

Carte SHiX 2.0

Alexis Polti
polti@enst.fr

16 mars 2005

Résumé

La carte SHiX 2.0 est un système embarqué, conçu originellement à des fins d'enseignement et de recherche. Elle associe un microprocesseur Super-H (SH4 7750R d'Hitachi) à un FPGA Stratix EP1S25 d'Altera, ainsi qu'un certain nombre de périphériques. Sa conception a été dictée par la puissance de calcul, la reconfigurabilité et le prix de revient.

Ce document décrit son architecture, ainsi que les développements qui ont été effectués dessus, tant du point de vue matériel que logiciel.

Bien entendu, les développements avançant rapidement, ce document ne prétend pas être à jour, mais il devrait fournir un bon point de départ pour toute personne cherchant de la documentation sur cette carte.

Chapitre 1

Hardware

Ce chapitre décrit le fonctionnement matériel de la carte, son architecture, certains des choix qui ont été faits, ainsi que quelques modules implémentés dans le FPGA.

1.1 Généralités

La carte SHiX 2.0 comporte, comme indiqué sur la figure [1.1](#) page suivante :

- un microprocesseur Super-H d'Hitachi : SH4 7750R,
- un Stratix EP1S25 d'Altera, disposant de sa propre flash de configuration EPC8,
- 32Mo de SDRAM,
- 32Mo de mémoire flash NAND,
- un circuit d'acquisition vidéo (décodeur multistandard SAA7113 de Philips),
- un circuit de restitution vidéo (encodeur CS4954 de Cirrus Logic),
- 36 Mbits de SSRAM, dédiée au FPGA,
- un contrôleur USB 1.1 hôte, SL811 de Cypress,
- un contrôleur USB 2.0 esclave CY7C68001,
- deux UART
 - l'un sans fifo ni signaux de contrôle (console),
 - l'autre avec fifo de 16 octets et signaux de contrôle (CTS / DTR),
- divers afficheurs (LEDs, afficheur 7 segments), et quelques switches,
- un générateur d'horloge non programmable CY22392,
- trois ports d'extension
 - un port d'extension general purpose 32 bits 3.3V et compatible 5V (protégé),
 - deux ports I2C,
- une alimentation à découpage,
- un capteur de la température du FPGA.

Le PCB comporte 8 couches, dont 4 de signaux. Le stackup a été soigneusement choisi, et toutes les lignes critiques du point de vue intégrité du signal ont été simulées.

1.2 Alimentation

Plusieurs alimentations sont nécessaires :

- 3.3V pour la majorité des périphériques, ainsi que pour les IO du processeur et du FPGA,
- 1.5V pour les coeurs du processeur et du FPGA,
- 5V pour l'alimentation du bus USB, du bus vidéo, des LEDs, et des switches de protection du bus d'extension.

Les consommations estimées se répartissent comme indiqué dans la table [1.1](#) page [5](#) (seuls certains périphériques 3.3V sont mentionnées). La consommation du Stratix est estimée à partir des paramètres suivants :

