NAME

gbz80 — Game Boy CPU instruction reference

DESCRIPTION

This is the list of instructions supported by *rgbasm*(1), including a short description, the number of bytes needed to encode them and the number of CPU cycles at 1MHz (or 2MHz in GBC double speed mode) needed to complete them.

Instructons are documented according to the syntax accepted by *rgbasm*(1), which does not always match one-to-one with the way instructions are *encoded*: https://gbdev.io/gb-opcodes/optables/.

Note: All arithmetic and logic instructions that use register **A** as a destination can omit the destination, since it is assumed to be register **A** by default. So the following two lines have the same effect:

```
OR A,B
OR B
```

Furthermore, the **CPL** instruction can take an optional **A** destination, since it can only be register **A**. So the following two lines have the same effect:

```
CPL
CPL A
```

LEGEND

List of abbreviations used in this document.

`r8` Any of the 8-bit registers (**A**, **B**, **C**, **D**, **E**, **H**, **L**).

`r16` Any of the general-purpose 16-bit registers (**BC**, **DE**, **HL**).

`n8` 8-bit integer constant (signed or unsigned, **-128** to **255**).

`n16` 16-bit integer constant (signed or unsigned, **-32768** to **65535**).

`e8` 8-bit signed offset (**-128** to **127**).

`u3` 3-bit unsigned bit index (**0** to **7**, with **0** as the least significant bit).

`cc` A condition code:

- **Z** Execute if Z is set.
- **NZ** Execute if Z is not set.
- **C** Execute if C is set.
- **NC** Execute if C is not set.

`vec` An **RST** vector (*0x00*, *0x08*, *0x10*, *0x18*, *0x20*, *0x28*, *0x30*, and *0x38*).

INSTRUCTION OVERVIEW

Load instructions

“LD r8,r8”
“LD r8,n8”
“LD r16,n16”
“LD [HL],r8”
“LD [HL],n8”
“LD r8,[HL]”
“LD [r16],A”
“LD [n16],A”
“LDH [n16],A”
“LDH [C],A”
“LD A,[r16]”
“LD A,[n16]”

"LDH A,[n16]"
"LDH A,[C]"
"LD [HLI],A"
"LD [HLD],A"
"LD A,[HLI]"
"LD A,[HLD]"

8-bit arithmetic instructions

"ADC A,r8"
"ADC A,[HL]"
"ADC A,n8"
"ADD A,r8"
"ADD A,[HL]"
"ADD A,n8"
"CP A,r8"
"CP A,[HL]"
"CP A,n8"
"DEC r8"
"DEC [HL]"
"INC r8"
"INC [HL]"
"SBC A,r8"
"SBC A,[HL]"
"SBC A,n8"
"SUB A,r8"
"SUB A,[HL]"
"SUB A,n8"

16-bit arithmetic instructions

"ADD HL,r16"
"DEC r16"
"INC r16"

Bitwise logic instructions

"AND A,r8"
"AND A,[HL]"
"AND A,n8"
"CPL"
"OR A,r8"
"OR A,[HL]"
"OR A,n8"
"XOR A,r8"
"XOR A,[HL]"
"XOR A,n8"

Bit flag instructions

"BIT u3,r8"
"BIT u3,[HL]"
"RES u3,r8"
"RES u3,[HL]"
"SET u3,r8"
"SET u3,[HL]"

Bit shift instructions

"RL r8"

“RL [HL]”
“RLA”
“RLC r8”
“RLC [HL]”
“RLCA”
“RR r8”
“RR [HL]”
“RRA”
“RRC r8”
“RRC [HL]”
“RRCA”
“SLA r8”
“SLA [HL]”
“SRA r8”
“SRA [HL]”
“SRL r8”
“SRL [HL]”
“SWAP r8”
“SWAP [HL]”

Jumps and subroutine instructions
“CALL n16”
“CALL cc,n16”
“JP HL”
“JP n16”
“JP cc,n16”
“JR n16”
“JR cc,n16”
“RET cc”
“RET”
“RETI”
“RST vec”

Carry flag instructions
“CCF”
“SCF”

Stack manipulation instructions
“ADD HL,SP”
“ADD SP,e8”
“DEC SP”
“INC SP”
“LD SP,n16”
“LD [n16],SP”
“LD HL,SP+e8”
“LD SP,HL”
“POP AF”
“POP r16”
“PUSH AF”
“PUSH r16”

Interrupt-related instructions
“DI”
“EI”

"HALT"

Miscellaneous instructions

"DAA"
"NOP"
"STOP"

INSTRUCTION REFERENCE

ADC A,r8

Add the value in *r8* plus the carry flag to **A**.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 0

H Set if overflow from bit 3.

