

**NAME****gbz80** - CPU opcode reference**DESCRIPTION**

This is the list of opcodes 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 dual speed mode) needed to complete them.

Note: All arithmetic/logic operations that use register **A** as destination can omit the destination as it is assumed to be register **A** by default. The following two lines have the same effect:

**OR A,B**

**OR B**

**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.

*n16* 16-bit integer constant.

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

*u3* 3-bit unsigned integer constant (**0** to **7**).

*cc* Condition codes:

**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* One of the **RST** vectors (**0x00**, **0x08**, **0x10**, **0x18**, **0x20**, **0x28**, **0x30** and **0x38**).

**INSTRUCTION OVERVIEW****8-bit Arithmetic and Logic Instructions**

*ADC A,r8*

*ADC A,[HL]*  
*ADC A,n8*  
*ADD A,r8*  
*ADD A,[HL]*  
*ADD A,n8*  
*AND A,r8*  
*AND A,[HL]*  
*AND A,n8*  
*CP A,r8*  
*CP A,[HL]*  
*CP A,n8*  
*DEC r8*  
*DEC [HL]*  
*INC r8*  
*INC [HL]*  
*OR A,r8*  
*OR A,[HL]*  
*OR A,n8*  
*SBC A,r8*  
*SBC A,[HL]*  
*SBC A,n8*  
*SUB A,r8*  
*SUB A,[HL]*  
*SUB A,n8*  
*XOR A,r8*  
*XOR A,[HL]*  
*XOR A,n8*

#### **16-bit Arithmetic Instructions**

*ADD HL,r16*  
*DEC r16*  
*INC r16*

#### **Bit Operations Instructions**

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

*SWAP r8*  
*SWAP [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]*

**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]*

### **Jumps and Subroutines**

*CALL n16*  
*CALL cc,n16*  
*JP HL*  
*JP n16*  
*JP cc,n16*  
*JR e8*  
*JR cc,e8*  
*RET cc*  
*RET*  
*RETI*  
*RST vec*

### **Stack Operations 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*

### **Miscellaneous Instructions**

*CCF*  
*CPL*  
*DAA*  
*DI*  
*EI*  
*HALT*  
*NOP*  
*SCF*  
*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**

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]**

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

Cycles: 2

Bytes: 1

Flags: See *AND A,r8*

### **AND A,n8**

Bitwise AND between the value in  $n8$  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**

Subtract the value in *r8* from **A** and set flags accordingly, but don't store the result. This is useful for ComParing values.

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]**

Subtract the byte pointed to by **HL** from **A** and set flags accordingly, but don't store the result.

Cycles: 2

Bytes: 1

Flags: See *CP A, r8*

**CP A, n8**

Subtract the value *n8* from **A** and set flags accordingly, but don't store the result.

Cycles: 2

Bytes: 2

Flags: See *CP A, r8*

**CPL**

ComPLement accumulator (**A** =  $\sim$ **A**).

Cycles: 1

Bytes: 1

Flags:

**N** 1

**H** 1

**DAA**

Decimal Adjust Accumulator to get a correct BCD representation after an arithmetic instruction.

Cycles: 1

Bytes: 1

Flags:

**Z** Set if result is 0.

**H** 0

**C** Set or reset depending on the operation.

**DEC r8**

Decrement 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 value in register *r16* by 1.

Cycles: 2

Bytes: 1

Flags: None affected.

### **DEC SP**

Decrement 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.

**IME** 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.

**IME** not set

The behavior depends on whether an interrupt is pending (i.e. '[IE] & [IF]' is non-zero).

## None pending

As soon as an interrupt becomes pending, the CPU resumes execution. This is like the above, except that the handler is *not* called.

## Some 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 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 value in register *r16* by 1.

Cycles: 2

Bytes: 1

Flags: None affected.

### **INC SP**

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

Cycles: 2

Bytes: 1

Flags: None affected.

### **JP n16**

Jump to address *n16*; effectively, store *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, load **PC** with value in register **HL**.

Cycles: 1

Bytes: 1

Flags: None affected.

**JR e8**

Relative Jump by adding *e8* to the address of the instruction following the **JR**. To clarify, an operand of 0 is equivalent to no jumping.

