

Cahier des charges conception de processeur MIPS : 2019

ENSIMAG

Table des matières

1	Introduction	3
2	Objectifs du projet processeur	3
2.1	Calendrier	4
2.1.1	Organisation en séances	4
2.1.2	Objectifs par séance	4
2.2	Evaluation	4
2.2.1	Auto-évaluation	5
2.2.2	Evaluation partielle via une application web	5
2.2.3	Format du rendu	5
3	Architecture du processeur et son environnement	5
3.1	Partie Opérative	5
3.2	Partie Contrôle	5
3.3	Système simple	6
3.4	Système complet	7
4	Méthode de conception	7
4.1	Identification de l'instruction	7
4.2	Projection sur la PO	7
4.3	Ajout d'états dans la PC	8
4.4	Mise en œuvre de l'instruction	8
4.5	Ecriture d'un programme de test en langage d'assemblage	8
4.6	Validation : simulation et carte	9
5	Spécification des périphériques	9
5.1	Les périphériques	9
5.2	Organisation de la mémoire	9
5.3	Le bus	9
5.4	Les périphériques fournis	10
5.5	Mécanisme d'interruption à une source	11
A	Gestion du dépôt et validations automatiques	14
A.1	Dépôt et sources initiales	14
A.2	Validations automatiques	14
B	Notations	14
C	Organisation du projet	15
C.1	Répertoires et fichiers	15
C.2	Interface à la PO	16

D	Environnement de conception	17
D.1	make	17
D.2	Simulation avec make	17
D.3	Programmation du FPGA avec make	18
D.4	Fonctionnement de l'autotest	18
D.4.1	Principe	18
D.4.2	Syntaxe des commentaires à ajouter	18
D.4.3	Base de test et regression	19
D.5	Le simulateur VHDL	19
D.5.1	Mise en page des chronogrammes	20
D.5.2	Mise à jour du VHDL	20
E	Cross-compilation	20
F	Documentation	21
G	Encodage des instructions	22
H	Description des instructions	23

1 Introduction

Le but de ce projet est de construire un processeur MIPS en VHDL, avec pour objectif de pouvoir l'intégrer dans un système complet pour exécuter une application qui effectue un affichage sur écran VGA. Le processeur sera construit progressivement, en lui ajoutant des instructions. Les concepts fondamentaux des familles d'instruction seront abordés au cours du projet. La progression du projet sera régulière et des corrections types vous seront données au fur et à mesure. Vous aurez à compléter les instructions manquantes par vous-même.

Un mécanisme d'auto-évaluation vous sera fourni afin de pouvoir valider votre avancement sur la base d'un échéancier. Même si toutes les étapes n'exigent pas la même quantité de travail, il est important de fournir le travail personnel nécessaire en dehors des séances encadrées pour s'assurer de mener à bien ce projet.

Les [sources de départ](#) se trouvent sur un dépôt GIT qui servira aussi au suivi de votre progression. Pour les détails, voir la section [A](#).

2 Objectifs du projet processeur

Dans un premier temps, l'objectif du projet sera de concevoir un processeur MIPS capable d'exécuter une partie du jeu d'instructions. Vous partirez d'un canevas d'architecture composé d'une partie opérative et d'une partie contrôle. Vous aurez à compléter la partie contrôle en ajoutant les états nécessaires à l'exécution des instructions.

Les notations utilisées dans la suite sont données en annexe [B](#), et la description des noms de fichiers, composants ainsi que des signaux sont en annexe [C](#). L'environnement de conception est décrit en section [D](#). Les aspects techniques du cahier des charges sont détaillés en section [3.3](#).

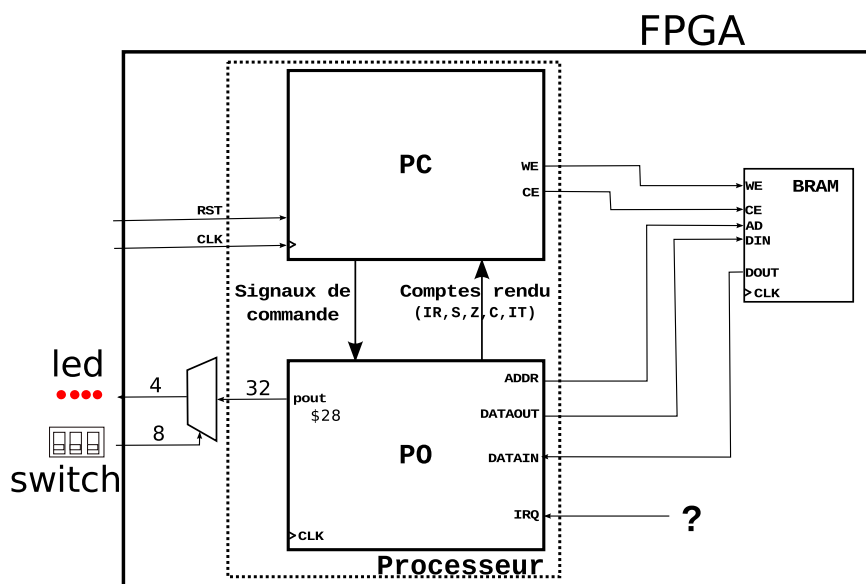


FIGURE 1 : Système simple, utilisation du debug

Afin de pouvoir valider le processeur, vous testerez le fonctionnement de programmes sur le processeur. Ces programmes seront stockés dans une mémoire du FPGA, connectée au processeur, contenant les instructions et les données. Le processeur sera également connecté à des LEDs sur la carte pour pouvoir observer son fonctionnement, ainsi qu'illustré sur la figure 1. L'exécution de certaines instructions permettra d'allumer et d'éteindre les LEDs.

Chaque instruction est à valider par simulation puis sur carte via l'écriture d'un programme de test en langage d'assemblage spécifique à cette instruction. Lorsqu'il y aura assez d'instructions, vous pourrez valider le processeur par de petites applications. Dans un second temps, vous pourrez utiliser votre processeur dans un environnement enrichi de périphériques : affichage sur LED, interfaces à des boutons poussoirs, interrupteurs

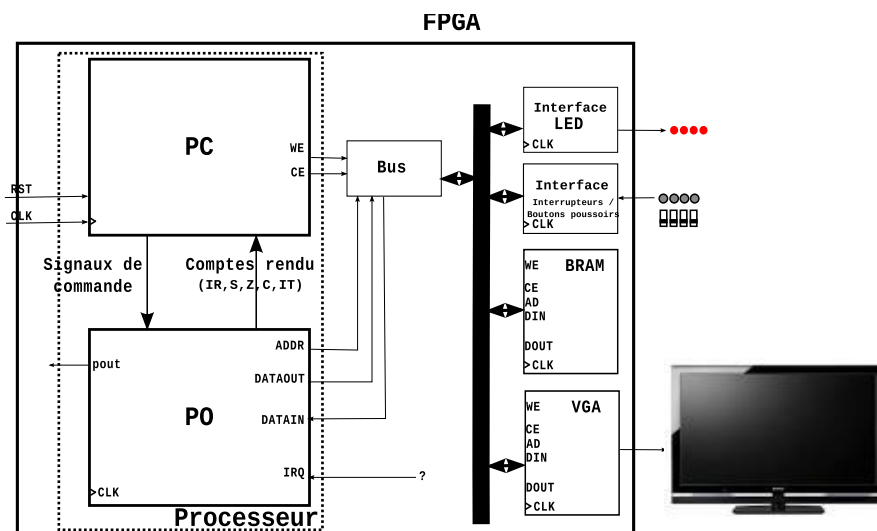


FIGURE 2 : Système avec périphériques sur bus

et écran VGA. Cet environnement, illustré sur la figure 2 et détaillé en section 3.4, vous permettra de faire fonctionner des applications graphiques.

2.1 Calendrier

2.1.1 Organisation en séances

La progression du projet est organisée par famille d'instructions. Chaque séance est consacrée à une ou plusieurs familles d'instructions. Chaque famille est représentée par une instruction typique, qu'il faut réaliser au cours de la séance. A défaut, une correction est fournie à la séance suivante aux groupes ne l'ayant pas réussi, afin de leur permettre de compléter la famille par eux même.

2.1.2 Objectifs par séance

Séance	Instruction typique	Famille	Programme fourni
1	ORI, LUI		
2	ADD SLL	ADDI, ADDU, ADDIU, AND, ANDI, NOR, OR, XOR, XORI, SUB, SUBU, SLIV, SRL, SRA, SRAV, SRLV	<i>chenillard_minimaliste.s</i> <i>compteur.s</i>
3	J BEQ JAL	JR BNE, BLEZ, BGTZ, BGEZ, BLTZ JALR, BLTZAL, BGEZAL	
4	LW, SW SLT	SLTI, SLTU, SLTIU sur système complet + IP Timer	<i>droite.s</i> <i>mipsinvader</i>
5	PO : <i>interruption 1 source</i> ERET Programme : démonstration d'interruption		

2.2 Evaluation

La notation de votre projet tiendra compte de la fonctionnalité de votre implémentation (nombre d'instructions fonctionnelles à la fin du projet), de votre progression (réalisation des instructions typiques dans les temps), de votre base de tests (quantité, pertinence, progression), de la qualité de votre rendu et des extensions réalisées.

2.2.1 Auto-évaluation

Pour évaluer vous-même votre projet, un mécanisme d'auto-évaluation vous est fourni et décrit en annexe [D.4](#). Il vous aidera à définir votre base de test au cours du projet et nous permettra de l'évaluer.

2.2.2 Evaluation partielle via une application web

Afin de valider votre progression, il faut utiliser l'application Web décrite en annexe [A](#). Cette application permet d'évaluer automatiquement l'état actuel de votre projet sur notre base de tests et d'en garder une trace. Pour fonctionner, elle récupère le contenu de votre dépôt GIT. Avant de l'utiliser, veillez à ce que votre projet fonctionne correctement (cf auto-évaluation) et que vous avez bien fait des "push" sur votre dépôt GIT. Vous disposez de 3 validations maximum par séance. Une partie de la note finale de votre projet est fixée grâce à l'application Web. Il est donc fortement recommandé de valider au moins une fois par séance.

