

Laboratoire d'architecture des ordinateurs semestre printemps 2023 - 2024

Microarchitecture DECODE et EXECUTE

Informations générales

Le rendu pour ce laboratoire sera **par groupe de deux**, chaque groupe devra rendre son travail sur Cyberlearn.

Le rendu du laboratoire sera évalué comme indiqué dans la planification des laboratoires. Tout retard impactera la note obtenue.

 **N'oublier pas de sauvegarder et d'archiver votre projet à chaque séance de laboratoire**

NOTE : Nous vous rappelons que si vous utilisez les machines de laboratoire situées au niveau A, il ne faut pas considérer les données qui sont dessus comme sauvegardées. Si les machines ont un problème nous les remettons dans leur état d'origine et toutes les données présentes sont effacées.

Objectifs du laboratoire

L'objectif principal de ce laboratoire est la réalisation de la partie DECODE d'un processeur simplifié. L'idée sera de développer un système de A à Z afin que vous puissiez faire chaque étape vous-mêmes et ainsi bien comprendre les concepts vus dans la théorie du cours afin de les appliquer dans un cas pratique.

Vous devez rendre les projets Logisim ainsi que les codes en assembleur, le tout sera noté. Vous pouvez également rendre les réponses aux questions qui seront corrigées mais pas évaluées.

Outils

Pour ce labo, vous devez utiliser les outils disponibles sur les machines de laboratoire (A07 / A09) ou votre ordinateur personnel avec la machine virtuelle fournie par le REDS.

Fichiers

Vous devez télécharger à partir du site Cyberlearn un ".zip" contenant un répertoire «workspace» où vous trouverez :

- **processeur_ARO2.circ** : Le fichier de travail Logisim
- **main.S** : fichier source du code assembleur pour tester le circuit final
- **Makefile** : fichier contenant les directives d'assemblage


 **Les fichiers main.S et Makefile ne doivent pas être modifiés !**

Workspace

Vous devez télécharger le nouveau workspace fourni sur la page Cyberlearn du cours car quelques corrections ont été apportées à l'ancienne version du workspace. Le nouveau workspace contient la correction du bloc FETCH, le bloc Execute et le bloc Memory-Access Le nouveau workspace devrait contenir :

- Mémoire d'instructions
- Mémoire de données
- Processeur_ARO2
- Contrôleur mémoire
- bloc FETCH
- bloc EXECUTE
- bloc MEMORY-ACCESS

Ainsi que la plupart des entités que vous allez réaliser dans le cadre de ce cours. Certaines sont déjà complétées, ce sont soit des parties qui prennent trop de temps selon nous ou alors qui ne sont pas très constructives à réaliser. Cependant, lorsque vous allez en avoir besoin, nous vous expliquerons leur fonctionnement.

 **Respectez l'architecture hiérarchique présentée dans le cours.**

 **Ne modifiez pas les entrées/sorties des composants/entités fournis.**

Travail demandé

Ce laboratoire est divisé en trois parties :

1. La banque de registres
2. Le bloc decode
3. Les blocs Execute et Memory Access

1 Banque de registres

Entité du bloc BANK_REGISTERS



Nom I/O	Description
clk_i	Entrée d'horloge
rst_i	Entrée du reset asynchrone
register_d_sel_i	Sélecteur du registre destination
register_d_data_i	Valeur à mettre dans le registre destination
register_s_sel_i	Sélecteur du registre S
register_n_sel_i	Sélecteur du registre N
register_mem_sel_i	Sélecteur du registre MEM
reg_bank_control_bus_i	Bus de contrôle de la banque de registres
PC_fut_i	Valeur future du Program Counter
LR_addr_i	Valeur à mettre dans le LR pour un retour futur (BL)
branch_i	Flag qu'un saut doit être effectué
link_en_i	Permet le support de l'instruction pour un long saut
reg_bank_en_i	Signal enable de la banque de registre
register_s_read_data_o	Valeur du registre sélectionné par S
register_n_read_data_o	Valeur du registre sélectionné par N
register_mem_read_data_o	Valeur du registre sélectionné par MEM
PC_pres_o	Valeur présente du Program Counter

TABLE 1 – Description des entrées/sorties du bloc bank register

Position	Taille	Description
0	1	Pas utilisé
1	1	Ecriture dans la banque de registres (write enable)

TABLE 2 – Construction du bus de contrôle de la banque de registres

1.1 Instanciation des registres dans la banque

Remarque : Implémenter et tester.

Dans le bloc *bank_registers*, instanciez 8 registres pour les données dont 3 registres pour les fonctions spécifiques. Ces registres **doivent avoir** la même "clock" et le même "reset".

Ces registres seront adressés de la façon suivante, **le nommage attendu doit être respecté** :

Adress reg	Fonctionnalité	Nom de l'élément
0	R0	R0
1	R1	R1
2	R2	R2
3	R3	R3
4	R4	R4
5	SP (R5)	SP
6	LR (R6)	LR
7	PC (R7)	PC

TABLE 3 – Registres

Remarque : Les registres 5, 6 et 7 sont ici utilisés pour le Stack Pointer (SP), le Link Register (LR) et le Program Counter (PC) à la place des registres 13, 14 et 15 vu dans le cours. Ceci est dû à une simplification des instructions et l'accès aux registres sur 3 bits.

Pour simplifier le debug durant le développement, il faut nommer les registres et tous les rendre visibles dans l'onglet "registers".

Pour cela : Sélectionnez un registre et dans les propriétés, "show in registers tab" -> Yes.

Cette opération doit être répétée pour chaque registre. Vous pouvez maintenant voir en tout temps l'état de ces registres. Ce qui pourra être utile pour la suite.

1.2 Ecriture dans la banque de registres

Remarque : Implémenter et tester.

Connectez le bus *register_d_data_i* aux différents registres. Ceci permettra d'écrire la donnée en entrée de la banque de registres dans le registre sélectionné par l'entrée *register_d_sel_i*.