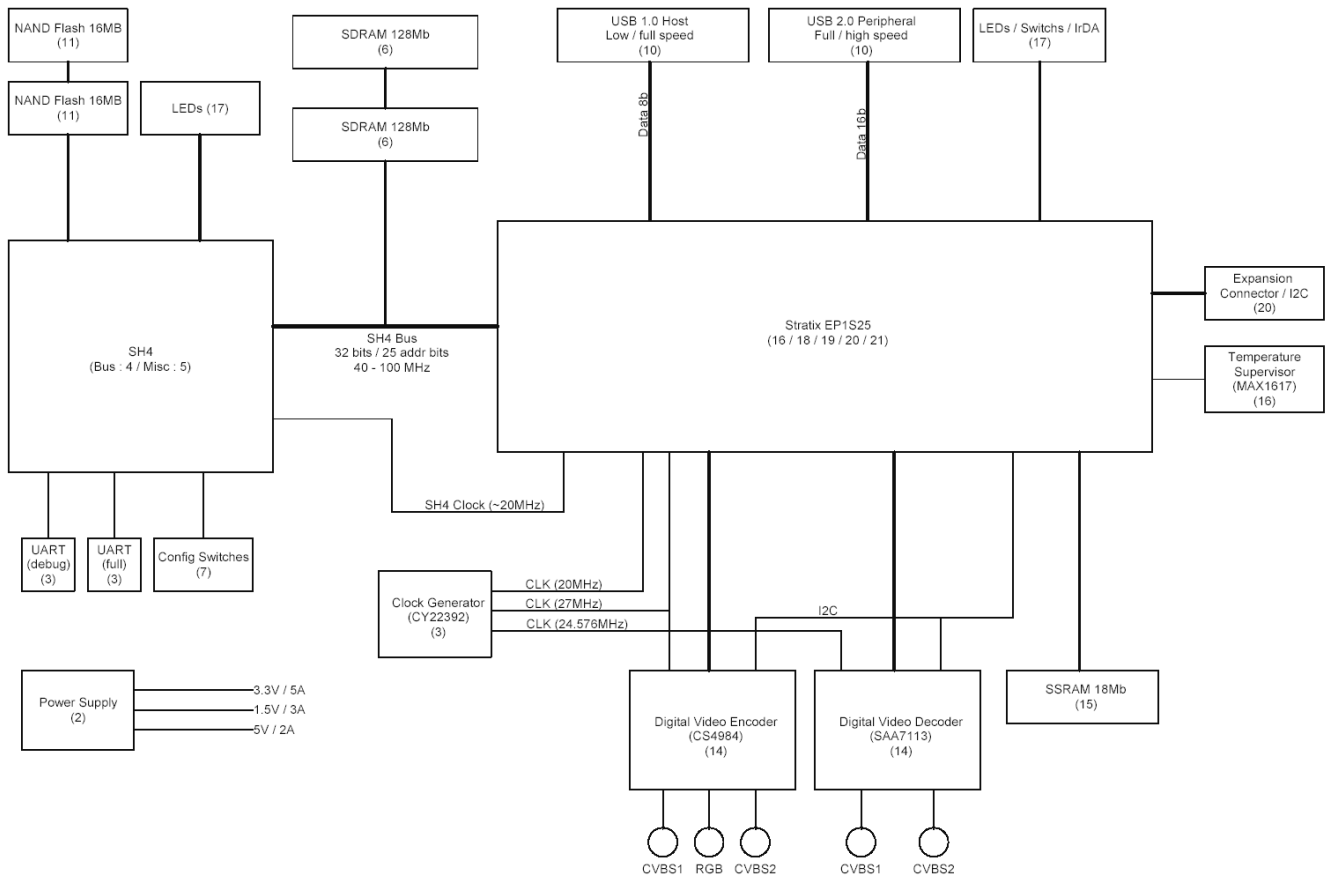


FIG. 1.1 – Architecture globale de la carte

- Clocktree
 - 40.000 flip-flop à 80MHz,
- LE
 - 10.000 flip-flop à 80MHz, toggle 50
 - 1.000 flip-flop à 80MHz, toggle 75
 - 15.000 flip-flop à 54MHz, toggle 50
- DSP
 - 10 blocs à 54 MHz, toggle 100
- PLL
 - toutes en marche, 82, 54 et 27MHZ,
- RAM
 - 300kb à 80Mhz,
- IO
 - 100 à 80MHz, toggle 100
 - 300 à 80MHz, toggle 50
 - 200 à 54MHz, toggle 50

Ces chiffres *ne représentent rien de réel*, ils permettent juste d'avoir une idée de la consommation du Stratix pour un design moyen. L'alimentation doit aussi prendre en compte les périphériques 3.3V oubliés, ainsi que ceux situés sur le bus d'extension, ainsi que le séquençage des power-up / power-down (voir datasheet du SH7750R).

Elle a donc été architecturée comme suit :

- 3.3V : deux régulateurs à découpage LT1959 en parallèle, capables de fournir à eux deux 6A en continu.

Tension	Device	Intensité
3.3V	IO Stratix	1.1A
	IO SH4	180mA
	2x SDRAM	640mA
	2x SSRAM	325mA
	Total	2.245A
2.5V	cœur Stratix	2.2A
	cœur SH4	580mA
	Total	2.8A

TAB. 1.1 – Consommations estimées

Pour éviter les problèmes des régulateurs en parallèle, l'un d'eux est maître de la régulation (compensation), l'autre est esclave. Un soft start évite les surtensions au power-up, et retarde la montée du 3.3V par rapport à celle du 1.5V conformément aux spécifications du SH7750R.

- 1.5 : un régulateur à découpage LT1959 suivi d'un LDO 1.5V LT1587. Le profil de consommation sur le 1.5V n'étant pas maîtrisé du tout, de forts pics de consommation peuvent apparaître, perturbant la boucle de régulation. Le 1.5 devant rester dans des marges très restreintes, j'ai préféré jouer la sécurité est réguler en deux étapes : abaissement par découpage à 2.2V environ, puis régulation linéaire 1.5V.
- 5V : peu chargé, il est généré par un régulateur linéaire 1A.

Les alimentations 3.3V et 1.5V ont été simulées (en régime continu, power-up et transient). La marge de stabilité est suffisante pour assurer leur stabilité même dans des conditions extrêmes irréalistes.

1.3 Processeur

Le processeur est un SH4 7750R d'Hitachi. Il a été choisi pour ses performances (430 MIPS à 240 MHz, 1.7GFlops), sa faible consommation (0.9W en crête), sa disponibilité et son coût. C'est un RISC 32bits superscalaire (2 instructions en parallèle), possédant 2 caches internes (16kB instructions et 32kB données), une MMU et divers périphérique intégrés, pour lequel la suite de compilation GNU C (gcc et binutils) a été portée. Il possède un bus externe 64 bits, utilisé ici en 32 bits, multi-protocoles.

1.3.1 Horloges, resets

Horloges

Son horloge principale (`clk_sh4`) est générée par le FPGA, à partir du générateur d'horloge Cypress. Une première PLL interne multiplie cette horloge par 6 ou 12. Puis des diviseurs se chargent de générer les différentes horloges (cœur, modules et bus). Une deuxième PLL spécifique au bus externe assure la génération d'une horloge bus (`sh4_ckio`) propre et stable.

Par défaut sur les cartes actuelles, on a :

- horloge entrée (`clk_sh4`) : 20MHz,
- horloge coeur : 240Mhz,
- horloge modules : 60MHz,
- horloge bus (`sh4_ckio`) : 60MHz.

Attention : si on modifie ces valeurs (par les switches de configuration), il faut penser à recompilier le noyau Linux avec la bonne valeur de fréquence modules, sinon les UART ne fonctionneront pas !

Le processeur dispose aussi d'une RTC, pilotée par un quartz de 32.768kHz.

Reset

Le processeur dispose de deux types de reset : manuel (power-up reset), et soft. Ils diffèrent entre eux par la façon dont ils sont générés, et par les parties du processeur qui sont affectées. De façon grossière, un reset manuel remet le processeur entier à zéro, alors que le reset soft n'affecte que son cœur. C'est l'état de la ligne `SH4_SCK2_MRESETn` quand `SH4_RESETEn` est bas qui détermine si le reset est manuel ou soft.

Un reset manuel (power-up reset) déclenche aussi la lecture de certaines lignes (`MD[8 :0]`), qui déterminent alors la configuration du processeur. Après le reset, ces lignes prennent éventuellement une autre fonction (UART, ...). Un buffer 3 états (U41) assure l'indépendance des lignes `MD[8 :0]` après la phase de reset.

