

NAME

gbz80 — CPU opcode reference uwu

DESCRIPTION

hOi!! Here's the opcodes supported by that dang ol' *rgbasm*(1) along with some details, the number of bytes and stuff ya need to encode them, and how many CPU cycles at 1MHz (or 2MHz in that **NASTY** GBC dual speed mode) needed to make 'em do the thing!

Note: All GROSS MATH STUFF that uses register (**â€ˆ**) as destination can omit the destination as it is assumed to be register (**â€ˆ**) by default. The following two lines have the same effect:

```
OR (â€ˆ), =B
OR =B
```

LEGEND

Here's some words and what they mean!

- r8* One of those 8-bit registers ((**â€ˆ**), **=B**, **â€ˆ** C), **;D**, (**â€ˆ**), **â€ˆ**, **â€ˆ** (**â€ˆ**)
- r16* One of those general-purpose 16-bit registers (**=B** C), **;D**, (**â€ˆ**), **â€ˆ**, **â€ˆ** (**â€ˆ**)
- n8* 8-bit number
- n16* 16-bit number
- e8* 8-bit offset (**-128** to **127**)
- u3* Weird 3-bit number (**0** to **7**)
- cc* Condition codes:
Z Do thing if Z is set
NZ Do thing if Z is not set
C Do thing if C is set
NC Do thing if C is not set
! cc Do the opposite thing
- vec* One of those dumb **RST** vectors (*0x00*, *0x08*, *0x10*, *0x18*, *0x20*, *0x28*, *0x30*, and *0x38*)

INSTRUCTION OVERVIEW**8-bit Math and Logic Doodads**

```
"ADC (â€ˆ),r8"
"ADC (â€ˆ),[â€ˆ ( â€ˆ )]"
"ADC (â€ˆ),n8"
"ADD (â€ˆ),r8"
"ADD (â€ˆ),[â€ˆ ( â€ˆ )]"
"ADD (â€ˆ),n8"
"AND (â€ˆ),r8"
"AND (â€ˆ),[â€ˆ ( â€ˆ )]"
"AND (â€ˆ),n8"
"CP (â€ˆ),r8"
"CP (â€ˆ),[â€ˆ ( â€ˆ )]"
"CP (â€ˆ),n8"
"DEC r8"
"DEC [â€ˆ ( â€ˆ )]"
"INC r8"
"INC [â€ˆ ( â€ˆ )]"
"OR (â€ˆ),r8"
"OR (â€ˆ),[â€ˆ ( â€ˆ )]"
```

“OR ($\hat{a}\hat{c}\hat{l}$),n8”
 “SBC ($\hat{a}\hat{c}\hat{l}$),r8”
 “SBC ($\hat{a}\hat{c}\hat{l}$),[$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “SBC ($\hat{a}\hat{c}\hat{l}$),n8”
 “SUB ($\hat{a}\hat{c}\hat{l}$),r8”
 “SUB ($\hat{a}\hat{c}\hat{l}$),[$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “SUB ($\hat{a}\hat{c}\hat{l}$),n8”
 “XOR ($\hat{a}\hat{c}\hat{l}$),r8”
 “XOR ($\hat{a}\hat{c}\hat{l}$),[$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “XOR ($\hat{a}\hat{c}\hat{l}$),n8”

16-bit Math Things

“ADD $\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$,r16”
 “DEC r16”
 “INC r16”

Bit Opurrations >=3c

“BIT u3,r8”
 “BIT u3,[$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “RES u3,r8”
 “RES u3,[$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “SET u3,r8”
 “SET u3,[$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “SWAP r8”
 “SWAP [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”

Shifty Bit Stuff \hat{o}

“RL r8”
 “RL [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “RLA”
 “RLC r8”
 “RLC [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “RLCA”
 “RR r8”
 “RR [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “RRA”
 “RRC r8”
 “RRC [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “RRCA”
 “SLA r8”
 “SLA [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “SRA r8”
 “SRA [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “SRL r8”
 “SRL [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”

Load Stuff

“LD r8,r8”
 “LD r8,n8”
 “LD r16,n16”
 “LD [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$],r8”
 “LD [$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$],n8”
 “LD r8,[$\hat{D}\frac{1}{2}\hat{a}$ ($\hat{a}\hat{a}$)i $\frac{1}{4}\hat{c}$]”
 “LD [r16],($\hat{a}\hat{c}\hat{l}$)”