Vous devez décoder la valeur de l'adresse de destination *register_d_sel_i* afin d'autoriser l'écriture dans le registre désiré. Sur chaque registre vous utiliserez l'entrée "write enable" pour commander indépendamment les écritures.

1.3 Lecture de 1 registre dans la banque de registres

Remarque : Implémenter et tester.

Lire le registre sélectionné par *register_s_sel_i* et fournir la valeur du registre désiré sur la sortie *register_s_read_data_o*.

1.4 Lecture de 3 registres dans la banque de registres

Remarque : Implémenter et tester.

Ajoutez les deux autres bus afin que les sorties *register_n_read_data_o* et *register_mem_read_data_o* soient connectées et permettent de lire les valeurs des registres nécessaires pour les calculs.

1.5 Enable pour la banque de registres

Remarque : Implémenter et tester.

Implémentez l’enable de la banque de registre en combinant les signaux *reg_bank_control_bus_i* et *reg_bank_en_i* afin de gérer l’autorisation de l’écriture de la banque de registres. Si l’enable n’est pas actif, alors aucun registre ne doit pouvoir être modifié.

1.6 Registre Program Counter (PC)

Remarque : Implémenter et tester.

Le registre PC est un registre particulier, comme vu dans le laboratoire Fetch, qui doit être géré différemment. Le registre PC a son entrée “write enable” qui est toujours actif (‘1’) et la donnée en entrée du registre doit répondre au tableau suivant :

write_en_PC	link_en	PC_reg_input+
0	0	PC_fut_i
0	1	PC_fut_i
1	0	register_d_data_i
1	1	PC_fut_i

TABLE 4 – Définition du PC

Remarque : Il faut que le signal *write_en_PC* en entrée de la table prennent en compte l’enable du registre 7 comme fait lors des étapes 1.2 et 1.5.

Ensuite, connectez la sortie *PC_pres* à votre registre PC.

1.7 Gestion du Link Register (LR)

Remarque : Implémenter et tester.

Vous trouverez un bloc *LR_manager* qui est déjà réalisé. Par exemple, ce bloc permet de gérer l’écriture dans le LR lorsque nous voulons faire un long saut avec un lien et/ou que les 11 bits du saut inconditionnel ne suffisent pas. Il faut instancier ce bloc devant le registre LR et le connecter. Les noms des signaux devraient être assez parlants (Attention, l’entrée *link_en_i* du bloc LR_Manager permet d’écrire dans le registre LR, sur cette entrée il ne faut pas oublier de prendre en compte l’entrée *reg_bank_en_i* avec *link_en_i* de la banque des registre)

Observer le contenu de ce bloc. Ce bloc permet de choisir si nous stockons la valeur du PC dans le LR afin de préparer un long saut. Sinon, c’est un registre accessible avec le nom de registre R6.

Question 1 :

Créez un petit programme de 5-6 instructions en assembleur. Buildez et chargez votre programme dans la mémoire d’instruction.

Faites le chronogramme de l’exécution de votre programme et vérifiez que le registre PC s’incrémente correctement.

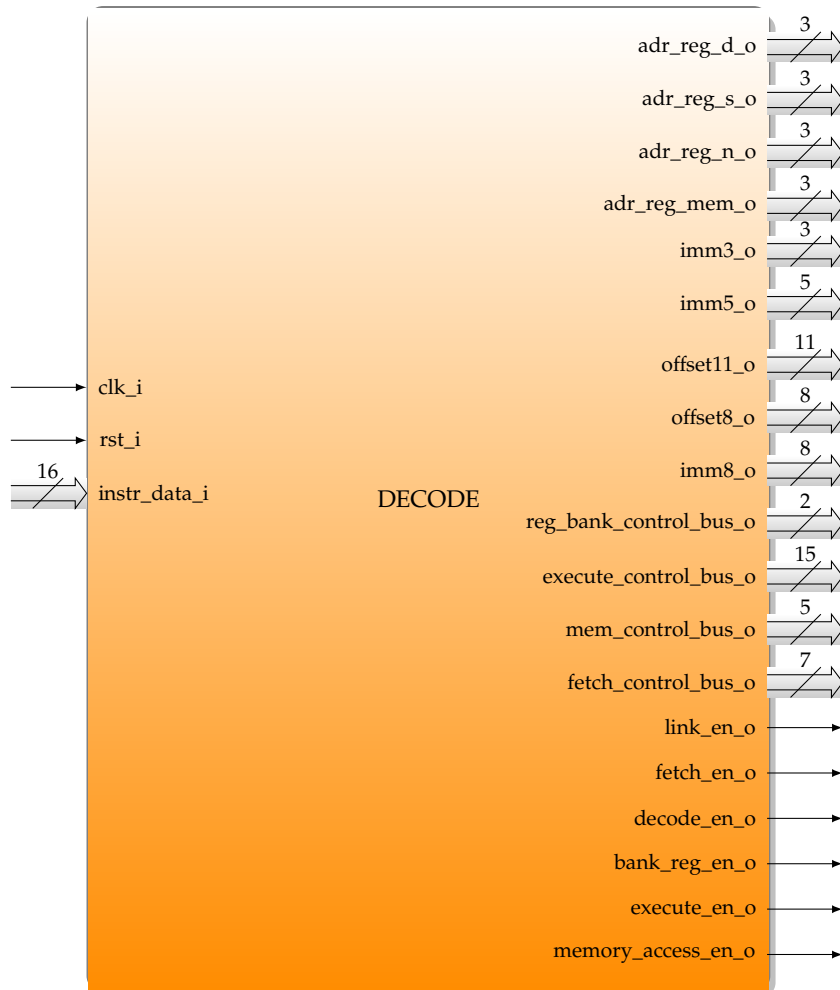
2 Bloc DECODE

⚠ Pour le décodage des instructions, il sera noté :

- Instruction_r_imm : instruction reg, immediate
- Instruction_r_r : instruction reg, reg
- etc ...