Cycles: 3

Bytes: 2

Flags: None affected.

**JR cc,e8**

Relative Jump by adding *e8* to the current address if condition *cc* is met.

Cycles: 3 taken / 2 untaken

Bytes: 2

Flags: None affected.

**LD r8,r8**

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

Cycles: 1

Bytes: 1

Flags: None affected.

**LD r8,n8**

Load value *n8* into register *r8*.

Cycles: 2

Bytes: 2

Flags: None affected.

**LD r16,n16**

Load value *n16* into register *r16*.

Cycles: 3

Bytes: 3

Flags: None affected.

**LD [HL],r8**

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

Cycles: 2

Bytes: 1

Flags: None affected.

**LD [HL],n8**

Store value *n8* into byte pointed to by register **HL**.

Cycles: 3

Bytes: 2

Flags: None affected.

**LD r8,[HL]**

Load value into register *r8* from byte pointed to by register **HL**.

Cycles: 2

Bytes: 1

Flags: None affected.

**LD [r16],A**

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

Cycles: 2

Bytes: 1

Flags: None affected.

**LD [n16],A**

Store value in register **A** into byte at address *n16*.

Cycles: 4

Bytes: 3

Flags: None affected.

**LDH [n16],A**

Store value in register **A** into byte at address *n16*, provided it is between *\$FF00* and *\$FFFF*.

Cycles: 3

Bytes: 2

Flags: None affected.

This is sometimes written as 'ldio [n16], a', or 'ld [\$ff00+n8], a'.



**LDH [C],A**

Store value in register **A** into byte at address  $\$FF00+C$ .

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as 'ldio [c], a', or 'ld [\$ff00+c], a'.

**LD A,[r16]**

Load value in register **A** from byte pointed to by register *r16*.

Cycles: 2

Bytes: 1

Flags: None affected.

**LD A,[n16]**

Load value in register **A** from byte at address *n16*.

Cycles: 4

Bytes: 3

Flags: None affected.

**LDH A,[n16]**

Load value in register **A** from byte at address *n16*, provided it is between  $\$FF00$  and  $\$FFFF$ .

Cycles: 3

Bytes: 2

Flags: None affected.

This is sometimes written as 'ldio a, [n16]', or 'ld a, [\$ff00+n8]'.

**LDH A,[C]**

Load value in register **A** from byte at address  $\$FF00+c$ .

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as 'ldio a, [c]', or 'ld a, [\$ff00+c]'.

**LD [HLI],A**

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

Cycles: 2

Bytes: 1

Flags: None affected.

**LD [HLD],A**

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

Cycles: 2

Bytes: 1

Flags: None affected.

**LD A,[HLD]**

Load value into register **A** from byte pointed by **HL** and decrement **HL** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

**LD A,[HLI]**

Load value into register **A** from byte pointed by **HL** and increment **HL** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

### **LD SP,n16**

Load value *n16* into register **SP**.

Cycles: 3

Bytes: 3

Flags: None affected.

### **LD [n16],SP**

Store **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 store 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**

Load 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**

Store into **A** the bitwise OR of 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]**

Store into **A** the bitwise OR of the byte pointed to by **HL** and **A**.

Cycles: 2

Bytes: 1

Flags: See *OR A,r8*

### **OR A,n8**

Store into **A** the bitwise OR of *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], flag_Z << 7 | flag_N << 6 | flag_H << 5 | flag_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 carry.

$$C \leftarrow [7 \leftarrow 0] \leftarrow C$$

Cycles: 2

Bytes: 2

Flags:

**Z** Set if result is 0.

**N** 0

**H** 0

**C** Set according to result.

### **RL [HL]**

Rotate byte pointed to by **HL** left through carry.

$$C \leftarrow [7 \leftarrow 0] \leftarrow C$$

Cycles: 4

Bytes: 2

Flags: See *RL r8*

### **RLA**

Rotate register **A** left through carry.

$$C \leftarrow [7 \leftarrow 0] \leftarrow C$$

Cycles: 1

Bytes: 1

Flags:

**Z** 0

**N** 0

**H** 0

**C** Set according to result.

### **RLC r8**

Rotate register *r8* left.