“LD [n16],($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$)”
 “LDH [n16],($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$)”
 “LDH [$\hat{a}\hat{\Psi}(\hat{E}\hat{a}\hat{f}\hat{E}\hat{C})$],($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$)”
 “LD ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$),[r16]”
 “LD ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$),[n16]”
 “LDH ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$),[n16]”
 “LDH ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$),[$\hat{a}\hat{\Psi}(\hat{E}\hat{a}\hat{f}\hat{E}\hat{C})$]”
 “LD [$\hat{D}\hat{1}/\hat{2}\hat{a}$ ($\acute{a}\hat{a}\hat{a}$)i $\hat{1}/\hat{4}\hat{c}$],($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$)”
 “LD [$\hat{D}\hat{1}/\hat{2}\hat{a}$ ($\acute{a}\hat{a}\hat{a}$)i $\hat{1}/\hat{4}\hat{c}$],($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$)”
 “LD ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$),[$\hat{D}\hat{1}/\hat{2}\hat{a}$ ($\acute{a}\hat{a}\hat{a}$)i $\hat{1}/\hat{4}\hat{c}$]]”
 “LD ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$),[$\hat{D}\hat{1}/\hat{2}\hat{a}$ ($\acute{a}\hat{a}\hat{a}$)i $\hat{1}/\hat{4}\hat{c}$]]”

Jumps and Things

“CALL n16”
 “CALL cc,n16”
 “JP $\hat{D}\hat{1}/\hat{2}\hat{a}$ ($\acute{a}\hat{a}\hat{a}$)i $\hat{1}/\hat{4}\hat{c}$ ”
 “JP n16”
 “JP cc,n16”
 “JR e8”
 “JR cc,e8”
 “RET cc”
 “RET”
 “RETI”
 “RST vec”

Stack Operations Instructions uwu

“ADD $\hat{D}\hat{1}/\hat{2}\hat{a}$ ($\acute{a}\hat{a}\hat{a}$)i $\hat{1}/\hat{4}\hat{c}$,SP”
 “ADD SP,e8”
 “DEC SP”
 “INC SP”
 “LD SP,n16”
 “LD [n16],SP”
 “LD $\hat{D}\hat{1}/\hat{2}\hat{a}$ ($\acute{a}\hat{a}\hat{a}$)i $\hat{1}/\hat{4}\hat{c}$,SP+e8”
 “LD SP, $\hat{D}\hat{1}/\hat{2}\hat{a}$ ($\acute{a}\hat{a}\hat{a}$)i $\hat{1}/\hat{4}\hat{c}$ ”
 “POP ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$) $\hat{d}\hat{d}\hat{3}/\hat{4}\hat{d}-\hat{d}$ ”
 “POP r16”
 “PUSH ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$) $\hat{d}\hat{d}\hat{3}/\hat{4}\hat{d}-\hat{d}$ ”
 “PUSH r16”

Weird Instructions?? O_o

“CCF”
 “CPL”
 “DAA”
 “DI”
 “EI”
 “HALTâ”
 “NOPE”
 “OWO”
 “SCF”
 “STOP!!ð”

INSTRUCTION REFERENCE

ADC ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$),r8

Add $r8$'s value plus the carry flag to ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$).

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 (\hat{r} , $\hat{D}^{1/2}(\hat{a})^{1/4}$)

Add the byte at $\hat{D}^{1/2}(\hat{a})^{1/4}$ plus the carry flag to (\hat{r}).

Cycles: 2

Bytes: 1

Flags: See “ADC (\hat{r} , $\hat{D}^{1/2}(\hat{a})^{1/4}$)”

ADC (\hat{r} , $n8$)

Add $n8$ plus the carry flag to (\hat{r}).

Cycles: 2

Bytes: 2

Flags: See “ADC (\hat{r} , $\hat{D}^{1/2}(\hat{a})^{1/4}$)”

ADD (\hat{r} , $r8$)

Add $r8$ ’s value to (\hat{r}).

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 (\hat{r} , $\hat{D}^{1/2}(\hat{a})^{1/4}$)

Add the byte at $\hat{D}^{1/2}(\hat{a})^{1/4}$ to (\hat{r}).

Cycles: 2

Bytes: 1

Flags: See “ADD (\hat{r} , $\hat{D}^{1/2}(\hat{a})^{1/4}$)”

ADD (\hat{r} , $n8$)

Add $n8$ to (\hat{r}).

