Ecole Nationale Supérieure d'Électronique, Informatique,
Télécommunications, Mathématique et Mécanique de Bordeaux

[PR310] Codesign sur carte Intel DE10-Standard avec FPGA Cyclone V

Compte Rendu

Objectifs : Ce projet a pour but de travailler sur la carte Intel DE10-Standard avec FPGA Cyclone V. Dans un premier temps, notre objectif est de découvrir le fonctionnement de la sortie VGA de la carte grâce aux logiciels Quartue et Eclipse. Ensuite, nous aurons comme nouvel objectif de créer une distribution Linux sur la carte nous permettant d'interagir avec elle à partir de son invité de commande est de pouvoir lancer nos programmes à partir de ce terminal.

Mots clés : FPGA, Système vidéo VGA, Quartus, Eclipse, Distribution Linux

Encadrants : Patrice KADIONIK (ENSEIRB-MATMECA)

Auteur : VIGNERON Émile & LEFEBVRE Louis & Ben Yahia Yosr& Pallaro Lucas

Table des matières

I Préambule

I.1 Présentation du projet

Ce rapport est structuré en deux parties distinctes, chacune explorant des aspects clés du projet. La première section s'attache à détailler la mise en place d'un système vidéo en exploitant le port VGA de la carte. Cette partie du rapport abordera les étapes spécifiques nécessaires à l'initialisation et à la configuration du système pour permettre une sortie vidéo via le port VGA. Des considérations telles que la résolution d'écran, la synchronisation, et les différentes composantes matérielles et logicielles impliquées dans la gestion de la vidéo seront examinées en détail.

La deuxième partie du rapport se focalise sur un volet tout aussi crucial du projet, à savoir l'implémentation de la distribution Linux sur le FPGA. Cette section présentera les différentes étapes nécessaires à l'intégration et à la configuration d'une distribution Linux adaptée aux spécificités du FPGA. Des considérations relatives à la compatibilité matérielle, la configuration du noyau Linux, ainsi que l'accès et la gestion des ressources matérielles spécifiques au FPGA seront discutées de manière approfondie.

I.2 Carte cible Intel DE10-Standard

La carte cible mise en œuvre pour le SoPC sur circuit FPGA Cyclone V d'Intel est une carte d'évaluation Intel (société Terasic) DE10-Standard.

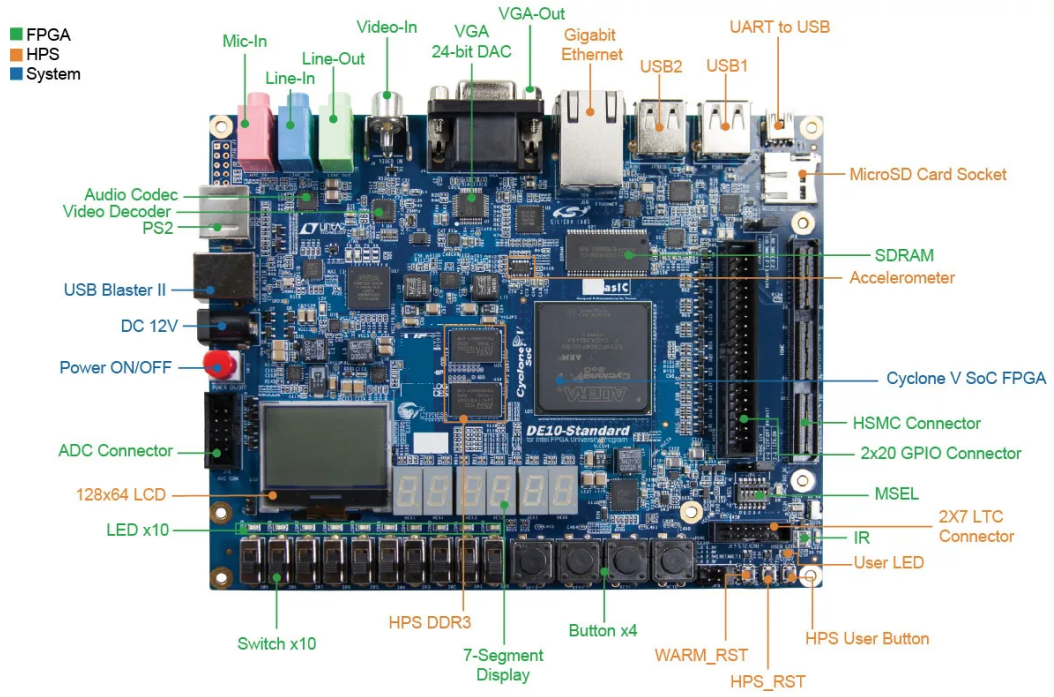


FIGURE 1 – Carte cible Intel DE10-Standard

La carte DE10-Standard intègre un circuit FPGA Cyclone V qui incorpore un processeur hardcore ARM dans la partie HPS (Hard Processor System) et une zone de programmation logique PL (Programmable Logic). Cette approche duale existe aussi chez Xilinx avec son circuit FPGA Zynq.

La figure suivante présente l'ensemble des périphériques accessibles sur la carte cible DE10-Standard. Cela correspond à un design de référence d'un système SoPC appelé Computer System par Intel mettant en œuvre deux processeurs softcore NIOS II et le processeur ARM.

