Game Boy: Complete Technical Reference

gekkio
`https://gekkio.fi`

April 18, 2020

Revision 106

Preface

IMPORTANT: This document focuses at the moment on 1st and 2nd generation devices (models before the Game Boy Color), and some hardware details are very different in later generations.

Be very careful if you make assumptions about later generation devices based on this document!

How to read this document

> 🧩
>
> This is something that hasn't been verified, but would make a lot of sense.

> ℹ
>
> This explains some caveat about this documentation that you should know.

> ❗
>
> This is a warning about something.

0.1 Formatting of numbers

When a single bit is discussed in isolation, the value looks like this: `0`, `1`.

Binary numbers are prefixed with `0b` like this: `0b0101101`, `0b11011`, `0b00000000`. Values are prefixed with zeroes when necessary, so the total number of digits always matches the number of digits in the value.

Hexadecimal numbers are prefixed with `0x` like this: `0x1234`, `0xDEADBEEF`, `0xFF04`. Values are prefixed with zeroes when necessary, so the total number of characters always matches the number of nibbles in the value.

Examples:

	4-bit	8-bit	16-bit
Binary	`0b0101`	`0b10100101`	`0b0000101010100101`
Hexadecimal	`0x5`	`0xA5`	`0x0AA5`

0.2 Register definitions

Register 0.1: `0x1234` - This is a hardware register definition

R/W-0	R/W-1	U-1	R-0	R-1	R-x	W-1	U-0
VALUE<1:0>		-	BIGVAL<7:5>			FLAG	-
bit 7	6	5	4	3	2	1	bit 0

Top row legend:

R	Bit can be read.
W	Bit can be written. If the bit cannot be read, reading returns a constant value defined in the bit list of the register in question.
U	Unimplemented bit. Writing has no effect, and reading returns a constant value defined in the bit list of the register in question.
-n	Value after system reset: `0`, `1`, or x.
1	Bit is set.
0	Bit is cleared.
x	Bit is unknown (e.g. depends on external things such as user input).

Middle row legend:

VALUE<1:0>	Bits 1 and 0 of VALUE
-	Unimplemented bit
BIGVAL<7:5>	Bits 7, 6, 5 of BIGVAL
FLAG	Single-bit value FLAG

In this example:

- After system reset, VALUE is `0b01`, BIGVAL is either `0b010` or `0b011`, FLAG is `0b1`.
- Bits 5 and 0 are unimplemented. Bit 5 always returns `1`, and bit 0 always returns `0`.
- Both bits of VALUE can be read and written. When this register is written, bit 7 of the written value goes to bit 1 of VALUE.
- FLAG can only be written to, so reads return a value that is defined elsewhere.
- BIGVAL cannot be written to. Only bits 5-7 of BIGVAL are defined here, so look elsewhere for the low bits 0-4.

Contents

Part I

Sharp SM83 CPU core

Chapter 1

CPU core timing

1.1 Fetch/execute overlap

Sharp SM83 uses a microprocessor design technique known as *fetch/execute overlap* to improve CPU performance by doing opcode fetches in parallel with instruction execution whenever possible. Since the CPU can only perform one memory access per M-cycle, it is worth it to try to do memory operations as soon as possible. Also, when doing a memory read, the CPU cannot use the data during the same M-cycle so the true minimum effective duration of instructions is 2 machine cycles, not 1 machine cycle.

Every instruction needs one machine cycle for the fetch stage, and at least one machine cycle for the decode/execute stage. However, the fetch stage of an instruction always overlaps with the last machine cycle of the execute stage of the previous instruction. The overlapping execute stage cycle may still do some work (e.g. ALU operation and/or register writeback) but memory access is reserved for the fetch stage of the next instruction.

Since all instructions effectively last one machine cycle longer, fetch/execute overlap is usually ignored in documentation intended for programmers. It is much easier to think of a program as a sequence of non-overlapping instructions and consider only the execute stages when calculating instruction durations. However, when emulating a SM83 CPU core, understanding and emulating the overlap can be useful.

Sharp SM831x is a family of single-chip SoCs from Sharp that use the SM83 CPU core, and their datasheet [5] includes a description of fetch/execute overlap. However, the description is not completely correct and can in fact be misleading.
For example, the timing diagram includes an instruction that does not involve opcode fetch at all, and memory operations for two instructions are shown to happen at the same time, which is not possible.

Fetch/execute overlap timing example

Let's assume the CPU is executing a program that starts from the address `0x1000` and contains the following instructions:

```
0x1000: INC A
0x1001: LDH (n), A
0x1003: RST 0x08
0x0008: NOP
```

The following timing diagram shows all memory operations done by the CPU, and the fetch and execute stages of each instruction:

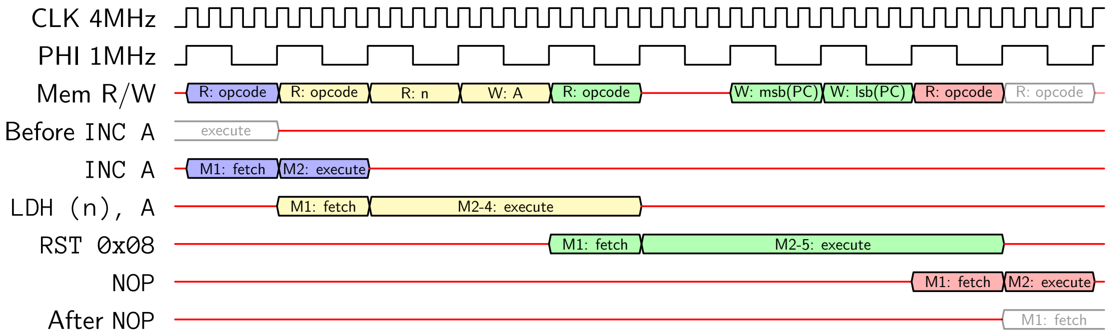


Figure 1.1: Fetch/execute overlap example

Chapter 2

Sharp SM83 instruction set

2.1 8-bit load instructions

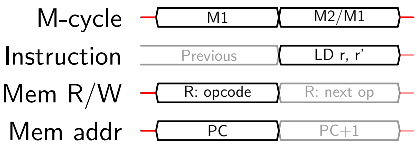
8-bit load instructions transfer one byte of data between two 8-bit registers, or between one 8-bit register and location in memory.

LD r, r′

Load to the 8-bit register `r`, data from the 8-bit register `r'`.

Opcode	`0b01xxxyyy`/various
Length	1 byte
Duration	1 machine cycle
Flags	-

Timing



Pseudocode

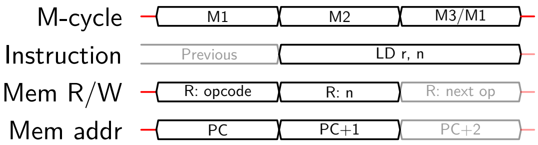
```
opcode = read(PC++)
# example: LD B, C
if opcode == 0x41:
  B = C
```

LD r, n

Load to the 8-bit register `r`, the immediate data `n`.

Opcode	`0b00xxx110`/various + `n`
Length	2 byte
Duration	2 machine cycles
Flags	-

Timing



Pseudocode

```
opcode = read(PC++)
# example: LD B, n
if opcode == 0x06:
  B = read(PC++)
```

LD r, (HL)

Load to the 8-bit register `r`, data from the absolute address specified by the 16-bit register HL.

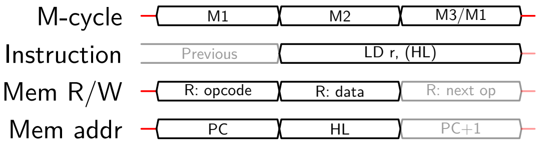
Opcode `0b01xxx110`/various

Length 1 byte

Duration 2 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
# example: LD B, (HL)
if opcode == 0x46:
  B = read(HL)
```

LD (HL), r

Load to the absolute address specified by the 16-bit register HL, data from the 8-bit register `r`.

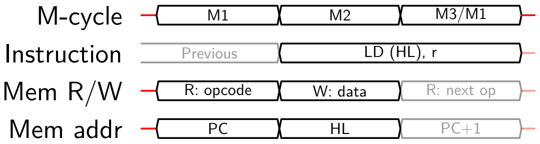
Opcode `0b01110xxx`/various

Length 1 byte

Duration 2 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
# example: LD (HL), B
if opcode == 0x70:
  write(HL, B)
```

LD (HL), n

Load to the absolute address specified by the 16-bit register HL, the immediate data `n`.

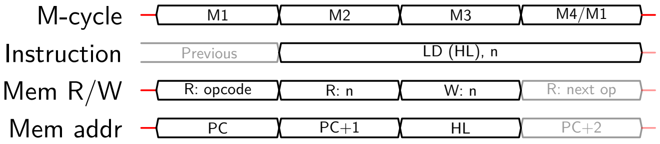
Opcode `0b00110110`/`0x36` + `n`

Length 2 bytes

Duration 3 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x36:
  n = read(PC++)
  write(HL, n)
```

LD A, (BC)

Load to the 8-bit A register, data from the absolute address specified by the 16-bit register BC.

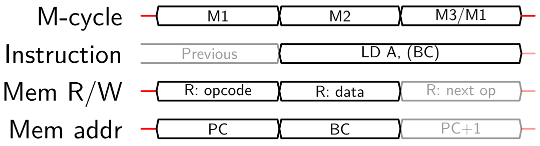
Opcode `0b00001010/0x0A`

Length 1 byte

Duration 2 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x0A:
  A = read(BC)
```

LD A, (DE)

Load to the 8-bit A register, data from the absolute address specified by the 16-bit register DE.

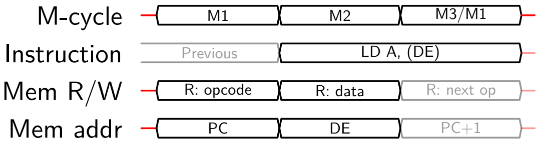
Opcode `0b00011010/0x1A`

Length 1 byte

Duration 2 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x1A:
  A = read(DE)
```

LD (BC), a

Load to the absolute address specified by the 16-bit register BC, data from the 8-bit A register.

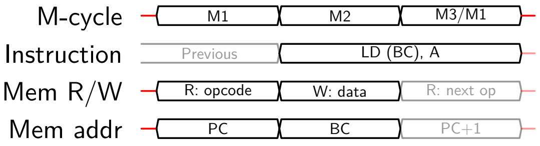
Opcode `0b00000010/0x02`

Length 1 byte

Duration 2 machine cycles

Flags -

Timing



Pseudocode

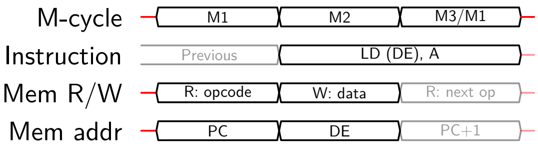
```
opcode = read(PC++)
if opcode == 0x02:
  write(BC, A)
```

LD (DE), a

Load to the absolute address specified by the 16-bit register DE, data from the 8-bit A register.

Opcode	`0b00010010`/`0x12`
Length	1 byte
Duration	2 machine cycles
Flags	-

Timing



Pseudocode

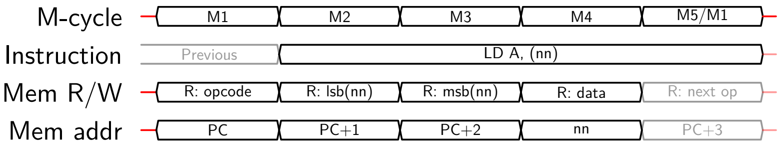
```
opcode = read(PC++)
if opcode == 0x12:
  write(DE, A)
```

LD A, (nn)

Load to the 8-bit A register, data from the absolute address specified by the 16-bit operand `nn`.

Opcode	`0b11111010`/`0xFA` + LSB of `nn` + MSB of `nn`
Length	3 bytes
Duration	4 machine cycles
Flags	-

Timing



Pseudocode

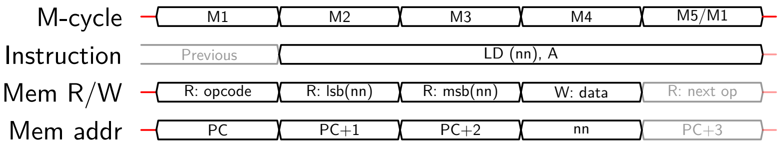
```
opcode = read(PC++)
if opcode == 0xFA:
  nn = unsigned_16(lsb=read(PC++), msb=read(PC++))
  A = read(nn)
```

LD (nn), A

Load to the absolute address specified by the 16-bit operand `nn`, data from the 8-bit A register.

Opcode	`0b11101010`/`0xEA` + LSB of `nn` + MSB of `nn`
Length	3 bytes
Duration	4 machine cycles
Flags	-

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0xFA:
  nn = unsigned_16(lsb=read(PC++), msb=read(PC++))
  write(nn, A)
```

LDH A, (C)

Load to the 8-bit A register, data from the address specified by the 8-bit C register. The full 16-bit absolute address is obtained by setting the most significant byte to `0xFF` and the least significant byte to the value of C, so the possible range is `0xFF00-0xFFFF`.