Cycles: 2

Bytes: 2

Flags: See “ADD (\hat{r} , $\hat{D}^{1/2}(\hat{a})^{1/4}$)”

ADD $\hat{D}^{1/2}(\hat{a})^{1/4}$, $r16$

Add $file \dots$ ’s value $r16$ to $\hat{D}^{1/2}(\hat{a})^{1/4}$.

Cycles: 2

Bytes: 1

Flags:

N 0

H Set if overflow from bit 11.

C Set if overflow from bit 15.

ADD $\text{D}^{1/2}_{\text{A}}(\text{A} \text{ \# } i)_{4}, \text{SP}$

Add **SP**'s value to $\text{D}^{1/2}_{\text{A}}(\text{A} \text{ \# } i)_{4}$.

Cycles: 2

Bytes: 1

Flags: See “ADD $\text{D}^{1/2}_{\text{A}}(\text{A} \text{ \# } i)_{4}, r16$ ”

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 ($\text{A} \text{ \# } i$), r8

Bitwise AND between $r8$'s value and ($\text{A} \text{ \# } i$).

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 0

H 1

C 0

AND ($\text{A} \text{ \# } i$), $[\text{D}^{1/2}_{\text{A}}(\text{A} \text{ \# } i)_{4}]$

Bitwise AND between the byte at $\text{D}^{1/2}_{\text{A}}(\text{A} \text{ \# } i)_{4}$ and ($\text{A} \text{ \# } i$).

Cycles: 2

Bytes: 1

Flags: See “AND ($\text{A} \text{ \# } i$), r8”

AND ($\text{A} \text{ \# } i$), n8

Bitwise AND between $n8$'s value and ($\text{A} \text{ \# } i$).

Cycles: 2

Bytes: 2

Flags: See “AND ($\text{A} \text{ \# } i$), 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,[D_{1/2} (á ãâ)i₄]

Test bit *u3* in the byte pointed by **D_{1/2} (á ãâ)i₄**, 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.

Note: It appreciates the compliment ^w^

Cycles: 1

Bytes: 1

Flags:

N 0

H 0

C Inverted.

CP (âçÌAâçÌ),r8

Subtract *r8*’s value from (**âçÌAâçÌ**) 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* > (**âçÌAâçÌ**)).

CP (âçÌAâçÌ),[D_{1/2} (á ãâ)i₄]

Subtract the byte at **D_{1/2} (á ãâ)i₄** from (**âçÌAâçÌ**) and set flags accordingly, but don’t store the result.

Cycles: 2

Bytes: 1

Flags: See “CP (âçÌAâçÌ),r8”

CP (âçÌAâçÌ),n8

Subtract the value *n8* from (**âçÌAâçÌ**) and set flags accordingly, but don’t store the result.

Cycles: 2

Bytes: 2

Flags: See “CP (A),r8”

CPL

ComPLeMent accumulator ($\mathbf{A} = \sim(\hat{\mathbf{a}} \hat{\mathbf{c}} \hat{\mathbf{I}} \mathbf{A} \hat{\mathbf{a}} \hat{\mathbf{c}} \hat{\mathbf{I}})$).

Note: This one doesn't appreciate the complement $\geq T$

Cycles: 1

Bytes: 1

Flags:

N 1

H 1

DAA

Decimal Adjust Accumulator to get a correct BCD representation after an arithmetic instruction. (Wha???)

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 [Đ^{1/2} (á ã)i^{1/4};

Decrement the byte at $\mathbf{D}^{1/2\hat{a}}(\hat{a}\hat{a})^{1/4}\mathfrak{c}$ 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 [D1/2 (á â)i1/4]

Increment the byte at **D1/2 (á â)i1/4** 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 D1/2â (á ãâ)i1/4;

Jump to address in **D1/2â (á ãâ)i1/4;**; effectively, load **PC** with value in register **D1/2â (á ãâ)i1/4;**.

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 [R1/2, (A)I/4],r8

Store value in register $r8$ into the byte pointed to by register $R1/2 (A)I/4$.

Cycles: 2

Bytes: 1

Flags: None affected.

LD [R1/2, (A)I/4],n8

Store value $n8$ into the byte pointed to by register $R1/2 (A)I/4$.

Cycles: 3

Bytes: 2

Flags: None affected.

LD r8,[R1/2, (A)I/4]

Load value into register $r8$ from the byte pointed to by register $R1/2 (A)I/4$.

