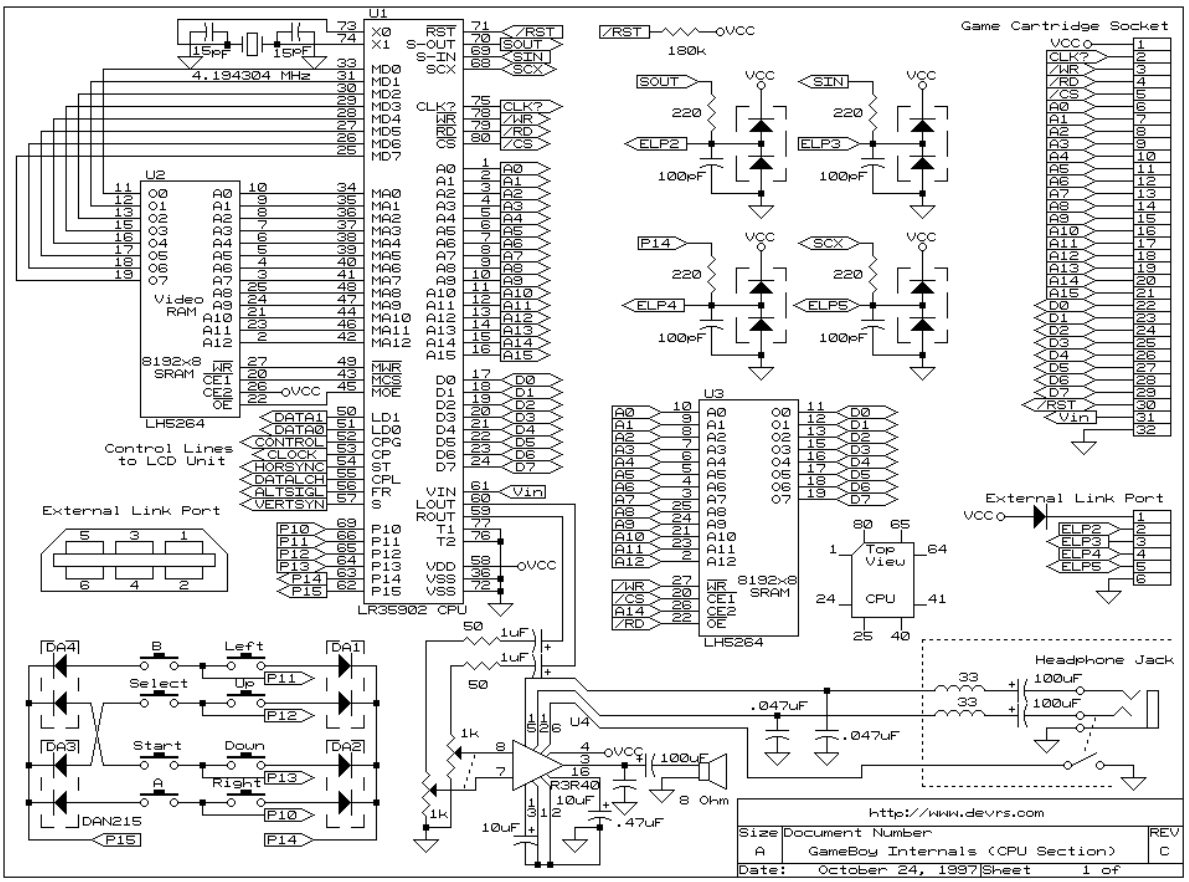
Programmation de la console de jeu Gameboy (Nintendo)

J.-M Friedt, 17 janvier 2016– 1 point/question, tous documents autorisés.

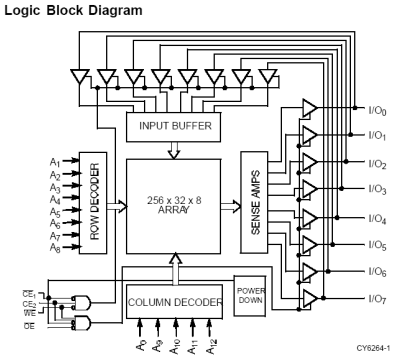
1 Aspects matériels : interfaçage à la console GameBoy

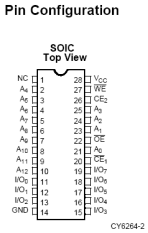
La console de jeu Gameboy a été commercialisée par Nintendo entre 1989 et 2003, avec près de 120 millions d'unités vendues dans le monde. Les chances d'en retrouver une dans une cave ne sont donc pas négligeables : il s'agit d'une excellente plateforme introductive au développement de systèmes embarqués. Le processeur, un clone de Z80 [1], est parfaitement documenté, ainsi que l'architecture des périphériques autour du processeur. Une chaîne de compilation est disponible à `https://github.com/gheja/gbdk.git` et un émulateur, `gngb` [2], permet de tester ses programmes sur PC.