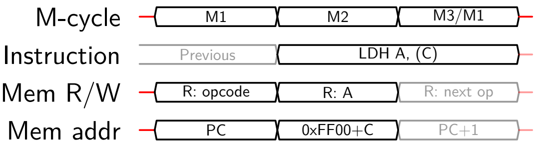
Opcode `0b11110010/0xF2`

Length 1 bytes

Duration 2 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0xF2:
  A = read(unsigned_16(lsb=C, msb=0xFF))
```

LDH (C), A

Load to the address specified by the 8-bit C register, data from the 8-bit A register. The full 16-bit absolute address is obtained by setting the most significant byte to `0xFF` and the least significant byte to the value of C, so the possible range is `0xFF00-0xFFFF`.

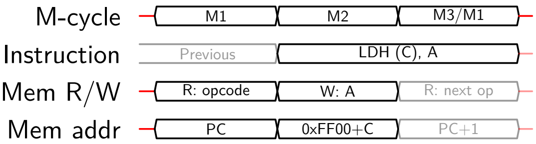
Opcode `0b11100010/0xE2`

Length 1 bytes

Duration 2 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0xE2:
  write(unsigned_16(lsb=C, msb=0xFF), A)
```

LDH A, (n)

Load to the 8-bit A register, data from the address specified by the 8-bit immediate data `n`. The full 16-bit absolute address is obtained by setting the most significant byte to `0xFF` and the least significant byte to the value of `n`, so the possible range is `0xFF00-0xFFFF`.

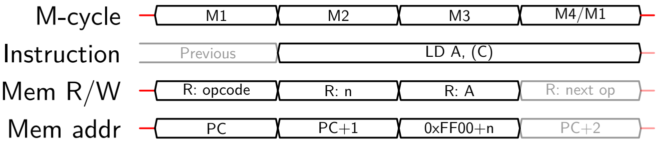
Opcode `0b11110000/0xF0`

Length 2 bytes

Duration 3 machine cycles

Flags -

Timing



Pseudocode

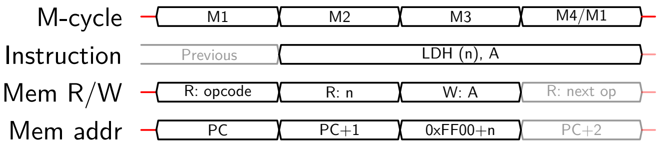
```
opcode = read(PC++)
if opcode == 0xF0:
  n = read(PC++)
  A = read(unsigned_16(lsb=n, msb=0xFF))
```

LDH (n), A

Load to the address specified by the 8-bit immediate data `n`, data from the 8-bit A register. The full 16-bit absolute address is obtained by setting the most significant byte to `0xFF` and the least significant byte to the value of `n`, so the possible range is `0xFF00`–`0xFFFF`.

Opcode	`0b11100000/0xE0`
Length	2 bytes
Duration	3 machine cycles
Flags	-

Timing



Pseudocode

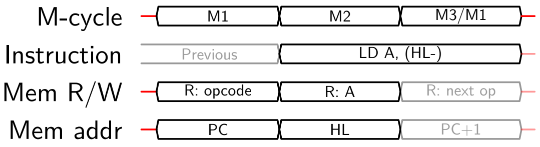
```
opcode = read(PC++)
if opcode == 0xE0:
  n = read(PC++)
  write(unsigned_16(lsb=n, msb=0xFF), A)
```

LD A, (HL-)

Load to the 8-bit A register, data from the absolute address specified by the 16-bit register HL. The value of HL is decremented after the memory read.

Opcode	`0b00111010/0x3A`
Length	1 bytes
Duration	2 machine cycles
Flags	-

Timing



Pseudocode

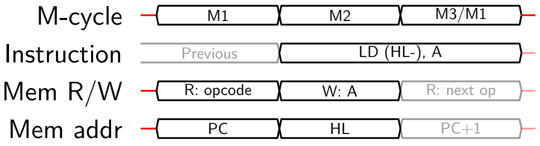
```
opcode = read(PC++)
if opcode == 0x3A:
  A = read(HL--)
```

LD (HL-), A

Load to the absolute address specified by the 16-bit register HL, data from the 8-bit A register. The value of HL is decremented after the memory write.

Opcode	`0b00110010/0x32`
Length	1 bytes
Duration	2 machine cycles
Flags	-

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x32:
  write(HL--, A)
```

LD A, (HL+)

Load to the 8-bit A register, data from the absolute address specified by the 16-bit register HL. The value of HL is incremented after the memory read.

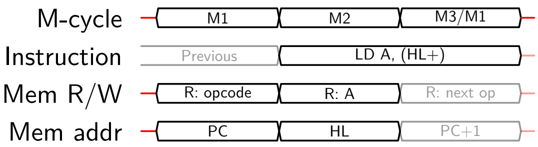
Opcode `0b00101010`/`0x2A`

Length 1 bytes

Duration 2 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x2A:
  A = read(HL++)
```

LD (HL+), A

Load to the absolute address specified by the 16-bit register HL, data from the 8-bit A register. The value of HL is incremented after the memory write.

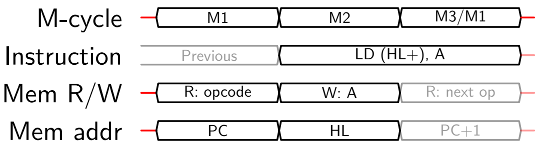
Opcode `0b00100010`/`0x22`

Length 1 bytes

Duration 2 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x22:
  write(HL++, A)
```

2.2 16-bit load instructions

16-bit load instructions transfer two bytes of data between two 16-bit registers, or between one 16-bit register and two sequential locations in memory.

LD rr, nn

Load to the 16-bit register `rr`, the immediate 16-bit data nn.

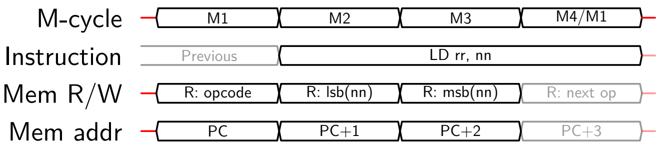
Opcode `0b00xx0001`/various + LSB of nn + MSB of nn

Length 3 byte

Duration 3 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
# example: LD BC, nn
if opcode == 0x01:
  nn = unsigned_16(lsb=read(PC++), msb=read(PC++))
  BC = nn
```

LD (nn), SP

Load to the absolute address specified by the 16-bit operand `nn`, data from the 16-bit SP register.

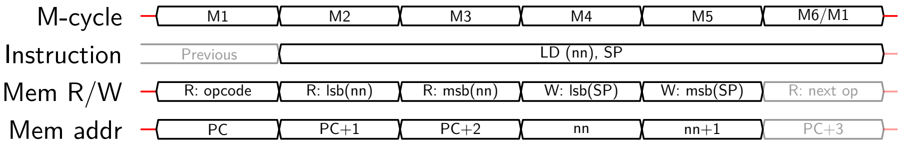
Opcode `0b00001000`/`0x08` + LSB of `nn` + MSB of `nn`

Length 3 byte

Duration 5 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x08:
  nn = unsigned_16(lsb=read(PC++), msb=read(PC++))
  write(nn, lsb(SP))
  write(nn+1, msb(SP))
```

LD SP, HL

Load to the 16-bit SP register, data from the 16-bit HL register.

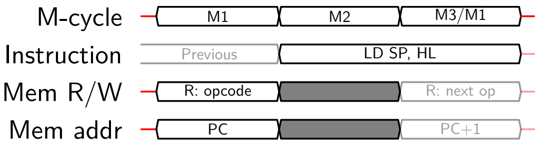
Opcode `0b11111001`/`0xF9`

Length 1 byte

Duration 2 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0xF9:
  SP = HL
```

PUSH rr

Push to the stack memory, data from the 16-bit register `rr`.

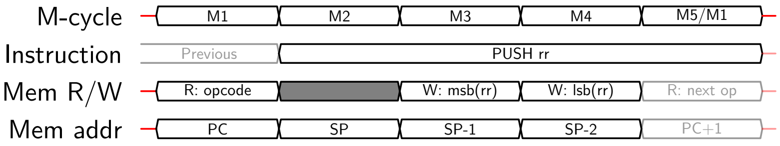
Opcode `0b11xx0101`/various

Length 1 byte

Duration 4 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
# example: PUSH BC
if opcode == 0xC5:
  SP--
  write(SP--, msb(BC))
  write(SP--, lsb(BC))
```

POP rr

Pops to the 16-bit register `rr`, data from the stack memory.

This instruction does not do calculations that affect flags, but POP AF completely replaces the F register value, so all flags are changed based on the 8-bit data that is read from memory.

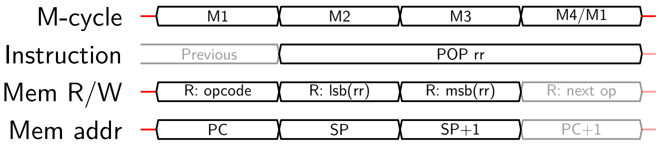
Opcode `0b11xx0001`/various

Length 1 byte

Duration 3 machine cycles

Flags see the instruction description

Timing



Pseudocode

```
opcode = read(PC++)
# example: POP BC
if opcode == 0xC1:
  BC = unsigned_16(lsb=read(SP++), msb=read(SP++))
```

2.3 8-bit arithmetic instructions

2.4 16-bit arithmetic instructions

2.5 Rotate, shift, and bit operation instructions

2.6 Control flow instructions

JP nn

Unconditional jump to the absolute address specified by the 16-bit operand nn.

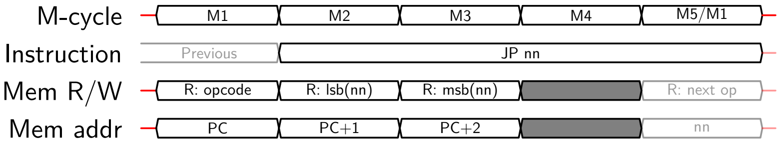
Opcode + data `0b11000011`/`0xC3` + LSB of nn + MSB of nn

Length 3 bytes

Duration 4 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0xC3:
  nn = unsigned_16(lsb=read(PC++), msb=read(PC++))
  PC = nn
```

JP HL

Unconditional jump to the absolute address specified by the 16-bit register HL.

