Very High Speed Integrated Circuit Hardware Description Language/Embarquer un Atmel ATMega8

Ce projet consiste à développer un jeu de pong sur un écran V.G.A. dans un F.P.G.A.. C'est un sujet assez classique pour lequel plusieurs versions existent sur Internet. Ce qui fait son originalité est l'utilisation d'un cœur (libre) de micro contrôleur 8 bits en interaction avec de la logique externe réalisée en V.H.D.L.. Le processeur soft core sera appelé **CoreAtMega8** dans la suite de ce document. Il est compatible avec un AVR ATMEGA8® de chez Atmel. La programmation du cœur devra si possible, se faire en langage C.

N'oubliez pas...

Pour accompagner ce document, télécharger le fichier : ATMega8_pong_VGA.zip [1]. En effet, nous avons préparé ce fichier pour faciliter le démarrage de projets pour des étudiants. Les choses ne se passant pas toujours comme prévu aujourd'hui ce fichier dispose des répertoires suivants :

- "/AVRComplet16_S4_S4/25MHzAVR/" qui contient la version complète fonctionnant à 25MHz (à partir d'une horloge 50MHz), gérant correctement les interruptions avec une compilation utilisant le modèle ATMega8 qui permet de démarrer n'importe quel projet,
- "/CorrProjet2010/" qui contient la correction complète de ce chapitre mais avec une version antérieure du cœur.

Le plus formateur me semble cependant de commencer par télécharger le code de départ chez OpenCore [2] à partir de SVN (et non le fichier zippé) pour avoir la toute dernière version des fichiers.

Pour récupérer la dernière version sous Linux

Dans une console, lancer la commande :

```
 svn export http://opencores.org/ocsvn/cpu_lecture/cpu_lecture/trunk/
cpu_lecture
```

qui créera le répertoire cpu_lecture et y déposera la dernière version. Pour réaliser cela il faut s'inscrire chez opencores puisque l'on vous demandera un mot de passe.

La version OpenCore ne contient pas l'interface avec l'écran VGA.

Remarque

Le contenu de ce paragraphe peut sembler redondant avec les chapitres silicore1657 et CQPIC 16F84. Je rappelle que tous ces chapitres sont des projets qui font exactement la même chose, gérer un jeu de pong, d'où cette redondance. Il vous est demandé de choisir votre architecture et de lire le chapitre correspondant. Désolé, mais il en faut pour tous les goûts !

Choix du cœur

Il existe plusieurs versions de descriptions de cœurs compatibles avec les RISC de chez Atmel sur Internet. Nous avons tendance à choisir les cœurs de processeur en fonction de l'épaisseur de leur documentation. Cela avait déjà été le cas avec le cœur de processeur PIC16C57 l'année précédente (voir autre chapitre de ce cours, ou le rapport complet SiliCore1657.pdf [3]).

Même s'il existe d'autre cœurs concurrents chez Opencores AVR core [4] ou encore un autre Atmel AVR ATtiny261/461/861 [5], nous avons choisi celui-ci car il est accompagné d'un cours. Le **CoreAtMega8** est un cœur de processeur (ou processeur softcore) compatible avec le ATMEGA8® de chez Atmel. Il a été développé par Dr. Juergen Sauermann (Allemagne) et publié en janvier 2010 chez OpenCore sous forme d'un cours pour apprendre à développer un "soft processor" en VHDL (CPU lecture [2]).

Un cœur bien réalisé doit proposer tout sauf la ROM la RAM et les PORTs parce que leur implémentation est très dépendante du fabricant du FPGA ciblé. Il n'y a, en effet, aucune norme pour implanter ceux-ci.

Nous allons commencer par présenter la partie hardware du cœur en nous basant sur la documentation de Atmel.

Architecture d'ATMEGA8® de chez Atmel

Il s'agit d'un processeur 8 bits avec des instructions codées sur 16 bits ou 32 bits. Ce processeur gère un certain nombre d'interruptions.

Architecture des registres mémoires

Il existe 3 espaces distincts en RAM :

- Les 32 premières adresses correspondent aux 32 registres. Ces registres sont directement reliés à l'UAL et tous les calculs ne peuvent être effectués qu'à partir de ces registres (addition, soustraction, multiplication, opérations logiques, tests).
- L'espace IO de l'adresse 0x20 à 0x5F. Dans cet espace se trouvent beaucoup de registres de gestion du matériels (les ports, Timers, UART,...). Pour accéder à ces registres les instructions in et out utilisent l'adresse 0 pour le premier registre et non l'adresse 0x20. Les instructions cli et sbi permettent de mettre à 0 ou à 1 un bit d'un registre
- Enfin à partir de l'adresse 0x60 se trouve la mémoire à proprement parler (le ATMEGA8 possède 1ko de RAM).Pour accéder à tout l'espace mémoire (Registre et zone I/O incluse) les instructions ST,LD et STS et LDS peuvent être utilisées. L'adressage peut être direct (STS ou LDS) ou indirect (ST et LD). Un adressage indirect utilise les registres X, Y ou Z (respectivement les registres R27:R26, R29:R28 et R31:R30) pour pointer sur une zone mémoire.

Les deux premiers espaces sont désignés en anglais par "register file". On utilisera indistinctement : mémoire de registres, fichier de registres et banc de registres pour les désigner en français.

Quelques registres importants du banc de registres

Ce tableau (partiel) des registres respecte les fichiers d'inclusion du compilateur avr-gcc. Par exemple, dans la documentation officielle, le bit b0 du **PORTB** s'appelle PORTB0 tandis que dans le fichier d'inclusion du GNU-C il s'appelle PB0.

Même si une majorité de ces registres sont implantés dans notre architecture, ce n'est pas le cas pour tous :

- les registres gris clair sont implantés dans notre architecture initiale, celle que l'on télécharge chez OpenCores
- les registres un peu plus foncés seront implantés dans le projet PONG de ce chapitre
- les registres blancs ne seront pas implantés dans ce chapitre

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x3F(0x5F)	SREG	I	T	H	S	V	N	Z	C
0x32(0x52)	TCTN0	Timer0 8 bits							
0x2D(0x4D)	TCTN1H	Timer1 8 bits de poids fort							
0x2C(0x4C)	TCTN1L	Timer1 8 bits de poids faible							
0x23(0x43)	OCR2	Timer/Counter2 output compare register							
0x21(0x41)	WDTCR	-	-	-	WDCE	WDE	WDP2	WDP1	WDP0
0x20(0x40)	UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
0x1B(0x3B)	PORTA	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
0x1A(0x3A)	DDRA	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0

0x18(0x38)	PORTB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
0x17(0x37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x16(0x36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
0x15(0x35)	PORTC	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
0x14(0x37)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
0x13(0x33)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
0x12(0x32)	PORTD	PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
0x11(0x31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
0x10(0x30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
0x0C(0x2C)	UDR	Registre de données USART I/O							
0x0B(0x2B)	UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
0x0A(0x2A)	UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8

Les connaisseurs de l'ATMega8 sont invités à détailler ce tableau pour apprendre ce que l'on peut faire avec notre cœur.

Remarque

Il est facile de constater qu'aucun timer n'est implanté. Cela a comme conséquence qu'il nous sera impossible de faire tourner un petit système temps réel sans modifier ce cœur.

Les instructions

Cette section est **difficile**. Même si elle ne fait intervenir que des notions du niveau indiqué, il est conseillé d'avoir du recul sur les notions présentées pour bien comprendre ce qui suit. Cependant, ce contenu n'est pas fondamental et peut être sauté en première lecture.

Il n'est absolument pas nécessaire de connaître la liste des instructions pour continuer ce projet. Mais n'ayant trouvé cette liste nulle part dans Wikipédia, je la donne ici sans plus d'explications.

Instructions arithmétiques et logiques

Mnemonics	Operands	Description	Opération	Flags	#Clocks
ADD	Rd, Rr	Additionne deux registres	Rd ←Rd+Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Additionne deux registres avec la retenue	Rd← Rd+Rr+C	Z,C,N,V,H	1
ADIW	Rdl, K	Addition immédiate de mots	Rdh:Rdl←Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Soustraire deux registres	Rd←Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Soustraire les constantes de deux registres	Rd←Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Soustraire deux registres avec une retenue	Rd←Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Soustraire une constante du registre avec retenue	Rd←Rd - K - C	Z,C,N,V,H	1
SBIW	Rdl, K	Soustraction immédiate du mot	Rdh:Rdl←Rdh:Rdl - K	Z,C,N,V,S	2
AND	Rd, Rr	Registres du ET logique	Rd ← Rd * Rr	Z,N,V	1
ANDI	Rd, K	Registres du ET logique et constante	Rd ← Rd * K	Z,N,V	1
OR	Rd, Rr	OU logique entre registres	Rd ← Rd v K	Z,N,V	1
ORI	Rd, K	OU logique entre Registre et constante	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	OU exclusif entre registres	Rd ← Rd Rr	Z,N,V	1

COM	Rd	Complément à 1	Rd ← 0xFF - Rd	Z,C,N,V	1
NEG	Rd	Complément à 2	Rd ← 0x00 - Rd	Z,C,N,V,H	1
SBR	Rd, K	Mettre le(s) bit(s) dans un registre	Rd ← Rd v K	Z,N,V	1
CBR	Rd, K	Effacer le(s) bit(s) dans un registre	Rd ← Rd ET (0xFF-K)	Z,N,V	1
INC	Rd	Incremente	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decremente un registre	Rd ← Rd - 1	Z,N,V	1
TST	Rd	Test si zero ou négatif	Rd ← Rd * Rd	Z,N,V	1
CLR	Rd	Effacer le registre	Rd ← Rd XOR Rd	Z,N,V	1
SER	Rd	Registre tout à 1	Rd ← 0xFF	None	1
MUL	Rd, Rr	Multiplication (non signée)	R1:R0← Rd * Rr	Z,C	2
MULS	Rd, Rr	Multiplication (signée)	R1:R0← Rd * Rr	Z,C	2
MULSU	Rd, Rr	Multiplication signé par non signé	R1:R0← (Rd * Rr) <<1	Z,C	2
FMUL	Rd, Rr	Multiplication fractionnaire non signée	R1:R0 ← (Rd * Rr) << 1	Z,C	2
FMULS	Rd, Rr	Multiplication fractionnaire signée	R1:R0← (Rd * Rr) << 1	Z,C	2
FMULSU	Rd, Rr	Multiplication fractionnaire non signée et non signée	R1:R0← (Rd * Rr) << 1	Z,C	2

Instructions de sauts

Mnemonics	Operands	Description	Opération	Flags	#Clocks
RJMP	K	Saut relatif	PC ← PC + k + 1	None	2
IJMP		Saut indirect vers (Z)	PC ← Z	None	2
RCALL	K	Appel du sous-programme en relatif	PC ← PC + k + 1	None	3
ICALL		Appel indirect de (Z)	PC ← Z	None	3
RET		Retour de sous-programme	PC ← STACK	None	4
RETI		Retour d'interruption	PC ← STACK	I	4
CPSE	Rd,Rr	Compare et saute si égal	if(Rd=Rr) PC← PC + 2 or 3	None	
CP	Rd, Rr	Compare	Rd - Rr	Z,N,V,C,H	1
CPC	Rd, Rr	Compare avec la retenue	Rd – Rr - C	Z,N,V,C,H	1
CPI	Rd, K	Comparaison en immédiat	Rd - K	Z,N,V,C,H	1
SBRC	Rr, b	Saut si le bit du registre est effacé	if(Rr(b)=0) PC← PC + 2 or 3	None	1/2/3
SBRS	Rr, b	Saute si le bit du registre et positionné à 1	if(Rr(b)=1) PC← PC + 2 or 3	None	
SBIC	P, b	Saute si bit registre d'entrées/sorties est à 0	if(P(b)=0) PC← PC + 2 or 3	None	
SBIS	S, K	Saute si bit registre d'entrées/sorties est à 1	if(P(b)=1) PC← PC + 2 or 3	None	
BRBS	S, K	Brancher si le drapeau est mis	if(SREG(s)=1) then PC PC+K+1	None	
BRBC	S,K	Brancher si le drapeau est effacé	if(SREG(s)=0) then PC PC+K+1	None	
BREQ	k	Brancher si égalité	if(Z=1) then PC ← PC + k +1	None	
BRNE	k	Brancher si non égal	if(Z=0) then PC← PC + k +1	None	
BRCS	k	Brancher si la retenue est mise	if(C=1) then PC ← PC + k +1	None	
BRCC	k	Brancher si la retenue est effacée	if(C=0) then PC ← PC + k +1	None	
BRSH	k	Brancher si idem ou supérieur	if(C=0) then PC ← PC + k +1	None	
BRLO	k	Brancher si inférieur	if(C=1) then PC ← PC + k +1	None	

