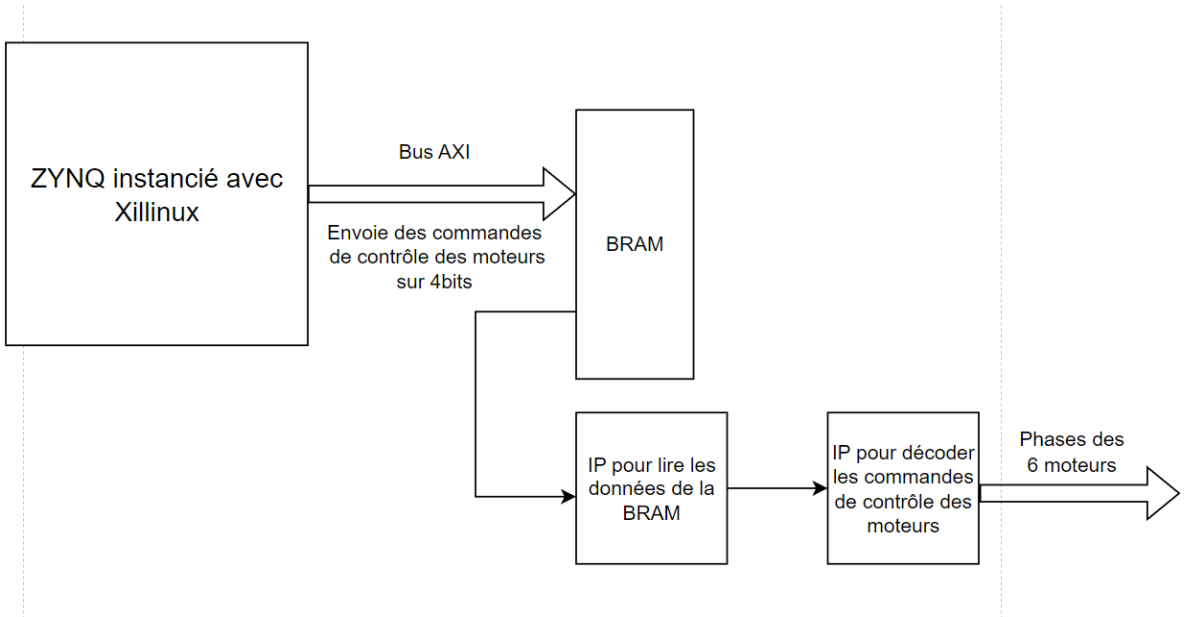
# Développement interface Software/FPGA et contrôle des moteurs

Cette partie du projet a été brillamment réalisée par LE GOAT — j'ai nommé Mateo Bertolelli**.**
Elle joue un rôle clé, puisqu'elle assure la liaison entre l'algorithme de résolution du Rubik's Cube et les moteurs chargés d'exécuter physiquement les mouvements.
Étant située en bout de chaîne du développement, cette phase a d'abord été entièrement simulée, faute de disponibilité de Xillinux sur la carte Zybo lors des premiers tests.

Lors de la phase initiale du projet, l'architecture que nous avons choisie se présentait comme suit :



Cette architecture nécessite le développement de deux IP personnalisées :

- La première IP (**Memory Reader**) est chargée de lire les données stockées par le Zynq dans la BRAM, puis de les transmettre à la seconde IP.
- La seconde IP (**Rubik resolver**), quant à elle, a pour mission de décoder les commandes extraites de la BRAM afin de piloter les moteurs associés.

Pour cela, il a d'abord été nécessaire d'étudier le fonctionnement de l'IP Xilinx "BRAM" : quels sont les signaux qu'elle attend en entrée, comment effectuer une lecture ou une écriture, etc...

# 1. Etude IP constructeur BRAM

**L'IP BRAM (Block RAM)** proposée par Xilinx permet d'accéder à une mémoire synchrone embarquée dans le FPGA. Elle peut être configurée de différentes manières, selon les besoins de l'architecture :

- **Single-Port** : la BRAM dispose d'un seul port d'accès (lecture ou écriture à chaque cycle).
- **Dual-Port** : deux ports indépendants permettent d'effectuer des lectures/écritures simultanées, ce qui offre plus de flexibilité pour des architectures parallèles.