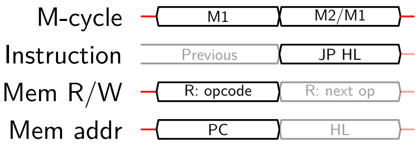
Opcode `0b11101001`/`0xE9`

Length 1 byte

Duration 1 machine cycle

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0xE9:
  PC = HL
```

> In some documentation this instruction is written as `JP [HL]`. This is very misleading, since brackets are usually used to indicate a memory read, and this instruction simply copies the value of HL to PC.

JP cc, nn

Conditional jump to the absolute address specified by the 16-bit operand `nn`, depending on the condition `cc`.

Note that the operand (absolute address) is read even when the condition is false!

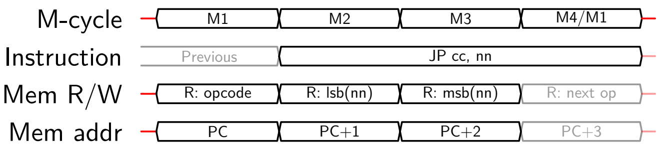
Opcode + data `0b110cc010`/various + LSB of `nn` + MSB of `nn`

Length 3 bytes

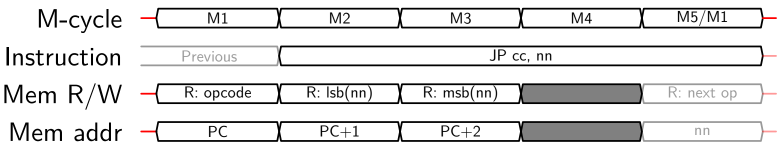
Duration 3 machine cycles (cc=false), or 4 machine cycles (cc=true)

Flags -

Timing (cc=false)



Timing (cc=true)



Pseudocode

```
opcode = read(PC++)
if opcode in [0xC2, 0xD2, 0xCA, 0xDA]:
  nn = unsigned_16(lsb=read(PC++), msb=read(PC++))
  if F.check_condition(cc):
    PC = nn
```

JR e

Unconditional jump to the relative address specified by the signed 8-bit operand `e`.

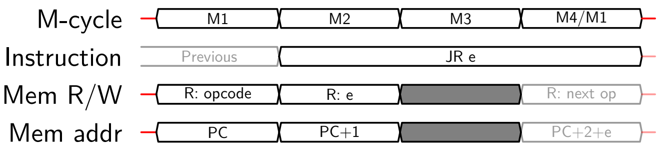
Opcode + data `0b00011000`/`0x18` + offset `e`

Length 2 bytes

Duration 3 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x18:
  e = signed_8(read(PC++))
  PC = PC + e
```

JR cc, e

Conditional jump to the relative address specified by the signed 8-bit operand `e`, depending on the condition `cc`.

Note that the operand (relative address offset) is read even when the condition is false!

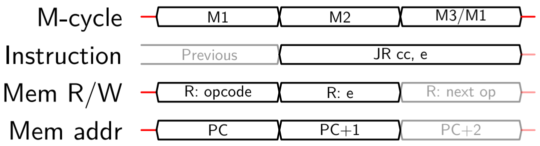
Opcode + data `0b001cc000`/various + offset `e`

Length 2 bytes

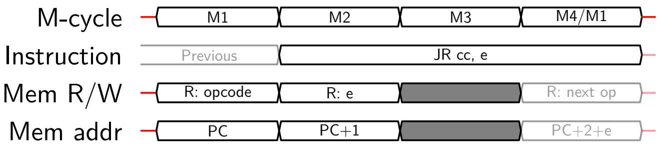
Duration 2 machine cycles (cc=false), or 3 machine cycles (cc=true)

Flags -

Timing (cc=false)



Timing (cc=true)



Pseudocode

```
opcode = read(PC++)
if opcode in [0x20, 0x30, 0x28, 0x38]:
  e = signed_8(read(PC++))
  if F.check_condition(cc):
    PC = PC + e
```

CALL nn

Unconditional function call to the absolute address specified by the 16-bit operand `nn`.

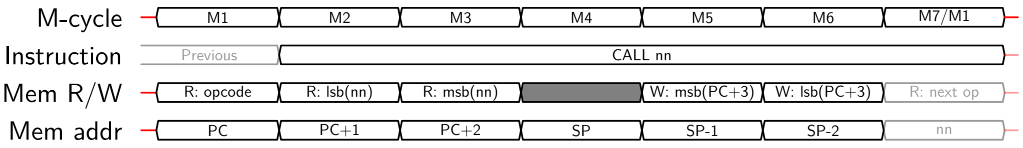
Opcode + data `0b11001101`/`0xCD` + LSB of `nn` + MSB of `nn`

Length 3 bytes

Duration 6 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0xCD:
  nn = unsigned_16(lsb=read(PC++), msb=read(PC++))
  write(--SP, msb(PC))
  write(--SP, lsb(PC))
  PC = nn
```

CALL cc, nn

Conditional function call to the absolute address specified by the 16-bit operand `nn`, depending on the condition `cc`.

Note that the operand (absolute address) is read even when the condition is false!

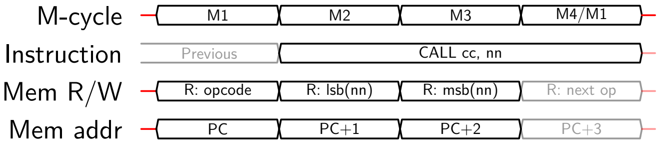
Opcode + data `0b110cc100`/various + LSB of `nn` + MSB of `nn`

Length 3 bytes

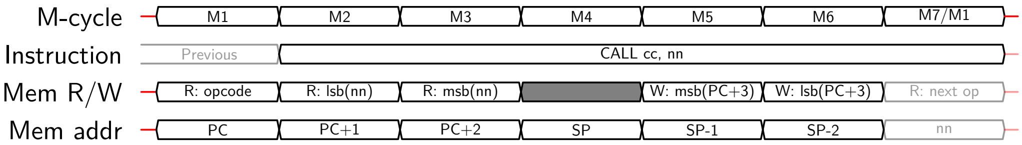
Duration 3 machine cycles (cc=false), or 6 machine cycles (cc=true)

Flags -

Timing (cc=false)



Timing (cc=true)



Pseudocode

```
opcode = read(PC++)
if opcode in [0xC4, 0xD4, 0xCC, 0xDC]:
  nn = unsigned_16(lsb=read(PC++), msb=read(PC++))
  if F.check_condition(cc):
    write(--SP, msb(PC))
    write(--SP, lsb(PC))
    PC = nn
```

RET

Unconditional return from a function.

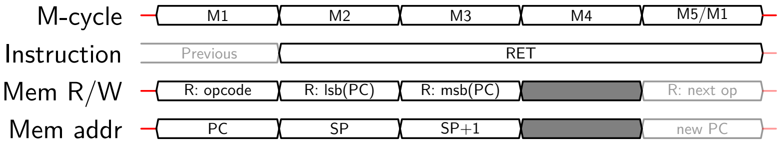
Opcode `0b11001001`/`0xC9`

Length 1 byte

Duration 4 machine cycles

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0xC9:
  PC = unsigned_16(lsb=read(SP++), msb=read(SP++))
```

RET cc

Conditional return from a function, depending on the condition `cc`.

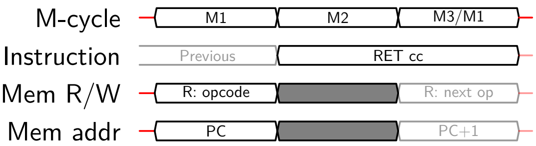
Opcode `0b110cc000`/various

Length 1 byte

Duration 2 machine cycles (cc=false), or 5 machine cycles (cc=true)

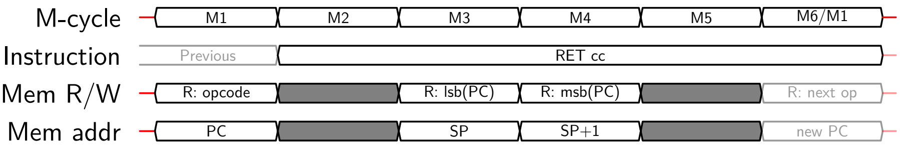
Flags -

Timing (cc=false)



M-cycle	M1	M2	M3/M1
Instruction	Previous	RET cc	
Mem R/W	R: opcode	▨	R: next op
Mem addr	PC	▨	PC+1

Timing (cc=true)



M-cycle	M1	M2	M3	M4	M5	M6/M1
Instruction	Previous	RET cc				
Mem R/W	R: opcode	▨	R: lsb(PC)	R: msb(PC)	▨	R: next op
Mem addr	PC	▨	SP	SP+1	▨	new PC

Pseudocode

```
opcode = read(PC++)
if opcode in [0xC0, 0xD0, 0xC8, 0xD8]:
  if F.check_condition(cc):
    PC = unsigned_16(lsb=read(SP++), msb=read(SP++))
```

RETI

Unconditional return from a function. Also enables interrupts by setting IME=1.

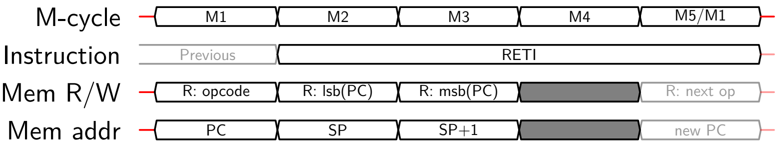
Opcode `0b11011001`/`0xD9`

Length 1 byte

Duration 4 machine cycles

Flags -

Timing



M-cycle	M1	M2	M3	M4	M5/M1
Instruction	Previous	RETI			
Mem R/W	R: opcode	R: lsb(PC)	R: msb(PC)	▨	R: next op
Mem addr	PC	SP	SP+1	▨	new PC

Pseudocode

```
opcode = read(PC++)
if opcode == 0xD9:
  PC = unsigned_16(lsb=read(SP++), msb=read(SP++))
  IME = 1
```

RST n

Unconditional function call to the absolute fixed address defined by the opcode.

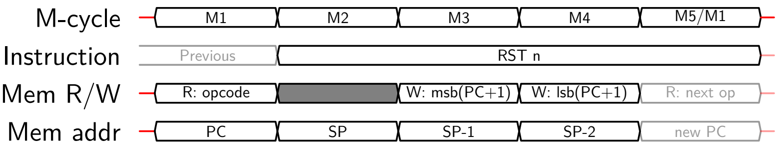
Opcode `0b11xxx111`/various

Length 1 byte

Duration 4 machine cycles

Flags -

Timing



M-cycle	M1	M2	M3	M4	M5/M1
Instruction	Previous	RST n			
Mem R/W	R: opcode	▨	W: msb(PC+1)	W: lsb(PC+1)	R: next op
Mem addr	PC	SP	SP-1	SP-2	new PC

Pseudocode

```
opcode = read(PC++)
if opcode in [0xC7, 0xD7, 0xE7, 0xF7, 0xCF, 0xDF, 0xEF, 0xFF]:
  n = rst_address(opcode)
  write(--SP, msb(PC))
  write(--SP, lsb(PC))
  PC = unsigned_16(lsb=n, msb=0x00)
```

2.7 Miscellaneous instructions

HALT

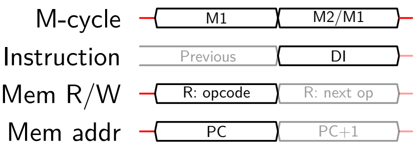
STOP

DI

Disables interrupt handling by setting IME=0 and cancelling any scheduled effects of the EI instruction if any.

Opcode	`0b11110011`/`0xF3`
Length	1 byte
Duration	1 machine cycle
Flags	-

Timing



Pseudocode

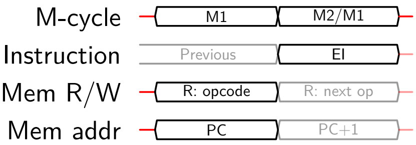
```
opcode = read(PC++)
if opcode == 0xF3:
  IME = 0
```

EI

Schedules interrupt handling to be enabled after the next machine cycle.