BRMI	k	Brancher si minimum	if(N=1) then PC ← PC + k +1	None
BRPL	k	Brancher si maxi	if(N=0) then PC ← PC + k +1	None
BRGE	k	Brancher si supérieur ou égale, signé	if(N V=0) then PC ← PC + k +1	None
BRLT	k	Brancher si inferieure a 0 (signé)	if(N V=1) then PC ← PC + k +1	None
BRHS	k	Brancher si toutes les retenues du drapeau sont mises	if(H=1) then PC ← PC + k +1	None
BRHC	k	Brancher si toutes les retenues du drapeau sont effacées	if(H=0) then PC ← PC + k +1	None
BRTS	k	Brancher si T flag est mis	if(T=1) then PC ← PC + k +1	None
BRTC	k	Brancher si T flag est	if(T=0) then PC ← PC + k +1	None
BRVS	k	Brancher si l'Overflow est à un	if(V=1) then PC ← PC + k +1	None
BRVC	k	Brancher si l'Overflow est effacé	if(T=0) then PC ← PC + k +1	None
BRIE	k	Brancher si l'interruption est permise	if(I=1) then PC ← PC + k +1	None
BRID	k	Brancher si l'interruption n'est pas permise	if(I=0) then PC ← PC + k +1	None

Instructions de transfert de données

Mnemonics	Operands	Description	Opération	Flags	#Clocks
MOV	Rd, Rr	Copier un registre dans un autre	Rd← Rr	None	1
MOVW	Rd, Rr	Copier un mot dans un autre	Rd+1:Rd← Rr+1:Rr	None	1
LDI	Rd, K	Charger en immédiat	Rd← Knone1		
LD	Rd, X	Chargeer en indirect	Rd← (x)	None	2
LD	Rd, X+	Chargement indirect et Post-inc.	Rd← (x), x← x+1	None	2
LD	Rd, - X	Chargement indirect et Pre-Dec.	x← x-1, Rd← (x)	None	2
LD	Rd, Y	Chargement indirect	Rd← (y)	None	2
LD	Rd, Y+	Chargement indirect et Post-inc	Rd← (y), y← y+1	None	2
LD	Rd, - Y	Chargement indirect et Pre-Dec.	y← y-1, Rd← (y)	None	2
LDD	Rd, Y+q	Chargement indirect avec déplacement	Rd← (y +q)	None	2
LD	Rd, Z	Chargement indirect	Rd← (z)	None	2
LD	Rd, Z+	Chargement indirect et Post-inc	Rd← (z), z← z+1	None	2
LD	Rd, -Z	Chargement indirect et Pre-Dec.	z← z-1, Rd← (z)	None	2
LDD	Rd, Z+q	Chargement indirect avec déplacement	Rd← (z +q)	None	2
LDS	Rd, K	Chargement direct avec SRAM	Rd← (k)	None	2
ST	X, Rr	Stockage indirect	(x) ← Rr	None	2
ST	X+, Rr	Stockage indirect et Post-inc	(x) ← Rr, x← x+1	None	2
ST	-X, Rr	Stockage indirect et Pre-Dec.	x← x-1, (x)← Rr	None	2
ST	Y, Rr	Stockage indirect	(y) ← Rr	None	2
ST	Y+, Rr	Stockage indirect et Post-inc.	(y) ← Rr, y← y+1	None	2
ST	-Y, Rr	Stockage indirect et Pre-Dec.	y← y-1, (y)← Rr	None	2
STD	Y+q, Rr	Stockage indirect avec déplacement	(y + q) ← Rr	None	2
ST	Z, Rr	Stockage indirect	(z) ← Rr	None	2
ST	Z+, Rr	Stockage indirect et Post-inc	(z) ← Rr, z← z+1	None	2
ST	-Z, Rr	Stockage indirect et Pre-dec.	z← z-1, (z)← Rr	None	2

STD	Z+q, Rr	Stockage indirect avec déplacement	(z + q) ← Rr	None	2
STS	K, Rr	Stockage direct de SRAM	(k) ← Rr	None	2
LPM		Chargement du programme de la mémoire	R0← (z)	None	3
LPM	Rd, Z	Chargement du programme de la mémoire	Rd← (z)	None	3
LPM	Rd, Z+	Chargement du programme de la mémoire et Post-inc	Rd← (z), z← z+1	None	3
SPM		Stockage du programme de la mémoire	(z)← R1:R0	None	-
IN	Rd, P	In port	Rd← PNone1	None	
OUT	P, Rr	OUT Port	P← Rr	None	1
PUSH	Rr	Pousse le registre dans la pile	STACK ← Rr	None	2
POP	Rd	Enlever le registre de la pile	Rd←STACK	None	2

Instructions sur bits et tests sur bit

Mnemonics	Operands	Description	Opération	Flags	#Clocks
SBI	P,b	Positionne un bit à 1	I/O(p,b)← 1	None	2
CBI	P,b	Positionne un bit à 0	I/O(p,b)← 0	None	2
LSL	Rd	décalage vers la gauche	Rd(n+1)←Rd(n),Rd(0)← 0	Z,C,N,V	1
LSR	Rd	décalage vers la droite	Rd(n)←Rd(n+1),Rd(7)← 0	Z,C,N,V	1
ROL	Rd	décalage circulaire gauche	Rd(0)←C,Rd(n+1)←Rd(n),C← Rd(7)	Z,C,N,V	1
ROR	Rd	décalage circulaire droite	Rd(7)←C,Rd(n)←Rd(n+1),C← Rd(0)	Z,C,N,V	1
ASR	Rd	décalage arithmétique droit	Rd(n)←Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	échange poids/fort/faible	Rd(3..0)←Rd(7..4),Rd(7..4),← Rd(3..0)	None	1
BSET	s		SREG(s)← 1	SREG(s)	1
BCLR	s		SREG(s)← 0	SREG(s)	1
BST	Rr, b		T← Rr(b)	T	1
BLD	Rd, b		Rd(b)←T	None	1
SEC		Mettre la retenue à 1	C←1	C	1
CLC		Mettre la retenue à 0	C←0	C	1
SEN			N←1	C	1
CLN			N←0	N	1
SEZ		mise à 1 de Z	Z←1	C	1
CLZ		mise à 0 de Z	Z←0	Z	1
SEI		Autorisation des interruptions globales	I←1	I	1
CLI		Désactivation des interruptions globales	I←0	I	1
SES		Positionnement à 1 le test de signe	S←1	S	1
CLS		Positionnement à 0 le test de signe	S←0	S	1
SEV		Positionne le dépassement en complément à deux	v←1	V	1
CLV		Annule le dépassement en complément à deux	V←0	V	1
SET		Set T in SREG	T←1	T	1

Le cœur CoreAtMega8

Nous allons commencer par aborder dans ce chapitre ce qu'il y a de spécifique pour notre système mono-puce, à savoir les PORTs, la RAM et la ROM. Avant de commencer quoi que ce soit vous devez donc vous poser trois questions :

- comment utiliser les ports existants et plus tard, comment en ajouter d'autres ?
- comment implanter la mémoire RAM ?
- comment générer de manière automatique la ROM ?

Nous ne prétendons pas répondre de manière exhaustive à ces trois questions simultanément, mais nous allons y revenir petit à petit.

Les ports du cœur CoreAtMega8

La gestion des PORTs dans ce cœur ressemble à cette même gestion dans le PicoBlaze. Il est facile d'étendre le nombre de PORTs jusqu'à 64 dans l'espace d'entrées/sorties. Nous nous contenterons cependant de l'existant, à savoir un PORT : **PORTB** dans le coeur initial comme cela est expliqué plus loin. Le **coreATMega8** est conçu pour une réalisation de PORTs bidirectionnels mais à l'extérieur : ces PORTs bidirectionnels ne sont pas réalisés dans le cœur car trop dépendants du fabricant du FPGA qui accueillera ce processeur softcore.

Remarque

Tous les ports sont accessibles de manière normale. L'écriture dans DDRA se fait par l'instruction :

```
// en langage C
DDRA=0xFF; // 1 <=> output
```

tandis qu'en assembleur on écrira :

```
; assembleur
ldi r10,0xFF
out DDRA,r10
```

Attention, comme on le verra plus loin, l'écriture dans DDRA n'a pas l'effet escompté ! Cela ne peut servir à choisir la direction du PORTA puisqu'il n'y a pas de PORT bidirectionnel d'implémenté.

Remarque

Ma deuxième remarque sera pour compléter notre description. On doit écrire dans un programme C :

```
  PORTA = PINA; // recopie du PORTA dans le PORTA
```

et non pas PORTA=PORTA.

La description des PORTs dans cette section n'est pas complète et sera reprise quand nous aurons besoin de les relier à une logique externe.

Abordons maintenant les deux autres périphériques nécessaires, la RAM et la ROM. Ces deux composants ne font pas partie du cœur tout simplement parce que les implantations de ceux-ci sont spécifiques aux FPGA cibles.

La RAM

On rappelle qu'il s'agit de l'espace à partir de l'adresse 0x60. C'est là que se trouve la mémoire à proprement parler.

Il existe peu de modèles de mémoire standard dans le monde des FPGA et ASIC. En fait le type le plus commun de mémoire que l'on peut trouver est ce que la norme WHISBONE appelle 'FASM', ou FPGA and ASIC Subset Model. Pour la RAM, sa spécificité est qu'il s'agit d'une mémoire à écriture et lecture synchrone. Le cœur CoreAtMega8 propose un exemple de mémoire en VHDL que nous utiliserons sans changement.

La mémoire est construite à partir de composants Xilinx "RAMB4_S4_S4" qui est une mémoire double ports. Cette façon de faire est peu portable et il vous faudra l'adapter si vous désirez faire tourner ce soft processor dans un FPGA concurrent.

La ROM

C'est là que se situe le programme à exécuter. Commençons à présenter un exemple de ROM.

```
-- VHDL File : prog_mem.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- the content of the program memory.
--
use work.prog_mem_content.all;

entity prog_mem is
    port (  I_CLK       : in  std_logic;

            I_WAIT      : in  std_logic;
            I_PC        : in  std_logic_vector(15 downto 0); -- word address
            I_PM_ADR    : in  std_logic_vector(11 downto 0); -- byte address

            Q_OPC       : out std_logic_vector(31 downto 0);
            Q_PC        : out std_logic_vector(15 downto 0);
            Q_PM_DOUT   : out std_logic_vector( 7 downto 0));
end prog_mem;

architecture Behavioral of prog_mem is

-- *************** retiré pour question de place ****************

end Behavioral;
```

Cet exemple incomplet ne montre aucun contenu. Chaque programme donnera une ROM différente. Il est à noter que ce que l'on présente est du VHDL alors que chaque compilateur ou assembleur ne délivre qu'un fichier au format HEX (Intel). Il faudra donc un utilitaire pour transformer le fichier HEX en fichier VHDL car il s'agit d'une opération qui peut se faire automatiquement. Nous utiliserons un programme en C++ que nous donnons en annexe 1 et qui s'appelle "make_mem.cc" qui est fourni avec le cœur. Compilez donc ce programme pour en faire un exécutable. Une fois l'exécutable réalisé, lancer

```
make_mem demo.hex prog_mem_content.vhd
```

ce qui vous génèrera un package dans un fichier prog_mem_content.vhd, qui sera utilisé dans le projet. La mémoire est construite à partir de composants Xilinx "RAMB4_S4_S4" qui est une mémoire double ports. Cette façon de faire est peu portable et il vous faudra l'adapter si vous désirez faire tourner ce soft processor dans un FPGA Altera ou ACTEL.