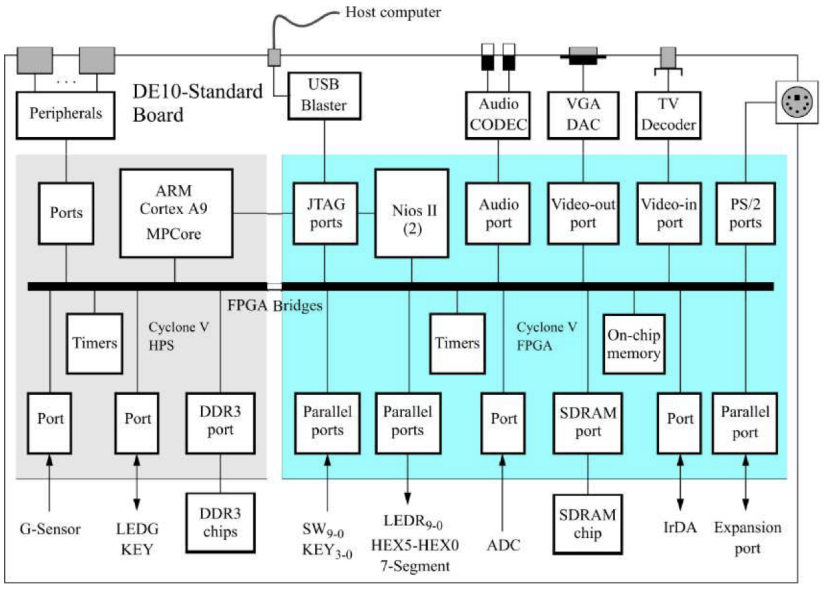


FIGURE 2 – Design de référence Computer System pour la carte cible Intel DE10-Standard

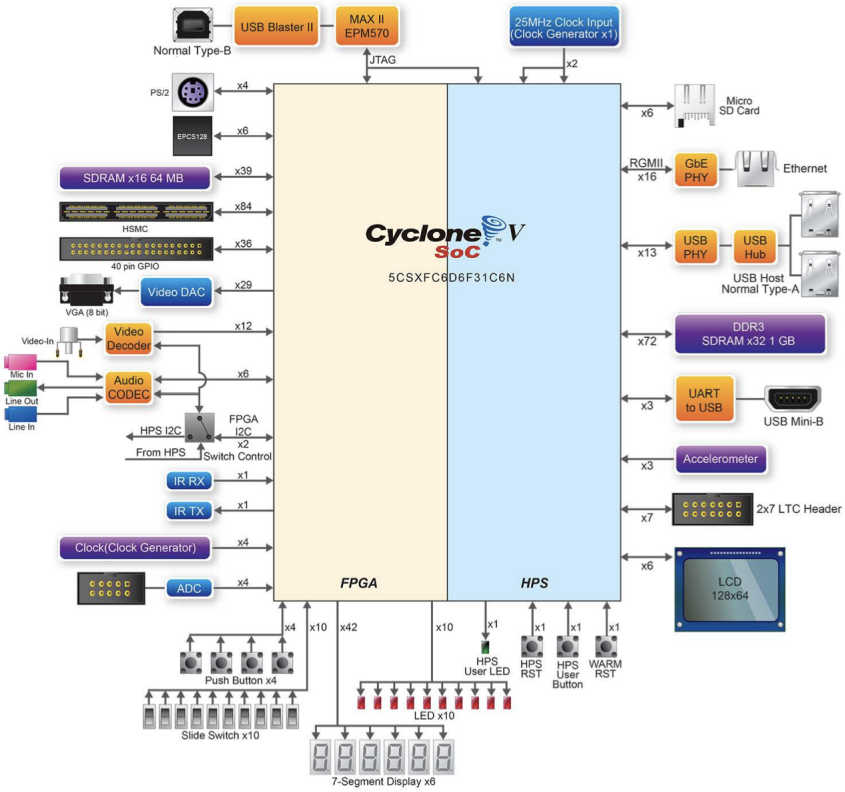


FIGURE 3 – Périphériques de la carte cible Intel DE10-Standard

Le processeur NIOS II (2ème génération du processeur NIOS) est un processeur RISC softcore entièrement synchrone, son architecture interne étant de type Harvard. Il possède au maximum 6 niveaux de pipeline, cadencé à quelques dizaines de MHz, avec une largeur de bus de 32 bits. Ses performances vont jusqu'à 250 MIPS (Million Instructions per Second).