2.2.3 Format du rendu

Vous devez rendre votre réalisation dans votre dépôt Git (bien veiller à pusher la bonne version). En pratique, on doit y trouver :

- un fichier README.TXT à la racine du dépôt dans lequel vous décrirez :
 - tout ce que vous avez implanté (étapes validées, extensions réalisées, etc.) ;
 - en cas d'extension, où (quels fichiers) elles sont réalisées et comment les valider ;
 - toutes les informations que vous jugerez pertinentes pour aider le correcteur à évaluer votre travail.
- dans le répertoire vhd, vos sources vhdl qui compilent correctement et implantent une réalisation validant le plus grand nombre de points verts sur l'application de validation ;
- dans le répertoire program, tous les tests intermédiaires et programmes que vous avez écrits pendant le projet pour tester votre réalisation. Donnez des noms à vos fichiers qui précisent l'instruction testée ou le programme (e.g. test_lui.s, program_chenille.s).

Attention : l'équipe de CEP a l'habitude d'utiliser des logiciels de détection de la fraude très efficace. Les copies avérées entre différentes équipes sont sanctionnées d'un 0/20 pour tous les membres des équipes impliquées.

3 Architecture du processeur et son environnement

Le processeur est construit sur le modèle PCPO. Les signaux échangés entre la PC et la PO sont regroupés dans des types enregistrement. Ces derniers sont décrits dans l'annexe [C.2](#). Par exemple, tous les signaux terminant par "_sel" correspondent à un fonctionnement similaire à un multiplexeur.

3.1 Partie Opérative

La figure [3](#) décrit l'architecture de la partie opérative fournie. Les blocs en jaune sont à ajouter afin de gérer les interruptions.

3.2 Partie Contrôle

Le graphe de contrôle de la PC initiale peut être représenté par la figure suivante :

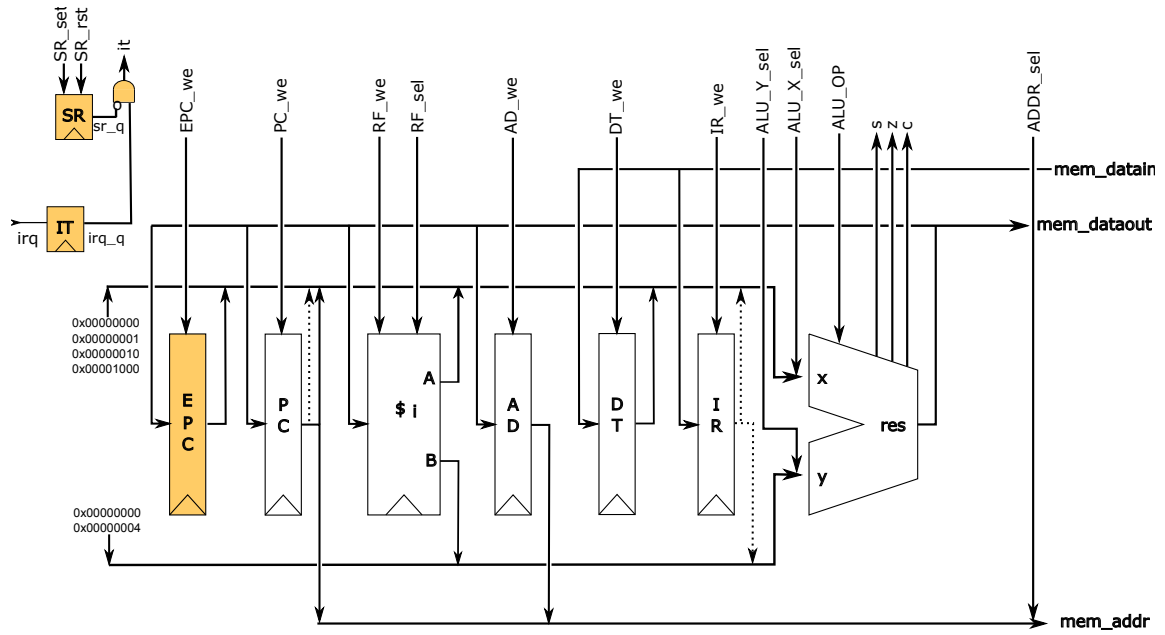
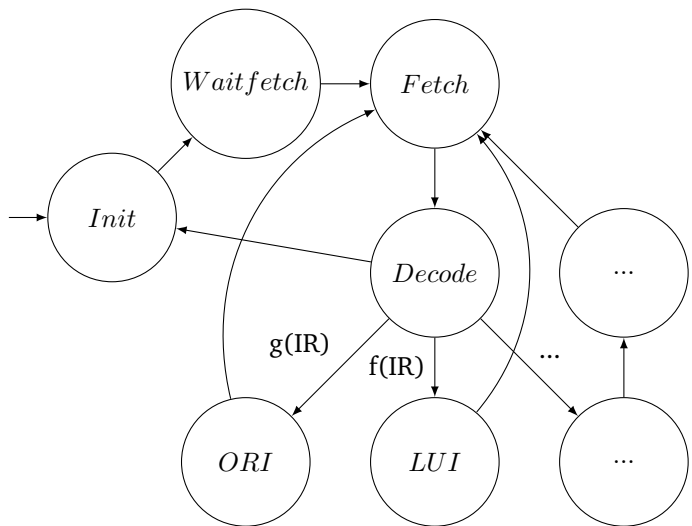


FIGURE 3 : Partie opérative



États	Opérations entre registres (RTL)
Init	$PC \leftarrow 0x0$
Waitfetch	$mem[PC]^a$
Fetch	$IR \leftarrow mem_datain$
Decode	$PC \leftarrow PC + 4$
ORI	$RT \leftarrow 0^{16} IR_{15...0} \text{ OR } RS; mem[PC]$
LUI	$RT \leftarrow IR_{15...0} 0^{16}; mem[PC]$

^aUn accès mémoire nécessite un cycle. On demande un accès à la valeur $mem[PC]$ qui sera fournit au cycle suivant sur le bus mem_datain .

Par défaut le décodage d'une instruction non implantée relance l'exécution de l'instruction à l'adresse 0x00, ce qui permet d'obtenir un comportement de type boucle infinie, sans avoir à implanter l'instruction de saut.

3.3 Système simple

Tant que les instructions **LW** et **SW** ne sont pas implantées, on fera fonctionner le processeur directement connecté à une mémoire, ainsi qu'illustré sur la figure 1. Dans ce système, le processeur, constitué de la PC et de la PO est connecté à une mémoire RAM de 8Ko qui contient le programme. Ce système est implanté dans le fichier `MMIPS_simple.vhd`, et on utilisera `TOP=MMIPS_simple` (voir l'utilisation du `Makefile` en annexe D).

Le debug, aussi bien en simulation que sur la carte, est permis grâce au signal `pout` qui sort de la PO. Ce signal est connecté au registre `$28` : une écriture dans ce registre met la valeur écrite sur le signal `pout`. Ainsi qu'indiqué sur le tableau 1, ce signal étant sur 32 bits et comme il n'y a que 4 LEDs, les interrupteurs permettent de sélectionner un des 8 mots de 4 bits de `pout` à afficher.

int ₂	int ₁	int ₀	LED _{3...0}
0	0	0	pout _{3...0}
0	0	1	pout _{7...4}
0	1	0	pout _{11...8}
0	1	1	pout _{15...12}
1	0	0	pout _{19...16}
1	0	1	pout _{23...20}
1	1	0	pout _{27...24}
1	1	1	pout _{31...28}

TABLE 1 : Configuration des interrupteurs pour affichage sur LEDs

3.4 Système complet

La figure 2 représente le système complet. Ce dernier intègre une mémoire RAM de 8Ko, ainsi que des périphériques d'entrée sorties. Ces périphériques sont des connections aux LEDs, interrupteurs et boutons poussoirs de la carte et au VGA. Ce système à périphériques est implanté dans le fichier `MMIPS_complet.vhd`, on utilisera `TOP=MMIPS_complet` (voir l'utilisation du `Makefile` en annexe D).

Les différents périphériques (ou entrées/sorties) sont vus comme de la mémoire par le processeur. Il n'est possible d'utiliser les périphériques que si on a réalisé les instructions `SW` et `LW`, respectivement d'écriture du contenu d'un registre vers la mémoire et d'écriture d'un mot mémoire dans un registre.

Ces périphériques sont interfacés avec le processeur par un **bus** (voir détails en 5.3), qui réalise un décodage des adresses. Lorsque le processeur réalise un accès vers une adresse particulière, le bus redirige l'accès vers le périphérique qui correspond à l'adresse indiquée.

La connexion des périphériques au bus et la configuration du décodage des adresses sont déjà effectuées (voir détails dans le fichier `MMIPS_complet.vhd` et explications en 5.1 et 5.3).

4 Méthode de conception

Cette section décrit étape par étape la démarche à suivre pour implanter une instruction ou groupe d'instructions. L'instruction `LUI` est prise comme exemple.

4.1 Identification de l'instruction

Accédez à la description de l'instruction dans l'annexe H par exemple en utilisant les hyperliens du tableau de la section 2.1.2. La description vous renseigne sur le format de l'instruction à implanter, son action, sa syntaxe et les opérations à réaliser.

Pour illustrer, l'analyse de la description de l'instruction `LUI` nous apprend qu'il faut mettre les 16 bits de poids faible du registre IR (champ `IMM16` dans le format I) dans les 16 bits de poids fort du registre RT et mettre à zéro ses bits de poids faibles.

4.2 Projection sur la PO

Il s'agit ensuite de décomposer les opérations à réaliser sur les composants (registres et opérateurs) de la PO (cf figure 3). Pour cela, il faut identifier les composants impliqués, trouver un chemin permettant de les relier, puis identifier les opérations à réaliser. L'annexe C.2 décrit l'interface complète de la PO. On y trouvera donc pour chaque opérateur les opérations possibles. A l'issue de cette étape, l'opération décrite par l'instruction est décomposée en une suite d'opérations RTL (i.e. opérations de registres à registres, qui se déroulent donc en un seul cycle).

Pour `LUI`, on peut repérer IR et le banc de registre sur la PO. Le seul chemin les reliant passe par l'UAL. Avec un décalage de 16 bits vers la gauche, cette dernière peut fournir le résultat escompté. L'annexe C.2 permet de valider ce choix, car il existe bien un décaleur logique à gauche (Valeur `A0_SLL`), l'opérande Y de l'UAL peut bien avoir les 16 bits de poids faible de IR en entrée (Valeurs `UYS_IR_imm16` ou `UYS_IR_imm16_ext`) et l'opérande X peut prendre la valeur 16 (Valeur `UXS_cst_x10`). De plus, on peut aussi vérifier que le registre RT est bien une destination valable du banc de registre pour le signal `RF_Sel` avec la valeur `RFS_RT`. L'opération

RTL identifiée est donc $RT \leftarrow IR_{15...0} \ll 16 || 0^{16}$ où $||$ représente la concaténation (voir annexe B pour le récapitulatif des notations).

