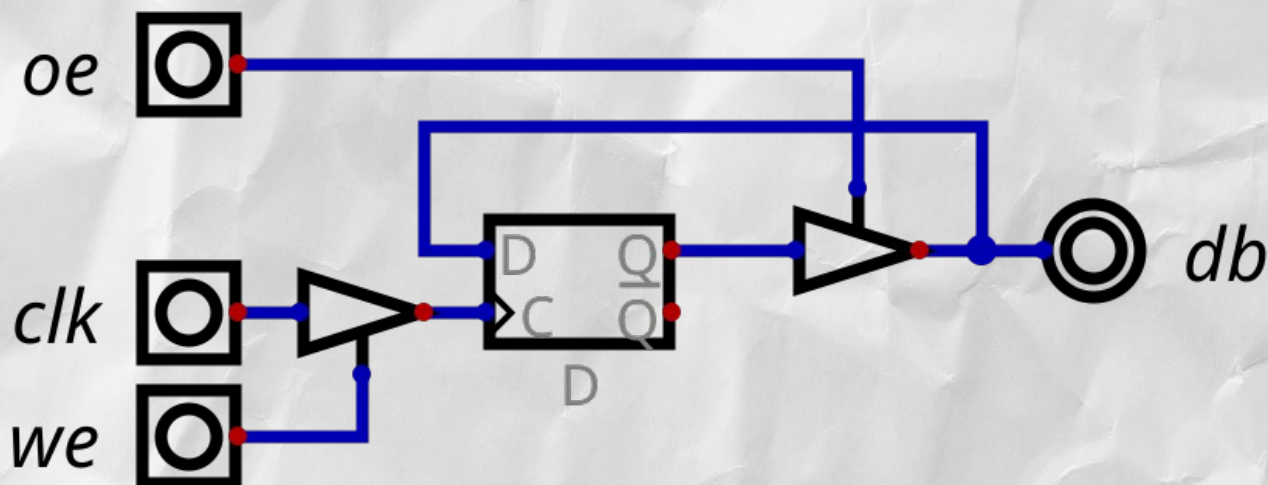
The R register is the memory refresh register. This is used for dynamic ram that requires refreshing. It is used for the lower 8 bits of addressing. After each hardware refresh the lower 7 bits are incremented while the 8th bit remains the same and can be set with the LD R, A command. An interesting use of this register is in the ZX81 computer. The R register is used as in getting the bitmaps for the characters mapped in ROM.

There is one hidden register that is used by the Z80 and not accessible to the user. It is the 16 bit WZ register. This holds the operand for 2 or 3 byte opcodes. If an opcode has a 16 bit address the Z80 can only hold one 8 bit value so for the opcode Jmp the WZ register holds the jump address while the jmp command is processing. This is also used in the Call command and in the EX HL,DE where 2 16 bit registers are exchanged.

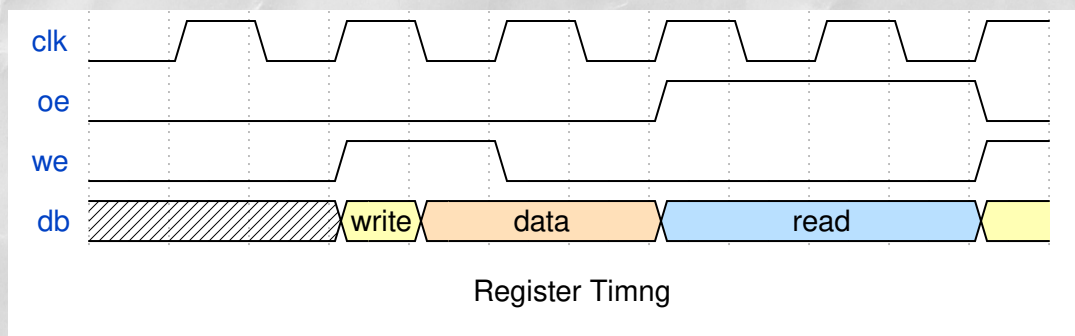
Now we will dive deeper into the logic of the registers and its control circuitry.

