# PROCESSEUR MONO-CYCLE

**Un compte-rendu global est demandé pour ce projet**

L'objectif de ces séances est de concevoir et de simuler le cœur d'un processeur. Ce processeur sera assemblé en utilisant des composants de base tels que des registres, des multiplexeurs, des bancs de mémoire, et une Unité Arithmétique et Logique. Ces éléments seront intégrés pour former les différents blocs du système, notamment l'unité de traitement, l'unité de gestion des instructions, et l'unité de contrôle. Le fonctionnement du processeur sera d'abord vérifié par la simulation de l'exécution d'un programme test simple, puis testé sur un FPGA.

Pour chaque conception, les circuits seront décrits en VHDL comportemental et simulés à l'aide de Modelsim (ou GHDL/GTKwave), en utilisant un banc de test spécialement développé pour cela.

Compétences :

- **Rendre compte de votre travail par la rédaction d'un compte rendu de projet**
- **Concevoir, synthétiser et valider une IP numérique décrite en VHDL RTL**

## PARTIE 1 – UNITE DE TRAITEMENT

L'unité de traitement du processeur est principalement composée d'une Unité Arithmétique et Logique, d'un banc de registres et d'une mémoire de données, ainsi que de composants annexes (multiplexeurs, extension de signe…)

### Unité Arithmétique et Logique (UAL ou ALU)

L'Unité Arithmétique et Logique (UAL) 32 bits possède :

- OP : Signal de commande sur 2 bits
- A, B : Deux bus 32 bits en entrée
- S : Bus 32 bits en sortie
- N : drapeau de sortie sur 1 bit signifiant que le résultat de la dernière opération est négatif
- Z : drapeau de sortie sur 1 bit signifiant que le résultat de la dernière opération est égale à zéro

Les valeurs du signal de commande OP déterminent les opérations implémentées dans l'UAL :

- ADD : Y = A + B; OP = "00";
- B : Y = B; OP = "01";
- SUB : Y = A - B; OP = "10";
- A : Y = A; OP = "11";

A l'issue de chaque opération :

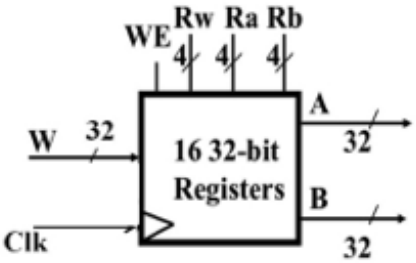
- le drapeau N prend la valeur 1 si le résultat est strictement Négatif, et 0 sinon.
- le drapeau Z prend la valeur 1 si le résultat est strictement égal à zéro, et 0 sinon.

*Conseil : La librairie ieee.numeric_std.all permet d'utiliser les opérateurs '+' et '-'*

# Ban de registre

Le banc possède 16 registres de 32 bits. Il comporte les entrées/ sorties suivantes :

- CLK: Horloge,
- **Reset** : reset asynchrone (actif à l'état haut) non représenté sur le schéma
- W: Bus de donnés en écriture sur 32 bits
- RA: Bus d'adresses en lecture du port A sur 4 bits
- RB: Bus d'adresses en lecture du port B sur 4 bits
- RW: Bus d'adresses en écriture sur 4 bits
- WE: Write Enable sur 1 bit
- A: Bus de données en lecture du port A
- B: Bus de données en lecture du port B




La **lecture** des registres se fait de manière **asynchrone et simultanée** :

- Le bus de sortie A prend la valeur du registre N° RA
- Le bus de sortie B prend la valeur du registre N° RB

L'**écriture** des registres se fait de manière **synchrone**, sur le front montant de l'horloge. Elle est commandée par le signal WE :

- Si WE = 1, au front montant on recopie la valeur du bus W dans le registre n° RW
- Si WE = 0, au front montant il n'y a pas d'écriture effectuée

## 1.1 Travail à réaliser

***1.1.1 Décrire et simuler ces modules en VHDL comportemental à l'aide d'un banc de test et un script de simulation. Reporter les résultats de simulations sur votre compte-rendu.***

Le banc de registres sera déclaré comme étant un tableau de std_logic_vector. Son initialisation sera réalisée à l'aide d'une fonction (voir code ci-dessous)

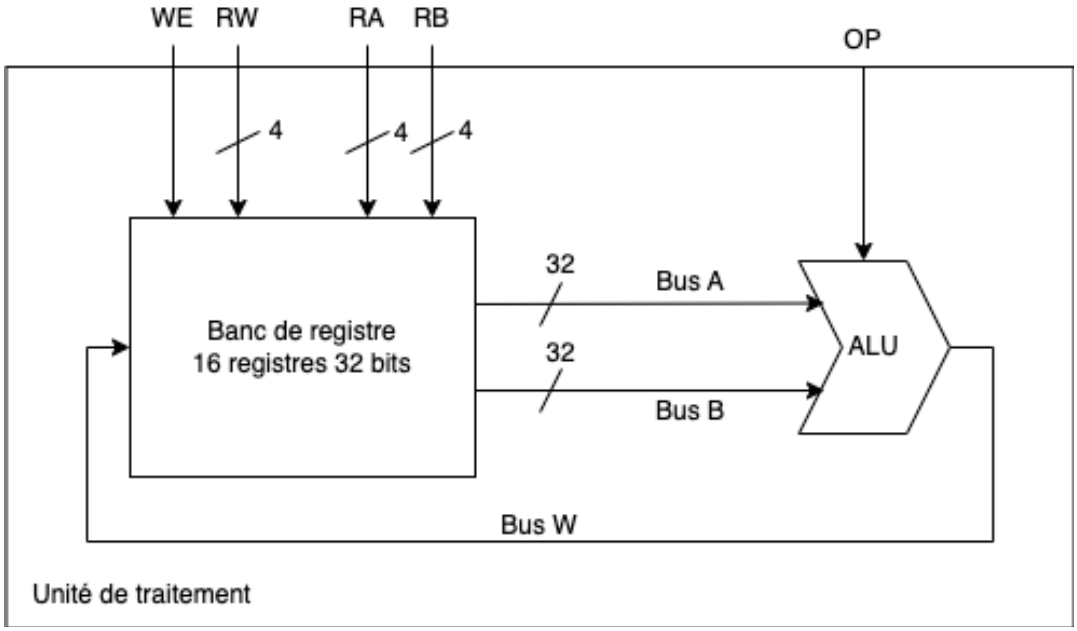
```
-- Declaration Type Tableau Memoire
type table is array(15 downto 0) of std_logic_vector(31 downto 0);

-- Fonction d'Initialisation du Banc de Registres
function init_banc return table is
variable result : table;
begin
   for i in 14 downto 0 loop
         result(i) := (others=>'0');
   end loop;
         result(15):=X"00000030";
         return result;
end init_banc;

-- Déclaration et Initialisation du Banc de Registres 16x32 bits
signal Banc: table:=init_banc;
```

