Game Boy: Complete Technical Reference

gekkio
`https://gekkio.fi`

September 6, 2018

Revision 50

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.

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

> ❶
> 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

Game Boy CPU and the Sharp LR35902 instruction set

Chapter 1

Sharp LR35902 instruction set

1.1 8-bit load and store instructions

1.2 16-bit load and store instructions

1.3 8-bit arithmetic instructions

1.4 16-bit arithmetic instructions

1.5 Rotate, shift, and bit operation instructions

1.6 Control flow instructions

JP nn

Unconditional jump to the absolute address specified by the operand nn.

Opcode + data	0b11000011 + LSB of nn + MSB of nn
Length	3 bytes
Duration	4 machine cycles
Flags	-
Timing	Purpose — Decode │ LSB of nn │ MSB of nn │ Internal delay │ Decode Memory — Read: PC │ Read: PC+1 │ Read: PC+2 │ ▓▓▓ │ Read: nn

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 register HL.

Opcode	0b11101001
Length	1 bytes
Duration	1 machine cycle
Flags	-
Timing	Purpose — Decode │ Decode Memory — Read: PC │ Read: HL

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 operand `nn`, depending on the condition `cc`.
   Note that the operand (absolute address) is read even when the condition is false!

Opcode + data	`0b110cc010` + 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)

Purpose	Decode	LSB of nn	MSB of nn	Decode
Memory	Read: PC	Read: PC+1	Read: PC+2	Read: PC+3

Timing (cc=true)

Purpose	Decode	LSB of nn	MSB of nn	Internal delay	Decode
Memory	Read: PC	Read: PC+1	Read: PC+2		Read: nn

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 r

Unconditional jump to the relative address specified by the signed operand `r`.
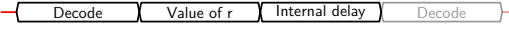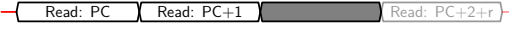
Opcode + data	`0b00011000` + offset r
Length	2 bytes
Duration	3 machine cycles
Flags	-

Timing

Purpose	Decode	Value of r	Internal delay	Decode
Memory	Read: PC	Read: PC+1		Read: PC+2+r

Pseudocode

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

JR cc, r

Conditional jump to the relative address specified by the signed operand `r`, depending on the condition `cc`.
   Note that the operand (relative address offset) is read even when the condition is false!
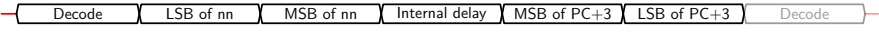
Opcode + data	`0b001cc000` + offset r
Length	2 bytes
Duration	2 machine cycles (cc=false), or 3 machine cycles (cc=true)
Flags	-

Timing (cc=false)

Purpose	Decode	Value of r	Decode
Memory	Read: PC	Read: PC+1	Read: PC+2

| Timing (cc=true) | Purpose | Decode | Value of r | Internal delay | Decode | |
| | Memory | Read: PC | Read: PC+1 | | Read: PC+2+r | |

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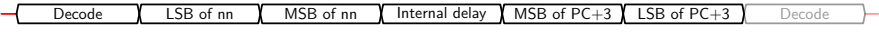
CALL nn

Unconditional function call to the absolute address specified by the operand nn.

Opcode + data   0b11001101 + LSB of nn + MSB of nn

Length   3 bytes

Duration   6 machine cycles

Flags   -

| Timing | Purpose | Decode | LSB of nn | MSB of nn | Internal delay | MSB of PC+3 | LSB of PC+3 | Decode |
| | Memory | Read: PC | Read: PC+1 | Read: PC+2 | | Write: SP-1 | Write: SP-2 | Read: nn |

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 operand nn, depending on the condition cc.
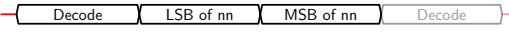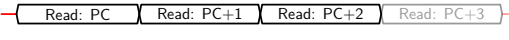   Note that the operand (absolute address) is read even when the condition is false!

Opcode + data   0b110cc100 + 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) | Purpose | Decode | LSB of nn | MSB of nn | Decode | | |
| | Memory | Read: PC | Read: PC+1 | Read: PC+2 | Read: PC+3 | | |

| Timing (cc=true) | Purpose | Decode | LSB of nn | MSB of nn | Internal delay | MSB of PC+3 | LSB of PC+3 | Decode |
| | Memory | Read: PC | Read: PC+1 | Read: PC+2 | | Write: SP-1 | Write: SP-2 | Read: nn |

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 function.