Signaux principaux :

| Signal | Direction | Taille (bit) | Description |
|---|---|---|---|
| clka | In | 1 | Horloge |
| Ena | In | 1 | Signal d'activation de la BRAM. Doit être a '1' pour autoriser lecture comme écriture |
| Wea | In | 4 | Bus de validité de l'écriture. Chaque bit à '1' correspond à 1 octets des données d'écriture valide. |
| Addra | In | 32 | Adresse de la mémoire. Sert de pointeur pour la lecture/ecriture |
| Dina | In | 32 | Données à écrire dans la mémoire |
| Douta | Out (32 bits) | 32 | Données à lire dans la mémoire |

Comportement général :

- Une écriture se fait en présentant une adresse (addra), des données (dina), et en activant ena et les bits de wea correspondant aux octets visés.

- Une lecture se fait en présentant simplement l'adresse sur addra, avec ena = 1 et wea = 0 (aucune écriture). La donnée est disponible sur douta au cycle suivant (mémoire synchrone).

- Le signal clka rythme l'ensemble : toute opération (lecture ou écriture) est cadencée par le front montant de l'horloge.

Détails importants :

- L'adresse est alignée sur les mots (par exemple 32 bits = adresses multiples de 4).

- Les signaux wea, ena et clka doivent être parfaitement synchronisés pour éviter des lectures/écritures invalides.

- En configuration **Dual-Port**, on dispose d'un second ensemble de signaux (clkb, enb, web, etc.) pour le port B.

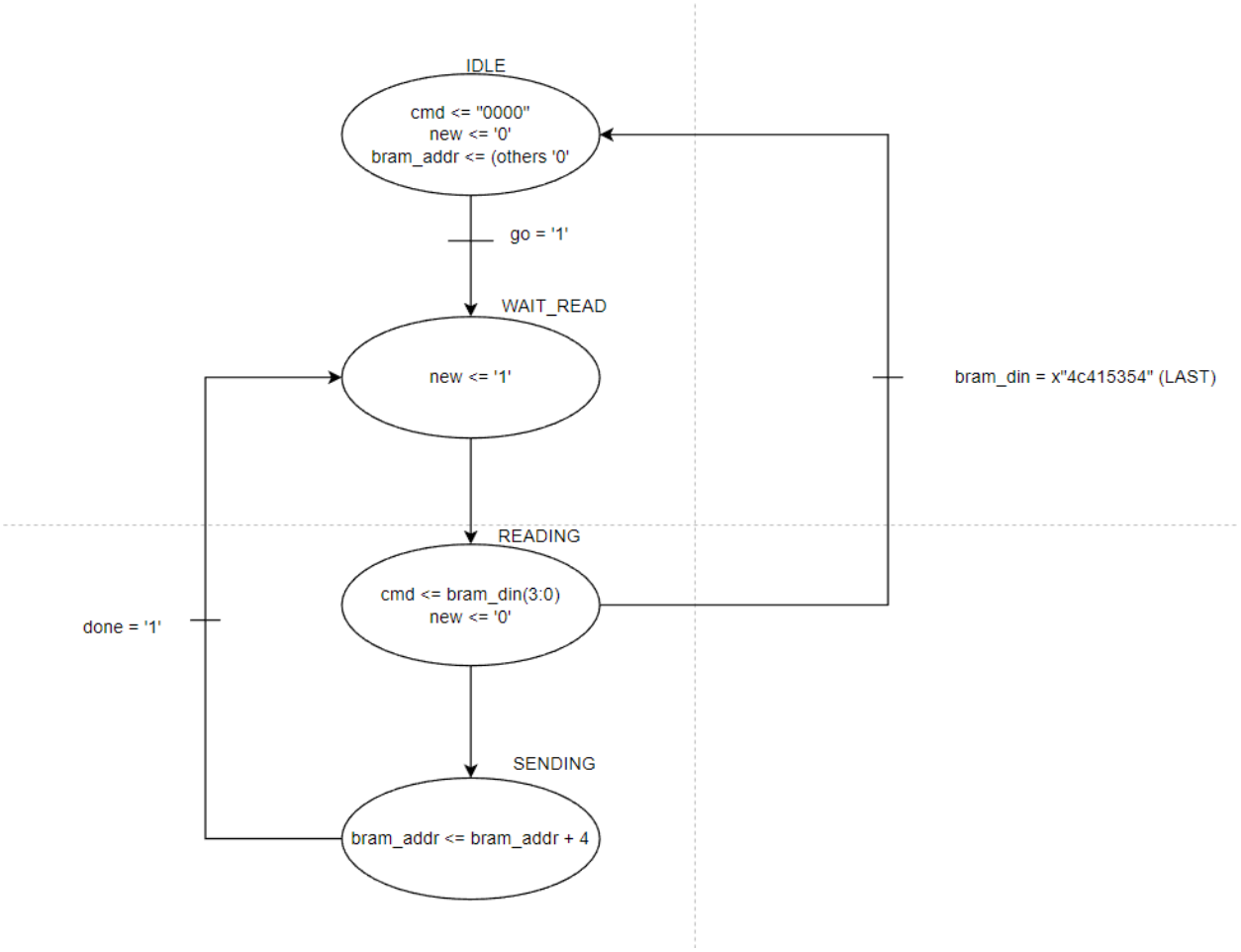
## 2. Développement IP custom “Memory Reader”

