My 8-bit CPU Instruction Set MANUAL

https://gitlab.com/sherifnasser/my-8-bit-cpu

- **Introduction**

  The program is written in the form of series bytes and it's loaded to the RAM. The program is a list of instructions. Each instruction allocates 32 bits i.e., 4 bytes in the RAM. Each instruction stores the condition to execute it (FLAGS conditions), the op-code of instruction, and the operands. The bytes order starts from the left.

- **Registers**

  Registers are used to store data. 7 system registers are inaccessible to the user: There are also 4 available general-purpose 8-bit registers in the register file that the user can write to and read from them. Each has its code on the next page.

Register	Code	Description
Clock counter	-	Indicates the current clock cycle counts since its reset. It's used for circuitry in instructions and maybe it's not like **TSC** in x64.
PC	-	The instruction **pointer** or the program counter. Used to point to the address of the next instruction. It **increments** its value at the end of the execution of the current instruction.
IR	-	The instruction register. Holds the fetched instruction for execution. It reads from memory at the address of the **PC** value.
RL	-	Stores (temporarily) the low part of **MUL** or **DIV** result. Also, used to read the first register in basic ALU operations when 2 operands are registers.
RH	-	Stores (temporarily) the high part of **MUL** or **DIV** result.
AL	0x00	The **accumulator** register (a general-purpose register).
BL	0x01	The **base** register (a general-purpose register).
CL	0x02	The **counter** register (a general-purpose register).
DL	0x03	The **data** register (a general-purpose register).

- **FLAGS**

  FLAGS are bits in the FLAGS register that hold the state of the CPU. Each FLAG reflects the result of some arithmetic operation. FLAGS are used in conditional instructions and branching. The FLAGS register is an 8-bit register with valid 6 bits representing the FLAGS. Here is a representation of the FLAGS register starting from bit 0 as LSB. Note that bits 6 and 7 are invalid.

Bit number	FLAG	Description
0	**CF**	The Carry FLAG: indicates if some arithmetic carry is produced after an operation.
1	**PF**	The Parity FLAG: indicates if a number with even parity is produced.
2	**VF**	The Overflow FLAG: indicates if an arithmetic overflow has happened.
3	**AF**	The Auxiliary Carry FLAG: indicates if a half carry is produced after an arithmetic operation.
4	**ZF**	The Zero FLAG: indicates if the generated number is zero.
5	**SF**	The Sign FLAG: indicates if the generated number is negative or if the sign bit (MSB) of the result is 1.

- **1st byte of instruction (Most left byte)**

  This byte is decoded into 2 things: the execution condition (FLAGS condition) and the supergroup of instructions.

  - *The execution condition* means the condition code to execute that instruction. If that condition is true, the instruction will be executed. If not, the CPU will go to the next instruction. Unlike x64, this adds the flexibility to execute instructions with boilerplate jump instructions. The conditions are checked according to FLAGS (ZF, SF, VF, CF, AF, and PF) and their negated values.

  - *The supergroup code of instructions* means the group that includes subgroups of instructions. This adds the flexibility to add more instructions in the future and make decoding them easier in implementation as all subgroups could be decoded to one wire but all subgroups and all instructions of it will not be decoded (they will be like an operand to the control circuit). Currently implemented supergroups are `Supergroup 0` and `Supergroup 1`.

- **Writing the 1st byte of instruction**

  This byte is constructed from 5 bits for execution condition (MSB), and the remaining 3 bits are for the supergroup. The byte is calculated by performing bitwise **OR** between the execution condition op-code and supergroup op-code of the desired instruction. Valid conditions' op-codes and supergroups are listed in the following tables. Note that there are more possible combinations of 5 bits for conditions op-codes and more than 2 possible combinations of 3 bits for supergroup op-codes, but they are all invalid op-codes.

Supergroup op-code	**Description**
0x00	Basic instructions
0x01	CMP, TEST, and ALU instructions