Opcode		0b11001001

Length		1 byte

Duration		4 machine cycles

Flags		-

Timing

Purpose: | Decode | LSB of PC | MSB of PC | Internal delay | Decode |

Memory: | Read: PC | Read: SP | Read: SP+1 | | Read: new PC |
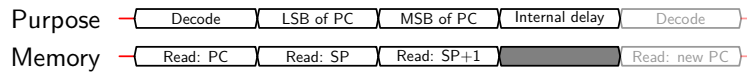
Pseudocode
```
opcode = read(PC++)
if opcode == 0xC9:
  PC = unsigned_16(lsb=read(SP++), msb=read(SP++))
```

RET cc

Conditional return from function, depending on the condition cc.

Opcode		0b110cc000

Length		1 byte

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

Flags		-

Timing (cc=false)

Purpose: | Decode | Internal delay | Decode |

Memory: | Read: PC | | Read: PC+1 |

Timing (cc=true)

Purpose: | Decode | Internal delay | LSB of PC | MSB of PC | Internal delay | Decode |

Memory: | Read: PC | | Read: SP | Read: SP+1 | | Read: new PC |
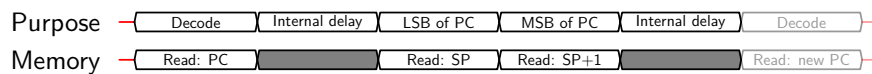
Pseudocode
```
opcode = read(PC++)
if opcode in [0xC0, 0xD0, 0xC8, 0xD8]:
  if F.check_condition(cc):
    PC = unsigned_16(lsb=read(SP++), msb=read(SP++))
```

RETI

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

Opcode		0b11011001

Length		1 byte

Duration		4 machine cycles

Flags		-

Timing

Purpose: | Decode | LSB of PC | MSB of PC | Internal delay | Decode |

Memory: | Read: PC | Read: SP | Read: SP+1 | | Read: new PC |
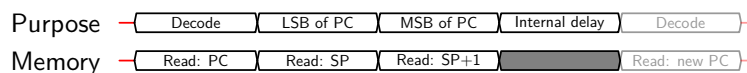
Pseudocode
```
opcode = read(PC++)
if opcode == 0xD9:
  PC = unsigned_16(lsb=read(SP++), msb=read(SP++))
  IME = 1
```

RST n

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

Opcode   0b11xxx111

Length   1 byte

Duration   4 machine cycles

Flags   -

Timing

Purpose ⊣ Decode | Internal delay | MSB of PC+1 | LSB of PC+1 | Decode ⊢

Memory ⊣ Read: PC | | Write: SP-1 | Write: SP-2 | Read: new PC ⊢
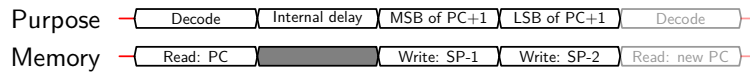
Pseudocode
```
opcode = read(PC++)
if opcode in [0xC7, 0xD7, 0xE7, 0xF7, 0xCF, 0xDF, 0xEF, 0xFF]:
  n = rst_address(opcode)
  write(--SP, msb(PC))
  write(--SP, lsb(PC))
  PC = unsigned_16(lsb=n, msb=0x00)
```

1.7   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

Length   1 byte

Duration   1 machine cycle

Flags   -

Timing

Purpose ⊣ Decode | Decode ⊢

Memory ⊣ Read: PC | Read: PC+1 ⊢
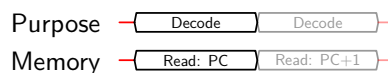
Pseudocode
```
opcode = read(PC++)
if opcode == 0xF3:
   IME = 0
```

EI

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

Opcode   0b11111011

Length   1 byte

Duration   1 machine cycle (+ 1 machine cycle for the effect)

Flags   -

Timing

Purpose ⊣ Decode | Decode ⊢

Memory ⊣ Read: PC | Read: PC+1 ⊢
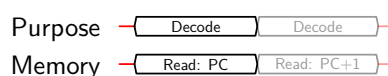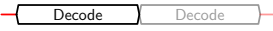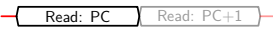
Pseudocode
```
opcode = read(PC++)
if opcode == 0xFB:
   IME_scheduled = true
```

CCF

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

Opcode   `0b00111111`

Length   1 byte

Duration   1 machine cycle

Flags   N = 0, H = 0, C = ★

Timing

Purpose	Decode	Decode
Memory	Read: PC	Read: PC+1

Pseudocode

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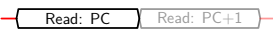
SCF

Sets the carry flag, and clears the N and H flags.
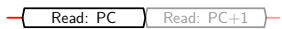
Opcode   `0b00110111`