Cycles: 2

Bytes: 1

Flags: None affected.

LD [r16],(A)I

Store value in register $(A)I$ into the byte pointed to by register $r16$.

Cycles: 2

Bytes: 1

Flags: None affected.

LD [n16],(A)I

Store value in register $(A)I$ into the byte at address $n16$.

Cycles: 4

Bytes: 3

Flags: None affected.

LDH [n16],(A)I

Store value in register $(A)I$ into the byte at address $n16$, provided the address is between $\$FF00$ and $\$FFFF$.

Cycles: 3

Bytes: 2

Flags: None affected.

This is sometimes written as LDIO [n16], (A)I, or LD [\$FF00+n8], (A)I.

LDH (âŸ(ĒâŸĒ C)],(âŸĬAâŸĬ)

Store value in register (âŸĬAâŸĬ) into the byte at address \$FF00+âŸ(ĒâŸĒ C).

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as LDIO [âŸ(ĒâŸĒ C)],(âŸĬAâŸĬ), or LD [\$FF00+âŸ(ĒâŸĒ C)],(âŸĬAâŸĬ).

LD (âŸĬAâŸĬ),[r16]

Load value in register (âŸĬAâŸĬ) from the byte pointed to by register r16.

Cycles: 2

Bytes: 1

Flags: None affected.

LD (âŸĬAâŸĬ),[n16]

Load value in register (âŸĬAâŸĬ) from the byte at address n16.

Cycles: 4

Bytes: 3

Flags: None affected.

LDH (âŸĬAâŸĬ),[n16]

Load value in register (âŸĬAâŸĬ) from the byte at address n16, provided the address is between \$FF00 and \$FFFF.

Cycles: 3

Bytes: 2

Flags: None affected.

This is sometimes written as LDIO (âŸĬAâŸĬ),[n16], or LD (âŸĬAâŸĬ),[\$FF00+n8].

LDH (âŸĬAâŸĬ),[âŸ(ĒâŸĒ C)]

Load value in register (âŸĬAâŸĬ) from the byte at address \$FF00+c.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as LDIO (âŸĬAâŸĬ),[âŸ(ĒâŸĒ C)], or LD (âŸĬAâŸĬ),[\$FF00+âŸ(ĒâŸĒ C)].

LD [D½â (á ââ)i¼δ],(âŸĬAâŸĬ)

Store value in register (âŸĬAâŸĬ) into the byte pointed by D½â (á ââ)i¼ and increment D½â (á ââ)i¼ afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as LD [D½â (á ââ)i¼+],(âŸĬAâŸĬ), or LDI [D½â (á ââ)i¼],(âŸĬAâŸĬ).

LD [D½â (á ââ)i¼δ],(âŸĬAâŸĬ)

Store value in register (âŸĬAâŸĬ) into the byte pointed by D½â (á ââ)i¼ and decrement D½â (á ââ)i¼ afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as LD $[\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}-]$, ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$), or LDD $[\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}]$, ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$).

LD ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$), $[\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}\delta]$

Load value into register ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$) from the byte pointed by $\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}$ and decrement $\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}$ afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as LD ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$), $[\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}-]$, or LDD ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$), $[\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}]$.

LD ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$), $[\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}\delta]$

Load value into register ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$) from the byte pointed by $\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}$ and increment $\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}$ afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as LD ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$), $[\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}+]$, or LDI ($\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}\mathcal{A}\hat{\mathcal{A}}\mathfrak{c}\mathfrak{I}$), $[\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}]$.

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 $\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}$,SP+e8

Add the signed value $e8$ to **SP** and store the result in $\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}$.

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, $\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}$

Load register $\mathcal{D}/\hat{\mathcal{A}} \text{ (} \acute{\mathcal{A}} \tilde{\mathcal{A}} \text{)}\mathfrak{i}/\mathfrak{z}$ into register **SP**.

Cycles: 2

Bytes: 1

Flags: None affected.

NOPE

No OPERation.

Cycles: 1

Bytes: 1

Flags: None affected.

OR ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$), r8

Store into ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$) the bitwise OR of r8's value and ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$).

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 0

H 0

C 0

OR ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$), [$\hat{D}\hat{1}\hat{2}\hat{a}$ ($\hat{a}\hat{a}\hat{a}$) $\hat{r}\hat{1}\hat{4}\hat{c}$]

Store into ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$) the bitwise OR of the byte at $\hat{D}\hat{1}\hat{2}\hat{a}$ ($\hat{a}\hat{a}\hat{a}$) $\hat{r}\hat{1}\hat{4}\hat{c}$ and ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$).