La configuration par défaut, sur laquelle se basent la chaîne de compilation ainsi que les différents modules du FPGA déjà écrits, est la suivante :

- horloge module : 60MHz, horloge bus 60MHz, horloge d'entrée externe,
- zone de boot (Area 0) en 32 bits, type SRAM,
- processeur en little-endian, maître du bus.

1.4 Bus

Le bus mémoire du processeur est un point critique du design. Il comporte 26 lignes d'adresse, de 8 à 64 lignes de données, des signaux de contrôle, et des signaux d'arbitrage. Les 32 lignes de données de poids fort sont partagées avec les GPIO du SH4. Il a donc fallu choisir entre GPIO et bus 64 bits. J'ai préféré utiliser les GPIO pour des LED et la flash NAND, et restreindre le bus données à 32 bits.

1.4.1 Etude électrique

Le bus externe du SH4 ne peut fonctionner à fréquence maximale (80Mhz) que si sa charge est limitée. Les timings donnés dans la datasheet portent sur un bus chargé à 30pF, pouvant alors atteindre sa fréquence max. Une charge plus grande augmente les timings d'environ 1.5ns pour 25pF, et ralentit donc le bus, sans compter les 1.5nS de délai supplémentaires du au boîtier QFP.

Il y a donc un compromis à faire entre le nombre de périphériques branchés directement sur le bus du SH4, et sa fréquence maximale. J'ai choisi arbitrairement de maximiser la fréquence du bus, et donc de limiter les périphériques y étant reliés aux deux SDRAM et au Stratix. Les autres périphériques (USB, ...) sont donc reliés au Stratix. Il est nécessaire d'implémenter un pont dans le Stratix pour que le SH4 puissent y accéder. Actuellement, rien n'est encore écrit pour l'USB.

La charge du bus choisie se répartit comme suit :

- SH4 : 10pF par pin
- Stratix : de 8.2pF (bancs 1, 2, 5 et 6) à 11.5pF (bancs 3, 4, 7 et 8). Pour les horloges, de 4.4 à 11.5pF.
- SDRAM : de 4.0 à 6.5pF selon le type de la pin.
- PCB : 3pF environ.

Des contraintes de placement ont fait que le banc 7 était le plus adapté à la connexion avec le SH4, aboutissant ainsi à une charge d'environ 30.5pF. Le bus peut donc fonctionner à fréquence maximale (80MHz).

Toutes les lignes du bus, horloge, données, adresses et contrôle, ont été simulées : temps de propagation, round trip délai, overshoot, undershoot, crosstalk, ... en se basant sur les modèles IBIS des constructeurs, garantissant ainsi le fonctionnement du bus, même à fréquence maximale.

1.4.2 Partitionnement logique

Les adresses présentes sur le bus comportent 26 bits, représentant les 26 bits de poids faible de l'adresse physique de l'accès en cours. Pour un descriptif précis et complet de l'espace mémoire du

SH4, on se rapportera à sa datasheet.

L'espace mémoire du SH4 est séparé en plusieurs niveaux : l'espace virtuel, propre à chaque processus, l'espace physique interne, et l'espace externe.

Espace virtuel : il comporte 32 bits d'adresses. La MMU se charge de traduire les adresses virtuelles en des adresses physiques internes sur 32 bits aussi. Lorsque la MMU n'est pas utilisée, l'espace virtuel est le même que l'espace physique interne.

Espace physique interne : séparé en plusieurs zones, appelées *zones physiques* ou *zones internes* : P0 à P4 en mode superviseur, et U0 en mode utilisateur. Ces zones diffèrent par le fait qu'elles sont ou non cachées : P2 et P4 sont non cachables, P1, P3 et U0 sont cachables. L'espace physique interne est ensuite mappé sur l'espace externe en supprimant les 3 bits de poids fort des adresses.

Espace physique externe : séparé lui aussi en 8 zones, appelées *zones externes*, de même taille : Area0 à Area7, et disposant chacune de 26 bits d'adresse et son propre chip select. Le bus du SH4 sort donc 26 lignes d'adresse, et 7 chip select (le 8^{me} n'est pas disponible).

A partir d'une adresse virtuelle, on obtient donc

- une adresse physique interne, par l'intermédiaire de la MMU,
- puis une adresse physique externe par suppression de 3 + 3 bits de poids fort. Les premiers trois bits détermine la zone physique (et l'accès ou non au cache), les derniers trois bits donnent la zone externe et donc le chip select.

Chaque zone externe est donc mappée dans plusieurs zones internes. Par exemple, Area0 est accessible dans la zone P0 (cachée), P1 (cachée), P2 (non-cachée), P3 (cachée), P4 (non cachée) et U0 (cachée). La zone Area7 étant réservée aux registres, un trou apparaît donc dans chaque zone P0 à P4 / U0.

On a donc, pour les zones internes, en mode privilégié¹ :

Adresse	Zone interne
0x0000 0000 - 0x7FFF FFFF	P0 area, cachable
0x8000 0000 - 0x9FFF FFFF	P1 area, cachable
0xA000 0000 - 0xBFFF FFFF	P2 area, non-cachable
0xC000 0000 - 0xDFFF FFFF	P3 area, cachable
0xE000 0000 - 0xFFFF FFFF	P4 area, non-cachable

Et chaque zone interne permet d'accéder aux zones externes², décrites ci-dessous :

Adresse	Zone externe
0x0000 0000 - 0x03FF FFFF	Area 0 (boot, ROM)
0x0400 0000 - 0x07FF FFFF	Area 1 (FPGA)
0x0800 0000 - 0x08FF FFFF	Area 2 (SDRAM 1, 16M)
0x0C00 0000 - 0x0CFF FFFF	Area 3 (SDRAM 2, 16M)
0x1000 0000 - 0x13FF FFFF	Area 4 (FPGA)
0x1400 0000 - 0x17FF FFFF	Area 5 (FPGA)
0x1800 0000 - 0x1BFF FFFF	Area 6 (FPGA)
0x1C00 0000 - 0x1FFF FFFF	Area 7 (reserved)