Mes premiers programmes en C

Nous ne présentons pas le langage C dans ce document. Si vous n'êtes pas familier, lisez :

- Le langage C chez Wikipédia
- La programmation C et ses exercices chez WikiBook
- Le langage C chez Wikiversité

La programmation du cœur se fait dans l'environnement AVRStudio. Le téléchargement d'AVRStudio nous permet d'utiliser l'assembleur ainsi que le compilateur C GNU (si l'on a installé winAVR). Puisque nous avons décidé pour ce projet de n'utiliser que des outils gratuits, cela nous convient parfaitement. Mon premier programme en C a été réalisé pour tester le fonctionnement du cœur CoreAtMega8 sur notre carte d'application Digilent.

Pour tester l'exécution d'un programme dans un coeur il faut naturellement que celui-ci fasse quelque chose de visible, autrement dit qu'il utilise les PORTs. C'est là que le travail sérieux commence. En effet le coeur a été développé et testé avec un spartan2 de chez Xilinx et nous désirons l'essayer sur une carte Digilent starter Board équipée d'un spartan3. Ceci nécessite donc quelques modifications de notre coeur sur au moins deux fichiers :

- le fichier de contraintes ucf qui relie les noms symboliques des entrées/sorties VHDL à des broches physiques du FPGA
- le fichier de description des PORTs que nous allons lire mais pas modifier pour le moment.

Connaître les PORTS implantés

Juergen Sauermann qui a développé le cœur utilisé dans ce chapitre (que nous appelons **CoreAtMega8**) le décrit comme compatible à un ATMega8. Cette compatibilité ne peut pas être complète :

- les fusibles ne sont pas implantés
- beaucoup de périphériques et leurs fonctionnalités ne sont pas implantées.

En général la compatibilité se fait plutôt au niveau des instructions : les seules instructions non implantées sont souvent celles qui mettent le micro-contrôleur en mode repos et éventuellement celles qui écrivent dans l'EEPROM.

Si l'on veut connaître lesPORTs il nous faut lire le fichier io.vhd du projet original dont on présente quelques extraits. Côté ports d'entrée on trouve :

```
-- IO read process
   --
   iord: process(I_ADR_IO, I_SWITCH,
                 U_RX_DATA, U_RX_READY, L_RX_INT_ENABLED,
                 U_TX_BUSY, L_TX_INT_ENABLED)
   begin
       -- addresses for mega8 device (use iom8.h or #define __AVR_ATmega8__).
       --
       case I_ADR_IO is
           when X"2A"  => Q_DOUT <=             -- UCSRB:
                          L_RX_INT_ENABLED  -- Rx complete int enabled.
                        & L_TX_INT_ENABLED  -- Tx complete int enabled.
```

```
                            & L_TX_INT_ENABLED  -- Tx empty int enabled.
                            & '1'               -- Rx enabled
                            & '1'               -- Tx enabled
                            & '0'               -- 8 bits/char
                            & '0'               -- Rx bit 8
                            & '0';              -- Tx bit 8
            when X"2B"  => Q_DOUT <=            -- UCSRA:
                              U_RX_READY       -- Rx complete
                            & not U_TX_BUSY    -- Tx complete
                            & not U_TX_BUSY    -- Tx ready
                            & '0'              -- frame error
                            & '0'              -- data overrun
                            & '0'              -- parity error
                            & '0'              -- double dpeed
                            & '0';             -- multiproc mode
            when X"2C"  => Q_DOUT <= U_RX_DATA; -- UDR
            when X"40"  => Q_DOUT <=            -- UCSRC
                              '1'              -- URSEL
                            & '0'              -- asynchronous
                            & "00"             -- no parity
                            & '1'              -- two stop bits
                            & "11"             -- 8 bits/char
                            & '0';             -- rising clock edge

            when X"36"  => Q_DOUT <= I_SWITCH;  -- PINB
            when others => Q_DOUT <= X"AA";
        end case;
    end process;
```

Ce qui nous intéresse est à la fin : la ligne when X"36" montre que le PORTB (plus exactement PINB) est implanté en lecture.

Cherchons la même chose en écriture :

```
-- IO write process
    --
    iowr: process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
            if (I_CLR = '1') then
                L_RX_INT_ENABLED  <= '0';
                L_TX_INT_ENABLED  <= '0';
            elsif (I_WE_IO = '1') then
                case I_ADR_IO is
                    when X"38"  => -- PORTB
                                   Q_7_SEGMENT <= I_DIN(6 downto 0);
                                   L_LEDS <= not L_LEDS;
                    when X"2A"  => -- UCSRB
                                   L_RX_INT_ENABLED <= I_DIN(7);
```

```
                                    L_TX_INT_ENABLED <= I_DIN(6);
                    when X"2B"  => -- UCSRA:       handled by uart
                    when X"2C"  => -- UDR:         handled by uart
                    when X"40"  => -- UCSRC/UBRRH: (ignored)
                    when others =>
                end case;
            end if;
        end if;
    end process;
```

Les nouvelles sont moins bonnes : le portB est bien implanté mais seulement sur 7 bits ! C'est une des choses que l'on changera dans la suite du projet, mais qu'on laisse tel quel pour le moment.

Notre architecture **CoreAtMega8** étant une architecture 8 bits avec peu de mémoire RAM, il peut sembler intéressant de commencer par présenter un programme en C mais qui ne contient que de l'assembleur.

Le premier programme simple en C avec assembleur

Le programme présenté montre comment inclure de l'assembleur dans un programme C.

```
#include <avr/io.h>
main(void)
{
asm volatile (
debut:      in      r24, 0x16      ; 22 : PINB
      com r24             ; one's complement
         out      0x18, r24      ; 24 : PORTB
         rjmp      debut           ;  infinite loop
)
}
```

Ne prenez pas ce programme à la lettre, il est juste donné comme exemple et n'a pas été testé.

Le premier programme simple en C pur

Un programme tout simple de test est présenté maintenant : il s'agit tout simplement de recopier le PORTB sur le PORTB. Remarquons aussi que ce programme fonctionne complètement si l'on a pris soin de prendre une version modifiée parce que la version originale sera incapable d'allumer la LED poids fort.

```
#include <avr/io.h>
main(void)
{ // La gestion de DDRB et DDRC est  inutile pour ce cœur
//  DDRB = 0xFF; // 8 sorties pour B retiré car non implanté (voir
ci-dessus)
//  DDRC = 0x00; // 8 entrees pour C retiré car non implanté
  while(1)
  PORTB = PINB; // recopie du PORTB dans le PORTB qui allume les LEDs

}
```

Ce programme est très simple et ne fait pas grand chose à part être pratique pour tester le fonctionnement correct.

Compilation

Tout programme C doit être compilé avec le compilateur GNU C facilement utilisable avec AVRStudio. Quelque soit l'environnement, quel que soit le langage, votre compilation donnera un fichier hex Intel. Ce fichier devra être transformé en fichier VHDL comme expliqué en Annexe 1. A ce point, seul un détail vous manque avant de compiler votre projet VHDL, la création du fichier de contraintes ucf.

Fichier ucf

Le fichier ucf doit être construit à partir de l'entité globale du projet que voici :

```
entity avr_fpga is
    port (  I_CLK_100   : in  std_logic;
            I_SWITCH    : in  std_logic_vector(9 downto 0);
            I_RX        : in  std_logic;

            Q_7_SEGMENT : out std_logic_vector(6 downto 0);
            Q_LEDS      : out std_logic_vector(3 downto 0);
            Q_TX        : out std_logic);
end avr_fpga;
```

Nous ne disposons que d'une horloge 50 MHz sur notre carte, mais nous garderons le nom "I_CLK_100" pour ne rien changer dans les fichiers VHDL.

```
NET     I_CLK_100       PERIOD = 20 ns;
NET     I_CLK_100       TNM_NET = I_CLK_100;
NET     "I_CLK_100"       LOC = "T9";
#liaison série
#net "serial_out" loc = "r13";
#net "serial_in" loc = "t13";
NET     I_RX            LOC = T13;
NET     Q_TX            LOC = R13;

# 7 LED display
#
NET     Q_7_SEGMENT<0>  LOC = "K12";
NET     Q_7_SEGMENT<1>  LOC = "P14";
NET     Q_7_SEGMENT<2>  LOC = "L12";
NET     Q_7_SEGMENT<3>  LOC = "N14";
NET     Q_7_SEGMENT<4>  LOC = "P13";
NET     Q_7_SEGMENT<5>  LOC = "N12";
NET     Q_7_SEGMENT<6>  LOC = "P12";

# single LEDs
#
#NET     Q_LEDS<0>       LOC = "K12";
#NET     Q_LEDS<1>       LOC = "P14";
#NET     Q_LEDS<2>       LOC = "L12";
#NET     Q_LEDS<3>       LOC = "N14";

# DIP switch(0 ... 7) and two pushbuttons (8, 9)
```

```
#
NET      I_SWITCH<0>      LOC = "f12";
NET      I_SWITCH<1>      LOC = "g12";
NET      I_SWITCH<2>      LOC = "h14";
NET      I_SWITCH<3>      LOC = "h13";
NET      I_SWITCH<4>      LOC = "j14";
NET      I_SWITCH<5>      LOC = "j13";
NET      I_SWITCH<6>      LOC = "k14";
NET      I_SWITCH<7>      LOC = "k13";


NET      I_Reset<0>      LOC = "L13";
NET      I_Reset<1>      LOC = "L14";
```

Vous avez certainement remarqué à ce stade que la liaison série est connectée. Nous l'utiliserons plus loin.

En résumé

Nous avons décrit comment modifier le cœur original pour l'adapter à une autre carte que celle pour lequel il a été conçu.

- le minimum à faire est de changer le fichier ucf
- ensuite écrire un fichier programme à compiler et à transformer en vhdl avec les outils appropriés.

Ce qui n'est présenté qu'en trame de fond ici, est la démarche à suivre pour interfacer ce cœur à de la logique externe :

- modifier le fichier io.vhd pour ajouter des registres internes au cœur et/ou des PORTs
- amener les entrées et sorties correspondantes au niveau de l'entité globale.

Nous allons voir dans la suite une série d'exemples qui fixera les idées sur ces points.

Un petit peu plus loin en C

Nous décidons de faire tourner le programme d'exemple donné avec le cœur. Ce programme utilise le seul afficheur sept segments de la carte spartan2 ainsi que la liaison série. Avec notre carte spartan3 nous disposons de 4 afficheurs sept segments et aussi de la liaison série. Une autre façon de dire les choses c'est qu'il nous faudra très légèrement changer la partie matérielle pour faire tourner le programme. Tout ce qui concerne la liaison série ne sera pas modifié (ce qui aura des conséquences sur la vitesse de transmission), seule la gestion de l'afficheur devra changer.