C Set if overflow from bit 7.

ADC A,[HL]

Add the byte pointed to by **HL** plus the carry flag to **A**.

Cycles: 2

Bytes: 1

Flags: See "ADC A,r8"

ADC A,n8

Add the value *n8* plus the carry flag to **A**.

Cycles: 2

Bytes: 2

Flags: See "ADC A,r8"

ADD A,r8

Add the value in *r8* to **A**.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 0

H Set if overflow from bit 3.

C Set if overflow from bit 7.

ADD A,[HL]

Add the byte pointed to by **HL** to **A**.

Cycles: 2

Bytes: 1

Flags: See "ADD A,r8"

ADD A,n8

Add the value *n8* to **A**.

Cycles: 2

Bytes: 2

Flags: See “ADD A,r8”

ADD HL,r16

Add the value in *r16* to **HL**.

Cycles: 2

Bytes: 1

Flags:

N 0

H Set if overflow from bit 11.

C Set if overflow from bit 15.

ADD HL,SP

Add the value in **SP** to **HL**.

Cycles: 2

Bytes: 1

Flags: See “ADD HL,r16”

ADD SP,e8

Add the signed value *e8* to **SP**.

Cycles: 4

Bytes: 2

Flags:

Z 0

N 0

H Set if overflow from bit 3.

C Set if overflow from bit 7.

AND A,r8

Set **A** to the bitwise AND between the value in *r8* and **A**.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 0

H 1

C 0

AND A,[HL]

Set **A** to the bitwise AND between the byte pointed to by **HL** and **A**.

Cycles: 2

Bytes: 1

Flags: See “AND A,r8”

AND A,n8

Set **A** to the bitwise AND between the value *n8* and **A**.

Cycles: 2

Bytes: 2

Flags: See “AND A,r8”

BIT u3,r8

Test bit *u3* in register *r8*, set the zero flag if bit not set.

Cycles: 2

Bytes: 2

Flags:

Z Set if the selected bit is 0.

N 0

H 1

BIT u3,[HL]

Test bit *u3* in the byte pointed by **HL**, set the zero flag if bit not set.

Cycles: 3

Bytes: 2

Flags: See “BIT u3,r8”

CALL n16

Call address *n16*.

This pushes the address of the instruction after the **CALL** on the stack, such that “RET” can pop it later; then, it executes an implicit “JP n16”.

Cycles: 6

Bytes: 3

Flags: None affected.

CALL cc,n16

Call address *n16* if condition *cc* is met.

Cycles: 6 taken / 3 untaken

Bytes: 3

Flags: None affected.

CCF

Complement Carry Flag.

Cycles: 1

Bytes: 1

Flags:

N 0

H 0

C Inverted.

CP A,r8

ComPare the value in **A** with the value in *r8*.

This subtracts the value in *r8* from **A** and sets flags accordingly, but discards the result.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 1

H Set if borrow from bit 4.

C Set if borrow (i.e. if *r8* > **A**).

CP A,[HL]

ComPare the value in **A** with the byte pointed to by **HL**.

This subtracts the byte pointed to by **HL** from **A** and sets flags accordingly, but discards the result.

Cycles: 2

Bytes: 1

Flags: See "CP A,r8"

CP A,n8

ComPare the value in **A** with the value *n8*.

This subtracts the value *n8* from **A** and sets flags accordingly, but discards the result.

Cycles: 2

Bytes: 2

Flags: See "CP A,r8"

CPL

ComPLement accumulator (**A** = **~A**); also called bitwise NOT.

