



POLYTECH SORBONNE UNIVERSITÉ

ARCHITECTURE DES SES
PROJET
RAPPORT

Radar 2D

Élève(s) :
Matheus SISTON GALDINO

Enseignant(s) :
Yann DOUZE

Table des matières

1 Introduction	3
2 Télémètre ultrason HC-SR04	3
2.1 Principe de fonctionnement	3
2.2 IP Télémètre Standalone	3
2.2.1 Architecture et machine d'états	4
2.2.2 Principe de mesure et calcul de la distance	4
2.2.3 Simulation fonctionnelle	5
2.2.4 Test sur carte DE10-Lite	6
2.3 Intégration de l'IP télémètre avec l'interface Avalon	7
2.3.1 Principe de l'interface Avalon	7
2.3.2 Validation par simulation	8
2.3.3 Gestion de la fréquence d'horloge dans le système SoC	9
2.3.4 Validation expérimentale sur la carte DE10-Lite	9
3 Servomoteur	11
3.1 IP Servomoteur Standalone	11
3.1.1 Architecture de l'IP	11
3.1.2 Calibration expérimentale des limites angulaires	11
3.1.3 Relation entre consigne et angle	12
3.1.4 Simulation fonctionnelle	12
3.1.5 Validation expérimentale sur la carte DE10-Lite	12
3.2 Extension de l'IP Servomoteur vers le bus Avalon	14
3.2.1 Architecture de l'IP Servomoteur Avalon	14
3.2.2 Validation par simulation	15
3.3 Programmation logicielle et test de l'IP Servomoteur	15
3.3.1 Validation expérimentale	16
4 Affichage des obstacles	17
4.1 Fonctionnement global	18
4.2 Affichage et résultats expérimentaux	18
5 Affichage VGA – Radar 2D	19
5.1 Mise en place du sous-système VGA	20
5.1.1 Nettoyage des buffers à l'initialisation	20
5.2 Principe de la représentation radar	21
5.2.1 Choix géométriques et repère	21
5.2.2 Conversion polaire → cartésienne	21
5.2.3 Affichage statique : axes et arcs de distance	21
5.3 Affichage dynamique : balayage et coloration des mesures	22
5.3.1 Balayage angulaire et acquisition	22
5.3.2 Codage couleur des obstacles	22
5.3.3 Affichage simultané sur 7 segments et terminal	22
5.4 Résultats expérimentaux	22

6	UART	24
6.1	Principe de fonctionnement de l'UART	24
6.2	Architecture de l'IP UART	24
6.3	Simulation fonctionnelle de l'IP UART	25
6.4	Validation expérimentale sur la carte DE10-Lite	25
6.5	Extension de l'IP UART vers le bus Avalon	26
6.5.1	Architecture de l'IP UART Avalon	26
6.5.2	Validation par simulation de l'interface Avalon	27
6.5.3	Validation expérimentale avec le processeur Nios II	28
7	Radar – Intégration finale et commande via UART	29
7.1	Architecture logicielle : deux modes <i>CMD</i> et <i>RUN</i>	29
7.2	Interface de commande UART	29
7.2.1	Commandes disponibles	29
7.3	Difficultés rencontrées et solutions retenues	30
7.3.1	Blocage en attente UART et impossibilité de démarrer le radar	30
7.3.2	Arrêt du radar sans FIFO : choix d'un contrôle matériel robuste	30
7.3.3	Exemple de configuration et lancement	30
8	Conclusion	32
A	Programme Nios II Radar Complet	33

1 Introduction

Ce projet a pour objectif la conception et la réalisation d'un radar ultrason bidimensionnel basé sur une architecture SoC-FPGA, utilisant la carte *DE10-Lite* et le processeur *Nios II*. Le système combine un télémètre ultrason monté sur un servomoteur afin de réaliser un balayage angulaire sur 180°, avec acquisition de distances et visualisation des résultats sous différentes formes.

Les mesures sont exploitées à travers un affichage sur afficheurs 7 segments, une communication série UART et une représentation graphique de type radar 2D sur un écran VGA. Le projet met ainsi en œuvre plusieurs IP matérielles intégrées au bus Avalon, ainsi qu'un programme applicatif en langage C exécuté sur le processeur Nios II, illustrant les principes de co-conception matériel/logiciel.

Les codes sources logiciels (.c) sont fournis en annexe de ce rapport. L'ensemble du projet est également disponible sur le dépôt GitHub suivant :

<https://github.com/matgaldino/FPGA-Ultrasonic-2D-Radar>

2 Télémètre ultrason HC-SR04

Le télémètre ultrason HC-SR04 est un capteur de distance basé sur la propagation d'ondes ultrasonores dans l'air. Il permet de mesurer la distance séparant le capteur d'un obstacle en exploitant le temps de vol aller-retour d'une impulsion ultrasonore réfléchie par cet obstacle.

Le module fonctionne à une fréquence ultrasonore de 40 kHz et offre une plage de mesure typique comprise entre 2 cm et 400 cm. Ces caractéristiques en font un capteur adapté à la réalisation d'un système de cartographie bidimensionnelle de l'environnement, tel qu'un radar 2D.

2.1 Principe de fonctionnement

Le fonctionnement du HC-SR04 repose sur l'échange de deux signaux numériques principaux : TRIG et ECHO.

Pour initier une mesure, le système de contrôle applique une impulsion logique haute d'une durée minimale de 10 µs sur l'entrée TRIG. À la réception de cette impulsion, le capteur émet une salve de huit impulsions ultrasonores à 40 kHz. Ces ondes se propagent dans l'air jusqu'à rencontrer un obstacle, sur lequel elles sont partiellement réfléchies vers le capteur.

Lorsque l'écho de ces ondes est détecté, le capteur positionne la sortie ECHO à l'état haut. La durée pendant laquelle ECHO reste à l'état haut est directement proportionnelle à la distance séparant le capteur de l'obstacle. Plus la distance est grande, plus la durée du signal ECHO est longue.

Le principe de mesure repose donc sur l'évaluation précise du temps de propagation aller-retour de l'onde ultrasonore.

2.2 IP Télémètre Standalone

Avant l'intégration de l'IP dans un système SoPC basé sur le processeur Nios II, le télémètre ultrason a été implémenté et validé en mode *standalone*, c'est-à-dire en fonc-

tionnement autonome sur la partie FPGA de la carte DE10-Lite. Cette étape permet de vérifier le bon comportement fonctionnel de l'IP indépendamment de toute interface bus ou logicielle.

Dans cette configuration, l'IP est directement cadencé par l'horloge principale de la carte, à savoir 50 MHz. Ce choix simplifie l'analyse temporelle et garantit une résolution suffisante pour la mesure précise de la durée du signal ECHO. Il est à noter que cette fréquence pourra être amenée à évoluer lors de l'intégration future dans un système Avalon, notamment en fonction des contraintes globales du SoPC.

2.2.1 Architecture et machine d'états

Le fonctionnement de l'IP repose sur une machine d'états finis assurant l'enchaînement des différentes phases de mesure. Cette machine d'états comporte les états suivants :

- **IDLE** : état initial de préparation de la mesure ;
- **TRIG_PULSE** : génération de l'impulsion de déclenchement TRIG d'une durée de 10 µs ;
- **WAIT_ECHO** : attente du front montant du signal ECHO ;
- **MEASURE_ECHO** : comptage du nombre de cycles d'horloge pendant lesquels ECHO reste à l'état haut ;
- **WAIT_BETWEEN** : temporisation entre deux mesures successives, d'une durée d'environ 60 ms.

L'organigramme de cette machine d'états est présenté en Figure 1.

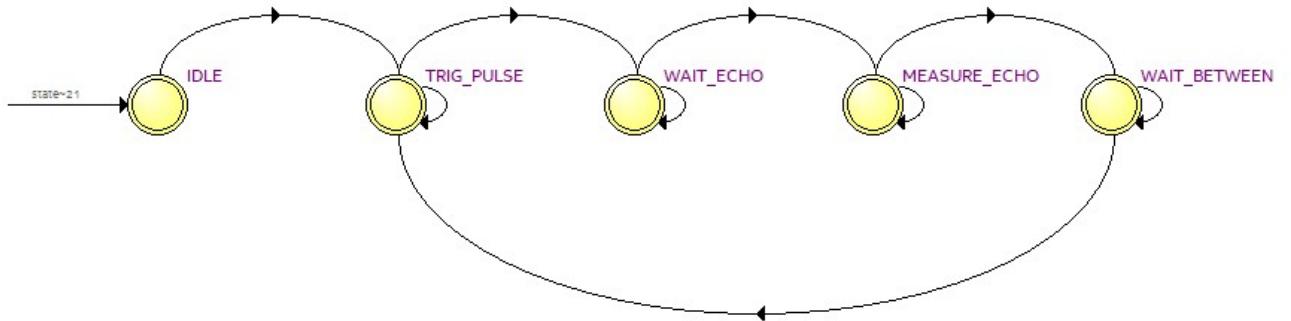


FIGURE 1 – Machine d'états de l'IP télémètre ultrason en mode standalone

2.2.2 Principe de mesure et calcul de la distance

La mesure de distance repose sur le comptage du nombre de cycles d'horloge pendant lesquels le signal ECHO est à l'état haut. La fréquence d'horloge du système n'est pas supposée fixe : elle est paramétrée dans l'IP à l'aide du *generic CLK_FREQ_HZ*, ce qui permet d'adapter automatiquement le calcul à la fréquence réelle du système SoC-FPGA.

Le capteur ultrason HC-SR04 fournit un signal ECHO dont la durée est proportionnelle au temps de vol aller-retour de l'onde ultrasonore. Une approximation couramment utilisée

pour ce capteur est :

$$1 \text{ cm} \approx 58 \mu\text{s}$$

À partir de cette relation, le nombre de cycles d'horloge correspondant à un centimètre est calculé dynamiquement en fonction de la fréquence du système :

$$N_{\text{cycles/cm}} = \frac{f_{\text{clk}} \times 58}{10^6}$$

où f_{clk} désigne la fréquence d'horloge en hertz. Cette valeur est évaluée à la compilation à partir du *generic CLK_FREQ_HZ* et inclut un arrondi afin de limiter les erreurs systématiques.

Soit N_{echo} le nombre de cycles comptés pendant l'état haut du signal ECHO. La distance en centimètres est alors obtenue par :

$$D_{\text{cm}} = \left\lfloor \frac{N_{\text{echo}}}{N_{\text{cycles/cm}}} \right\rfloor$$

où $\lfloor \cdot \rfloor$ représente un arrondi à l'entier le plus proche. Dans l'implémentation matérielle, cet arrondi est réalisé en ajoutant la moitié du dénominateur avant la division entière.

Enfin, lorsque la distance calculée dépasse la portée maximale utile du capteur, la valeur de sortie est forcée à **0**. Ce choix permet de signaler un résultat hors plage tout en conservant une largeur de sortie de 10 bits :

$$D_{\text{cm}} > 400 \Rightarrow \text{Dist_cm} = 0$$

2.2.3 Simulation fonctionnelle

La validation fonctionnelle de l'IP a été réalisée à l'aide du simulateur ModelSim. Un banc de test a été développé afin de générer des signaux ECHO synchronisés avec les impulsions TRIG émises par l'IP, reproduisant ainsi le comportement réel du capteur.

Plusieurs distances ont été simulées afin de vérifier la cohérence de la conversion : 20 cm, 100 cm et 350 cm. Un cas hors plage a également été testé (distance supérieure à 400 cm), permettant de valider le comportement de saturation choisi (`Dist_cm` forcé à 0).

La Figure 2 illustre les chronogrammes obtenus : la génération périodique de TRIG, les impulsions ECHO de durées différentes, ainsi que la mise à jour de la sortie `Dist_cm` à la fin de chaque mesure.

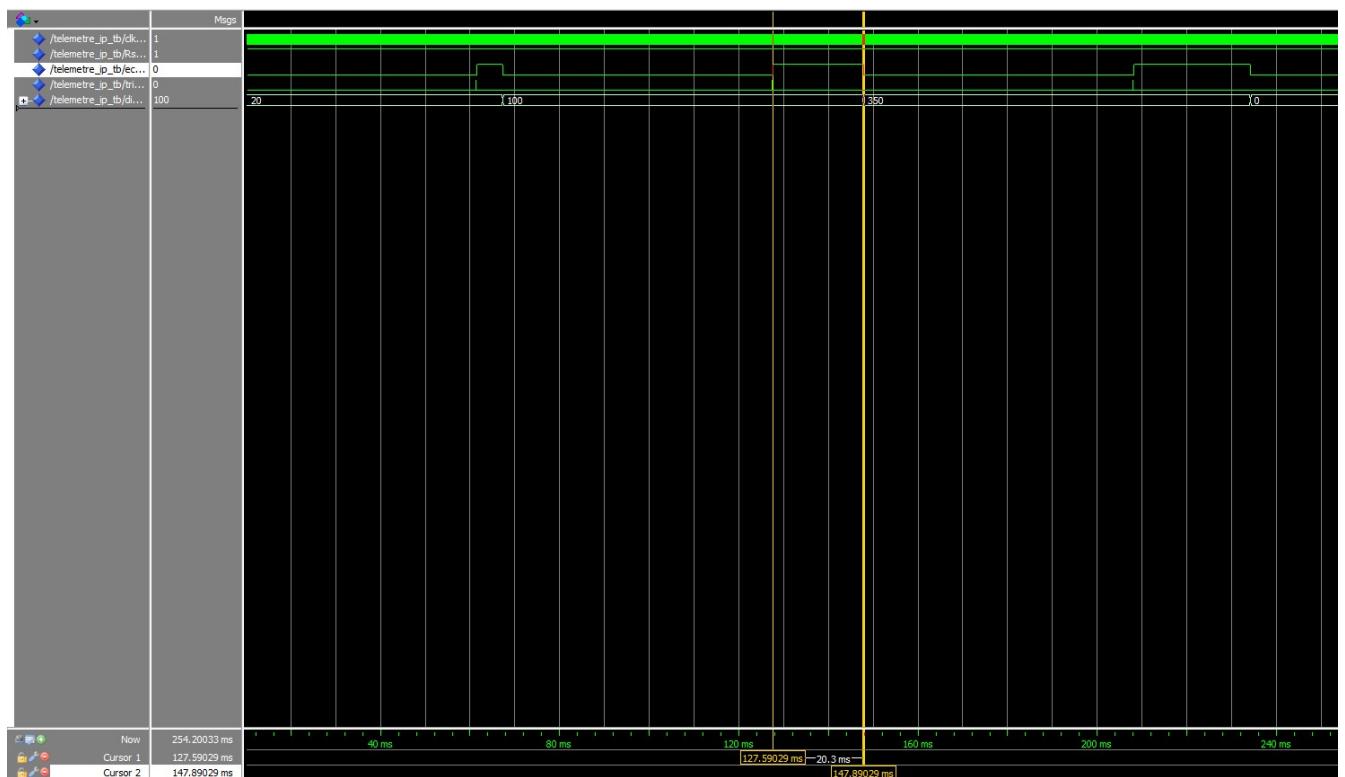


FIGURE 2 – Simulation ModelSim de l’IP télémètre en mode stand-alone (cas 20 cm, 100 cm, 350 cm et hors plage)

2.2.4 Test sur carte DE10-Lite

Après validation en simulation, l’IP a été implémenté sur la carte DE10-Lite afin de vérifier son fonctionnement en conditions réelles. Le télémètre HC-SR04 a été connecté aux broches GPIO de la carte, tandis que la sortie `Dist_cm` a été reliée aux LEDs rouges `LEDR[9:0]`.

