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¢ÌAâ¢Ì)** as destination can omit the destination as it is assumed to be register **(â¢ÌAâ¢Ì)** by default. The following two lines have the same effect:

```
OR ( â¢ÌAâ¢Ì),=B
OR =B
```

LEGEND

Here's some words and what they mean!

r8 One of those 8-bit registers (**(â¢ÌAâ¢Ì)**, **=B**, **â¥(Ëâ£Ë C)**, **;D**, **(Â´Îµï½)â¡**, **Ð½**, **â (á ãâ)ï¼¿**)

r16 One of those general-purpose 16-bit registers (**=Bâ¥(Ëâ£Ë C)**, **;D(Â´Îµï½)â¡**, **Ð½â (á ãâ)ï¼¿**)

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¢ÌAâ¢Ì),r8"
"ADC (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]"
"ADC (â¢ÌAâ¢Ì),n8"
"ADD (â¢ÌAâ¢Ì),r8"
"ADD (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]"
"ADD (â¢ÌAâ¢Ì),n8"
"AND (â¢ÌAâ¢Ì),r8"
"AND (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]"
"AND (â¢ÌAâ¢Ì),n8"
"CP (â¢ÌAâ¢Ì),r8"
"CP (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]"
"CP (â¢ÌAâ¢Ì),n8"
"DEC r8"
"DEC [Ð½â (á ãâ)ï¼¿]"
"INC r8"
"INC [Ð½â (á ãâ)ï¼¿]"
"OR (â¢ÌAâ¢Ì),r8"
"OR (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]"

“OR (â¢ÌAâ¢Ì),n8”
“SBC (â¢ÌAâ¢Ì),r8”
“SBC (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]”
“SBC (â¢ÌAâ¢Ì),n8”
“SUB (â¢ÌAâ¢Ì),r8”
“SUB (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]”
“SUB (â¢ÌAâ¢Ì),n8”
“XOR (â¢ÌAâ¢Ì),r8”
“XOR (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]”
“XOR (â¢ÌAâ¢Ì),n8”

16-bit Math Things

“ADD Ð½â (á ãâ)ï¼¿,r16”
“DEC r16”
“INC r16”

Bit Opurrations >=3c

“BIT u3,r8”
“BIT u3,[Ð½â (á ãâ)ï¼¿]”
“RES u3,r8”
“RES u3,[Ð½â (á ãâ)ï¼¿]”
“SET u3,r8”
“SET u3,[Ð½â (á ãâ)ï¼¿]”
“SWAP r8”
“SWAP [Ð½â (á ãâ)ï¼¿]”

Shifty Bit Stuff ð

“RL r8”
“RL [Ð½â (á ãâ)ï¼¿]”
“RLA”
“RLC r8”
“RLC [Ð½â (á ãâ)ï¼¿]”
“RLCA”
“RR r8”
“RR [Ð½â (á ãâ)ï¼¿]”
“RRA”
“RRC r8”
“RRC [Ð½â (á ãâ)ï¼¿]”
“RRCA”
“SLA r8”
“SLA [Ð½â (á ãâ)ï¼¿]”
“SRA r8”
“SRA [Ð½â (á ãâ)ï¼¿]”
“SRL r8”
“SRL [Ð½â (á ãâ)ï¼¿]”

Load Stuff

“LD r8,r8”
“LD r8,n8”
“LD r16,n16”
“LD [Ð½â (á ãâ)ï¼¿],r8”
“LD [Ð½â (á ãâ)ï¼¿],n8”
“LD r8,[Ð½â (á ãâ)ï¼¿]”
“LD [r16],(â¢ÌAâ¢Ì)”

“LD [n16],(â¢ÌAâ¢Ì)”
“LDH [n16],(â¢ÌAâ¢Ì)”
“LDH [â¥(Ëâ£Ë C)],(â¢ÌAâ¢Ì)”
“LD (â¢ÌAâ¢Ì),[r16]”
“LD (â¢ÌAâ¢Ì),[n16]”
“LDH (â¢ÌAâ¢Ì),[n16]”
“LDH (â¢ÌAâ¢Ì),[â¥(Ëâ£Ë C)]”
“LD [Ð½â (á ãâ)ï¼¿ð],(â¢ÌAâ¢Ì)”
“LD [Ð½â (á ãâ)ï¼¿ð],(â¢ÌAâ¢Ì)”
“LD (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿ð]”
“LD (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿ð]”