Remarque : Vous trouverez sur Cyberlearn dans la section “**Documents**” la documentation des instructions THUMB correspondantes (**ARM7-TDMI-manual-pt3**).

Entité du bloc DECODE




Nom I/O	Description
clk_i	Entrée d'horloge
instr_data_i	Instruction à décoder, donnée par le fetch
adr_reg_d_o	Adresse registre destination
adr_reg_s_o	Adresse registre source principal
adr_reg_n_o	Adresse registre source secondaire
adr_reg_mem_o	Adresse registre utilisé pour memory access
imm3_o	Valeur immédiate 3 bits (pour instruction correspondante)
imm5_o	Valeur immédiate 5 bits (pour instruction correspondante)
offset11_o	Valeur d'offset 11 bits (pour instruction correspondante)
offset8_o	Valeur d'offset 8 bits (pour instruction correspondante)
imm8_o	Valeur immédiate 8 bits (pour instruction correspondante)
reg_bank_control_bus_o	Bus de contrôle du bloc registers bank
execute_control_bus_o	Bus de contrôle du bloc execute
mem_control_bus_o	Bus de contrôle du bloc memory access
fetch_control_bus_o	Bus de contrôle du bloc fetch
link_en_o	Autorisation d'écrire dans le LR
fetch_en_o	Enable du bloc fetch
decode_en_o	Enable du bloc decode
bank_reg_en_o	Enable de la banque de registres
execute_en_o	Enable du bloc execute
memory_access_en_o	Enable du bloc memory access

TABLE 5 – Description des entrées/sorties du bloc decode

Les tableaux ci-dessous indiquent comment les différents bus de contrôles sont construits.

Position	Taille	Description
0	1	Sélection PC (équivalent de saut)
1	1	Flag qui indique une instruction conditionnelle
5-2	4	Sélection de la condition à tester
6	1	Pas utilisé

TABLE 6 – Construction du bus de contrôle du bloc fetch

 **Ce bus ne sort pas du bloc decode.**

Position	Taille	Description
1-0	2	Sélection Rs (source)
2	1	Sélection Rn (source)
3	1	Link enable (Utilisé pour long saut)
4	1	Sélection Rd (destination)
5	1	Pas utilisé

TABLE 7 – Construction du bus de contrôle du bloc decode

Position	Taille	Description
0	1	Pas utilisé
1	1	Ecriture dans la banque de registres (write enable)

TABLE 8 – Construction du bus de contrôle du bloc bank_registers

Position	Taille	Description
2-0	3	Sélection de l'opération du shift
5-3	3	Sélection de l'opération du bloc ALU
7-6	2	Sélection de l'opérande 2
8	1	Sélection de l'opérande 1
9	1	Sélection de la valeur d'entrée pour le shift
10	1	Flag qui met à jour le CPSR
13-11	3	Sélection de l'opération du bloc Mult_2
14	1	Pas utilisé

TABLE 9 – Construction du bus de contrôle du bloc execute


Position	Taille	Description
0	1	Flag qui indique que l'opération courante utilise memory access
1	1	Flag qui indique un stockage de la valeur en mémoire
2	1	Flag qui indique une lecture de la valeur en mémoire
3	1	Flag qui indique une opération en byte (halfword sinon)
4	1	Pas utilisé

TABLE 10 – Construction du bus de contrôle du bloc memory access

2.1 Circuit Decode

Remarque : Implémenter et tester.

Dans le circuit *decode*, implémentez la sélection des adresses (créer les bus) **adr_reg_s**, **adr_reg_d** et **adr_reg_n** en fonction du bus de contrôle du DECODE (la construction du bus est spécifié dans le Tableau 7).

 **Connecter la sortie *link_en_o* au bus de contrôle du DECODE (bit 3).**

sel_rs	adr_reg_s	Remarque
0	rs_5_3_s	Partie de l'instruction à considérer
1	adr_reg_d	Copie de la valeur mise sur adr_reg_d_o
2	7	Utilisé lorsque BL msB
3	6	Utilisé lorsque BL lsB

TABLE 11 – Table de vérité de l'adresse du registre S

Remarque : L'instruction BL est séparée en deux sous-instructions lors de la compilation et va devoir accéder au registre PC ainsi qu'au registre LR (donc 6 et 7 dans notre cas). Ces valeurs sont donc des constantes car elles sont imposées par notre architecture.

sel_rs	sel_rd	adr_reg_d	Remarque
0	0	rd_2_0_s	Partie de l'instruction à considérer
0	1	rd_10_8_s	Partie de l'instruction à considérer
1	0	rd_2_0_s	Partie de l'instruction à considérer
1	1	rd_10_8_s	Partie de l'instruction à considérer
2	0	6	Utilisé lorsque BL msB
2	1	6	Utilisé lorsque BL msB
3	0	7	Utilisé lorsque BL lsB
3	1	7	Utilisé lorsque BL lsB

TABLE 12 – Table de vérité de l'adresse du registre D

sel_rn	adr_reg_n	Remarque
0	rn_8_6_s	Partie de l'instruction à considérer
1	rs_5_3_s	Partie de l'instruction à considérer

TABLE 13 – Table de vérité de l'adresse du registre N

2.2 Circuit decode_instr_splitter

Remarque : Analyser et tester.

Cette partie n'a pas besoin d'être modifiée car elle a déjà été réalisée mais il est important de comprendre qu'on prépare les diverses informations nécessaires dont nous aurons besoin pour exécuter les instructions que nous voulons supporter.

Question 2 :

Lors d'une opération de type "Addition/substraction (add/substract)", sur quelles sorties de *decode_instr_splitter* pourriez vous lire les informations indiquant les registres et offsets utilisés ?

Question 3 :

Lors d'une opération de type "Branchement/saut conditionnel (conditional branch)", sur quelles sorties de *decode_isntr_splitter* pourriez vous lire les informations indiquant les bits de condition et l'offset utilisés ?