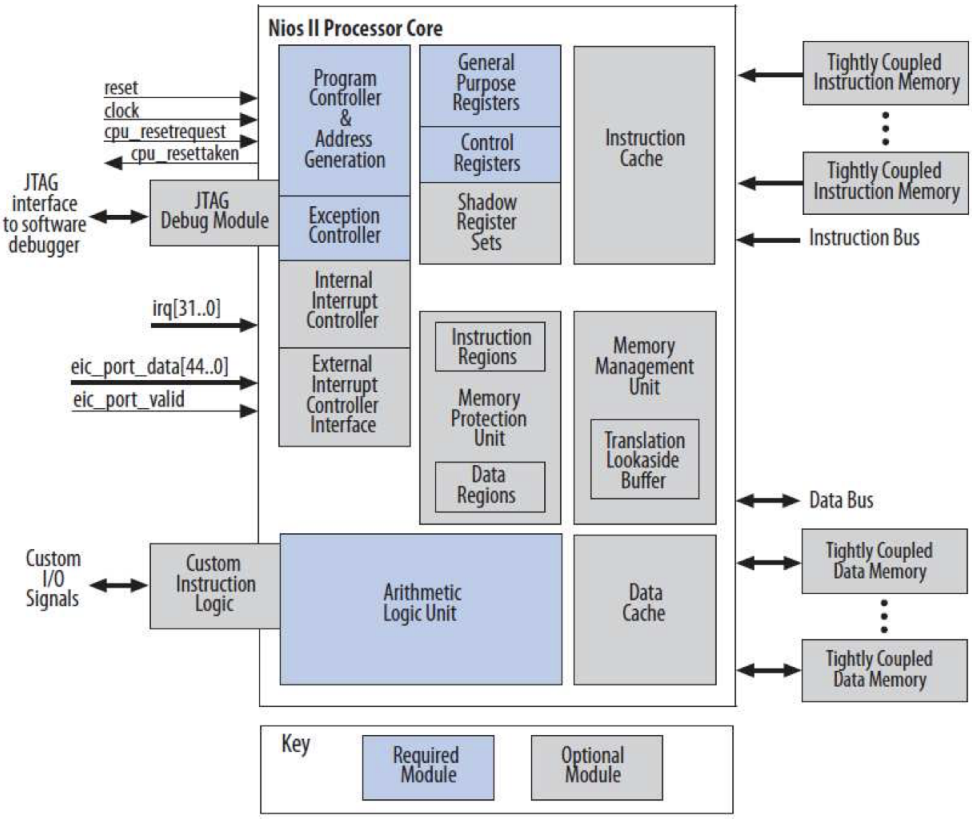


Figure 4 – Architecture du processeur NIOS II

II Mise place d'un système vidéo

II.1 Introduction

La mise en œuvre d'un système vidéo sur la carte DE10-Standard constitue un bon exemple afin d'exploiter au maximum les capacités du FPGA Intel Cyclone V embarqué. Cette section présente une approche progressive, débutant par la prise en main du logiciel Quartus, qui sert de plateforme de configuration du softcore de la carte. Ensuite, nous explorons en détail la configuration spécifique requise pour établir un système vidéo fonctionnel et développons la programmation d'une librairie dédiée au contrôle de l'écran.

II.2 Quartus : premier pas

Construction du design de référence

Pour commencer notre projet, il nous a fallu nous familiariser avec l'environnement de développement, notamment *Quartus Prime* et *Platform Designer*, afin de construire le design de référence.

La construction du design s'effectue en plusieurs étapes. Tout d'abord, nous avons intégré le processeur NIOS II et ses périphériques. À partir du catalogue IP, nous ajoutons les éléments suivants : pll, nios2, sdram, timer, jtag-uart, bp, switchs, leds et seg7.

Ensuite, nous avons établi les connexions pour les signaux d'horloge, de reset, de bus de données, ainsi que les bus d'instructions reliés à la mémoire SDRAM.

Par la suite, nous avons incorporé les interruptions pour les blocs IP timer, jtag-uart et bp. Il est essentiel de spécifier leur numéro d'interruption respectif, de 0 à 31.

De plus, nous avons défini le mappage mémoire des périphériques du système pour éviter tout chevauchement de mémoire. Nous avons attribu l'espace mémoire à la SDRAM à partir de l'adresse 0x0000_0000. Ensuite, nous avons défini les vecteurs d'exception du processeur NIOS II, qui sont les suivants :

— Le vecteur de reset qui indique le point d'entrée du programme lors du reset du processeur.

— Le vecteur d'interruption en cas de survenue d'une interruption.

Ces deux vecteurs sont placés dans la mémoire SDRAM aux adresses respectives 0x0000_0000 et 0x0000_0020.

Une exportation des signaux externes au circuit FPGA est nécessaire, car ces signaux sont connectés à des périphériques externes tels que la mémoire SDRAM, les boutons-poussoirs, les interrupteurs, les LED et les afficheurs 7 segments.

Après cela, nous avons procédé à la génération HDL et à la synthèse du système. Dans l'outil Quartus Prime, nous ajoutons le fichier nios2.qip afin d'établir la liaison entre les signaux exportés du système SoPC et les signaux externes au niveau Top-Level. Nous avons incorporé le contenu du fichier *nios2_inst.v* dans le fichier principal, qui a été ensuite modifié pour connecter les signaux externes du système SoPC aux signaux externes du circuit FPGA.

La dernière étape de cette conception consiste à programmer le circuit FPGA. À partir de Quartus Prime, nous avons chargé le programme dans le FPGA Cyclone V de la carte DE10 Standard en sélectionnant le composant 5CSXFC6D6 et en lui attribuant le fichier .sof.

À ce stade, la phase matérielle de la création du système est achevée. Nous sommes passé donc à la programmation du circuit FPGA depuis Eclipse.

Pour valider notre conception, nous avons testé l'application "Hello World !" de manière à constater le bon fonctionnement de notre programme.

II.3 Configuration du système vidéo

Pour établir le design softcore de notre carte, nous devons commencer par établir les différents blocs nécessaires pour le bon fonctionnement de notre système vidéo.

Dans un premier temps, nous commençons par les blocs principaux du système :

1. **SDRAM** : qui est un type de mémoire vive utilisée dans les systèmes informatiques pour stocker des données de manière temporaire. Voici quelques-unes des fonctions principales de la SDRAM dans un système :
   - Stockage temporaire de données
   - Exécution de programmes
   - Stockage des variables et des données temporaires
   - Manipulation de données en temps réel
   - Buffer pour les transferts de données
   - Support du multitâche
2. **JTAG** : qui est une interface standard qui joue un rôle crucial dans la programmation, le débogage et la configuration des composants électroniques, notamment des FPGA tels que ceux présents sur la carte DE10-Standard. Voici certaines fonctions principales du bloc JTAG dans un système :
   - Programmation du FPGA
   - Débogage du FPGA
   - Configuration et Test des Périphériques
   - Conception et Développement de Matériel
3. **PLL Système** : qui est un circuit utilisé pour générer une horloge stable et précise dans un système électronique. Les PLL sont essentielles pour synchroniser les opérations du processeur et des périphériques, assurant une horloge cohérente et synchronisée dans diverses applications, notamment les FPGA comme ceux présents sur la carte DE10-Standard.