Programme de départ

Voici donc le programme de départ de Juergen Sauermann, donné dans le répertoire "app"

```
#include "stdint.h"
#include "avr/io.h"
#include "avr/pgmspace.h"
#undef F_CPU
#define F_CPU 25000000UL
#include "util/delay.h"

//-----------------------------------------------------------------------//
    //
```

```
//
   //   print char cc on UART.
                                                //
  //    return number of chars printed (i.e. 1).
                           //
 //

//
//------------------------------------------------------------------------//
uint8_t uart_putc(uint8_t cc) {
    while ((UCSRA & (1 << UDRE)) == 0)      ;
    UDR = cc;
    return 1;
}

//------------------------------------------------------------------------//
    //

//
   //   print char cc on 7 segment display.
                                  //
  //    return number of chars printed (i.e. 1).
                           //
 //

//
//------------------------------------------------------------------------//
// The segments of the display are encoded like this:
//
//      segment     PORT B
//      name        Bit number
//      ----A----   ----0----
//      |       |   |       |
//      F       B   5       1
//      |       |   |       |
//      ----G----   ----6----
//      |       |   |       |
//      E       C   4       2
//      |       |   |       |
//      ----D----   ----3----
//
//------------------------------------------------------------------------

#define SEG7(G, F, E, D, C, B, A)    (~(G<<6|F<<5|E<<4|D<<3|C<<2|B<<1|A))

uint8_t seg7_putc(uint8_t cc) {
    uint16_t t;
```

```
    switch(cc) {         //   G F E D C B A
    case ' ':   PORTB = SEG7(0,0,0,0,0,0,0);        break;
    case 'E':   PORTB = SEG7(1,1,1,1,0,0,1);        break;
    case 'H':   PORTB = SEG7(1,1,1,0,1,1,0);        break;
    case 'L':   PORTB = SEG7(0,1,1,1,0,0,0);        break;
    case 'O':   PORTB = SEG7(0,1,1,1,1,1,1);        break;
    default:    PORTB = SEG7(1,0,0,1,0,0,1);        break;
    }
    // wait 800 + 200 ms. This can be quite boring in simulations,
    // so we wait only if DIP switch 6 is closed.
    if (!(PINB & 0x20))     for (t = 0; t < 800; ++t)   _delay_ms(1);
    PORTB = SEG7(0,0,0,0,0,0,0);
    if (!(PINB & 0x20))     for (t = 0; t < 200; ++t)   _delay_ms(1);
    return 1;
}


//-------------------------------------------------------------------------//
    //

//
   //   print string s on UART.
                                              //
  //    return number of chars printed.
                                    //
 //

//
//-------------------------------------------------------------------------//
uint16_t uart_puts(const char * s) {
const char * from = s;
uint8_t cc;
    while ((cc = pgm_read_byte(s++)))   uart_putc(cc);
    return s - from - 1;
}


//-------------------------------------------------------------------------//
    //

//
   //   print string s on 7 segment display.
                                  //
  //    return number of chars printed.
                                    //
 //
```

```
//
//-------------------------------------------------------------------------//
uint16_t seg7_puts(const char * s) {
const char * from = s;
uint8_t cc;
    while ((cc = pgm_read_byte(s++)))   seg7_putc(cc);
    return s - from - 1;
}

//-------------------------------------------------------------------------
int main(int argc, char * argv[]) {
    for (;;) {
        if (PINB & 0x40)   {  // DIP switch 7 open.
                // print 'Hello world' on UART.
                uart_puts(PSTR("Hello, World!\r\n"));
        }
        else   {             // DIP switch 7 closed.
                // print 'HELLO' on 7-segment display
                seg7_puts(PSTR("HELLO "));
        }
    }
}
//-------------------------------------------------------------------------
```

Si l'on regarde le main, on voit que ce programme teste le bit b6 du PORTB : s'il est à un il envoie "Hello Word" sur la liaison série autrement il envoie "HELLO" sur l'afficheur sept segments.

Remarque

La directive #define F_CPU 25000000UL sert à définir la fréquence d'horloge. Elle est importante pour les sous-programmes _delay_ms. Elle est pourtant fausse pour notre carte. En effet ce cœur utilise un diviseur par 4 en entrée sur une fréquence originelle de 100MHz. Nous, nous devons nous contenter de 50MHz il faudrait donc diviser la fréquence par deux... Comme nous avons choisi l'option de modifier le moins de choses possible dans le **CoreATMega8**, nous laissons cela de côté. Par contre la liaison série fonctionne elle aussi deux fois moins vite : elle sera réglée à 19200 bauds 8 bits parité paire deux bits de stop. Tout ce qui vient d'être dit dans cette remarque concerne le cœur téléchargeable chez OpenCore. Si vous téléchargez le fichier ATMega8_pong_VGA.zip [1] et allez dans le répertoire "/AVRComplet16_S4_S4/25MHzAVR/" vous n'aurez plus l'affichage sept segments mais la liaison série fonctionne bien à 38400 bauds.

Nous allons redire les choses autrement de peur de perdre des lecteurs en route :

- le répertoire **/AVRComplet16_S4_S4/25MHzAVR/** du fichier ATMega8_pong_VGA.zip [1] fonctionne bien à 38400 bauds,
- le répertoire **/CorrProjet2010** contient la correction de ce chapitre branché sur 50MHz d'horloge mais avec une division par 4 qui réduit l'horloge effective à 12,5MHz et aussi la vitesse de la liaison série à 19200 bauds.

Modification matérielle en conséquence

Nos afficheurs sont multiplexés et il nous faut donc gérer cette situation. Voir TP 1 pour plus de détails sur les afficheurs.

Voici donc le nouveau fichier pour lequel nous avons décidé d'ajouter le reset et la gestion des afficheurs multiplexés pour nous adapter à notre carte spartan 3 :

A ce niveau tout fonctionne correctement sauf ....

Remarque
Le fonctionnement correct nécessite que l'interrupteur sw7 soit à un ! Nous n'avons pas vraiment d'idée sur la raison de cela pour le moment.. sauf peut être que ce n'est probablement pas une cause logicielle... mais plutôt matérielle. Est-ce lié au fait que la lecture du portB se fait sur 7 bits ?

Et si l'on voulait recevoir de l'information par la RS232...

Il n'est pas difficile de recevoir des données par la RS232 : il suffit d'attendre qu'un certain bit soit positionné puis de lire. Voici un programme complet qui fonctionne correctement mais avec l'affichage sur 7 segments si vous n'avez rien changé par rapport à la section précédente.

```
#include "avr/io.h"

#undef F_CPU
#define F_CPU 25000000UL

int main(int argc, char * argv[])
{
    for (;;) {
// 2 lignes ci-dessous pour recevoir
                while (!(UCSRA & (1<<RXC)));  //attente donnée RS232
                PORTB = UDR;
    }
}
```

Contrairement au Silicore1657 déjà évoqué, le cœur CoreAtMega8 dispose du mécanisme d'interruption que nous allons détailler maintenant.

Ajoutons une interruption

Nous avons appris à nos dépend que la gestion des interruptions nécessite d'utiliser un protocole précis si l'on ne veut pas se retrouver avec des programmes ne fonctionnant pas.

Gestion des Interuptions (Juergen Sauermann)

Suite à un problème avec les interruptions, nous avons été amené à échanger avec Juergen Sauermann, auteur de ce cœur **coreATMega8**. Il nous a très gentiment répondu le mail suivant qui est suffisamment pédagogique pour être traduit sans transformation.

Il y a des différences subtiles dans les entrées/sorties (E/S dans la suite) des différents composants ATMega. Quand je dis que le cœur est (principalement) compatible mega8 cela vaut pour le cœur mais pas pour les E/S (comme les registres pour l'UART les bits d'autorisation d'interruption etc.) Ainsi beaucoup de choses de io.h ne peuvent pas être utilisés et ces quelques différences sur les composants ont un impact sur mon coeur.

Laissez-moi décrire en premier comment les interruptions sont supposées fonctionner et vous donner quelques astuces pour réparer cela. Aussi loin que je me rappelle j'ai utilisé les interruptions pour UART Rx, UART Tx, et

timer dans quelques uns de mes projets.

Il y a des pré requis pour que les interruptions soient prises en charge:

1. les interruptions doivent être autorisées par l'intermédiaire du registre status (instruction EI), et
2. les interruptions individuelles doivent avoir été autorisées elles aussi (notant une différence entre le code donné dans le fichier html/pdf et le code VHDL dans le répertoire src, dans la suite je me réfèrerai uniquement au code VHDL qui est plus proche de l'ATMega)

Quand une interruption arrive alors L_INTVEC est positionné tant qu'aucune autre interruption est déjà en cours (io.vhd lignes 161-180).

Cela a pour conséquence le remplacement par opc_fetch.vhd de l'instruction courante par un pseudo opcode 00000000001vvvvv où vvvvv sont les 5 bits de poids faible de INTVEC.

Quand le pseudo-opcode est decodé il exécute une instruction jump vers le début du programme source. A ce point les différences entre les ATMegas entrent en jeu. Si vous générez du code pour de modèles mémoire "small" (jusqu'à, je crois, ATMega 8) alors avr_gcc génère une instruction (deux octets) RJMP par vecteur d'interruption. Pour des modèles plus grands, une instruction 2x2 octets JMP est générée.

Si vous regardez dans le fichier opc_deco.vhd autour de la ligne 99, alors vous voyez:

```
    Q_JADR <= "0000000000" & I_OPC(4 downto 0) & "0";
```

qui est pour un modèle de mémoire large (instruction 2 mots JMP). Si vous utilisez une instruction sur un mot RJMP, alors changez en :

```
    Q_JADR <= "00000000000" & I_OPC(4 downto 0);
```

Il faut faire aussi attention que avr_gcc peut ou pas changer les numéros de vecteurs d'interruption en fonction des composants.

Vous pouvez déclarer un service d'interruption de deux manières: par un numéro de vecteur ou par un nom. Si vous déclarez par nom (comme USART_RXC_vect) alors le compilateur pourra générer différents vecteurs pour différents composants. Si vous déclarez par des nombres (comme _VECTOR(13)) alors vous obtenez toujours le même vecteur quel que soit le composant spécifié comme cible. Déclarer par nom est certainement mieux pour des programmes qui seront portés dans plusieurs catégories de ATMegas mais en ce qui nous concerne déclarer avec des nombres est certainement plus approprié.

Ayant expliqué tout cela, la manière de procéder est ainsi:

1. vérifier que les interrutions sont autoridées globalement
2. vérifier que l'interruption en question est autorisée (attention aux incompatibilités déjà mentionnées plus haut)
3. compiler le code et générer un listing assembleur (avec avr-objdump "avr-objdump -d hello.out")
4. tester si les instructions JMP ou RJMP sont générées et changer le fichier opc_deco.vhd si nécessaire
5. tester si le numéro de vecteur en assembleur correspond au numéro de vecteur dans io.vhd.

That hopefully does it. Good luck,

Juergen Sauermann

Remarque
Pour votre information nous avons toujours compilé nos programmes en ciblant l'ATMega8 (voir début de l'Annexe III). Ainsi nous avons été obligé de faire le changement présenté par Juergen Sauermann dans le mail ci-dessus.

Mise en application

Nous présentons un petit programme d'interruption qui affiche ce qui arrive du PORT RS232 sur le PORTB

```
/*    Ce programme fonctionne avec les modifications présentées plus
haut */
#include "avr/io.h"
#include <avr/interrupt.h>

#undef F_CPU
#define F_CPU 25000000UL

// interruption de réception
ISR(USART_RXC_vect)  // ISR(_VECTOR(11))
{
      PORTB = UDR;
}

int main(int argc, char * argv[])
{
    UCSRB = (1<<RXEN)|(1<<RXCIE); // pour pouvoir déclencher interruption RS232
    sei(); // autorise interruption générale
    for (;;);
}
```

Le bit "RXCIE" qui autorise les interruptions est implanté dans notre cœur mais pas le bit qui autorise la réception. Ainsi il est en fait inutile d'écrire (1<<RXEN) dans le registre **UCSRB**.

L'autorisation des interruptions par sei() est implantée dans le **coreATMega8**.