Opcode	`0b11111011`/`0xFB`
Length	1 byte
Duration	1 machine cycle (+ 1 machine cycle for the effect)
Flags	-

Timing



Pseudocode

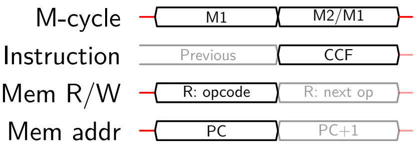
```
opcode = read(PC++)
if opcode == 0xFB:
  IME_scheduled = true
```

CCF

Flips the carry flag, and clears the N and H flags.

Opcode	`0b00111111`/`0x3F`
Length	1 byte
Duration	1 machine cycle
Flags	N = 0, H = 0, C = ★

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x3F:
  flags.N = 0
  flags.H = 0
  flags.C = ~flags.C
```

SCF

Sets the carry flag, and clears the N and H flags.

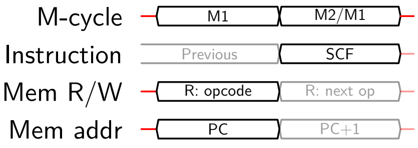
Opcode `0b00110111/0x37`

Length 1 byte

Duration 1 machine cycle

Flags N = 0, H = 0, C = 1

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x37:
  flags.N = 0
  flags.H = 0
  flags.C = 1
```

NOP

No-operation. This instruction doesn't do anything, but can be used to add a delay of one machine cycle and increment PC by one.

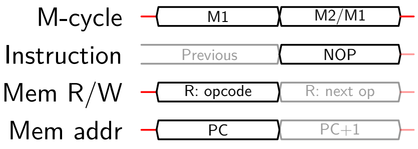
Opcode `0b00000000/0x00`

Length 1 byte

Duration 1 machine cycle

Flags -

Timing



Pseudocode

```
opcode = read(PC++)
if opcode == 0x00:
  // nothing
```

DAA

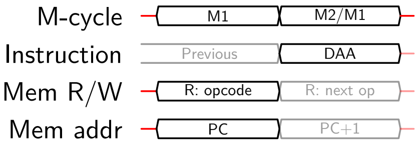
Opcode `0b00100111/0x27`

Length 1 byte

Duration 1 machine cycle

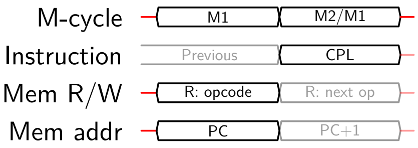
Flags Z = ★, H = 0, C = ★

Timing



CPL

Flips all the bits in the 8-bit A register, and sets the N and H flags.

Opcode	`0b00101111`/`0x2F`
Length	1 byte
Duration	1 machine cycle
Flags	N = 1, H = 1
Timing	

Pseudocode

```
opcode = read(PC++)
if opcode == 0x2F:
  A = ~A
  flags.N = 1
  flags.H = 1
```

Part II

Game Boy SoC peripherals and features

Chapter 3

Boot ROM

The Game Boy SoC includes a small embedded boot ROM, which can be mapped to the `0x0000-0x00FF` memory area. While mapped, all reads from this area are handled by the boot ROM instead of the external cartridge, and all writes to this area are ignored and cannot be seen by external hardware (e.g. the cartridge MBC).

The boot ROM is enabled by default, so when the system exits the reset state and the CPU starts execution from address `0x0000`, it executes the boot ROM instead of instructions from the cartridge ROM. The boot ROM is responsible for showing the initial logo, and checking that a valid cartridge is inserted into the system. If the cartridge is valid, the boot ROM unmaps itself before execution of the cartridge ROM starts at `0x0100`. The cartridge ROM has no chance of executing any instructions before the boot ROM is unmapped, which prevents the boot ROM from being read byte by byte in normal conditions.

Don't confuse the boot ROM with the additional SNES ROM in SGB/SGB2 that is executed by the SNES CPU.

Register 3.1: `0xFF50` - BOOT - Boot ROM lock register

`U-1`	`U-1`	`U-1`	`U-1`	`U-1`	`U-1`	`U-1`	`R/W-0`
`-`	`-`	`-`	`-`	`-`	`-`	`-`	`BOOT_OFF`
`bit 7`	`6`	`5`	`4`	`3`	`2`	`1`	`bit 0`

bit 7-1 **Unimplemented**: Read as `1`

bit 0 **BOOT_OFF**: Boot ROM lock bit
`0b1` = Boot ROM is disabled and `0x0000-0x00FF` works normally.
`0b0` = Boot ROM is active and intercepts accesses to `0x0000-0x00FF`.

BOOT_OFF can only transition from `0b0` to `0b1`, so once `0b1` has been written, the boot ROM is permanently disabled until the next system reset. Writing `0b0` when BOOT_OFF is `0b0` has no effect and doesn't lock the boot ROM.

The 1-bit BOOT register controls mapping of the boot ROM. Once `1` has been written to it to unmap the boot ROM, it can only be mapped again by resetting the system.

3.1 Boot ROM types

Table 3.1: Summary of boot ROM file hashes

Type	CRC32	MD5	SHA1
DMG	`59c8598e`	`32fbbd84168d3482956eb3c5051637f5`	`4ed31ec6b0b175bb109c0eb5fd3d193da823339f`
MGB	`e6920754`	`71a378e71ff30b2d8a1f02bf5c7896aa`	`4e68f9da03c310e84c523654b9026e51f26ce7f0`
SGB	`ec8a83b9`	`d574d4f9c12f305074798f54c091a8b4`	`aa2f50a77dfb4823da96ba99309085a3c6278515`
SGB2	`53d0dd63`	`e0430bca9925fb9882148fd2dc2418c1`	`93407ea10d2f30ab96a314d8eca44fe160aea734`
DMG0	`c2f5cc97`	`a8f84a0ac44da5d3f0ee19f9cea80a8c`	`8bd501e31921e9601788316dbd3ce9833a97bcbc`

DMG boot ROM

The most common boot ROM is the DMG boot ROM used in almost all original Game Boy units. If a valid cartridge is inserted, the boot ROM scrolls a logo to the center of the screen, and plays a "di-ding" sound recognizable by most people who have used Game Boy consoles.

This boot ROM was originally dumped by neviksti in 2003 by decapping the Game Boy SoC and visually inspecting every single bit.

MGB boot ROM

This boot ROM was originally dumped by Bennvenn in 2014 by using a simple clock glitching method that only requires one wire.

SGB boot ROM

This boot ROM was originally dumped by Costis Sideris in 2009 by using an FPGA-based clock glitching method [6].

SGB2 boot ROM

This boot ROM was originally dumped by gekkio in 2015 by using a Teensy 3.1 -based clock glitching method [2].

Early DMG boot ROM

Very early original Game Boy units released in Japan (often called "DMG0") included the launch version "DMG-CPU" SoC chip, which used a different boot ROM than later units.

This boot ROM was originally dumped by gekkio in 2016 by using a clock glitching method invented by BennVenn.

Chapter 4

DMA (Direct Memory Access)

4.1 Object Attribute Memory (OAM) DMA

OAM DMA is a high-throughput mechanism for copying data to the OAM area (a.k.a. Object Attribute Memory, a.k.a. sprite memory). It can copy one byte per machine cycle without involving the CPU at all, which is much faster than the fastest possible `memcpy` routine that can be written with the SM83 instruction set. However, a transfer cannot be cancelled and the transfer length cannot be controlled, so the DMA transfer always updates the entire OAM area (= 160 bytes) even if you actually want to just update the first couple of bytes.

The Game Boy CPU chip contains a DMA controller that coordinates transfers between a *source area* and the *OAM area* independently of the CPU. While a transfer is in progress, it takes control of the source bus and the OAM area, so some precaution is needed with memory accesses (including instruction fetches) to avoid OAM DMA bus conflicts. OAM DMA uses a different address decoding scheme than normal memory accesses, so the source bus is always either the external bus or the video RAM bus, and the contents normally visible to the CPU in the `0xFE00`–`0xFFFF` address range cannot be used as a source for OAM DMA transfers.

The upper 8 bits of the OAM DMA source address are stored in the DMA register, while the lower 8 bits used by both the source and target address are stored in the DMA controller and are not accessible directly. A transfer always begins with `0x00` in the lower bits and copies exactly 160 bytes, so the lower bits are never in the `0xA0`–`0xFF` range.

Writing to the DMA register updates the upper bits of the DMA source address and also triggers an OAM DMA transfer request, although the DMA transfer does not begin immediately.

Register 4.1: `0xFF46` - DMA - OAM DMA control register

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
DMA<7:0>							
bit 7	6	5	4	3	2	1	bit 0

bit 0 **DMA<7:0>**: OAM DMA source address
Specifies the top 8 bits of the OAM DMA source address.

Writing to this register requests an OAM DMA transfer, but it's just a request and the actual DMA transfer starts with a delay.

Reading this register returns the value that was previously written to the register. The stored value is not cleared on reset, so the initial value before the first write is unknown and should not be relied on.

> ❗
>
> Avoid writing `0xE0`–`0xFF` to the DMA register, because some poorly designed flash carts can trigger bus conflicts or other dangerous behaviour.

OAM DMA address decoding

The OAM DMA controller uses a simplified address decoding scheme, which leads to some addresses being unusable as source addresses. Unlike normal memory accesses, OAM DMA transfers interpret all accesses in the `0xA000`–`0xFFFF` range as external RAM transfers. For example, if the OAM DMA wants to read `0xFF00`, it will output `0xFF00` on the external address bus and will assert the external RAM chip select signal. The P1 register which is normally at `0xFF00` is not involved at all, because OAM DMA address decoding only uses the external bus and the video RAM bus. Instead, the resulting behaviour depends on several factors, including the connected cartridge. Some flash carts are not prepared for this unexpected scenario, and a bus conflict or worse behaviour can happen.

Table 4.1: OAM DMA address decoding scheme

DMA register value	Used bus	Asserted chip select signal
`0x00-0x7F`	external bus	external ROM (A15)
`0x80-0x9F`	video RAM bus	video RAM (MCS)
`0xA0-0xFF`	external bus	external RAM (CS)

OAM DMA transfer timing

TODO

OAM DMA bus conflicts

TODO

Chapter 5

PPU (Picture Processing Unit)

Register 5.1: `0xFF40` - LCDC - PPU control register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
LCD_EN	WIN_MAP	WIN_EN	TILE_SEL	BG_MAP	OBJ_SIZE	OBJ_EN	BG_EN
bit 7	6	5	4	3	2	1	bit 0

Register 5.2: `0xFF41` - LCDC - PPU status register

U-1	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
-	INTR_LYC	INTR_M2	INTR_M1	INTR_M0	LYC_STAT	LCD_MODE<1:0>	
bit 7	6	5	4	3	2	1	bit 0

Register 5.3: `0xFF42` - SCY - Vertical scroll register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SCY<7:0>							
bit 7	6	5	4	3	2	1	bit 0

Register 5.4: `0xFF43` - SCX - Horizontal scroll register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SCX<7:0>							
bit 7	6	5	4	3	2	1	bit 0

Register 5.5: `0xFF44` - LY - Scanline register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
LY<7:0>							
bit 7	6	5	4	3	2	1	bit 0

Register 5.6: `0xFF45` - LYC - Scanline compare register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
LYC<7:0>							
bit 7	6	5	4	3	2	1	bit 0

Chapter 6

Port P1 (Joypad, Super Game Boy communication)

Register 6.1: `0xFF00` - P1 - Joypad/Super Game Boy communication register

`U-1`	`U-1`	`W-0`	`W-0`	`R-x`	`R-x`	`R-x`	`R-x`
`-`	`-`	`P15`	`P14`	`P13`	`P12`	`P11`	`P10`
`bit 7`	`6`	`5`	`4`	`3`	`2`	`1`	`bit 0`

bit 7-6 **Unimplemented**: Read as `1`

bit 5 **P15**:

bit 4 **P14**:

bit 3 **P13**:

bit 2 **P12**:

bit 1 **P11**:

bit 0 **P10**:

Chapter 7

Serial communication

Register 7.1: `0xFF01` - SB - Serial data register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SB<7:0>							
bit 7	6	5	4	3	2	1	bit 0

bit 7-0 **SB<7:0>**: Serial data

Register 7.2: `0xFF02` - SC - Serial control register

R/W-0	U-1	U-1	U-1	U-1	U-1	U-1	R/W-0
SIO_EN	-	-	-	-	-	-	SIO_CLK
bit 7	6	5	4	3	2	1	bit 0

bit 7 **SIO_EN**:

bit 6-1 **Unimplemented**: Read as `1`

bit 0 **SIO_CLK**:

Part III

Game Boy game cartridges

Chapter 8

MBC1 mapper chip

The majority of games for the original Game Boy use the MBC1 chip. MBC1 supports ROM sizes up to 16 Mbit (128 banks of `0x4000` bytes) and RAM sizes up to 256 Kbit (4 banks of `0x2000` bytes). The information in this section is based on my MBC1 research, Tauwasser's research notes [7], and Pan Docs [3].