Cycles: 1

Bytes: 1

Flags:

N 1

H 1

DAA

Decimal Adjust Accumulator.

Designed to be used after performing an arithmetic instruction (**ADD**, **ADC**, **SUB**, **SBC**) whose inputs were in Binary-Coded Decimal (BCD), adjusting the result to likewise be in BCD.

The exact behavior of this instruction depends on the state of the subtract flag **N**:

If the subtract flag **N** is set:

1. Initialize the adjustment to 0.
2. If the half-carry flag **H** is set, then add *$6* to the adjustment.
3. If the carry flag is set, then add *$60* to the adjustment.

4. Subtract the adjustment from **A**.

If the subtract flag **N** is not set:

1. Initialize the adjustment to 0.
2. If the half-carry flag **H** is set or **A** & *$F* > *$9*, then add *$6* to the adjustment.
3. If the carry flag is set or **A** > *$99*, then add *$60* to the adjustment and set the carry flag.
4. Add the adjustment to **A**.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

H 0

C Set or unaffected depending on the operation.

DEC r8

Decrement the value in register `r8` by 1.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 1

H Set if borrow from bit 4.

DEC [HL]

Decrement the byte pointed to by **HL** by 1.

Cycles: 3

Bytes: 1

Flags: See “DEC r8”

DEC r16

Decrement the value in register `r16` by 1.

Cycles: 2

Bytes: 1

Flags: None affected.

DEC SP

Decrement the value in register **SP** by 1.

Cycles: 2

Bytes: 1

Flags: None affected.

DI

Disable Interrupts by clearing the **IME** flag.

Cycles: 1

Bytes: 1

Flags: None affected.

EI

Enable Interrupts by setting the **IME** flag.

The flag is only set *after* the instruction following **EI**.

Cycles: 1

Bytes: 1

Flags: None affected.

HALT

Enter CPU low-power consumption mode until an interrupt occurs.

The exact behavior of this instruction depends on the state of the **IME** flag, and whether interrupts are pending (i.e. whether `[IE] & [IF]` is non-zero):

If the **IME** flag is set:
: The CPU enters low-power mode until *after* an interrupt is about to be serviced. The handler is executed normally, and the CPU resumes execution after the **`HALT`** when that returns.

If the **IME** flag is not set, and no interrupts are pending:
: As soon as an interrupt becomes pending, the CPU resumes execution. This is like the above, except that the handler is *not* called.

If the **IME** flag is not set, and some interrupt is pending:
: The CPU continues execution after the **`HALT`**, but the byte after it is read twice in a row (**PC** is not incremented, due to a hardware bug).

Cycles: -

Bytes: 1

Flags: None affected.

INC r8

Increment the value in register *`r8`* by 1.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 0

H Set if overflow from bit 3.

INC [HL]

Increment the byte pointed to by **HL** by 1.

Cycles: 3

Bytes: 1

Flags: See “INC r8”

INC r16

Increment the value in register *`r16`* by 1.

Cycles: 2

Bytes: 1

Flags: None affected.

INC SP

Increment the value in register **SP** by 1.

Cycles: 2

Bytes: 1

Flags: None affected.

JP n16

Jump to address *n16*; effectively, copy *n16* into **PC**.

Cycles: 4

Bytes: 3

Flags: None affected.

JP cc,n16

Jump to address *n16* if condition *cc* is met.

Cycles: 4 taken / 3 untaken

Bytes: 3

Flags: None affected.

JP HL

Jump to address in **HL**; effectively, copy the value in register **HL** into **PC**.

Cycles: 1

Bytes: 1

Flags: None affected.

JR n16

Relative Jump to address *n16*.

The target address *n16* is *encoded* as a signed 8-bit offset from the address immediately following the **JR** instruction, so it must be between **-128** and **127** bytes away. For example:

```
    JR Label  ; no-op; encoded offset of 0
Label:
    JR Label  ; infinite loop; encoded offset of -2
```

Cycles: 3

Bytes: 2

Flags: None affected.

JR cc,n16

Relative Jump to address *n16* if condition *cc* is met.