Il est maintenant grand temps de revenir sur notre problème original, à savoir interfacer un écran VGA.

Assemblage du cœur, des mémoires et du module VGA

🔍 Avant d'aller plus loin, relisez **../Interfaces_VGA_et_PS2/**.

Arrivé à ce point, il nous faut certainement rappeler simplement ce que l'on cherche à faire.

Interface VGA et processeur
La gestion de l'écran VGA peut se présenter comme une boite, certes complexe, mais qui sait dessiner : • une balle • deux raquettes La seule chose qui sera demandé au processeur sera de gérer toutes les coordonnées de ces éléments.

Toute interface entre un processeur et de la logique externe passe par des ports.

À la fin d'un précédent chapitre Interfaces VGA, nous avons présenté un module capable de dessiner une balle et deux raquettes, ces trois objets étant mobiles : nous avions besoin de deux fois 10 bits pour piloter nos coordonnées de balles. Une autre façon de dire les choses est qu'il nous faut 4 ports de 8 bits en sortie pour notre cœur rien que pour gérer les positions X et Y d'une balle sur l'écran. Il faut naturellement ajouter un port par raquette.

Retour sur la gestion des PORTs dans le CoreAtMega8

La gestion des ports du CoreAtMege8 est relativement simple. Il suffit de modifier le fichier io.vhd du projet initial. Les PORTs existants (je veux dire ceux connus du compilateur C) sont en nombre suffisant pour ce projet. Les raquettes sont finalement gérées sur 8 bits seulement, ce qui fait en tout, seulement 6 PORTs à trouver.

Le fichier VHDL qui fait la gestion s'appelle io2.vhd et remplace le fichier io.vhd du projet initial. Nous en donnons le contenu maintenant. (On a laissé le composant qui gère la communication RS232 du cœur initial, on a ajouté le composant VGA_Top et modifié le "IO Write" process). Ce fichier un peu long est, par conséquent, mis dans une boîte déroulante :

La lecture de ce programme VHDL nous montre immédiatement les choix technologiques effectués pour la connexion des PORTs aux coordonnées des balles et raquettes.

choix technologiques

E/S module VGA	E/S AVR
x_rect<9:8>	PORTD<1:0>
x_rect<7:0>	DDRD<7:0>
y_rect<9:8>	PORTB<1:0>
y_rect<7:0>	DDRB<7:0>
y_raqG<7:0>	PORTC<7:0>
y_raqD<7:0>	DDRC<7:0>

Remarque

Il peut être choquant de voir utiliser les ports de direction **DDRB**, **DDRC** et **DDRD** comme ports de sortie. Il est impossible de faire cela avec un processeur réel. Pour les processeurs softcore, c'est parfois possible comme ici et pour le silicore1657 mais pas pour le CQPIC 16F84. Il n'y a donc aucune règle générale à ce sujet ! Seuls des tests permettent de trancher !

Nous sommes prêt pour la programmation en C de notre ensemble cœur plus gestion écran VGA.

Programmation du nouveau cœur avec écran VGA

Nous avons déjà présenté quelques programmes en C, mais nous allons examiner ce qu'il faut changer dans ces programmes pour gérer les nouveaux PORTs.

Programmation en C

Nous commençons par présenter un sous-programme qui écrit une valeur 16 bits dans deux des nouveaux PORTs pour changer la position en X de la balle. Le sous-programme "setX" Une version en C pur est montrée ci-dessous :

```
void setX(uint16_t x){
  DDRD=x; //poids faible
  PORTD=x>>8;//poids fort
}
```

Le contenu de ce programme est complètement déterminé par la partie matérielle.

Recopie des interrupteurs pour déplacer la balle

Nous allons présenter maintenant un programme fonctionnel qui déplace la balle dans une position déterminée par les interrupteurs.

```
#include <avr/io.h>
void setX(uint16_t x);
void setY(unsigned int x);
unsigned int posRaqu_16;

void main (void)
{
      unsigned int posX,posY;
      unsigned char raqD_y=0,raqG_y=0;
      signed char deltaX=1,deltaY=1;
      posX=113;
      posY=101;
      setX(posX);
      setY(posY);
            while(1){
      //setY directement
            DDRB=PINB;
            setX(PINB);
            }
}

void setX(uint16_t x){
  DDRD=x; //poids faible
  PORTD=x>>8;//poids fort
}

void setY(unsigned int y){
  DDRB=y; //poids faible
  PORTB=y>>8;//poids fort
}
```

Remarquez qu'au lieu d'utiliser le sous-programme setY on a réalisé l'affectation directe dans DDRB. Remarquez aussi les deux types distincts pour travailler sur 16 bits: uint16_t et unsigned int.

Programme complet de gestion simple de la balle

Nous présentons maintenant un programme simple de gestion de la balle avec des rebonds :

```
#include <avr/io.h>
#include <util/delay.h>
void setX(uint16_t x);
void setY(unsigned int x);
void main(){
int posX=0,posY=0;
signed char deltaX=1,deltaY=1;
```

```
      while(1){
            if ((posX>=620) && (deltaX>0)) deltaX= -deltaX;
            if ((posX<=40) && (deltaX<0)) deltaX= -deltaX;
            posX=posX+deltaX;
            setX(posX);
            if ((posY>=460) && (deltaY>0)) deltaY= -deltaY;
            if ((posY<=10) && (deltaY<0)) deltaY= -deltaY;
            posY=posY+deltaY;
            setY(posY);
            _delay_loop_2(30000);
      }
}
void setX(uint16_t x){
  DDRD=x; //poids faible
  PORTD=x>>8;//poids fort
}
void setY(unsigned int y){
  DDRB=y; //poids faible
  PORTB=y>>8;//poids fort
}
```

Ce programme réalise une balle rebondissant sur les bords. Le petit nombre des trajectoires gérées, peut devenir lassant pour un jeu. Mais à ce stade personne ne peut jouer car les raquettes sont invisibles.

Ajouter les bords, et rendre les raquettes mobiles

Nous avons déjà eu l'occasion de présenter les choix technologiques réalisés pour la connexion des deux raquettes. Le hardware doit permettre de voir les raquettes, ce sera le logiciel qui les fera bouger.

Solution simple sans bord

Nous allons présenter un ensemble fonctionnant mais avec des positions de raquette fixes. Ce sera aux étudiants de les faire bouger. Pour pouvoir gérer des rectangles de tailles différentes et de couleur différentes, on complique un peu la partie destinée à dessiner un rectangle en lui donnant une couleur de rectangle une largeur et une hauteur. Voici donc notre nouveau composant :

```
--VHDL
COMPONENT rect IS PORT(
  row,col,x_rec,y_rec,delta_x,delta_y :in STD_LOGIC_VECTOR(9 DOWNTO 0);
  colorRGB : in  STD_LOGIC_VECTOR(2 DOWNTO 0);
  red1,green1,blue1 : out std_logic);
END component;
```

L'instanciation des rectangles pour la balle et les raquettes se fera alors de la manière suivante :

```
-- VHDL
balle:rect port map(row=>srow, col=>scol,r ed1=>sred,
green1=>sgreen, blue1=>sblue,   colorRGB=>"111",
delta_x=>"0000001010", delta_y=>"0000001100",
                        x_rec => x_rect, y_rec => y_rect);
```

```
raquetteG:rect port map(row=>srow, col=>scol, red1=>sred1,
green1=>sgreen1,
         blue1=>sblue1, colorRGB=>"100",
delta_x=>"0000001010",
         delta_y=>"0000111010",       x_rec => "0000010110",
         y_rec(8 downto 1) => y_raquG,
y_rec(9)=>'0',y_rec(0)=>'0');
raquetteD:rect port map(row=>srow, col=>scol, red1=>sred2,
green1=>sgreen2,
          blue1=>sblue2,colorRGB=>"100",
delta_x=>"0000001010",
          delta_y=>"0000111010", x_rec => "1001001000",
          y_rec(8 downto 1) =>
y_raquD,y_rec(9)=>'0',y_rec(0)=>'0');

  red <= sred or sred1 or sred2;
  green <= sgreen or sgreen1 or sgreen2;
  blue <= sblue or sblue1 or sblue2;
```

Les déclarations des signaux dans ce morceau de programme sont omises.

Voici maintenant le programme C permettant le rebond sur les raquettes. Rappelons que les raquettes sont fixes, il vous faudra modifier ce programme pour les faire bouger. Le matériel est lui prévu pour les faire bouger comme on l'a vu dans la section "La gestion des PORTs dans le CoreAtMega8".

```
#include <avr/io.h>
#include "util/delay.h"
/* Port A */
//#define PINA      _SFR_IO8(0x19)
//#define DDRA      _SFR_IO8(0x1A)
//#define PORTA      _SFR_IO8(0x1B)
void setX(uint16_t x);
void setY(unsigned int x);
//void wait(unsigned char tempo);
//void wait(int tempo);
unsigned int posRaqu_16;

void main (void)
{
      unsigned int posX,posY;
      unsigned char raqD_y=0,raqG_y=0;
      signed char deltaX=1,deltaY=1;
      while(1) {
        posX=130;
        posY=301;
        setX(posX);
        setY(posY);
        while( (posX>30) && (posX<580)){
            posRaqu_16=raqD_y<<1;
```

```
            if ((posX>=574) && (posY<posRaqu_16+58) &&
                    (posY+10>posRaqu_16) && (deltaX>0)) deltaX= -deltaX;
            posRaqu_16=raqG_y<<1;
            if ((posX<=32) && (posY<posRaqu_16+58) &&
                     (posY+10>posRaqu_16) && (deltaX<0)) deltaX= -deltaX;
            posX=posX+deltaX;
            setX(posX);
            if ((posY>=460) && (deltaY>0)) deltaY= -deltaY;
            if ((posY<=10) && (deltaY<0)) deltaY= -deltaY;
            posY=posY+deltaY;
            setY(posY);
// manque la gestion des raquettes ici
            _delay_loop_2(30000);
      }
    }
}

void setX(uint16_t x){
  DDRD=x; //poids faible
  PORTD=x>>8;//poids fort
}

void setY(unsigned int x){
  DDRB=x; //poids faible
  PORTB=x>>8;//poids fort
}
```

Comme on peut le remarquer en lisant les commentaires de ce programme, la gestion des raquettes avec leurs déplacement n'est pas réalisée dans ce programme. Ce travail est présenté un peu plus loin avec sa correction.

Travail à réaliser

Remarque
Le contenu de cette section est lié au fait qu'un des auteurs a utilisé cette page comme énoncé de projet avec ses étudiants (année scolaire 2010/2011). Le fait qu'on y trouve les solutions est lié aussi à l'avancement du projet qui est terminé maintenant. Si vous utilisez cette page aussi avec vos étudiants, il vous faudra certainement adapter complètement cette section "Travail à réaliser".

Comprendre la partie matérielle (projet tutoré)

Vous devez être capable de dessiner les composants et leurs liaisons à partir des fichiers VHDL donnés dans le projet initial.

Vous allez traduire en français le chapitre 8 sur les Entrées/Sorties du cours (lecture.pdf) et me refaire le tableau des instructions.

Pour information le fichier lecture.pdf est un cours en anglais, fourni avec le projet original.