Une fois l'ensemble des blocs principaux du système ajouté, il nous faut ajouter le reste des blocs permettant de contrôler le port VGA. Pour cela, nous sommes partis d'un exemple de design d'Intel, ce qui a grandement simplifié la configuration de chaque bloc. Nous avons donc trois nouvelles parties :

1. **PLL VGA** : de la même manière que pour le système, permet de générer une horloge stable et précise. Elle est essentielle pour synchroniser le transfert des données vers le port VGA et l'affichage sur l'écran.
2. **VGA Subsystème** : qui est une composante prédéfinie qui simplifie la mise en place d'un système vidéo en intégrant les éléments nécessaires pour gérer une interface VGA. Ce bloc offre plusieurs avantages dans un projet vidéo :
   - Génération Automatique des Signaux VGA : Le VGA Subsystem génère automatiquement les signaux essentiels tels que HSYNC (Horizontal Sync), VSYNC (Vertical Sync), les signaux de couleur (RGB), etc., nécessaires pour créer une sortie VGA standard.
   - Support de Résolutions Variées : Le bloc VGA Subsystem peut être configuré pour prendre en charge différentes résolutions vidéo, offrant ainsi une flexibilité pour s'adapter aux besoins spécifiques du projet.

- Réduction de la Complexité du Design
- Facilité d'Intégration
- Accélération du Processus de Conception

3. **VGA Communication** :
   - **Le VGA Pixel DMA** : facilite le transfert efficace des données graphiques depuis la mémoire vers le contrôleur VGA. Il optimise la gestion des pixels en permettant des transferts directs, réduisant ainsi la charge sur le processeur principal et améliorant les performances graphiques.
   - **Le Char Buf Systeme** : gère le stockage des caractères à afficher à l'écran dans une mémoire tampon. Il facilite l'affichage de texte en stockant les données de manière organisée, permettant au VGA Controller de lire les caractères et de les afficher à l'écran avec les propriétés de format appropriées.

Une fois que tous les blocs essentiels ont été mis en place et configurés, le processus peut progresser vers la phase de la synthèse du design. À cette étape, l'ensemble de la conception matérielle est soumis à une évaluation approfondie afin de créer un design cohérent et fonctionnel.

La synthèse englobe la transformation des descriptions matérielles abstraites en une netlist, qui est une représentation basée sur des composants logiques prêts pour l'implémentation physique sur le FPGA. Cette étape cruciale s'appuie sur les configurations précédemment établies, intégrant les spécifications matérielles définies pour chaque bloc.

Une fois la synthèse réussie, l'attention se tourne vers le démarrage du programme. Ce programme permettra d'afficher à l'écran non seulement des caractères, mais aussi des formes graphiques. Il communique avec les blocs préalablement configurés, initiants les séquences appropriées pour générer les signaux VGA, contrôler les pixels, et mettre en œuvre les fonctionnalités graphiques souhaitées.

II.4 Bibliothèque VGA

II.5 Ecriture de la bibliothèque VGA

En nous appuyant sur le système softcore et le design matériel que nous avons créé, nous avons ensuite entrepris le développement de notre propre bibliothèque VGA. Cette bibliothèque propose un ensemble de fonctions et de routines spécialement conçues pour simplifier la programmation de l'affichage VGA (Video Graphics Array) dans le cadre du développement matériel et logiciel. L'utilisation de cette bibliothèque VGA facilite la création d'applications graphiques destinées à des applications nécessitant une sortie vidéo. Dans notre cas, nous avons testé la capacité de la bibliothèque à écrire des caractères et à dessiner des formes, notamment un rectangle ou une boîte pleine.

Description du fonctionnement de la bibliothèque

Nous avons créé un projet comprenant cinq fichiers principaux :

- **vgalib.h :** Ce fichier d'en-tête déclare les prototypes de fonctions, les constantes et les structures nécessaires pour utiliser la bibliothèque VGA.

- **vgalib.c :** Il s'agit du fichier source en langage C contenant l'implémentation des fonctions de la bibliothèque VGA, notamment les fonctions VgaDrawBox, VgaWriteCharacter, VgaSetCursor, VgaClearScreen et VgaInit.
- **rom8x8.c :** Ce fichier source contient des données ou du code liés à une police de caractères 8x8.
- **adress_map_arm.h :** Ce fichier d'en-tête définit la carte mémoire (adressage) pour une application s'exécutant sur une architecture ARM.
- **main.c :** Ce fichier source principal contient la fonction main() du programme.

Dans le fichier vgalib.c, nous détaillons les fonctions citées ci-dessus.

Fonction VgaDrawBox

La fonction VgaDrawBox dessine un rectangle sur un écran graphique en remplissant chaque pixel à l'intérieur du rectangle avec une couleur spécifiée (pixel_color). La position mémoire correspondant à chaque pixel dans le rectangle est calculée à l'aide de la formule donnée dans le code ci-dessous. Les coordonnées du coin supérieur gauche et du coin inférieur droit du rectangle sont définies par les paramètres x1, y1, x2, et y2.

```
void VgaDrawBox(int x1, int y1, int x2, int y2, short pixel_color) {
  short row, col;

  for (row = y1; row < y2; row++) {
    for (col = x1; col < x2; col++) {
      *(mem_buffer + (row * LINE_ALIGNMENT) + col) = pixel_color;
    }
  }
}
```