Un test simple a été réalisé en plaçant un obstacle à environ 10 cm du capteur. La valeur mesurée est alors affichée sur les LEDs en codage binaire : pour 10 cm, le vecteur `LEDR[9:0]` représente 10_{10} , soit 0000001010 en binaire. La Figure 3 illustre ce test et la correspondance entre la distance réelle et l’affichage obtenu.

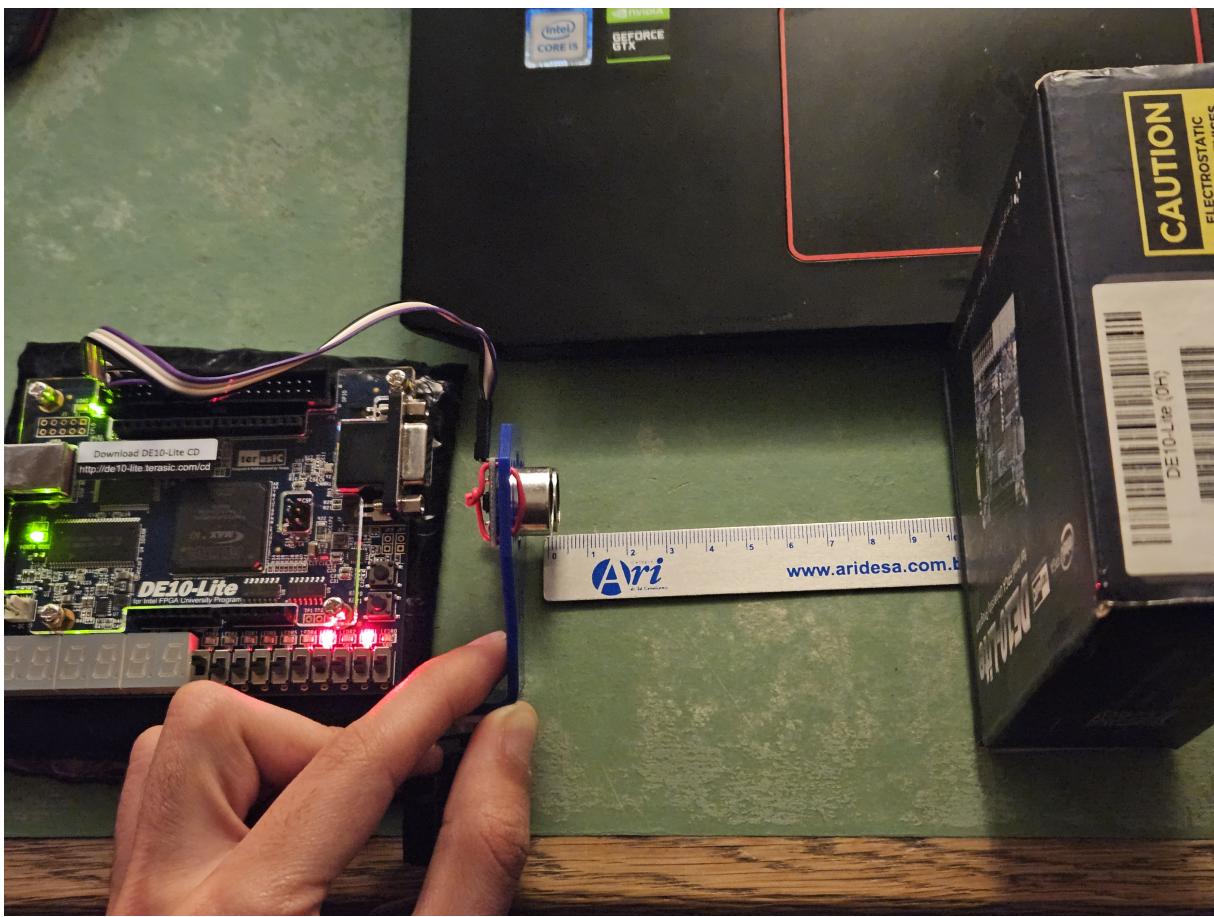


FIGURE 3 – Validation de l’IP télémètre sur carte DE10-Lite en mode standalone
(obstacle à 10 cm affiché sur LEDR[9:0])

Ces résultats confirment le bon fonctionnement de l’IP télémètre ultrason en mode autonome, tant en simulation qu’en implémentation matérielle, et permettent d’envisager son intégration ultérieure dans un système basé sur le bus Avalon et le processeur Nios II.

2.3 Intégration de l’IP télémètre avec l’interface Avalon

Afin de permettre l’intégration du télémètre ultrason dans un système SoPC basé sur le processeur *Nios II*, une interface compatible Avalon a été ajoutée à l’IP précédemment validée en mode autonome. Cette étape vise à rendre la distance mesurée accessible au processeur via une lecture sur le bus Avalon.

L’intégration a été réalisée en conservant l’architecture interne du télémètre inchangée, et en ajoutant un module d’encapsulation assurant l’interface avec le bus. Cette approche permet de séparer clairement la partie opérative (mesure ultrason) de la partie communication (bus Avalon).

2.3.1 Principe de l’interface Avalon

L’interface Avalon implémentée est volontairement simplifiée et limitée à des opérations de lecture, conformément au sujet. Elle repose sur les signaux suivants :

- **chipselect** : indique que le périphérique est sélectionné par le processeur ;

- **Read_n** : signal de lecture actif à l'état bas ;
- **readdata[31:0]** : bus de données retourné au processeur.

Lorsque **chipselect** = 1 et **Read_n** = 0, l'IP place la valeur de la distance mesurée sur le bus **readdata**. La distance, codée sur 10 bits (**Dist_cm[9:0]**), est insérée dans les bits de poids faible du bus, tandis que les bits supérieurs sont forcés à zéro :

$$\text{readdata}[9:0] = \text{Dist_cm}, \quad \text{readdata}[31:10] = 0$$

La valeur lire est mémorisée dans un registre interne et conservée jusqu'à la prochaine opération de lecture, ce qui permet d'éviter toute instabilité du signal en dehors des phases d'accès au bus.

2.3.2 Validation par simulation

Le bon fonctionnement de l'interface Avalon a été validé par simulation sous ModelSim. Le banc de test génère des signaux **ECHO** correspondant à différentes distances simulées (notamment 20 cm et 100 cm), puis déclenche une lecture Avalon après chaque mesure.

La Figure 4 présente les chronogrammes obtenus lors de cette simulation. On y observe :

- la génération du signal **TRIG** et la mesure correcte de la durée du signal **ECHO** ;
- la mise à jour de la sortie **Dist_cm** à la fin de chaque mesure ;
- l'activation de **chipselect** et de **Read_n** lors de la lecture Avalon ;
- la cohérence entre la valeur de **Dist_cm** et les bits **readdata[9:0]** lus sur le bus.

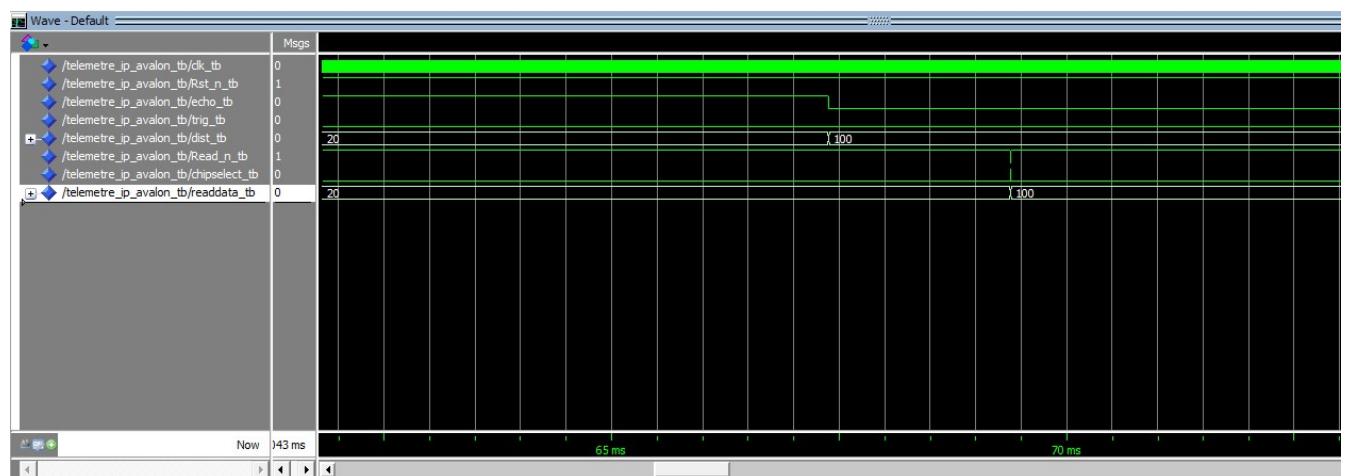


FIGURE 4 – Simulation de l'IP télémètre avec interface Avalon : mesure ultrason et lecture de la distance via **readdata**

Cette simulation confirme que l'IP télémètre est correctement intégrée au bus Avalon et que la distance mesurée peut être lue de manière fiable par un processeur Nios II. L'IP est ainsi prête à être intégrée dans un système SoPC complet pour les étapes suivantes du projet.

2.3.3 Gestion de la fréquence d'horloge dans le système SoC

Lors de l'intégration de l'IP télémètre dans le système SoC-FPGA, un comportement différent de celui observé en mode autonome a été mis en évidence. En particulier, la distance mesurée semblait être limitée à environ 200 cm, alors que le capteur ultrason HC-SR04 est spécifié pour une portée maximale d'environ 400 cm.

Cette anomalie est due au fait que l'IP télémètre avait initialement été conçue en supposant une fréquence d'horloge fixe de 50 MHz. Or, dans le système basé sur *Platform Designer*, l'horloge fournie à l'IP Avalon est générée par le PLL du système et peut différer de cette valeur. Dans la configuration retenue, la fréquence effective du bus est de 100 MHz. Le comptage du temps de propagation de l'écho étant directement proportionnel à la période d'horloge, une fréquence incorrecte entraîne une erreur systématique sur la distance calculée.

Afin de corriger ce problème de manière propre et générique, la fréquence d'horloge a été paramétrée dans l'IP télémètre à l'aide d'un *generic CLK_FREQ_HZ*. Toutes les constantes temporelles internes (durée de l'impulsion TRIG, période minimale entre deux mesures et facteur de conversion cycles/cm) sont désormais dérivées de ce paramètre. Le *generic* est configuré lors de l'instanciation de l'IP dans sa version Avalon, garantissant ainsi une mesure correcte et cohérente avec la fréquence réelle du système.

Cette approche supprime toute correction logicielle artificielle et rend l'IP télémètre entièrement portable et réutilisable dans différents contextes matériels.

2.3.4 Validation expérimentale sur la carte DE10-Lite

Après l'intégration complète de l'IP télémètre dans le système SoC-FPGA et la correction de la gestion de la fréquence d'horloge, une validation expérimentale a été réalisée sur la carte DE10-Lite. Le capteur HC-SR04 est connecté aux broches GPIO de la carte, tandis que la distance mesurée est lue par le processeur Nios II via le bus Avalon.

La Figure 5 présente la configuration expérimentale retenue. Un obstacle est placé à une distance d'environ 15 cm du capteur, mesurée à l'aide d'une règle graduée. La valeur retournée par l'IP est affichée directement sur les afficheurs 7 segments de la carte, confirmant la cohérence entre la distance réelle et la distance mesurée.

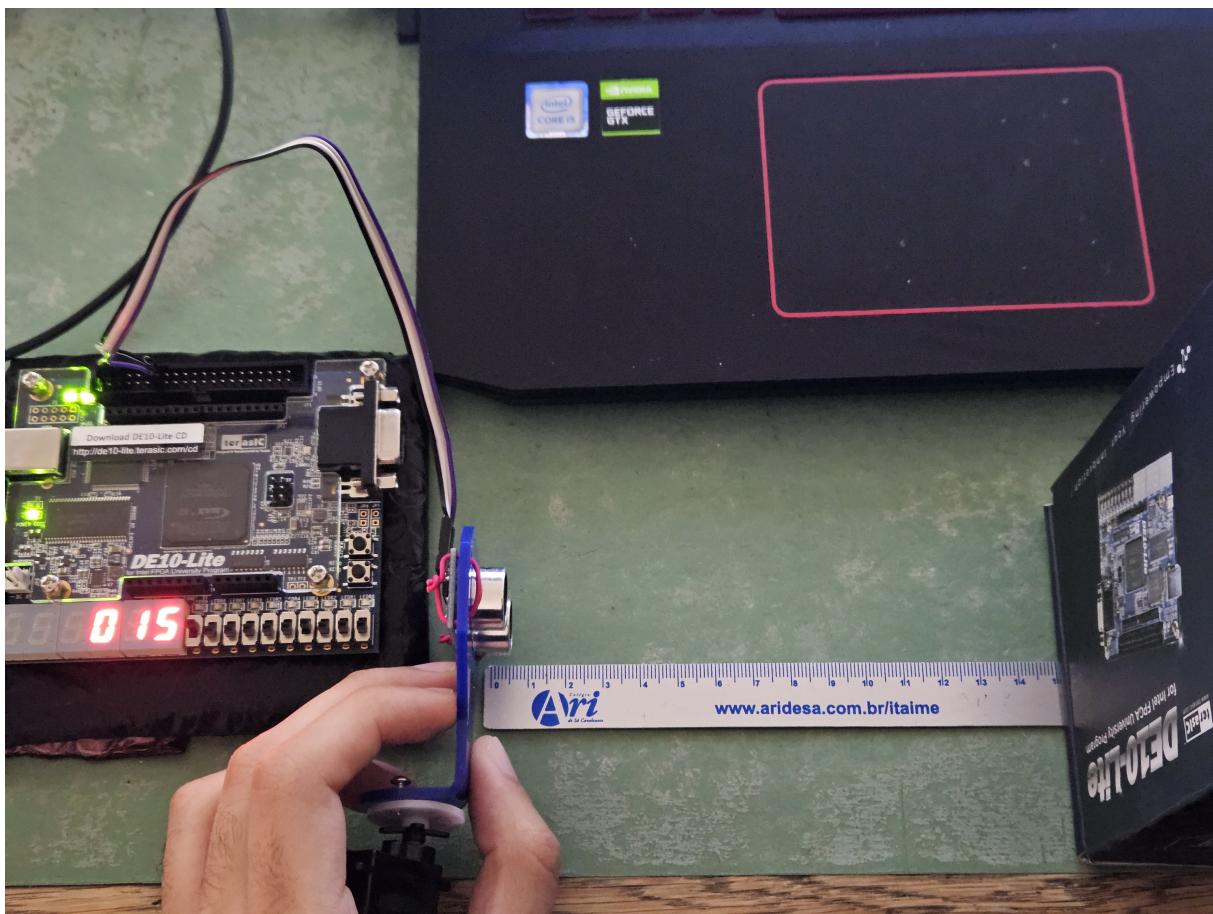
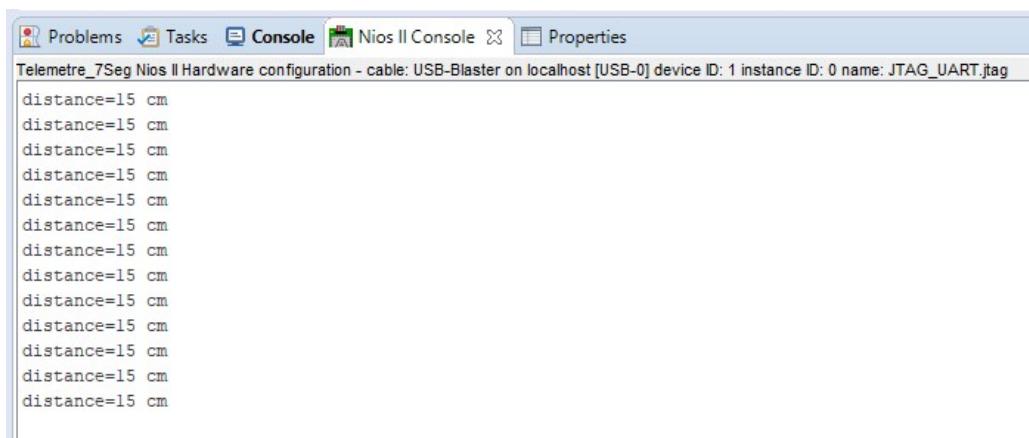


FIGURE 5 – Validation expérimentale du télémètre ultrason sur la carte DE10-Lite : mesure d'une distance d'environ 15 cm et affichage sur les afficheurs 7 segments

En complément, la Figure 6 montre la sortie de la console série du processeur Nios II. La distance mesurée est lue via l'interface Avalon et affichée périodiquement sur le terminal, ce qui permet de confirmer la bonne communication entre le matériel et le logiciel.



```

Problems Tasks Console Nios II Console ✘ Properties
Telemetre_7Seg Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: JTAG_UART.jtag
distance=15 cm

```

FIGURE 6 – Affichage de la distance mesurée dans la console série du processeur Nios II après lecture du périphérique Avalon

Ces résultats valident le bon fonctionnement de l'IP télémètre dans un environnement

SoC-FPGA complet, depuis la génération du signal ultrason jusqu'à la lecture logicielle via le bus Avalon. L'IP est ainsi considérée comme pleinement fonctionnelle et prête à être utilisée pour les étapes suivantes du projet.