Développer en C

Étendre la partie logicielle pour gérer le déplacement des deux raquettes. Tests de bits des PORTs (pour le déplacement des raquettes) Chaque port est défini dans le fichier "avr/io.h". Par exemple PORTB est défini comme #define PORTB _SFR_IO8(0x18) car il est en position 0x18=24 dans l'espace d'entrées/sorties (soit en adresse 0x38). Chacun des bits des PORTs est aussi prédéfini :

```
//********** langage C ********
/* PORTB */
#define PB7      7
#define PB6      6
#define PB5      5
#define PB4      4
#define PB3      3
#define PB2      2
#define PB1      1
#define PB0      0

/* DDRB */
#define DDB7      7
#define DDB6      6
#define DDB5      5
#define DDB4      4
#define DDB3      3
#define DDB2      2
#define DDB1      1
#define DDB0      0

/* PINB */
#define PINB7      7
#define PINB6      6
#define PINB5      5
#define PINB4      4
#define PINB3      3
#define PINB2      2
#define PINB1      1
#define PINB0      0
```

Muni de ces informations, vous pouvez comprendre que pour tester que le bit PINB1 du port PINB est à un il suffit d'écrire en C :

```
//********** langage C ********
/* Port B */
if ((PINB &(1<<PINB1))==(1<<PINB1))
/* ou mieux encore */
if (bit_is_set(PINB, PINB1))
```

Nous avons choisi le PINB connecté aux interrupteurs sw0,sw1,sw6 et sw7 de la carte, pour faire descendre et monter les raquettes mais si vous disposez de joysticks, tout autre choix peut être fait. Ajouter un PORT dans notre

cœur (inutile si raquettes sur 8 bits) Par défaut le PORTA n'existe pas dans l'ATMega8. Mais pour nous ce n'est pas un problème pour l'ajouter matériellement (dans le fichier io2.vhd) et logiciellement. Il nous suffit d'ajouter :

```
//********** langage C ********
/* Port A */
#define PINA      _SFR_IO8(0x19)
#define DDRA      _SFR_IO8(0x1A)
#define PORTA      _SFR_IO8(0x1B)
```

dans nos programmes en C. Notre fichier "io2.vhd" (voir la section La gestion des PORTs dans le CoreAtMega8) doit être modifié en conséquence (en utilisant les adresses ci-dessus auxquelles on ajoute 0x20 :

```
--************ extraits de io2.vhd ************
case I_ADR_IO is
      when X"3A"  => Balle_xLow <= I_DIN;  --DDRA
      when X"3B"  => Balle_xHigh <= I_DIN;  --PORTA
      when X"31"  => Balle_xLow <= I_DIN;  --DDRD
      when X"32"  => Balle_xHigh <= I_DIN;  --PORTD
      when X"37"  => Balle_yLow <= I_DIN;  --DDRB
      when X"38"  => Balle_yHigh <= I_DIN;  --PORTB
      when X"34"  => raqD_y <= I_DIN;  --DDRC
      when X"35"  => raqG_y <= I_DIN;  --PORTC
```

Tracé de Bresenham

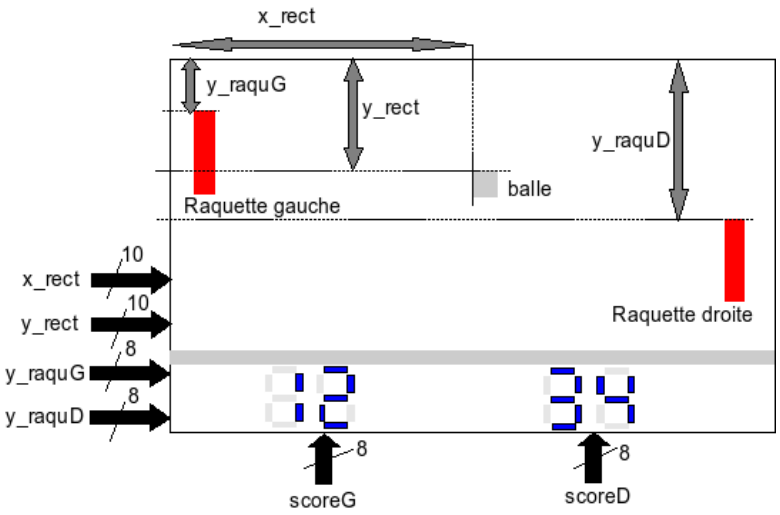
Explorer s'il n'est pas possible d'utiliser l'algorithme de tracé de segment de droite de Bresenham pour les trajectoires de balles. Cet algorithme est expliqué dans le WIKI : Algorithme de tracé de segment de Bresenham Il est possible de trouver directement une version en C : tapez Bresenham en C dans google.

Gestion des scores

Gérer un affichage des scores dans le moniteur RS232. Chaque fois qu'un nouveau score est réalisé, il est envoyé sur la liaison série qui est affichée à l'aide d'un hyperterminal. C'est pour répondre à cette question que l'on a laissé la partie concernant la RS232 dans le cœur initial.

Correction complète

La correction complète est dans le fichier ATMega8_pong_VGA.zip [1] et se trouve dans le répertoire "CorrProjet2010". Dans cette correction, nous n'avons pas respecté le cahier des charges pour la gestion des scores : ils sont affichés directement sur l'écran VGA et non envoyés par la liaison série, comme le montre la figure. Bien sûr les entrées présentées dans la figure sont reliées à des PORTs du processeur embarqué que l'on a pris soin de relier comme suit :



Gestion du jeu de Pong sur un écran VGA

choix technologiques

E/S module VGA	E/S AVR
x_rect<9:8>	PORTD<1:0>
x_rect<7:0>	DDRD<7:0>
y_rect<9:8>	PORTB<1:0>
y_rect<7:0>	DDRB<7:0>
y_raqG<7:0>	PORTC<7:0>
y_raqD<7:0>	DDRC<7:0>
scoreD<7:0>	PORTA<7:0>
scoreG<7:0>	DDRA<7:0>

Ainsi l'affichage d'une valeur sur le score droit se fait simplement par PORTA = valeur; en prenant bien soin d'y mettre une valeur BCD.

Une version légèrement différente est donnée sous forme de figure dans Autres projets pour ATMEL ATMega8 si vous avez besoin d'explications supplémentaires.

Recherche de bogue dans ce cœur

Quand ce projet a débuté, nous avons travaillé avec des anciens fichiers montrant un bogue. En fait les dernières versions et la version que nous proposons à télécharger (ATMega8_pong_VGA.zip [1]) sont corrigées. Ne sachant pas qu'une correction avait été faite, nous nous sommes mis en quête de corriger le bogue et il nous semble utile d'en raconter l'histoire.

Nous partons de la situation suivante : nous connaissons mal l'assembleur de l'AVR. Nous allons donc travailler en C et montrer qu'il est plus difficile traquer un bogue en langage C qu'en assembleur.

Erreur complètement incompréhensible en C

Voici présenté succinctement ce qui nous arrive. Le morceau de programme ci-dessous fonctionne correctement.

```
//******** petit bout de code qui fonctionne ***************
if ((posX>=574) && (posY<posRaqu_16+58) && (posY+10>posRaqu_16) && (deltaX>0)){
      deltaX= -deltaX; // rebond sur raquette droite
}
```

On en déduit qu'il est capable de faire les tests du if correctement. Mais dès que l'on ajoute quelque chose dans ce if, après le deltaX=-deltaX, comme par exemple :

```
//******** petit bout de code qui ne fonctionne pas ***************
if ((posX>=574) && (posY<posRaqu_16+58) && (posY+10>posRaqu_16) && (deltaX>0)){
      deltaX= -deltaX; // rebond sur raquette droite
      if (delta_raqD_y) { // delta_raqD_y!=0 on change la pente
            if ((delta_raqD_y >0 && deltaY >0) ||(delta_raqD_y <0 && deltaY <0)) {
                  dy+=5; if (dy > 15) dy=15; // ne peut dépasser 15
            } else {
                  dy-=5; if (dy < 5) dy=5; // ne peut dépasser 5
            }
      }
}
```

eh bien cela ne fonctionne plus correctement : en fait l'instruction deltaX = - deltaX n'est jamais réalisée !!!

Si l'on reste à ce niveau, il n'y a aucune chance de trouver pourquoi. Comment ajouter des instructions peut-il perturber le test puisqu'il est déjà fait là où l'on ajoute ces instructions ?

On descend donc au niveau de l'assembleur pour voir ce qui s'y passe.

Même erreur en assembleur

Il est possible de visualiser le programme assembleur généré. C'est un fichier qui a l'extension lss et voici les deux extraits incriminés. (La commande qui permet d'avoir le fichier lss est : avr-objdump -h -S hello.out > hello.lss)

Commençons par celui qui fonctionne :

```
           if ((posX>=574) && (posY<posRaqu_16+58) && (posY+10>posRaqu_16) && (deltaX>0)){
  96:      82 e0             ldi      r24, 0x02      ; 2
  98:      ee 33             cpi      r30, 0x3E      ; 62
  9a:      f8 07             cpc      r31, r24
  9c:      68 f0             brcs      .+26          ; 0xb8 <main+0x70>
  9e:      c9 01             movw      r24, r18
  a0:      ca 96             adiw      r24, 0x3a      ; 58
  a2:      48 17             cp      r20, r24
  a4:      59 07             cpc      r21, r25
  a6:      40 f4             brcc      .+16          ; 0xb8 <main+0x70>
  a8:      ca 01             movw      r24, r20
  aa:      0a 96             adiw      r24, 0x0a      ; 10
  ac:      28 17             cp      r18, r24
  ae:      39 07             cpc      r19, r25
  b0:      18 f4             brcc      .+6           ; 0xb8 <main+0x70>
  b2:      17 fd             sbrc      r17, 7
```

```
  b4:      11 95             neg      r17
  b6:      11 95             neg      r17
```

Puis maintenant celui qui ne fonctionne pas :

```
           if ((posX>=574) && (posY<posRaqu_16+58) && (posY+10>posRaqu_16) && (deltaX>0)){
  9c:      82 e0             ldi      r24, 0x02      ; 2
  9e:      ee 33             cpi      r30, 0x3E      ; 62
  a0:      f8 07             cpc      r31, r24
  a2:      28 f1             brcs      .+74           ; 0xee <main+0xa6>
  a4:      c9 01             movw      r24, r18
  a6:      ca 96             adiw      r24, 0x3a      ; 58
  a8:      68 17             cp      r22, r24
  aa:      79 07             cpc      r23, r25
  ac:      00 f5             brcc      .+64           ; 0xee <main+0xa6>
  ae:      cb 01             movw      r24, r22
  b0:      0a 96             adiw      r24, 0x0a      ; 10
  b2:      28 17             cp      r18, r24
  b4:      39 07             cpc      r19, r25
  b6:      d8 f4             brcc      .+54           ; 0xee <main+0xa6>
  b8:      11 16             cp      r1, r17
  ba:      cc f4             brge      .+50           ; 0xee <main+0xa6>
                 deltaX= -deltaX; // rebond sur raquette droite
  bc:      11 95             neg      r17
```

Comme on peut le voir le fait d'ajouter des instructions dans le if du test a changé sa façon de faire le test. Pour les développeurs de compilateurs cela doit paraître évident mais pour les simples utilisateurs que nous sommes, c'est difficile à imaginer.

Si dans le deuxième cas on sait que l'on n'a jamais delaX=-deltaX de réalisé, on en déduit **que c'est l'instruction brge qui ne fonctionne pas correctement**. Après 24 heures de réflexion nous n'étions plus aussi catégorique mais il fallait plutôt dire : **c'est l'instruction brge qui ne fonctionne pas correctement quand elle est précédée de cp ou cpc** et donc qu'en final c'est cp et cpc qui ne positionnent pas correctement le bit utilisé par brge, à savoir le drapeau S.

Nous prévoyons dans un futur proche (d'ici fin 2011) de tester l'ensemble des instructions de test, mais en assembleur cette fois-ci... mais il y a tellement de choses à faire... d'ici la fin 2011...

Remarque
Le bogue ci-dessus a été en fait corrigé par l'auteur original du cœur (Juergen Sauermann). Si vous allez chez Opencores [2] préférez le téléchargement par SVN plutôt que le fichier zip, ce premier étant plus à jour.

Conclusion

Cette mésaventure de bogue nous montre clairement l'intérêt de disposer d'un cœur libre : on peut toujours lire et modifier son code source. Certains nous rétorqueront que lorsqu'on achète, on a automatiquement des processeurs softcore sans bogue. Nous laissons au lecteur le soin de se faire sa propre opinion à partir de ses expériences passées dans le domaine du logiciel...