Une fois ces informations maîtrisées, nous avons pu développer notre IP « **Memory Reader** ». Comme dit précédemment, elle a pour rôle principal d’accéder aux données stockées dans la BRAM afin de les transmettre à l’IP suivante pour décodage et exécution.

Les principaux signaux de cette IP sont :

| Signal | Direction | Taille (bit) | Description |
|---|---|---|---|
| Clk | In | 1 | Horloge |
| Rst | In | 1 | Reset Système |
| Go | In | 1 | Déclencheur de démarrage de ma machine à état (Associé a un bouton de la carte) |
| Done | In | 1 | Reçu de Rubik Resolver, permet de passer à la lecture de là donnée contenu à l’adresse suivante |
| Cmd | Out | 4 | Commande extraite des 4 bits de poids faible de la donnée lue en BRAM |
| New | Out | 1 | Signal pour indiquer au Rubik Resolver qu’un nouvelle commande arrive |
| Bram_addr | Out | 32 | Adresse de l’lecture de la BRAM |
| Bram_clk | Out | 1 | Signal de clk envoyé à la BRAM pour synchroniser les échanges |
| Bram_din | In | 32 | Données en provenance de la BRAM (contient les commandes) |
| Bram_dout | Out | 32 | Donnée à écrire dans la BRAM (Jamais utilisé ici) |
| Bram_en | Out | 1 | Signal d’activation de la BRAM |
| Bram_rst | Out | 1 | Signal de Reset de la BRAM |
| Bram_we | Out | 4 | Bus de validité de l’écriture (toujours à 0 puisqu’aucune écriture n’est effectuée) |

Cette IP est une machine à état de la forme suivante :




Cette machine à états assure ainsi une lecture séquentielle des commandes depuis la BRAM, en s'arrêtant automatiquement dès qu'elle détecte le mot-clé spécial « LAST », signalant la fin des instructions à transmettre.