8.1 MBC1 registers

> These registers don't have any standard names and are usually referred to using their address ranges or purposes instead. This document uses names to clarify which register is meant when referring to one.

The MBC1 chip includes four registers that affect the behaviour of the chip. Of the cartridge bus address signals, only A13-A15 are connected to the MBC, so lower address bits don't matter when the CPU is accessing the MBC and all registers are effectively mapped to address ranges instead of single addresses. All registers are smaller than 8 bits, and unused bits are simply ignored during writes. The registers are not directly readable.

Register 8.1: `0x0000-0x1FFF` - RAMG - MBC1 RAM gate register

U	U	U	U	W-0	W-0	W-0	W-0
				RAMG<3:0>			
bit 7	6	5	4	3	2	1	bit 0

bit 7-4 **Unimplemented**: Ignored during writes

bit 3-0 **RAMG<3:0>**: RAM gate register
`0b1010`= enable access to cartridge RAM
All other values disable access to cartridge RAM

The RAMG register is used to enable access to the cartridge SRAM if one exists on the cartridge circuit board. RAM access is disabled by default but can be enabled by writing to the `0x0000-0x1FFF` address range a value with the bit pattern `0b1010` in the lower nibble. Upper bits don't matter, but any other bit pattern in the lower nibble disables access to RAM.

When RAM access is disabled, all writes to the external RAM area `0xA000-0xBFFF` are ignored, and reads return undefined values. Pan Docs recommends disabling RAM when it's not being accessed to protect the contents [3].

> We don't know the physical implementation of RAMG, but it's certainly possible that the `0b1010` bit pattern check is done at write time and the register actually consists of just a single bit.

Register 8.2: `0x2000-0x3FFF` - BANK1 - MBC1 bank register 1

`U`	`U`	`U`	`W-0`	`W-0`	`W-0`	`W-0`	`W-1`
			`BANK1<4:0>`				
`bit 7`	`6`	`5`	`4`	`3`	`2`	`1`	`bit 0`

bit 7-5 **Unimplemented**: Ignored during writes

bit 4-0 **BANK1<4:0>**: Bank register 1
Never contains the value `0b00000`.
If `0b00000` is written, the resulting value will be `0b00001` instead.

The 5-bit BANK1 register is used as the lower 5 bits of the ROM bank number when the CPU accesses the `0x4000-0x7FFF` memory area.

MBC1 doesn't allow the BANK1 register to contain zero (bit pattern `0b00000`), so the initial value at reset is `0b00001` and attempting to write `0b00000` will write `0b00001` instead. This makes it impossible to read banks `0x00`, `0x20`, `0x40` and `0x60` from the `0x4000-0x7FFF` memory area, because those bank numbers have `0b00000` in the lower bits. Due to the zero value adjustment, requesting any of these banks actually requests the next bank (e.g. `0x21` instead of `0x20`).

Register 8.3: `0x4000-0x5FFF` - BANK2 - MBC1 bank register 2

`U`	`U`	`U`	`U`	`U`	`U`	`W-0`	`W-0`
						`BANK2<1:0>`	
`bit 7`	`6`	`5`	`4`	`3`	`2`	`1`	`bit 0`

bit 7-2 **Unimplemented**: Ignored during writes

bit 1-0 **BANK2<1:0>**: Bank register 2

The 2-bit BANK2 register can be used as the upper bits of the ROM bank number, or as the 2-bit RAM bank number. Unlike BANK1, BANK2 doesn't disallow zero, so all 2-bit values are possible.

Register 8.4: `0x6000-0x7FFF` - MODE - MBC1 mode register

`U`	`U`	`U`	`U`	`U`	`U`	`U`	`W-0`
							`MODE`
`bit 7`	`6`	`5`	`4`	`3`	`2`	`1`	`bit 0`

bit 7-1 **Unimplemented**: Ignored during writes

bit 0 **MODE**: Mode register
`0b1`= BANK2 affects accesses to `0x0000-0x3FFF`,`0x4000-0x7FFF`,`0xA000-0xBFFF`
`0b0`= BANK2 affects only accesses to `0x4000-0x7FFF`

The MODE register determines how the BANK2 register value is used during memory accesses.

> Most documentation, including Pan Docs [3], calls value `0b0` ROM banking mode, and value `0b1` RAM banking mode. This terminology reflects the common use cases, but "RAM banking" is slightly misleading because value `0b1` also affects ROM reads in multicart cartridges and cartridges that have a 8 or 16 Mbit ROM chip.

8.2 ROM in the `0x0000-0x7FFF` area

In MBC1 cartridges, the A0-A13 cartridge bus signals are connected directly to the corresponding ROM pins, and the remaining ROM pins (A14-A20) are controlled by the MBC1. These remaining pins form the ROM bank number.

When the `0x0000–0x3FFF` address range is accessed, the effective bank number depends on the MODE register. In MODE `0b0` the bank number is always 0, but in MODE `0b1` it's formed by shifting the BANK2 register value left by 5 bits.

When the `0x4000–0x7FFF` addess range is accessed, the effective bank number is always a combination of BANK1 and BANK2 register values.

If the cartridge ROM is smaller than 16 Mbit, there are less ROM address pins to connect to and therefore some bank number bits are ignored. For example, 4 Mbit ROMs only need a 5-bit bank number, so the BANK2 register value is always ignored because those bits are simply not connected to the ROM.

Table 8.1: Mapping of physical ROM address bits in MBC1 carts

	ROM address bits		
Accessed address	Bank number		Address within bank
	20-19	18-14	13-0
`0x0000–0x3FFF`, MODE = `0b0`	`0b00`	`0b00000`	A<13:0>
`0x0000–0x3FFF`, MODE = `0b1`	BANK2	`0b00000`	A<13:0>
`0x4000–0x7FFF`	BANK2	BANK1	A<13:0>

ROM banking example 1

Let's assume we have previously written `0x12` to the BANK1 register and `0b01` to the BANK2 register. The effective bank number during ROM reads depends on which address range we read and on the value of the MODE register:

Value of the BANK1 register
`0b 10010`

Value of the BANK2 register
`0b 01`

Effective ROM bank number (reading `0x4000–0x7FFF`)
`0b 01 10010 (= 50 = 0x32)`

Effective ROM bank number (reading `0x0000–0x3FFF`, MODE = `0b0`)
`0b 00 00000 (= 0 = 0x00)`

Effective ROM bank number (reading `0x0000–0x3FFF`, MODE = `0b1`)
`0b 01 00000 (= 32 = 0x20)`

ROM banking example 2

Let's assume we have previously requested ROM bank number 68, MBC1 mode is `0b0`, and we are now reading a byte from `0x72A7`. The actual physical ROM address that will be read is going to be `0x1132A7` and is constructed in the following way:

Value of the BANK1 register	`0b 00100`
Value of the BANK2 register	`0b 10`
ROM bank number	`0b 10 00100 (= 68 = 0x44)`
Address being read	`0b 01 11 0010 1010 0111 (= 0x72A7)`
Actual physical ROM address	`0b 1 0 001 00 11 0010 1010 0111 (= 0x1132A7)`

8.3 RAM in the `0xA000–0xBFFF` area

Some MBC1 carts include SRAM, which is mapped to the `0xA000–0xBFFF` area. If no RAM is present, or RAM is not enabled with the RAMG register, all reads return undefined values and writes have no effect.

On boards that have RAM, the A0-A12 cartridge bus signals are connected directly to the corresponding RAM pins, and pins A13-A14 are controlled by the MBC1. Most of the time the RAM size is 64 Kbit, which corresponds to a single bank of `0x2000` bytes. With larger RAM sizes the BANK2 register value can be used for RAM banking to provide the two high address bits.

In MODE `0b0` the BANK2 register value is not used, so the first RAM bank is always mapped to the `0xA000–0xBFFF` area. In MODE `0b1` the BANK2 register value is used as the bank number.

Table 8.2: Mapping of physical RAM address bits in MBC1 carts

Accessed address	RAM address bits	
	Bank number	Address within bank
	14-13	12-0
`0xA000–0xBFFF`, MODE = `0b0`	`0b00`	A<12:0>
`0xA000–0xBFFF`, MODE = `0b1`	BANK2	A<12:0>

RAM banking example 1

Let's assume we have previously written `0b10` to the BANK2 register, MODE is `0b1`, RAMG is `0b1010` and we are now reading a byte from `0xB123`. The actual physical RAM address that will be read is going to be `0x5123` and is constructed in the following way:

Value of the BANK2 register `0b 10`

Address being read `0b 101 1 0001 0010 0011 (= 0xB123)`

Actual physical RAM address `0b 10 1 0001 0010 0011 (= 0x5123)`

8.4 MBC1 multicarts ("MBC1M")

MBC1 is also used in a couple of "multicart" cartridges, which include more than one game on the same cartridge. These cartridges use the same regular MBC1 chip, but the circuit board is wired a bit differently. This alternative wiring is sometimes called "MBC1M", but technically the mapper chip is the same. All known MBC1 multicarts use 8 Mbit ROMs, so there's no definitive wiring for other ROM sizes.

In MBC1 multicarts bit 4 of the BANK1 register is not physically connected to anything, so it's skipped. This means that the bank number is actually a 6-bit number. In all known MBC1 multicarts the games reserve 16 banks each, so BANK2 can actually be considered "game number", while BANK1 is the internal bank number within the selected game. At reset BANK2 is `0b00`, and the "game" in this slot is actually a game selection menu. The menu code selects MODE `0b1` and writes the game number to BANK2 once the user selects a game.

From a ROM banking point of view, multicarts simply skip bit 4 of the BANK1 register, but otherwise the behaviour is the same. MODE `0b1` guarantees that all ROM accesses, including accesses to `0x0000–0x3FFF`, use the BANK2 register value.