The target address *n16* is *encoded* as a signed 8-bit offset from the address immediately following the **JR** instruction, so it must be between **-128** and **127** bytes away.

Cycles: 3 taken / 2 untaken

Bytes: 2

Flags: None affected.

LD r8,r8

Copy (aka Load) the value in register on the right into the register on the left.

Storing a register into itself is a no-op; however, some Game Boy emulators interpret **LD B,B** as a break-point, or **LD D,D** as a debug message (such as *BGB*: https://bgb.bircd.org/manual.html#expressions).

Cycles: 1

Bytes: 1

Flags: None affected.

LD r8,n8

Copy the value *n8* into register *r8*.

Cycles: 2

Bytes: 2

Flags: None affected.

LD r16,n16

Copy the value *n16* into register *r16*.

Cycles: 3

Bytes: 3

Flags: None affected.

LD [HL],r8

Copy the value in register *r8* into the byte pointed to by **HL**.

Cycles: 2

Bytes: 1

Flags: None affected.

LD [HL],n8

Copy the value *n8* into the byte pointed to by **HL**.

Cycles: 3

Bytes: 2

Flags: None affected.

LD r8,[HL]

Copy the value pointed to by **HL** into register *r8*.

Cycles: 2

Bytes: 1

Flags: None affected.

LD [r16],A

Copy the value in register **A** into the byte pointed to by *r16*.

Cycles: 2

Bytes: 1

Flags: None affected.

LD [n16],A

Copy the value in register **A** into the byte at address *n16*.

Cycles: 4

Bytes: 3

Flags: None affected.

LDH [n16],A

Copy the value in register **A** into the byte at address *n16*.

The destination address `n16` is *encoded* as its 8-bit low byte and assumes a high byte of *$FF*, so it must be between *$FF00* and *$FFFF*.

Cycles: 3

Bytes: 2

Flags: None affected.

LDH [C],A

Copy the value in register **A** into the byte at address *$FF00+C*.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD [$FF00+C],A`.

LD A,[r16]

Copy the byte pointed to by `r16` into register **A**.

Cycles: 2

Bytes: 1

Flags: None affected.

LD A,[n16]

Copy the byte at address `n16` into register **A**.

Cycles: 4

Bytes: 3

Flags: None affected.

LDH A,[n16]

Copy the byte at address `n16` into register **A**.

The source address `n16` is *encoded* as its 8-bit low byte and assumes a high byte of *$FF*, so it must be between *$FF00* and *$FFFF*.

Cycles: 3

Bytes: 2

Flags: None affected.

LDH A,[C]

Copy the byte at address *$FF00+C* into register **A**.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD A,[$FF00+C]`.

LD [HLI],A

Copy the value in register **A** into the byte pointed by **HL** and increment **HL** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD [HL+],A`, or `LDI [HL],A`.

LD [HLD],A

Copy the value in register **A** into the byte pointed by **HL** and decrement **HL** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD [HL-],A`, or `LDD [HL],A`.

LD A,[HLD]

Copy the byte pointed to by **HL** into register **A**, and decrement **HL** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD A,[HL-]`, or `LDD A,[HL]`.

LD A,[HLI]

Copy the byte pointed to by **HL** into register **A**, and increment **HL** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD A,[HL+]`, or `LDI A,[HL]`.

LD SP,n16

Copy the value *n16* into register **SP**.

Cycles: 3

Bytes: 3

Flags: None affected.

LD [n16],SP

Copy **SP** & *$FF* at address *n16* and **SP** >> 8 at address *n16* + 1.

Cycles: 5

Bytes: 3

Flags: None affected.

LD HL,SP+e8

Add the signed value *e8* to **SP** and copy the result in **HL**.

Cycles: 3

Bytes: 2

Flags:

Z 0

N 0

H Set if overflow from bit 3.

C Set if overflow from bit 7.

LD SP,HL

Copy register **HL** into register **SP**.

Cycles: 2

Bytes: 1

Flags: None affected.

NOP

No OPeration.

Cycles: 1

Bytes: 1

Flags: None affected.