Jumps and Things

“CALL n16”
“CALL cc,n16”
“JP Ð½â (á ãâ)ï¼¿”
“JP n16”
“JP cc,n16”
“JR e8”
“JR cc,e8”
“RET cc”
“RET”
“RETI”
“RST vec”

Stack Operations Instwuctions uwu

“ADD Ð½â (á ãâ)ï¼¿,SP”
“ADD SP,e8”
“DEC SP”
“INC SP”
“LD SP,n16”
“LD [n16],SP”
“LD Ð½â (á ãâ)ï¼¿,SP+e8”
“LD SP,Ð½â (á ãâ)ï¼¿”
“POP (â¢ÌAâ¢Ì)ðð¾ð¬ð´”
“POP r16”
“PUSH (â¢ÌAâ¢Ì)ðð¾ð¬ð´”
“PUSH r16”

Weird Instructions?? O_o

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

INSTRUCTION REFERENCE

ADC (â¢ÌAâ¢Ì),r8

Add *r8*’s value plus the carry flag to **(â¢ÌAâ¢Ì)**.

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 (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]
Add the byte at **Ð½â (á ãâ)ï¼¿** plus the carry flag to **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 1

Flags: See “ADC (â¢ÌAâ¢Ì),r8”

ADC (â¢ÌAâ¢Ì),n8
Add *n8* plus the carry flag to **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 2

Flags: See “ADC (â¢ÌAâ¢Ì),r8”

ADD (â¢ÌAâ¢Ì),r8
Add *r8*’s value to **(â¢ÌAâ¢Ì)**.

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 (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]
Add the byte at **Ð½â (á ãâ)ï¼¿** to **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 1

Flags: See “ADD (â¢ÌAâ¢Ì),r8”

ADD (â¢ÌAâ¢Ì),n8
Add *n8* to **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 2

Flags: See “ADD (â¢ÌAâ¢Ì),r8”

ADD Ð½â (á ãâ)ï¼¿,r16
Add *file ...*’s value r16 to **Ð½â (á ãâ)ï¼¿**.

Cycles: 2

Bytes: 1

Flags:
N 0
H Set if overflow from bit 11.

C Set if overflow from bit 15.

ADD Ð½â (á ãâ)ï¼¿,SP
Add **SP**'s value to **Ð½â (á ãâ)ï¼¿**.

Cycles: 2

Bytes: 1

Flags: See "ADD Ð½â (á ãâ)ï¼¿,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 (â¢ÌAâ¢Ì),r8
Bitwise AND between *r8*'s value and **(â¢ÌAâ¢Ì)**.

Cycles: 1

Bytes: 1

Flags:
Z Set if result is 0.
N 0
H 1
C 0

AND (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]
Bitwise AND between the byte at **Ð½â (á ãâ)ï¼¿** and **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 1

Flags: See "AND (â¢ÌAâ¢Ì),r8"

AND (â¢ÌAâ¢Ì),n8
Bitwise AND between *n8*'s value and **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 2

Flags: See "AND (â¢ÌAâ¢Ì),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,[Ð½â (á ãâ)ï¼¿]
Test bit *u3* in the byte pointed by **Ð½â (á ãâ)ï¼¿**, 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 (â¢ÌAâ¢Ì),r8
Subtract *r8*’s value from **(â¢ÌAâ¢Ì)** 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* > **(â¢ÌAâ¢Ì)**).

CP (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]
Subtract the byte at **Ð½â (á ãâ)ï¼¿** from **(â¢ÌAâ¢Ì)** and set flags accordingly, but don’t store the result.

Cycles: 2

Bytes: 1

Flags: See “CP (â¢ÌAâ¢Ì),r8”

CP (â¢ÌAâ¢Ì),n8
Subtract the value *n8* from **(â¢ÌAâ¢Ì)** and set flags accordingly, but don’t store the result.

Cycles: 2

Bytes: 2

Flags: See “CP (â¢ÌAâ¢Ì),r8”

CPL

ComPLement accumulator (**A** = **˜(â¢ÌAâ¢Ì)**).

Note: This one doesn’t appreciate the complement >=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 [Ð½â (á ãâ)ï¼¿]

Decrement the byte at **Ð½â (á ãâ)ï¼¿** 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.

HALTâ

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 **`HALTâ`** 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 **`HALTâ`**, 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 [Ð½â (á ãâ)ï¼¿]

Increment the byte at **Ð½â (á ãâ)ï¼¿** 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 Ð½â (á ãâ)ï¼¿
Jump to address in **Ð½â (á ãâ)ï¼¿**; effectively, load **PC** with value in register **Ð½â (á ãâ)ï¼¿**.