Length   1 byte

Duration   1 machine cycle

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

Timing

Purpose	Decode	Decode
Memory	Read: PC	Read: PC+1

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`

Length   1 byte

Duration   1 machine cycle

Flags   -

Timing

Purpose	Decode	Decode
Memory	Read: PC	Read: PC+1

Pseudocode

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

DAA

Opcode   0b00100111

Length   1 byte

Duration   1 machine cycle

Flags   Z = ★, H = 0, C = ★

Timing
Purpose	Decode	Decode	
Memory	Read: PC	Read: PC+1	

CPL

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

Opcode   0b00101111

Length   1 byte

Duration   1 machine cycle

Flags   N = 1, H = 1

Timing
Purpose	Decode	Decode	
Memory	Read: PC	Read: PC+1	

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

Part II

Game Boy CPU peripherals and features

Chapter 2

Boot ROM

Register 2.1: 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.

Chapter 3

DMA (Direct Memory Access)

3.1 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 LR35902 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 3.1: 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 3.1: 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 4

PPU (Picture Processing Unit)

Register 4.1: `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 4.2: `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 4.3: `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 4.4: `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 4.5: `0xFF43` - 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 4.6: `0xFF44` - 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 5

Port 1 (Joypad, Super Game Boy communication)

Register 5.1: `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 6

Serial communication

Register 6.1: `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 6.2: `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 7

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 [3], and Pan Docs [2].

7.1 MBC1 registers

> ℹ️
>
> 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 7.1: `0x0000–0x1FFF` - RAM_EN - MBC1 RAM enable register

U	U	U	U	W–0	W–0	W–0	W–0
				RAM_EN<3:0>			
bit 7	6	5	4	3	2	1	bit 0

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

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

The RAM_EN 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 `0xFF`. Pan Docs recommends disabling RAM when it's not being accessed to protect the contents [2].

> 🧩
>
> We don't know the physical implementation of RAM_EN, 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 7.2: `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 7.3: `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 7.4: `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 [2], 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.

7.2   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 0 the bank number is always 0, but in MODE 1 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 7.1: Mapping of physical ROM address bits in MBC1 carts

	Bank number		Address within bank
ROM address bits	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 `0x0000–0x–3FFF, MODE = 0b0`)
      `0b 00 00000` (= 0 = `0x00`)

Effective ROM bank number (reading `0x0000–0x–3FFF, MODE = 0b1`)
      `0b 01 00000` (= 32 = `0x40`)

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`)

7.3   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 RAM_EN register, all reads return `0xFF` 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 0 the BANK2 register value is not used, so the first RAM bank is always mapped to the `0xA000–0xBFFF` area. In MODE 1 the BANK2 register value is used as the bank number.

Table 7.2: Mapping of physical RAM address bits in MBC1 carts

	Bank number	Address within bank
RAM address bits	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`, RAM_EN 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`)

7.4   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 1 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 1 guarantees that all ROM accesses, including accesses to `0x0000–0x3FFF`, use the BANK2 register value.

Table 7.3: Mapping of physical ROM address bits in MBC1 multicarts

	Bank number		Address within bank
ROM address bits	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.