Table 8.3: Mapping of physical ROM address bits in MBC1 multicarts

Accessed address	ROM address bits		
	Bank number		Address within bank
	19-18	17-14	13-0
`0x0000–0x3FFF`, MODE = `0b0`	`0b00`	`0b0000`	A<13:0>
`0x0000–0x3FFF`, MODE = `0b1`	BANK2	`0b0000`	A<13:0>
`0x4000–0x7FFF`	BANK2	BANK1<3:0>	A<13:0>

ROM banking example 1

Let's assume we have previously requested "game number" 3 (= `0b11`) and ROM bank number 29 (= `0x1D`), MBC1 mode is `0b1`, and we are now reading a byte from `0x6C15`. The actual physical ROM address that will be read is going to be `0xF6C15` and is constructed in the following way:

Value of the BANK1 register `0b 1 1101`

Value of the BANK2 register `0b 11`

ROM bank number `0b 11 1101 (= 61 = 0x3D)`

Address being read	`0b 01 10 1100 0001 0101` (= `0x6C15`)
Actual physical ROM address	`0b 11 11 01 10 1100 0001 0101` (= `0xF6C15`)

Detecting multicarts

MBC1 multicarts are not detectable by simply looking at the ROM header, because the ROM type value is just one of the normal MBC1 values. However, detection is possible by going through BANK2 values and looking at "bank 0" of each multicart game and doing some heuristics based on the header data. All the included games, including the game selection menu, have proper header data. One example of a good heuristic is logo data verification.

So, if you have a 8 Mbit cart with MBC1, first assume that it's a multicart and bank numbers are 6-bit values. Set BANK1 to zero and loop through the four possible BANK2 values while checking the data at `0x0104-0x0133`. In other words, check logo data starting from physical ROM locations `0x00104`, `0x40104`, `0x80104`, and `0xC0104`. If proper logo data exists with most of the BANK2 values, the cart is most likely a multicart. Note that multicarts can just have two actual games, so one of the locations might not have the header data in place.

8.5 Dumping MBC1 carts

MBC1 cartridge dumping is fairly straightforward with the right hardware. The total number of banks is read from the header, and each bank is read one byte at a time. However, BANK1 register zero-adjustment and multicart cartridges need to be considered in ROM dumping code.

Banks `0x20`, `0x40` and `0x60` can only be read from the `0x0000-0x3FFF` memory area and only when MODE register value is `0b1`. Using MODE `0b1` has no undesirable effects when doing ROM dumping, so using it at all times is recommended for simplicity.

Multicarts should be detected using the logo check described earlier, and if a multicart is detected, BANK1 should be considered a 4-bit register in the dumping code.

```
write_byte(0x6000, 0x01)
for bank in range(0, num_banks):
    write_byte(0x2000, bank)
    if is_multicart:
        write_byte(0x4000, bank >> 4)
        bank_start = 0x4000 if bank & 0x0f else 0x0000
    else:
        write_byte(0x4000, bank >> 5)
        bank_start = 0x4000 if bank & 0x1f else 0x0000
    for addr in range(bank_start, bank_start + 0x4000):
        buf += read_byte(addr)
```

Listing 1: Python pseudo-code for MBC1 ROM dumping

Chapter 9

MBC2 mapper chip

MBC2 supports ROM sizes up to 2 Mbit (16 banks of `0x4000` bytes) and includes an internal 512x4 bit RAM array, which is its unique feature. The information in this section is based on my MBC2 research, Tauwasser's research notes [8], and Pan Docs [3].

> MBC1 is strictly more powerful than MBC2 because it supports more ROM and RAM. This raises a very important question: why does MBC2 exist? It's possible that Nintendo tried to integrate a small amount of RAM on the MBC chip for cost reasons, but it seems that this didn't work out very well since all later MBCs revert this design decision and use separate RAM chips.

9.1 MBC2 registers

> These registers don't have any standard names and are usually referred to using one of their addresses or purposes instead. This document uses names to clarify which register is meant when referring to one.

The MBC2 chip includes two registers that affect the behaviour of the chip. The registers are mapped a bit differently compared to other MBCs. Both registers are accessible within `0x0000–0x3FFF`, and within that range, the register is chosen based on the A8 address signal. In practice, this means that the registers are mapped to memory in an alternating pattern. For example, `0x0000`, `0x2000` and `0x3000` are RAMG, and `0x0100`, `0x2100` and `0x3100` are ROMB. Both registers are smaller than 8 bits, and unused bits are simply ignored during writes. The registers are not directly readable.

Register 9.1: `0x0000–0x3FFF` when A8=`0b0` - RAMG - MBC2 RAM gate register

`U`	`U`	`U`	`U`	`W-0`	`W-0`	`W-0`	`W-0`
				`RAMG<3:0>`			
`bit 7`	`6`	`5`	`4`	`3`	`2`	`1`	`bit 0`

bit 7-4 **Unimplemented**: Ignored during writes

bit 3-0 **RAMG<3:0>**: RAM gate register
`0b1010`= enable access to chip RAM
All other values disable access to chip RAM

The 4-bit MBC2 RAMG register works in a similar manner as MBC1 RAMG, so the upper bits don't matter and only the bit pattern `0b1010` enables access to RAM.

When RAM access is disabled, all writes to the external RAM area `0xA000–0xBFFF` are ignored, and reads return undefined values. Pan Docs recommends disabling RAM when it's not being accessed to protect the contents [3].

> We don't know the physical implementation of RAMG, but it's certainly possible that the `0b1010` bit pattern check is done at write time and the register actually consists of just a single bit.

Register 9.2: `0x0000–0x3FFF` when A8=`0b1` - ROMB - MBC2 ROM bank register

U	U	U	U	W-0	W-0	W-0	W-1
				ROMB<3:0>			
bit 7	6	5	4	3	2	1	bit 0

bit 7-4 **Unimplemented**: Ignored during writes

bit 3-0 **ROMB<3:0>**: ROM bank register
Never contains the value `0b0000`.
If `0b0000` is written, the resulting value will be `0b0001` instead.

The 4-bit ROMB register is used as the ROM bank number when the CPU accesses the `0x4000–0x7FFF` memory area.

Like MBC1 BANK1, the MBC2 ROMB register doesn't allow zero (bit pattern `0b0000`) in the register, so any attempt to write `0b0000` writes `0b0001` instead.

9.2 ROM in the `0x0000–0x7FFF` area

In MBC2 cartridges, the A0-A13 cartridge bus signals are connected directly to the corresponding ROM pins, and the remaining ROM pins (A14-A17) are controlled by the MBC2. These remaining pins form the ROM bank number.

When the `0x0000–0x3FFF` address range is accessed, the effective bank number is always 0.

When the `0x4000–0x7FFF` address range is accessed, the effective bank number is the current ROMB register value.

Table 9.1: Mapping of physical ROM address bits in MBC2 carts

Accessed address	ROM address bits	
	Bank number	Address within bank
	17-14	13-0
`0x0000–0x3FFF`	`0b0000`	A<13:0>
`0x4000–0x7FFF`	ROMB	A<13:0>

9.3 RAM in the `0xA000–0xBFFF` area

All MBC2 carts include SRAM, because it is located directly inside the MBC2 chip. These cartridges never use a separate RAM chip, but battery backup circuitry and a battery are optional. If RAM is not enabled with the RAMG register, all reads return undefined values and writes have no effect.

MBC2 RAM is only 4-bit RAM, so the upper 4 bits of data do not physically exist in the chip. When writing to it, the upper 4 bits are ignored. When reading from it, the upper 4 data signals are not driven by the chip, so their content is undefined and should not be relied on.

MBC2 RAM consists of 512 addresses, so only A0-A8 matter when accessing the RAM region. There is no banking, and the `0xA000–0xBFFF` area is larger than the RAM, so the addresses wrap around. For example, accessing `0xA000` is the same as accessing `0xA200`, so it is possible to write to the former address and later read the written data using the latter address.

Table 9.2: Mapping of physical RAM address bits in MBC2 carts

Accessed address	RAM address bits
	8-0
`0xA000–0xBFFF`	A<8:0>

9.4 Dumping MBC2 carts

MBC2 cartridges are very simple to dump. The total number of banks is read from the header, and each bank is read one byte at a time. ROMB zero adjustment must be considered in the ROM dumping code, but this only means that bank 0 should be read from `0x0000-0x3FFF` and not from `0x4000-0x7FFF` like other banks.

```
for bank in range(0, num_banks):
    write_byte(0x2100, bank)
    bank_start = 0x4000 if bank > 0 else 0x0000
    for addr in range(bank_start, bank_start + 0x4000):
        buf += read_byte(addr)
```

Listing 2: Python pseudo-code for MBC2 ROM dumping

Chapter 10

MBC3 mapper chip

MBC3 supports ROM sizes up to 16 Mbit (128 banks of `0x4000` bytes), and RAM sizes up to 256 Kbit (4 banks of `0x2000` bytes). It also includes a real-time clock (RTC) that can be clocked with a quartz crystal on the cartridge even when the Game Boy is powered down. The information in this section is based on my MBC3 research, and Pan Docs [3].

Chapter 11

MBC30 mapper chip

MBC30 is a variant of MBC3 used by Japanese Pokemon Crystal to support a larger ROM chip and a larger RAM chip. Featurewise MBC30 is almost identical to MBC3, but supports ROM sizes up to 32 Mbit (256 banks of `0x4000` bytes), and RAM sizes up to 512 Kbit (8 banks of `0x2000` bytes). Information in this section is based on my MBC30 research.

> ❗
>
> The circuit board of Japanese Pokemon Crystal includes a 1 Mbit RAM chip, but MBC30 is limited to 512 Kbit RAM. One of the RAM address pins is unused, so half of the RAM is wasted and is inaccessible without modifications. So, the game only uses 512 Kbit and there is a mismatch between accessible and the physical amounts of RAM.

Chapter 12

MBC5 mapper chip

The majority of games for Game Boy Color use the MBC5 chip. MBC5 supports ROM sizes up to 64 Mbit (512 banks of `0x4000` bytes), and RAM sizes up to 1 Mbit (16 banks of `0x2000` bytes). The information in this section is based on my MBC5 research, and The Cycle-Accurate Game Boy Docs [1].

12.1 MBC5 registers

Register 12.1: `0x0000–0x1FFF` - RAMG - MBC5 RAM gate register

`W-0`	`W-0`	`W-0`	`W-0`	`W-0`	`W-0`	`W-0`	`W-0`
`RAMG<7:0>`							
`bit 7`	`6`	`5`	`4`	`3`	`2`	`1`	`bit 0`

bit 7-0 **RAMG<7:0>**: RAM gate register
`0b00001010`= enable access to cartridge RAM
All other values disable access to cartridge RAM

The 8-bit MBC5 RAMG register works in a similar manner as MBC1 RAMG, but it is a full 8-bit register so upper bits matter when writing to it. Only `0b00001010` enables RAM access, and all other values (including `0b10001010` for example) disable access to RAM.

When RAM access is disabled, all writes to the external RAM area `0xA000–0xBFFF` are ignored, and reads return undefined values. Pan Docs recommends disabling RAM when it's not being accessed to protect the contents [3].

We don't know the physical implementation of RAMG, but it's certainly possible that the `0b00001010` bit pattern check is done at write time and the register actually consists of just a single bit.

Register 12.2: `0x2000–0x2FFF` - ROMB0 - MBC5 lower ROM bank register

`W-0`	`W-0`	`W-0`	`W-0`	`W-0`	`W-0`	`W-0`	`W-1`
`ROMB0<7:0>`							
`bit 7`	`6`	`5`	`4`	`3`	`2`	`1`	`bit 0`

bit 7-0 **ROMB0<7:0>**: Lower ROM bank register

The 8-bit ROMB0 register is used as the lower 8 bits of the ROM bank number when the CPU accesses the `0x4000–0x7FFF` memory area.

Register 12.3: `0x3000–0x3FFF` - ROMB1 - MBC5 upper ROM bank register

U	U	U	U	U	U	U	W-0
							ROMB1
bit 7	6	5	4	3	2	1	bit 0

bit 7-1 **Unimplemented**: Ignored during writes

bit 0 **ROMB1**: Upper ROM bank register

The 1-bit ROMB1 register is used as the most significant bit (bit 9) of the ROM bank number when the CPU accesses the `0x4000–0x7FFF` memory area.