Référez-vous au manuel **ARM7-TDMI-manual-pt3** que vous trouverez sur le site Cyberlearn afin d'identifier le formatage du set d'instruction THUMB.

2.3 Circuit main_control_unit

Remarque : Analyser et tester.

Cette partie n'a pas besoin d'être réalisée car elle a déjà été réalisée. Cette partie sert à fournir **tous** les flags des instructions détectées et ainsi construire les informations qui construiront les bus de contrôle.

2.4 Circuit opcode_supported_unit

Remarque : Implémenter et tester.

Cette partie est partiellement réalisée et vous montre comment les instructions sont détectées.

Ajouter le décodage des instructions suivantes pour pouvoir tester les accès à la mémoire de données :

- **strb_r_r_r** : Store Byte using 1 data reg, 1 address reg and 1 offset reg
- **ldrb_r_r_r** : Load Byte using 1 data reg, 1 address reg and 1 offset reg
- **strh_r_r_r** : Store Halfword using 1 data reg, 1 address reg and 1 offset reg, **not sign-extended**
- **ldrh_r_r_r** : Load Halfword using 1 data reg, 1 address reg and 1 offset reg, **not sign-extended**
- **strb_r_r_imm** : Store Byte using 1 data reg, 1 address reg and 1 immediate offset
- **ldrb_r_r_imm** : Load Byte using 1 data reg, 1 address reg and 1 immediate offset
- **strh_r_r_imm** : Store Halfword using 1 data reg, 1 address reg and 1 immediate offset
- **ldrh_r_r_imm** : Load Halfword using 1 data reg, 1 address reg and 1 immediate offset

Référez-vous au manuel **ARM7-TDMI-manual-pt3** que vous trouverez sur le site Cyberlearn afin de pouvoir identifier l'opcode des instructions.

 **Prêtez bien attention à l'ordre des différents bits de l'opcode dans le manuel.**

Question 4 :

Pour les 4 instructions travaillant uniquement avec des registres, quel est le nombre de bits de l'opcode ? Et pour les 4 instructions avec un offset immédiat ?

2.5 Circuit fetch_control_unit

Remarque : Implémenter et tester.

Compléter les *valeurs manquantes* du **Tableau 14** en vous aidant du manuel ARM7-TDMI-manual-pt3, puis **implémenter cette partie** :

Instruction	sel_cpsr	link_en	sel_PC	cond_en
lsl_r_r_imm_s	1	0	0	0
lsr_r_r_imm_s	1	0	0	0
asr_r_r_imm_s	1	0	0	0
add_r_r_r_s				
add_r_r_imm_s				
sub_r_r_r_s	1	0	0	0
sub_r_r_imm_s	1	0	0	0
mov_r_imm_s	1	0	0	0
add_r_imm_s	1	0	0	0
sub_r_imm_s	1	0	0	0
and_r_r_s	1	0	0	0
eor_r_r_s	1	0	0	0
lsl_r_r_s	1	0	0	0
lsr_r_r_s	1	0	0	0
asr_r_r_s	1	0	0	0
ror_r_r_s	1	0	0	0
neg_r_r_s	1	0	0	0
orr_r_r_s	1	0	0	0
mvn_r_r_s	1	0	0	0
strb_r_r_r_s	0	0	0	0
ldrb_r_r_r_s				
strh_r_r_r_s	0	0	0	0
ldrh_r_r_r_s	0	0	0	0
strb_r_r_imm_s	0	0	0	0
ldrb_r_r_imm_s	0	0	0	0
strh_r_r_imm_s	0	0	0	0
ldrh_r_r_imm_s	0	0	0	0
b_cond_s	0	0	1	1
b_incond_s	0	0	1	0
bl_msb_s	0	1	0	0
bl_lsb_s	0	0	1	0

TABLE 14 – Table de vérité du fetch_control_unit

Explications des signaux qui sortent de ce bloc :

- **sel_cpsr_o** : Flag qui met à jour le CPSR avec l'instruction courante
- **link_en_o** : Flag qui permet de sauver la valeur de PC dans le LR pour un long saut
- **sel_PC_o** : Flag qui indique qu'on va écrire dans le PC
- **cond_en_o** : Flag qui indique l'utilisation de la condition

2.6 Circuit decode_control_unit

Remarque : Implémenter et tester.

Compléter les *valeurs manquantes* du **Tableau 15** en vous aidant du manuel **ARM7-TDMI-manual-pt3**, puis **implémenter cette partie**.

Souvenez-vous des **Tableaux 11, 12 et 13** où les adresses de lecture et d'écriture sont définies en fonction des signaux *sel_x*, il faut maintenant créer ces signaux de sélection.

Instruction	sel_rs(0)	sel_rs(1)	sel_rn	sel_rd
lsl_r_r_imm_s	0	0	0	0
lsr_r_r_imm_s	0	0	0	0
asr_r_r_imm_s	0	0	0	0
add_r_r_r_s	0	0	0	0
add_r_r_imm_s				
sub_r_r_r_s	0	0	0	0
sub_r_r_imm_s	0	0	0	0
mov_r_imm_s				
add_r_imm_s	1	0	0	1
sub_r_imm_s	1	0	0	1
and_r_r_s	1	0	1	0
eor_r_r_s	1	0	1	0
lsl_r_r_s	1	0	1	0
lsr_r_r_s	1	0	1	0
asr_r_r_s	1	0	1	0
ror_r_r_s				
neg_r_r_s	0	0	0	0
orr_r_r_s	1	0	1	0
mvn_r_r_s	0	0	0	0
strb_r_r_r_s	0	0	0	0
ldrb_r_r_r_s	0	0	0	0
strh_r_r_r_s	0	0	0	0
ldrh_r_r_r_s	0	0	0	0
strb_r_r_imm_s	0	0	0	0
ldrb_r_r_imm_s	0	0	0	0
strh_r_r_imm_s	0	0	0	0
ldrh_r_r_imm_s	0	0	0	0
b_cond_s	0	1	0	0
b_incond_s	0	1	0	0
bl_msb_s	0	1	0	0
bl_lsb_s	1	1	0	0

TABLE 15 – Table de vérité du decode_control_unit

Explications des signaux qui sortent de ce bloc :

- **sel_rs_o** : bus 2 bits pour choisir Rs
- **sel_rn_o** : sélection Rn
- **sel_rd_o** : sélection Rd

2.7 Circuit reg_bank_control_unit

Remarque : Implémenter et tester.