Register Latch

The register latch is simple a D type flip flop with the clock tied to a tristate buffer which is driven by the we (write enable) input. Although the diagram below show db (data bus) as an 8 bit output it is bidirectional. The tristate buffer driven high by oe it allows Q (output) to appear on db. When oe is driven low the data present at db is sent to d (input).



When we is high it enables the clock. When oe is low and clk goes high the data at db is loaded in the flip flop. At the next clk transition to high it will put the data on Q and if oe is set to high it will enable the buffer at Q to pass the data to db. Then if we is set low it will retain the data acting as a latch since the tristate buffer blocks the clock input.



This is the timing of the register operation. The data segment is the data that was written in during the write cycle.

Register File

The register file contains a bank of register latches that we covered in the last section. The heart of the bank is the register select circuit that controls the selection of the proper register.

Each register contains a pair of byte register latches that are tied to a low and high byte data bus. The control of each register latch is done with two three input And gates.

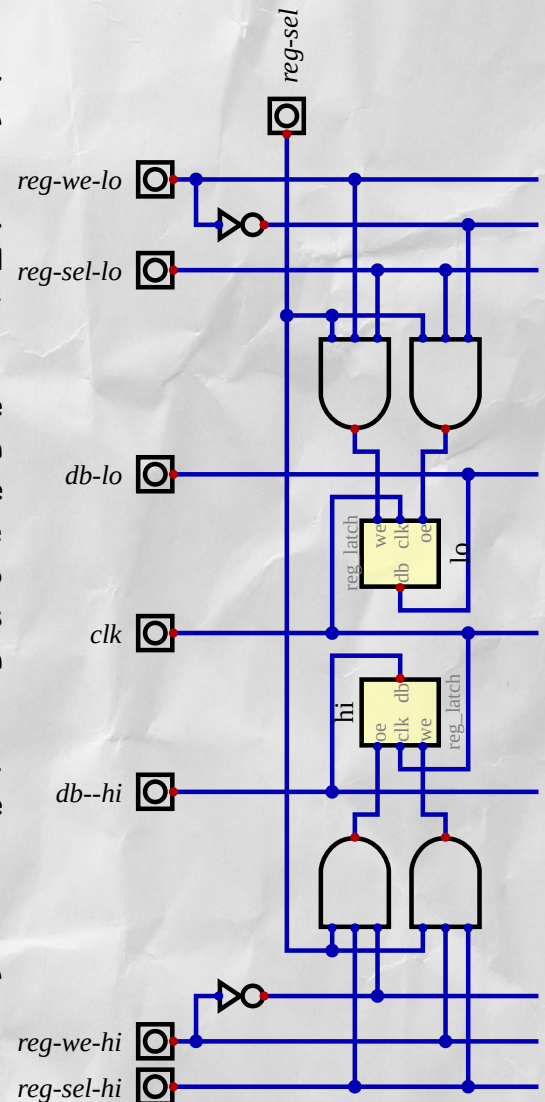
The *reg-sel* line is tied to one input of all the And gates. When it is high it enables the other control lines to control the actions of the register latch. When it is low the register is deselected.

The *reg-we-xx* line permits the data that is on the appropriate data bus to be written into the register. When high enables the And gate that controls the write enable (*we*) of the register latch. It is inverted to control the gate that controls the output enable of the register latch. So when *reg-sel* is low the data stored in the register latch is placed on the data bus and when high it stores the data on the bus into the register.

The *reg-sel-xx* line is tied to the two And gates that control either the high or low register for handling byte transfers.

The clock line clocks the flip flop in the register latch.

A truth table of the control of the high byte register latch is illustrated as shown below.



clock	reg-sel	reg-sel-hi	reg-we-hi	db-hi
0	x	x	x	x
1	0	x	x	x
1	1	0	x	x
1	1	1	0	Read
1	1	1	1	Write