$$C \leftarrow [7 \leftarrow 0] \leftarrow [7]$$

Cycles: 2

Bytes: 2



Flags:

**Z** Set if result is 0.

**N** 0

**H** 0

**C** Set according to result.

### **RLC [HL]**

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

$$C \leftarrow [7 \leftarrow 0] \leftarrow [7]$$

Cycles: 4

Bytes: 2

Flags: See *RLC r8*

### **RLCA**

Rotate register **A** left.

$$C \leftarrow [7 \leftarrow 0] \leftarrow [7]$$

Cycles: 1

Bytes: 1

Flags:

**Z** 0

**N** 0

**H** 0

**C** Set according to result.

### **RR r8**

Rotate register *r8* right through carry.

$$C \rightarrow [7 \rightarrow 0] \rightarrow C$$

Cycles: 2

Bytes: 2

Flags:

**Z** Set if result is 0.

**N** 0

**H** 0

**C** Set according to result.

### **RR [HL]**

Rotate byte pointed to by **HL** right through carry.

$C \rightarrow [7 \rightarrow 0] \rightarrow C$

Cycles: 4

Bytes: 2

Flags: See *RR r8*

### **RRA**

Rotate register **A** right through carry.

$C \rightarrow [7 \rightarrow 0] \rightarrow C$

Cycles: 1

Bytes: 1

Flags:

**Z** 0

**N** 0

**H** 0

**C** Set according to result.

### **RRC r8**

Rotate register *r8* right.

$[0] \rightarrow [7 \rightarrow 0] \rightarrow C$

Cycles: 2

Bytes: 2

Flags:

**Z** Set if result is 0.

**N** 0

**H** 0

**C** Set according to result.

### **RRC [HL]**

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

[0] -> [7 -> 0] -> C

Cycles: 4

Bytes: 2

Flags: See *RRC r8*

### **RRCA**

Rotate register **A** right.

[0] -> [7 -> 0] -> C

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  $(r8 + \text{carry}) > \mathbf{A}$ ).

**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 Arithmetic register *r8*.

$$C \leftarrow [7 \leftarrow 0] \leftarrow 0$$

Cycles: 2

Bytes: 2

Flags:

**Z** Set if result is 0.

**N** 0

**H** 0

**C** Set according to result.

### **SLA [HL]**

Shift Left Arithmetic byte pointed to by **HL**.

$$C \leftarrow [7 \leftarrow 0] \leftarrow 0$$

Cycles: 4

Bytes: 2

Flags: See *SLA r8*

### **SRA r8**

Shift Right Arithmetic register *r8*.

$[7] \rightarrow [7 \rightarrow 0] \rightarrow C$

Cycles: 2

Bytes: 2

Flags:

**Z** Set if result is 0.

**N** 0

**H** 0

**C** Set according to result.

### **SRA [HL]**

Shift Right Arithmetic byte pointed to by **HL**.

$[7] \rightarrow [7 \rightarrow 0] \rightarrow C$

Cycles: 4

Bytes: 2

Flags: See *SRA r8*

### **SRL r8**

Shift Right Logic register *r8*.

$0 \rightarrow [7 \rightarrow 0] \rightarrow C$

Cycles: 2

Bytes: 2

Flags:

**Z** Set if result is 0.

**N** 0

**H** 0

**C** Set according to result.

### **SRL [HL]**

Shift Right Logic byte pointed to by **HL**.

0 -> [7 -> 0] -> C

Cycles: 4

Bytes: 2

Flags: See *SRA r8*

### **STOP**

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

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 (set 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 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 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**

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]**

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

Cycles: 2

Bytes: 1

Flags: See *XOR A,r8*

**XOR A,n8**

Bitwise XOR between the value in *n8* and **A**.

Cycles: 2

Bytes: 2

Flags: See *XOR A,r8*

**SEE ALSO**

rgbasm(1), rgbds(7)

**HISTORY**

**rgbds** was originally written by Carsten Sørensen as part of the ASMotor package, and was later packaged in RGBDS by Justin Lloyd. It is now maintained by a number of contributors at <https://github.com/gbdev/rgbds>.