3 Servomoteur

Le servomoteur constitue un élément essentiel du projet, car il sera utilisé dans les étapes suivantes pour réaliser un balayage angulaire de la scène mesurée par le télémètre ultrason. Cette section présente la conception, la calibration et la validation expérimentale d'une première version autonome de l'IP servomoteur, commandée directement à l'aide des interrupteurs de la carte DE10-Lite.

3.1 IP Servomoteur Standalone

3.1.1 Architecture de l'IP

L'IP servomoteur a été conçue de manière **comportementale**, sans recourir à une machine à états explicite. Cette approche permet une implémentation simple et efficace, tout en respectant les contraintes temporelles imposées par le pilotage d'un servomoteur standard.

L'architecture repose sur un compteur synchrone cadencé par l'horloge système à 50 MHz. Ce compteur génère une période fixe de 20 ms, correspondant à la période typique du signal de commande PWM d'un servomoteur. À chaque début de période, la consigne de position, codée sur 10 bits, est échantillonnée et utilisée pour calculer la largeur de l'impulsion à l'état haut.

La sortie **commande** est maintenue à l'état haut tant que la valeur du compteur est inférieure à la largeur d'impulsion calculée, puis repasse à l'état bas jusqu'à la fin de la période. Cette méthode garantit une fréquence constante du signal PWM et une largeur d'impulsion stable sur chaque période, évitant ainsi tout phénomène de jitter susceptible de perturber le fonctionnement mécanique du servomoteur.

3.1.2 Calibration expérimentale des limites angulaires

En théorie, les servomoteurs sont généralement commandés à l'aide d'impulsions comprises entre 1 ms et 2 ms, correspondant respectivement aux positions angulaires extrêmes. Toutefois, en pratique, ces valeurs peuvent varier significativement selon le modèle de servomoteur et le montage mécanique.

Une phase de calibration expérimentale a donc été réalisée afin de déterminer des limites de commande adaptées au servomoteur utilisé. Cette calibration a consisté à ajuster progressivement les valeurs minimales et maximales de la largeur d'impulsion PWM, tout en observant le comportement mécanique du servomoteur.

Les tests ont permis d'identifier les valeurs suivantes comme étant sûres et reproducibles, sans bruit de butée ni contrainte mécanique excessive :

$$\text{PULSE_MIN} = \frac{f_{\text{clk}}}{1850} \approx 0.54 \text{ ms}$$

$$\text{PULSE_MAX} = \frac{f_{\text{clk}}}{458} \approx 2.18 \text{ ms}$$

Ces valeurs permettent de couvrir un intervalle angulaire effectif d'environ 0° à 180° , avec une bonne linéarité et une stabilité satisfaisante.

3.1.3 Relation entre consigne et angle

La consigne de position est codée sur 10 bits et fournie directement par les interrupteurs de la carte DE10-Lite. Elle est convertie linéairement en une largeur d'impulsion comprise entre les valeurs calibrées `PULSE_MIN` et `PULSE_MAX`. Cette correspondance permet d'associer la position minimale des interrupteurs à un angle de 0° , et la position maximale à un angle proche de 180° .

3.1.4 Simulation fonctionnelle

Un banc de test (*testbench*) a été développé afin de valider le fonctionnement temporel de l'IP servomoteur. La simulation permet de visualiser directement la période du signal PWM ainsi que la largeur de l'impulsion à l'état haut pour différentes valeurs de consigne.

Deux cas ont été simulés :

- une consigne minimale, correspondant à la largeur d'impulsion minimale ;
- une consigne maximale, correspondant à la largeur d'impulsion maximale.

La Figure 7 illustre la forme d'onde obtenue en simulation, mettant en évidence une période de 20 ms et des largeurs d'impulsion conformes aux valeurs attendues.

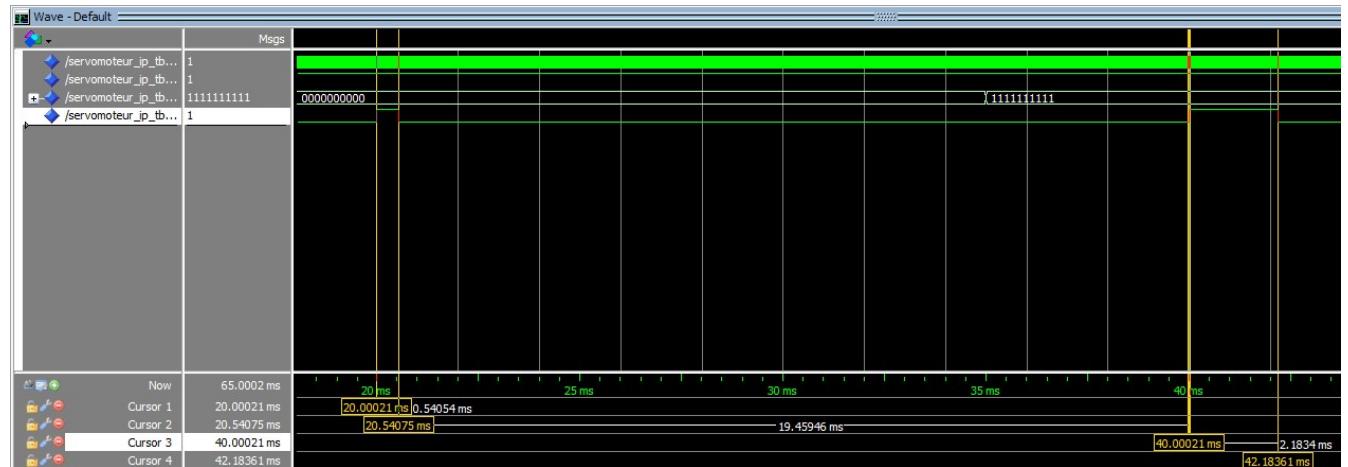


FIGURE 7 – Simulation temporelle du signal PWM généré par l'IP servomoteur pour les positions minimale et maximale

3.1.5 Validation expérimentale sur la carte DE10-Lite

Après validation en simulation, l'IP servomoteur a été intégrée sur la carte DE10-Lite en mode autonome. La consigne de position est fournie par les interrupteurs, tandis que le signal de commande PWM est appliqué directement à l'entrée du servomoteur.

Les Figures 8, 9 et 10 présentent respectivement les positions correspondant à une consigne minimale, intermédiaire et maximale. L'orientation de la flèche fixée sur l'axe du servomoteur permet de visualiser clairement l'angle atteint.

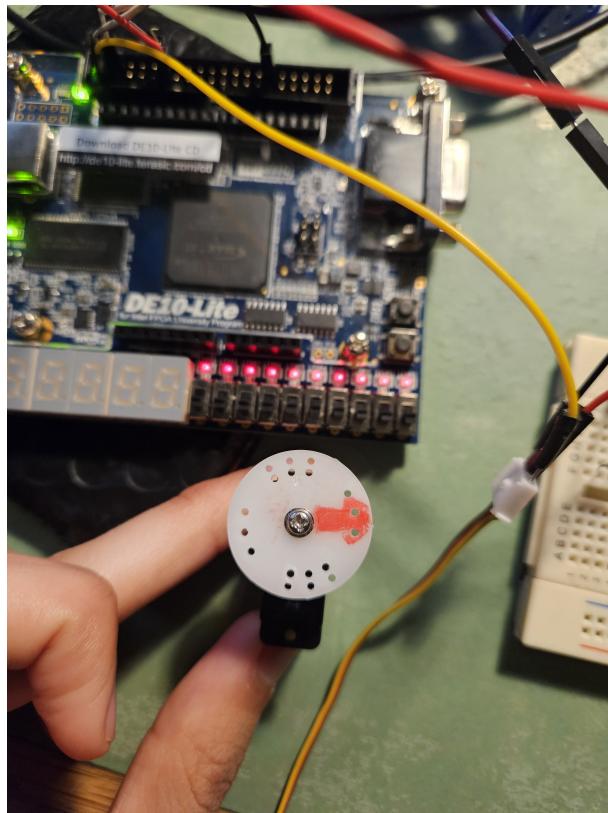


FIGURE 8 – Position du servomoteur pour une consigne minimale (0°)

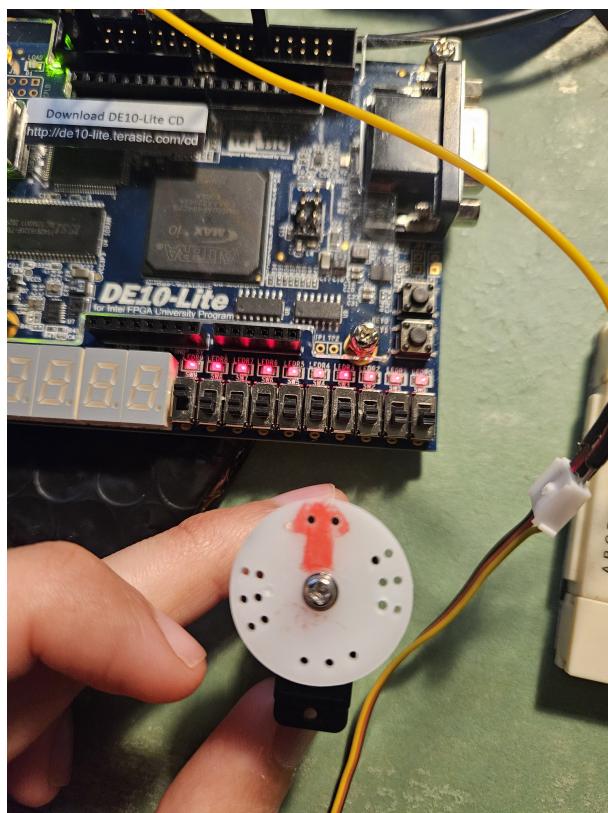


FIGURE 9 – Position intermédiaire du servomoteur pour une consigne médiane ($\sim 90^\circ$)

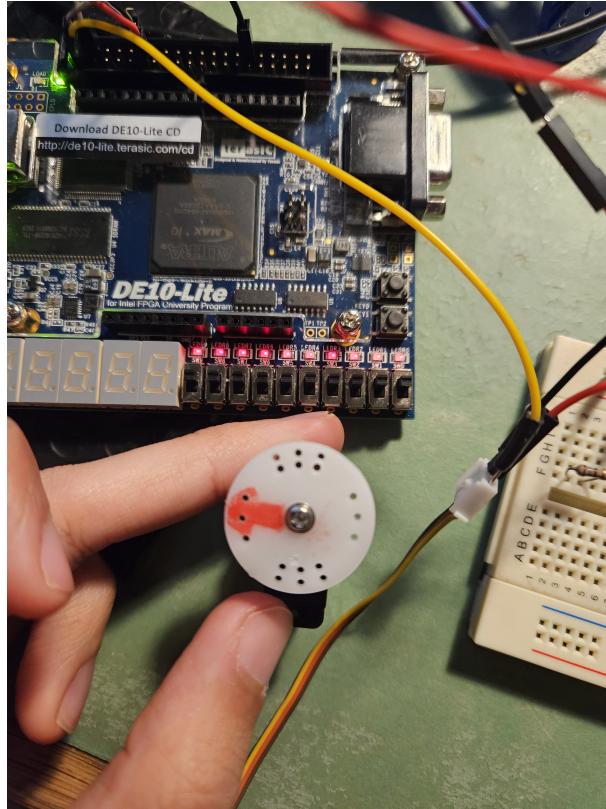


FIGURE 10 – Position maximale du servomoteur pour une consigne élevée ($\sim 180^\circ$)

Ces résultats expérimentaux confirment le bon fonctionnement de l'IP servomoteur en mode autonome. La génération du signal PWM est conforme aux attentes, la variation angulaire est continue et stable, et les limites mécaniques du servomoteur sont respectées grâce à la calibration préalable des temps d'impulsion.

3.2 Extension de l'IP Servomoteur vers le bus Avalon

Afin de permettre le pilotage du servomoteur depuis le processeur Nios II, l'IP servomoteur autonome a été étendue pour devenir un périphérique compatible avec le bus Avalon. Cette extension permet d'écrire dynamiquement la consigne de position depuis le logiciel, tout en conservant le cœur de génération PWM précédemment validé.

3.2.1 Architecture de l'IP Servomoteur Avalon

L'IP Servomoteur Avalon repose sur une architecture modulaire. Le cœur fonctionnel de génération du signal PWM est réutilisé sans modification, tandis qu'une couche d'interface Avalon est ajoutée afin de gérer les transactions sur le bus.

Cette interface implémente un périphérique **en écriture seule**. Elle comporte un registre interne de position, mis à jour uniquement lorsque les signaux `chipselect` et `write_n` indiquent une opération d'écriture valide sur le bus Avalon. Le signal `reset_n`, actif à l'état bas, permet de réinitialiser ce registre ainsi que le cœur de génération PWM.

La valeur écrite sur le bus est fournie via le signal `WriteData`. Les 10 bits de poids faible sont utilisés pour coder la consigne de position, ce qui permet une compatibilité directe

avec des accès en 8, 16 ou 32 bits depuis le processeur Nios II. Cette consigne est ensuite transmise au cœur servomoteur, qui génère le signal de commande PWM correspondant.

3.2.2 Validation par simulation

Un banc de test spécifique a été développé afin de valider le comportement de l'IP servomoteur connectée au bus Avalon. L'objectif principal de cette simulation est de vérifier que la consigne de position n'est prise en compte que lors d'une écriture valide sur le bus, c'est-à-dire lorsque les signaux `chipselect` et `write_n` sont correctement positionnés.