4.3 Ajout d'états dans la PC

Chaque opération RTL est réalisé en un cycle et doit donc être associé à un état de la PC. Pour les instructions où plusieurs opérations RTL ont été identifiées à l'étape précédente, il faudra éventuellement les répartir sur plusieurs états consécutifs. Dans l'automate de la PC (cf section 3.2), cette suite d'états sera connectée à l'état Decode et rebouclera vers un état existant, qui permettra de charger correctement l'instruction suivante. Le passage de l'état Decode au premier état de l'instruction est décidé en fonction de l'encodage des instructions. L'annexe G donne le code correspondant à chaque instruction.

Avant de poser une question à un professeur sur une instruction, il est impératif d'avoir dessiné sur papier les états impliqués et les opérations RTL que vous comptez y réaliser.

Dans le cas du LUI, on n'a besoin que d'un seul état pour réaliser l'opération RTL identifiée. La détection du code 001111 sur les 6 bits de poids forts de IR suffit pour entrer dans cet état. De cet état, on peut passer à l'état Waitfetch, qui initiera correctement l'instruction suivante en demandant un accès en lecture à l'adresse PC de la mémoire. On peut remarquer que cette opération peut être réalisée sans conflit dans notre état LUI. En faisant ce choix, on peut directement passer à l'état Fetch comme proposé dans la section 3.2.

4.4 Mise en œuvre de l'instruction

Il s'agit de décrire dans le fichier VHDL *MMIPS_CPU_PC.vhd* le comportement spécifié dans les étapes précédentes. Dans ce fichier, il faut ajouter des noms d'états au type *State_type*, modifier l'état Decode pour qu'il détecte le codage de l'instruction concernée pour passer dans l'état correspondant et, ajouter pour tous les nouveaux états un ensemble de commande qui feront réaliser à la PO les opérations RTL concernées.

Pour LUI, après avoir déclaré *S_LUI* au type *State_type*, on pourra insérer le code VHDL suivant dans le processus qui décrit les fonctions de transition et de sortie :

```
when S_LUI =>
  -- RT <- IR(15:0) << 16
  cmd.ALU_X_sel <= UXS_cst_x10;
  cmd.ALU_Y_sel <= UYS_IR_imm16;
  cmd.ALU_OP <= AO_SLL;
  cmd.RF_Sel <= RFS_RT;
  cmd.RF_we <= true;
  -- mem[PC]
  cmd.mem_ce <= true;

  state_d <= S_Fetch;
```

Les champs, types et valeurs utilisés sont décrits dans l'annexe C.2. La description explicite complètement l'opération RTL définie à l'étape 2 en précisant la valeur prise par l'opérande X (le champ *ALU_X_sel* prend la valeur *UXS_cst_x10* correspondant à 16), celle de l'opérande Y, l'opération réalisée par l'UAL, le registre de destination dans le banc de registre et l'activation du banc de registre en écriture.

On notera qu'il est judicieux de commenter l'opération RTL réalisée en amont d'un groupe d'instructions VHDL.

4.5 Ecriture d'un programme de test en langage d'assemblage

Cette étape peut aussi être réalisée juste après l'étape 1, car il suffit de comprendre la syntaxe de l'instruction pour pouvoir la tester. Pour écrire vos tests, il faut utiliser le registre 28, qui est connecté sur une sortie dans notre projet. N'oubliez pas d'indiquer dans votre fichier de test les sorties attendues de manière à pouvoir utiliser le mécanisme d'autotest (annexe D.4).

Le fichier *test_lui.s* dans le répertoire *program* vous est donné à titre d'exemple.

4.6 Validation : simulation et carte

Chacun de vos tests doit être validé au minimum en simulation et sur carte lorsque c'est pertinent. Pour lancer la simulation, depuis le *répertoire du projet*, exécutez :

```
> make run_simu TOP=MMIPS_simple PROG=test_lui
```

Vous pourrez alors comparer dans le simulateur les valeurs obtenues à celles espérées. En cas d'erreur, le simulateur vous permettra de remonter à la source de l'erreur comme vu au premier semestre. En utilisant le mécanisme d'autotest, vous n'aurez à utiliser le simulateur que pour déboguer des erreurs dans votre code.

Pour tester sur la carte, exécuter :

```
> make run_fpga TOP=MMIPS_simple PROG=test_lui
```

Sur les LEDs de la carte, on voit les deux valeurs du test "en même temps" car les LEDs changent à une fréquence de l'ordre de la dizaine de MHz et la persistance rétinienne produit une apparence de superposition des valeurs. Il vaut mieux afficher une seule valeur sur les LEDs.

5 Spécification des périphériques

5.1 Les périphériques

L'ajout de périphériques au système est relativement simple. Chaque périphérique est considéré comme de la mémoire du point de vue du processeur. Concrètement, accéder à un périphérique consiste à réaliser des accès en lecture et écriture à des adresses spécifiques. Afin de différencier les accès à la mémoire de ceux aux périphériques, un élément dénommé *bus* est ajouté au système. Un bus intercepte les accès mémoire et sélectionne le périphérique concerné selon l'adresse de l'accès. Pour éviter les conflits, chaque périphérique se verra attribuer une plage d'adresses entre une adresse basse et une adresse haute. Le bus a connaissance de la liste des plages de tous les périphériques.

Pour insérer un périphérique dans le système, il suffit de lui associer une plage d'adresses et de modifier le paramétrage du bus. Tous les périphériques respectent un canevas d'interface, ce qui permet de les connecter directement au bus.

5.2 Organisation de la mémoire

La carte mémoire des périphériques est la suivante :

Périphérique	Accès	Plage d'adresses	Action
BRAM	RW	0x0-0x1FFF	Mémoire RAM pour les programmes, données et traitant d'interruption (optionnel).
LEDs	W	0x4000	Mot de 32 bits à afficher sur les LED
interrupteurs + boutons poussoirs	R	0x4004	Valeur des 4 interrupteurs dans l'octet de poids faible, valeur des 3 boutons poussoirs aux bits 16,17,18 et 0 sur les autres bits
timer	R W W	0x4010 0x4010 0x4014	Valeur du test du compteur Période du compteur Seuil du test
RAM vidéo	RW	0x80000-0xCAFFF	RAM Vidéo double port

Chacun de ces périphériques est décrit dans la section [5.4](#).

5.3 Le bus

Description

Le bus permet de connecter des périphériques au processeur. Il intercepte tous les signaux de lecture et écriture vers la mémoire et les redirige vers le périphérique adressé. Le bus sélectionne le périphérique adressé selon

l'adresse émise par le processeur. Il envoie un signal de sélection vers le périphérique et lui transmet l'adresse et les données sortantes du processeur. En lecture, le bus sélectionne le mot qui provient du périphérique adressé pour l'envoyer vers le processeur.

Du point de vue du processeur, le bus se comporte comme une mémoire et respecte le chronogramme d'une mémoire. Lors d'une lecture, le périphérique doit envoyer la donnée lue au cycle suivant l'adresse (mémoire synchrone).

Le bus réalise le décodage d'adresse *global* et un périphérique est identifié par son adresse de base et son adresse haute, dite la *plage* d'adresses. Si un périphérique a plusieurs registres entre ces deux adresses, c'est au périphérique de réaliser le décodage *local*. Le périphérique peut utiliser les bits de poids faible de l'adresse pour sélectionner le registre adéquat (pour exemple, regarder le code du timer décrit en section 5.4). Tous les périphériques doivent donc avoir l'interface suivante :

Port	Sens	Type	Description
clk	In	std_logic	
rst	In	std_logic	
ad	In	waddr	Adresse en provenance du bus
datai	In	w32	Donnée en provenance du bus
datao	Out	w32	Donnée vers le bus
we	In	std_logic	Signale une écriture '1' ou une lecture '0'

Paramétrage

Le bus est paramétré par les plages d'adresses des périphériques. Pour comprendre l'implantation du bus dans le fichier `MMIPS_complet.vhd`, voici quelques clés :

- La constante `nCE` correspond au nombre de périphériques
- La plage d'adresses a été configurée sur le bus
C'est-à-dire les adresses de base et haut ont été ajoutées aux tableaux `base` et `high` du bus.
- Les périphériques ont été instanciés et connectés aux signaux internes :
connexion au périphérique via les signaux `bus2IP_we`, `bus2IP_addr`, et `bus2IP_datai`, d'indice adéquat, ainsi que le signal `mem_dataout` en provenance du processeur.

Attention : ces signaux sont vus du bus : les entrées du bus sont les sorties du périphérique !

5.4 Les périphériques fournis

Horloge (Timer)

Fonctionnement L'horloge permet de gérer le temps indépendamment du temps d'exécution du programme. Ce périphérique, codé dans `IP_Timer.vhd`, contient un compteur qui s'incrémente à chaque cycle de l'horloge du circuit (50MHz sur notre carte). De façon à avoir un comportement périodique, le compteur repasse à zéro automatiquement lorsqu'il atteint une valeur maximale. Un registre booléen indique si le compteur est en dessous d'une valeur seuil ou non. Les valeurs de la *période* et du *seuil* sont paramétrables et peuvent être changées depuis le logiciel. Le registre booléen est lu depuis le logiciel, ce qui permet d'attendre que le compteur repasse à zéro.

Utilisation Selon la carte mémoire (c.f. section 5.2), pour modifier la *période*, il faut écrire à l'adresse `0x4010`. Pour modifier la valeur du seuil, il faut écrire à l'adresse `0x4014`. Pour lire la valeur du booléen, il faut lire à l'adresse `0x4010`.

Un exemple de boucle d'attente :

```
li $2, 14000          # période de la boucle dans $2
li $3, 0x4010         # l'adresse du registre de la période
sw $2, 0($3)          # écrit la période
li $2, 40             # valeur seuil pour le test
li $3, 0x4014         # l'adresse du registre du seuil
sw $2, 0($3)          # écrit le seuil
```

```

li $3, x4010          # l'adresse du registre du test (en lecture)
attente:
lw $2, 0($3)          # lit le résultat du test
beq $2, $0, attente   # boucle tant que le compteur est en dessous du seuil

```

Attention : la valeur du seuil doit être plus longue que la durée de la boucle d'attente. Dans le cas contraire, la boucle peut rater un passage à 'vrai' du booléen s'il se produit avant ou après la lecture.