Le bus peut être configuré de différentes manières pour chaque zone externe, en terme de

¹dit aussi, mode superviseur. En mode utilisateur (non privilégié), le mapping des zones est différent. On se reportera à la datasheet du 7750 pour plus de précision

²dont les adresses sont données ici pour un accès à partir de la zone interne P0

Pin SH4	Broche micromatch	Broche prise RS232 femelle	Pin PC	Direction
RX	3	3	TX	PC vers SH4
TX	5	2	RX	SH4 vers PC
CTS	6	4	DTR	PC vers SH4
RTS	4	1, 8, 6	CD, DSR, RI	SH4 vers PC
GND	1	5	GND	

TAB. 1.2 – Câblage de la liaison RS232 avec signaux de contrôle

- type d'accès : SRAM, Burst ROM, byte controlled SRAM, multiplexé, SDRAM, PCMCIA,
- wait state,
- largeur de bus de données : 8, 16, 32 ou 64 bits.

La configuration de la zone 0 (celle de boot) est donnée par les lignes MD[8 :0] lors d'un reset manuel. La configuration des autres zones est déterminée par des registres internes, qui sont programmés par le boot loader. La configuration actuelle choisie est la suivante :

- CS2 et CS3 : SDRAM, Cas latency 2, 32 bits
- CS0, CS1, CS4, CS5, CS6 : 32 bits, SRAM, 1 wait state interne (+ autant qu'on veut supplémentaires par la pin RDY_n)

1.4.3 Autres périphériques

Flash

La flash est de type NAND, pour son faible coût de revient. Elle est arrangée en deux chips de 16Mo chacun. Du fait de l'interface particulière des flashes NAND, elle a été câblée sur les GPIO du SH4. Par conséquent, *le SH4 ne peut pas booter directement sur la flash*. Une ROM de boot doit être instanciée dans le Stratix, permettant de lire la Flash et de charger le noyau Linux. Plus de détails à la section 2.2 page 12.

UARTs

Les deux UART du SH4 sont utilisés, et disposent de transceivers électriques RS232. Les signaux de contrôle du SCIF (RTS et CTS) sont câblés.

Attention :

- CTS : ce signal est une entrée du SH4, qui est bien un Clear To Send : si elle est à l'état bas, le SH4 peut envoyer des données, sinon la transmission est bloquée.
- RTS : ce signal est une sortie du SH4. Il est asserté (état bas) quand le SH4 peut recevoir des données (le seuil de déclenchement, en fonction du remplissage de la FIFO est réglable en soft).

La gestion des ports série sous Windows ayant quelques lacunes, tous les protocoles de contrôle RTS/CTS et DSR/DTR ne sont pas implémentés. Il faut donc réaliser un câblage des pins de contrôle du SH4 sur celle du port série du PC en adéquation avec le protocole utilisé par le logiciel de terminal sur le PC. En utilisant TeraTerm 3.1 pour le logiciel de contrôle du port série, avec flow-control *hardware*, le câble est réalisé comme indiqué tableau 1.2. En d'autres termes, les broches CTS/RTS côté SH4 sont mappées sur DSR/DTR côté PC, on utilise alors le protocole DSR/DTR sur le logiciel de terminal (TeraTerm, mais Hyperterminal fonctionne correctement aussi avec ce mapping-ci).

Autres périphériques

Les autres périphériques (ROM de boot, USB, ...) sont accessibles à travers le Stratix, comme expliqué précédemment.

1.5 Stratix

1.5.1 Configuration

La configuration du Stratix au power up peut s'effectuer de deux façons : par sa flash de configuration, une EPC8, ou par l'interface ByteBlaster. Le jumper J29 permet de choisir l'interface utilisée.

Si J29 est en position ByteBlaster, le connecteur J20 est relié directement au Stratix, permettant de reprogrammer par ByteBlaster. Si J29 est en position EPC8, le connecteur J20 est relié à l'interface JTAG de l'EPC8, et le Stratix est configuré automatiquement au boot par l'EPC8 avec le dernier programme qui y a été flashé.

La led rouge D22, située à côté du connecteur de programmation J20, est allumée si le Stratix est correctement configuré.

Un même câble permet de relier J20 à un PC, et sert, selon la configuration de Quartus et de J20, soit d'interface ByteBlaster, soit d'interface JTAG. *Bug* : à cause d'une inversion de pistes sur le PCB, l'ordre des signaux est inversé quand on est en mode ByteBlaster. Il faut donc, soit patcher la carte entre U47 et U44 pour décroiser les signaux (ainsi le même câble est utilisé que ce soit en ByteBlaster ou en JTAG), soit utiliser une carte d'adaptation du câble quand on est en mode ByteBlaster si la carte SH4 n'est pas patchée.

Configuration par EPC8

Pour configurer le Stratix par sa flash EPC8 automatiquement au power-up, il faut avoir au préalable programmé l'EPC avec le fichier de programmation idoine. L'EPC8 est configurée en mode JTAG, par le même connecteur (J20) que l'interface ByteBlaster. Quartus génère par défaut un fichier de programmation `.sof` destiné au Stratix en mode ByteBlaster. Il faut donc convertir ce fichier en un fichier de programmation pour l'EPC8. Cela est fait par le menu *File / Convert Programming File...*, et en utilisant les options suivantes :

- *Configuration device* : EPC8
- *Mode* : 1-bit Passive Serial
- *Soft Data / Page 0 : Add File...*, puis choisir le fichier `.sof`
- *Options / Compression Mode* : coché
- *Options / Clock source* : internal
- *Options / Clock frequency* : 10MHz
- *Options / Divide clock frequency by* : 1,0
- *Options / Disable nCS and OE...* : non coché
- *File name* : choisir un fichier de sortie `.pof`

puis en cliquant sur OK.