Rappel sur les versions de l'AVR

Le projet décrit dans ce chapitre se trouve dans le répertoire **/CorrProjet2010** de ATMega8_pong_VGA.zip [1]. C'est une version sans bogue dans le processeur mais n'utilise pas la dernière version du processeur comme l'indique la remarque suivante.

Remarque
L'écriture du chapitre Programmer in Situ et déboguer nous a obligé à faire évoluer le cœur embarqué **coreATMega8**. Même si la gestion d'interruptions, l'implantation de l'instruction SPM ne sont pas nécessaires pour le projet étudiant de PONG de cette section, le simple fait de vouloir utiliser data2mem (décrit lui aussi dans Programmer in Situ et déboguer) nous oblige à utiliser la dernière version de notre processeur. Cette version n'est pas disponible chez Opencores pour le moment mais nous la mettons à disposition dans le répertoire **/AVRComplet16_S4_S4/25MHzAVR/** de notre ATMega8_pong_VGA.zip [1]. C'est la version la plus à jour pour le moment, version pour laquelle nous avons vérifié les interruptions RS232 et l'instruction SPM.

Pour dire les choses autrement, vous ne pourrez pas utiliser data2mem avec la correction de ce chapitre (dans **/CorrProjet2010** de ATMega8_pong_VGA.zip [1]). Peut être qu'un jour viendra où tout sera parfait.... En tout cas la correction du projet 2011 sera complète de ce point de vue.

Un autre projet pour l'année suivante

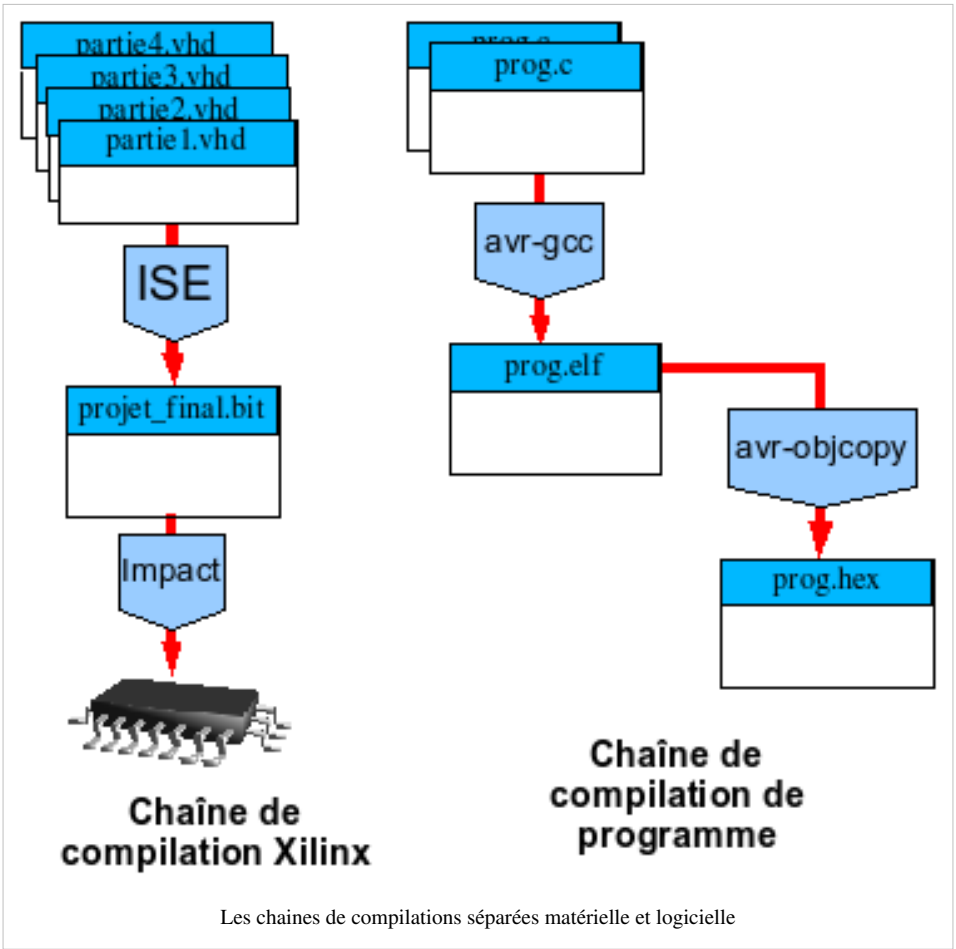
Pour ne pas trop alourdir ce chapitre, le contenu de cette section a été déplacé en Autres projets pour Atmel ATMega8.

Annexe I (transformer un fichier HEX en VHDL)

Présentation de la chaîne de compilation

La compilation d'un cœur embarqué proposé par un fondeur de FPGA nécessite l'EDK et le SDK chez Xilinx ainsi que l'ISE. L'EDK n'est pas gratuit mais l'ISE l'est à condition de s'inscrire sur le site de Xilinx. Il faut alors utiliser une licence de type webpack (gratuite). L'utilisation d'un processeur softcore libre représente l'énorme avantage de n'utiliser que l'ISE pour sa synthèse.

Nous présentons la chaîne de compilation pour savoir à quel gendre de problème nous devrons nous attaquer pour enfin exécuter un programme compilé par un compilateur du GNU dans notre FPGA.

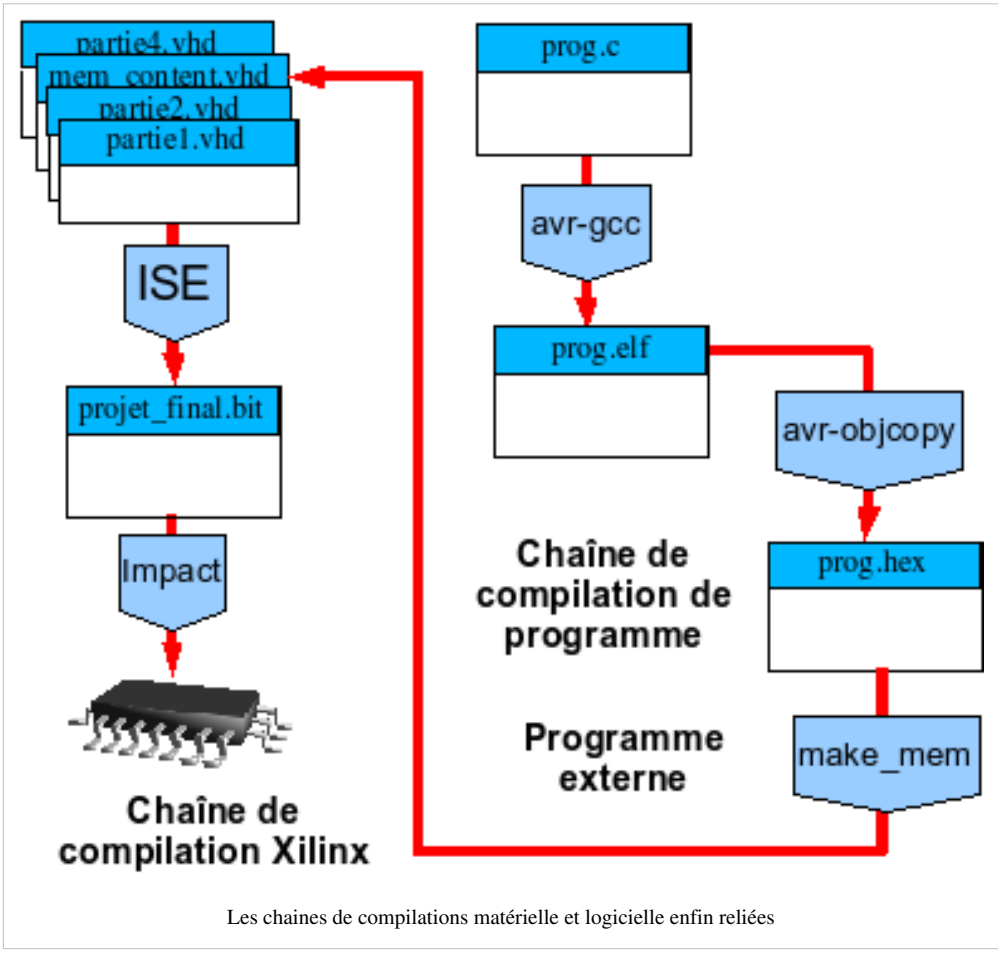


Les chaines de compilations séparées matérielle et logicielle

A gauche nous avons présenté sans détail la chaîne de compilation Xilinx qui part de fichiers VHDL pour réaliser un fichier BIT avec l'ISE. Ce fichier sera téléchargé par Impact (ou adept ou autre) dans le composant. A droite nous avons la chaine de compilation de notre (ou nos) programme(s) C. L'obtention d'un fichier HEX n'est pas obligatoire, un fichier ELF est parfois suffisant.

Utilisation de make_mem

Si l'on veut que l'ISE nous génère un processeur susceptible d'exécuter un programme, il nous faut trouver un moyen de relier ces deux chaines de compilation. J'espère que vous aviez remarqué qu'elles sont séparées, aucune interaction entre les deux ! Il nous faudra en effet d'une manière ou d'une autre mettre le programme compilé dans le FPGA. Il y a plusieurs moyens de faire cela que nous allons présenter encore à l'aide de figures.



Les chaines de compilations matérielle et logicielle enfin reliées

La première façon est présentée ci-dessus. On a ajouté un programme externe "make_mem" capable de transformer un fichier HEX en fichier VHDL qui sera inséré dans le projet (pour l'exemple avec le fichier mem_content.vhd). Cette façon de faire est à mon avis la plus simple et tout projet devrait la proposer au moins pour les débutants. Mais elle est cependant loin d'être un must. En effet tout changement dans le programme C nécessite de réaliser la compilation des programmes C ce qui prend quelques secondes puis de transformer le fichier HEX en fichier VHD ce qui ne prend aussi que quelques secondes et enfin de compiler le projet VHDL en entier ce qui peut prendre un certain temps (10 - 15 mn sur un vieux PC). Lancer ISE pour compiler un projet important est toujours consommateur de temps. Il nous faut donc explorer d'autres solutions : ce sera fait après la section suivante.

Source de make_mem

Le programme C++ pour transformer le fichier HEX en fichier VHDL est présenté ci-dessous.

Une fois compilé mon utilitaire s'appelle make_mem.exe (sous windows) ou make_mem sous Linux.

L'utilisation est faite par la ligne de commande :

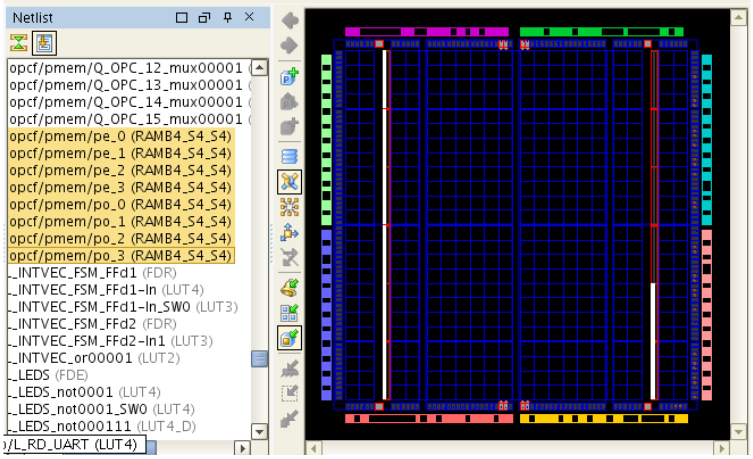
```
make_mem demo1.hex prog_mem_content.vhd
```

si le fichier à convertir s'appelle demo1.hex. Le fichier généré s'appelle prog_mem_content.vhd et se trouve sous la forme d'un package.