VGA

Fonctionnement Le périphérique VGA, codé dans `IP_VGA.vhd`, permet d'afficher une image sur un écran, ainsi qu'illustré sur la figure 2, page 4. L'image est une image de taille 320x240 où chaque pixel est codé sur 16 bits. Le périphérique VGA contient une mémoire double port lue par la sortie VGA et lue/écrite par le processeur. Vu du processeur, le périphérique VGA est une mémoire dont la première adresse est `0x80000`. A partir de cette adresse, chaque mot contient un pixel sur les 16 bits de poids faibles et des zéros sur les bits de poids forts. L'écran est balayé de gauche à droite, puis de haut en bas. On peut donc accéder au pixel de coordonnées (x, y) à l'adresse $0x80000 + 4 * (y * 320 + x)$.

5.5 Mécanisme d'interruption à une source

Le mécanisme d'interruption se déroule en 3 étapes :

- **Départ en interruption** : lorsque le signal d'interruption est valide, le processeur interrompt le programme en cours.
- **Traitement de l'interruption** : Le processeur exécute alors le programme associé à l'interruption, que l'on appelle un *traitant d'interruption* ou encore *vecteur d'interruption*.
- **Retour d'interruption** : A la fin du traitant, le processeur reprend le programme qui a été interrompu. Le traitant d'interruption se termine par une instruction spécifique qui permet le retour à l'instruction interrompue.

Pour implanter le mécanisme d'interruption, il faut effectuer des modifications des parties opérative et contrôle. L'interruption est une demande extérieure d'un traitement spécial : c'est l'entrée `irq` du CPU qui permet de prendre en compte cette demande extérieure. Pour rappel, l'interruption n'est prise en compte que si le mécanisme d'interruption est autorisé : c'est ce que l'on appelle démasquer une interruption. Il faut donc ajouter dans votre processeur tout le mécanisme de traitement d'une interruption : masquage, sauvegarde de PC dans EPC, exécution du traitant, retour à l'instruction précédant l'interruption, démasquage.

Dans la partie opérative, à partir du schéma de la PO en figure 3, on ajoute :

- Les registres et bascules (dans la partie synchrone) :
 - EPC pour sauvegarder la valeur du PC (`EPC_q`)
 - SR pour masquer/démasquer les interruptions (`SR_q`)
 - IT pour signaler la demande d'interruption en provenance de l'extérieur (`irq_q`).

Attention : Le signal de demande d'interruption `irq_q` doit passer par un registre dans la PO, car, s'il est utilisé directement par de la combinatoire, il peut y avoir une violation de temps de propagation. En effet, lorsque le signal d'interruption provient de l'extérieur, sa date d'arrivée est aléatoire par rapport au front d'horloge, ce qui provoque des aléas pendant le front et génère une violation des contraintes temporelles sur les signaux. En général, tout signal provenant de l'extérieur doit être resynchronisé par une bascule D avant de pouvoir être utilisé dans le reste du système.
- Les opérations (dans la partie combinatoire) et les signaux de commande et états correspondants (pour communiquer avec la PC) :
 - modification de la valeur EPC (activée par le signal de commande `cmd.EPC_we`)
 - mise à 1 de SR pour masquer les interruptions (activée par le signal de commande `cmd.SR_set`)
 - mise à 0 de SR pour démasquer les interruptions (activée par le signal de commande `cmd.SR_reset`)

- détection d'une interruption valide (mise à jour du signal d'état `status.it`)

N'oubliez pas d'ajouter les signaux de contrôle associés en créant des champs dans les types `MMIPS_PO_cmd` et `MMIPS_PO_Status` du fichier `MMIPS_pkg.vhd`.

Dans la partie contrôle, il y a trois étapes à implanter (évaluer pour chacune s'il est nécessaire d'ajouter des états et combien) :

- Détection d'une interruption

Dans un (ou des états), prendre en compte le signal `status.it` pour lancer le départ en interruption. Il y a deux stratégies pour gérer les interruptions. La première consiste à détecter le départ en interruption dans un état par lequel on est certain de passer (conseillé). La seconde consiste à le faire à la fin de chaque instruction.

- Départ en interruption

Après avoir détecté une interruption, il faut :

- Sauvegarder le registre `PC` dans le registre `EPC`.
`PC` est sauvegardé afin de pouvoir relancer ultérieurement l'instruction arrêtée, lors du retour d'interruption.
- Masquer les interruptions pour ne pas être interrompu à nouveau.
Mettre `SR` à '1' pour inhiber toute nouvelle interruption.
- Sauter à l'adresse du traitant d'interruption : i.e. mettre `PC` à la valeur `0x00001FFC` (`UXS_IT_vec`)¹

- Retour d'interruption

Ajouter l'instruction `ERET`, qui réalise le retour d'interruption :

- Restaure `PC` à la valeur `EPC` ;
- Démasque le registre `SR` en le remettant à '0'

Cette instruction `ERET` doit se trouver à la fin de tout traitant d'interruption.

Avec ces modifications votre processeur MIPS peut gérer une interruption, en complément, il faut une façon de demander une interruption depuis l'extérieur du processeur et le code du traitant d'interruption. Pour demander une interruption au processeur, on vous propose d'utiliser l'environnement `MMIPS_simple` comme top. Ce dernier intègre le composant `IP_ITPush` qui vous permet de mettre le signal `irq` à 1 lors d'un appui sur le bouton poussoir 1 (`BTN1`, le second en partant de la droite). Attention ! Le banc de simulation fourni ne simule pas d'appui sur ce bouton ! Pour obtenir un banc de simulation générant des entrées satisfaisantes pour tester votre mécanisme d'interruption, il vous faudra modifier le fichier `bench/tb_MIPS_simple.vhd` en faisant passer le signal `push` à 1 après que le processeur ait déjà exécuté un certain nombre d'instructions du programme principal (par exemple en passant `push` à 1 au bout de 50 cycles d'horloge, et en le repassant à 0 3 cycles d'horloge plus tard). Pour le traitant d'interruption, il faut que le code commence en mémoire à l'adresse `0x00001FFC` (directive `.org` en assembleur). Comme il s'agit du dernier emplacement de la mémoire, placez-y un saut vers votre traitant.

Attention : le traitant d'interruption pouvant utiliser les mêmes registres que le programme principal, il est nécessaire de les sauvegarder dans la pile dès le début du traitant d'interruption puis les restaurer à la fin. De plus, ne pas oublier de terminer par l'instruction `ERET`.

Un programme de test pour les interruptions ressemblera à quelque chose de ce type :

```
.text
.org 0
progPrincipal:
    or $2,$0,$0          # Le programme principal est un boucle infini
                        # Par exemple, ici, un compteur dans $2 initialisé à 0
boucle_infinie:
    addi $2,$2,1          #
```

¹Normalement le vecteur d'interruption MIPS est à l'adresse `0x80000180` mais on ne peut pas adresser autant de mémoires sur les FPGA.

```

    or $28,$2,$0      # Sort la valeur du compteur sur le port (=$28)
    j boucle_infinie  #
traitantIT:
    ori $2,$0,33      # En cas d'interruption le compteur est ici modifié pour voir que
    eret              #    l'interruption a bien été prise en compte

    .org 0x1FFC        # Si une interruption arrive le processeur va à l'adresse 0x1FFC
    j traitantIT       # Il faut donc à cette adresse le code du programme d'interruption
                      # Mais comme il s'agit de la dernière instruction possible,
                      #    on place un saut à l'adresse du programme d'interruption

```

A Gestion du dépôt et validations automatiques

A.1 Dépôt et sources initiales

Vous devrez utiliser Git pour récupérer les sources de départ ainsi que pour valider votre réalisation au fur et à mesure que vous implantez des étapes. Vous pouvez aussi vous en servir pour effectuer des sauvegardes, vu que votre dépôt Git sera localisé physiquement sur le serveur `depots.ensimag.fr`.

Pour récupérer les sources initiales, vous devez d'abord exécuter la commande

```
git clone LOGIN1@depots:/depots/2018/cep/cep_LOGIN1_LOGIN2
```

Cette commande crée un sous-répertoire `cep_LOGIN1_LOGIN2` dans lequel vous trouverez les sources de départ. LOGIN1 et LOGIN2 sont les logins des membres du binôme par ordre alphabétique.

A.2 Validations automatiques

Pour vous aider à mettre au point votre processeur, ainsi que pour nous permettre d'évaluer votre progression et votre travail, on fournit un mécanisme d'évaluation semi-automatique de votre code VHDL basé sur une base de tests standards.

Pour faire passer ces tests, vous devez déposer vos fichiers dans le dépôt Git sur `depots` : pour cela, vous utiliserez les commandes suivantes :

```
git add <fichiers modifiés>
git commit -m "un commentaire décrivant les modifications"
git push
```

La commande `commit` enregistre vos modifications dans votre dépôt de travail local (i.e. sur le PC sur lequel vous travaillez) alors que la commande `push` envoie ces modifications sur `depots`.

Attention : Seuls les fichiers `MMIPS_CPU.vhd`, `MMIPS_CPU_PC.vhd` et `MMIPS_CPU_PO.vhd` sont analysés par l'outil de validation. Mais vous pouvez déposer tous vos fichiers dans le dépôt `depots` tout de même car cela permettra d'en garder une sauvegarde. Comme précisé plus haut, l'interface du composant `MMIPS_CPU` ne doit pas être modifiée.

Une fois les fichiers déposés, vous pouvez accéder à l'outil de validation à l'adresse suivante : <https://eval.ensimag.fr/cep/> et vous connecter avec vos login et mot de passe habituels.

Note : cette adresse n'est accessible que depuis les machines raccordées au réseau de l'Ensimag. Si vous souhaitez y accéder depuis l'extérieur ou via le wifi, utilisez le VPN.

Une fois connecté à l'application il vous suffit de cliquer sur le lien "Lancer la validation de la séance". Après quelques minutes, une page vous indiquera quelles instructions sont validées.

B Notations

<code>=</code>	test d'égalité
<code>+</code>	addition entière en complément à deux
<code>-</code>	soustraction entière en complément à deux
<code>×</code>	multiplication entière en complément à deux
<code>÷</code>	division entière en complément à deux
<code>mod</code>	reste de la division entière en complément à deux
<code>and</code>	opérateur et bit-à-bit
<code>or</code>	opérateur ou bit-à-bit
<code>nor</code>	opérateur non-ou bit-à-bit
<code>xor</code>	opérateur ou-exclusif bit-à-bit
<code>mem[a]</code>	contenu de la mémoire à l'adresse <i>a</i>
<code>←</code>	assignation
<code>⇒</code>	implication
<code> </code>	concaténation de chaînes de bits
<code>xⁿ</code>	réplication du bit <i>x</i> dans une chaîne de <i>n</i> bits. Notons que <i>x</i> est un unique bit
<code>x_{p...q}</code>	sélection des bits <i>p</i> à <i>q</i> de la chaîne de bits <i>x</i>

Certains opérateurs n'étant pas évidents, nous donnons ici quelques exemples.