On peut aussi cliquer sur *Open Conversion Setup Data* et choisir le fichier `to_EPC8Q100_10MHz_compression.cof` qui positionne automatiquement les options à leurs valeurs correctes. Puis ouvrir la fenêtre de programmation (*Tools / Programmer*), choisir le mode JTAG, choisir le fichier `.pof` généré, cocher *Program / Configure* et *Verify* et cliquer sur *Start* (après avoir mis la carte sous tension, et déplacé J20 en position EPC8).

Configuration directe du Stratix en ByteBlaster

Quartus génère directement un fichier `.sof` adapté à la configuration du Stratix en ByteBlaster. Il suffit d'ouvrir la fenêtre du programmeur (*Tools / Programmer*), choisir *Passive Serial*, puis *Add File...* du fichier `.sof`, et cliquer sur *Start* après avoir mis la carte sous tension et positionné J20 sur ByteBlaster (BB).

Si la configuration s'est effectuée correctement, la led rouge D20 doit s'allumer. Il est parfois nécessaire de faire plusieurs tentatives de programmation, la longueur des câbles générant des parasites.

1.6 Autres périphériques

A écrire...

Chapitre 2

Logiciel

Linux a été porté sur la plateforme SHiX 2.0, même si son utilisation n'est pas imposée. Mais qu'il soit utilisé ou non, il est important de connaître certains points clefs comme

- l'organisation mémoire de la carte,
- l'ordre d'exécution des différents modules,
- l'organisation des systèmes de fichiers,
- ...

C'est l'objectif de cette partie.

2.1 Organisation de la mémoire

2.1.1 Zones mémoire

L'organisation mémoire a déjà été abordée à la section 1.4.2 page 6. L'espace mémoire physique est partagé en 8 zones, disposant chacune de son chip select ($SH4_CS_{xn}$). Pour chacune de ces zones, le comportement du bus est configurable, en type (ROM, SRAM, multiplexé, SDRAM, PCMCIA), en nombre de wait state à l'intérieur de chaque accès, en nombre de wait state entre chaque accès, etc.

L'organisation choisie actuellement est la suivante :

Zone	Description	Adresse ¹	Type	Wait state internes
CS0	Zone de boot	0xA0000000-0xA3FFFFFF	SRAM	1 + $SH4_RDY_n$
CS1	Zone utilisateur (stratix)	0xA4000000-0xA7FFFFFF	SRAM	1 + $SH4_RDY_n$
CS2	SDRAM 0 (U5)	0xA8000000-0xABFFFFFF	SRAM	0
CS3	SDRAM 1 (U37)	0xAC000000-0xAFFFFFFF	SRAM	0
CS4	Zone utilisateur (stratix)	0xB0000000-0xB3FFFFFF	SRAM	1 + $SH4_RDY_n$
CS5	Zone utilisateur (stratix)	0xB4000000-0xB7FFFFFF	SRAM	1 + $SH4_RDY_n$
CS6	Zone utilisateur (stratix)	0xB8000000-0xBBFFFFFF	SRAM	1 + $SH4_RDY_n$
CS7	reservé	non accessible	-	-

Le SH4 boot à partir de l'adresse 0 (0xA0000000). Mais le bus, malgré les nombreuses configurations qu'il possède, ne sait pas accéder de lui-même à une flash NAND. Il n'est pas donc possible de booter directement sur la flash NAND. Une ROM est donc instanciée dans le FPGA, qui contient un bootloader minimal, décrit à la section 2.2 page suivante.

¹les adresses sont données à partir de la zone physique interne P2 (0xA...-0xB...)

2.1.2 Emplacement des exécutables

Le bootloader en ROM est logé en 0xA0000000 (adresse 0, de boot). Le bootloader peut charger un exécutable par la ligne série (puis l'exécuter), ou bien aller lire le kernel Linux dans la flash NAND, et l'exécuter. Dans les deux cas, les exécutables sont prévus pour être logés en 0x8C000000, c'est-à-dire zone P1 (cachée) / CS3. Le noyau Linux est donc configuré pour être exécuté à partir de 0x8C000000.

La pile du bootloader est en 0xA8FF0000 (P2 / CS3), c'est-à-dire quelque part vers la fin du CS3. Cela implique que le noyau Linux ou un exécutable chargé par ligne série doivent être plus petit que 15.9 Mo (0xFF0000) moins la taille maximale de la pile (sur le SH4 elle est descendante). Cela ne pose aucun problème pour le noyau Linux (rarement plus gros que 2Mo), et ne devrait pas en poser pour la majorité des exécutables stand-alone.

2.1.3 Emplacement des périphériques

Le Stratix est vu par le SH4 comme un périphérique mappé en mémoire, accessible à partir des zones externes 0, 1, 4, 5 et 6. Chaque application hardware va définir ses propres périphériques dans les différentes zones, et donc son propre mapping mémoire. Dans un souci d'organisation, et de partage de code, il est important de respecter certaines règles.

Certaines adresses sont allouées de façon globale, et personne n'a le droit de les modifier sans accord du responsable principal du code (actuellement Alexis). D'autres adresses sont libres, et utilisables à volonté.

Les adresses figées :

- CS0, pour le boot,
- CS1, pour des registres divers
- CS4, pour la vidéo.

Les adresses libres :

- CS5 ,
- CS6.

Si on souhaite ajouter un périphérique ou un registre dans une zone réservée, il convient de prévenir le responsable de l'archive code pour éviter tout conflit (immédiat ou futur).