Fonction VgaWriteCharacter

La fonction VgaWriteCharacter écrit un caractère graphique sur un écran VGA à la position du curseur actuelle en utilisant une table de bits pour représenter le motif du caractère. La couleur des pixels est déterminée en fonction des bits de la table, et elle est modifiée dans le tampon mémoire graphique en conséquence.

```
void VgaWriteCharacter(char c) {
  /**
   * cur_y * LINE_ALIGNMENT car lignes sont alignees
   */
  short *pixel_buf_ptr = mem_buffer + cur_x + (cur_y * LINE_ALIGNMENT);

  short *pixel_ptr;
  short row, col;

  for (row = 0; row < 8; row++) {
    short row_color = rom8x8_bits[(c * 8) + row];
    for (col = 0; col < 8; col++) {
      pixel_ptr = pixel_buf_ptr + (row * LINE_ALIGNMENT) + col;
      if ((row_color >> (15 - col)) & 1) {
        *pixel_ptr = 0xFFFF;
      }
      else {
        *pixel_ptr = 0;
      }
    }
```

```
  }
}
```

Fonction VgaSetCursor

La fonction VgaSetCursor ajuste les coordonnées du curseur pour qu'elles restent à l'intérieur des limites de l'écran VGA, puis met à jour la position du curseur.

```
void VgaSetCursor(int col, int row) {
    if(row > SIZE_Y - 1) row = SIZE_Y - 1;
    if(col > SIZE_X - 1) col = SIZE_X - 1;
    if(row < 0) row = 0;
    if(col < 0) col = 0;
    cur_y = row;
    cur_x = col;
}
```

Fonction VgaClearScreen

La fonction VgaClearScreen utilise la fonction 'VgaDrawBox' pour dessiner un rectangle rempli de couleur noire (0) sur tout l'écran VGA, effaçant ainsi le contenu affiché à l'écran. La taille du rectangle est déterminée par les constantes 'SIZE_X' et 'SIZE_Y'.

```
void VgaClearScreen(void) {
  VgaDrawBox(0, 0, SIZE_X, SIZE_Y, 0);
}
```

Fonction VgaInit

La fonction VgaInit initialise la bibliothèque VGA en associant le pointeur video_buf à la variable globale mem_buffer. De plus, elle positionne le curseur à la première ligne (cur_y = 0) et à la première colonne (cur_x = 0).

```
void VgaInit(short *video_buf) {
  mem_buffer = video_buf;
  cur_y = 0;
  cur_x = 0;
}
```

Dans le fichier main.c, nous avons rédigé le code permettant d'insérer le caractère 'a' au début de la ligne et de dessiner un rectangle bleu. Lors de la première phase de test, nous avons exécuté ce code en mode Bare-Metal, et nous avons constaté qu'il fonctionne de manière optimale, ne dépendant pas d'un système d'exploitation.