*Conseil : Il faut impérativement éviter que la variable d'indexation du tableau soit un entier négatif, non initialisé, ou supérieur à la taille du tableau. Si un de ces cas se produit, la simulation s'arrêtera sur une erreur fatale.*

### 1.1.2 *Assembler l'UAL et le banc comme sur le schéma ci-dessous pour valider l'unité de traitement.*




### 1.1.3 *Écrire un banc de test et un script de simulation pour valider par simulation le bon fonctionnement des opérations suivantes :*

- ***R(1) = R(15)***
- ***R(1) = R(1) + R(15)***
- ***R(2) = R(1) + R(15)***
- ***R(3) = R(1) – R(15)***
- ***R(5) = R(7) – R(15)***

***Reporter les résultats de simulations sur votre compte-rendu.*** Pour la vérification de ces opérations, on vous conseille d'utiliser les alias et des assertions qui sont toujours plus efficaces qu'une vérification visuelle des chronogrammes.

# Multiplexeur 2 vers 1

Ce multiplexeur dispose d'un paramètre générique N fixant la taille des données en entrée et en sortie

- A, B : Deux bus sur N bits en entrée
- COM: Commande de sélection sur 1 bit
- S : Bus sur N bits en sortie

Les valeurs du signal COM fixe le comportement de la sortie

## Extension de signe

Ce module permet d'étendre en sortie sur 32 bits le signe d'une entrée codée sur N bits. Il possède donc un paramètre générique fixant la valeur de N. Il comporte également

- E : Bus sur N bits en entrée
- S : Bus sur 32 bits en sortie

# Mémoire de données

Cette mémoire permet de charger et stocker 64 mots de 32 bits. Il comporte les entrées sorties suivantes :

| COM | Opération Effectuée |
|---|---|
| 0 | S = A |
| 1 | S = B |

**CLK** : Horloge
**Reset** : reset asynchrone (actif à l'état haut) non représentés sur le schéma.
**DataIn** : Bus de donnés en écriture sur 32 bits
**DataOut** : Bus de données en lecture sur 32 bits
**Addr** : Bus d'adresses en lecture et écriture sur 6 bits
**WrEn** : Write Enable sur 1 bit

La **lecture** des registres se fait de manière **asynchrone et simultanée** :
Le bus de sortie **DataOut** porte la valeur de la données N° **Addr**

L'**écriture** des registres se fait de manière **synchrone**, sur le front montant de l'horloge. Elle est commandée par le signal WE :

- Si WE = 1, au front montant on recopie la valeur du bus **DataIn** dans le registre n° Addr
- Si WE = 0, au front montant il n'y a pas d'écriture effectuée

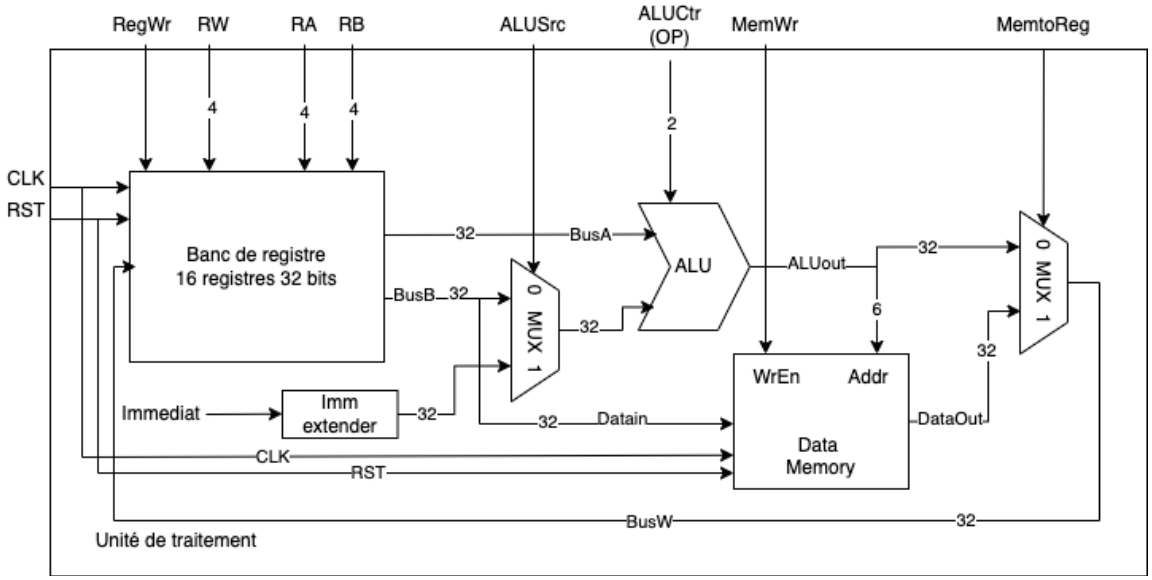
*Conseil: Ce module est très proche du banc de registre et peut être conçu à partir de ce dernier.*

## 1.2 Travail à réaliser

### *1.2.1 Décrire et simuler ces modules en VHDL comportemental. Reporter les résultats de simulations sur votre compte-rendu.*

## 1.3 Assemblage Unité de Traitement

Les modules décrits précédemment doivent être assemblés conformément au schéma ci-dessous :




*Le signal provenant de l'Alu étant sur 32 bits, la réduction avec les 6 bits de Addr peut être réalisée au niveau du* ***Port Map*** *(Addr => SignalAluOut (5 downto 0), …)*

### *1.3.1 Écrire un module VHDL effectuant l'assemblage de l'unité de traitement.*

### *1.3.2 Écrire un banc de test et un script de simulation pour valider le bon fonctionnement de :*

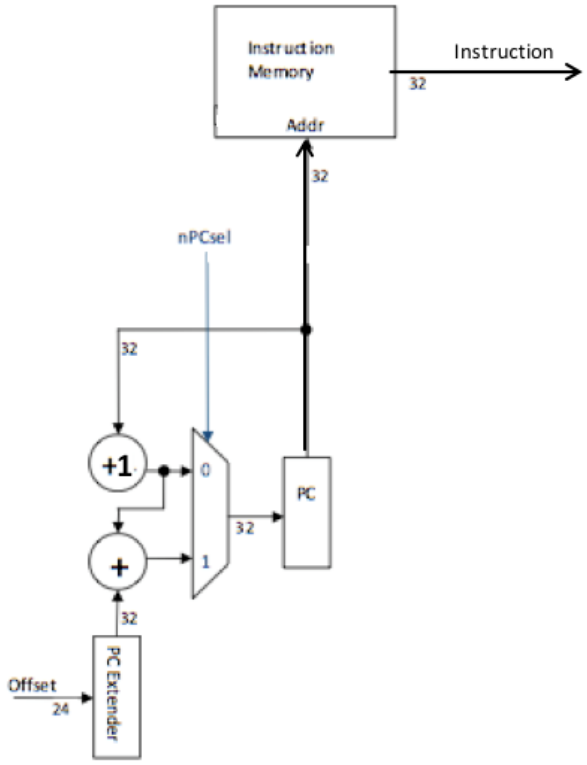
- ***L'addition de 2 registres***
- ***L'addition d'1 registre avec une valeur immédiate***
- ***La soustraction de 2 registres***
- ***La soustraction d'1 valeur immédiate à 1 registre***
- ***La copie de la valeur d'un registre dans un autre registre***
- ***L'écriture d'un registre dans un mot de la mémoire.***
- ***La lecture d'un mot de la mémoire dans un registre.***