Les périphériques actuellement définis sont donnés dans le fichier Verilog `top.v`, et récapitulés ci-dessous :

```
CS0 :
    0xA0000000 Internal boot ROM (contain bootloader)
    0xA0400000 Video sram direct access (write only)
CS1 :
    0xA4000000 I2C 0 (on board video controllers)
    0xA4000001 I2C 1 on JP38
    0xA4000002 I2C 2 on JP39
    0xA4000003 I2C 3 on expansion connector
```

Attention : *il est primordial qu'aucun driver ne change les settings d'accès au bus!!!* Les settings, en terme de format et de wait states (registres BCR1, BCR2, BCR3, WCR1, WCR2 et WCR3), sont programmés par le bootloader, et uniquement par celui-ci. Toute tentative de les modifier ailleurs risquerait de rendre un périphérique inutilisable et son debug impossible.

2.2 Bootloader

2.2.1 Organisation de la flash

Cette partie décrit l'organisation actuelle de la flash NAND. Il est probable qu'elle sera modifiée prochainement pour avoir plus de souplesse dans la gestion des partitions, et un boot plus rapide.

Introduction

La flash est une flash de type NAND. Elle diffère des flashs classiques (NOR) par :

- sa densité, plus grande que les flash NOR, grâce à des points mémoire plus petits,
- son coût, qui à capacité égale est donc plus faible,
- sa rapidité, plus grande que celle des flashs NOR,
- son type d'interface, différente des interfaces type SRAM des flashs habituelles,
- la présence autorisée et fréquente (5%) de blocs défectueux,
- son layout : chaque page comporte des données habituelles, plus quelques octets dits Out Of Band (OOB) permettant entre autres d'implémenter de l'ECC.

Les deux caractéristiques les plus importantes ici sont l'interface et le layout (OOB). L'interface ne comporte ni bus d'adresse ni bus de données, mais un bus de commande bidirectionnel et des lignes de contrôle. Chaque accès se compose d'une phase d'envoi de commande (lecture, écriture, effacement, reset, ...), puis d'une ou plusieurs phases d'exécution proprement dite. En d'autres termes, il n'est pas possible de connecter cette flash directement au bus processeur du SH4. C'est pour cela qu'elle a été connectée aux GPIO du SH4.

Pour le layout, la flash est partagée en 1024 blocs de 16ko environ.

Bloc Un bloc est la plus petite partie qu'on puisse effacer. Autrement dit, pour effacer 20 octets, on doit lire le bloc qui les contient, effacer ce bloc, puis ré-écrire tout le bloc, sauf les 20 octets en question. Chaque bloc est subdivisé en 32 pages.

Pages Une page est la plus petite section qu'on puisse écrire. Il ne faut pas confondre écriture et effacement. Une écriture ne peut mettre des bits qu'à 0, un effacement qu'à 1. Une flash vierge ne contient que des 1. Quand on écrit, on programme des 0 aux endroits où il faut. On peut programmer une page sans avoir le bloc qui la contient si on ne fait que passer des bits à 0, et si on limite à 3 écritures. Au delà de trois écritures, le bloc doit être effacé pour être ré-écrit.

Données et OOB Chaque page comporte 512 octets de données, situés à des adresses consécutives, plus 16 octets dits Out Of Band (OOB). Ces 16 octets sont réservés à l'ECC, ainsi qu'à différents marquage. La plupart des file-systems spécifiques aux flashs NAND les utilisent comme marqueurs d'inode. Un de ces octets est important, c'est le numéro 517 (le 6ème des OOB) : le nombre de bits à 0 dans cet octet est le nombre de bit défectueux dans la page.

- Une page correcte contient donc 0xFF en 517.
- Une page avec un bit défectueux contiendra par exemple 0xFE (ou 0xEF, ou...).
- Une page avec deux bits défectueux contiendra par exemple 0xFC (ou 0x3F, ou...).

Les OOB comportent aussi six octets d'ECC : 520-521-522, qui portent sur les derniers 256 octets de data de la page, et 525-526-527 qui portent sur les 256 premiers octets de data de la page. Chaque triplet permet de corriger un bit défectueux.

Une page contenant plus d'un bit défectueux sera donc généralement non récupérable (sauf si chaque bit est dans une moitié différente de la page). On n'admet donc qu'un seul bit à 0 dans l'octet de statut (517).

Si une page comporte plus d'un bit défectueux, le bloc qui la contient en entier est marqué comme défectueux. La façon d'effectuer ce marquage dépend des file-systems et conventions utilisées, mais généralement soit on marque toutes les pages du bloc comme défectueuses, soit on ne marque que la page en question ainsi que la première du bloc.

De plus, certains flash NAND (les Disk On Chip par exemple) maintiennent une table des blocs défectueux. Ce n'est pas notre cas, la table des blocs est construite au boot en scannant la flash, et maintenue seulement en RAM par le système de fichier (YAFFS).

ECC Une flash NAND peut comporter jusqu'à 5% de blocs défectueux. Il est donc important d'utiliser l'ECC et de détecter et marquer les blocs défectueux. Normalement cela est géré de façon transparente par le système de fichier et l'OS, mais dans le cas du bootloader c'est à faire manuellement. En d'autres termes, l'écriture d'un fichier en flash par le bootloader (indépendamment de tout système de fichier) se décompose ainsi :

- tronçonnage du fichier en pages de 512 octets,

- calcul de l'ECC pour chaque page, ajout de 16 octets à ce fichier à la fin de chaque blocs de 512,
- écriture du nouveau fichier (data + OOB), mais seulement sur les blocs marqués valides.

File system

La flash doit contenir, au minimum, le noyau Linux (`vmlinuxx`) et le root file-system. Elle peut aussi stocker un file-system additionnel (utilisateur), d'autres noyaux, etc. . . Le choix du file-system est crucial : les flash NAND peuvent avoir des secteurs défectueux, et sont limitées en termes de ré-écriture. Il faut donc choisir un file-system adapté à leur fonctionnement. A ce jour, il en existe deux : JFFS2 et YAFFS.