OR A,r8

Set **A** to the bitwise OR between the value in *r8* and **A**.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 0

H 0

C 0

OR A,[HL]

Set **A** to the bitwise OR between the byte pointed to by **HL** and **A**.

Cycles: 2

Bytes: 1

Flags: See “OR A,r8”

OR A,n8

Set **A** to the bitwise OR between the value *n8* and **A**.

Cycles: 2

Bytes: 2

Flags: See “OR A,r8”

POP AF

Pop register **AF** from the stack. This is roughly equivalent to the following *imaginary* instructions:

```
LD F, [SP]  ; See below for individual flags
INC SP
LD A, [SP]
INC SP
```

Cycles: 3

Bytes: 1

Flags:

Z Set from bit 7 of the popped low byte.

N Set from bit 6 of the popped low byte.

H Set from bit 5 of the popped low byte.

C Set from bit 4 of the popped low byte.

POP r16

Pop register *r16* from the stack. This is roughly equivalent to the following *imaginary* instructions:

```
LD LOW(r16), [SP]   ; C, E or L
INC SP
LD HIGH(r16), [SP]  ; B, D or H
INC SP
```

Cycles: 3

Bytes: 1

Flags: None affected.

PUSH AF

Push register **AF** into the stack. This is roughly equivalent to the following *imaginary* instructions:

```
DEC SP
LD [SP], A
DEC SP
LD [SP], F.Z << 7 | F.N << 6 | F.H << 5 | F.C << 4
```

Cycles: 4

Bytes: 1

Flags: None affected.

PUSH r16

Push register *r16* into the stack. This is roughly equivalent to the following *imaginary* instructions:

```
DEC SP
LD [SP], HIGH(r16)  ; B, D or H
DEC SP
LD [SP], LOW(r16)   ; C, E or L
```

Cycles: 4

Bytes: 1

Flags: None affected.

RES u3,r8

Set bit *u3* in register *r8* to 0. Bit 0 is the rightmost one, bit 7 the leftmost one.

Cycles: 2

Bytes: 2

Flags: None affected.

RES u3,[HL]

Set bit *u3* in the byte pointed by **HL** to 0. Bit 0 is the rightmost one, bit 7 the leftmost one.

Cycles: 4

Bytes: 2

Flags: None affected.

RET

Return from subroutine. This is basically a **POP PC** (if such an instruction existed). See “POP r16” for an explanation of how **POP** works.

Cycles: 4

Bytes: 1

Flags: None affected.

RET cc

Return from subroutine if condition *cc* is met.

Cycles: 5 taken / 2 untaken

Bytes: 1

Flags: None affected.

RETI

Return from subroutine and enable interrupts. This is basically equivalent to executing “EI” then “RET”, meaning that **IME** is set right after this instruction.

Cycles: 4

Bytes: 1

Flags: None affected.

RL r8

Rotate bits in register *r8* left, through the carry flag.

```
  ââ Flags ââ ââââââââ r8 âââââââ
ââââ   C   âââââ b7 â ... â b0 ââââ
â âââââââââââ ââââââââââââââââââââ â
âââââââââââââââââââââââââââââââââââ
```

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

RL [HL]

Rotate the byte pointed to by **HL** left, through the carry flag.

```
  ââ Flags ââ âââââââ [HL] ââââââ
ââââ   C   âââââ b7 â ... â b0 ââââ
â âââââââââââ ââââââââââââââââââââ â
âââââââââââââââââââââââââââââââââââ
```

Cycles: 4

Bytes: 2

Flags: See “RL r8”

RLA

Rotate register **A** left, through the carry flag.

```
  ââ Flags ââ ââââââââ A ââââââââ
ââââ   C   âââââ b7 â ... â b0 ââââ
â âââââââââââ ââââââââââââââââââââ â
âââââââââââââââââââââââââââââââââââ
```

Cycles: 1

Bytes: 1

Flags:

Z 0

N 0

H 0

C Set according to result.