Register Control Table

The data bus that is shared with all the registers in the following order

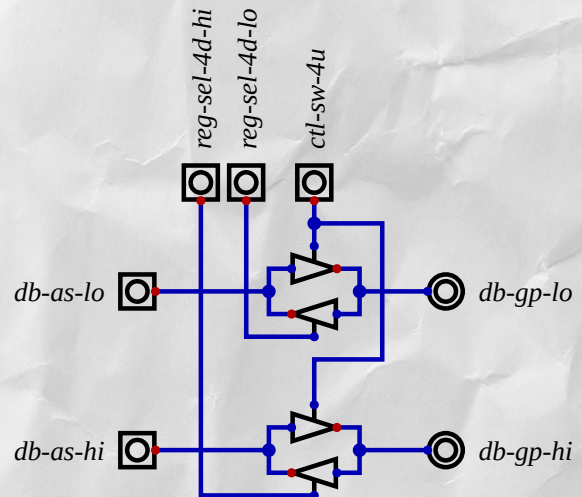
IR, PC, WZ, SP, IY, IX, HL2, HL, DE2, DE, BC2, BC, AF2, AF

Registers such as HL2 and HL are the alternate HL pairs and so on.

The IR, and PC registers handle data on the 16 bit Address bus. The other registers are connected to the 8 bit data bus. So there needs to be a way to isolate the shared data bus for addressing functions and general purpose functions. To do this the following bi-directional buffer is inserted between the PC and WZ registers.

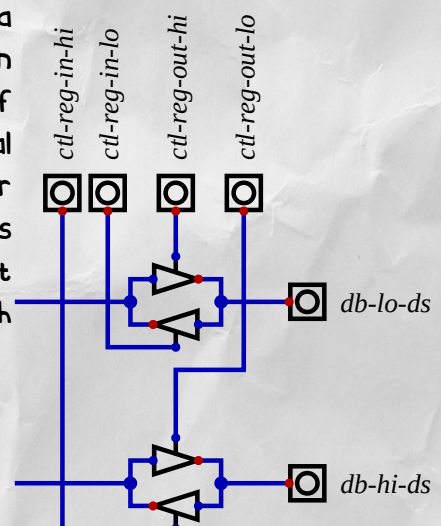
ctl-sw-4u controls the isolation. When low the IR and PC registers can access the contents of the general purpose registers on the data bus. The general purpose registers cannot access the PC or IR registers though.

When this line is low the general purpose registers and the reg-sel-4d-low is high can access the low registers of PC and IR. When reg-sel-4d-high is high the high registers are accessible. At this time the PC and IR registers cannot access the general purpose registers.



The IR, PC, and WZ registers control lines that we see in the previous circuit are controlled by the addressing system. PC is incremented by the number of bytes in the opcodes by the addressing system but can be manipulated by the general system for subroutines, jumps, branching and looping. The IR register is used for interrupt routines and the R register is incremented for refreshing which is under the addressing system control but can be manipulated by the general purpose registers. The WZ is under the address system control to pass addressing or exchange information and is temporary storage of byte registers used in the general purpose registers to make the addressing or exchange data so it cannot be isolated from them.

There is one more bidirectional buffer that isolates the data bus from the general purpose registers and is placed between the AF register and the data bus. There are two sets of bidirectional buffers for either the high byte of the general purpose registers or the low byte. The two control lines for each byte register controls the flow of the data from in (bus to register) or out (register to bus) so that it cannot conflict with operations done on the data bus from other systems such as the ALU.



Where the action is, the ALU (Arithmetic Logic Unit)