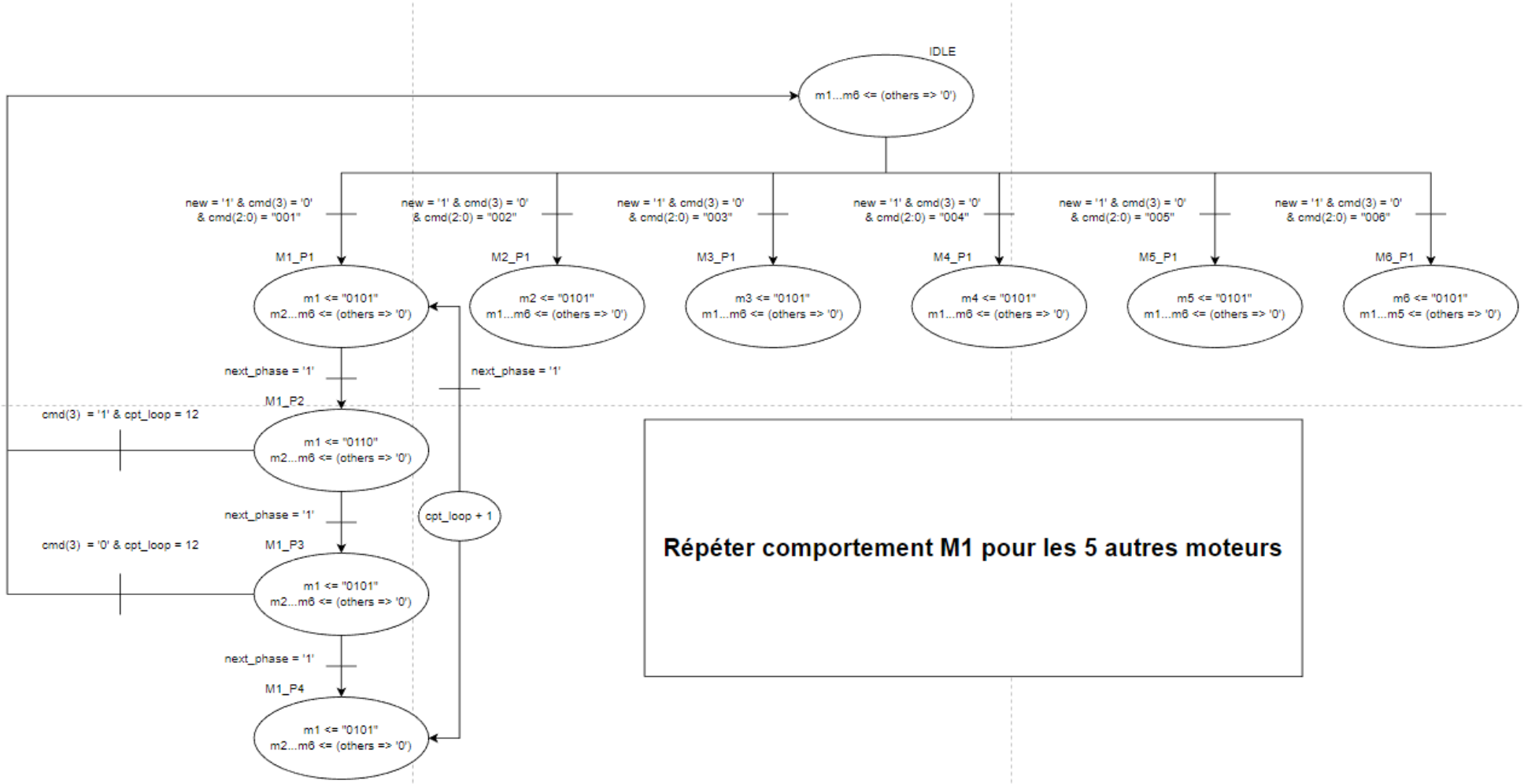
# 3. Développement IP custom “Rubik Resolver”

Ce composant reçoit, à chaque nouvelle commande 4 bits (cmd) avec un signal de validation (new), un code moteur (1…6) et une direction (bit MSB). Il pilote six sorties moteurs (o_m1…o_m6), chacune codée sur 4 bits pour générer les quarts de pas successifs. Une FSM interne passe par quatre phases (P1→P4 ou P4→P1 en fonction de la direction) pour chaque moteur, jusqu’à satisfaction du nombre de pas à effectuer. Chaque phase doit durer un temps donné donc un compteur a été mis en place afin d’indiquer a la FSM quand passer à la phase suivante du moteur.