Compléter les *valeurs manquantes* du **Tableau 16** en vous aidant du manuel ARM7-TDMI-manual-pt3, puis **implémenter cette partie** :

Instruction	reg_bank_wr
lsl_r_r_imm_s	
lsr_r_r_imm_s	1
asr_r_r_imm_s	1
add_r_r_r_s	1
add_r_r_imm_s	1
sub_r_r_r_s	1
sub_r_r_imm_s	1
mov_r_imm_s	1
add_r_imm_s	1
sub_r_imm_s	1
and_r_r_s	1
eor_r_r_s	
lsl_r_r_s	1
lsr_r_r_s	1
asr_r_r_s	1
ror_r_r_s	1
neg_r_r_s	1
orr_r_r_s	1
mvn_r_r_s	1
strb_r_r_r_s	0
ldrb_r_r_r_s	1
strh_r_r_r_s	
ldrh_r_r_r_s	1
strb_r_r_imm_s	0
ldrb_r_r_imm_s	1
strh_r_r_imm_s	0
ldrh_r_r_imm_s	1
b_cond_s	0
b_incond_s	0
bl_msb_s	1
bl_lsb_s	1

TABLE 16 – Table de vérité du reg_bank_control_unit

Explications des signaux qui sortent de ce bloc :

— **reg_bank_wr_o** : Flag qui indique une écriture dans un registre

2.8 Circuit execute_control_unit

Remarque : Implémenter et tester.

Implémenter les signaux *sel_operand_1* et *sel_operand_2* dans le circuit *execute_control_unit* à l'aide du **Tableau 17**. Les valeurs sont données en **décimales**. Les signaux restants sont déjà implémentés :

Instruction	sel_operand_1	sel_operand_2	sel_op_shift	sel_op_alu	sel_mult_2
lsl_r_r_imm_s	0	0	2	0	0
lsr_r_r_imm_s	0	0	3	0	0
asr_r_r_imm_s	0	0	1	0	0
add_r_r_r_s	0	0	0	1	0
add_r_r_imm_s	0	1	0	1	0
sub_r_r_r_s	0	0	0	2	0
sub_r_r_imm_s	0	1	0	2	0
mov_r_imm_s	1	3	0	1	0
add_r_imm_s	0	3	0	1	0
sub_r_imm_s	0	3	0	2	0
and_r_r_s	0	0	0	3	0
eor_r_r_s	0	0	0	7	0
lsl_r_r_s	0	0	2	0	0
lsr_r_r_s	0	0	3	0	0
asr_r_r_s	0	0	1	0	0
ror_r_r_s	0	0	4	0	0
neg_r_r_s	0	0	0	6	0
orr_r_r_s	0	0	0	4	0
mvn_r_r_s	0	0	0	5	0
strb_r_r_r_s	0	0	0	1	4
ldrb_r_r_r_s	0	0	0	1	4
strh_r_r_r_s	0	0	0	1	0
ldrh_r_r_r_s	0	0	0	1	0
strb_r_r_imm_s	0	2	0	1	4
ldrb_r_r_imm_s	0	2	0	1	4
strh_r_r_imm_s	0	2	0	1	0
ldrh_r_r_imm_s	0	2	0	1	0
b_cond_s	0	2	0	1	2
b_incond_s	0	2	0	1	1
bl_msb_s	0	2	0	1	5
bl_lsb_s	0	2	0	1	3

TABLE 17 – Table de vérité du execute_control_unit

Explications des signaux qui sortent de ce bloc :

- **sel_operand_1_o** : Sélection opérand 1 de l'execute
- **sel_operand_2_o** : Sélection opérand 2 de l'execute
- **sel_op_shift_o** : Sélection opération du shifter
- **sel_op_alu_o** : Sélection opération de l'ALU
- **sel_mult_2_o** : Sélection pour multiplicateur de l'execute
- **sel_shift_o** : Sélection de la valeur de shift

Pour *sel_shift_o*, on obtient la sortie à partir des instructions et de l'équation :

$$sel_shift_o = lsl_r_r_s + lsr_r_r_s + asr_r_r_s + ror_r_r_s$$

2.9 Circuit memory_access_control_unit

Remarque : Implémenter et tester.

Compléter les *valeurs manquantes* du **Tableau 18** en vous aidant du manuel ARM7-TDMI-manual-pt3, puis **implémenter cette partie** :

Instruction	sel_mem	str_data	ldr_data	byte
lsl_r_r_imm_s	0	0	0	0
lsr_r_r_imm_s	0	0	0	0
asr_r_r_imm_s	0	0	0	0
add_r_r_r_s	0	0	0	0
add_r_r_imm_s	0	0	0	0
sub_r_r_r_s	0	0	0	0
sub_r_r_imm_s	0	0	0	0
mov_r_imm_s	0	0	0	0
add_r_imm_s	0	0	0	0
sub_r_imm_s	0	0	0	0
and_r_r_s				
eor_r_r_s	0	0	0	0
lsl_r_r_s	0	0	0	0
lsr_r_r_s	0	0	0	0
asr_r_r_s	0	0	0	0
ror_r_r_s	0	0	0	0
neg_r_r_s	0	0	0	0
orr_r_r_s	0	0	0	0
mvn_r_r_s	0	0	0	0
strb_r_r_r_s	1	1	0	1
ldrb_r_r_r_s	1	0	1	1
strh_r_r_r_s	1	1	0	0
ldrh_r_r_r_s				
strb_r_r_imm_s				
ldrb_r_r_imm_s	1	0	1	1
strh_r_r_imm_s	1	1	0	0
ldrh_r_r_imm_s	1	0	1	0
b_cond_s	0	0	0	0
b_incond_s	0	0	0	0
bl_msb_s	0	0	0	0
bl_lsb_s	0	0	0	0