Cycles: 2

Bytes: 1

Flags: See "OR ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$), r8"

OR ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$), n8

Store into ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$) the bitwise OR of n8 and ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$).

Cycles: 2

Bytes: 2

Flags: See "OR ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$), r8"

OWO

Load *bulge* into register *notice*.

Cycles: 0.25

Bytes: **eyes widen in surprise** r-rgbds! what are you doing?! <///< **starts to blush** xD

Flags:

$\hat{o}'\hat{a}\hat{a}\hat{i}$ Pirate

\hat{o} Checkered

$\hat{o}\ll\hat{o}$ France

$\hat{o}'\hat{o}\hat{s}\hat{o}\hat{c}\hat{o}\cdot\hat{o}\neg\hat{o}\hat{s}\hat{o}$ Dragon

POP ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$) $\hat{o}\hat{o}\hat{4}\hat{o}\neg\hat{o}'$

Pop register ($\hat{a}\hat{c}\hat{l}\hat{A}\hat{a}\hat{c}\hat{l}$) $\hat{o}\hat{o}\hat{4}\hat{o}\neg\hat{o}'$ from the stack. This is roughly equivalent to the following *âˆˆCUTEâˆˆ* 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 *“CUTE”* instructions:

```
ld LOW(r16), [sp] ;  (E C), (i) ; or  (  )i
inc sp
ld HIGH(r16), [sp] ; =B, ;D or D
inc sp
```

Cycles: 3

Bytes: 1

Flags: None affected.

PUSH ()

Push register () into the stack. This is roughly equivalent to the following *“CUTE”* 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 *“CUTE”* instructions:

```
dec sp
ld [sp], HIGH(r16) ; =B, ;D or D
dec sp
ld [sp], LOW(r16) ;  (E C), (i) ; or  (  )i
```

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,[D ()]

Set bit *u3* in the byte pointed by *D ()* 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 [D_{1/2} (á ãâ)r₄]

Rotate the byte at **D_{1/2} (á ãâ)r₄** left through carry.

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

Cycles: 4

Bytes: 2

Flags: See “RL r8”

RLA

Rotate register (**âçÌAâçÌ**) 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 [D₁₆ (A₁₆)]

Rotate the byte at D₁₆ (A₁₆) left.

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

Cycles: 4

Bytes: 2

Flags: See “RLC r8”

RLCA

Rotate register (A₁₆) 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 [D₁₆ (A₁₆)]

Rotate the byte at D₁₆ (A₁₆) right through carry.

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

Cycles: 4

Bytes: 2

Flags: See “RR r8”

RRA

Rotate register (rA) 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 [$\text{D}/\text{A}(\text{A})/4$]

Rotate the byte at $\text{D}/\text{A}(\text{A})/4$ right.

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

Cycles: 4

Bytes: 2

Flags: See “RRC r8”

RRCA

Rotate register (rA) right.

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

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 ($\text{PC} \rightarrow \text{PC}$), r8

Subtract $r8$'s value and the carry flag from ($\text{PC} \rightarrow \text{PC}$).

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 $(r8 + \text{carry}) > (\text{PC} \rightarrow \text{PC})$).

SBC ($\text{PC} \rightarrow \text{PC}$), [$\text{PC} \rightarrow \text{PC} + 1$]

Subtract the byte at $\text{PC} \rightarrow \text{PC} + 1$ and the carry flag from ($\text{PC} \rightarrow \text{PC}$).

Cycles: 2

Bytes: 1

Flags: See "SBC ($\text{PC} \rightarrow \text{PC}$), r8"

SBC ($\text{PC} \rightarrow \text{PC}$), n8

Subtract the value $n8$ and the carry flag from ($\text{PC} \rightarrow \text{PC}$).

Cycles: 2

Bytes: 2

Flags: See "SBC ($\text{PC} \rightarrow \text{PC}$), 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, [$\text{PC} \rightarrow \text{PC} + 1$]

Set bit $u3$ in the byte pointed by $\text{PC} \rightarrow \text{PC} + 1$ 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 $r8$.