Le schéma bloc [3] de la console de jeu est décrit ci-dessous :



Nous rappelons par ailleurs la logique d'accès à un composant de RAM statique classiquement utilisé à cette époque, la 6264 (broche à broche compatible avec le composant LH5264 de Sharp) :





Truth Table

$\overline{CE_1}$	CE_2	$\overline{WE}$	$\overline{OE}$	Input/Output	Mode
H	X	X	X	High Z	Deselect/Power-Down
X	L	X	X	High Z	Deselect
L	H	H	L	Data Out	Read
L	H	L	X	Data In	Write
L	H	H	H	High Z	Deselect

Ainsi, on constate que les broches $\overline{CE_i}$ (*Chip Enable*)) activent la RAM pour passer son bus de données de haute impédance (la RAM subit l'état imposé par les autres périphériques) à basse impédance si la lecture est en plus requise ($\overline{OE}$ pour *Output Enable*) ou pour s'en approprier la valeur ($\overline{OWR}$ pour indiquer que le processeur est en train d'écrire une valeur à destination de la RAM).

1. Quel est l'état par défaut de la broche de ré-initialisation du microprocesseur $\overline{RST}$? justifier.
2. Combien d'octets peut adresser un processeur Z80 ? justifier.
3. Sur combien de bit(s) s'effectue chaque transaction entre les périphériques et le processeur ? justifier.
4. Quelles sont les plages d'adresses permettant d'accéder à la RAM interne à la console de jeu ?
5. Comment câbler le bus d'adresse d'une ROM (brochage supposé identique à celui de la RAM) sur le bus reliant la cartouche (bus en haut à droite) au processeur, sachant que le premier opcode exécuté par un Z80 se trouve à l'adresse 0x0000 ? Justifier.

1. R.S. Gaonkar, *The Z80 Microprocessor : Architecture, Interfacing, Programming*, Prentice Hall (1992)
2. `http://m.peponas.free.fr/gngb/`
3. `http://www.devrs.com/gb/files/gameboy1.gif`

6. Supposons que nous voulions lire la mesure d'un convertisseur analogique-numérique câblé sur la cartouche contenant la ROM du jeu. Comment adresser ce périphérique pour ne pas entrer en conflit avec les accès mémoire ?
7. Quelle(s) ligne(s) de programme en C permettrait d'accéder au convertisseur analogique-numérique ?

2 Copie d'une image de la RAM vers la mémoire vidéo (VRAM)

Il arrive que nous voulions copier une zone mémoire – contenant par exemple une image – vers l'emplacement représentatif de l'image affichée à l'écran. Il s'agit d'une opération de tracé *bitmap*, à l'opposé du tracé *vectoriel* que nous verrons plus loin.

Le temps de transfert de la zone mémoire limite le taux de raffraîchissement de l'écran : nous devons être capables de transférer l'intégralité des informations entre deux raffraîchissements de l'écran, faute de quoi l'image clignote, offrant un effet visuel désagréable.

Le Z80 de la console de jeu Gameboy était cadencée à environ [4] 4 MHz (que nous prendrons comme valeur pour simplifier les calculs). Le taux de raffraîchissement de l'écran, de résolution 160×144 pixels, est de 59,73 Hz, que nous approximerons à 60 Hz. Chaque pixel est représenté par un bit – allumé ou éteint.