Ce programme (une fois compilé) génèrera une description en BRAM (Block RAM) du programme (à exécuter par l'AVR), comme le montre la figure ci-dessous :

Conventions pour lire la figure :

- les 12 rectangles rouges en deux rangées non contigües représentent la BRAM du FPGA
- les parties en blancs (sous parties de certains rectangles rouges) représentent l'ensemble des blocs mémoires utilisés pour le programme.



Les blocs mémoires utilisés dans le FPGA

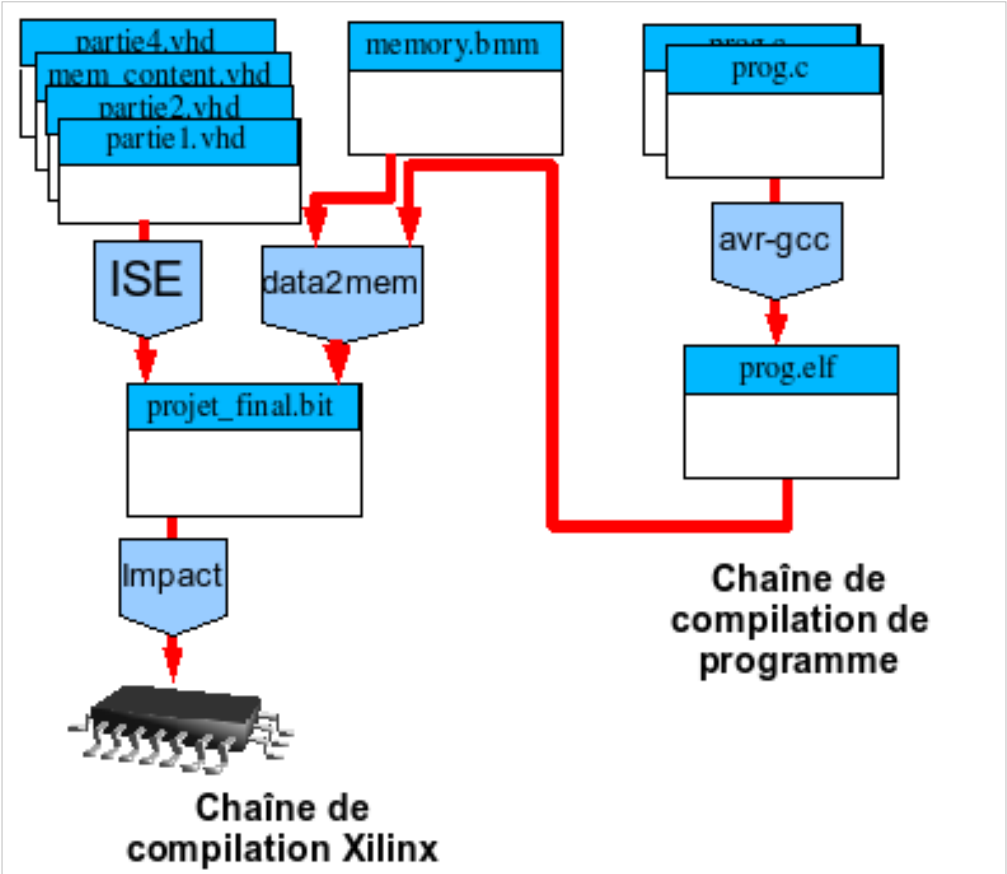
Les blocs rouges non utilisés le sont en fait avec la mémoire RAM de ce cœur.

Puisque les rectangles blancs sont plus petit que les rectangles rouges qui les contiennent, cela veut dire qu'il est possible d'augmenter la taille programme de ce **coreATMega8**.

Examiner d'autres solutions

Cette solution est sur certains points plus détaillée dans le chapitre Programmer in Situ et déboguer de ce livre.

Voici donc cette autre solution. Comme le montre le dessin, elle utilise un utilitaire Xilinx qui s'appelle **data2mem**. Cette solution ne présente plus l'inconvénient précédent puisqu'elle ne nécessite plus une recompilation complète du projet VHDL : elle modifie directement le fichier BIT en y mettant le programme compilé.



Les chaines de compilations pour éviter les longues compilations

Il nous faut retenir que data2mem est capable de traiter un fichier ELF. C'est le format de sortie par défaut des compilateurs du GNU et, je l'espère, d'autres compilateurs. Donc tout cœur embarqué qui utilise un compilateur du GNU peut utiliser cette technique. Elle est par exemple utilisée avec le microBlaze (de manière complètement transparente), et openMSP430 par exemple. Par contre, elle devient difficile à utiliser avec un PIC de la famille 16F qui a souvent une chaîne de compilation qui génère un fichier HEX !

La mauvaise nouvelle, il y en faut bien une, est que si l'on regarde attentivement la figure on voit qu'un nouveau fichier BMM est nécessaire (ici memory.bmm). En gros, si vous utilisez les cœurs embarqués estampillés Xilinx (picoBlaze ou microBlaze) vous obtiendrez ce fichier BMM de manière automatique (plus ou moins facilement) mais si vous sortez des sentiers battus vous devrez le créer de toute pièce et donc lire la documentation de 50 pages de data2mem. Nous avons lu quelque part que PlanAhead savait le créer automatiquement mais nous n'avons pas réussi à l'utiliser pour cela mais seulement pour trouver l'information nécessaire à la création de ce fichier BMM. Peut être qu'un lecteur sait le faire ?

Annexe II : le fichier ucf

Voici sans commentaires le fichier ucf :

```
NET     I_CLK_50       PERIOD = 20 ns;
NET     L_CLK           PERIOD = 35 ns;

NET     I_CLK_50       TNM_NET = I_CLK_100;
NET     L_CLK           TNM_NET = L_CLK;

NET     "I_CLK_50"       LOC = "T9";
#NET     I_RX            LOC = M3;
#NET     Q_TX            LOC = M4;

# single LEDs
#
NET     Q_LEDS<0>       LOC = "K12";
NET     Q_LEDS<1>       LOC = "P14";
NET     Q_LEDS<2>       LOC = "L12";
NET     Q_LEDS<3>       LOC = "N14";

# DIP switch(0 ... 7) and two pushbuttons (8, 9)
#
NET     I_SWITCH<0>     LOC = "f12";
NET     I_SWITCH<1>     LOC = "g12";
NET     I_SWITCH<2>     LOC = "h14";
NET     I_SWITCH<3>     LOC = "h13";
NET     I_SWITCH<4>     LOC = "j14";
NET     I_SWITCH<5>     LOC = "j13";
NET     I_SWITCH<6>     LOC = "k14";
NET     I_SWITCH<7>     LOC = "k13";

NET     I_Reset<0>     LOC = "L13";
NET     I_Reset<1>     LOC = "L14";

#NET     I_SWITCH<*>     PULLUP;

#VGA
net "hsynch" loc="R9";
net "vsynch" loc="T10";
net "red" loc="R12";
net "blue" loc="R11";
net "green" loc="T12";
```

Annexe III : Le GNU C

Pour compiler manuellement vous lancez :

```
avr-gcc -g -mmcu=atmega8 -Wall -Os -c hello.c
avr-gcc -g -mmcu=atmega8 -o hello.out -Wl,-Map,hello.map hello.o
avr-objcopy -R .eeprom -O ihex hello.out hello.hex
```

Si vous voulez connaitre le code source assembleur :

```
avr-objdump -S hello.out
```

puisqu' hello.out a été créé avec la deuxième commande à partir de hello.o

Quelques fonctions non standard de la librairie du C pour AVR sont données :

Fonction	Définition	Exemple
_BV(x)	positionne un bit spécifique	`char result = _BV(PINA6); // met 64 dans result`
void sbi (uint8_t port, uint8_t bit)	positionne un bit spécifique à 1	`sbi (PORTB, 3); //#define PORTB 0x18` `sbi (PORTB, PINA3);`
void cbi (uint8_t port, uint8_t bit)	positionne un bit spécifique à 0	`cbi (PORTB, 3); //#define PORTB 0x18` `cbi (PORTB, PINA3);`
uint8_t bit_is_set (uint8_t port, uint8_t bit);	teste si un bit spécifique est à 1	`//#define PINB 0x16` `uint8_t result = bit_is_set (PINB, PINB3);`
uint8_t bit_is_clear (uint8_t port, uint8_t bit);	teste si un bit spécifique est à 0	`//#define PINB 0x16` `uint8_t result = bit_is_clear (PINB, PINB3);`
uint8_t inp (uint8_t port);	lit un registre spécifique et retourne le résultat	`//#define SREG 0x3F` `uint8_t res = inp (SREG);`
uint16_t __inw (uint8_t port);	lit un registre spécifique 16 bits et retourne le résultat	`//#define TCNT1 0x2C` `uint16_t res = __inw (TCNT1);`
uint16_t __inw_atomic (uint8_t port);	lit un registre spécifique 16 bits et retourne le résultat sans interruption possible	`//#define TCNT1 0x2C` `uint16_t res = __inw (TCNT1);`
outp (uint8_t val, uint8_t port);	sort une valeur spécifique sur un port spécifique	`//#define PORTB 0x18` `outp(0xFF, PORTB);`
__outw (uint16_t val, uint8_t port);	sort une valeur spécifique 16 bits sur un port spécifique	`//#define OCR1A 0x2A` `__outw(0xAAAA, OCR1A);`
__outw_atomic (uint16_t val, uint8_t port);	sort une valeur spécifique 16 bits sur un port spécifique sans interruption	`//#define OCR1A 0x0A` `__outw_atomic(0xAAAA, OCR1A);`

Références

[1] http://moutou.pagesperso-orange.fr/ER2/ATMega8_pong_VGA.zip
[2] http://opencores.org/project,cpu_lecture
[3] http://perso.wanadoo.fr/moutou/ER2/SiliCore1657.pdf
[4] http://opencores.org/project,avr_core
[5] http://opencores.org/project,avrtinyx61core

Sources et contributeurs de l'article

Very High Speed Integrated Circuit Hardware Description Language/Embarquer un Atmel ATMega8 *Source*: http://fr.wikiversity.org/w/index.php?oldid=334354 *Contributeurs*: Crochet.david, Cynddl, JackPotte, SergeMoutou, 11 modifications anonymes

Source des images, licences et contributeurs

Fichier:Zeichen 114.svg *Source*: http://fr.wikiversity.org/w/index.php?title=Fichier:Zeichen_114.svg *Licence*: Public Domain *Contributeurs*: Bundesministerium für Verkehr, Bau- und Wohnungswesen (German Federal Ministry of Traffic, Building and Housing)

Fichier:Searchtool.svg *Source*: http://fr.wikiversity.org/w/index.php?title=Fichier:Searchtool.svg *Licence*: GNU Lesser General Public License *Contributeurs*: David Vignoni, Ysangkok

Fichier:Pong Screen.png *Source*: http://fr.wikiversity.org/w/index.php?title=Fichier:Pong_Screen.png *Licence*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributeurs*: SergeMoutou

Fichier:ChaineCompil1.png *Source*: http://fr.wikiversity.org/w/index.php?title=Fichier:ChaineCompil1.png *Licence*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributeurs*: SergeMoutou

Fichier:ChaineCompil2.png *Source*: http://fr.wikiversity.org/w/index.php?title=Fichier:ChaineCompil2.png *Licence*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributeurs*: SergeMoutou

Fichier:MemoryPlanAhead1.png *Source*: http://fr.wikiversity.org/w/index.php?title=Fichier:MemoryPlanAhead1.png *Licence*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributeurs*: SergeMoutou

Fichier:ChaineCompil3.png *Source*: http://fr.wikiversity.org/w/index.php?title=Fichier:ChaineCompil3.png *Licence*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributeurs*: SergeMoutou

Licence