Les principaux signaux de cette IP :

| Signal | Direction | Taille (bit) | Description |
|---|---|---|---|
| Clk | In | 1 | Horloge |
| Rst | In | 1 | Reset Système |
| Done | Out | 1 | ‘1’ lorsque la séquence (nombre de tours ou pas désirés) est terminée et que la FSM retourne en IDLE |
| Cmd | In | 4 | Code de sélection de moteur + direction : bit[3]=direction (0=sens trigonométrique,1=inverse), bits[2:0]=1…6 |
| New | In | 1 | Impulsion de lancement : déclenche la lecture de cmd et le démarrage de la séquence mécanique |
| m1…m6 | Out | 4 chacun | Sorties 4 bits envoyées à un driver pas-à-pas (quart de pas 1→2→3→4), l’une par moteur |
| Leds | Out | 4 | Affichage directe de cmd pour débogage visuel |

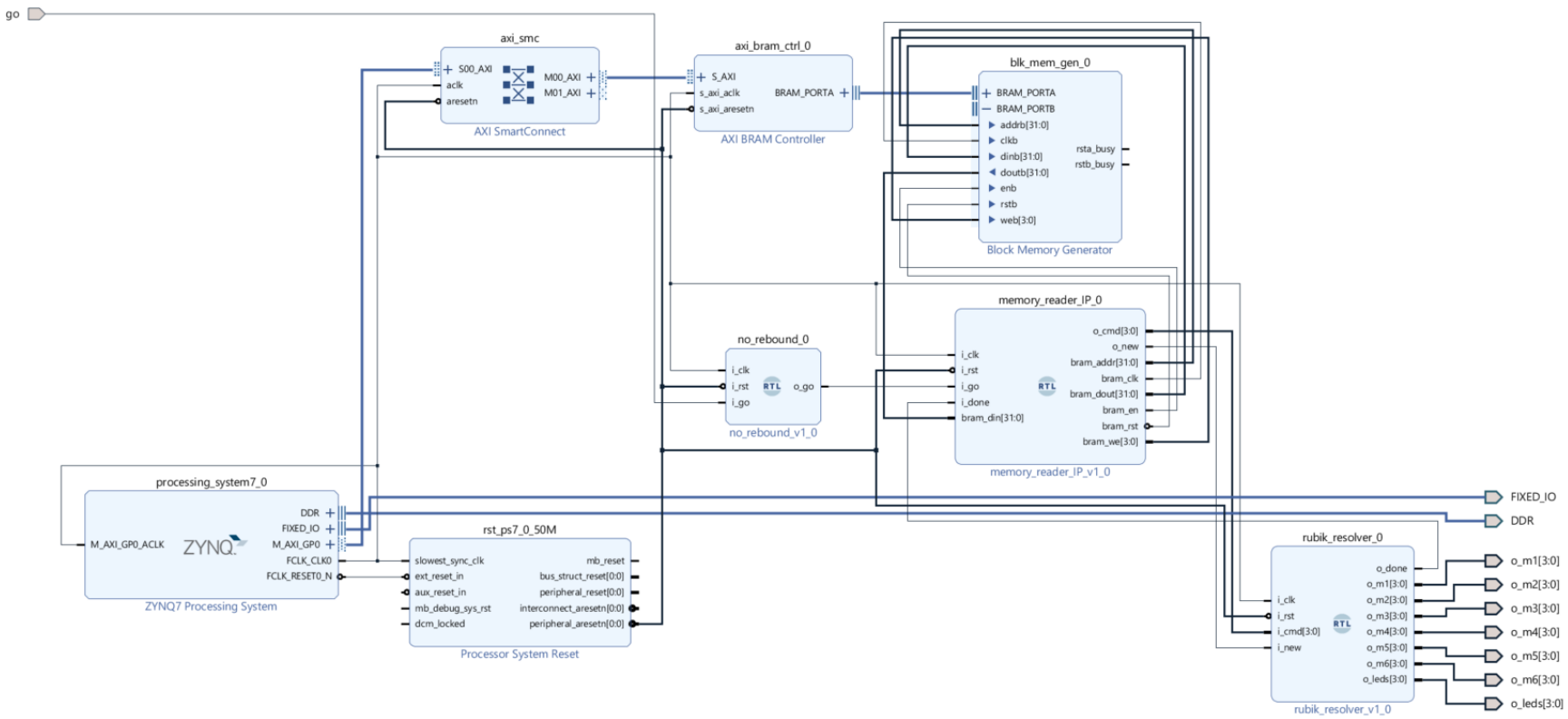
Cette IP est elle aussi une grande Machine a état de la forme :




Cette conception garantit un pilotage temps-réel des moteurs pas à pas, configurable dynamiquement par la logique extérieure via cmd/new, sans nécessiter de micro-contrôleur.