RLC r8

Rotate register *r8* left.

```
ââ Flags ââ   ââââââââ r8 âââââââ
â    C   ââââ¬âââ b7 â ... â b0 ââââ
âââââââââââ â ââââââââââââââââââââ â
            ââââââââââââââââââââââââ
```

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

RLC [HL]

Rotate the byte pointed to by **HL** left.

```
ââ Flags ââ   âââââââ [HL] ââââââ
â    C   ââââ¬âââ b7 â ... â b0 ââââ
âââââââââââ â ââââââââââââââââââââ â
            ââââââââââââââââââââââââ
```

Cycles: 4

Bytes: 2

Flags: See “RLC r8”

RLCA

Rotate register **A** left.

```
ââ Flags ââ   ââââââââ A ââââââââ
â    C   ââââ¬âââ b7 â ... â b0 ââââ
âââââââââââ â ââââââââââââââââââââ â
            ââââââââââââââââââââââââ
```

Cycles: 1

Bytes: 1

Flags:

Z 0

N 0

H 0

C Set according to result.

RR r8

Rotate register *r8* right, through the carry flag.

```
  ââââââââ r8 âââââââ ââ Flags ââ
ââââ b7 â ... â b0 âââââ   C   ââââ
â ââââââââââââââââââââ ââââââââââ â
ââââââââââââââââââââââââââââââââââââ
```

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

RR [HL]

Rotate the byte pointed to by **HL** right, through the carry flag.

```
  âââââââ [HL] ââââââ ââ Flags ââ
ââââ b7 â ... â b0 âââââ   C   ââââ
â ââââââââââââââââââââ ââââââââââ â
ââââââââââââââââââââââââââââââââââââ
```

Cycles: 4

Bytes: 2

Flags: See “RR r8”

RRA

Rotate register **A** right, through the carry flag.

```
  ââââââââ A ââââââââ ââ Flags ââ
ââââ b7 â ... â b0 âââââ   C   ââââ
â ââââââââââââââââââââ ââââââââââ â
ââââââââââââââââââââââââââââââââââââ
```

Cycles: 1

Bytes: 1

Flags:

Z 0

N 0

H 0

C Set according to result.

RRC r8

Rotate register *r8* right.

```
  ââââââââ r8 âââââââ   ââ Flags ââ
ââââ b7 â ... â b0 ââââ¬âââ   C    â
â ââââââââââââââââââââ â ââââââââââ
ââââââââââââââââââââââââ
```

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

RRC [HL]

Rotate the byte pointed to by **HL** right.

```
  âââââââ [HL] ââââââ   ââ Flags ââ
ââââ b7 â ... â b0 ââââ¬âââ   C    â
â âââââââââââââââââââ â ââââââââââ
âââââââââââââââââââââââ
```

Cycles: 4

Bytes: 2

Flags: See “RRC r8”

RRCA

Rotate register **A** right.

```
  ââââââââ A ââââââââ   ââ Flags ââ
ââââ b7 â ... â b0 ââââ¬âââ   C    â
â âââââââââââââââââââ â ââââââââââ
âââââââââââââââââââââââ
```

Cycles: 1

Bytes: 1

Flags:

Z 0

N 0

H 0

C Set according to result.

RST vec

Call address *vec*. This is a shorter and faster equivalent to “CALL” for suitable values of *vec*.

Cycles: 4

Bytes: 1

Flags: None affected.

SBC A,r8

Subtract the value in *r8* and the carry flag from **A**.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 1

H Set if borrow from bit 4.

C Set if borrow (i.e. if (*r8* + carry) > **A**).

SBC A,[HL]

Subtract the byte pointed to by **HL** and the carry flag from **A**.

Cycles: 2

Bytes: 1

Flags: See “SBC A,r8”

SBC A,n8

Subtract the value *n8* and the carry flag from **A**.

Cycles: 2

Bytes: 2

Flags: See “SBC A,r8”

SCF

Set Carry Flag.

Cycles: 1

Bytes: 1

Flags:

N 0

H 0

C 1

SET u3,r8

Set bit *u3* in register *r8* to 1. Bit 0 is the rightmost one, bit 7 the leftmost one.

Cycles: 2

Bytes: 2

Flags: None affected.