JFFS2 : file-system compressé. Nécessite un parsing entier des secteurs de la flash au boot, pouvant être long. La génération d'une image (pour le formatage de la flash et la première écriture du root file-system) est compliquée.

YAFFS : file-system non compressé. Simple et rapide. Génération d'une image simplissime.

C'est donc YAFFS qui a été retenu, dans sa version 1. Il existe une version 2, plus adaptée aux flash de grande taille (>128Mo). Il sera possible, dans le futur, de l'implémenter, même si cela n'est pas d'actualité aujourd'hui.

Partitionnement

Le bootloader a pour rôle, en temps normal,

- de lire le kernel Linux sur la flash NAND,
- de le recopier en SDRAM en 0x8C000000,
- de l'exécuter (jump en 0x8C002000).

On aurait pu partir sur une partition unique, mais cela aurait nécessité de doter le bootloader des fonctions nécessaires à la lecture du file-system de cette partition, en plus des fonctions d'accès à la flash NAND. Or sa taille doit être minimale (il est stocké dans une ROM instanciée dans le FPGA). Ce n'est donc pas l'approche retenue.

La flash est partitionnée en deux partitions :

Numéro	Contenu	FileSystem	Offset	Longueur
0	Kernel (<code>vmlinux</code>)	juste fichier + OOB (no FS)	0	200 blocs (3.2Mo de données)
1	Root file-system	YAFFS	200 blocs	tout le reste de la flash

La première partition contient donc le noyau Linux, écrit tel quel (avec OOB), et la deuxième le Root File-System (YAFFS) qui s'étend du 201^{me} bloc du premier chip jusqu'à la fin du deuxième chip.

Le bootloader ne voit que le premier chip (16Mo). Linux est configuré pour agréger les deux chips de 16Mo en un seul virtuel de 32Mo.

Formatage et première écriture

Avant de pouvoir booter Linux, il est nécessaire d'avoir écrit un noyau en flash ainsi qu'un root filesystem minimal. Cela est effectué au moyen d'un bootloader dit de deuxième niveau : le bootloader en ROM charge celui de niveau 2 par la ligne série, puis il l'exécute. C'est le bootloader de niveau 2 qui se charge de scanner les flash pour détecter d'éventuels blocs défectueux, de n'effacer que les blocs valides, et de programmer au bon endroit le noyau et le root file system initial avec leurs OOB. Ce mécanisme est décrit en détail à la section sur le bootloader de deuxième niveau, ?? page ??.

2.2.2 Fonctionnement du bootloader

A la mise sous tension, le bootloader (de premier niveau, celui en ROM) affiche un message de bienvenue puis attend une seconde. Si, dans cette seconde, une touche est tapée (un caractère est reçu sur la ligne série SCI), alors il attend de recevoir un fichier binaire sur la deuxième ligne série (SCIF). Les signaux de contrôle sont utilisés pour pouvoir bénéficier du débit maximal. Si aucune touche n'est tapée pendant la première seconde, il va lire les 200 premiers blocs de la flash (en sautant les blocs défectueux), et recopie le tout en 0x8C000000. Puis il effectue un jump en 0x8C002000, et donne ainsi la main au noyau Linux.

Initialisations

Ce bootloader se charge de faire les première initialisation du processeur :

- initialisation de la SDRAM,
- initialisation du bus,
- initialisation des ports série,
- initialisation du pointeur de pile.

Puis il entre dans sa boucle principale (attente d'une seconde etc...).

Attention, pour le garder à une taille minimale, il n'est linké avec rien! Entre autres, la libC n'est *pas* utilisée. Cela implique qu'aucune fonction telle que printf n'est disponible, et que le segment de data, résidant en ROM, n'est pas recopié en RAM. Seul le BSS est mis à zéro.

Une implication est qu'on ne peut utiliser *que des variables locales et non statiques*. Elles sont alors sur la pile, sauf les constantes qui se retrouvent dans text ou rodata. Si on a besoin de variable dans le segment data (variable globale ou locales statiques), il est nécessaire de définir `NEED_COPY_DATA_SEG` dans le Makefile. Le code pour recopier le segment de data en RAM est alors inséré dans l'exécutable.

Mode réception ligne série

Dans le cas où un caractère est envoyé sur le SCI dans la première seconde, le bootloader se met en attente de réception d'un fichier sur l'autre ligne (SCIF). Le fichier doit être envoyé au format binaire (*sans* protocole de type XMODEM, ZMODEM ou autre, juste le binaire). Le contrôle de flux matériel RTS/DTS doit être activé. Le fichier doit être envoyé dans les 10 secondes. Puis dès que le premier octet est reçu, une pause de plus d'une seconde signifie la fin du fichier. Chaque octet reçu est recopié en RAM, en commençant en 0x8C000000. A la fin de la réception du fichier, un jump est effectué en 0x8C000000.

Le binaire à exécuter doit donc avoir été compilé pour s'exécuter en 0x8C000000 (ou toute autre zone interne, du moment que la zone externe est la même). Un exemple de script de link est donné ci-dessous² :

```
alexis@farfadet# cat linkram2.lds

/*OUTPUT_FORMAT("elf32-sh-linux", "elf32-sh-linux", "elf32-sh-linux")

OUTPUT_ARCH(sh)*/

MEMORY
{
    ROM (rx):    ORIGIN = 0xa0000000, LENGTH = 24k
    RAM1 (rw):   ORIGIN = 0xa8000000, LENGTH = 16M
    RAM2 (rw):   ORIGIN = 0xac000000, LENGTH = 16M
}

SECTIONS
{
    .text :

```

²ce script loge l'exécutable en zone P2 non cachable : 0xAC000000

```

{
    init.o(.text)
    *(.text)
    *(.*rodata*)
} > RAM2

. = ALIGN(4);

.data :
{
    *(.data)
    . = ALIGN(4);
} > RAM2

. = ALIGN(4);
bss_start = .;                /* BSS */
.bss :
{
    *(.bss)
    *(COMMON)
} > RAM2
. = ALIGN(4);
bss_end = .;
.stack :
{
    *(.stack)
    . = . + 10k; /* reserve room for stack */
    top_stack = .;
} > RAM2
.stab :
{
    *(.stab)
}
.comment :
{
    *(.comment)
}
.stabstr :
{
    *(.stabstr)
}
}

```

Mode Linux

Si aucun caractère n'est envoyé sur SCI pendant la première seconde, le bootloader va lire les 200 premiers blocs de la flash, qui contiennent normalement le noyau Linux. Il les recopie en RAM (en sautant les blocs défectueux) à l'adresse 0x8C000000, et effectue un jump en 0x8C002000, donnant ainsi la main au noyau.