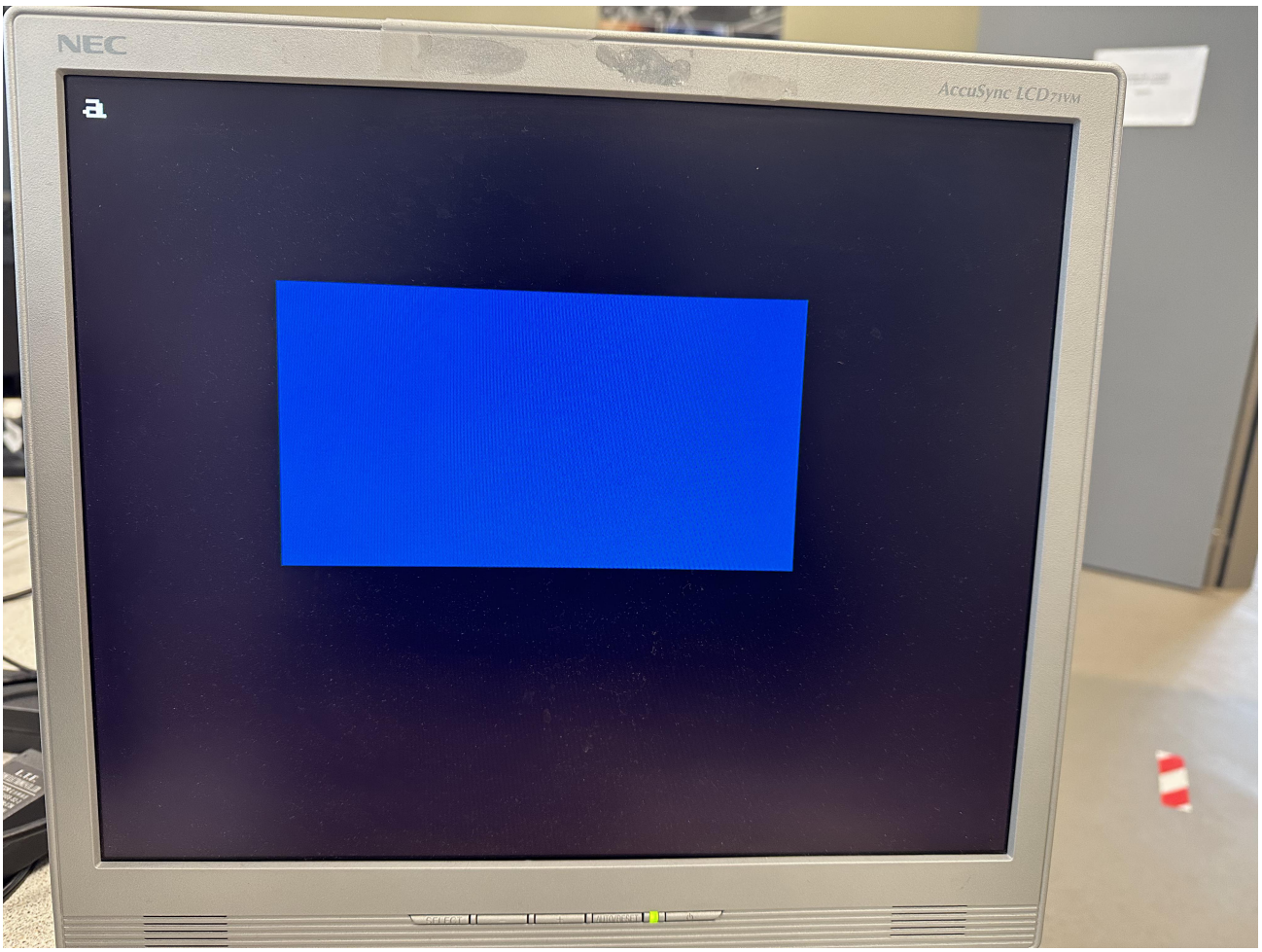


FIGURE 5 – Résultat visualisé sur l'écran

III Mise place d'un système Linux

III.1 Introduction

La mise en place d'un système Linux sur la carte DE10-Standard constitue une étape cruciale pour exploiter les performances avancées du FPGA embarqué. Cette démarche complexe s'articule autour de trois phases distinctes, chacune jouant un rôle essentiel dans l'intégration du système d'exploitation Linux au sein de l'environnement FPGA de la carte.

Alors que notre application est spécifiquement conçue pour interagir avec un écran VGA connecté à notre carte Intel DE10-Standard avec FPGA Cyclone V, le système Linux offre une gestion avancée des ressources matérielles, permettant une utilisation plus efficace du FPGA Cyclone V et des périphériques associés. La mise en place d'un système Linux ouvre également la porte à une flexibilité accrue. Nous pouvons ainsi tirer parti d'un écosystème logiciel robuste, facilitant le développement, la maintenance et l'optimisation continue de notre application.

Enfin, cette démarche élargit considérablement les possibilités de communication avec d'autres systèmes ou périphériques, créant ainsi un terrain propice à l'exploration de fonctionnalités futures et à une intégration plus poussée.

Dans les sections à venir, nous explorerons en détail le processus de conception du système Linux, ainsi que son intégration réussie avec notre application dédiée à l'affichage VGA.

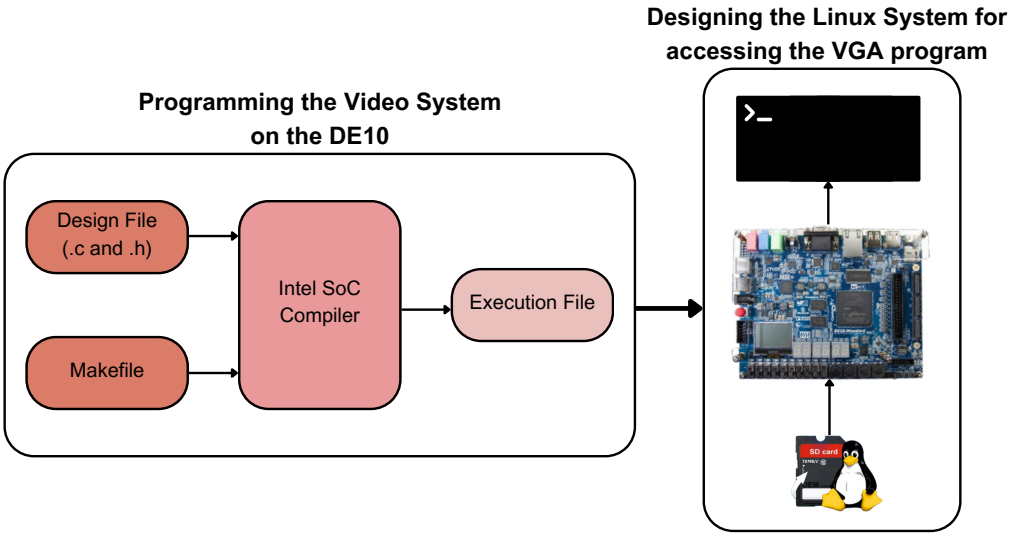


FIGURE 6 – Répartition du projet

III.2 Conception du système Linux au niveau matériel

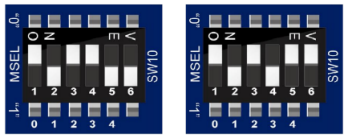
État des Lieux de la Carte et Prérequis pour le Système Linux

Avant de plonger dans la conception du système Linux, une évaluation approfondie de la carte Intel DE10-Standard s'impose. La Figure 1 illustre la disposition générale de la carte et met en évidence les éléments cruciaux pour la mise en place réussie du système Linux.

Port JTAG : Ce port jouent le rôle essentiel dans le processus de développement et de débogage. Il permet la programmation du FPGA Cyclone V, une compréhension approfondie est cruciale pour assurer une interaction efficace avec la carte pendant le développement du système Linux.

Micro SD Card Socket : La présence d'un socket pour une carte Micro SD revêt une importance particulière, car c'est sur cette carte que le système Linux sera stocké et exécuté. L'utilisation judicieuse de la carte Micro SD est donc une étape clé dans le processus d'amorçage du système d'exploitation.

Configuration MSEL pour le Boot via SD Card : Un élément critique à prendre en compte est la configuration du MSEL (Master Select) switch. Ce commutateur doit être positionné correctement (voir figure 7) pour permettre au FPGA d'être programmé depuis U-Boot via la carte SD. La vérification et le réglage approprié de cette configuration sont indispensables pour garantir le bon amorçage du système Linux.



Mode FPGA Programmer Mode SD card Boot

FIGURE 7 – Configuration du MSEL

SD Card Flash and U-Boot

Avant d'entrer dans le détail des étapes du processus de démarrage, la préparation de la carte SD est une phase préliminaire cruciale. Il s'agit de flasher la carte SD avec une image de distribution Linux. Cela consiste à flasher la carte SD avec une image de distribution Linux adaptée au FPGA Cyclone V de la carte Intel DE10-Standard. Cette image comprend le noyau Linux, le système de fichiers racine, et d'autres composants essentiels.