La simulation met en évidence les points suivants :

- une écriture valide sur le bus Avalon entraîne une mise à jour du registre de position interne ;
- une tentative d'écriture avec `chipselect` inactif ou `write_n` désactivé n'a aucun effet sur la commande du servomoteur ;
- la largeur d'impulsion du signal PWM est modifiée uniquement à la suite d'une écriture valide.

La Figure 11 illustre cette séquence de simulation. On observe que le signal `commande` ne change de largeur que lorsque les conditions d'écriture Avalon sont satisfaites, validant ainsi le bon fonctionnement de l'interface.

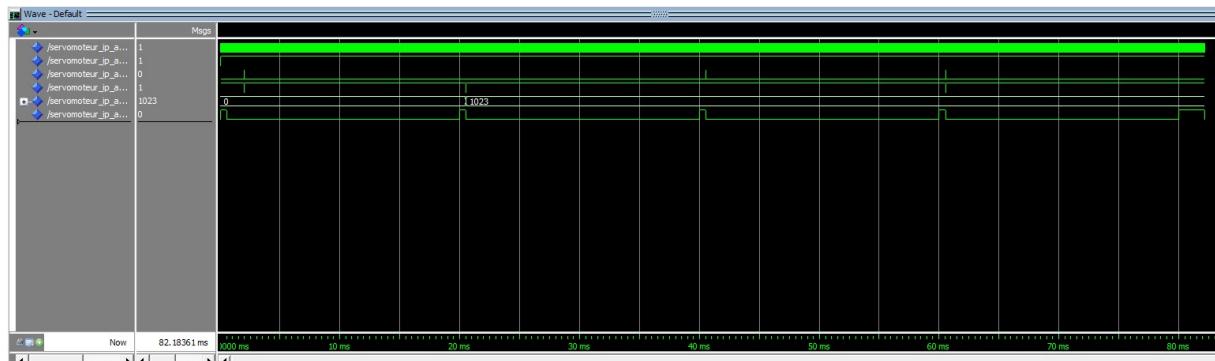


FIGURE 11 – Simulation de l'IP Servomoteur connectée au bus Avalon : mise à jour de la commande PWM uniquement lors d'écritures valides

Cette simulation confirme que l'IP Servomoteur Avalon respecte le protocole Avalon attendu et qu'elle est prête à être intégrée dans le système SoC-FPGA complet, où la consigne de position sera pilotée par le logiciel exécuté sur le processeur Nios II.

3.3 Programmation logicielle et test de l'IP Servomoteur

Après l'intégration de l'IP Servomoteur dans le système SoC-FPGA via le bus Avalon, une application logicielle a été développée sur le processeur Nios II afin de piloter la position angulaire du servomoteur à partir des interrupteurs matériels de la carte.

Le principe retenu est similaire à celui utilisé pour l'IP télémètre. Les interrupteurs `SW[9:0]` fournissent une consigne codée sur 10 bits, comprise entre 0 et 1023. Cette valeur est lue par le processeur, convertie en un angle compris entre 0° et 180° , puis transmise à l'IP Servomoteur par une écriture sur le bus Avalon.

La communication avec l'IP est réalisée à l'aide d'une écriture Avalon (`chipselect = 1, write_n = 0`), ce qui met à jour le registre interne de position du périphérique. Le signal PWM est ensuite généré entièrement en matériel par l'IP Servomoteur, garantissant une précision temporelle indépendante de l'exécution logicielle.

Afin de faciliter la visualisation et la validation du comportement du système, l'angle calculé par le processeur est également affiché sur les afficheurs 7 segments de la carte DE10-Lite. Les trois afficheurs de poids faible indiquent directement la valeur angulaire en degrés.

Le programme assure successivement la lecture des interrupteurs, le calcul de l'angle, l'écriture de la consigne dans l'IP Servomoteur et la mise à jour de l'affichage. Une temporisation logicielle de l'ordre de quelques dizaines de millisecondes est ajoutée afin de stabiliser l'affichage et de limiter la charge processeur.

3.3.1 Validation expérimentale

Des tests expérimentaux ont été réalisés sur la carte DE10-Lite afin de valider le bon fonctionnement de la chaîne complète, depuis la commande logicielle jusqu'au déplacement mécanique du servomoteur.

La Figure 12 présente le comportement du système pour une consigne nulle ($SW = 0$). Le servomoteur se positionne correctement à l'angle minimal, correspondant à environ 0° . L'affichage 7 segments indique également une valeur proche de zéro.

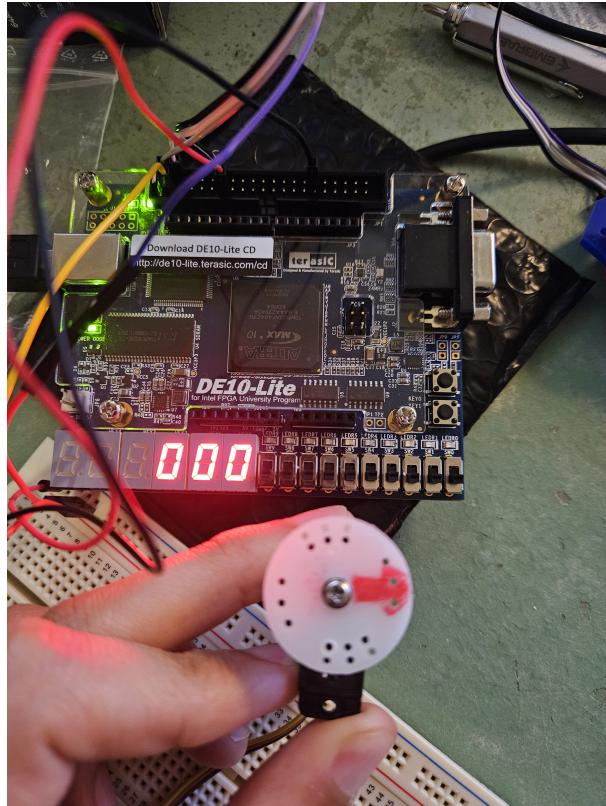


FIGURE 12 – Position du servomoteur pour une consigne nulle pilotée par le Nios II via Avalon

La Figure 13 illustre un cas intermédiaire, pour lequel les interrupteurs sont positionnés

de manière à produire une consigne d'environ 112° . Le servomoteur adopte une position cohérente avec la valeur affichée, confirmant la linéarité de la conversion logicielle et la stabilité de la génération PWM matérielle.

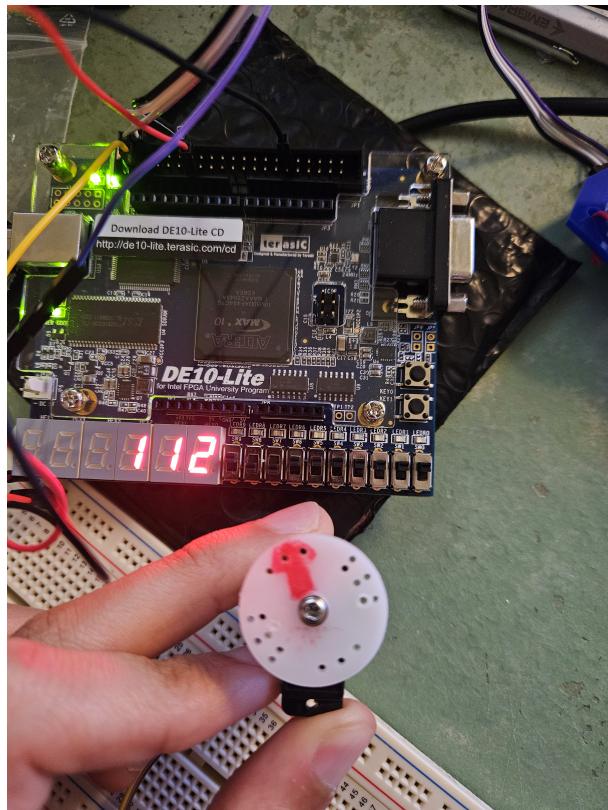


FIGURE 13 – Position intermédiaire du servomoteur pour une consigne logicielle d'environ 112°

Ces tests valident le bon fonctionnement de l'IP Servomoteur dans un environnement SoC-FPGA complet. La commande du servomoteur est réalisée de manière fiable via le bus Avalon, avec une séparation claire entre le calcul logiciel de la consigne et la génération matérielle du signal PWM. L'IP est ainsi prête à être exploitée dans les étapes suivantes du projet, notamment pour la réalisation d'un système de balayage angulaire associé au télémètre ultrason.

4 Affichage des obstacles

Cette étape vise à mettre en œuvre une détection angulaire des obstacles en combinant la rotation du servomoteur sur une plage de 180° et des mesures régulières de distance effectuées par le télémètre ultrason. Le système ainsi obtenu constitue une première approche d'un radar bidimensionnel, dans lequel chaque mesure de distance est associée à un angle précis.

Le principe retenu consiste à faire balayer le servomoteur de 0° à 180° , puis de 180° à 0° , de manière continue. À chaque position angulaire, une mesure de distance est effectuée via l'IP télémètre, puis les résultats sont affichés à la fois sur les afficheurs 7 segments de la carte et dans le terminal du Nios II SBT.

4.1 Fonctionnement global

Le balayage angulaire est piloté par le processeur Nios II, qui génère une consigne angulaire croissante ou décroissante. Cette consigne est convertie en une valeur codée sur 10 bits et transmise à l'IP Servomoteur via une écriture sur le bus Avalon. Le signal PWM correspondant est généré entièrement en matériel par l'IP Servomoteur.

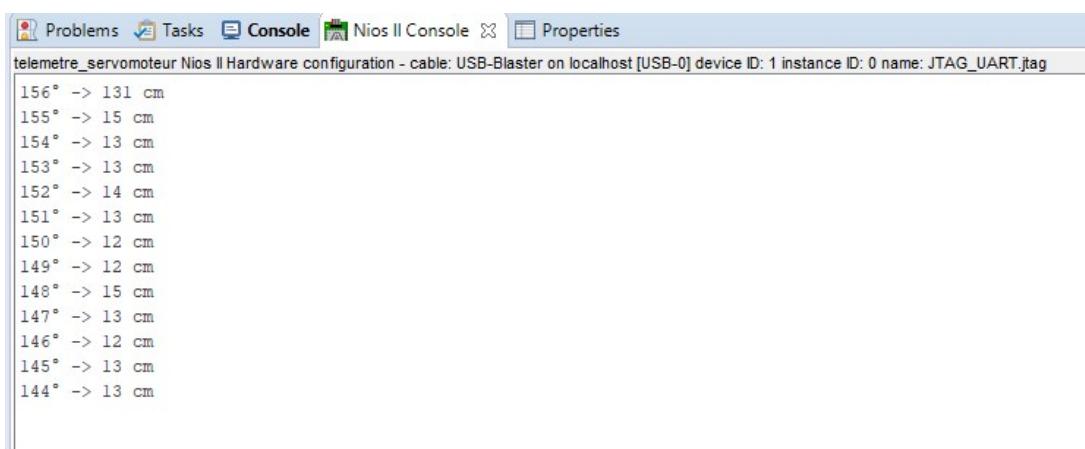
Après une temporisation permettant la stabilisation mécanique du servomoteur, le processeur lit la distance mesurée par l'IP télémètre. Le couple *angle – distance* est alors :

- affiché dans le terminal du Nios II SBT sous forme textuelle,
- affiché sur les afficheurs 7 segments de la carte DE10-Lite.

Les trois afficheurs de poids faible (HEX2–HEX0) indiquent l'angle en degrés, tandis que les trois afficheurs de poids fort (HEX5–HEX3) affichent la distance mesurée en centimètres.

4.2 Affichage et résultats expérimentaux

La Figure 14 illustre un extrait des résultats affichés dans le terminal du Nios II SBT. Chaque ligne correspond à une position angulaire donnée et à la distance mesurée dans cette direction. Ce format textuel permet de vérifier facilement la cohérence des mesures et constitue une base exploitable pour un traitement ultérieur (cartographie ou visualisation graphique).



```

Problems Tasks Console Nios II Console Properties
telemetre_servomoteur Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: JTAG_UART.jtag
156° -> 131 cm
155° -> 15 cm
154° -> 13 cm
153° -> 13 cm
152° -> 14 cm
151° -> 13 cm
150° -> 12 cm
149° -> 12 cm
148° -> 15 cm
147° -> 13 cm
146° -> 12 cm
145° -> 13 cm
144° -> 13 cm

```

FIGURE 14 – Affichage des couples angle–distance dans le terminal du Nios II SBT lors du balayage angulaire

Les Figures 15 et 16 montrent le dispositif expérimental en fonctionnement. Le servomoteur entraîne le module ultrason, tandis que l'affichage sur la carte indique simultanément l'angle courant et la distance mesurée face à un obstacle placé à distance fixe.

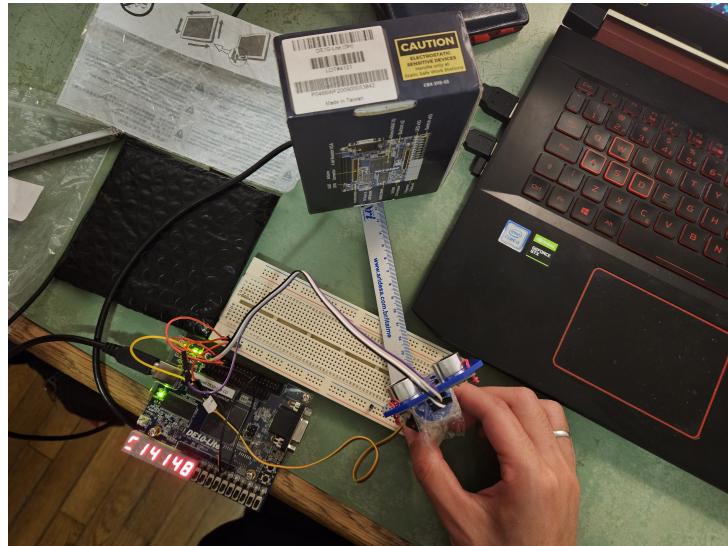


FIGURE 15 – Mesure de distance associée à une position angulaire intermédiaire du servomoteur | 148° et 14 cm

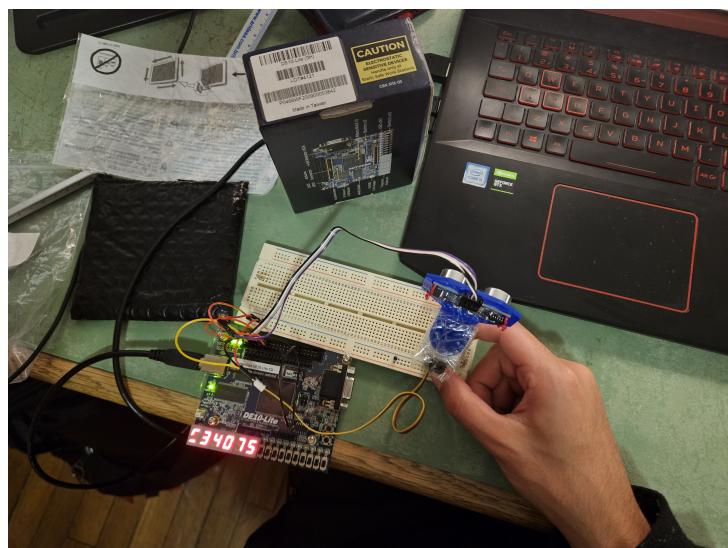


FIGURE 16 – Balayage angulaire du télémètre ultrason et affichage simultané de l'angle et de la distance | 75° et 34 cm

Lors des tests, un dysfonctionnement matériel a été observé sur l'afficheur HEX5, dont certains segments ne s'allument pas correctement. Ce défaut est attribué à la carte et non au fonctionnement du système conçu, les valeurs affichées restant cohérentes avec celles observées dans le terminal série. L'exploitation conjointe de l'affichage textuel et de l'affichage sur 7 segments permet ainsi de valider le bon fonctionnement global malgré cette limitation matérielle.