TABLE 18 – Table de vérité du memory_access_control_unit

Explications des signaux qui sortent de ce bloc :

- **sel_mem_o** : Flag qui indique que le bloc memory access est utilisé
- **str_data_o** : Flag qui indique qu'on veut sauver une donnée en mémoire
- **ldr_data_o** : Flag qui indique qu'on veut lire une donnée en mémoire
- **byte_o** : Flag qui indique un accès par Byte

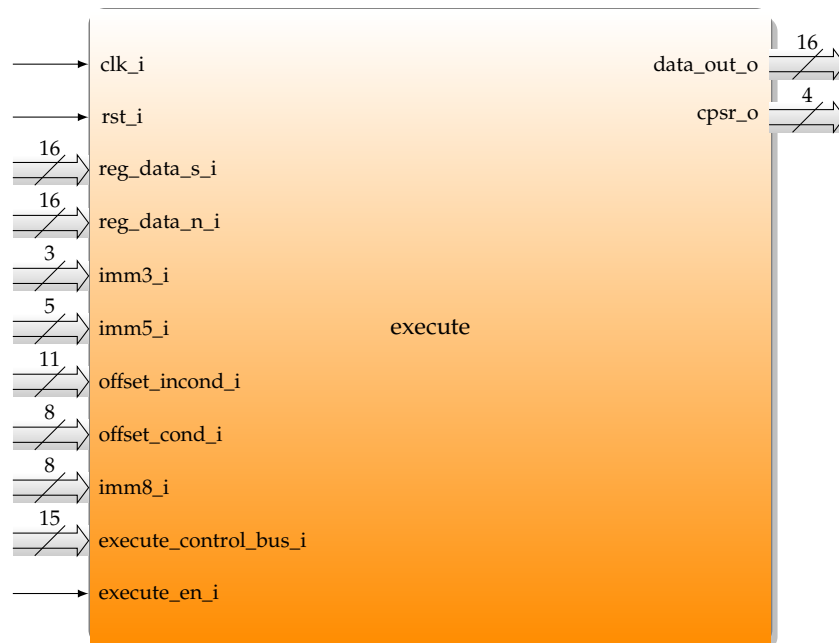
2.10 Circuit bus_constructor

Remarque : Analyser et tester.

Cette partie sert à construire les différents bus en suivant les tables fournies dans l'énoncé.

3 Bloc EXECUTE

Entité du bloc EXECUTE



Nom I/O	Description
clk_i	Entrée d'horloge
rst_i	Entrée du reset asynchrone
reg_data_s_i	Données du registre choisi par Rs
reg_data_n_i	Données du registre choisi par Rn
imm3_i	Valeur immédiate sur 3 bits
imm5_i	Valeur immédiate sur 5 bits
offset_incond_i	Valeur d'offset pour le saut inconditionnel
offset_cond_i	Valeur d'offset pour le saut conditionnel
imm8_i	Valeur immédiate sur 8 bits
execute_control_bus_i	Bus de contrôle du bloc execute
execute_en_i	Signal enable du bloc execute
data_out_o	Résultat en sortie du bloc execute
cpsr_o	Registre contenant l'état actuel du programme

Construction du bus de contrôle du bloc execute :

Position	Taille	Description
2-0	3	Sélection de l'opération du shift
5-3	3	Sélection de l'opération du bloc ALU
7-6	2	Sélection de l'opérande 2
8	1	Sélection de l'opérande 1
9	1	Sélection de la valeur d'entrée pour le shift
10	1	Flag qui met à jour le CPSR
13-11	3	Sélection de l'opération du bloc Mult_2
14	1	Pas utilisé

3.1 Réalisation de la sélection de l'opérande 2

Remarque : Analyser et tester.

Dans le circuit *execute*, la logique qui permet de donner la bonne valeur au bloc *alu* a été implémentée et doit répondre à la table de vérité suivante :

sel_operand_2_s	operand_2_s
0	reg_data_n_s
1	ext16_unsigned(imm3_s)
2	mult_2_out_s
3	ext16_unsigned(imm8_s)

Remarque : Le signal *mult_2_out_s* correspond à la sortie du bloc *mult_2*.

3.2 Réalisation de la sélection de l'opérande 1

Remarque : Analyser et tester.

Dans le circuit *execute*, la logique qui permet de donner la bonne valeur au bloc *alu* a été implémentée et doit répondre à la table de vérité suivante :

sel_operand_1_s	operand_1_s
0	reg_data_s_s
1	0x0000

3.3 Réalisation de la sélection de la valeur pour le shift

Remarque : Analyser et tester.

Dans le circuit *execute*, la logique qui permet de donner la bonne valeur de décalage au bloc *shift*.

La valeur du décalage pour le shift doit répondre à la table de vérité suivante :

sel_shift_i	sel_shift_data_s
0	imm5_i(3:0)
1	reg_data_m_i(3:0)

3.4 Réalisation du bloc shift

Remarque : Analyser et tester.

Afin de faire le lien avec cette partie, vous devez revenir à la logique que vous avez dû implémenter afin de gérer ce bloc dans le bus de contrôle (depuis le bloc décode).