7.5   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 1 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.

```python
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 8

MBC2

TODO.

Chapter 9

MBC3

TODO.

Chapter 10

MBC5

TODO.

Chapter 11

MBC6

TODO.

Chapter 12

MBC7

TODO.

Chapter 13

HuC-1

TODO.

Chapter 14

HuC-3

TODO.

Chapter 15

MMM01

TODO.

Chapter 16

TAMA5

TODO.

Appendices

Appendix A

Instruction set tables

These tables include all the opcodes in the Sharp LR35902 instruction set. The style and layout of these tables was inspired by the opcode tables available at pastraiser.com [1].

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

Table A.1: Sharp LR35902 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 r	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,n	RRA
2x	JR NZ,r	LD HL,nn	LD (HL+),A	INC HL	INC H	DEC H	LD H,n	DAA	JR Z,r	ADD HL,HL	LD A,(HL+)	DEC HL	INC L	DEC L	LD L,n	CPL
3x	JR NC,r	LD SP,nn	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),n	SCF	JR C,r	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	LD (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	LD 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 LR35902 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

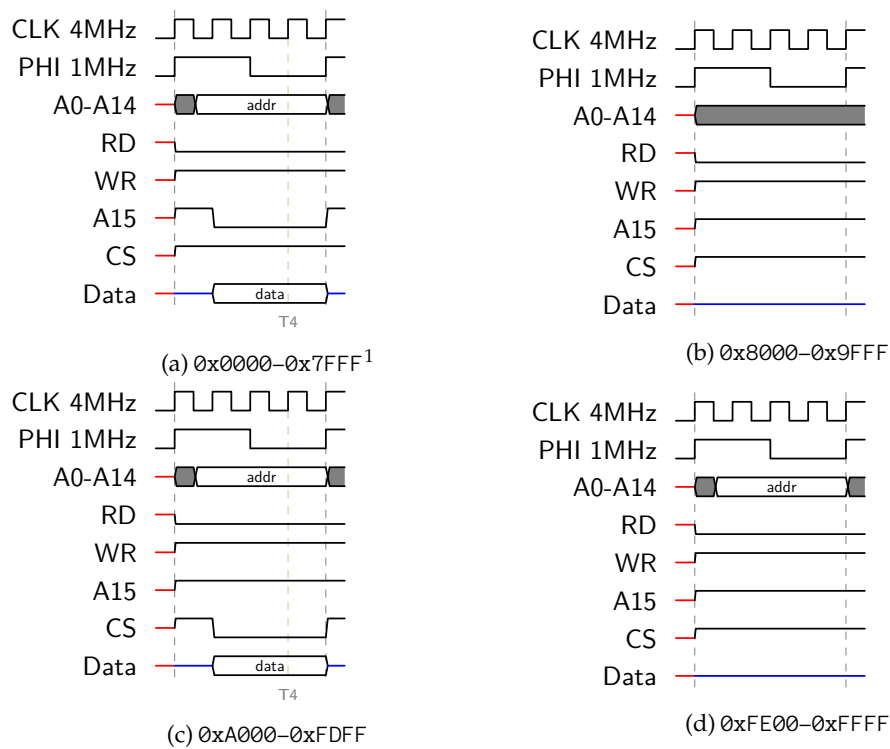(a) 0x0000–0x7FFF[1]

(b) 0x8000–0x9FFF

(c) 0xA000–0xFDFF

(d) 0xFE00–0xFFFF

Figure C.1: External bus timings for CPU read cycles

---

[1]Does not apply to 0x0000–0x00FF accesses while boot ROM is enabled

(a) 0x0000–0x7FFF[2]

(b) 0x8000–0x9FFF

(c) 0xA000–0xFDFF
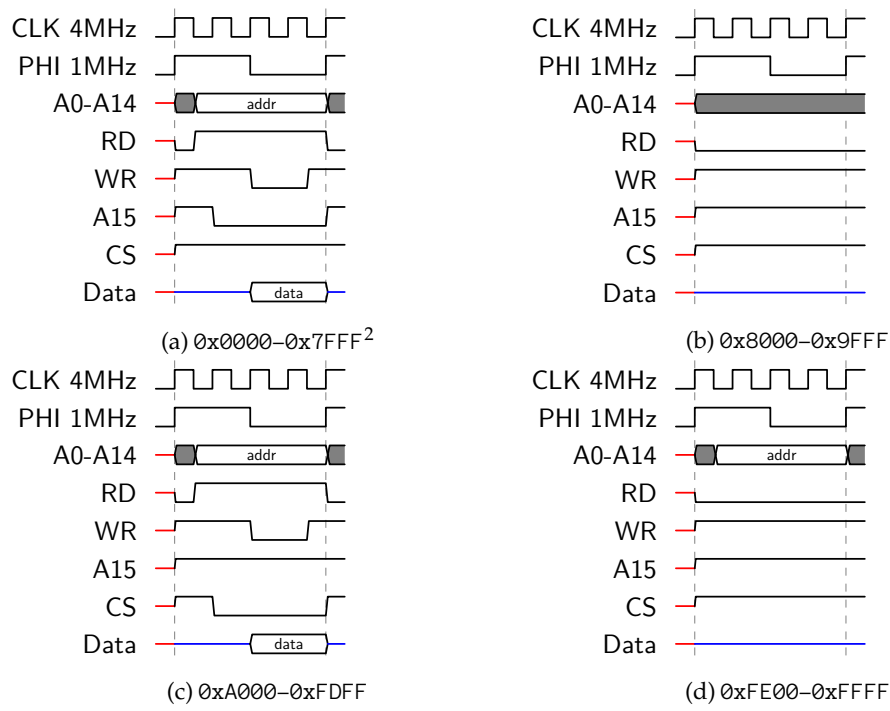
(d) 0xFE00–0xFFFF

Figure C.2: External bus timings for CPU write cycles

---

[2]Does not apply to 0x0000–0x00FF accesses while boot ROM is enabled

Bibliography

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

[2] 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`.

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