Certaines distributions sont optimisées pour les systèmes embarqués et offrent une compatibilité étroite avec le matériel de la carte DE10-Standard. Nous avons choisi une distribution compatible avec le FPGA Cyclone V et l'architecture matérielle spécifique comprenant la gestion correcte des périphériques telle que l'écran VGA.

Nous détaillons maintenant les étapes spécifiques du démarrage du système Linux.

Reset : La première étape du processus de démarrage consiste en une opération de reset. Cette étape initialise l'ensemble du système, mettant tous les composants dans un état défini. C'est le point de départ de la séquence de démarrage.

SD Card Boot : La carte Micro SD, correctement préparée avec le système Linux (flashée antérieurement avec la distribution adaptée), est insérée dans le socket. Le FPGA est configuré pour amorcer le système depuis la carte SD selon la configuration du MSEL. Cette étape est cruciale pour le chargement initial du système d'exploitation.

Preloader : Le preloader intervient comme une étape intermédiaire entre le boot depuis la carte SD et le chargement d'U-Boot. Il assure l'initialisation de certains paramètres et prépare le FPGA pour la phase suivante du démarrage.

U-Boot : U-Boot, ou Universal Bootloader, est une étape centrale du processus. Il charge le noyau Linux depuis la carte SD vers la mémoire du FPGA. U-Boot offre également une interface en ligne de commande pour une interaction pendant le démarrage.

Linux : Enfin, une fois que U-Boot a réussi à charger le noyau Linux, le système d'exploitation s'initialise. Cette étape marque la transition complète vers l'environnement Linux, où l'application dédiée à l'affichage VGA sera exécutée.

III.3 Conception du système Linux au niveau logiciel

IV Concevoir le système Linux au niveau logiciel

IV.1 Introduction

Dans cette partie, l'architecture logicielle du système est décrite ; elle comprend :

— le bootloader U-Boot ;
— un *Device Tree Blob* (DTB) qui décrit le mappage mémoire et le matériel présent sur le SoC ;
— le noyau Linux ;
— *un root file system* (rootfs) qui est utilisé comme racine du système de fichiers par Linux.

IV.2 Le bootloader U-Boot

U-Boot est utilisé pour :

1. télécharger le noyau Linux et le DTB depuis le PC hôte vers la mémoire RAM ; programmer le fpga ;
2. charger en mémoire et démarrer le noyau Linux.

U-Boot lance automatiquement un script au démarrage qui est contenu dans sa variable d'environnement bootcmd.

```
bootcmd=run gofpga\; run gok \; run gor \; run godtb \; run go
```

Ce script lance les scripts suivants :

— gofpga : Programme le fpga à partir du fichier soc_system.rbf sur la partition 1 de la carte sd.

— gok : Télécharge le noyau en mémoire.
— gor : Télécharge le ramdisk en mémoire mais ce dernier n'est pas utilisé car nous n'avons pas réussi à le faire fonctionner.
— godtb : Télécharge le DTB en mémoire.
— go : Démarre le noyau avec les adresses du noyau et du DTB en argument.

Les scripts sont les suivants :

```
gofpga=fatload mmc 0:1 $fpgadata soc_system.rbf\; fpga load 0 $fpgadata $filesize\; \
run bridge_enable_handoff
gok=tftpboot \$\{kernel_addr_r\} zImage
gor=tftpboot \$\{ramdisk_addr_r\} uramdisk.image.gz
godtb=tftpboot \$\{fdt_addr_r\} socfpga.dtb
go=bootz \${kernel_addr_r} - \${fdt_addr_r}
```

Le script pour configurer U-Boot est donné dans les livrables dans `/etc/u-boot.txt`.

IV.3 Le noyau Linux

Le noyau Linux a été compilé avec le cross-compilateur buildroot. Le fichier de configuration est donné dans les livrables dans `/linux/.config`. Cette configuration est celle de `linux/arch/arm/configs/socfpga_defconfig` dans les sources du noyau.

IV.4 Le rootfs

Le rootfs utlisé est exactement le même que celui utilisé dans le TP Zedboard. Le rootfs est monté par Linux depuis la partition 2 de la carte sd car nous n'avons pas réussi à le faire passer comme argument du noyau à son démarrage.

IV.5 Le device tree blob

Le DTB a été récupéré depuis l'image disponible à `github.com/zangman/de10-nano`. Étant fonctionnel, nous l'avons gardé tel quel.

IV.6 Téléchargement des fichiers

Les fichiers téléchargés le sont via tftp. Cela accélère le développement en évitant de sans cesse brancher/débrancher la carte sd.

IV.7 Test Final de l'Application sur Linux en tant que User-Space Driver

Avant d'entrer dans les détails du fichier de test *test_vgalib_linux.c*, examinons brièvement les mémoires partagées qui seront utilisées dans le cadre de ce test.

Approche de l'affichage VGA : L'approche de l'affichage VGA constitue la méthode par laquelle notre application interagit avec l'écran VGA. Cela implique l'utilisation de mémoires partagées pour définir les pixels et les couleurs affichés à l'écran.

Adressage des Registres VGA : L'adressage des registres VGA est crucial pour accéder aux paramètres spécifiques de l'écran, tels que la position du curseur, la couleur de fond, etc. Ces registres ont été mappés en mémoire pour une manipulation facile.

Memory Mapping : Le memory mapping est une technique utilisée pour lier la mémoire du programme avec les registres matériels comme détaillé dans la partie précédente. Cela garantit un accès rapide et direct aux ressources matérielles, en l'occurrence, les registres VGA.