***Reporter les résultats de simulations sur votre compte-rendu.***

# PARTIE 2 – UNITE DE GESTION DES INSTRUCTIONS

L'unité de gestion des instructions 32 bits, représentée sur la figure ci-dessous, possède :

- Une mémoire d'instruction de 64 mots de 32 bits similaire à celle de l'unité de traitement, celle-ci est fourni en annexe dans le fichier **instruction_memory.vhd**
- Il n'y a pas de bus de données en écriture ni de Write Enable.
- Un registre 32 bits (Registre PC). Ce registre possède en entrée une une horloge et un reset asynchrone (actif à l'état haut) non représentés sur le schéma.
- Une unité d'extension de 24 à 32 bits signés similaire au module décrit précédemment
- Une unité de mise à jour du compteur de programme PC suivant le signal de contrôle nPCsel :
  - Si nPCsel = 0 alors PC = PC + 1
  - Si nPCsel = 1 alors PC = PC + 1 + SignExt(offset)




Vous pouvez décrire ce composant de manière complètement structurelle ou alors choisir d'utiliser une description comportementale pour la partie de génération des nouvelles adresses.

***2.1 Décrire et simuler l'unité de gestion des instructions avec un banc de test et un script de simulation. Reporter les résultats de simulations sur votre compte-rendu.***

# PARTIE 3 – UNITE DE CONTROLE

L'unité de contrôle est constituée d'un registre 32 bits et d'un décodeur combinatoire.

## Registre 32 bits avec commande de chargement

Ce registre servira à stocker l'état du processeur (Processor State Register ou PSR) et doit permettre de stocker sur RegAff une valeur à afficher sur les LEDs ou les afficheurs 7 segments lors des instructions STR.

Il possède les entrées/sorties suivantes :

- DATAIN : Bus de chargement sur 32 bits
- RST : Reset asynchrone, actif à l'état haut
- CLK : Horloge
- WE : Commande de chargement
- DATAOUT : Bus de sortie sur 32 bits

Si WE=1, le registre mémorise la valeur placée sur le bus DATAIN.
Si WE=0, le registre conserve sa valeur précédente (état mémoire)

## Décodeur d'instructions

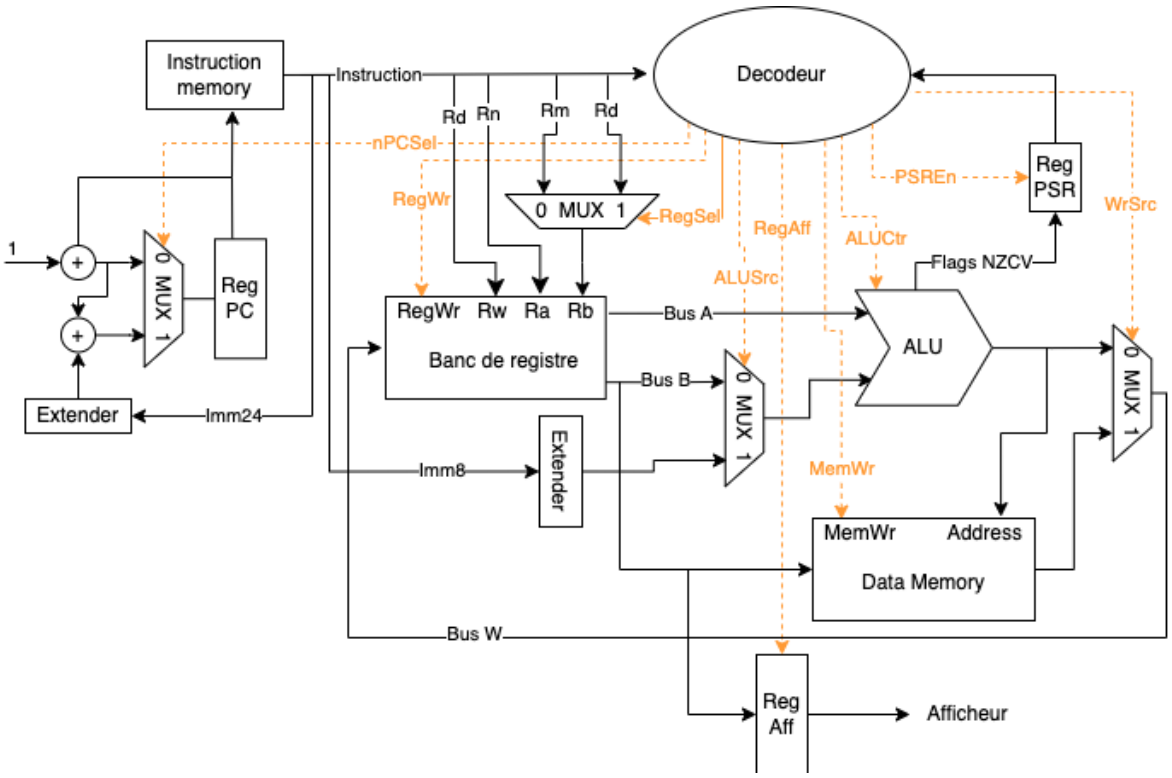
Ce module combinatoire génère les signaux de contrôle de l'unité de traitement, de l'unité de gestion des instructions, ainsi que des registre PSR, tous décrits précédemment.

Les valeurs de ces commandes dépendent de l'instruction récupérée dans la mémoire instructions, et éventuellement de l'état du registre PSR. La structure binaire des différentes instructions est décrite en Annexe.