Register 12.4: `0x4000–0x5FFF` - RAMB - MBC5 RAM bank register

U	U	U	U	W-0	W-0	W-0	W-0
				RAMB<3:0>			
bit 7	6	5	4	3	2	1	bit 0

bit 7-4 **Unimplemented**: Ignored during writes

bit 3-0 **RAMB<3:0>**: RAM bank register

The 4-bit RAMB register is used as the RAM bank number when the CPU accesses the `0xA000–0xBFFF` memory area.

Chapter 13

MBC6 mapper chip

MBC6 supports ROM sizes up to 16 Mbit (256 banks of `0x2000` bytes), and RAM sizes up to 4 Mbit (128 banks of `0x1000` bytes). The information in this section is based on my MBC6 research.

Chapter 14

MBC7

TODO.

Chapter 15

HuC-1 mapper chip

HuC-1 supports ROM sizes up to 8 Mbit (64 banks of `0x4000` bytes), and RAM sizes up to 256 Kbit (4 banks of `0x2000` bytes). It also includes a sensor and a LED for infrared communication. The information in this section is based on my HuC-1 research.

Chapter 16

HuC-3 mapper chip

HuC-3 supports ROM sizes up to 16 Mbit (128 banks of `0x4000` bytes), and RAM sizes up to 1 Mbit (16 banks of `0x2000` bytes). Like HuC-1, it includes support for infrared communication, but also includes a real-time-clock (RTC) and output pins used to control a piezoelectric buzzer. The information in this section is based on my HuC-3 research.

Chapter 17

MMM01

TODO.

Chapter 18

TAMA5

TODO.

Appendices

Appendix A

Instruction set tables

These tables include all the opcodes in the Sharp SM83 instruction set. The style and layout of these tables was inspired by the opcode tables available at pastraiser.com [4].

8-bit loads | 16-bit loads | 8-bit arithmetic | 16-bit arithmetic | Rotates, shifts, and bit operations | Control flow | Miscellaneous | Undefined

Table A.1: Sharp SM83 instruction set

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,nn	LD (BC),A	INC BC	INC B	DEC B	LD B,n	RLCA	LD (nn),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,n	RRCA
1x	STOP	LD DE,nn	LD (DE),A	INC DE	INC D	DEC D	LD D,n	RLA	JR e	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,n	RRA
2x	JR NZ,e	LD HL,nn	LD (HL+),A	INC HL	INC H	DEC H	LD H,n	DAA	JR Z,e	ADD HL,HL	LD A,(HL+)	DEC HL	INC L	DEC L	LD L,n	CPL
3x	JR NC,e	LD SP,nn	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),n	SCF	JR C,e	ADD HL,SP	LD A,(HL-)	DEC SP	INC A	DEC A	LD A,n	CCF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8x	ADD B	ADD C	ADD D	ADD E	ADD H	ADD L	ADD (HL)	ADD A	ADC B	ADC C	ADC D	ADC E	ADC H	ADC L	ADC (HL)	ADC A
9x	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC B	SBC C	SBC D	SBC E	SBC H	SBC L	SBC (HL)	SBC A
Ax	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
Cx	RET NZ	POP BC	JP NZ,nn	JP nn	CALL NZ,nn	PUSH BC	ADD n	RST 0x00	RET Z	RET	JP Z,nn	CB op	CALL Z,nn	CALL nn	ADC n	RST 0x08
Dx	RET NC	POP DE	JP NC,nn		CALL NC,nn	PUSH DE	SUB n	RST 0x10	RET C	RETI	JP C,nn		CALL C,nn		SBC n	RST 0x18
Ex	LDH (n),A	POP HL	LDH (C),A			PUSH HL	AND n	RST 0x20	ADD SP,e	JP HL	LD (nn),A				XOR n	RST 0x28
Fx	LDH A,(n)	POP AF	LDH A,(C)	DI		PUSH AF	OR n	RST 0x30	LD HL,SP+e	LD SP,HL	LD A,(nn)	EI			CP n	RST 0x38

n	unsigned 8-bit immediate data
nn	unsigned 16-bit immediate data
e	signed 8-bit immediate data
r	signed 8-bit immediate data, relative to PC

Table A.2: Sharp SM83 CB-prefixed instructions

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	RLC B	RLC C	RLC D	RLC E	RLC H	RLC L	RLC (HL)	RLC A	RRC B	RRC C	RRC D	RRC E	RRC H	RRC L	RRC (HL)	RRC A
1x	RL B	RL C	RL D	RL E	RL H	RL L	RL (HL)	RL A	RR B	RR C	RR D	RR E	RR H	RR L	RR (HL)	RR A
2x	SLA B	SLA C	SLA D	SLA E	SLA H	SLA L	SLA (HL)	SLA A	SRA B	SRA C	SRA D	SRA E	SRA H	SRA L	SRA (HL)	SRA A
3x	SWAP B	SWAP C	SWAP D	SWAP E	SWAP H	SWAP L	SWAP (HL)	SWAP A	SRL B	SRL C	SRL D	SRL E	SRL H	SRL L	SRL (HL)	SRL A
4x	BIT 0,B	BIT 0,C	BIT 0,D	BIT 0,E	BIT 0,H	BIT 0,L	BIT 0,(HL)	BIT 0,A	BIT 1,B	BIT 1,C	BIT 1,D	BIT 1,E	BIT 1,H	BIT 1,L	BIT 1,(HL)	BIT 1,A
5x	BIT 2,B	BIT 2,C	BIT 2,D	BIT 2,E	BIT 2,H	BIT 2,L	BIT 2,(HL)	BIT 2,A	BIT 3,B	BIT 3,C	BIT 3,D	BIT 3,E	BIT 3,H	BIT 3,L	BIT 3,(HL)	BIT 3,A
6x	BIT 4,B	BIT 4,C	BIT 4,D	BIT 4,E	BIT 4,H	BIT 4,L	BIT 4,(HL)	BIT 4,A	BIT 5,B	BIT 5,C	BIT 5,D	BIT 5,E	BIT 5,H	BIT 5,L	BIT 5,(HL)	BIT 5,A
7x	BIT 6,B	BIT 6,C	BIT 6,D	BIT 6,E	BIT 6,H	BIT 6,L	BIT 6,(HL)	BIT 6,A	BIT 7,B	BIT 7,C	BIT 7,D	BIT 7,E	BIT 7,H	BIT 7,L	BIT 7,(HL)	BIT 7,A
8x	RES 0,B	RES 0,C	RES 0,D	RES 0,E	RES 0,H	RES 0,L	RES 0,(HL)	RES 0,A	RES 1,B	RES 1,C	RES 1,D	RES 1,E	RES 1,H	RES 1,L	RES 1,(HL)	RES 1,A
9x	RES 2,B	RES 2,C	RES 2,D	RES 2,E	RES 2,H	RES 2,L	RES 2,(HL)	RES 2,A	RES 3,B	RES 3,C	RES 3,D	RES 3,E	RES 3,H	RES 3,L	RES 3,(HL)	RES 3,A
Ax	RES 4,B	RES 4,C	RES 4,D	RES 4,E	RES 4,H	RES 4,L	RES 4,(HL)	RES 4,A	RES 5,B	RES 5,C	RES 5,D	RES 5,E	RES 5,H	RES 5,L	RES 5,(HL)	RES 5,A
Bx	RES 6,B	RES 6,C	RES 6,D	RES 6,E	RES 6,H	RES 6,L	RES 6,(HL)	RES 6,A	RES 7,B	RES 7,C	RES 7,D	RES 7,E	RES 7,H	RES 7,L	RES 7,(HL)	RES 7,A
Cx	SET 0,B	SET 0,C	SET 0,D	SET 0,E	SET 0,H	SET 0,L	SET 0,(HL)	SET 0,A	SET 1,B	SET 1,C	SET 1,D	SET 1,E	SET 1,H	SET 1,L	SET 1,(HL)	SET 1,A
Dx	SET 2,B	SET 2,C	SET 2,D	SET 2,E	SET 2,H	SET 2,L	SET 2,(HL)	SET 2,A	SET 3,B	SET 3,C	SET 3,D	SET 3,E	SET 3,H	SET 3,L	SET 3,(HL)	SET 3,A
Ex	SET 4,B	SET 4,C	SET 4,D	SET 4,E	SET 4,H	SET 4,L	SET 4,(HL)	SET 4,A	SET 5,B	SET 5,C	SET 5,D	SET 5,E	SET 5,H	SET 5,L	SET 5,(HL)	SET 5,A
Fx	SET 6,B	SET 6,C	SET 6,D	SET 6,E	SET 6,H	SET 6,L	SET 6,(HL)	SET 6,A	SET 7,B	SET 7,C	SET 7,D	SET 7,E	SET 7,H	SET 7,L	SET 7,(HL)	SET 7,A

Appendix B

Memory map tables

Table B.1: `0xFFxx` registers: `0xFF00–0xFF1F`

	bit 7	6	5	4	3	2	1	bit 0
0xFF00 P1			P15 buttons	P14 d-pad	P13 ⬇ start	P12 ⬆ select	P11 ⬅ B	P10 ➡ A
0xFF01 SB	SB<7:0>							
0xFF02 SC	SIO_EN						SIO_FAST	SIO_CLK
0xFF03								
0xFF04 DIV	DIVH<7:0>							
0xFF05 TIMA	TIMA<7:0>							
0xFF06 TMA	TMA<7:0>							
0xFF07 TAC						TAC_EN	TAC_CLK<1:0>	
0xFF08								
0xFF09								
0xFF0A								
0xFF0B								
0xFF0C								
0xFF0D								
0xFF0E								
0xFF0F IF				IF_JOYPAD	IF_SERIAL	IF_TIMER	IF_STAT	IF_VBLANK
0xFF10 NR10								
0xFF11 NR11								
0xFF12 NR12								
0xFF13 NR13								
0xFF14 NR14								
0xFF15								
0xFF16 NR21								
0xFF17 NR22								
0xFF18 NR23								
0xFF19 NR24								
0xFF1A NR30								
0xFF1B NR31								
0xFF1C NR32								
0xFF1D NR33								
0xFF1E NR34								
0xFF1F								
	bit 7	6	5	4	3	2	1	bit 0

Table B.2: `0xFFxx` registers: `0xFF20`–`0xFF3F`

	bit 7	6	5	4	3	2	1	bit 0
0xFF20 NR41								
0xFF21 NR42								
0xFF22 NR43								
0xFF23 NR44								
0xFF24 NR50								
0xFF25 NR51								
0xFF26 NR52								
0xFF27								
0xFF28								
0xFF29								
0xFF2A								
0xFF2B								
0xFF2C								
0xFF2D								
0xFF2E								
0xFF2F								
0xFF30 WAV00								
0xFF31 WAV01								
0xFF32 WAV02								
0xFF33 WAV03								
0xFF34 WAV04								
0xFF35 WAV05								
0xFF36 WAV06								
0xFF37 WAV07								
0xFF38 WAV08								
0xFF39 WAV09								
0xFF3A WAV10								
0xFF3B WAV11								
0xFF3C WAV12								
0xFF3D WAV13								
0xFF3E WAV14								
0xFF3F WAV15								
	bit 7	6	5	4	3	2	1	bit 0