## 4. Design final sur Vivado

Dans cette section, nous présentons le design final implémenté et validé sous Vivado. Après avoir conçu et testé successivement chaque bloc fonctionnel (lecteur de mémoire, comparateur VCD, pilote de moteurs pas-à-pas, etc.), l'ensemble a été intégré dans un projet Vivado unique. Celui-ci se présente sous la forme suivante :



Dans ce design nous pouvons retrouver les blocs suivants :

### 1. processing_system7_0 (ZYNQ7 Processing System)

Ce bloc représente le processeur ARM Cortex-A9 du SoC Zynq.
Il est responsable :

- de la gestion générale du système (RAM, IO, FCLK)
- du transfert des données via AXI vers la BRAM

---

### 2. axi_smc (AXI SmartConnect)

Il s'agit du switch AXI principal permettant de router les transactions entre le processeur et les périphériques AXI comme le contrôleur de BRAM.
Il connecte :

- le port M_AXI_GP0 du Zynq
- au axi_bram_ctrl_0

---

### 3. axi_bram_ctrl_0 (AXI BRAM Controller)

Ce contrôleur fait l'interface entre le bus AXI et la mémoire BRAM.
Il reçoit les commandes de lecture/écriture de la part du PS et accède physiquement à la **Block Memory Generator**.

---

### 4. blk_mem_gen_0 (Block Memory Generator)

Il s'agit de la BRAM elle-même (générée avec l'IP BRAM de Xilinx).
Elle utilise la configuration **Dual-Port**:

- **PORTA** utilisé par le contrôleur AXI (écriture des données via le PS)
- **PORTB** utilisé par notre IP personnalisée memory_reader pour lecture directe dans le tissu logique (PL)

---

## 5. no_rebound_0

Petit module RTL permettant de générer une impulsion propre et stable (o_go) à partir d'un signal de type bouton ou commutateur (i_go).
Cela évite les rebonds lors de la mise à 1 du signal go manuel.

---

## 6. memory_reader_IP_0

IP personnalisée développée pour ce projet.
Elle lit séquentiellement les données de la BRAM (port B), détecte le mot-clé LAST, et envoie chaque commande décodée au bloc de contrôle moteur (rubik_resolver).

---

## 7. rubik_resolver_0

Deuxième IP développée dans le projet.
Elle interprète chaque commande reçue pour piloter un ou plusieurs des **6 moteurs** du cube (via o_m1 à o_m6).
Chaque moteur reçoit un signal de phase 4 bits, correspondant au pas à envoyer à son driver.

## 8. IN/OUT

Enfin, le signal go est directement relié au bouton utilisateur n°0 de la carte Zybo, permettant de déclencher manuellement la lecture des commandes.

Les sorties o_m1 à o_m6, quant à elles, sont connectées aux broches des ports PMOD de la carte, permettant ainsi le pilotage physique des six moteurs du Rubik's Cube via des drivers externes.

## 5. Chargement de commandes en BRAM via un script

Dans le but de tester le bon fonctionnement du système global, un programme en langage C a été rédigé, celui-ci permettant de lire un fichier texte contenant une suite de commandes codées sur 4 bits. À la fin du fichier, une séquence spéciale représentant le mot-clé LAST était ajoutée afin de signaler la fin des données. Le programme lit ces informations, les convertit en valeurs numériques, puis les écrit séquentiellement dans la mémoire BRAM de la carte. Pour cela, nous avons utilisé la fonction mmap (qui permet de mapper une zone de la mémoire physique dans l'espace mémoire du programme utilisateur), afin d'accéder directement à la BRAM via /dev/mem. Cette approche nous a permis de simuler un enchaînement complet de commandes, comme le système devra les traiter pendant le fonctionnement final.