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 [Ð½â (á ãâ)ï¼¿],r8
Store value in register *r8* into the byte pointed to by register **Ð½â (á ãâ)ï¼¿**.

Cycles: 2

Bytes: 1

Flags: None affected.

LD [Ð½â (á ãâ)ï¼¿],n8
Store value *n8* into the byte pointed to by register **Ð½â (á ãâ)ï¼¿**.

Cycles: 3

Bytes: 2

Flags: None affected.

LD r8,[Ð½â (á ãâ)ï¼¿]
Load value into register *r8* from the byte pointed to by register **Ð½â (á ãâ)ï¼¿**.

Cycles: 2

Bytes: 1

Flags: None affected.

LD [r16],(â¢ÌAâ¢Ì)
Store value in register **(â¢ÌAâ¢Ì)** into the byte pointed to by register *r16*.

Cycles: 2

Bytes: 1

Flags: None affected.

LD [n16],(â¢ÌAâ¢Ì)
Store value in register **(â¢ÌAâ¢Ì)** into the byte at address *n16*.

Cycles: 4

Bytes: 3

Flags: None affected.

LDH [n16],(â¢ÌAâ¢Ì)
Store value in register **(â¢ÌAâ¢Ì)** 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],( â¢ÌAâ¢Ì)`, or `LD [$FF00+n8],( â¢ÌAâ¢Ì)`.

LDH [â¥(Ëâ£Ë C)],(â¢ÌAâ¢Ì)
Store value in register **(â¢ÌAâ¢Ì)** into the byte at address *$FF00+â¥(Ëâ£Ë C)*.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LDIO [â¥(Ëâ£Ë C)],( â¢ÌAâ¢Ì)`, or `LD [$FF00+â¥(Ëâ£Ë C)],( â¢ÌAâ¢Ì)`.

LD (â¢ÌAâ¢Ì),[r16]
Load value in register **(â¢ÌAâ¢Ì)** from the byte pointed to by register *r16*.

Cycles: 2

Bytes: 1

Flags: None affected.

LD (â¢ÌAâ¢Ì),[n16]
Load value in register **(â¢ÌAâ¢Ì)** from the byte at address *n16*.

Cycles: 4

Bytes: 3

Flags: None affected.

LDH (â¢ÌAâ¢Ì),[n16]
Load value in register **(â¢ÌAâ¢Ì)** 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 ( â¢ÌAâ¢Ì),[n16]`, or `LD ( â¢ÌAâ¢Ì),[$FF00+n8]`.

LDH (â¢ÌAâ¢Ì),[â¥(Ëâ£Ë C)]
Load value in register **(â¢ÌAâ¢Ì)** from the byte at address *$FF00+c*.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LDIO ( â¢ÌAâ¢Ì),[â¥(Ëâ£Ë C)]`, or `LD ( â¢ÌAâ¢Ì),[$FF00+â¥(Ëâ£Ë C)]`.

LD [Ð½â (á ãâ)ï¼¿ð],(â¢ÌAâ¢Ì)
Store value in register **(â¢ÌAâ¢Ì)** into the byte pointed by **Ð½â (á ãâ)ï¼¿** and increment **Ð½â (á ãâ)ï¼¿** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD [Ð½â ( á ãâ )ï¼¿+],( â¢ÌAâ¢Ì)`, or `LDI [Ð½â ( á ãâ )ï¼¿],( â¢ÌAâ¢Ì)`.

LD [Ð½â (á ãâ)ï¼¿ð],(â¢ÌAâ¢Ì)
Store value in register **(â¢ÌAâ¢Ì)** into the byte pointed by **Ð½â (á ãâ)ï¼¿** and decrement **Ð½â (á ãâ)ï¼¿** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD [Ð½â ( á ãâ )ï¼¿-],( â¢ÌAâ¢Ì)`, or `LDD [Ð½â ( á ãâ )ï¼¿],( â¢ÌAâ¢Ì)`.

LD (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿ð]
Load value into register **(â¢ÌAâ¢Ì)** from the byte pointed by **Ð½â (á ãâ)ï¼¿** and decrement **Ð½â (á ãâ)ï¼¿** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD ( â¢ÌAâ¢Ì),[Ð½â ( á ãâ )ï¼¿-]`, or `LDD ( â¢ÌAâ¢Ì),[Ð½â ( á ãâ )ï¼¿]`.