Les commandes à générer concernent :

- Le multiplexeur d'entrée du registre PC (**nPC_SEL**)
- La commande de chargement du registre PSR (**PSREn**)
- Les bus d'adresse **Rn, Rm, Rd**, ainsi que le **WE** du banc de registres (**RegWr**)
- La commande **RegSel** du multiplexeur situé en amont du bus d'adresse RB de ce banc.
- La commande OP de l'UAL (**ALUCtrl**)
- ~~La commande du multiplexeur (**MemToReg**) qui permet de choisir le signal à écrire dans le banc de registre.~~
- La commande **ALUSrc** du multiplexeur situé en amont de l'entrée B de l'UAL.
- La commande **WrSrc** du multiplexeur situé en sortie de l'UAL et de la mémoire de données.
- Le Write Enable de la mémoire de données (**MemWr**)
- Le signal qui permet d'afficher une valeur en sortie lors de l'opération STR (**RegAff**)

Schéma global du processeur :




Pour la conception de ce décodeur, on s'appuiera sur le schéma d'ensemble du processeur présenté ci-dessus, du tableau de commande (voir page suivante) ainsi que sur le descriptif du jeu d'instructions fournis en annexe.

Pour décrire ce décodeur d'instructions, on déclarera tout d'abord le type énuméré suivant :

```
type enum_instruction is (MOV, ADDi, ADDr, CMP, LDR, STR, BAL, BLT);
signal instr_courante: enum_instruction;
```

On pourra ensuite utiliser deux process dans l'architecture

- Un process sensible sur la sortie de la mémoire instructions (nommée instruction sur le schéma en annexe) dont le rôle est de fixer la valeur du signal instr_courante
- Un process sensible sur les signaux **instructions** et **instr_courante** qui donnera la valeur des commandes des registres et opérateurs du processeur.

***3.1 Compléter le tableau « valeurs des commandes » se trouvant ci-dessous, qui synthétisera les actions du décodeur en fonction des instructions.***

***3.2 Décrire en VHDL les deux modules de l'unité de contrôle : Le décodeur d'instructions et le registre PSR sur 32 bits, s'il reçoit une commande de chargement, fera l'acquisition des drapeaux N et Z de l'ALU sur les bits de poids fort.***

# Valeurs des commandes

| INSTRUCTION | nPCSel | RegWr | ALUSrc | ALUCtr | PSREn | MemWr | WrSrc | RegSel | RegAff |
|---|---|---|---|---|---|---|---|---|---|
| ADDi | | | | | | | | | |
| ADDr | | | | | | | | | |
| BAL | | | | | | | | | |
| BLT | | | | | | | | | |
| CMP | | | | | | | | | |
| LDR | | | | | | | | | |
| MOV | | | | | | | | | |
| STR | | | | | | | | | |

# PARTIE 4 – ASSEMBLAGE ET VALIDATION DU PROCESSEUR

Il faut à présent finaliser la modélisation du processeur en assemblant ses trois unités principales.

- L'unité de gestion des instructions
- L'unité de traitement
- L'unité de contrôle

4.1 Compléter l'unité de traitement conçue précédemment en ajoutant un multiplexeur à 2 entrées sur 4 bits piloté par le signal de commande RegSel généré par l'unité de contrôle. Ce multiplexeur sera placé en entrée de l'adresse RB du banc de registres, comme cela est représenté sur le schéma du processeur.

4.2 Assembler le processeur à partir de ses trois unités.

4.3 Simuler l'exécution du programme de test par le processeur et vérifier son bon fonctionnement. Il sera nécessaire d'initialiser quelques données dans la Mémoire Data (de 0x10 à 0x1A)

## Programme de test

```
0x0 _main :  MOV R1,#0x10        ; --R1 <= 0x10
0x1          MOV R2,#0           ; --R2 <= 0
0x2 _loop :  LDR R0,0(R1)        ; --R0 <= DATA_MEM[R1]
0x3          ADD R2,R2,R0        ; --R2 <= R2 + R0
0x4          ADD R1,R1,#1        ; --R1 <= R1 + 1
0x5          CMP R1,0x1A         ; -- R1 - 0x1A, mise à jour de N
0x6          BLT loop            ; --branchement à _loop si R1 inferieur a 0x1A
0x7  _end  : STR R2,0(R1)        ; --DATA_MEM[R1] <= R2
0x8          BAL main            ; --branchement à _main
```

# PARTIE 5 : TEST DU PROCESSEUR SUR CARTE FPGA

Dans cette partie, vous allez implémenter votre processeur monocycle sur une carte FPGA.

Pour vérifier que l'exécution du programme de test se fait sans erreur, vous allez afficher sur les afficheurs 7 segments la valeur qui se trouve en sortie du registre **Reg Aff**.

Pour cela, il faut rajouter 4 décodeurs 7 segments pour décoder les bits 0 à 15 de la sortie du registre **Reg Aff** afin d'afficher cette valeur en hexadécimal sur 4 digits.

**Travail à réaliser :**

5.1 Créer un projet sur Quartus, instancier le processeur et les 4 décodeurs 7 segments dans un fichier **top_level.vhd**. Faire l'assignement des pins, lancer la synthèse et programmer la carte

5.2 Vérifier que vous trouvez la valeur attendue sur les afficheurs 7 segments.

# Partie 6 : Gestion des Interruptions externes

Pour gérer les interruptions externes, il faut rajouter un contrôleur d'interruption vectorisé que l'on appelle communément un VIC (Vector Interrupt Controler).

Il faudra également modifier le décodeur d'instruction afin de décoder l'instruction BX qui se trouve à la fin d'un sous-programme d'instruction pour signifier qu'il faut revenir au programme interrompu.

Ensuite il faudra modifier l'unité de gestion des instructions pour prendre en compte les interruptions.

## 6.1 VIC : Contrôleur d'interruption vectorisé

Entrées :

- CLK : Horloge
- RESET : Reset asynchrone, actif à l'état haut
- IRQ_SERV : Acquittement de l'interruption venant de l'unité de gestion des instructions
- IRQ0, IRQ1 : Requêtes d'interruption externe

Sorties :

- IRQ : Requête d'interruption envoyée vers l'unité de gestion des instructions
- VICPC : Adresse de début du sous-programme d'interruption (sur 32 bits)

Fonctionnement :

- Ce module échantillonne les valeurs de IRQ0 et IRQ1 afin de connaître la dernière (IRQi(n)) et l'avant dernière (IRQi(n-1)) valeur de ces deux signaux.
- Si on détecte une transition montante (IRQi(n)=1 ET IRQi(n-1)=0) sur l'un de ces signaux, on force à l'état haut un signal IRQ0_memo ou IRQ1_memo, signalant une requête d'interruption.
- IRQ1_memo et IRQ0_memo restent à l'état haut tant que l'on n'a pas reçu d'acquittement sur l'entrée IRQ_SERV.
- S'il n'y a aucune requête d'interruption, VICPC est forcé à 0.
- S'il s'agit de l'IRQ0 = 1, on force la sortie VICPC à la valeur 0x9 (adresse de début de ce sous-programme d'interruption)
- S'il s'agit de IRQ1_memo = 1, on force la sortie VICPC à la valeur 0x15 (adresse de début de ce sous-programme d'interruption)
- L'IRQ0 est prioritaire sur l'IRQ1.
- La sortie IRQ envoyée à la MAE est un OU logique entre IRQ1_memo et IRQ0_memo.