Le Z80 possède 8 registres généraux – A, B, C, D, E, F, H, L – qui peuvent être concaténés en registres 16 bits – BC, DE, HL. A est l'accumulateur, F le registres des drapeaux d'état de l'ALU du microcontrôleur. Basé sur `http://www.phy.davidson.edu/FacHome/dmb/py310/Z80.Instruction%20set.pdf`, nous connaissons la durée d'exécution de chaque instruction assembleur du Z80, donnée en cycles d'horloge (cycle T). Pour reprendre la documentation de Zilog sur le Z80 :

> "All instructions are a series of basic operations. Each of these operations can take from three to six clock periods to complete, or they can be lengthened to synchronize the CPU to the speed of external devices. These clock periods are referred to as time (T) cycles, and the operations are referred to as machine (M) cycles. Figure 4 shows how a typical instruc- tion is a series of specific M and T cycles. In Figure 4, this instruction consists of the three machine cycles M1, M2, and M3. The first machine cycle of any instruction is a fetch cycle that is four, five, or six T cycles long (unless lengthened by the WAIT signal, which is described in the next section). The fetch cycle (M1) is used to fetch the op code of the next instruction to be executed. Subsequent machine cycles move data between the CPU and memory or I/O devices, and they can feature anywhere from three to five T cycles ... "

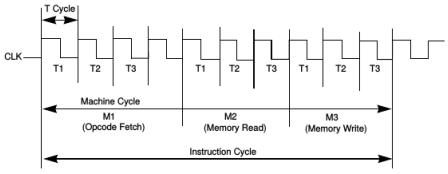


Figure 4. Basic CPU Timing Example

```
lp:ld   a,b     ;Test BC,   78
   or   c       ;If BC==0   B1
   ret  z       ;Return     C8
   ld   a,(hl)  ;A <- (HL)  1A
   ld   (de),a  ;(DE) <- A  77
   inc  de      ;Inc DE     13
   inc  hl      ;Inc HL     23
   dec  bc      ;Dec BC     0B
   jp   lp  ;Repeat C3 00 10
```

Le code de droite fournit un exemple de copie de données entre deux blocs mémoire, avec en fin de ligne l'opcode correspondant à chaque instruction, tandis que nous fournissons ci-dessous les extraits de datasheet [a] utiles :

a. `www.zilog.com/manage_directlink.php?filepath=docs/z80/um0080`

1. DE contient l'adresse de début du tableau cible, HL l'adresse du tableau source. Combien de cycles d'horloge faut-il pour copier 1 octet ?
2. BC contient le nombre d'octets à copier. Combien de temps faut-il pour copier BC octets ?
3. Quelle taille d'image maximale peut-on copier compte tenu du taux de raffraîchissement de l'écran ? Est-ce suffisant pour raffraîchir tout l'écran ?
4. Commenter sur le temps d'exécution de chaque instruction sur cette architecture CISC développée en 1976, et comparer à l'architecture RISC des AVR8 vue en cours.
5. Justifier des 3 premières lignes du code. Pourquoi ne peut-on pas simplement quitter la boucle après avoir décrémenté BC ? Peut-on optimiser ce code ?

DEC ss

Operation

ss ← ss – 1

Op Code

DEC

Operand

ss

0	0	s	s	1	0	1	1

Description

The contents of register pair ss (any of the register pairs BC, DE, HL, or SP) are decremented. In the assembled object code, operand *ss* is specified as follows:

Register Pair	ss
BC	00
DE	01
HL	10
SP	11

M Cycles	T States	4 MHz E.T.
1	6	1.50

Condition Bits Affected

None.

Example

If register pair HL contains `1001h`, then upon the execution of an DEC *HL* instruction, HL contains `1000h`.

LD A, R

Operation

A ← R

Op Code

LD

Operands

A, R

1	1	1	0	1	1	0	1	ED
0	1	0	1	1	1	1	1	5F

Description

The contents of Memory Refresh Register R are loaded to the Accumulator.

M Cycles	T States	MHz E.T.
2	9 (4, 5)	2.25

Condition Bits Affected

S is set if, R-Register is negative; otherwise, it is reset.

Z is set if the R Register is 0; otherwise, it is reset.

H is reset.

P/V contains contents of IFF2.

N is reset.

C is not affected.

If an interrupt occurs during execution of this instruction, the parity flag contains a 0.

LDIR

Operation

(DE) ← (HL), DE ← DE + 1, HL ← HL + 1, BC F↔ BC – 1

Op Code

LDIR

Operand

B8

1	1	1	0	1	1	0	1	ED
1	0	1	1	0	0	0	0	B0

Description

This 2-byte instruction transfers a byte of data from the memory location addressed by the contents of the HL register pair to the memory location addressed by the DE register pair. Both these register pairs are incremented and the Byte Counter (BC) Register pair is decremented. If decrementing allows the BC to go to 0, the instruction is terminated. If BC is not 0, the program counter is decremented by two and the instruction is repeated. Interrupts are recognized and two refresh cycles are executed after each data transfer. When the BC is set to 0 prior to instruction execution, the instruction loops through 64KB.