Posons 0001101101001000 la chaîne de bit x , qui a une longueur de 16 bits, le bit le plus à droite étant le bit de poids faible et de numéro zéro, et le bit le plus à gauche étant le bit de poids fort et de numéro 15. $x_{6...3}$ est la chaîne 1001. x_{15}^{16} crée une chaîne de 16 bits de long dupliquant le bit 15 de x , zéro dans le cas présent. $x_{15}^{16} \parallel x_{15...0}$ est la valeur 32 bits avec extension de signe d'un immédiat en complément à deux de 16 bits.

C Organisation du projet

C.1 Répertoires et fichiers

Les différents fichiers du projet sont rangés dans les répertoires suivants :

Répertoires utiles	
vhd	Sources VHDL du processeur MIPS et ses périphériques
vhd/bench	Sources VHDL des environnements de simulation
program	Les sources des programmes en langage d'assemblage
Divers (A ne pas modifier)	
bin	Différents scripts et programmes pour gérer le projet
config	Fichiers de configurations des outils de CAO
.CEPcache	Répertoire caché de travail

Les fichiers VHDL de la liste ci-dessous sont utilisés au cours du projet. La mention **top** indique les fichiers qui contiennent une entité de plus haut niveau. Une entité "top" correspond à l'interface externe du FPGA. Elle contient les entités internes du projet. À chaque fichier "top" correspond un environnement de simulation. Certains fichiers seront à compléter au fur et à mesure du projet.

Module			Description
MMIPS_pkg.vhd			Bibliothèque contenant les déclarations des types utilisés dans le projet
MMIPS_CPU.vhd			Assemblage PC+PO
MMIPS_CPU_PC.vhd		A compléter	PC
MMIPS_CPU_PO.vhd		A compléter	PO
MMIPS_simple.vhd	top		Processeur MIPS + debug
MMIPS_complet.vhd	top		Processeur MIPS + Périphériques
MMIPS_bus.vhd			Gère le décodage d'adresse pour les périphériques
IP_LED.vhd			Périphériques sortie LED
IP_PIN.vhd			Périphérique bouton poussoir + interrupteurs
IP_Timer.vhd			
IP_VGA.vhd			
IP_ITPush.vhd			Pour déclencher une IT depuis un bouton poussoir

Compléments

RAM_Video_data.vhd		Généré automatiquement	Mémoire double port pour vidéo
RAM_Video_mem.vhd			Contient l'image à afficher
RAM_PROG_data.vhd		Généré automatiquement	Mémoire SRAM simple port pour processeur
RAM_PROG_mem.vhd			Contient les instructions après assemblage
VGA_gene_sync.vhd			Générateur de la synchronisation VGA

Le fichier MMIPS_CPU_PC.vhd contient le début de la machine à état du processeur.

Dans l'objectif de laisser à la synthèse logique le choix de l'encodage optimal, les commandes des multiplexeurs sont implantées de façon "abstraite" à l'aide de types énumérés. Cela permet également d'analyser simplement les chronogrammes car les valeurs énumérées "parlent d'elles-mêmes".

C.2 Interface à la PO

Les signaux de commandes de la PO sont regroupés dans une structure de type `MMIPS_PO_cmd`, définie dans le fichier `MMIPS_pkg.vhd`. Voici les différents champs de cette structure :

Champ	Type VHDL	Rôle
ALU_X_Sel	UXS_type	Sélection de l'opérande X sur l'ALU
ALU_Y_Sel	UYS_type	Sélection de l'opérande Y sur l'ALU
ALU_OP	AO_type	Sélection de l'opération effectuée par l'ALU
ALU_extension_signe	<code>std_logic</code>	'1' si les opérations arithmétiques sont effectuées avec extension de signe sur 33 bits, '0' sinon.
RF_Sel	RF_sel_type	Sélection du numéro de registre destination
RF_we	<code>boolean</code>	Valide l'écriture dans RF
EPC_we	<code>boolean</code>	Valide l'écriture dans EPC
PC_we	<code>boolean</code>	Valide l'écriture dans PC
AD_we	<code>boolean</code>	Valide l'écriture dans AD
DT_we	<code>boolean</code>	Valide l'écriture dans DT
IR_we	<code>boolean</code>	Valide l'écriture dans IR
ADDR_sel	ADDR_select	Sélection de l'adresse vers la mémoire
mem_we	<code>boolean</code>	Valide une écriture dans la mémoire
mem_ce	<code>boolean</code>	Valide une transaction vers la mémoire (lecture ou écriture)

Les types utilisés dans cette structure sont également définis dans le fichier `MMIPS_pkg.vhd` comme spécifié ci-dessous :

`UXS_type` est le type énuméré utilisé pour sélectionner la valeur à fournir sur l'opérande X de l'UAL.

Valeur	Sémantique
UXS_RF_RS	Port A du banc de registre pointé par $IR_{25...21}$ (RS)
UXS_PC	Registre PC
UXS_EPC	Registre EPC
UXS_DT	Registre DT
UXS_cst_x00	Constante $0x00000000$
UXS_cst_x01	Constante $0x00000001$
UXS_cst_x10	Constante $0x00000010$
UXS_IT_vec	Constante $0x00001FFC$
UXS_PC_up	$PC_{31...28} \parallel 0^{28}$
UXS_IR_SH	$0^{27} \parallel IR_{10...6}$

`RF_sel_type` est le type énuméré utilisé pour sélectionner le registre de destination.

Valeur	Sémantique
RFS_RD	$IR_{15...11}$ (RD)
RFS_RT	$IR_{20...16}$ (RT)
RFS_31	registre R31

`ADDR_select` est le type énuméré utilisé pour la sélection de l'origine de l'adresse vers la mémoire.

Valeur	Sémantique
ADDR_from_PC	Registre PC
ADDR_from_AD	Registre AD

`UYS_type` est le type énuméré utilisé pour sélectionner la valeur à fournir sur l'opérande Y de l'UAL.

Valeur	Sémantique
UYS_IR_imm16	$0^{16} \parallel IR_{15...0}$
UYS_IR_imm16_ext	$IR_{15}^{16} \parallel IR_{15...0}$
UYS_IR_imm16_ext_up	$IR_{15}^{14} \parallel IR_{15...0} \parallel 0^2$
UYS_IR_imm26	$0^4 \parallel IR_{25...0} \parallel 0^2$
UYS_RF_RT	Port B du banc de registre pointé par $IR_{20...16}$ (RT)
UYS_cst_x00	Constante $0x00000000$
UYS_cst_x04	Constante $0x00000004$

`AO_type` est le type énuméré utilisé pour sélectionner l'opération à réaliser par l'UAL.

Valeur	Sémantique
AO_plus	$RES \leftarrow (ext(X) \parallel X) + (ext(Y) \parallel Y)$
AO_moins	$RES \leftarrow (ext(X) \parallel X) - (ext(Y) \parallel Y)$
AO_and	$RES \leftarrow X \text{ and } Y$
AO_or	$RES \leftarrow X \text{ or } Y$
AO_xor	$RES \leftarrow X \oplus Y$
AO_nor	$RES \leftarrow X \text{ nor } Y$
AO_SLL	$RES \leftarrow Y \ll X_{4...0}$ (logique)
AO_SRL	$RES \leftarrow Y \gg X_{4...0}$ (logique)
AO_SRA	$RES \leftarrow Y \ggg X_{4...0}$ (arithmétique)

La fonction $ext(a)$ étend le bit de signe ou non selon le signal `ALU_extension_signe`.

La PO retourne un ensemble de signaux d'états (*status*), regroupés dans la structure `status` de type `MMIPS_PO_status`. Les différents champs sont les suivants :

Champ	Type VHDL	Valeur
IR	w32	L'instruction en cours
s	boolean	Le bit de signe du résultat de l'ALU (bit 31)
c	boolean	Le bit de retenue du résultat de l'ALU (bit 32)
z	boolean	true si le résultat de l'ALU vaut 0, false sinon

Le type `w32` est un vecteur de 32 bits.

Gestion du signe et débordements `ALU_extension_signe` permet de gérer les débordements des opérations arithmétiques sur 32 bits. En réalité l'ALU réalise des opérations avec un bit supplémentaire, donc sur 33 bits. En effet, le résultat d'une addition (ou soustraction) sur 32 bits donne un résultat sur 33 bits sans débordements. Les arguments de l'addition (ou soustraction) sont sur 33 bits et le bit supplémentaire peut être soit '0', lorsque `ALU_extension_signe` vaut '0', soit une copie du bit 31 de chaque argument lorsque `ALU_extension_signe` vaut '1'.

Les débordements sont donc gérés selon le type des arguments (signé ou non-signé) et d'après les champs du signal `status` qui renseignent sur le signe du résultat et la retenue sortante. Il vous est laissé en exercice de trouver les conditions qui conduisent à un débordement.

D Environnement de conception

D.1 make

Un **Makefile** regroupe l'ensemble des actions effectuées au cours du projet. Ces commandes sont à lancer dans le répertoire du projet. Par exemple, la commande

```
make clean
```

permet de nettoyer votre répertoire de travail.

D.2 Simulation avec make

Pour lancer la simulation de l'entité `<top>`, depuis le répertoire du projet, exécutez la commande :

```
make run_simu [TOP=<top>] [SIM_TIME=n] [PROG=<prog>]
```

ou, pour simplement compiler le VHDL sans lancer le simulateur :

```
make compil [TOP=<top>] [SIM_TIME=n] [PROG=<prog>]
```

Les arguments entre `[]` sont optionnels :

`TOP=<top>` Sélectionne l'entité à simuler :

`<top>` est à choisir parmi : `MMIPS_simple`, `MMIPS_complet`

Valeur par défaut : `MMIPS_simple`

Exemple : `TOP=MMIPS_simple`

`SIM_TIME=n` Durée de la simulation en nanosecondes. `n` est sans unité.

Valeur par défaut : 1000

Exemple : `SIM_TIME=1000`

`PROG=<prog>` Initialise la mémoire programme avec le programme `<prog>` :

S'il existe un programme assembleur `<prog>.s` dans le répertoire

program, il est d'abord assemblé.

S'il existe un exécutable `<prog>.elf`, ce dernier est directement utilisé.