**Signal envoyé à l'unité de gestion des instructions : IRQ**
**Commande à générer par l'unité de gestion des instructions : IRQ_SERV.**

A FAIRE

***6.1.1 Créer une nouvelle source VHDL pour décrire le VIC.***

***6.1.2 Simuler ce composant à l'aide d'un banc de test et un script de simulation.***

## 6.2 Modification du décodeur d'instruction

Afin de gérer la sortie des sous-programmes d'interruption, on utilise l'instruction assembleur BX. Cette instruction permet de sortir d'un sous-programme d'interruption et revenir au programme principal.

Pour cela, il faut rajouter un signal de contrôle en sortie sur l'entité du décodeur :

- **IRQ_END** : mettre à '1' lorsque l'instruction BX est exécuté.
- Voici le code binaire de l'instruction BX : 0xEB00 0000 ;

## 6.3 Modification sur l'unité de gestion des instructions

Il faut reprendre l'unité de gestion des instructions et rajouter les signaux suivants sur son entité :

**ENTREE :**

- **IRQ** : si actif, signifie une nouvelle interruption, généré par le VIC.
- **VICPC** : transmet l'adresse (vecteur) du sous-programme d'interruption.
- **IRQ_END** : si actif, indique la fin du sous-programme d'interruption.

**SORTIE :**

- **IRQ_SERV :** Acquittement de l'interruption lorsqu'elle a été prise en compte

**Fonctionnement :**

- Lorsqu'une interruption est active (IRQ = 1), il faudra copier la valeur de VICPC dans PC et en même temps sauvegarder le registre PC dans un registre appelé LR (Link Register).
- Lorsque l'interruption est prise en compte mettre le signal IRQ_SERV à '1', sinon le laisser à '0' tout le reste du temps.
- Lorsque le signal IRQ_END est actif, remettre dans PC la valeur de LR afin que le programme principal puisse continuer à s'exécuter.
- Il faudra faire attention au timing des signaux pour que chaque instruction ne s'exécute qu'une seule fois !

## 6.4 Test par simulation

Simuler le processeur avec le VIC à l'aide d'un banc de test et un script de simulation. Reporter vos résultats de simulations sur le compte-rendu.

## 6.5 Test du VIC sur FPGA

Implémenter le processeur avec le VIC sur FPGA et vérifier qu'il fonctionne correctement.

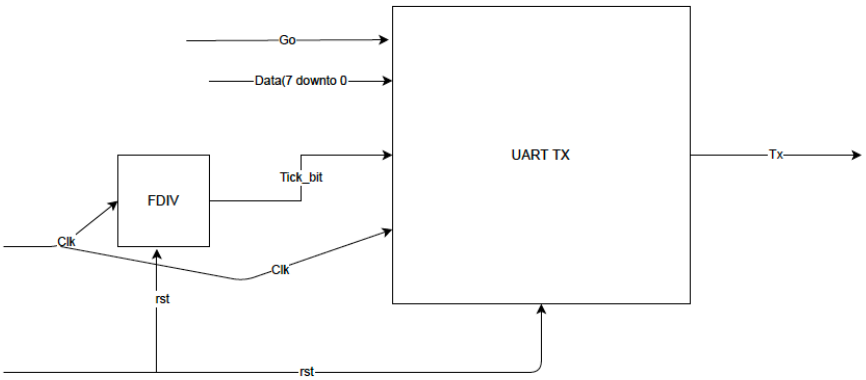
# PARTIE 7 : Rajouter un périphérique UART au processeur

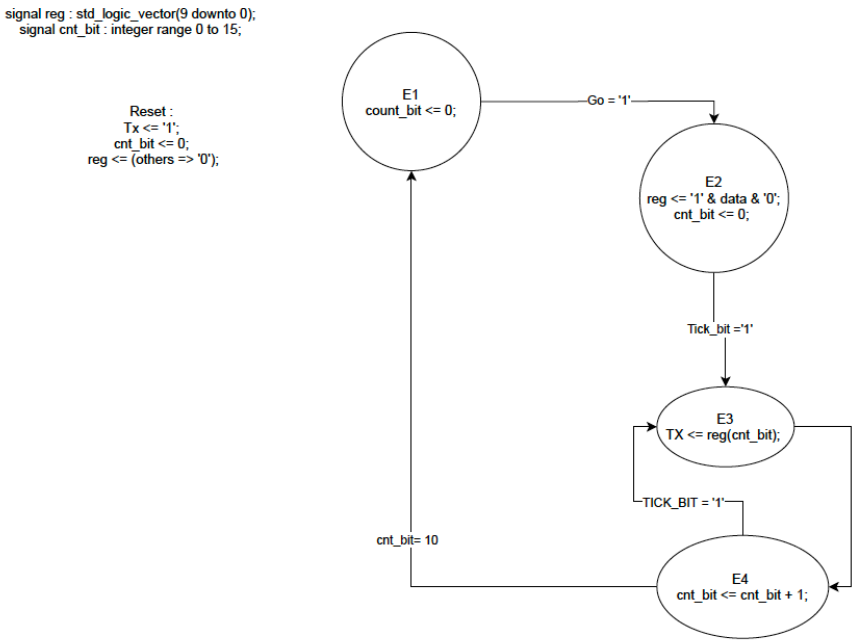
Dans cette partie, nous allons voir comment rajouter un périphérique UART à notre processeur. Vous allez commencer par décrire la partie opérative de l'UART et ensuite vous allez voir comment la connecter au processeur.

Pour tester cet UART, il faudra exécuter un petit code assembleur qui permet d'envoyer la phrase « HELLO WORLD !» vers le terminal d'un PC.