LD (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿ð]
Load value into register **(â¢ÌAâ¢Ì)** from the byte pointed by **Ð½â (á ãâ)ï¼¿** and increment **Ð½â (á ãâ)ï¼¿** afterwards.

Cycles: 2

Bytes: 1

Flags: None affected.

This is sometimes written as `LD ( â¢ÌAâ¢Ì),[Ð½â ( á ãâ )ï¼¿+]`, or `LDI ( â¢ÌAâ¢Ì),[Ð½â ( á ãâ )ï¼¿]`.

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 Ð½â (á ãâ)ï¼¿,SP+e8
Add the signed value *e8* to **SP** and store the result in **Ð½â (á ãâ)ï¼¿**.

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,Ð½â (á ãâ)ï¼¿
Load register **Ð½â (á ãâ)ï¼¿** into register **SP**.

Cycles: 2

Bytes: 1

Flags: None affected.

NOPE
No OPEration.

Cycles: 1

Bytes: 1

Flags: None affected.

OR (â¢ÌAâ¢Ì),r8
Store into **(â¢ÌAâ¢Ì)** the bitwise OR of *r8*'s value and **(â¢ÌAâ¢Ì)**.

Cycles: 1

Bytes: 1

Flags:
Z Set if result is 0.
N 0
H 0
C 0

OR (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]
Store into **(â¢ÌAâ¢Ì)** the bitwise OR of the byte at **Ð½â (á ãâ)ï¼¿** and **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 1

Flags: See "OR (â¢ÌAâ¢Ì),r8"

OR (â¢ÌAâ¢Ì),n8
Store into **(â¢ÌAâ¢Ì)** the bitwise OR of *n8* and **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 2

Flags: See "OR (â¢ÌAâ¢Ì),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:
ð´ââ ï¸ Pirate
ð Checkered
ð«ð· France
ð´ó §ó ¢ó ·ó ¬ó ³ó ¿ Dragon

POP (â¢ÌAâ¢Ì)ðð¾ð¬ð´
Pop register **(â¢ÌAâ¢Ì)ðð¾ð¬ð´** from the stack. This is roughly equivalent to the following *â¨CUTEâ¨* 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 *â¨CUTEâ¨* instructions:

```
ld LOW(r16), [sp] ; â¥(Ëâ£Ë C), (Â´Îµï½ )â¡ or â ( á ãâ )ï¼¿
inc sp
ld HIGH(r16), [sp] ; =B, ;D or Ð½
inc sp
```

Cycles: 3

Bytes: 1

Flags: None affected.

PUSH (â¢ÌAâ¢Ì)ðð¾ð¬ð´

Push register **(â¢ÌAâ¢Ì)ðð¾ð¬ð´** into the stack. This is roughly equivalent to the following *â¨CUTEâ¨* 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 *â¨CUTEâ¨* instructions:

```
dec sp
ld [sp], HIGH(r16) ; =B, ;D or Ð½
dec sp
ld [sp], LOW(r16) ; â¥(Ëâ£Ë C), (Â´Îµï½ )â¡ or â ( á ãâ )ï¼¿
```

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,[Ð½â (á ãâ)ï¼¿]

Set bit *u3* in the byte pointed by **Ð½â (á ãâ)ï¼¿** 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 <- [7 <- 0] <- C

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.
N 0
H 0
C Set according to result.

RL [Ð½â (á ãâ)ï¼¿]

Rotate the byte at **Ð½â (á ãâ)ï¼¿** left through carry.

C <- [7 <- 0] <- C

Cycles: 4

Bytes: 2

Flags: See “RL r8”

RLA

Rotate register **(â¢ÌAâ¢Ì)** left through carry.

C <- [7 <- 0] <- C

Cycles: 1

Bytes: 1

Flags:

Z 0
N 0
H 0
C Set according to result.

RLC r8
Rotate register *r8* left.

C <- [7 <- 0] <- [7]

Cycles: 2

Bytes: 2

Flags:
Z Set if result is 0.
N 0
H 0
C Set according to result.

RLC [Ð½â (á ãâ)ï¼¿]
Rotate the byte at **Ð½â (á ãâ)ï¼¿** left.

C <- [7 <- 0] <- [7]

Cycles: 4

Bytes: 2

Flags: See “RLC r8”

RLCA
Rotate register **(â¢ÌAâ¢Ì)** left.

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

Cycles: 2

Bytes: 2

Flags:
Z Set if result is 0.
N 0
H 0
C Set according to result.