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

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

SLA [$\text{D}^{1/2}\hat{\mathbf{a}} \text{ (} \acute{\mathbf{a}} \hat{\mathbf{a}} \text{)i}^{1/4}\hat{\mathbf{z}}$]

Shift Left Arithmetically the byte at $\text{D}^{1/2}\hat{\mathbf{a}} \text{ (} \acute{\mathbf{a}} \hat{\mathbf{a}} \text{)i}^{1/4}\hat{\mathbf{z}}$.

$\text{C} \leftarrow [7 \leftarrow 0] \leftarrow 0$

Cycles: 4

Bytes: 2

Flags: See “SLA r8”

SRA r8

Shift Right Arithmetically register r8.

$[7] \rightarrow [7 \rightarrow 0] \rightarrow \text{C}$

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

SRA [$\text{D}^{1/2}\hat{\mathbf{a}} \text{ (} \acute{\mathbf{a}} \hat{\mathbf{a}} \text{)i}^{1/4}\hat{\mathbf{z}}$]

Shift Right Arithmetically the byte at $\text{D}^{1/2}\hat{\mathbf{a}} \text{ (} \acute{\mathbf{a}} \hat{\mathbf{a}} \text{)i}^{1/4}\hat{\mathbf{z}}$.

$[7] \rightarrow [7 \rightarrow 0] \rightarrow \text{C}$

Cycles: 4

Bytes: 2

Flags: See “SRA r8”

SRL r8

Shift Right Logically register r8.

$0 \rightarrow [7 \rightarrow 0] \rightarrow \text{C}$

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.

N 0

H 0

C Set according to result.

SRL [$\text{D}^{1/2}\hat{\mathbf{a}} \text{ (} \acute{\mathbf{a}} \hat{\mathbf{a}} \text{)i}^{1/4}\hat{\mathbf{z}}$]

Shift Right Logically the byte at $\text{D}^{1/2}\hat{\mathbf{a}} \text{ (} \acute{\mathbf{a}} \hat{\mathbf{a}} \text{)i}^{1/4}\hat{\mathbf{z}}$.

$0 \rightarrow [7 \rightarrow 0] \rightarrow \text{C}$

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 (PC^{L} ,r8

Subtract r8's value from (PC^{L}).

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 $r8 > (\text{PC}^{\text{L}})$).

SUB (PC^{L} ,[$\text{DP}^{\text{L}}(\text{PC}^{\text{L}})_{4}$]

Subtract the byte at $\text{DP}^{\text{L}}(\text{PC}^{\text{L}})_{4}$ from (PC^{L}).

Cycles: 2

Bytes: 1

Flags: See “SUB (PC^{L} ,r8”

SUB (PC^{L} ,n8

Subtract the value n8 from (PC^{L}).

Cycles: 2

Bytes: 2

Flags: See “SUB (PC^{L} ,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 [$\text{DP}^{\text{L}}(\text{PC}^{\text{L}})_{4}$]

Swap the upper 4 bits in the byte pointed by $\text{DP}^{\text{L}}(\text{PC}^{\text{L}})_{4}$ and the lower 4 ones.

Cycles: 4

Bytes: 2

Flags: See “SWAP r8”

XOR (PC^{L} ,r8

Bitwise XOR between r8's value and (PC^{L}).

Cycles: 1

Bytes: 1

Flags:

Z Set if result is 0.

N 0

H 0

C 0

XOR ($\text{PC} + 1, \text{PC} + 1 \text{ (PC + 1)} \text{,r8}]$

Bitwise XOR between the byte at $\text{PC} + 1 \text{ (PC + 1)} \text{,r8}$ and $(\text{PC} + 1)$.

Cycles: 2

Bytes: 1

Flags: See “XOR ($\text{PC} + 1, \text{PC} + 1 \text{ (PC + 1)} \text{,r8}$ ”

XOR ($\text{PC} + 1, \text{PC} + 1 \text{ (PC + 1)} \text{,r8}$

Bitwise XOR between $\text{PC} + 1$'s value and $(\text{PC} + 1)$.

Cycles: 2

Bytes: 2

Flags: See “XOR ($\text{PC} + 1, \text{PC} + 1 \text{ (PC + 1)} \text{,r8}$ ”

SEE ALSO

rgbasm(1), *rgbds(7)*

HISTORY

Carsten Sørensen made this dang cool **rgbds** thingy as part of some ASMMotor program, then Justin Lloyd put it in RGBDS. Now some DUMB NERDS at <https://github.com/gbdev/rgbds> take care of it.