**Partie émission de l'UART :**

Voici ci-joint le schéma et la MAE de la partie émission de l'UART :



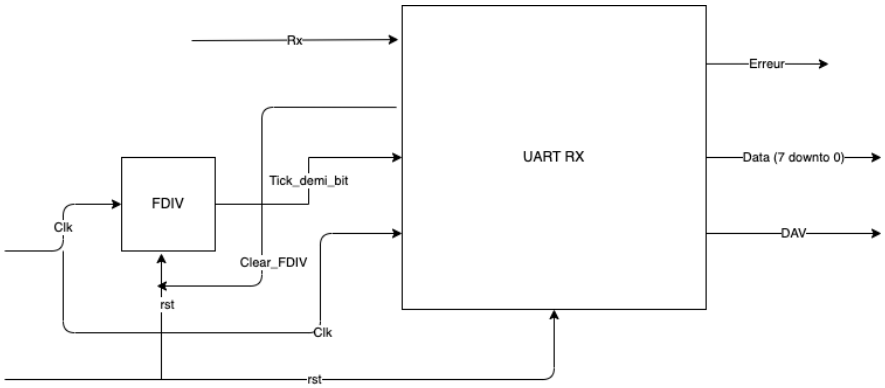



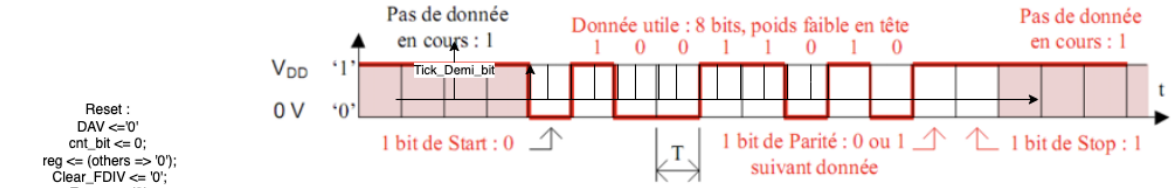
## A FAIRE

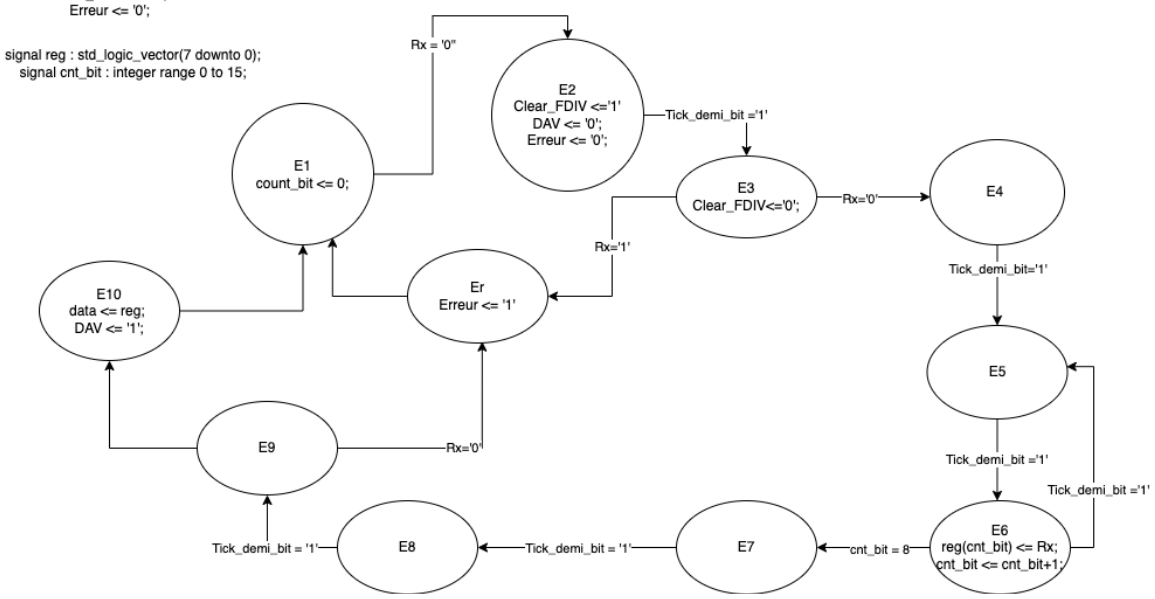
1) Créer une nouvelle source VHDL pour décrire la partie émission de l'UART
2) Simuler ce composant à l'aide d'un banc de test et un script de simulation.
3) Tester la partie opérative de l'UART sur la carte FPGA en connectant la sortie Tx sur une pin du port GPIO (GPIO(0) par exemple), l'entrée data sur SW(7 ..0) et l'entrèe Go sur KEY(1). Ensuite utiliser un convertisseur série/USB pour recevoir les caractères sur le terminal de votre PC (Puty sur Windows ou Screen sur Linux et Mac). Il faudra connecter la sortie Tx (GPIO(0)) sur le pin TXI du convertisseur série/USB.
4) Rajouter un registre 32 bit UART_Conf à votre périphérique. Celui-ci permet de recevoir l'octet à envoyer sur les bits 7 à 0. Chaque fois que vous écrivez sur le registre, vous devez mettre le Go à 1. Rajouter ce périphérique UART dans votre processeur. En utilisant une instruction STR à l'adresse 0x40 vous devez pouvoir écrire dans le registre UART_Conf. Pour cela, il faut faire un décodage d'adresse qui permet d'écrire dans le registre que lorsque l'adresse est égale à 0x40.
5) Simuler le processeur avec ce nouveau périphérique en envoyant par exemple la valeur 0x31 ('1' en ASCII) à chaque fois que vous rentrez dans l'IRQ1.
6) Tester votre processeur avec le périphérique UART sur la carte FPGA. Pour communiquer avec le PC, il faudra utiliser un convertisseur série/USB et utiliser un Terminal sur votre ordinateur (Puty sur Windows ou Screen sur Mac).
7) L'envoi d'un caractère prend un certain temps. Avec un baud rate de 115200, il faut attendre environ 87 us. Vous pouvez gérer cela par un mécanisme d'interruption. Lorsqu'un caractère a été envoyé vous pouvez générer une interruption et ainsi envoyer le prochain caractère dans le sous-programme d'interruption. Pour cela il faut générer un signal TxIrq dans votre UART à la fin de l'émission d'un caractère. Et il faudra également changer le VIC pour gérer cette nouvelle source d'interruption.
8) Modifier votre périphérique UART et le VIC afin de le gérer par interruption. Modifier le code assembleur à exécuter afin d'envoyer la chaine de caractère HELLO WORLD dans le sous-programme d'interruption. Simuler le et tester le sur la carte FPGA.

# Partie réception de l'UART :

Faites la même chose pour la partie réception en vous aidant du schéma et de la MAE proposé ci-dessous.