Condition	**Op-code**	**FLAGS condition**	**Description**
None	0x00	**-**	Execute the instruction **unconditionally**.
Z / E	0x08	**ZF**	Execute if **Zero FLAG** is set ***or*** 2 numbers are equal after **CMP**.
NZ / NE	0x10	**~ZF**	Execute if **Zero FLAG** is cleared ***or*** 2 numbers are not equal after **CMP**.
S	0x20	**SF**	Execute if **Sign FLAG** is set ***or*** a negative number is produced after **CMP**.
NS	0x30	**~SF**	Execute if **Sign FLAG** is cleared ***or*** a non-negative number is produced after **CMP**.
G / NLE	0x40	**~(ZF | (SF ^ VF))**	Execute if (greater) > 0 ***or not*** <= 0 (**Signed**).
GE / NL	0x50	**~(SF ^ VF)**	Execute if >= 0 ***or not*** < 0 (**Signed**).
L / NGE	0x60	**SF ^ VF**	Execute if (less) < 0 ***or not*** >= 0 (**Signed**).
LE / NG	0x70	**ZF | (SF ^ VF)**	Execute if <= 0 ***or not*** >= 0 (**Signed**).
A / NBE	0x80	**~(ZF | CF)**	Execute if (above) > 0 ***or not*** <= 0 (**Unsigned**).
NC / AE / NB	0x90	**~CF**	Execute if **Carry FLAG** is cleared. ***Or*** Execute if >= 0 ***or not*** < 0 (**Unsigned**).
C / B / NAE	0xA0	**CF**	Execute if **Carry FLAG** is set. ***Or*** Execute if (below) < 0 ***or not*** >= 0 (**Unsigned**).
BE / NA	0xB0	**ZF | CF**	Execute if <= 0 ***or not*** >= 0 (**Unsigned**).
V	0xC0	**VF**	Execute if **Overflow FLAG** is set.
NV	0xD0	**~VF**	Execute if **Overflow FLAG** is cleared.
P	0xE0	**PF**	Execute if **Parity FLAG** is set.
NP	0xF0	**~PF**	Execute if **Parity FLAG** is cleared.
HC	0xE1	**AF**	Execute if **Auxiliary FLAG** is set.
NHC	0xF1	**~AF**	Execute if **Auxiliary FLAG** is cleared.

- **2nd byte of instruction**

  This byte is decoded into 2 things: the subgroup of instructions and the instruction op-code.

  - *The subgroup of instructions:* for every supergroup, there exist many subgroups, such that, each instruction in the subgroup may share the same circuitry of execution.

  - *The instruction op-code:* an instruction inside supergroup 1 in subgroup 0 may share the same circuitry as an instruction inside supergroup 1 in subgroup 1, so we could pass the decode subgroup op-code to achieve a task that is a little different from the other instruction.

- **Writing the 2nd byte of instruction**

  This byte is constructed from a nibble (4 bits) for a subgroup of instructions op-code (most significant bits) and the other one is for the instruction op-code. The byte is calculated by performing bitwise **OR** between the two nibbles. Valid subgroups' op-codes and instructions are listed in the following tables for each supergroup. Note that a supergroup may not contain all 16 possible subgroups, or a subgroup may not contain all 16 possible instructions, this depends on available instructions.

- **Supergroup 0 (Basic instructions)**

  The supergroup is for basic instructions done in the CPU. This supergroup includes only the subgroup 0x00. Note that MUL and DIV are included here because they are not pure ALU operations and require more tasks to get the final result of them. Note also that they store the low and high parts of the results in the first and second registers respectively, unlike x64 which takes one REG as an operand for the multiplier or the divisor, taking AL as the multiplicand or the dividend and restricts storing the low part of the result in AL and the high part in DL.

Op-code	x64 syntax equivalent	Description	# of CLK cycles
0x00	NOP	No operation.	1
0x01	HLT	Halt the CPU.	1
0x02	JMP Label	Jump to a specific label in the program, i.e., set the program counter (PC) to that label.	1
0x03	JMP REG	Jump to the label stored in the REG.	1
0x04	MOV REG, IMM	Store the constant value inside REG.	1
0x05	MOV REG, REG	Copy the value stored in the second REG to the first one.	1
0x06	MUL REG, REG (Not x64 syntax)	Multiply the first REG by the second one. Store the low part of the result in the first REG, and store the high part of the result in the second REG. Clear **CF** if the high part of the result is 0, otherwise, set it to 1. Other FLAGS will have undefined values.	12
0x07	DIV REG, REG (Not x64 syntax)	Divide the first REG by the second one. Store the low part of the result (quotient) in the first REG, and store the high part of the result (remainder) in the second REG. Clear **CF** if the high part of the result is 0, otherwise, set it to 1. Other FLAGS will have undefined values.	12

- **Supergroup 1 (ALU instructions)**

  This includes ALU operations. It includes 2 subgroups of instructions. Subgroup 0 includes ALU operations of 2 register operands, CMP, and TEST instructions. Subgroup 1 includes ALU operations for a register and immediate (constant) value and other unary operators. Note shared ALU operations between two subgroups, these operations have the same ALU op-code (the right nibble). This helps in decoding, as the right nibble of the op-code will passed to ALU. Note also there are NOR and XNOR instructions, and shift instructions don’t restrict the second register to be CL like x64. Note also the MIRROR instruction which is like RBIT in ARM architecture.