Les différentes sortes de shift ont été créées afin de pouvoir exécuter les opérations supportées comme indiqué dans la table suivante.

sel_op_shift_s	shift_data_out_s
0	operand_s (bypass)
1	shift arithmétique (ASR)
2	shift logique à gauche (LSL)
3	shift logique à droite (LSR)
4	shift rotatif à droite (ROR)
5	0x0000 (pas assigné)
6	0x0000 (pas assigné)
7	0x0000 (pas assigné)

3.5 Réalisation du bloc ALU

Afin de faire le lien avec cette partie, vous devez revenir à la logique que vous avez dû implémenter afin de gérer ce bloc dans le bus de contrôle (depuis le bloc décode).

3.5.1 Operations arithmétiques

Remarque : Analyser et tester.

La partie arithmétique de votre processeur a été réalisée et donne le résultat de l'opération choisie correspondant au tableau suivant :

sel_op_alu_s	data_out_s
0	operand_1_s (bypass)
1	operand_1_s + operand_2_s
2	operand_1_s - operand_2_s
3	operand_1_s AND operand_2_s
4	operand_1_s OR operand_2_s
5	NOT operand_1_s
6	NEG operand_1_s ($\Rightarrow * -1$)
7	operand_1_s XOR operand_2_s

3.5.2 Gestion du carry

Remarque : Implémenter et tester.

Dans le circuit *alu*, créer le signal *carry* afin de donner l'information du carry de l'opération qui vient d'être exécutée. Cette information permettra de mettre à jour le CPSR et donc de pouvoir prendre des décisions par la suite (conditions).

3.5.3 Gestion de l'overflow

Remarque : Implémenter et tester.

Dans le circuit *alu*, créer le signal *overflow* afin de donner l'information de l'overflow de l'opération qui vient d'être exécutée. Ceci permet également de donner cette information au CPSR.

3.6 Implémentation du CPSR

Remarque : Implémenter et tester.

Implémenter le CPSR (Current Program State Register). Ce registre est composé de 4 bits, définis :

Position	Nom	Description
0	Z	Comparaison avec zéro
1	C	Indication du carry
2	N	Indication d'une valeur négative
3	V	Indication de l'overflow

Nous vous avons déjà préparé un bloc *zcnv_unit* qui permet de faire les diverses comparaisons qui permettent de construire ces flags.

Remarque : Il vous faut compléter ce bloc afin de créer les signaux Z, C, N et V.

Ce bloc *zcnv_unit* est déjà instanciée dans le circuit *execute*. Vous pouvez analyser comment il est utilisé dans ce circuit et à quel moment ce registre est mis à jour.

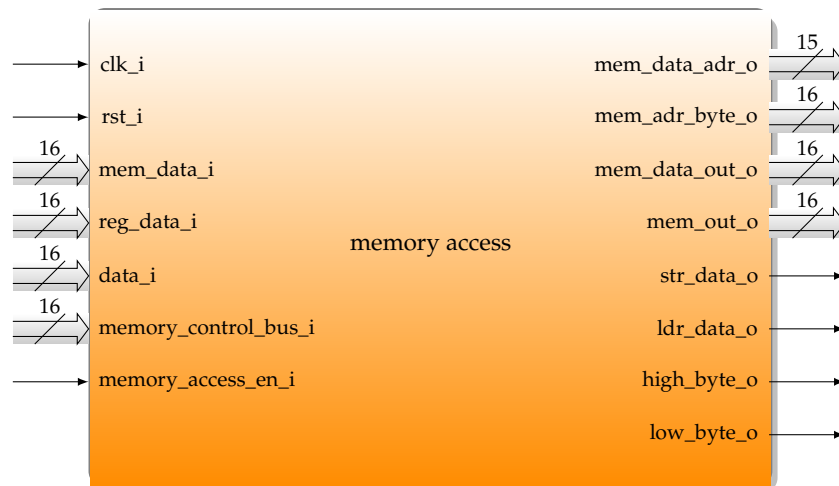
3.7 Implémentation du bloc *mult_2*

Remarque : Analyser et tester.

Cette partie n'a pas besoin d'être modifiée car elle a déjà réalisée mais il est important de comprendre que l'on doit donner une valeur cohérente en sortie lorsque l'on appelle la fonctionnalité de ce bloc.

4 MEMORY ACCESS

Entité du bloc MEMORY ACCESS



Nom I/O	Description
clk_i	Entrée d'horloge
rst_i	Entrée du reset asynchrone
mem_data_i	Donnée qui arrivent de la mémoire de données
reg_data_i	Donnée du registre choisi par sel_mem (Voir labo decode)
data_i	Donnée qui arrive depuis le bloc execute
memory_control_bus_i	Bus de contrôle du bloc Memory Access
memory_access_en_i	Signal enable du bloc Memory Access
mem_data_adr_o	Adresse dans la mémoire de données
mem_adr_byte_o	Adresse brute (uniquement pour DEBUG)
mem_data_out_o	Donnée qui part vers la mémoire de données
mem_out_o	Valeur de sortie qui va dans la banque de registres
str_data_o	Flag qui indique qu'on veut stocker une donnée en mémoire
ldr_data_o	Flag qui indique qu'on veut lire une donnée en mémoire
high_byte_o	Lors d'une écriture par byte on veut accéder la partie haute de la mémoire
low_byte_o	Lors d'une écriture par byte on veut accéder la partie basse de la mémoire

4.1 Implémentation du bloc memory access

Remarque : Analyser et tester.

Cette partie n'a pas besoin d'être modifiée car elle a déjà réalisée mais il est important de **tester** que tout se passe bien avec le reste du design.

4.2 Test des instructions d'accès à la mémoire de données

Utilisez les instructions qui accèdent à la mémoire afin de tester son bon fonctionnement.

Il est important de valider que les valeurs que l'on souhaite accéder transitent correctement au travers du design.

Par exemple, la valeur du registre à lire/écrire arrive sur la bonne connexion et les deux autres valeurs de registres forment bien la bonne adresse source/destination.

Vous devez indiquer les instructions que vous allez utiliser pour faire les accès à la mémoire de données.