5 Affichage VGA – Radar 2D

Cette étape a pour objectif de représenter sur un écran VGA une cartographie bidimensionnelle de l'environnement, en associant à chaque angle du servomoteur une mesure

de distance issue du télémètre ultrason. Le résultat attendu est un affichage de type *radar* : un demi-disque représentant le champ de vision du capteur sur 180°, avec une ligne de balayage et une coloration indiquant la présence (ou l'absence) d'obstacle dans la direction considérée.

5.1 Mise en place du sous-système VGA

L'affichage VGA est réalisé à l'aide du sous-système `VGA_Subsystem` fourni dans l'environnement SoC, comprenant :

- un *pixel buffer* permettant d'écrire des pixels en mémoire vidéo (format RGB565) ;
- un *character buffer* permettant d'afficher des caractères ASCII ;
- un contrôleur DMA assurant la lecture continue du buffer vidéo vers la sortie VGA.

Un premier programme de test issu de la documentation DE10-Lite a été utilisé afin de valider la chaîne graphique (accès mémoire, conversion de couleurs, et affichage réel sur écran). La Figure 17 présente la sortie obtenue lors de ce test, confirmant le fonctionnement correct du sous-système VGA et l'écriture dans les buffers.

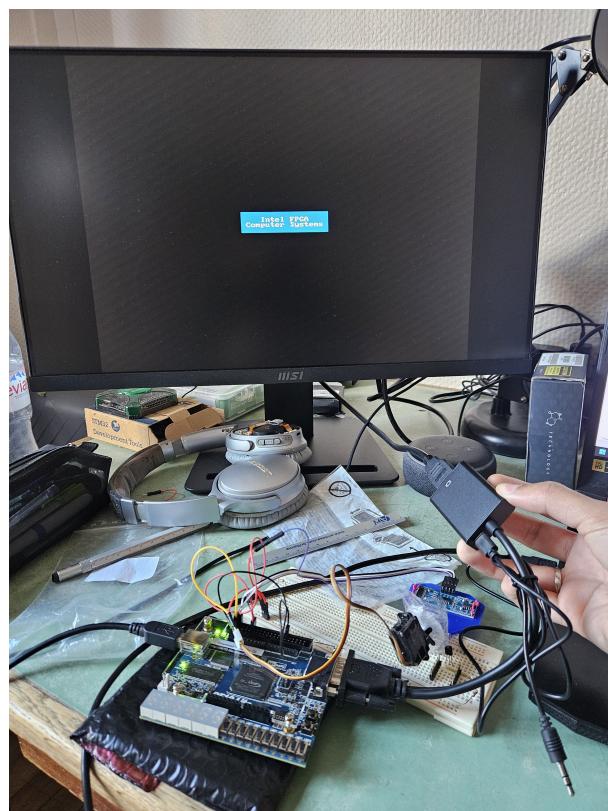


FIGURE 17 – Validation du sous-système VGA : affichage d'un rectangle et d'un texte sur l'écran

5.1.1 Nettoyage des buffers à l'initialisation

Afin d'éviter l'apparition de résidus visuels liés aux données précédemment présentes en mémoire vidéo, le logiciel effectue au démarrage :

- une remise à zéro complète du *pixel buffer* (fond noir) ;

- une remise à zéro complète du *character buffer* (écriture d'espaces).

Cette étape garantit un affichage propre et reproductible à chaque exécution.

5.2 Principe de la représentation radar

5.2.1 Choix géométriques et repère

L'écran est exploité en résolution logique 320 x 240 pixels. L'origine du radar est choisie en bas au centre de l'écran, de coordonnées :

$$(c_x, c_y) = \left(\frac{W}{2}, H - 10 \right)$$

où $W = 320$ et $H = 240$. Le champ de vision est représenté par un demi-disque (semi-cercle) de rayon R en pixels. Dans l'implémentation, un rayon fixe $R = 200$ est utilisé afin d'obtenir une représentation large et lisible sur l'écran.

Le balayage angulaire s'effectue sur 0° à 180° , avec la convention suivante :

- $\theta = 0^\circ$ correspond à la direction **droite** de l'écran ;
- $\theta = 90^\circ$ correspond à la direction **verticale** (vers le haut) ;
- $\theta = 180^\circ$ correspond à la direction **gauche**.

5.2.2 Conversion polaire → cartésienne

À chaque angle θ , le télémètre fournit une distance D (en cm). Cette mesure est saturée à une portée maximale choisie pour cette étape :

$$D \leq D_{\max} \quad \text{avec} \quad D_{\max} = 75 \text{ cm}$$

La distance est ensuite convertie en pixels par une mise à l'échelle linéaire :

$$r(D) = \frac{D}{D_{\max}} \cdot R$$

La conversion polaire–cartésienne permettant d'obtenir le point mesuré sur l'écran est alors :

$$x = c_x + r(D) \cos(\theta), \quad y = c_y - r(D) \sin(\theta)$$

Le signe négatif sur y provient du repère image, où l'axe vertical est orienté vers le bas.

De manière similaire, le point maximal sur le bord du radar (rayon R) est donné par :

$$x_{\max} = c_x + R \cos(\theta), \quad y_{\max} = c_y - R \sin(\theta)$$

5.2.3 Affichage statique : axes et arcs de distance

L'arrière-plan du radar est composé :

- d'un axe horizontal et d'un axe vertical (couleur bleue) ;
- de plusieurs arcs concentriques (couleur grise) servant de repères de distance ;
- d'un arc externe (couleur blanche) matérialisant la limite du radar.

Pour conserver un affichage indépendant du choix de D_{\max} , les arcs internes sont placés à $\frac{1}{4}D_{\max}$, $\frac{1}{2}D_{\max}$ et $\frac{3}{4}D_{\max}$, tandis que l'arc externe correspond à D_{\max} .

5.3 Affichage dynamique : balayage et coloration des mesures

5.3.1 Balayage angulaire et acquisition

Le processeur Nios II pilote le servomoteur en faisant varier l'angle de 0° à 180° , puis de 180° à 0° , afin d'éviter un retour brutal. La commande est discrétisée avec un pas angulaire fixé à :

$$\Delta\theta = 2^\circ$$

Après l'écriture de la consigne au servomoteur (via l'IP Avalon), une temporisation logicielle est introduite afin de garantir la stabilisation mécanique avant lecture de la distance.

5.3.2 Codage couleur des obstacles

L'objectif visuel retenu est le suivant :

- la portion **sans obstacle** est affichée en **vert** ;
 - la portion **au-delà de l'obstacle** est affichée en **rouge** jusqu'à la limite du radar.
- Ainsi, pour chaque angle, deux segments sont tracés :
- un segment vert de (c_x, c_y) jusqu'au point mesuré (x, y) ;
 - un segment rouge de (x, y) jusqu'au bord (x_{\max}, y_{\max}) .

Ce choix permet d'obtenir une lecture immédiate : une zone fortement rouge traduit une proximité d'obstacle (distance faible), tandis qu'une zone majoritairement verte indique l'absence d'obstacle à courte distance. Les mesures restent volontairement *persistentes* (accumulation des tracés) afin de construire progressivement une cartographie 2D.

5.3.3 Affichage simultané sur 7 segments et terminal

En complément de l'affichage VGA, le système conserve les moyens de validation mis en œuvre aux étapes précédentes :

- affichage de l'angle (HEX2–HEX0) et de la distance (HEX5–HEX3) sur les afficheurs 7 segments ;
- affichage des couples *angle-distance* sur la console série du Nios II SBT.

5.4 Résultats expérimentaux

La Figure 18 montre l'affichage radar obtenu sur écran VGA lors d'un test expérimental. Les axes et arcs concentriques permettent d'interpréter la scène, tandis que l'accumulation des balayages forme progressivement une cartographie semi-circulaire. Les zones rouges traduisent les directions où un obstacle est détecté à une distance inférieure à 75 cm, tandis que les zones vertes indiquent un espace libre à courte portée.

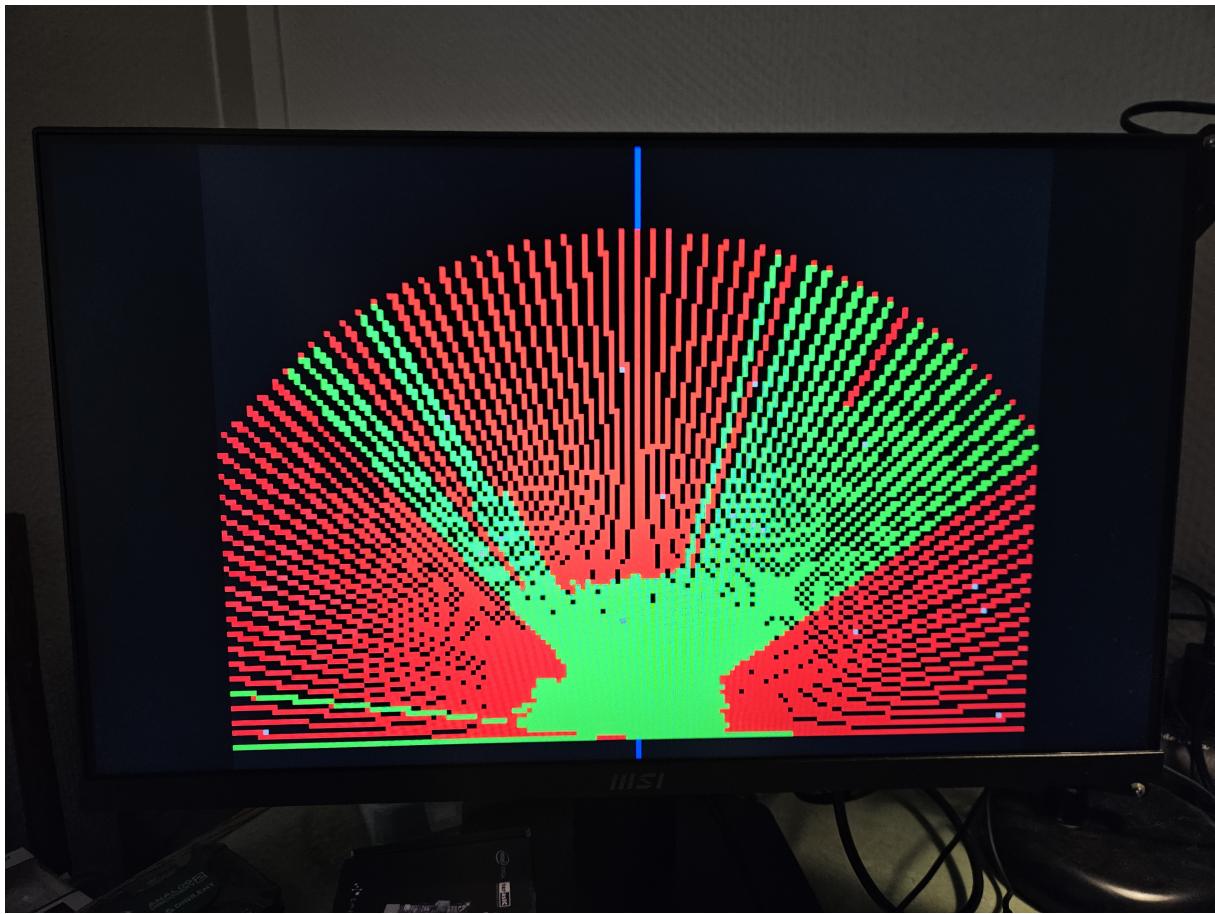


FIGURE 18 – Affichage VGA du radar 2D : demi-disque, axes, arcs de distance et balayage coloré (vert/rouge)

La Figure 19 présente le montage expérimental utilisé pour cette étape : la carte DE10-Lite, le servomoteur portant le module HC-SR04 et l’obstacles.

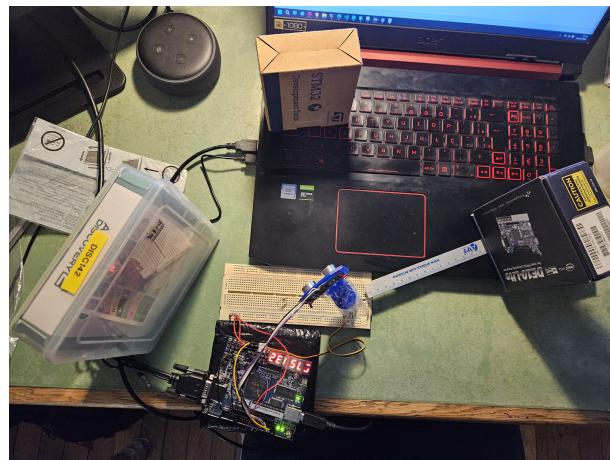


FIGURE 19 – Montage expérimental complet : DE10-Lite, servomoteur, HC-SR04 et des obstacles

Une démonstration vidéo du fonctionnement (balayage, mesures et affichage radar) est

également disponible¹.

L'ensemble des résultats valide le bon fonctionnement de la chaîne complète en environnement SoC-FPGA : pilotage du servomoteur via Avalon, acquisition de distance via l'IP télémètre, et visualisation en temps réel sur écran VGA sous forme de radar 2D.

6 UART

Dans cette dernière étape du projet, un périphérique de communication série de type UART a été conçu afin de permettre l'échange de données entre le système SoC-FPGA et un ordinateur hôte. Cette interface UART constitue un moyen simple et universel d'exporter les données cartographiées par le radar, ainsi que de recevoir ultérieurement des commandes de configuration.

L'implémentation a été réalisée sous forme d'une IP matérielle en VHDL, indépendante du processeur Nios II dans un premier temps, puis validée expérimentalement à l'aide d'un convertisseur USB-série de type FTDI.

6.1 Principe de fonctionnement de l'UART

L'interface UART implémentée repose sur une communication série asynchrone classique, configurée selon les paramètres suivants :

- débit : 115 200 bauds ;
- format : 8 bits de données, pas de parité, 1 bit de stop (8N1) ;
- transmission des bits de données en commençant par le bit de poids faible (LSB first).

Le signal est composé d'un bit de départ (*start bit*) à l'état bas, suivi des 8 bits de données, puis d'un bit de stop à l'état haut. La synchronisation est assurée par un diviseur d'horloge interne calculé à partir de la fréquence du système, paramétrée par un *generic CLK_FREQ_HZ*. Dans le cadre de ce projet, l'IP UART est cadencée par une horloge de 50 MHz.

6.2 Architecture de l'IP UART

L'IP UART est structurée autour de deux machines à états finis indépendantes :

- une FSM d'émission (TX), responsable de la génération du trame UART à partir d'un octet fourni par le système ;
- une FSM de réception (RX), chargée de reconstruire les octets reçus sur la ligne série.