Valeur par défaut : `PROG=test_lui`

Exemple : `PROG=test_ori_a`

D.3 Programmation du FPGA avec make

Les arguments optionnels utilisés dans les commandes suivantes ont la même signification que dans la section précédente.

Pour télécharger le fichier de configuration sur le FPGA, lancer :

```
make run_fpga [TOP=<top>] [PROG=<prog>]
```

Pour générer le fichier de configuration (bitfile), sans lancer la configuration du FPGA, pour faire des essais.

```
make bitfile [TOP=<top>] [PROG=<prog>]
```

D.4 Fonctionnement de l'autotest

D.4.1 Principe

L'autotest permet de vérifier automatiquement que votre processeur exécute correctement un programme en langage d'assemblage. Ce dernier est enrichi de commentaires indiquant les valeurs attendues en sortie. Le mécanisme d'autotest vérifie que les valeurs produites à l'exécution du programme sont conformes à celles attendues. Si c'est le cas, l'autotest signale que le test est passé (PASSED). Si une valeur en sortie du processeur est différente de la valeur attendue, la simulation s'arrête et signale une erreur (FAILED). Enfin, si les résultats n'arrivent pas dans le temps imparti, le test terminera avec le message TIMEOUT.

D.4.2 Syntaxe des commentaires à ajouter

- Nombre de cycles maximal de la simulation

```
# max_cycle <n>
```

La simulation s'arrête au bout de <n> cycles d'horloge.

Exemple :

```
# max_cycle 50
```

- Spécification d'une suite de valeurs de sorties attendues

```
# pout_start
# <s0> [x]
# <s1> [x]
# ...
# pout_end
```

La suite peut être vide. Les valeurs <sn> sont en hexadécimal sur 32 bits. Ces valeurs sont comparées aux valeurs en sortie de pout à chaque écriture du registre 28.

Le caractère x est optionnel. Sa présence indique que la valeur peut être répétée plusieurs fois (par exemple si on écrit dans le registre 28 dans une boucle d'attente). Exemple :

```
# pout_start
# 000000AD
# 000000EF x
# 0000000A
# 000000FA
# pout_end
```

- Génération d'un signal d'interruption

```
# irq_start
# <n0>
# <n1>
# ...
# irq_end
```

Génère une interruption au bout de <nx> cycles d'horloge. Exemple :

```
# irq_start
# 50
# 100
# irq_end
```

Génère une interruption à la date 50 puis à la date 150 (50+100).

D.4.3 Base de test et regression

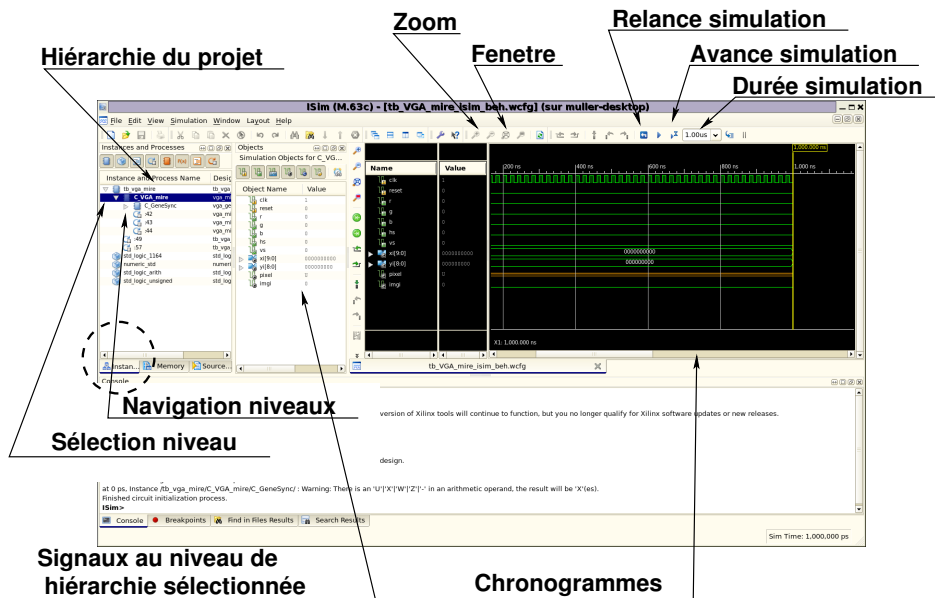
Pour définir votre base de test, il suffit d'écrire ligne par ligne dans le fichier `program/sequence_test` les noms de vos programmes en langage d'assemblage, qui respectent la syntaxe ci-dessus. Dès lors, il devient possible de vérifier le bon fonctionnement de toute votre base de test sans ouvrir le simulateur simplement en tapant :

```
make autotest_res
```



Le fichier de résultat `autotest_res` indique l'état de chaque test, ce qui permet de se concentrer sur les tests qui ne fonctionnent pas. Cette stratégie, appelée tests de régression, permet de s'assurer que les modifications apportées à un développement (ici, votre processeur) n'introduisent pas d'erreurs sur des parties déjà fonctionnelles. A utiliser donc sans modération avant de faire un push sur votre dépôt GIT et/ou une validation sur l'application Web.



D.5 Le simulateur VHDL

Le simulateur **fuse** se présente sous la forme suivante :



Afin de visualiser un signal, il faut naviguer dans la hiérarchie, le sélectionner dans la fenêtre des signaux puis le glisser/déposer dans la fenêtre des chronogrammes. Dans cette fenêtre, il est possible de sélectionner une zone, faire des zooms et contre-zooms.

Si vous voulez visualiser un signal qui n'est pas dans la fenêtre, le chercher dans la hiérarchie, l'ajouter au chronogramme, redémarrer la simulation avec  puis la faire avancer à nouveau avec .

Le bouton  permet de voir le chronogramme sur la durée complète de la simulation. Une zone peut être zoomée à l'aide de .

D.5.1 Mise en page des chronogrammes

Le format des signaux affichés est par défaut celui du type déclaré dans le VHDL. Cependant, la lecture du binaire n'étant pas aisée, il est possible de modifier l'affichage d'un signal en le sélectionnant dans la fenêtre de chronogramme puis click-bouton-droit, sélection *Radix* et sélection du format souhaité. Attention, la sélection de *Decimal* ou *Unsigned* n'a pas le même effet selon que votre signal binaire a été déclaré signé ou non. Il est aussi possible de modifier la couleur d'un signal (*Signal color*).

La mise en page des chronogrammes peut être modifiée en click-bouton-droit sur la fenêtre de chronogramme puis *New Divider* ou *New Group*.

Il est possible de sauvegarder la mise en page et le format des signaux *File->Save*, sous le nom `tb_<top>_isim_beh.wcfg` avec `top` le nom donné en argument de `make`.


La configuration de l'affichage sera automatiquement utilisée si vous la sauvegardez dans le répertoire `config/tb_<top>_isim_beh.wcfg`.

D.5.2 Mise à jour du VHDL

Attention, lorsque vous modifiez le VHDL, il est nécessaire de quitter le simulateur. La procédure à suivre est la suivante :

1. Modifier le VHDL, le programme assembleur ou l'image
2. Exécuter

```
make run_simu [TOP=<top>] [PROG=<prog>]
```

3. Avancer la simulation à l'aide de 

E Cross-compilation

Une fois le jeu d'instructions du processeur implémenté dans votre processeur (et uniquement à ce moment là !), il est possible de générer des programmes exécutables pour votre système directement à partir d'un code C à l'aide d'un cross-compileur. À la différence d'un compilateur qui génère du code machine pour la machine sur laquelle il est exécuté, un cross-compileur génère du code machine pour un autre processeur. Le compilateur pour MIPS utilisé pour cela se trouve dans le répertoire `/opt/mips-toolchain-bin/bin/` sur les machines de l'école. Son usage est détaillé dans le fichier README du répertoire `crossTools` de votre projet.

Le binaire généré par le cross-compileur doit être placé dans le répertoire `program` du projet MIPS, le Makefile du répertoire `crossTools` indique à `make` de le faire

Vous trouverez dans le répertoire `crossTools` :

1. Le fichier README détaillant comment générer un fichier exécutable pour votre environnement.
2. Le Makefile permettant d'automatiser, la génération de l'exécutable, la recopie dans le répertoire `program` du projet et la liste des instructions que le processeur doit implémenter pour que le programme puisse être exécuté
3. Un script `instructions_necessaires.sh` prenant en paramètre un fichier exécutable (au format elf), indiquant les instructions nécessaires à son exécution.
4. Les fichiers nécessaires à l'initialisation de la mémoire (`.s`) et la configuration de l'éditeur de lien (`.scr`) pour un programme sans interruption (`mmips_link_no_irq.scr` et `mmips_no_irq.s`), et un programme avec interruption (`mmips_link_irq.scr` et `mmips_irq.s`)

5. Un programme d'exemple n'utilisant pas les interruptions, réalisant le jeu space invader (mips_invader_zybo.c et sprite2.h)
6. Un programme d'exemple simple utilisant les interruptions (test_irq.c)

Par exemple, si vous voulez tester votre système avec le jeu Space Invader voilà comment procéder :

```
# cd crossTools
# make mips_invader_zybo.elf IRQ_FLAG=no_irq
mips-sde-elf-gcc -mips32 -Xassembler -EB -Xassembler -mabi=32 -Xassembler -W -G0 -fno-delayed-branch -c -o mips_invader_zybo.o mips_invader_zybo.c
mips-sde-elf-as -mips32 -EB -mabi=32 -non_shared -G0 -o mmips_no_irq.o mmips_no_irq.s
mips-sde-elf-ld -G0 -mips32 --static -T mmips_link_no_irq.scr -o mips_invader_zybo.elf mmips_no_irq.o mips_invader_zybo.o
cp mips_invader_zybo.elf ../program
echo "Les instructions nécessaires pour exécuter ce programme sont:"
Les instructions nécessaires pour exécuter ce programme sont:
./instructions_necessaires.sh mips_invader_zybo.elf
addiu addu andi beq beqz bgez bltz bne bnez j jal jr lui lw ori sll slt slti sltiu sltu srl subu sw xori
```

Vous voilà avec le fichier mips_invader_zybo.elf dans le répertoire program de votre projet.

Il ne reste plus qu'à tester sur carte avec l'environnement complet :

```
# cd ..
# make run_fpga TOP=MMIPS_complet PROG=mips_invader_zybo
```

Vous pouvez procéder de même pour le programme de test des interruptions, modifier à loisir ces programmes et en créer de nouveaux pour tester votre système.