5 Programme complet pour le processeur que vous avez réalisé

Voici la liste des instructions supportées par le processeur :

- lsl_r_r_imm_s
- lsr_r_r_imm_s
- asr_r_r_imm_s
- add_r_r_r_s
- add_r_r_imm_s
- sub_r_r_r_s
- sub_r_r_imm_s
- mov_r_imm_s
- add_r_imm_s
- sub_r_imm_s
- and_r_r_s
- eor_r_r_s
- lsl_r_r_s
- lsr_r_r_s
- asr_r_r_s
- ror_r_r_s
- neg_r_r_s
- orr_r_r_s
- mvn_r_r_s
- strb_r_r_r_s
- ldrb_r_r_r_s
- strh_r_r_r_s
- ldrr_r_r_r_s
- strb_r_r_imm_s
- ldrb_r_r_imm_s
- strh_r_r_imm_s
- ldrr_r_r_imm_s
- b_cond_s
- b_incond_s
- bl_msb_s
- bl_lsb_s

5.1 Programme qui teste les instructions supportées

Un code assembleur permettant de tester toutes les instructions supportées vous est fourni dans le workspace dans le fichier **main.S**.

Compilez-le et chargez-le dans la mémoire d'instruction comme vu lors du laboratoire fetch.

Voici le contenu de **main.S** :

main.S :

```

START :
MOV r0 , # 15
MOV r1 , # 26
MOV r3 , # 5
ADD r0 , r0 , # 17
ADD r2 , r0 , r 3
ADD r2 , #1
SUB r3 , r1 , r 2
SUB r4 , r1 , #2
SUB r1 , # 29

STEP_1 :
MOV r1 , # 4
STRH r3 , [ r 0 , # 8 ]
ORR r1 , r 0
LDRH r2 , [ r 0 , # 8 ]

STEP_2 :
STRB r2 , [ r 1 , # 2 ]
LDRB r0 , [ r 1 , # 2 ]

STEP_3 :
STRB r4 , [ r 0 , r 1 ]
LDRB r2 , [ r 0 , r 1 ]

STEP_4 :
STRH r3 , [ r 0 , r 1 ]
LDRH r4 , [ r 0 , r 1 ]
AND r0 , r 2
LSL r0 , r0 , #2
LSR r0 , r 0 , #3

STEP_5 :
ORR r0 , r 2
NEG r4 , r 0
EOR r4 , r 3
MOV r0 , # 2
ROR r2 , r 0
MVN r2 , r 3

STEP_6 :
ASR r2 , r 0
ASR r2 , r3 , # 2

STEP_7 :
MOV r0 , # 0
MOV r1 , # 0
ADD r2 , r0 , r 1
BEQ STEP_8
MOV r0 , # 12

STEP_8 :
B STEP_10

STEP_9 :
MOV r1 , # 0

.org 0 x60
STEP_10 :
MOV r0 , # 42
MOV r1 , # 60
NOP
NOP

```

5.2 Phase de test

⚠ Les résultats obtenus doivent être rendus. Le format du tableau ci-dessous doit être respecté. Les valeurs doivent être notées en hexadécimal.

Rendez vous dans le circuit *main_control_unit* (*Control_Unit_inst*). Exécutez le programme **main.S** et observez les sorties. Les valeurs des registres sont disponibles dans l'onglet register en bas à gauche de Logisim.

Pour chaque étape notée par les labels **STEP_XX** dans **main.S** dans le tableau ci-dessous, complétez les valeurs de *fetch_control_bus_o*, *decode_control_bus_o*, *reg_bank_control_bus_o*, *execute_control_bus_o*, *mem_control_bus_o* et les valeurs des registres.

Vérifiez aussi que l'instruction sélectionnée en sortie sur le circuit *opcode_supported_unit* correspond à l'instruction exécutée avant d'atteindre le label.

Etape	fetch_control_bus_o	decode_control_bus_o	reg_bank_control_bus_o	execute_control_bus_o	mem_control_bus_o	opcode_supported_unit
STEP_1	0x0024	0x0011	0x0002	0x04d0	0x0000	sub_r_imm_o
STEP_3	0x0020	0x0000	0x0002	0x2088	0x0005	ldrb_r_r_imm_s
STEP_4	0x0030	0x0000	0x0002	0x2008	0x000d	ldrb_r_r_r_s
STEP_7	0x0000	0x0000	0x0002	0x0401	0x0000	asr_r_t_imm_s
STEP_8	0x0003	0x0002	0x0000	0x1088	0x0000	b_cond_s
STEP_10	0x0001	0x0002	0x0000	0x0888	0x0000	b_incond_s

Etape	R0	R1	R2	R3	R4	SP	LR	PC
STEP_1	0x0020	0x001a	0x0026	0xfff4	0x0018	0x0000	0x0000	0x0010
STEP_3	0x0020	0x0024	0x00f4	0xfff4	0x0018	0x0000	0x0000	0x001c
STEP_4	0x0000	0x0024	0x00f4	0xfff4	0x0018	0x0000	0x0000	0x0020
STEP_7	0x0002	0x0024	0x000b	0xfff4	0x001c	0x0000	0x0000	0x003a
STEP_8	0x0000	0x0000	0x0000	0xfff4	0x001c	0x0000	0x0000	0x0042
STEP_10	0x0000	0x0000	0x0000	0xfff4	0x001c	0x0000	0x0000	0x0046

Rendu

Pour ce laboratoire, vous devez rendre une archive qui comporte les 2 noms de votre binôme (ex : **rendu_nom1_nom2.zip**) et qui est composée de :

- votre fichier *.circ*
- les valeurs que vous avez trouvé à l'étape 5.2
- les réponses aux questions (recommandé)

Votre rendu sera évalué sur le projet Logisim ainsi que les valeurs trouvées à l'étape 5.2. Les réponses aux questions seront corrigées mais pas évaluées.

CONSEIL : Faire une petite documentation sur cette partie vous fera directement un résumé pour l'examen.