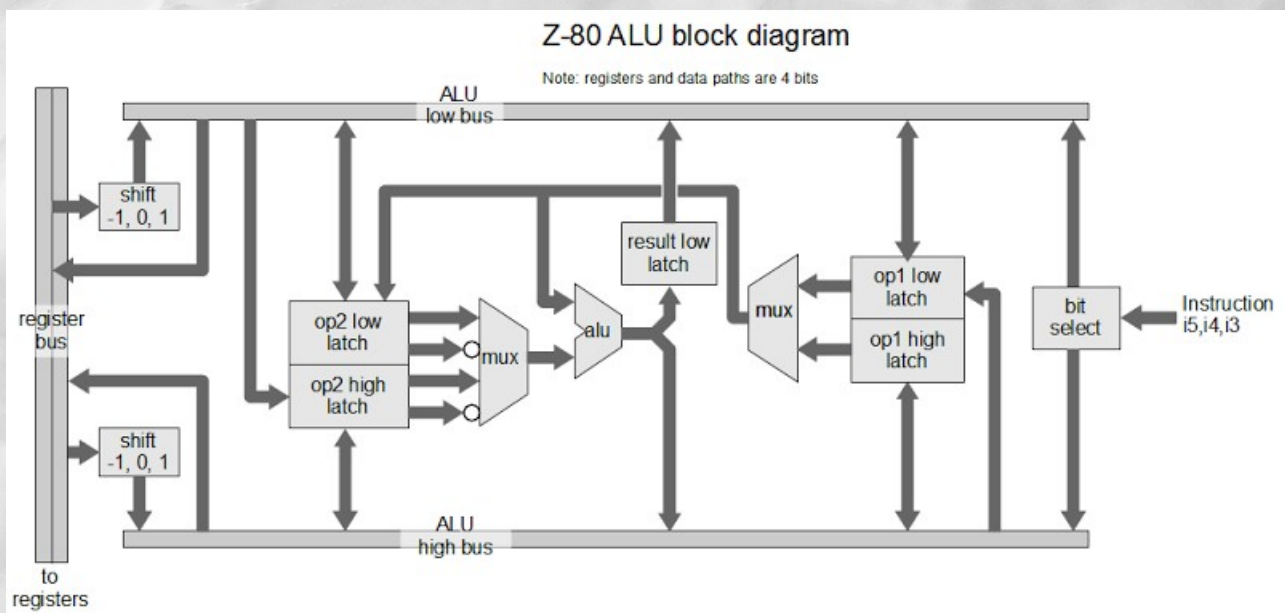
The next stop in our adventure is the heart of the Z80, the ALU. This is where all the action is.

Wikepeada states:

An arithmetic logic unit (ALU) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers.

A **combinational digital circuit** is a logic circuit that is not dependent on a clock signal to function. The results instantaneously change with changes from the inputs.

One very interesting fact is even though the Z80 is an eight bit CPU, the ALU is only four bits. This means each of the functions of the ALU must first be performed on the lower 4 bits then on the upper 4 bits.



You will notice the alu appears in the center of the diagram with support logic around it. This is because the ALU only adds, ands, ors and xors the two operands. All other functions are handled by other logic blocks.

On the left is the register bus. This bus allows the registers to directly access the ALU. The memory can also access the ALU high and low busses. One of the flags in the flag register, the Half Carry, is used in the ALU in adding to hold the carry status from the low four bit add into the high 4 bit add. Then from the register bus it goes through the shift logic. This will do a shift left one bit, right one bit or none before placing it on the high and low ALU busses. So as the data from the registers gets loaded it is shifted as needed as well.

Once the data is loaded on the ALU busses it is accessible to the op1 and op2 high and low latches. You will notice the ALU high bus can load the op1 low latch and the ALU low bus can load the op2 high latch. This enables 4 bit binary coded decimal (BCD) shifts RRD and RLD.

The muxes are multiplexers that allow the switching from low 4 bit to high 4 bit latches since the alu only performs 4 bit functions. Because of this, a result low latch needs to latch the results onto the ALU low bus so once the high bits result is placed on the ALU high bus all 8 bits are available. The op2 mux also permits the data of the op2 latch to be inverted. This permits subtraction by using 2s complement addition.

Finally the result can have each individual bit of the 8 bits to be set or cleared. The bits 3, 4, and 5 of the instruction byte designate which bit to perform the operation on.

The operation of the ALU follows the sequence below for the Add A, B command

- The AF register is placed on the register bus. The A register is placed on the op1 latches and the F register is placed on the Flag latches
- The B register is then placed on the register bus. Then B is latched into the op2 latches.
- The lower 4 bits of B is added into the lower 4 bits of A and the result is latched into the low result latch which sets the ALU low bus
- The carry from this 4 bit add is then placed into the Half Carry bit of the Flag latch.
- The upper 4 bits of B with the Half Carry bit is added to the upper 4 bits of the A register and the results are placed on the ALU high bus
- The ALU bus is placed into the accumulator latch and put on the register bus. The flag register is updated by the results.

PLA (Programmable Logic Array)



Control



Putting it all together



Making it run



A quick dip into FPGA



Making it real on a FPGA