F Documentation

- Le consortium en charge du maintien du standard et de la norme du VHDL : www.vhdl.org
- Le wikipédia sur le sujet : fr.wikipedia.org/wiki/VHDL
Attention : wikipedia n'est pas exempte d'erreurs !
- Un site web très complet sur le VHDL : [Hamburg VHDL archive](#)
- La fameuse bible de la syntaxe du VHDL, le "VHDL-Cookbook" : [VHDL-Cookbook](#)

G Encodage des instructions

Les instructions sont toutes codées sur 32 bits. Les tables suivantes présentent sous forme compacte le codage des différentes instructions.

31	26	25	21	20	16	15	11	10	6	5	0
----	----	----	----	----	----	----	----	----	---	---	---

Les 3 formats d'instruction

Format R :	opcode	RS	RT	RD	SH	FUNC
Format I :	opcode	RS	RT	IMM16		
Format J :	opcode	IMM26				

Champ **opcode**

31...29 \ 28...26	000	001	010	011	100	101	110	111
000	special	regimm	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	cop0	-	-	-	-	-	-	-
011	-	-	-	-	-	-	-	-
100	LB	LH	-	LW	LBU	LHU	-	-
101	SB	SH	-	SW	-	-	-	-
11X	-	-	-	-	-	-	-	-

Champ **FUNC**, lorsque l'**opcode** vaut **special**.

5...3 \ 2...0	000	001	010	011	100	101	110	111
000	SLL	-	SRL	SRA	SLLV	-	SRLV	SRAV
001	JR	JALR	-	-	SYSCALL	BREAK	-	-
010	MFHI	MTHI	MFLO	MTLO	-	-	-	-
011	MULT	MULTU	DIV	DIVU	-	-	-	-
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101	-	-	SLT	SLTU	-	-	-	-
11X	-	-	-	-	-	-	-	-

Champ **RT**, lorsque l'**opcode** vaut **regimm**.

20...19 \ 18...16	000	001	010	011	100	101	110	111
00	BLTZ	BGEZ	-	-	-	-	-	-
10	BLTZAL	BGEZAL	-	-	-	-	-	-
X1	-	-	-	-	-	-	-	-

Champ **RS**, lorsque l'**opcode** vaut **cop0**.

25...24 \ 23...21	000	001	010	011	100	101	110	111
00	MFC0	-	-	-	MTC0	-	-	-
01	-	-	-	-	-	-	-	-
1X	-	-	-	-	-	-	-	-

Champ **FUNC**, lorsque l'**opcode** vaut **cop0**.

5...3 \ 2...0	000	001	010	011	100	101	110	111
00X	-	-	-	-	-	-	-	-
010	-	-	-	-	-	-	-	-
011	ERET	-	-	-	-	-	-	-
1XX	-	-	-	-	-	-	-	-

H Description des instructions

Cette section décrit les instructions du MIPS R3000.

add/addu

action

Addition registre registre signée ²

syntaxe

add \$rd, \$rs, \$rt

description

Les contenus des registres \$rs et \$rt sont ajoutés pour former un résultat sur 32 bits qui est placé dans le registre \$rd.

opération

$rd \leftarrow rs + rt$

format R

addi/addiu

action

Addition registre immédiat signée ²

syntaxe

addi \$rt, \$rs, imm

description

La valeur immédiate sur 16 bits subit une extension de signe, et est ajoutée au contenu du registre \$rs pour former un résultat sur 32 bits qui est placé dans le registre \$rt.

opération

$rt \leftarrow (\text{IMM16}_{15}^{16} \parallel \text{IMM16}_{15\dots0}) + rs$

format I

and

action

Et bit-à-bit registre registre

syntaxe

and \$rd, \$rs, \$rt

description

Un et bit-à-bit est effectué entre les contenus des registres \$rs et \$rt. Le résultat est placé dans le registre \$rd.

opération

$rd \leftarrow rs \text{ and } rt$

format R

andi

action

Et bit-à-bit registre immédiat

syntaxe

andi \$rt, \$rs, imm

description

La valeur immédiate sur 16 bits subit une extension de zéros. Un et bit-à-bit est effectué entre cette valeur étendue et le contenu du registre \$rs pour former un résultat placé dans le registre \$rt.

opération

$rt \leftarrow 0^{16} \parallel \text{IMM16 and } rs$

format I

²Les instructions add, sub et addi devraient générer une exception lors d'un dépassement de capacité. Les levées d'exception n'étant pas implantées, ces instructions sont identiques à leur version non-signée (addu, subu et addiu).

beq

action

Branchement si registre égal registre

syntaxe

beq \$rs, \$rt, label

description

Les contenus des registres \$rs et \$rt sont comparés. S'ils sont égaux, le programme saute à l'adresse correspondant à l'étiquette. Le champ IMM contient l'écart relatif en instruction correspondant à ce saut. Cet écart est calculé par l'assembleur.

opération

$rs = rt \Rightarrow pc \leftarrow pc + 4 + (IMM16_{15}^{14} \parallel IMM16_{15\dots0} \parallel 0^2)$

format I

bgez

action

Branchement si registre supérieur ou égal à zéro

syntaxe

bgez \$rs, label

description

Si le contenu du registre \$rs est supérieur ou égal à zéro, le programme saute à l'adresse correspondant à l'étiquette. Le champ IMM contient l'écart relatif en instruction correspondant à ce saut. Cet écart est calculé par l'assembleur.

opération

$rs \geq 0 \Rightarrow pc \leftarrow pc + 4 + (IMM16_{15}^{14} \parallel IMM16_{15\dots0} \parallel 0^2)$

format I

bgezal

action

Branchement à une fonction si registre supérieur ou égal à zéro

syntaxe

bgezal \$rs, label

description

Inconditionnellement, l'adresse de l'instruction suivant le bgezal est sauvée dans le registre \$31. Si le contenu du registre \$rs est supérieur ou égal à zéro, le programme saute à l'adresse correspondant à l'étiquette. Le champ IMM contient l'écart relatif en instruction correspondant à ce saut. Cet écart est calculé par l'assembleur.

opération

$\$31 \leftarrow pc + 4$

$rs \geq 0 \Rightarrow pc \leftarrow pc + 4 + (IMM16_{15}^{14} \parallel IMM16_{15\dots0} \parallel 0^2)$

format I

bgtz

action

Branchement si registre strictement supérieur à zéro

syntaxe

bgtz \$rs, label

description

Si le contenu du registre \$rs est strictement supérieur à zéro, le programme saute à l'adresse correspondant à l'étiquette. Le champ IMM contient l'écart relatif en instruction correspondant à ce saut. Cet écart est calculé par l'assembleur.

opération

$rs > 0 \Rightarrow pc \leftarrow pc + 4 + (IMM16_{15}^{14} \parallel IMM16_{15\dots0} \parallel 0^2)$

format I

blez

action

Branchement si registre inférieur ou égal à zéro

syntaxe

blez \$rs, label

description

Si le contenu du registre \$rs est inférieur ou égal à zéro, le programme saute à l'adresse correspondant à l'étiquette. Le champ IMM contient l'écart relatif en instruction correspondant à ce saut. Cet écart est calculé par l'assembleur.

opération

$rs \leq 0 \Rightarrow pc \leftarrow pc + 4 + (IMM16_{15}^{14} \parallel IMM16_{15...0} \parallel 0^2)$

format I

bltz

action

Branchement si registre strictement inférieur à zéro

syntaxe

bltz \$rs, label

description

Si le contenu du registre \$rs est strictement inférieur à zéro, le programme saute à l'adresse correspondant à l'étiquette. Le champ IMM contient l'écart relatif en instruction correspondant à ce saut. Cet écart est calculé par l'assembleur.

opération

$rs < 0 \Rightarrow pc \leftarrow pc + 4 + (IMM16_{15}^{14} \parallel IMM16_{15...0} \parallel 0^2)$

format I

bltzal

action

Branchement à une fonction si registre inférieur à zéro

syntaxe

bltzal \$rs, label

description

Inconditionnellement, l'adresse de l'instruction suivant le bltzal est sauvée dans le registre \$31. Si le contenu du registre \$rs est strictement inférieur à zéro, le programme saute à l'adresse correspondant à l'étiquette. Le champ IMM contient l'écart relatif en instruction correspondant à ce saut. Cet écart est calculé par l'assembleur.

opération

$\$31 \leftarrow pc + 4$

$rs < 0 \Rightarrow pc \leftarrow pc + 4 + (IMM16_{15}^{14} \parallel IMM16_{15...0} \parallel 0^2)$

format I

bne

action

Branchement si registre différent de registre

syntaxe

bne \$rs, \$rt, label

description

Les contenus des registres \$rs et \$rt sont comparés. S'ils sont différents, le programme saute à l'adresse correspondant à l'étiquette. Le champ IMM contient l'écart relatif en instruction correspondant à ce saut. Cet écart est calculé par l'assembleur.

opération

$rs \neq rt \Rightarrow pc \leftarrow pc + 4 + (IMM16_{15}^{14} \parallel IMM16_{15...0} \parallel 0^2)$

format I

break

action

Arrêt et saut à la routine d'exception

syntaxe

break

description

Un point d'arrêt est détecté, et le programme saute à l'adresse de la routine de gestion des exceptions.

opération

$pc \leftarrow 0x00000080$

exception

Déclenchement d'une exception de type point d'arrêt.

format R

div

action

Division entière et reste signé registre registre

syntaxe

div \$rs, \$rt

description

Le contenu du registre \$rs est divisé par le contenu du registre \$rt, le contenu des deux registres étant considéré comme des nombres en complément à deux. Le résultat de la division est placé dans le registre spécial \$lo, et le reste dans \$hi.

opération

$lo \leftarrow \frac{rs}{rt}$
 $hi \leftarrow rs \bmod rt$

format R

divu

action

Division entière et reste non-signé registre registre

syntaxe

divu \$rs, \$rt

description

Le contenu du registre \$rs est divisé par le contenu du registre \$rt, le contenu des deux registres étant considéré comme des nombres non signés. Le résultat de la division est placé dans le registre spécial \$lo, et le reste dans \$hi.

opération

$lo \leftarrow \frac{0 \parallel rs}{0 \parallel rt}$
 $hi \leftarrow (0 \parallel rs) \bmod (0 \parallel rt)$

format R

eret

action

Retour d'exception (ou d'interruption)

syntaxe

eret

description pour les extensions

Le programme saute à l'adresse stockée dans EPC. Afin de démasquer les interruptions, le registre SR est mis à 0.