Linux doit donc être configuré pour s'exécuter à partir de 0x8C000000. Plus précisément, la page 0 doit être en 0x8C000000, et le point d'entrée en 0x8C002000.

Si aucun noyau n'a été écrit en flash, le jump aura pour effet de finir par effectuer une instruction invalide provoquant le reboot de la carte.

Compilation et remarques

Ce bootloader, dans sa version actuelle, fait 2154 octets. Il est possible de réduire sa taille en optimisant certaines portions du code, mais des fonctionnalités supplémentaires lui seront bientôt ajoutées, portant sa taille à un peu moins de 4ko.

La compilation de ce bootloader est très simple : il se trouve dans le répertoire `standalone/tiny_bootloader`. Il faut commencer par effectuer un `source env.sh` pour ajouter (entre autres) à son PATH le chemin vers les outils GU de compilation. Puis un Makefile se charge de tout : `make clean && make && make mif`.

Un binaire ELF est alors, généré, puis transformé en binaire plat (pur). Un utilitaire, `bin2altmif` se charge de transformer ce binaire en un fichier d'initialisation de RAM (`.mif`) compréhensible par Quartus. C'est lors de la compilation du Stratix que le fichier `.mif` sera lu pour initialiser la ROM de boot.

2.3 Bootloader de niveau 2

Ce programme est appelé "bootloader de niveau 2" pour des raisons historiques. Ce n'est pas un bootloader, mais plutôt un utilitaire de formatage de la flash. Des fonctionnalités de debug et de moniteur lui seront ajoutées plus tard.

2.3.1 Utilisation

Avant tout, il faut le compiler. `make binary` dans le répertoire `standalone/bootloader_2nd_stage` devrait suffire. Puis mise sous tension de la carte, et envoi d'un caractère sur la ligne série SCI pour passer le bootloader en ROM en mode réception de fichier. On envoie alors le binaire sur la deuxième ligne série (SCIF), et le prompt du bootloader de niveau 2 devrait apparaître :

```
Starting Flash writer utility
```

```
>
```

Le programme attend alors des commandes sur la ligne série SCI. "h" affiche l'aide.

```
> h
```

```
h :      display this help
r :      read image from serial port (raw binary)
f [n] :  flash previously downloaded image to partition n
s :      scan flash for bad blocks
e [n] :          erase (and format) partition n
z [n] :          erase (and format) partition n, forcing bad blocks to be
erased
>
```

Réception d'une image

La commande "r" lance la réception d'une image (kernel ou image de file-system) par la ligne série SCIF. Comme pour le bootloader en ROM, aucun protocole particulier n'est à utiliser, on envoie juste le binaire tel quel (avec contrôle de flux matériel). A la fin de la réception, un checksum est calculé. C'est la somme sur 32 bits de tous les octets du fichier reçu.

Le fichier reçu est stocké en SDRAM CS2, à l'adresse 0XA8000000.

Effacement d'une partition

Avant toute écriture de partition, il est conseillé de l'effacer. Pour cela, utiliser la commande "e" suivie d'un espace et du numéro de partition (0 pour le kernel, 1 pour le root file system). Le

programme commence par scanner la flash pour détecter les blocs marqués défectueux (si cela n'a pas déjà été effectué), et n'efface que les blocs valides.

Certaines cartes, plus particulièrement celles manipulées par V. Palatin et E. Sandre, ont utilisées avec un driver flash NAND buggé. Ce driver se trompait dans l'écriture des octets OOB, et marquait comme invalide tous les blocs écrits. Pour remettre ces flashs là dans un état correct, on peut forcer l'effacement de *tous* les blocs, même ceux marqués défectueux, à l'aide de la commande "z". Il faut noter que cette commande est dangeureuse : les blocs défectueux seront marqués valides, et cela peut causer des problèmes par la suite sur la partition 0. La partition 1 est gérée de façon complète par YAFFS, qui vérifie chaque écriture. Un bloc défectueux sera donc détecté au vol, et marqué invalide.

Flashage d'un noyau

Une fois le noyau Linux compilé (`make` dans le répertoire `linux-2.6.10`), il faut en générer une image contenant les OOB. Cela est fait par un `make clean && make` dans le répertoire `shix-linux`, qui appelle l'utilitaire `mkZimage`. L'image ainsi générée peut être envoyée au boot-loader de niveau 2 par la commande "r" (cf. ci-dessus), puis flashée sur la partition 0, par la commande "f 0".

Flashage d'un root file system

le flashage d'un root file system est similaire à celui d'un noyau : `make clean && make` dans le répertoire `shix-linux`. Cette commande appelle l'utilitaire `mkyaffsimage`, qui prend en argument le top d'un root file system, et en produit une image prête à être flashée sur une flash NAND. Puis "r" et "f 1", pour flasher cette image en partition 1.

Scan des bad blocks

Le scan des bad blocks peut être effectué manuellement, ou automatiquement avant toute opération d'écriture s'il n'a pas déjà été effectué. Il est lancé par la commande "s". A la fin, une liste des blocs défectueux est affichée. Cette liste est vide si tous les blocs sont valides.

2.4 Linux