# ANNEXES :

## Code assembleur de la partie 4 : instruction_memory.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity instruction_memory is
  port(
    PC: in std_logic_vector (31 downto 0);
    Instruction: out std_logic_vector (31 downto 0)
    );
end entity;

architecture RTL of instruction_memory is
  type RAM64x32 is array (0 to 63) of std_logic_vector (31 downto 0);

function init_mem return RAM64x32 is
  variable result : RAM64x32;
begin
for i in 63 downto 0 loop
  result (i):=(others=>'0');
end loop;           -- PC        -- INSTRUCTION  -- COMMENTAIRE
result (0):=x"E3A01010";-- 0x0 _main -- MOV R1,#0x10 -- R1 = 0x10
result (1):=x"E3A02000";-- 0x1       -- MOV R2,#0x00 -- R2 = 0
result (2):=x"E6110000";-- 0x2 _loop -- LDR R0,0(R1) -- R0 = DATAMEM[R1]
result (3):=x"E0822000";-- 0x3       -- ADD R2,R2,R0 -- R2 = R2 + R0
result (4):=x"E2811001";-- 0x4       -- ADD R1,R1,#1 -- R1 = R1 + 1
result (5):=x"E351001A";-- 0x5      -- CMP R1,0x1A  -- Flag = R1-0x1A,si R1 <= 0x1A
result (6):=x"BAFFFFFB";-- 0x6       -- BLT loop -- PC =PC+1+(-5) si N = 1
result (7):=x"E6012000";-- 0x7       -- STR R2,0(R1) -- DATAMEM[R1] = R2
result (8):=x"EAFFFFF7";-- 0x8       -- BAL main       -- PC=PC+1+(-9)
  return result;
end init_mem;

signal mem: RAM64x32 := init_mem;

begin
  Instruction <= mem(to_integer(unsigned (PC)));
end architecture;
```

# Code assembleur de la partie 6 : instruction_memory_IRQ.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity instruction_memory_IRQ is
    port(
        PC: in std_logic_vector (31 downto 0);
        Instruction: out std_logic_vector (31 downto 0)
    );
end entity;

architecture RTL of instruction_memory_IRQ is
    type RAM64x32 is array (0 to 63) of std_logic_vector (31 downto 0);

function init_mem return RAM64x32 is
    variable ram_block : RAM64x32;
begin
        -- PC        -- INSTRUCTION  -- COMMENTAIRE
    ram_block(0 ) :=  x"E3A01010"; -- _main :   MOV R1,#0x10    ; --R1 <= 0x10
    ram_block(1 ) :=  x"E3A02000"; --           MOV R2,#0       ; --R2 <= 0
    ram_block(2 ) :=  x"E6110000"; -- _loop :   LDR R0,0(R1)    ; --R0 <= MEM[R1]
    ram_block(3 ) :=  x"E0822000"; --           ADD R2,R2,R0    ; --R2 <= R2 + R0
    ram_block(4 ) :=  x"E2811001"; --           ADD R1,R1,#1    ; --R1 <= R1 + 1
    ram_block(5 ) :=  x"E351001A"; --           CMP R1,0x1A     ; --? R1 = 0x1A
    ram_block(6 ) :=  x"BAFFFFFB"; --     BLT loop    ; --branchement à _loop si R1 inferieur a 0x1A
    ram_block(7 ) :=  x"E6012000"; --           STR R2,0(R1)    ; --MEM[R1] <= R2
    ram_block(8 ) :=  x"EAFFFFF7"; --           BAL main        ; --branchement à _main
                                -- ISR 0 : interruption 0
                                        --sauvegarde du contexte
    ram_block(9 ) :=  x"E60F1000"; --           STR R1,0(R15)       ; --MEM[R15] <= R1
    ram_block(10) :=  x"E28FF001"; --           ADD R15,R15,1       ; --R15 <= R15 + 1
    ram_block(11) :=  x"E60F3000"; --           STR R3,0(R15)       ; --MEM[R15] <= R3
                                        --traitement
    ram_block(12) :=  x"E3A03010"; --           MOV R3,0x10         ; --R3 <= 0x10
    ram_block(13) :=  x"E6131000"; --           LDR R1,0(R3)        ; --R1 <= MEM[R3]
    ram_block(14) :=  x"E2811001"; --           ADD R1,R1,1         ; --R1 <= R1 + 1
    ram_block(15) :=  x"E6031000"; --           STR R1,0(R3)        ; --MEM[R3] <= R1
                                        -- restauration du contexte
    ram_block(16) :=  x"E61F3000"; --           LDR R3,0(R15)       ; --R3 <= MEM[R15]
    ram_block(17) :=  x"E28FF0FF"; --           ADD R15,R15,-1      ; --R15 <= R15 - 1
    ram_block(18) :=  x"E61F1000"; --           LDR R1,0(R15)       ; --R1 <= MEM[R15]
    ram_block(19) :=  x"EB000000"; --           BX           ; -- instruction de fin d'interruption
    ram_block(20) :=  x"00000000";
                                 -- ISR1 : interruption 1
                                --sauvegarde du contexte - R15 correspond au pointeur de pile
    ram_block(21) :=  x"E60F4000"; --           STR R4,0(R15)       ; --MEM[R15] <= R4
    ram_block(22) :=  x"E28FF001"; --           ADD R15,R15,1       ; --R15 <= R15 + 1
```

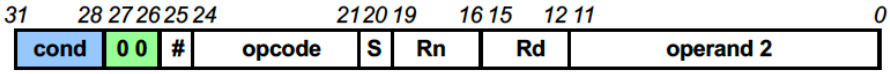
```
    ram_block(23) :=  x"E60F5000"; --           STR R5,0(R15)   ; --MEM[R15] <= R5
                                                --traitement
    ram_block(24) :=  x"E3A05010"; --           MOV R5,0x10         ; --R5 <= 0x10
    ram_block(25) :=  x"E6154000"; --           LDR R4,0(R5)        ; --R4 <= MEM[R5]
    ram_block(26) :=  x"E2844002"; --           ADD R4,R4,2         ; --R4 <= R1 + 2
    ram_block(27) :=  x"E6054000"; --           STR R4,0(R5)        ; --MEM[R5] <= R4
                                               -- restauration du contexte
    ram_block(28) :=  x"E61F5000";--           LDR R5,0(R15)    ; --R5 <= MEM[R15]
    ram_block(29) :=  x"E28FF0FF";  --           ADD R15,R15,-1     ; --R15 <= R15 - 1
    ram_block(30) :=  x"E61F4000"; --           LDR R4,0(R15)   ; --R4 <= MEM[R15]
    ram_block(31) :=  x"EB000000";--           BX     ; -- instruction de fin d'interruption
    ram_block(32) :=  x"00000001";
    ram_block(33) :=  x"00000002";
    ram_block(34) :=  x"00000003";
    ram_block(35) :=  x"00000004";
    ram_block(36) :=  x"00000005";
    ram_block(37) :=  x"00000006";
    ram_block(38) :=  x"00000007";
    ram_block(39) :=  x"00000008";
    ram_block(40) :=  x"00000009";
    ram_block(41) :=  x"0000000A";
    ram_block(42 to 63) := (others=> x"00000000");
    return ram_block;
end init_mem;

signal mem: RAM64x32 := init_mem;

begin
            Instruction <= mem(to_integer (unsigned (PC)));
end architecture;
```

## Codage binaire d'une instruction de traitement :

- Instructions de traitement (1)