RR [Ð½â (á ãâ)ï¼¿]
Rotate the byte at **Ð½â (á ãâ)ï¼¿** right through carry.

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

Cycles: 4

Bytes: 2

Flags: See “RR r8”

RRA

Rotate register **(â¢ÌAâ¢Ì)** right through carry.

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

Cycles: 1

Bytes: 1

Flags:

Z 0
N 0
H 0
C Set according to result.

RRC r8

Rotate register *r8* right.

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

Cycles: 2

Bytes: 2

Flags:

Z Set if result is 0.
N 0
H 0
C Set according to result.

RRC [Ð½â (á ãâ)ï¼¿]

Rotate the byte at **Ð½â (á ãâ)ï¼¿** right.

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

Cycles: 4

Bytes: 2

Flags: See “RRC r8”

RRCA

Rotate register **(â¢ÌAâ¢Ì)** 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 (â¢ÌAâ¢Ì),r8
Subtract *r8*'s value and the carry flag from **(â¢ÌAâ¢Ì)**.

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) > **(â¢ÌAâ¢Ì)**).

SBC (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]
Subtract the byte at **Ð½â (á ãâ)ï¼¿** and the carry flag from **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 1

Flags: See "SBC (â¢ÌAâ¢Ì),r8"

SBC (â¢ÌAâ¢Ì),n8
Subtract the value *n8* and the carry flag from **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 2

Flags: See "SBC (â¢ÌAâ¢Ì),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,[Ð½â (á ãâ)ï¼¿]
Set bit *u3* in the byte pointed by **Ð½â (á ãâ)ï¼¿** 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 <- [7 <- 0] <- 0

Cycles: 2

Bytes: 2

Flags:
Z Set if result is 0.
N 0
H 0
C Set according to result.

SLA [Ð½â (á ãâ)ï¼¿]
Shift Left Arithmetically the byte at **Ð½â (á ãâ)ï¼¿**.

C <- [7 <- 0] <- 0

Cycles: 4

Bytes: 2

Flags: See “SLA r8”

SRA r8
Shift Right Arithmetically register *r8*.

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

Cycles: 2

Bytes: 2

Flags:
Z Set if result is 0.
N 0
H 0
C Set according to result.

SRA [Ð½â (á ãâ)ï¼¿]
Shift Right Arithmetically the byte at **Ð½â (á ãâ)ï¼¿**.

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

Cycles: 4

Bytes: 2

Flags: See “SRA r8”

SRL r8
Shift Right Logically register *r8*.

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

Cycles: 2

Bytes: 2

Flags:
Z Set if result is 0.
N 0
H 0
C Set according to result.

SRL [Ð½â (á ãâ)ï¼¿]
Shift Right Logically the byte at **Ð½â (á ãâ)ï¼¿**.

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

Cycles: 4

Bytes: 2

Flags: See “SRA r8”

STOP!!ð

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 (â¢ÌAâ¢Ì),r8

Subtract *r8*’s value from **(â¢ÌAâ¢Ì)**.

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* > **(â¢ÌAâ¢Ì)**).

SUB (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]

Subtract the byte at **Ð½â (á ãâ)ï¼¿** from **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 1

Flags: See “SUB (â¢ÌAâ¢Ì),r8”

SUB (â¢ÌAâ¢Ì),n8

Subtract the value *n8* from **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 2

Flags: See “SUB (â¢ÌAâ¢Ì),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 [Ð½â (á ãâ)ï¼¿]

Swap the upper 4 bits in the byte pointed by **Ð½â (á ãâ)ï¼¿** and the lower 4 ones.

Cycles: 4

Bytes: 2

Flags: See “SWAP r8”

XOR (â¢ÌAâ¢Ì),r8

Bitwise XOR between *r8*’s value and **(â¢ÌAâ¢Ì)**.

Cycles: 1

Bytes: 1

Flags:

Z	Set if result is 0.
N	0
H	0
C	0

XOR (â¢ÌAâ¢Ì),[Ð½â (á ãâ)ï¼¿]

Bitwise XOR between the byte at **Ð½â (á ãâ)ï¼¿** and **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 1

Flags: See “XOR (â¢ÌAâ¢Ì),r8”

XOR (â¢ÌAâ¢Ì),n8

Bitwise XOR between *n8*’s value and **(â¢ÌAâ¢Ì)**.

Cycles: 2

Bytes: 2

Flags: See “XOR (â¢ÌAâ¢Ì),r8”

SEE ALSO

rgbasm(1), *rgbds*(7)

HISTORY

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