For BC ≠ 0:

M Cycles	T States	4 MHz E.T.
5	21 (4, 4, 3, 5, 5)	5.25

For BC = 0:

M Cycles	T States	4 MHz E.T.
4	16 (4, 4, 3, 5)	4.00

Condition Bits Affected

S is not affected.

Z is not affected.

H is reset.

4. `http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf`

OR s

Operation

A ← A ∨ s

Op Code

OR

Operand

s

The *s* operand is any of *r, n, (HL), (IX+d),* or *(IY+d)*, as defined for the analogous ADD instructions. These possible op code/operand combinations are assembled as follows in the object code:

OR r*	1	0	1	1	0	◄ r* ►		
OR n	1	1	1	1	0	1	1	0	F6
	◄ n ►							
OR (HL)	1	0	1	1	0	1	1	0	B6
OR (IX+d)	1	1	0	1	1	1	0	1	DD
	1	0	1	1	0	1	1	0	B6
	◄ d ►							
OR (IY+d)	1	1	1	1	1	1	0	1	FD
	1	0	1	1	0	1	1	0	B6
	◄ d ►							

r identifies registers B, C–, D, E, H, L, or A specified in the assembled object code field, as follows:

Register	r
B	000
C	001
D	010
E	011
H	100
L	101
A	111

Description

A logical OR operation is performed between the byte specified by the *s* operand and the byte contained in the Accumulator; the result is stored in the Accumulator.

Instruction	M Cycles	T States	4 MHz E.T.
OR r	1	4	1.00
OR n	2	7 (4, 3)	1.75
OR (HL)	2	7 (4, 3)	1.75
OR (IX+d)	5	19 (4, 4, 3, 5, 3)	4.75
OR (IY+d)	5	19 (4, 4, 3, 5, 3)	4.75

Condition Bits Affected

S is set if result is negative; otherwise, it is reset.

Z is set if result is 0; otherwise, it is reset.

H is reset.

P/V is set if overflow; otherwise, it is reset.

N is reset.

C is reset.

JP nn

Operation

PC ← nn

Op Code

JP

Operand

nn

	1	1	0	0	0	0	1	1	C3
	◄ n ►							
	◄ n ►							

Note: The first operand in this assembled object code is the low-order byte of a two-byte address.

Description

Operand *nn* is loaded to register pair Program Counter (PC). The next instruction is fetched from the location designated by the new contents of the PC.

M Cycles	T States	4 MHz E.T.
3	10 (4, 3, 3)	2.50

Condition Bits Affected

None.

RET cc

Operation

If cc true: PCL ← (sp), pCH ← (sp+1)

Op Code

RET

Operand

cc

1	1	◄ cc ►			0	0	0

Description

If condition *cc* is true, the byte at the memory location specified by the contents of the Stack Pointer (SP) Register pair is moved to the low-order eight bits of the Program Counter (PC). The SP is incremented and the byte at the memory location specified by the new contents of the SP are moved to the high-order eight bits of the PC. The SP is incremented again. The next op code following this instruction is fetched from the memory location specified by the PC. This instruction is normally used to return to the main line program at the completion of a routine entered by a CALL instruction. If condition *cc* is false, the PC is simply incremented as usual, and the program continues with the next sequential instruction. Condition *cc* is programmed as one of eight status that correspond to condition bits in the Flag Register (Register F). These eight status are defined in the following table, which also specifies the corresponding *cc* bit fields in the assembled object code.

cc	Condition	Relevant Flag
000	Non-Zero (NZ)	Z
001	Zero (Z)	Z
010	Non Carry (NC)	C
011	Carry (C)	C
100	Parity Odd (PO)	P/V
101	Parity Even (PE)	P/V
110	Sign Positive (P)	S
111	Sign Negative (M)	S

If *cc* is true, then the following data is returned:

M Cycles	T States	4 MHz E.T.
3	11 (5, 3, 3)	2.75

If *cc* is false, then the following data is returned:

M Cycles	T States	4 MHz E.T.
1	5	1.25

Condition Bits Affected

None.

Example

The S flag in the F Register is set, the Program Counter contains `3535h`, the Stack Pointer contains `2000h`, memory location `2000h` contains `B5h`, and memory location `2001h` contains `18h`. Upon the execution of a RET *M* instruction, the Stack Pointer contains `2002h` and the Program Counter contains `18B5h`, thereby pointing to the address of the next program op code to be fetched.