Notre fichier de test*test_vgalib_linux.c* (Voir VI.1) illustre une utilisation concrète des mémoires partagées et des fonctions de la bibliothèque VGA pour afficher un carré bleu et un caractère 'a' sur l'écran. Il démontre le fonctionnement du pilote dans l'espace utilisateur pour interagir avec l'écran VGA. La particularité réside dans les fonctions d'accès aux mémoires physiques :

- *int open_physical(int fd)* : Ouvre /dev/mem pour accéder aux adresses physiques.
- *void close_physical(int fd)* : Ferme l'accès aux adresses physiques.
- *void *map_physical(int fd, unsigned int base, unsigned int span)* : Établit une correspondance entre les adresses physiques et virtuelles.
- *int unmap_physical(void *virtual_base, unsigned int span)* : Ferme la correspondance d'adresse virtuelle.

Ensuite dans la fonction principale, nous suivons ces étapes :

1. Établit une connexion aux adresses physiques via /dev/mem.
2. Mappe la mémoire physique du FPGA pour un accès virtuel.
3. Initialise la bibliothèque VGA via VgaInit().
4. Efface l'écran (VgaClearScreen()), dessine un carré bleu (VgaDrawBox()), et écrit un caractère 'a' à la position spécifiée (VgaSetCursor() et VgaWriteCharacter()).
5. Ferme les accès physiques.

Ce test démontre la capacité du programme à interagir avec l'écran VGA de manière graphique en utilisant la bibliothèque VGA. Il offre un exemple concret d'utilisation de la mémoire partagée et des fonctions de la bibliothèque pour manipuler l'affichage à l'écran.

V Conclusion

À l'issue du projet, les répertoires suivants ont été remis :

```
-> de10-distro/
        -> design/
            Contient les projets Quartus Prime.
        -> etc/
            Contient les fichiers de configuration.
        -> linux/
            Contient les sources du noyau Linux.
        -> tst/
            Contient le programme de test de la bibliothèque
            vgalib sous Linux.
        -> ramdisk/
            Contient le nécessaire pour générer le ramdisk.
        -> vgalib/
            Contient la bibliothèque vga.
```

Les objectifs suivants ont été atteints :

— Écrire et tester la bibliothèque.

— Concevoir le système au niveau matériel via Quartus Prime.

— Concevoir le système au niveau logiciel : créer un ramdisk, compiler notre propre noyau, configurer U-Boot adéquatement.

Une chose que nous n'avons pas réussie à faire est de passer un ramdisk en argument de Linux à son démarrage.

VI Annexes

VI.1 Fichier C de test - *test_vgalib_linux.c*

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "address_map_arm.h"
#include "vgalib.h"

#define LW_BRIDGE_SPAN 0x2000u
#define FPGA_PIXEL_BUF_SPAN (FPGA_PIXEL_BUF_END - FPGA_PIXEL_BUF_BASE)

void * map_physical(int, unsigned int, unsigned int);
void close_physical(int);
int unmap_physical(void *, unsigned int);

/* Affiche un carre bleu et un caractere sur l'ecran. */
int main(void)
{
    volatile short * mem_ptr; // pointeur d'adresse virtuelle vers les LED rouges
    int fd = -1; // utilise pour ouvrir /dev/mem

    // Cree un acces memoire virtuel au light bridge du FPGA
    if ((fd = open_physical(fd)) == -1)
        return (-1);
    if (!(mem_ptr = map_physical(fd, FPGA_PIXEL_BUF_BASE, FPGA_PIXEL_BUF_SPAN)))
        return (-1);

    printf("VgaInit()\n");
    VgaInit(mem_ptr);

    short background_color = 0xF8;

    printf("VgaClearScreen()\n");
    VgaClearScreen();
    printf("VgaDrawBox()\n");
    VgaDrawBox(31 * 2, 28 * 2, 59 * 4 - 1, 36 * 4 - 1, background_color);

    printf("VgaSetCursor()\n");
    VgaSetCursor(4, 4);
    printf("VgaWriteCharacter()\n");
    VgaWriteCharacter('a');

    unmap_physical(mem_ptr, FPGA_PIXEL_BUF_SPAN);
    close_physical(fd);
    return 0;
}

/* Ouvre /dev/mem pour donner acces aux adresses physiques */
int open_physical(int fd)
{
    if (fd == -1) // verifie si deja ouvert
        if ((fd = open("/dev/mem", (O_RDWR | O_SYNC))) == -1)
        {
            printf("ERROR: could not open \"/dev/mem\"...\n");
            return (-1);
        }
    return fd;
```

```c
}

/* Ferme /dev/mem pour donner acces aux adresses physiques */
void close_physical(int fd)
{
    close(fd);
}

/* Etablit une correspondance d'adresse virtuelle pour les adresses physiques a
    partir de base et s'etendant sur span octets */
void *map_physical(int fd, unsigned int base, unsigned int span)
{
    void *virtual_base;

    // Obtenir une correspondance entre les adresses physiques et les adresses
    virtuelles
    virtual_base = mmap(NULL, span, (PROT_READ | PROT_WRITE), MAP_SHARED,
                        fd, base);
    if (virtual_base == MAP_FAILED)
    {
        printf("ERROR: mmap() failed...\n");
        close(fd);
        return (NULL);
    }
    return virtual_base;
}

/* Ferme la correspondance d'adresse virtuelle precedemment ouverte */
int unmap_physical(void *virtual_base, unsigned int span)
{
    if (munmap(virtual_base, span) != 0)
    {
        printf("ERROR: munmap() failed...\n");
        return (-1);
    }
    return 0;
}
```