Op-code	x64 syntax equivalent	Description	# of CLK cycles
0x00	AND REG, REG	Perform bitwise **AND** between the values of the first and second registers, and store the result in the first register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	2
0x01	ANDN REG, REG (NAND)	Perform bitwise **NAND** between the values of the first and second registers, and store the result in the first register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	2
0x02	OR REG, REG	Perform bitwise **OR** between the values of the first and second registers, and store the result in the first register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	2
0x03	NOR REG, REG (Not x64 syntax)	Perform bitwise **NOR** between the values of the first and second registers, and store the result in the first register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	2
0x04	XOR REG, REG	Perform bitwise **XOR** between the values of the first and second registers, and store the result in the first register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	2
0x05	XNOR REG, REG (Not x64 syntax)	Perform bitwise **XNOR** between the values of the first and second registers, and store the result in the first register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	2
0x06	ADD REG, REG	**Add** the first register's value to the second register's value, and store the result in the first register. **FLAGS** are set according to the result.	2
0x07	SUB REG, REG	**Subtract** the second register's value from the first register's value, and store the result in the first register. **FLAGS** are set according to the result.	2
0x08	CMP REG, IMM	**Compare** the register's value to the immediate (constant) value by **subtracting** the immediate value from the register value but not storing anything like the **SUB** instruction. **FLAGS** are set according to the result.	1
0x09	CMP REG, REG	**Compare** the first register's value to the second register's value by **subtracting** the second register from the first register but not storing anything like the **SUB** instruction. **FLAGS** are set according to the result.	2

0x0A	TEST REG, IMM	**Compare** the register's value to the immediate (constant) value by performing bitwise **AND** between the two values but not storing anything like the **AND** instruction. **FLAGS** are set according to the result.	1
0x0B	TEST REG, REG	**Compare** the first register's value to the second register's value by performing bitwise **AND** between the two values but not storing anything like the **AND** instruction. **FLAGS** are set according to the result.	2
0x0C	SHR REG, REG	**Shift to the right** the value of the first register by the value of the second register. **CF** contains the value of the last shifted bit out. It’s undefined where the count exceeds or equals 8 (register size in bits). **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	2
0x0D	SHL / SAL REG, REG	**Shift to the left** the value of the first register by the value of the second register. **CF** contains the value of the last shifted bit out. It’s undefined where the count exceeds or equals 8 (register size in bits). **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	2
0x0E	SAR REG, REG	**(Arithmetic) Shift to the right (sign extending)** the value of the first register by the value of the second register. **CF** contains the value of the last shifted bit out. It’s undefined where the count exceeds or equals 8 (register size in bits). **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	2
0x10	AND REG, IMM	Perform bitwise **AND** between the values of the register and the immediate, and store the result in the register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	1
0x11	ANDN REG, IMM (NAND)	Perform bitwise **NAND** between the values of the register and the immediate, and store the result in the register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	1

0x12	OR REG, IMM	Perform bitwise **OR** between the values of the register and the immediate, and store the result in the register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	1
0x13	NOR REG, IMM (Not x64 syntax)	Perform bitwise **NOR** between the values of the register and the immediate, and store the result in the register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	1
0x14	XOR REG, IMM	Perform bitwise **XOR** between the values of the register and the immediate, and store the result in the register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	1
0x15	XNOR REG, IMM (Not x64 syntax)	Perform bitwise **XNOR** between the values of the register and the immediate, and store the result in the register. **CF** and **VF** are cleared. **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	1
0x16	ADD REG, REG	**Add** the register's value to the immediate, and store the result in the register. **FLAGS** are set according to the result.	1
0x17	SUB REG, REG	**Subtract** the register's value to the immediate, and store the result in the register. **FLAGS** are set according to the result.	1
0x18	NOT REG	Perform **one's** complement negation (Bitwise **NOT**) on the register value. None of the **FLAGS** are affected.	1
0x19	NEG REG	Perform **two's** complement negation (Subtract from zero) on the register value. **CF** is cleared if the register's value is zero, otherwise it is set. Other **FLAGS** are set according to the result.	1
0x1A	INC REG	**Add** 1 to the register. This differs from the (**ADD** REG, 1), as **CF** is not affected here. Other **FLAGS** are set according to the result.	1
0x1B	DEC REG	**Subtract** 1 from the register. This differs from the (**SUB** REG, 1), as **CF** is not affected here. Other **FLAGS** are set according to the result.	1

0x1C	SHR REG, IMM	**Shift to the right** the value of the register by the value of the immediate. **CF** contains the value of the last shifted bit out. It's undefined where the count exceeds or equals 8 (register size in bits). **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	1
0x1D	SHL / SAL REG, IMM	**Shift to the left** the value of the register by the value of the immediate. **CF** contains the value of the last shifted bit out. It's undefined where the count exceeds or equals 8 (register size in bits). **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	1
0x1E	SAR REG, IMM	**(Arithmetic) Shift to the right (sign extending)** the value of the register by the value of the immediate. **CF** contains the value of the last shifted bit out. It's undefined where the count exceeds or equals 8 (register size in bits). **SF**, **ZF**, and **PF** are set according to the result. **AF** will have an undefined value.	1
0x1F	MIRROR REG (Not x64 syntax)	Reverse bits' order of the register's value.	1