INC ss

Operation

ss ← ss + 1

Op Code

INC

Operand

ss

0	0	s	s	0	0	1	1

Description

The contents of register pair ss (any of register pairs BC, DE, HL, or SP) are incremented. In the assembled object code, operand *ss* is specified as follows:

Register Pair	ss
BC	00
DE	01
HL	10
SP	11

M Cycles	T States	4 MHz E.T.
1	6	1.50

Condition Bits Affected

None.

Example

If the register pair contains `1000h`, then upon the execution of an INC *HL* instruction, HL contains `1001h`.

INC (HL)

Operation

(HL) ← (HL) + 1

Op Code

INC

Operand

(HL)

0	0	1	1	0	1	0	0	34

Description

The byte contained in the address specified by the contents of the HL register pair is incremented.

M Cycles	T States	4 MHz E.T.
3	11 (4, 4, 3)	2.75

Condition Bits Affected

S is set if result is negative; otherwise, it is reset.

Z is set if result is 0; otherwise, it is reset.

H is set if carry from bit 3; otherwise, it is reset.

P/V is set if (HL) was `7Fh` before operation; otherwise, it is reset.

N is reset.

C is not affected.

Example

If the HL register pair contains `3434h` and address `3434h` contains `82h`, then upon the execution of an INC *(HL)* instruction, memory location `3434h` contains `83h`.

LD (DE), A

Operation

(DE) ← A

Op Code

LD

Operands

(DE), A

0	0	0	1	0	0	1	0	12

Description

The contents of the Accumulator are loaded to the memory location specified by the contents of the DE register pair.

M Cycles	T States	4 MHz E.T.
2	7 (4, 3)	1.75

Condition Bits Affected

None.

Example

If register pair DE contains `1128h` and the Accumulator contains byte `A0h`, then the execution of a LD *(DE), A* instruction results in `A0h` being stored in memory location `1128h`.

LD r, (HL)

Operation

r ← (HL)

Op Code

LD

Operands

r, (HL)

0	1	◄ r ►			1	1	0

Description

The 8-bit contents of memory location (HL) are loaded to register *r*, in which *r* identifies registers A, B, C, D, E, H, or L, assembled as follows in the object code:

Register	r
A	111
B	000
C	001
D	010
E	011
H	100
L	101

M Cycles	T States	4 MHz E.T.
2	7 (4, 3)	1.75

Condition Bits Affected

None.

Example

If register pair HL contains the number `75A1h`, and memory address `75A1h` contains byte `58h`, the execution of LD C, *(HL)* results in `58h` in Register C.

3 Aspects logiciels : algorithme de tracé de Bresenham

La fonction première d'une console de jeu est d'interagir avec l'utilisateur au travers de graphiques. Le motif le plus simple à afficher est une droite : il s'agit de l'élément de base de toute interface graphique *vectorielle*, bien plus élégante que l'approche *bitmap* vue auparavant, et initialement utilisée dans des jeux tels que Virus (publié en 1987) ou Star Wars (1983). Cependant, même une droite peut se tracer de façon optimisée : J.E. Bresenham a proposé en 1962 son algorithme désormais classique pour tracer une droite entre deux points (x_0, y_0) et (x_1, y_1). Son algorithme se généralise à toutes les courbes mais plus particulièrement aux coniques pour lesquelles la fonction de coût est "simple" à établir. Nous supposerons avoir à notre disposition la fonction `plot_point(x,y);` qui illumine un 8ixel aux coordonnées (x,y). Nous ne nous intéressons qu'au cas de l'octant pour lequel la **pente est positive et inférieure à 1**.

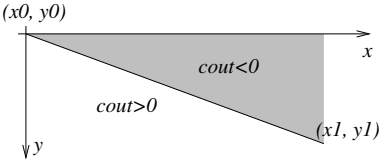


Figure 1: Fonction de coût

L'algorithme de Bresenham [5] se résume par la séquence suivante :

5. J.E. Bresenham, *Algorithm for computer control of a digital plotter* IBM Systems Journal **4** (1) (1965), pp.25-30.

— nous commençons par tracer le point de départ, (x_0, y_0)

— la question consiste à savoir quel point suivant afficher lorsque x est incrémenté, à savoir $(x+1, y)$ ou $(x+1, y+1)$. Par symétrie, nous ne considérons que l'octant dans lequel la pente de la droite est inférieure à 1 et positive,

— une fonction de coût est établie qui détermine lequel de $(x+1, y)$ ou $(x+1, y+1)$ est plus probablement sur la courbe à afficher,

— la fonction de coût est incrémentée pour chaque itération sur x jusqu'à atteindre x_1.