SET u3,[HL]

Set bit *u3* in the byte pointed by **HL** to 1. Bit 0 is the rightmost one, bit 7 the leftmost one.

Cycles: 4

Bytes: 2

Flags: None affected.

SLA r8

Shift Left Arithmetically register *r8*.

```
ââ Flags ââ âââââââââ r8 âââââââ
â    C   âââââ b7 â ... â b0 âââ 0
ââââââââââââ ââââââââââââââââââââ
```

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

SLA [HL]

Shift Left Arithmetically the byte pointed to by **HL**.

```
ââ Flags ââ âââââââ [HL] ââââââ
â    C   âââââ b7 â ... â b0 âââ 0
âââââââââââ ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ
```

Cycles: 4

Bytes: 2

Flags: See “SLA r8”

SRA r8

Shift Right Arithmetically register *r8* (bit 7 of *r8* is unchanged).

```
âââââââ r8 âââââââ ââ Flags ââ
â b7 â ... â b0 âââââ   C    â
ââââââââââââââââââââ âââââââââââ
```

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

SRA [HL]

Shift Right Arithmetically the byte pointed to by **HL** (bit 7 of the byte pointed to by **HL** is unchanged).

```
ââââââ [HL] ââââââ ââ Flags ââ
â b7 â ... â b0 âââââ   C    â
ââââââââââââââââââââ âââââââââââ
```

Cycles: 4

Bytes: 2

Flags: See “SRA r8”

SRL r8

Shift Right Logically register *r8*.

```
   ââââââââ r8 âââââââ ââ Flags ââ
0 âââ b7 â ... â b0 âââââ   C    â
   âââââââââââââââââââ âââââââââââ
```

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

SRL [HL]

Shift Right Logically the byte pointed to by **HL**.

```
   ââââââââ [HL] ââââââââ ââ Flags ââ
0 âââ b7 â ... â b0 âââââ   C    â
   ââââââââââââââââââââââââ ââââââââââââ
```

Cycles: 4

Bytes: 2

Flags: See “SRL r8”

STOP

Enter CPU very low power mode. Also used to switch between GBC double speed and normal speed CPU modes.

The exact behavior of this instruction is fragile and may interpret its second byte as a separate instruction (see *the Pan Docs*: https://gbdev.io/pandocs/Reducing_Power_Consumption.html#using-the-stop-instruction), which is why *rgbasm*(1) allows explicitly specifying the second byte (**STOP** *n8*) to override the default of *$00* (a **NOP** instruction).

Cycles: -

Bytes: 2

Flags: None affected.

SUB A,r8

Subtract the value in *r8* from **A**.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 1

H Set if borrow from bit 4.

C Set if borrow (i.e. if *r8* > **A**).

SUB A,[HL]

Subtract the byte pointed to by **HL** from **A**.

Cycles: 2

Bytes: 1

Flags: See “SUB A,r8”

SUB A,n8

Subtract the value *n8* from **A**.

Cycles: 2

Bytes: 2

Flags: See “SUB A,r8”

SWAP r8

Swap the upper 4 bits in register *r8* and the lower 4 ones.

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C 0

SWAP [HL]

Swap the upper 4 bits in the byte pointed by **HL** and the lower 4 ones.

Cycles: 4

Bytes: 2

Flags: See “SWAP r8”

XOR A,r8

Set **A** to the bitwise XOR between the value in *r8* and **A**.

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 0

H 0

C 0

XOR A,[HL]

Set **A** to the bitwise XOR between the byte pointed to by **HL** and **A**.

Cycles: 2

Bytes: 1

Flags: See “XOR A,r8”

XOR A,n8

Set **A** to the bitwise XOR between the value *n8* and **A**.

Cycles: 2

Bytes: 2

Flags: See “XOR A,r8”

SEE ALSO

rgbasm(1), *rgblink*(1), *rgbfix*(1), *rgbgfx*(1), *rgbasm-old*(5), *rgbds*(7)

HISTORY

rgbasm(1) was originally written by Carsten Sørensen as part of the ASMotor package, and was later repackaged in RGBDS by Justin Lloyd. It is now maintained by a number of contributors at https://github.com/gbdev/rgbds.