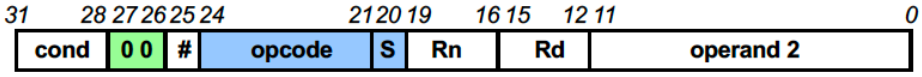

- *Cond*: l'instruction est exécutée si le registre d'état CPSR vérifie la condition spécifiée

| Asm | Cond |
|---|---|
| EQ | 0000 |
| NE | 0001 |
| CS/HS | 0010 |
| CC/LO | 0011 |
| MI | 0100 |
| PL | 0101 |

| Asm | Cond |
|---|---|
| VS | 0110 |
| VC | 0111 |
| HI | 1000 |
| LS | 1001 |
| GE | 1010 |
| LT | 1011 |

| Asm | Cond |
|---|---|
| GT | 1100 |
| LE | 1101 |
| AL | 1110 |
| NV | 1111 |

- Instructions de traitement (2)



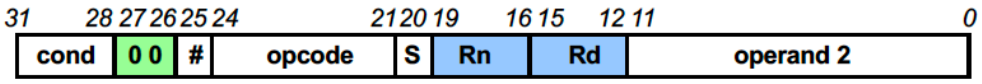

- *S = 1* : affecte CPSR
- *Opcode* : code de l'opération à effectuer

| Asm | Opcode |
|---|---|
| AND | 0000 |
| EOR | 0001 |
| SUB | 0010 |
| RSB | 0011 |
| ADD | 0100 |
| ADC | 0101 |

| Asm | Opcode |
|---|---|
| SBC | 0110 |
| RSC | 0111 |
| TST | 1000 |
| TEQ | 1001 |
| CMP | 1010 |
| CMN | 1011 |

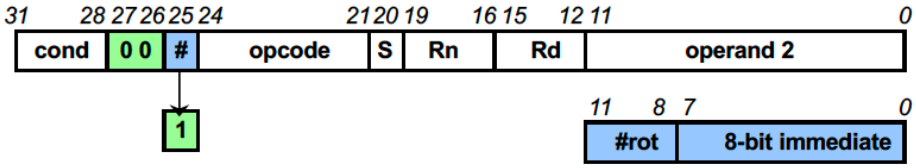
| Asm | Opcode |
|---|---|
| ORR | 1100 |
| MOV | 1101 |
| BIC | 1110 |
| MVN | 1111 |

- Instructions de traitement (3)



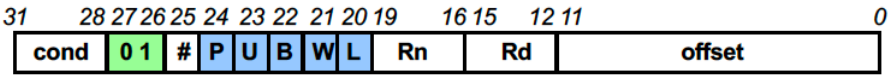

- *Rd* → numéro du registre de destinations
  - 4 bits pour sélection parmi les 16 registres possibles
- *Rn* → numéro du registre qui sert de premier opérande
  - 4 bits pour sélection parmi les 16 registres possibles
- *operand2* → relié à l'entrée B
  - possibilité de rotation/décalage

- Instructions de traitement (4)



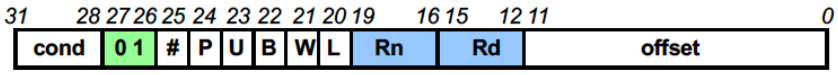

- Si le bit # est à 1, *operand2* est une valeur littérale sur 32 bits.
  - Une instruction est codée sur 32 bits, il n'est donc pas possible de coder n'importe quelle valeur littérale 32 bits.
  - Cette valeur est construite par rotation vers la droite d'une valeur sur 8 bits.

- Instructions de transfert
  - Lecture/écriture
  - Données accédées
    - Mots, demi-mots, octets
    - Signées/non signées
  - Mode d'accès (pré/post indexé, +/- offset, write back)
  - Transfert simple/multiple
  - Registre source/destination, liste de registres
  - Registre de base
  - Offset
  - Condition
    - Exécution conditionnelle d'un transfert
- Instructions de transfert de mots ou d'octets non-signés (LDR, STR, LDRB, STRB)



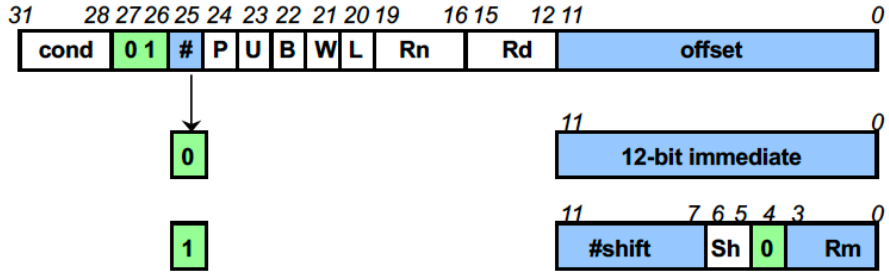

- *P* : pre/post index
  - 1 pré-indexé, 0 post-indexé
- *U* : up/down
  - 1 + offset, 0 -offset
- *B* : byte/word
  - M1 accès 8 bits, 0 accès 32 bits
- *W* : write-back
  - Si P=1, W=1 adressage pré-indexé automatique
- *L* : load/store
  - 1 load, 0 store

- Instructions de transfert de mots ou d'octets non-signés (LDR, STR, LDRB, STRB)




- *Rd* → registre source (si L=0, store) ou destination (si L=1, load)
- *Rn* → registre de base

- Instructions de transfert de mots ou d'octets non-signés (LDR, STR, LDRB, STRB)




- Offset: soit un littéral non signé sur 12 bits, soit un registre d'index (*Rm*) éventuellement décalé sur un nombre constant de bits (*#shift*)

# Instructions de branchement relatif

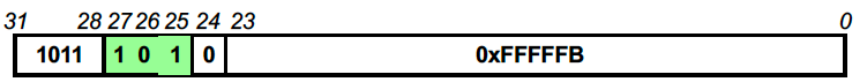
- Branch (B)
- Branch and Link (BL)


- Offset : déplacement signé sur 24 bits
- L : Link (0 branch, 1 branch and link)
- Effets:
  - B: PC ← PC + offset
  - BL: r14 ← PC – 4; PC = PC + offset
  - r14 : Link Register (contient l'adresse de retour)

# Instructions de branchement relatif

- Exemple traduction d'une boucle *for*
- Utilisation *de labels* en langage d'assemblage: le déplacement est calculé automatiquement par l'assembleur

```
                MOV     r4, #0          @ tmp=0
                MOV     r5, #0          @ i=0
loop:           ADD     r4, r4, r5      @ tmp+=i
                ADD     r5, r5, #1      @ i++
                CMP     r5, #5
                BLT     loop            @ i<5 : réitérer
                …
```




LT B $(FFFFFB)_{hex} = (-5)_{dec}$

**Représentation binaire: BAFFFFFB**