La partie émission est déclenchée par un signal de commande `tx_start`. Lorsqu'une transmission est en cours, le signal `tx_busy` est positionné à l'état haut afin d'éviter toute écriture concurrente. La partie réception détecte automatiquement le front descendant correspondant au bit de départ, puis échantillonne les bits de données au centre de chaque période de baud. Une fois un octet valide reçu, le signal `rx_valid` est activé jusqu'à acquittement explicite via `rx_ack`.

1. <https://drive.google.com/file/d/1671gZXqpN0m7eP0eZDkDI1RWoT3RkAhV/view?usp=sharing>

Cette architecture garantit une séparation claire entre émission et réception, ainsi qu'un fonctionnement robuste vis-à-vis des contraintes temporelles de la communication série asynchrone.

6.3 Simulation fonctionnelle de l'IP UART

Avant toute intégration matérielle, le bon fonctionnement de l'IP UART a été validé par simulation à l'aide du simulateur ModelSim. Un banc de test dédié a été développé afin de tester indépendamment les chemins d'émission et de réception.

Le scénario de simulation retenu est volontairement simple et se limite à la transmission et à la réception d'un unique octet (0x41, correspondant au caractère ASCII 'A'), ce qui permet une lecture claire des chronogrammes.

La Figure 20 présente les résultats de cette simulation. On y observe successivement :

- l'activation du signal `tx_start` et la mise à l'état actif de `tx_busy` ;
- la génération correcte du trame UART sur la ligne `tx` (bit de départ, bits de données, bit de stop) ;
- l'injection d'un trame UART sur la ligne `rx` par le banc de test ;
- l'activation du signal `rx_valid` accompagnée de la valeur correcte sur `rx_data` ;
- l'acquittement de la réception via le signal `rx_ack`.

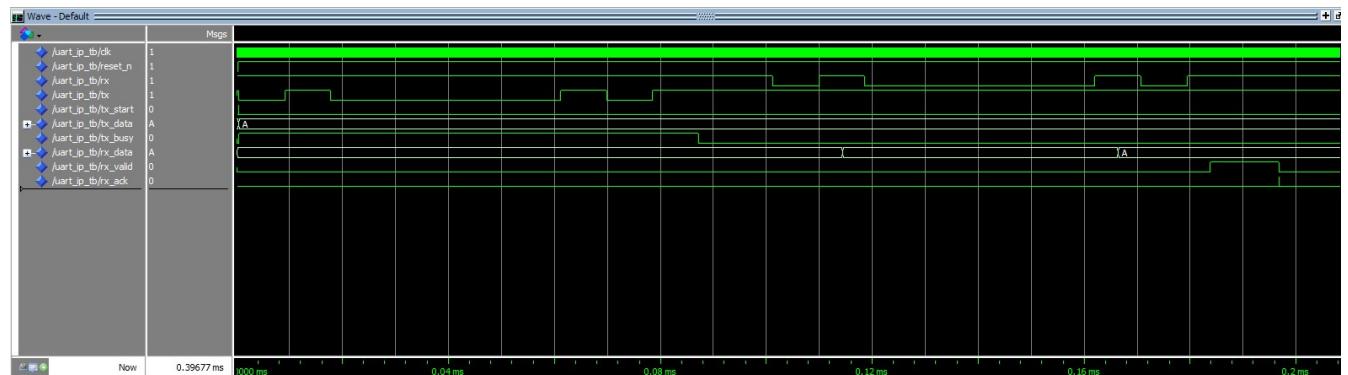


FIGURE 20 – Simulation fonctionnelle de l'IP UART : validation des chemins d'émission (TX) et de réception (RX)

Cette simulation confirme que l'IP UART respecte le protocole série attendu et qu'elle est capable d'émettre et de recevoir correctement des octets dans les conditions nominales.

6.4 Validation expérimentale sur la carte DE10-Lite

Après validation en simulation, l'IP UART a été testée en conditions réelles sur la carte DE10-Lite. La communication avec le PC est assurée par un convertisseur USB-série de type FTDI, connecté aux broches GPIO de la carte. La ligne TX de l'IP est reliée à l'entrée RX du convertisseur, tandis que la ligne RX est reliée à sa sortie TX. Les masses de la carte et du convertisseur sont communes.

Un programme de test a été chargé sur la carte afin de réaliser une fonction d'écho (*echo*) : chaque caractère reçu depuis le PC est immédiatement retransmis sur la ligne

série. Cette configuration permet de valider simultanément la réception et l'émission de données en situation réelle.

La Figure 21 montre la sortie du terminal série sur le PC. Plusieurs chaînes de caractères ("test", "ok", "123") sont envoyées depuis le terminal et sont correctement renvoyées par la carte, confirmant le bon fonctionnement bidirectionnel de l'IP UART à 115 200 bauds.

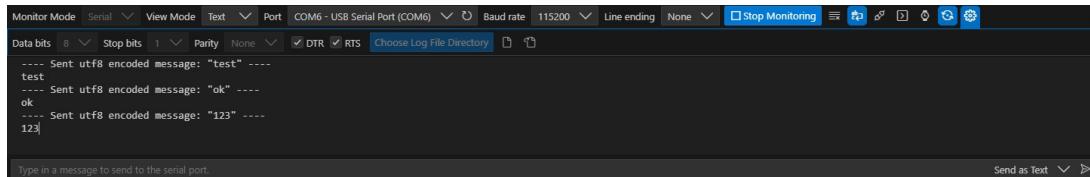


FIGURE 21 – Validation expérimentale de l'IP UART sur la carte DE10-Lite : test d'écho via un terminal série

Ces résultats expérimentaux confirment la cohérence entre les simulations et le comportement observé sur le matériel réel. L'IP UART est ainsi considérée comme fonctionnelle et prête à être intégrée dans le système SoC-FPGA complet autour du processeur Nios II, afin de permettre l'exportation des données radar et la configuration dynamique des paramètres du système.

6.5 Extension de l'IP UART vers le bus Avalon

Afin de permettre l'échange de données entre le système SoC-FPGA et un ordinateur hôte, l'IP UART précédemment validée en mode autonome a été étendue pour devenir un périphérique compatible avec le bus *Avalon Memory-Mapped*. Cette extension permet au processeur Nios II de transmettre et de recevoir des données série via des opérations de lecture et d'écriture sur le bus, sans recourir à des interruptions matérielles.

Le choix a été fait d'implémenter une interface Avalon volontairement simple, basée exclusivement sur un mécanisme de **polling**, afin de limiter la complexité matérielle et de rester cohérent avec les autres IP du projet (télémètre et servomoteur).

6.5.1 Architecture de l'IP UART Avalon

L'architecture de l'IP UART Avalon repose sur la réutilisation directe du cœur UART autonome validé précédemment. Ce cœur assure l'ensemble des fonctions de transmission et de réception série (génération des bits de start, données et stop, échantillonnage côté réception, synchronisation temporelle au baud rate).

Une couche d'interface Avalon est ajoutée afin d'exposer les registres nécessaires au processeur. Deux adresses Avalon sont définies :

- **Adresse 0 (DATA) :**
 - en écriture : envoi d'un octet vers la ligne série (TX) ;
 - en lecture : récupération de l'octet reçu (RX_DATA).
- **Adresse 1 (STATUS) :**
 - bit 0 : indicateur RX_VALID signalant la réception d'un octet ;
 - en écriture : acquittement logiciel (ACK) permettant de réinitialiser le flag RX_VALID.

Lors d'une écriture valide sur le bus (`chipselect = 1` et `write_n = 0`), les données présentes sur `writedata[7:0]` sont transmises au cœur UART pour être envoyées sur la ligne série. De manière analogue, une lecture valide (`chipselect = 1` et `read_n = 0`) permet au processeur de consulter l'état ou les données reçues.

Aucune interruption matérielle n'est utilisée : le processeur interroge périodiquement le registre de statut afin de détecter l'arrivée d'un nouveau caractère.

6.5.2 Validation par simulation de l'interface Avalon

La validation fonctionnelle de l'IP UART Avalon a été réalisée par simulation sous ModelSim à l'aide d'un banc de test dédié. Ce banc de test génère des transactions Avalon ainsi que des signaux UART réalistes, afin de vérifier la cohérence entre le bus et la communication série.

Dans un premier temps, une écriture Avalon est effectuée à l'adresse DATA (adresse 0) afin de transmettre le caractère 'A' (0x41). La Figure 22 montre clairement :

- la mise à l'état actif de `chipselect` ;
- le passage de `write_n` à l'état bas ;
- la valeur 0x41 présente sur le bus `writedata` ;
- le déclenchement de la trame UART correspondante sur la sortie `uart_tx`.

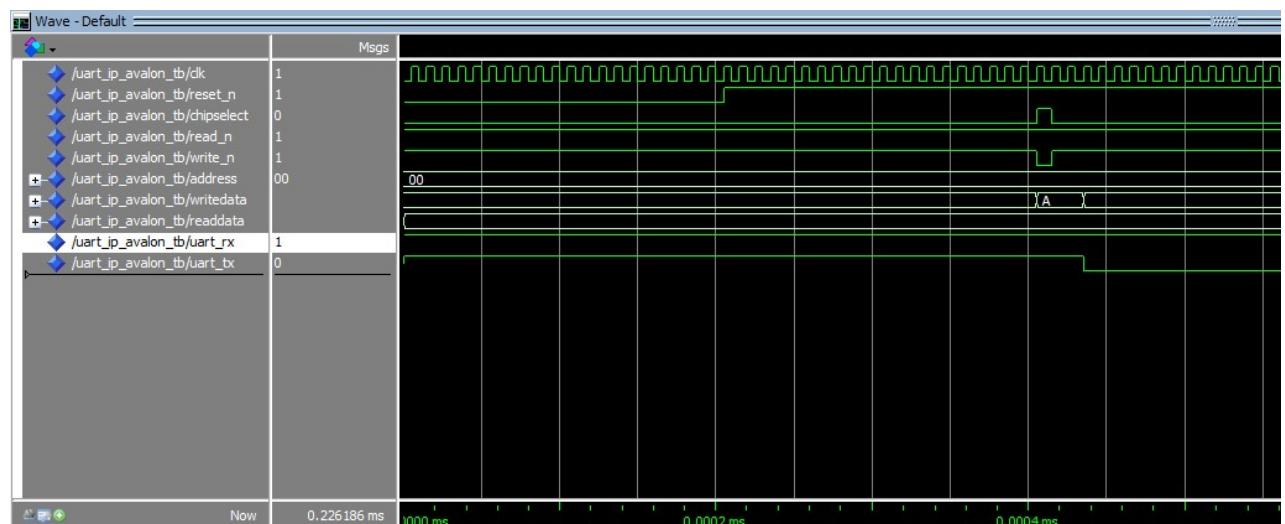


FIGURE 22 – Simulation de l'écriture Avalon vers l'IP UART : transmission du caractère 0x41 sur la ligne série

Dans un second temps, le banc de test simule la réception d'un caractère 'Z' (0x5A) sur la ligne `uart_rx`. Une fois la trame correctement reçue par le cœur UART, le flag `RX_VALID` est positionné. Le processeur simulé interroge alors le registre de statut jusqu'à détecter la disponibilité des données.

La Figure 23 illustre :

- la lecture Avalon du registre de statut ;
- la détection du bit `RX_VALID` ;
- la lecture du registre DATA, retournant la valeur 0x5A sur `readdata`.

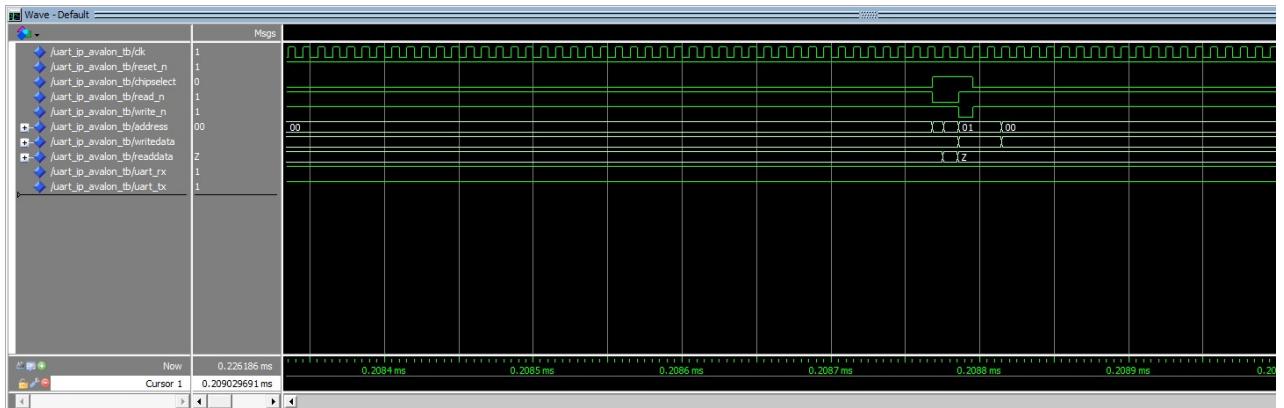


FIGURE 23 – Simulation de la lecture Avalon : récupération du caractère reçu 0x5A et validation via RX_VALID

Enfin, une écriture sur le registre de statut permet d’acquitter la réception, ce qui force la remise à zéro du flag RX_VALID. Une lecture ultérieure confirme que ce flag est correctement effacé, validant ainsi le mécanisme d’acquittement logiciel.

6.5.3 Validation expérimentale avec le processeur Nios II

Après la validation par simulation de l’interface Avalon, l’IP UART a été intégrée au système complet autour du processeur *Nios II*. Un programme en langage C a été exécuté sur le processeur afin de tester la communication série via des accès mémoire-mappés au périphérique UART.

Le programme réalise les opérations suivantes :

- affichage d’un message d’initialisation sur le terminal série au démarrage ;
- lecture bloquante des caractères reçus via l’IP UART Avalon ;
- retransmission immédiate des caractères reçus (*echo*).

Cette approche permet de valider simultanément :

- l’interface Avalon entre le Nios II et l’IP UART ;
- le fonctionnement du cœur UART en émission et en réception ;
- la cohérence entre le matériel et le logiciel.

La Figure 24 présente la sortie du terminal série sur le PC. Les chaînes de caractères saisies par l’utilisateur ("a", "123", "ok") sont correctement reçues par le processeur et renvoyées sans erreur, ce qui confirme le bon fonctionnement bidirectionnel de la communication UART à 115 200 bauds.

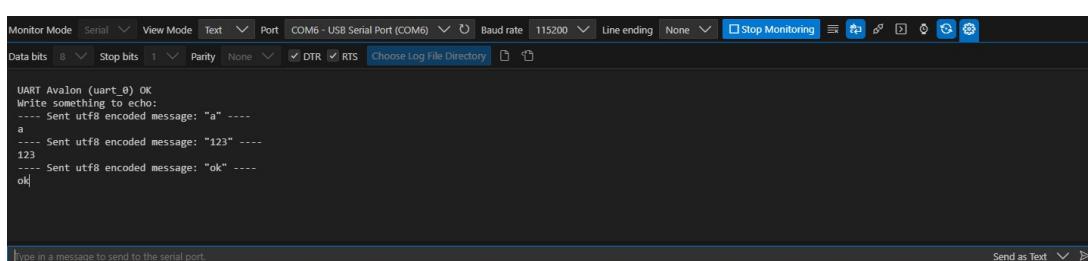


FIGURE 24 – Validation finale de l’IP UART Avalon avec le processeur Nios II : test d’écho via un terminal série

Ces résultats confirment que l'IP UART Avalon est pleinement fonctionnelle et correctement intégrée au système SoC-FPGA. Elle constitue désormais une interface fiable pour l'échange de données entre le radar ultrason et un ordinateur hôte, ainsi que pour l'envoi futur de commandes de configuration.