Table B.3: `0xFFxx` registers: `0xFF40–0xFF5F`

	bit 7	6	5	4	3	2	1	bit 0
0xFF40 LCDC	LCD_EN	WIN_MAP	WIN_EN	TILE_SEL	BG_MAP	OBJ_SIZE	OBJ_EN	BG_EN
0xFF41 STAT		INTR_LYC	INTR_M2	INTR_M1	INTR_M0	LYC_STAT	LCD_MODE<1:0>	
0xFF42 SCY								
0xFF43 SCX								
0xFF44 LY								
0xFF45 LYC								
0xFF46 DMA	DMA<7:0>							
0xFF47 BGP								
0xFF48 OBP0								
0xFF49 OBP1								
0xFF4A WY								
0xFF4B WX								
0xFF4C ????								
0xFF4D KEY1	KEY1_FAST							KEY1_EN
0xFF4E								
0xFF4F VBK							VBK<1:0>	
0xFF50 BOOT								BOOT_OFF
0xFF51 HDMA1								
0xFF52 HDMA2								
0xFF53 HDMA3								
0xFF54 HDMA4								
0xFF55 HDMA5								
0xFF56 RP								
0xFF57								
0xFF58								
0xFF59								
0xFF5A								
0xFF5B								
0xFF5C								
0xFF5D								
0xFF5E								
0xFF5F								
	bit 7	6	5	4	3	2	1	bit 0

Table B.4: `0xFFxx` registers: `0xFF60–0xFF7F,0xFFFF`

	bit 7	6	5	4	3	2	1	bit 0
0xFF60								
0xFF61								
0xFF62								
0xFF63								
0xFF64								
0xFF65								
0xFF66								
0xFF67								
0xFF68 BCPS								
0xFF69 BCPD								
0xFF6A OCPS								
0xFF6B OCPD								
0xFF6C ????								
0xFF6D								
0xFF6E								
0xFF6F								
0xFF70 SVBK							SVBK<1:0>	
0xFF71								
0xFF72 ????								
0xFF73 ????								
0xFF74 ????								
0xFF75 ????								
0xFF76 PCM12	PCM12_CH2				PCM12_CH1			
0xFF77 PCM34	PCM34_CH4				PCM34_CH3			
0xFF78								
0xFF79								
0xFF7A								
0xFF7B								
0xFF7C								
0xFF7D								
0xFF7E								
0xFF7F								
0xFFFF IE	IE_UNUSED<2:0>			IE_JOYPAD	IE_SERIAL	IE_TIMER	IE_STAT	IE_VBLANK
	bit 7	6	5	4	3	2	1	bit 0

Appendix C

Game Boy external bus

C.1 Bus timings

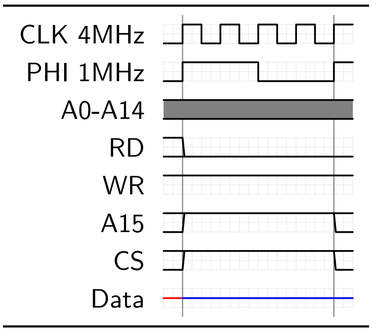
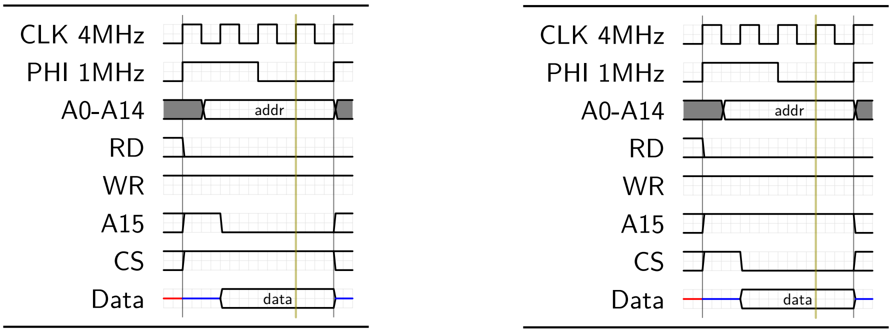
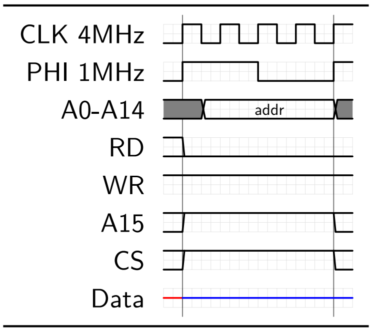


Figure C.1: External bus idle machine cycle



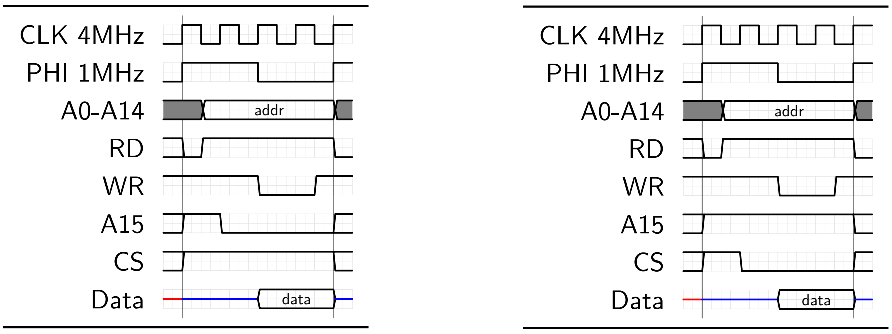
(a) `0x0000–0x7FFF`[1]

(b) `0xA000–0xFDFF`



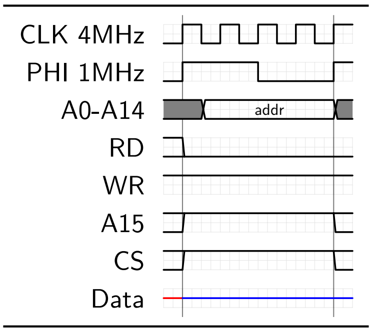
(c) `0xFE00–0xFFFF`

Figure C.2: External bus CPU read machine cycles



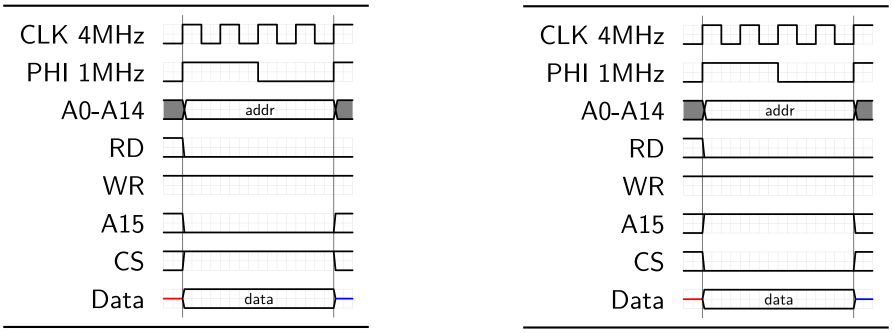
(a) `0x0000–0x7FFF`[2]

(b) `0xA000–0xFDFF`



(c) `0xFE00–0xFFFF`

Figure C.3: External bus timings for CPU write cycles



(a) `0x0000–0x7FFF`[3]

(b) `0xA000–0xFFFF`

Figure C.4: External bus timings for OAM DMA read cycles

---

[1] Does not apply to `0x0000–0x00FF` reads while the boot ROM is enabled. Boot ROM accesses do not affect the external bus, so it is in the idle state.

[2] Does not apply to `0x0000–0x00FF` writes while the boot ROM is enabled. Boot ROM accesses do not affect the external bus, so it is in the idle state.

[3] Does not apply to `0x0000–0x00FF` accesses while the boot ROM is enabled. Boot ROM accesses do not affect the external bus, so it is in the idle state.

Appendix D

Chip pinouts

D.1 CPU chips

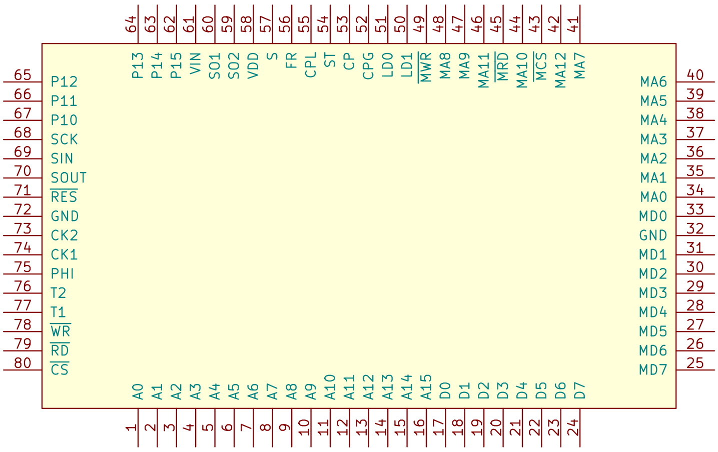


Figure D.1: DMG/SGB CPU (Sharp QFP080-P-1420)

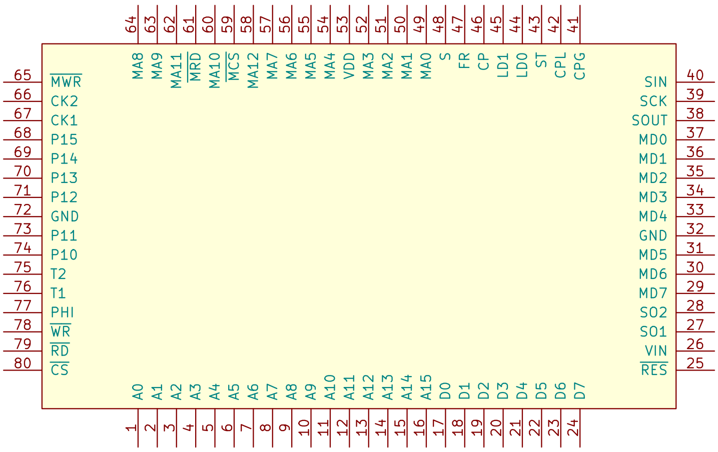


Figure D.2: MGB/SGB2 CPU (Sharp QFP080-P-1420)

D.2 Cartridge chips

Figure D.3: MBC1 (Sharp SOP24-P-450) [7]

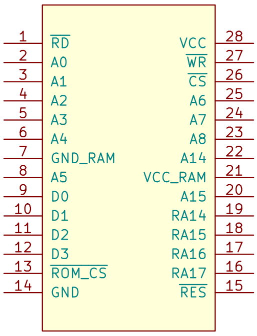


Figure D.4: MBC2 (Sharp SOP28-P-450) [8]

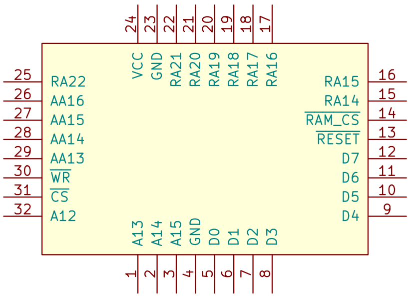


Figure D.5: MBC5 (Sharp QFP32-P-0707)

Bibliography

[1] Antonio Niño Díaz (AntonioND). The Cycle-Accurate Game Boy Docs.
`https://github.com/AntonioND/giibiiadvance/tree/master/docs`.

[2] gekkio. Dumping the Super Game Boy 2 boot ROM.
`https://gekkio.fi/blog/2015/dumping-the-super-game-boy-2-boot-rom/`.

[3] Pan of ATX, Marat Fayzullin, Felber Pascal, Robson Paul, and Korth Martin. Pan Docs - Everything You Always Wanted To Know About GAMEBOY.
`http://bgb.bircd.org/pandocs.htm`.

[4] Gameboy CPU (LR35902) instruction set.
`http://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html`.

[5] Sharp. SM8311/SM8313/SM8314/SM8315 - 8-Bit Single-Chip Microcomputers (Controllers For Home Appliances).

[6] Costis Sideris. The quest for dumping GameBoy Boot ROMs!
`http://www.its.caltech.edu/~costis/sgb_hack/`.

[7] Tauwasser. MBC1 - Tauwasser's Wiki.
`https://wiki.tauwasser.eu/view/MBC1`.

[8] Tauwasser. MBC2 - Tauwasser's Wiki.
`https://wiki.tauwasser.eu/view/MBC2`.