1. Connaissant (x_0, y_0) et (x_1, y_1) avec $x_1 > x_0$ et $y_1 > y_0$, rappeler l'équation de la droite passant par ces deux extrémités $y = f(x)$,
2. établir une fonction de coût $y - f(x)$ qui, si elle est positive, indique que (x, y) doit être allumé, sinon $(x, y+1)$ doit être allumé,
3. exprimer l'évolution de cette fonction de coût entre deux points adjacents, lorsque nous passons de (x, y) à $(x+1, y+1)$ ou $(x+1, y)$, selon le cas le plus favorable établi par la fonction de coût,
4. proposer une implémentation ne faisant intervenir que des **nombres entiers** dans la description de l'algorithme.
5. Sachant que la plus grande dimension de l'écran de la console de jeu est 160 pixels, quelle est la nature de la donnée stockant la variable de coût ?
6. Reprendre le raisonnement dans le cas d'un cercle de centre (x_0, y_0) et de rayon *rayon*. On ne voudra, pas symétrie, que calculer les valeurs comprises entre $x = x_0 + rayon$ (valeur initiale) et $x - x_0 < y - y_0$. On considère le premier octant où y croît de façon monotone et x décrémente de 0 ou 1.

L'application au tracé d'une ellipse est un peu plus calculatoire mais développée dans `https://web.archive.org/web/20120225095359/http://homepage.smc.edu/kennedy_john/belipse.pdf`.

4 Questions de cours

1. Exprimer 0xfc en décimal
2. Exprimer 0d254 en hexadécimal
3. Comment créer une variable de type "nombre entier sur 32 bits" en C ?
4. Quelle est la durée de communication de 3 caractères par une liaison asynchrone RS232 cadencée à 9600 bauds ? Comment cette durée se compare-t-elle avec une période d'échantillonnage typique de carte son ?
5. Comment passer à 0 le 6ème bit d'une variable `char c` ?
6. Comment allumer uniquement la LED connectée à la broche 2 du port B de l'Atmega32U4 ? (voir tableau ci-dessous)
7. Quelle est la taille, en bits, du produit de deux `short` ?
8. Quelle est la taille, en bits, de la somme de 1024 valeurs codées sur 6 bits ?
9. Comment afficher le contenu d'un répertoire dans un shell unix ?
10. Comment entrer dans le sous répertoire `windows` depuis le répertoire courant dans un shell unix ?

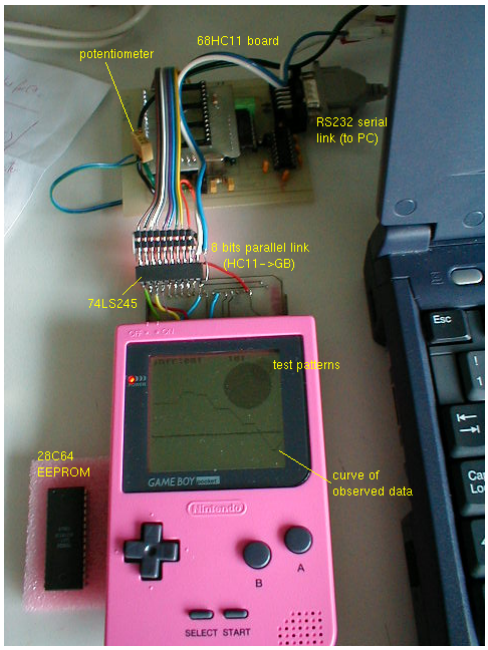


Figure 2: Interfaçage d'un 68HC11 comme périphérique de Gameboy, incluant ses convertisseurs analogique-numériques (`http://friedtj.free.fr/gb_eng.pdf`).

0x0C (0x2C)	PINE	-	PINE6	-	-	-	PINE2	-	-	
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	
0x08 (0x28)	PORTC	PORTC7	PORTC6	-	-	-	-	-	-	
0x07 (0x27)	DDRC	DDC7	DDC6	-	-	-	-	-	-	
0x06 (0x26)	PINC	PINC7	PINC6	-	-	-	-	-	-	
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	
0x02 (0x22)	Reserved	-	-	-	-	-	-	-	-	