7 Radar – Intégration finale et commande via UART

Cette dernière étape consiste à intégrer l'ensemble des blocs développés au cours du projet dans une application unique exécutée sur le processeur *Nios II*. Le système final regroupe :

- l'IP **Télémètre** (lecture Avalon) pour la mesure de distance ;
- l'IP **Servomoteur** (écriture Avalon) pour le balayage angulaire ;
- le sous-système **VGA** pour l'affichage radar 2D ;
- l'IP **UART Avalon** pour la commande et la configuration depuis un PC.

L'objectif est de proposer une interface de contrôle simple permettant de configurer dynamiquement les paramètres de balayage, puis de lancer/arrêter le radar sans recompiler le programme.

7.1 Architecture logicielle : deux modes **CMD** et **RUN**

Le programme est organisé autour de deux états logiciels :

- **Mode CMD** : le système attend des commandes en UART (lecture bloquante) et permet de modifier les paramètres ;
- **Mode RUN** : le radar effectue le balayage angulaire, lit la distance, met à jour l'affichage VGA et les afficheurs 7 segments.

Le passage de **CMD** vers **RUN** se fait via la commande **RUN**. Pour garantir un arrêt robuste (sans dépendre d'une réception UART), le retour **RUN** vers **CMD** est assuré par un basculement matériel du switch **SW0**, selon une logique de type « va-et-vient » : au moment du lancement, l'état initial de **SW0** est mémorisé et l'arrêt est déclenché lorsque **SW0** prend l'état opposé.

À chaque transition vers **RUN**, le fond est redessiné (`draw_static()`) afin de supprimer les traces résiduelles avant un nouveau balayage. Lors du retour en **CMD** (message **STOP**), les afficheurs 7 segments sont réinitialisés à 000000 afin de signaler clairement que le système n'est plus en acquisition.

7.2 Interface de commande UART

La communication UART (115200 bauds, 8N1) permet de configurer le radar à l'aide d'un jeu de commandes volontairement minimaliste. Les retours sont courts afin de faciliter la lecture et l'automatisation des tests.

7.2.1 Commandes disponibles

- **HELP** : affiche l'aide et le format des commandes ;
- **P** : affiche les paramètres courants ;

- R <min_deg> <max_deg> : définit la plage angulaire du balayage ;
- S <step_deg> : définit le pas angulaire ;
- D <max_cm> : définit la portée maximale (mise à l'échelle de l'affichage) ;
- RUN : lance le balayage radar (basculement en **RUN**).

Les commandes invalides (commande inconnue ou paramètres absents/mal formés) renvoient une erreur explicite, ce qui permet de diagnostiquer rapidement un problème d'entrée.

7.3 Difficultés rencontrées et solutions retenues

L'intégration finale a mis en évidence plusieurs difficultés typiques des systèmes SoC-FPGA, en particulier lorsque des IP validées séparément sont regroupées dans une application unique.

7.3.1 Blocage en attente UART et impossibilité de démarrer le radar

Un premier problème est apparu lors de la fusion du mode commande UART avec la boucle de balayage : une lecture UART bloquante peut empêcher le radar de démarrer si l'utilisateur ne saisit aucun caractère, alors même que le démarrage devait être contrôlé par une entrée matérielle (interrupteur/bouton). Inversement, rendre la lecture UART non bloquante sans mécanisme de buffer pouvait conduire à des pertes de caractères et à une interprétation erronée des commandes.

La solution retenue a consisté à structurer le logiciel en deux états explicites : **CMD** (attente bloquante et traitement des commandes UART) et **RUN** (balayage radar). Ainsi, le mode commande reste entièrement bloquant (comportement robuste et déterministe), et le radar n'est jamais ralenti par un polling UART pendant l'acquisition.

7.3.2 Arrêt du radar sans FIFO : choix d'un contrôle matériel robuste

L'ajout d'une commande logicielle **STOP** a été envisagé, mais sans FIFO de réception, l'arrêt restait dépendant d'une réception fiable en UART pendant que le radar exécutait sa boucle (délais, pertes potentielles, ou synchronisation imparfaite). Afin d'obtenir un arrêt immédiat et indépendant de la communication série, un arrêt matériel a été préféré.

Le choix final repose sur **SW0** avec une logique de type « va-et-vient » : lors de la commande **RUN**, l'état initial du switch est mémorisé et l'arrêt est déclenché dès que **SW0** prend l'état opposé. Cette méthode garantit un arrêt systématique, sans ajout de FIFO ni d'interruptions.

7.3.3 Exemple de configuration et lancement

La Figure 25 montre un exemple de session utilisateur : définition de la plage angulaire, lecture des paramètres, modification du pas et de la portée, puis lancement du mode **RUN**. Le basculement de **SW0** déclenche ensuite l'arrêt (message **STOP**) et le retour en mode commande.

```

Radar ready. Use HELP.
---- Sent utf8 encoded message: "R 45 90\n" ----
OK
---- Sent utf8 encoded message: "P\n" ----
R 45 90 S 2 D 75
---- Sent utf8 encoded message: "S 5\n" ----
OK
---- Sent utf8 encoded message: "P\n" ----
R 45 90 S 5 D 75
---- Sent utf8 encoded message: "D 30\n" ----
OK
---- Sent utf8 encoded message: "RUN\n" ----
RUN
STOP
Radar ready. Use HELP.
|
    
```

FIGURE 25 – Exemple de commande UART : configuration des paramètres (R, S, D, P) puis lancement/arrêt (RUN, STOP)

Une fois lancé, le radar effectue un balayage continu sur la plage définie, en associant à chaque angle une distance mesurée par le télémètre. L'affichage VGA représente alors la scène sous forme d'un demi-disque, avec conservation des tracés afin de construire progressivement une cartographie. La Figure 26 illustre un résultat typique obtenu en conditions réelles, avec accumulation des balayages et apparition des zones correspondant aux obstacles détectés.

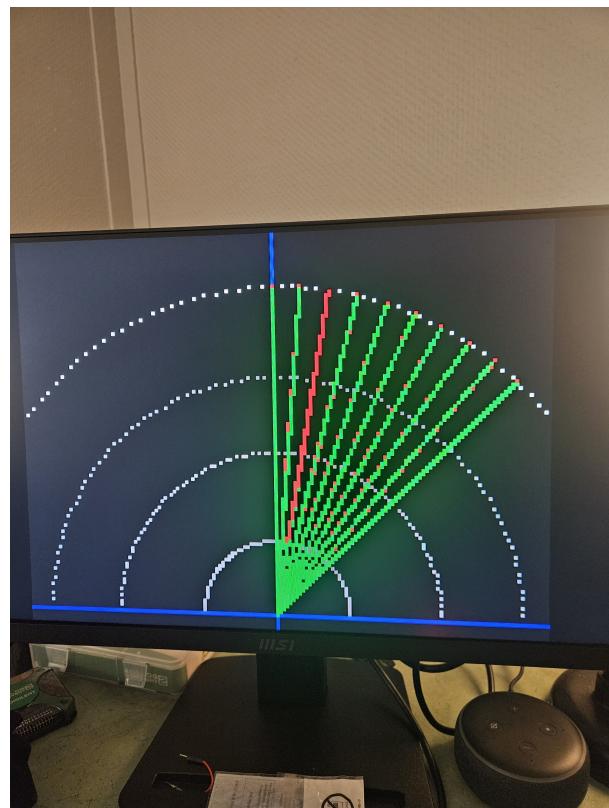


FIGURE 26 – Résultat expérimental : cartographie radar obtenue après plusieurs balayages (tracés persistants) et détection d'obstacles

Cette intégration finale valide le bon fonctionnement de la chaîne complète : commande utilisateur via UART, pilotage servomoteur, acquisition télémètre, affichage VGA

et signalisation via afficheurs 7 segments.

8 Conclusion

Ce projet a permis de concevoir et de valider un radar 2D complet basé sur une architecture SoC-FPGA autour du processeur *Nios II*. Les différentes IP développées (télémètre ultrason, servomoteur, UART et affichage VGA) ont d'abord été validées indépendamment, puis intégrées avec succès dans un système unique et cohérent.

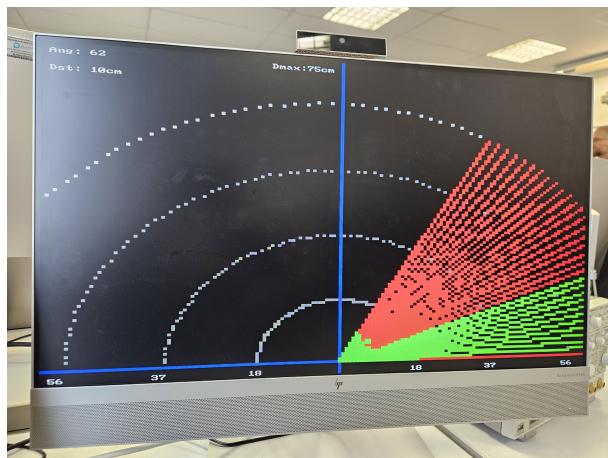


FIGURE 27 – Radar Final

L'intégration matérielle via le bus Avalon, associée à une gestion logicielle structurée en modes **CMD** et **RUN**, a permis d'obtenir un fonctionnement robuste et déterministe. Le pilotage du radar par commandes UART et l'arrêt matériel via interrupteur assurent une utilisation simple et fiable, tandis que l'affichage VGA fournit une représentation claire et intuitive de l'environnement mesuré.

Les résultats expérimentaux confirment la cohérence entre les distances réelles et les valeurs mesurées, ainsi que le bon fonctionnement de la chaîne complète. Le système obtenu constitue ainsi une démonstration aboutie des possibilités offertes par les architectures SoC-FPGA pour la réalisation de systèmes embarqués mêlant traitement matériel et logiciel.

```

1 #include <stdint.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include "system.h"
5 #include "io.h"
6 #include "unistd.h"

7
8 #define PIXEL_BUF_CTRL_BASE VGA_SUBSYSTEM_VGA_PIXEL_DMA_BASE
9 #define RGB_RESAMPLER_BASE
10    VGA_SUBSYSTEM_VGA_PIXEL_RGB_RESAMPLER_BASE
11 #define CHAR_BUF_BASE
12    VGA_SUBSYSTEM_CHAR_BUF_SUBSYSTEM_ONCHIP_SRAM_BASE

13 #define SCREEN_W 320
14 #define SCREEN_H 240
15 #define PI 3.14159265358979323846

16 #define BLACK 0x0000
17 #define GREEN 0x07E0
18 #define RED 0xF800
19 #define BLUE 0x001F
20 #define WHITE 0xFFFF
21 #define GRAY 0x8410

22
23 #define UART_DATA_OFFSET 0x00
24 #define UART_STATUS_OFFSET 0x04
25 #define UART_RX_VALID (1u << 0)
26 #define UART_TX_BUSY (1u << 1)
27 #define UART_RX_ACK (1u << 0)

28
29 static const uint8_t SEG7_LUT[10] = {
30     0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F
31 };
32
33 static int res_offset, col_offset;
34 static int cx, cy, radar_r;

35
36 static uint32_t angle_min = 0;
37 static uint32_t angle_max = 180;
38 static uint32_t step_deg = 2;
39 static uint32_t dist_max_cm = 75;

40
41 typedef enum { STATE_CMD = 0, STATE_RUN = 1 } radar_state_t;
42 static radar_state_t state = STATE_CMD;

43
44 static int stop_level_sw0 = 1;
45

```

```

46 static inline uint8_t seg7(uint8_t d) { return (d < 10u) ? SEG7_LUT
47     [d] : 0u; }
48
49 static inline void split3(uint32_t v, uint8_t *h, uint8_t *t,
50     uint8_t *u)
51 {
52     if (v > 999u) v = 999u;
53     *u = (uint8_t)(v % 10u);
54     *t = (uint8_t)((v / 10u) % 10u);
55     *h = (uint8_t)((v / 100u) % 10u);
56 }
57
58 static inline uint32_t clamp_u32(uint32_t v, uint32_t lo, uint32_t
59     hi)
60 {
61     if (v < lo) return lo;
62     if (v > hi) return hi;
63     return v;
64 }
65
66 static inline uint32_t pos_from_angle(uint32_t a)
67 {
68     a = clamp_u32(a, 0u, 180u);
69     return (a * 1023u) / 180u;
70 }
71
72 static inline int cm_to_px(int cm)
73 {
74     if (cm < 0) cm = 0;
75     int m = (int)dist_max_cm;
76     if (m < 1) m = 1;
77     if (cm > m) cm = m;
78     return (cm * radar_r) / m;
79 }
80
81 static inline uint32_t uart_status(void)
82 {
83     return IORD_32DIRECT(UART_0_BASE, UART_STATUS_OFFSET);
84 }
85
86 static void uart_putchar(char c)
87 {
88     while (uart_status() & UART_TX_BUSY) { }
89     IOWR_32DIRECT(UART_0_BASE, UART_DATA_OFFSET, (uint32_t)(uint8_t
90         )c);
91 }
92
93 static void uart_puts(const char *s)
94 {
95     while (*s) uart_putchar(*s++);

```

```

92 }
93
94 static char uart_getchar_blocking(void)
95 {
96     while ((uart_status() & UART_RX_VALID) == 0u) { }
97     char c = (char)(IORD_32DIRECT(UART_0_BASE, UART_DATA_OFFSET) &
98                     0xFFu);
99     IOWR_32DIRECT(UART_0_BASE, UART_STATUS_OFFSET, UART_RX_ACK);
100    return c;
101 }
102
103 static void uart_put_u32(uint32_t v)
104 {
105     char buf[11];
106     int i = 0;
107     if (v == 0u) { uart_putchar('0'); return; }
108     while (v && i < 10) {
109         buf[i++] = (char)('0' + (v % 10u));
110         v /= 10u;
111     }
112     while (i--) uart_putchar(buf[i]);
113 }
114
115 static int is_space(char c)
116 {
117     return (c == ' ' || c == '\t' || c == '\r' || c == '\n');
118 }
119
120 static char up(char c)
121 {
122     if (c >= 'a' && c <= 'z') return (char)(c - 'a' + 'A');
123     return c;
124 }
125
126 static inline int sw0_level(void)
127 {
128     uint32_t sw = IORD_32DIRECT(SLIDER_SWITCHES_BASE, 0);
129     return ((sw & 0x1u) != 0u);
130 }
131
132 static void msg_init(void) { uart_puts("Radar ready. Use HELP.\r\n");
133     }
134
135 static void reply_ok(void) { uart_puts("OK\r\n"); }
136 static void reply_run(void) { uart_puts("RUN\r\n"); }
137 static void reply_stop(void) { uart_puts("STOP\r\n"); }
138 static void reply_err(void) { uart_puts("ERR\r\n"); }
139 static void reply_unknown(void) { uart_puts("UNKNOWN\r\n"); }
140
141 static void clear_char_buffer(void)
142 {

```

```

140     volatile char *cb = (volatile char *)CHAR_BUF_BASE;
141     for (int y = 0; y < 60; y++) {
142         int off = (y << 7);
143         for (int x = 0; x < 128; x++) cb[off + x] = ' ';
144     }
145 }
146
147 static void video_text(int x, int y, const char *text)
148 {
149     volatile char *cb = (volatile char *)CHAR_BUF_BASE;
150     int off = (y << 7) + x; // 128 colunas (stride 128)
151
152     while (*text) {
153         cb[off++] = *text++;
154     }
155 }
156
157 // imprime texto em coordenadas de pixel (converte p/ char: 4x4 px
158 // por char)
159 static void video_text_px(int x_px, int y_px, const char *text)
160 {
161     int x = x_px / 4;
162     int y = y_px / 4;
163     if (x < 0) x = 0;
164     if (y < 0) y = 0;
165     if (x > 127) x = 127;
166     if (y > 59) y = 59;
167     video_text(x, y, text);
168 }
169
170 // imprime e "limpa o resto" com espacos (pra atualizar sem
171 // sujeira)
172 static void video_text_px_pad(int x_px, int y_px, const char *text,
173                             int pad_len)
174 {
175     char buf[64];
176     int i = 0;
177
178     // copia text
179     while (text[i] && i < (int)sizeof(buf) - 1) {
180         buf[i] = text[i];
181         i++;
182     }
183
184     // completa com espacos ate pad_len
185     while (i < pad_len && i < (int)sizeof(buf) - 1) {
186         buf[i++] = ' ';
187     }
188     buf[i] = '\0';
189 }
```

```

187     video_text_px(x_px, y_px, buf);
188 }
189
190
191 static void video_box(int x1, int y1, int x2, int y2, uint16_t
192 color)
193 {
194     int base = *(volatile int *)PIXEL_BUF_CTRL_BASE;
195     int xf = 1 << (res_offset + col_offset);
196     int yf = 1 << res_offset;
197
198     x1 /= xf; x2 /= xf;
199     y1 /= yf; y2 /= yf;
200
201     for (int row = y1; row <= y2; row++) {
202         for (int col = x1; col <= x2; col++) {
203             int ptr = base + (row << (10 - res_offset - col_offset)
204                             ) + (col << 1);
205             *(volatile uint16_t *)ptr = color;
206         }
207     }
208
209 static void plot(int x, int y, uint16_t c)
210 {
211     if (x < 0 || x >= SCREEN_W || y < 0 || y >= SCREEN_H) return;
212
213     int base = *(volatile int *)PIXEL_BUF_CTRL_BASE;
214     int xf = 1 << (res_offset + col_offset);
215     int yf = 1 << res_offset;
216
217     x /= xf;
218     y /= yf;
219
220     int ptr = base + (y << (10 - res_offset - col_offset)) + (x <<
221                     1);
222     *(volatile uint16_t *)ptr = c;
223 }
224
225 static void draw_line(int x0, int y0, int x1, int y1, uint16_t c)
226 {
227     int dx = (x1 > x0) ? (x1 - x0) : (x0 - x1);
228     int sx = (x0 < x1) ? 1 : -1;
229     int dy = (y1 > y0) ? -(y1 - y0) : -(y0 - y1);
230     int sy = (y0 < y1) ? 1 : -1;
231     int err = dx + dy;
232
233     for (;;) {
234         plot(x0, y0, c);
235         if (x0 == x1 && y0 == y1) break;

```

```

234         int e2 = err << 1;
235         if (e2 >= dy) { err += dy; x0 += sx; }
236         if (e2 <= dx) { err += dx; y0 += sy; }
237     }
238 }
239
240 static void draw_arc_cm(int cm, uint16_t c)
241 {
242     int r = cm_to_px(cm);
243     for (int a = 0; a <= 180; a += 2) {
244         double rad = (double)a * PI / 180.0;
245         int x = cx + (int)(cos(rad) * r);
246         int y = cy - (int)(sin(rad) * r);
247         plot(x, y, c);
248     }
249 }
250
251 static void draw_live_status(uint32_t angle_deg, uint32_t dist_cm)
252 {
253     char s1[32], s2[32];
254
255     // exemplo: "Ang: 123"
256     sprintf(s1, sizeof(s1), "Ang:%3u", (unsigned)angle_deg);
257     sprintf(s2, sizeof(s2), "Dst:%3ucm", (unsigned)dist_cm);
258
259     // topo esquerdo
260     video_text_px_pad(4, 4, s1, 12);
261     video_text_px_pad(4, 16, s2, 12);
262 }
263
264 static void draw_range_labels(void)
265 {
266     int m = (int)dist_max_cm;
267     int d1 = (m * 1) / 4;
268     int d2 = (m * 2) / 4;
269     int d3 = (m * 3) / 4;
270     if (d1 < 1) d1 = 1;
271
272     // Dmax no topo (centro)
273     {
274         char s[32];
275         sprintf(s, sizeof(s), "Dmax:%ucm", (unsigned)m);
276         video_text_px_pad(cx - 40, 4, s, 16);
277     }
278
279     // Eixo X: somente 1/4, 1/2, 3/4 (sem o m ximo)
280     int marks[3] = { d1, d2, d3 };
281
282     // linha um pouquinho abaixo do eixo
283     int y_px = cy + 6;

```

```

284
285     for (int i = 0; i < 3; i++) {
286         int r = cm_to_px(marks[i]);
287
288         char s[8];
289         sprintf(s, sizeof(s), "%d", marks[i]);
290
291         // largura em pixels aproximada do texto no char buffer:
292         // 1 char ~ 4 px (porque usamos /4)
293         int len = 0;
294         while (s[len]) len++;
295         int text_w_px = len * 4;
296
297         // quer posicionar o texto centralizado na marca:
298         // x_center - metade da largura
299         int x_right = (cx + r) - (text_w_px / 2);
300         int x_left = (cx - r) - (text_w_px / 2);
301
302         // clamp para não sair da tela (deixa 1 char de margem)
303         int min_x = 4; // 1 char
304         int max_x = SCREEN_W - text_w_px - 4;
305
306         if (x_right < min_x) x_right = min_x;
307         if (x_right > max_x) x_right = max_x;
308
309         if (x_left < min_x) x_left = min_x;
310         if (x_left > max_x) x_left = max_x;
311
312         // escreve nos dois lados
313         video_text_px_pad(x_right, y_px, s, 5);
314         video_text_px_pad(x_left, y_px, s, 5);
315     }
316 }
317
318
319
320
321 static void draw_static(void)
322 {
323     video_box(0, 0, SCREEN_W - 1, SCREEN_H - 1, BLACK);
324     clear_char_buffer();
325
326     int m = (int)dist_max_cm;
327     int a1 = (m * 1) / 4;
328     int a2 = (m * 2) / 4;
329     int a3 = (m * 3) / 4;
330     if (a1 < 1) a1 = 1;
331
332     draw_arc_cm(m, WHITE);
333     draw_arc_cm(a3, GRAY);

```

```

334     draw_arc_cm(a2, GRAY);
335     draw_arc_cm(a1, GRAY);
336
337     draw_line(0, cy, SCREEN_W - 1, cy, BLUE);
338     draw_line(cx, 0, cx, SCREEN_H - 1, BLUE);
339     draw_range_labels();
340 }
341
342 static void print_help(void)
343 {
344     uart_puts("HELP\r\n");
345     uart_puts("  HELP\r\n");
346     uart_puts("  P\r\n");
347     uart_puts("  R<min_deg><max_deg>\r\n");
348     uart_puts("  S<step_deg>\r\n");
349     uart_puts("  D<max_cm>\r\n");
350     uart_puts("  RUN\r\n");
351     uart_puts("  (stop:toggle SW0)\r\n");
352 }
353
354 static void print_params(void)
355 {
356     uart_puts("R");
357     uart_put_u32(angle_min);
358     uart_putchar(' ');
359     uart_put_u32(angle_max);
360     uart_puts("S");
361     uart_put_u32(step_deg);
362     uart_puts("D");
363     uart_put_u32(dist_max_cm);
364     uart_puts("\r\n");
365 }
366
367 static void skip_spaces(char **pp)
368 {
369     while (**pp && is_space(**pp)) (*pp)++;
370 }
371
372 static uint32_t parse_u32_adv(char **pp, int *ok)
373 {
374     uint32_t v = 0;
375     int any = 0;
376
377     skip_spaces(pp);
378
379     while (**pp >= '0' && **pp <= '9') {
380         any = 1;
381         v = (uint32_t)(v * 10u + (uint32_t)(**pp - '0'));
382         (*pp)++;
383     }

```

```

384     *ok = any;
385     return v;
386 }
387
388
389 static int is_run_cmd(const char *ln)
390 {
391     const char *p = ln;
392     while (*p && is_space(*p)) p++;
393     if (up(p[0]) != 'R') return 0;
394     if (up(p[1]) != 'U') return 0;
395     if (up(p[2]) != 'N') return 0;
396     if (p[3] != '\0' && !is_space(p[3])) return 0;
397     return 1;
398 }
399
400
401 static int is_help_cmd(const char *ln)
402 {
403     const char *p = ln;
404     while (*p && is_space(*p)) p++;
405     return (up(p[0]) == 'H' && up(p[1]) == 'E' && up(p[2]) == 'L'
406         && up(p[3]) == 'P' &&
407             (p[4] == '\0' || is_space(p[4])));
408 }
409
410
411 static int is_p_cmd(const char *ln)
412 {
413     const char *p = ln;
414     while (*p && is_space(*p)) p++;
415     return (up(p[0]) == 'P' && (p[1] == '\0' || is_space(p[1])));
416 }
417
418
419 static void handle_cmd_line(char *ln)
420 {
421     char *p = ln;
422     while (*p && is_space(*p)) p++;
423     if (*p == '\0') return;
424
425     char cmd0 = up(*p);
426
427     if (cmd0 == 'H') {
428         if (is_help_cmd(p)) { print_help(); return; }
429         reply_unknown();
430         return;
431     }
432
433     if (cmd0 == 'P') {
434         if (is_p_cmd(p)) { print_params(); return; }
435         reply_unknown();
436         return;
437     }
438 }
```

```

433     }
434
435     if (cmd0 == 'R') {
436         p++;
437         int ok1 = 0, ok2 = 0;
438         uint32_t a = parse_u32_adv(&p, &ok1);
439         uint32_t b = parse_u32_adv(&p, &ok2);
440         if (!(ok1 && ok2)) { reply_err(); return; }
441         a = clamp_u32(a, 0u, 180u);
442         b = clamp_u32(b, 0u, 180u);
443         if (a > b) { uint32_t t = a; a = b; b = t; }
444         angle_min = a;
445         angle_max = b;
446         reply_ok();
447         return;
448     }
449
450     if (cmd0 == 'S') {
451         p++;
452         int ok1 = 0;
453         uint32_t s = parse_u32_adv(&p, &ok1);
454         if (!ok1) { reply_err(); return; }
455         step_deg = clamp_u32(s, 1u, 30u);
456         reply_ok();
457         return;
458     }
459
460     if (cmd0 == 'D') {
461         p++;
462         int ok1 = 0;
463         uint32_t d = parse_u32_adv(&p, &ok1);
464         if (!ok1) { reply_err(); return; }
465         dist_max_cm = clamp_u32(d, 1u, 200u);
466         draw_static();
467         reply_ok();
468         return;
469     }
470
471     reply_unknown();
472 }
473
474 static void cmd_wait_and_process(void)
475 {
476     static char ln[64];
477     int i = 0;
478
479     for (;;) {
480         char c = uart_getchar_blocking();
481
482         if (c == '\r' || c == '\n') {

```

```

483         ln[i] = '\0';
484
485     if (is_run_cmd(ln)) {
486         int start_level = sw0_level();
487         stop_level_sw0 = start_level ? 0 : 1;
488         draw_static();
489         state = STATE_RUN;
490         reply_run();
491         return;
492     }
493
494     handle_cmd_line(ln);
495     i = 0;
496 } else {
497     if (i < (int)sizeof(ln) - 1) ln[i++] = c;
498 }
499 }
500 }
501
502 static void init_hex_displays_zero(void)
503 {
504     uint32_t z = seg7(0);
505     IOWR_32DIRECT(HEX3_HEX0_BASE, 0,
506                 (z << 0) | (z << 8) | (z << 16) | (z << 24));
507     IOWR_32DIRECT(HEX5_HEX4_BASE, 0,
508                 (z << 0) | (z << 8));
509 }
510
511 static void run_step(uint32_t *angle, int *dir)
512 {
513     if (*angle < angle_min) *angle = angle_min;
514     if (*angle > angle_max) *angle = angle_max;
515
516     IOWR_32DIRECT(SERVOMOTEUR_0_BASE, 0, pos_from_angle(*angle));
517     usleep(60000);
518
519     uint32_t raw = IORD_32DIRECT(TELEMETRE_0_BASE, 0);
520     uint32_t dist = raw & 0x3FFu;
521     if (dist > dist_max_cm) dist = dist_max_cm;
522
523     draw_live_status(*angle, dist);
524
525     double rad = (double)(*angle) * PI / 180.0;
526     int r_obj = cm_to_px((int)dist);
527
528     int x_obj = cx + (int)(cos(rad) * r_obj);
529     int y_obj = cy - (int)(sin(rad) * r_obj);
530     int x_max = cx + (int)(cos(rad) * radar_r);
531     int y_max = cy - (int)(sin(rad) * radar_r);
532 }
```

```

533     draw_line(cx, cy, x_obj, y_obj, GREEN);
534     draw_line(x_obj, y_obj, x_max, y_max, RED);
535
536     uint8_t ah, at, au, dh, dt, du;
537     split3(*angle, &ah, &at, &au);
538     split3(dist, &dh, &dt, &du);
539
540     IOWR_32DIRECT(HEX3_HEX0_BASE, 0,
541         ((uint32_t)seg7(au) << 0) |
542         ((uint32_t)seg7(at) << 8) |
543         ((uint32_t)seg7(ah) << 16) |
544         ((uint32_t)seg7(du) << 24));
545
546     IOWR_32DIRECT(HEX5_HEX4_BASE, 0,
547         ((uint32_t)seg7(dt) << 0) |
548         ((uint32_t)seg7(dh) << 8));
549
550     if (*dir > 0) {
551         if (*angle + step_deg >= angle_max) { *angle = angle_max; *
552             dir = -1; }
553         else *angle += step_deg;
554     } else {
555         if (*angle <= angle_min + step_deg) { *angle = angle_min; *
556             dir = +1; }
557         else *angle -= step_deg;
558     }
559     usleep(40000);
560 }
561
562 int main(void)
563 {
564     volatile int *res = (int *) (PIXEL_BUF_CTRL_BASE + 8);
565     int sx = (*res) & 0xFFFF;
566
567     volatile int *rgb = (int *) (RGB_RESAMPLER_BASE);
568     int db = (*rgb) & 0x3F;
569
570     res_offset = (sx == 160) ? 1 : 0;
571     col_offset = (db == 8) ? 1 : 0;
572
573     cx = SCREEN_W / 2;
574     cy = SCREEN_H - 10;
575     radar_r = 200;
576
577     draw_static();
578     init_hex_displays_zero();
579     msg_init();
580
581     uint32_t angle = 0;

```

```
581     int dir = +1;
582
583     for (;;) {
584         if (state == STATE_CMD) {
585             cmd_wait_and_process();
586             continue;
587         }
588
589         if (sw0_level() == stop_level_sw0) {
590             state = STATE_CMD;
591             reply_stop();
592             init_hex_displays_zero();
593             msg_init();
594             continue;
595         }
596
597         run_step(&angle, &dir);
598     }
599 }
```

Listing 1 – Programme Nios II – Radar Complet