opération

$pc \leftarrow epc$
 $sr \leftarrow sr_{31..1} \parallel 0$

format R

j

action

Branchement inconditionnel immédiat

syntaxe

j label

description

Le programme saute inconditionnellement à l'adresse correspondant à l'étiquette. Le champ IMM26, calculé par l'assembleur, contient la position de l'étiquette (l'unité de comptage étant l'instruction) dans la même zone mémoire de 256Mo que l'instruction suivant le saut.

opération

$pc \leftarrow (pc + 4)_{31...28} \parallel IMM26_{25...0} \parallel 0^2$

format J

jal

action

Appel de fonction inconditionnel immédiat

syntaxe

jal label

description

L'adresse de l'instruction suivant le jal est sauvée dans le registre \$31. Le programme saute inconditionnellement à l'adresse correspondant à l'étiquette. Le champ IMM26, calculé par l'assembleur, contient la position de l'étiquette (l'unité de comptage étant l'instruction) dans la même zone mémoire de 256Mo que l'instruction suivant le saut.

opération

$\$31 \leftarrow pc + 4$

$pc \leftarrow (pc+4)_{31...28} \parallel IMM26_{25...0} \parallel 0^2$

format J

jalr

action

Appel de fonction inconditionnel registre

syntaxe

jalr \$rd, \$rs

description

Le programme saute à l'adresse contenue dans le registre \$rs. L'adresse de l'instruction suivant le jalr est sauvée dans le registre \$rd. Attention, l'adresse contenue dans le registre \$rs doit être aligné sur une frontière de mots.

opération

$rd \leftarrow pc + 4$

$pc \leftarrow rs$

format R

jr

action

Branchement inconditionnel registre

syntaxe

jr \$rs

description

Le programme saute à l'adresse contenue dans le registre \$rs. Attention, cette adresse doit être aligné sur une frontière de mots.

opération

$pc \leftarrow rs$

format R

lb

action

Lecture d'un octet signé de la mémoire

syntaxe

lb \$rt, imm(\$rs)

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre \$rs. Le contenu de cette adresse subit une extension de signe et est ensuite placé dans le registre \$rt.

opération

$rt \leftarrow mem[(IMM16_{15}^{16} \parallel IMM16_{15...0}) + rs]_7^{24} \parallel mem[(IMM16_{15}^{16} \parallel IMM16_{15...0}) + rs]_{7...0}$

exception

- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

format I

lbu

action

Lecture d'un octet non-signé de la mémoire

syntaxe

lbu \$rt, imm(\$rs)

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre \$rs. Le contenu de cette adresse est étendu avec des zéro et est ensuite placé dans le registre \$rt.

opération

$rt \leftarrow 0^{24} \parallel mem[(IMM16_{15}^{16} \parallel IMM16_{15...0}) + rs]_{7...0}$

exception

- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

format I

lh

action

Lecture d'un demi-mot signé de la mémoire

syntaxe

lh \$rt, imm(\$rs)

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre \$rs. Le contenu de cette adresse subit une extension de signe et est ensuite placé dans le registre \$rt. Attention, le bit de poids faible de l'adresse résultante doit être à zéro.

opération

$rt \leftarrow mem[(IMM16_{15}^{16} \parallel IMM16_{15...0}) + rs]_{15}^{16} \parallel mem[(IMM16_{15}^{16} \parallel IMM16_{15...0}) + rs]_{15...0}$

exception

- Adresse non alignée sur une frontière de demi-mot. ;
- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

format I

lhu

action

Lecture d'un demi-mot non-signé de la mémoire

syntaxe

lhu \$rt, imm(\$rs)

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre \$rs. Le contenu de cette adresse est étendu avec des zéro et est ensuite placé dans le registre \$rt. Attention, le bit de poids faible de l'adresse résultante doit être à zéro.

opération

$rt \leftarrow 0^{16} \parallel \text{mem}[(\text{IMM16}_{15}^{16} \parallel \text{IMM16}_{15\dots0}) + rs]_{15\dots0}$

exception

- Adresse non alignée sur une frontière de demi-mot ;
- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

format I

lui

action

Lecture d'une constante dans les poids forts

syntaxe

lui \$rt, imm

description

La constante immédiate de 16 bits est décalée de 16 bits à gauche, et est complétée de zéro. La valeur ainsi obtenue est placée dans \$rt.

opération

$rt \leftarrow \text{IMM16} \parallel 0^{16}$

format I

lw

action

Lecture d'un mot de la mémoire

syntaxe

lw \$rt, imm(\$rs)

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre \$rs. Le contenu de cette adresse est placé dans le registre \$rt. Attention, les deux bits de poids faible de l'adresse résultante doivent être à zéro.

opération

$rt \leftarrow \text{mem}[(\text{IMM16}_{15}^{16} \parallel \text{IMM16}_{15\dots0}) + rs]$

exception

- Adresse non alignée sur une frontière de mot ;
- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

format I

mfc0

action

Copie d'un registre spécialisé dans d'un registre général

syntaxe

`mfc0 $rt, $rd`

description

Le contenu du registre spécialisé `$rd` — non directement accessible au programmeur — est recopié dans le registre général `$rt`. Les registres possibles pour `$rd` servent à la gestion des exceptions et interruptions, et sont les suivants : \$8 pour **BAR** (*bad address register*), \$12 pour **SR** (*status register*), \$13 pour **CR** (*cause register*) et \$14 pour **EPC** (*exception program counter*).

opération

$rt \leftarrow copro[rd]$

exception

- Utilisation de l'instruction en mode utilisateur.

format R

mfhi

action

Copie le registre `$hi` dans un registre général

syntaxe

`mfhi $rd`

description

Le contenu du registre spécialisé `$hi` — qui est mis à jour par l'opération de multiplication ou de division — est recopié dans le registre général `$rd`.

opération

$rd \leftarrow hi$

format R

mflo

action

Copie le registre `$lo` dans un registre général

syntaxe

`mflo $rd`

description

Le contenu du registre spécialisé `$lo` — qui est mis à jour par l'opération de multiplication ou de division — est recopié dans le registre général `$rd`.

opération

$rd \leftarrow lo$

format R

mtc0

action

Copie d'un registre général dans un registre spécialisé

syntaxe

`mtc0 $rt, $rd`

description

Le contenu du registre général `$rt` est recopié dans le registre spécialisé `$rd` — non directement accessible au programmeur —. Ces registres servent à la gestion des exceptions et interruptions, et sont les suivants : \$8 pour **BAR** (*bad address register*), \$12 pour **SR** (*status register*), \$13 pour **CR** (*cause register*) et \$14 pour **EPC** (*exception program counter*).

opération

$copro[rd] \leftarrow rt$

format R

mthi

action
Copie d'un registre général dans le registre \$hi

syntaxe
mthi \$rs

description
Le contenu du registre général \$rs est recopié dans le registre spécialisé \$hi.

opération
 $hi \leftarrow rs$

format R

mtlo

action
Copie d'un registre général dans le registre \$lo

syntaxe
mtlo \$rs

description
Le contenu du registre général \$rs est recopié dans le registre spécialisé \$lo.

opération
 $lo \leftarrow rs$

format R

mult

action
Multiplication signé registre registre

syntaxe
mult \$rs, \$rt

description
Le contenu du registre \$rs est multiplié par le contenu du registre \$rt, le contenu des deux registres étant considéré comme des nombres en complément à deux. Les 32 bits de poids fort du résultat sont placés dans le registre \$hi, et les 32 bits de poids faible dans \$lo.

opération
 $lo \leftarrow (rs \times rt)_{31..0}$
 $hi \leftarrow (rs \times rt)_{63..32}$

format R

multu

action
Multiplication signé registre registre

syntaxe
multu \$rs, \$rt

description
Le contenu du registre \$rs est multiplié par le contenu du registre \$rt, le contenu des deux registres étant considéré comme des nombres non-signés. Les 32 bits de poids fort du résultat sont placés dans le registre \$hi, et les 32 bits de poids faible dans \$lo.

opération
 $lo \leftarrow (0 \parallel rs \times 0 \parallel rt)_{31..0}$
 $hi \leftarrow (0 \parallel rs \times 0 \parallel rt)_{63..32}$

format R

nor

action
Non-ou bit-à-bit registre registre

syntaxe

`nor $rd, $rs, $rt`

description

Un non-ou bit-à-bit est effectué entre les contenus des registres \$rs et \$rt. Le résultat est placé dans le registre \$rd.

opération

$rd \leftarrow rs \text{ nor } rt$

format R

or

action

Ou bit-à-bit registre registre

syntaxe

`or $rd, $rs, $rt`

description

Un ou bit-à-bit est effectué entre les contenus des registres \$rs et \$rt. Le résultat est placé dans le registre \$rd.

opération

$rd \leftarrow rs \text{ or } rt$

format R

ori

action

Ou bit-à-bit registre immédiat

syntaxe

`ori $rt, $rs, imm`

description

La valeur immédiate sur 16 bits subit une extension de zéros. Un ou bit-à-bit est effectué entre cette valeur étendue et le contenu du registre \$rs pour former un résultat placé dans le registre \$rt.

opération

$rt \leftarrow (0^{16} \parallel IMM16) \text{ or } rs$

format I

sb

action

Écriture d'un octet en mémoire

syntaxe

`sb $rt, imm($rs)`

description

L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre \$rs. L'octet de poids faible du registre \$rt est écrit à l'adresse ainsi calculée.

opération

$\text{mem}[(IMM16_{15}^{16} \parallel IMM16_{15...0}) + rs] \leftarrow rt_{7...0}$

exception

- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

format I

sh

action

Écriture d'un demi-mot en mémoire

syntaxe

sh \$rt, imm(\$rs)

description

L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre \$rs. Les deux octets de poids faible du registre \$rt sont écrit à l'adresse ainsi calculée. Le bit de poids faible de cette adresse doit être à zéro.

opération

$\text{mem}[(\text{IMM16}_{15}^{16} \parallel \text{IMM16}_{15\dots0}) + \text{rs}] \leftarrow \text{rt}_{15\dots0}$

exception

- Adresse non alignée sur une frontière de demi-mot ;
- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

format I

sll

action

Décalage à gauche immédiat

syntaxe

sll \$rd, \$rt, sh

description

Le registre \$rt est décalé à gauche de la valeur immédiate codée dans les 5 bits du champ SH, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre \$rd.

opération

$\text{rd} \leftarrow \text{rt}_{31-sh\dots0} \parallel 0^{sh}$

format R

sllv

action

Décalage à gauche registre

syntaxe

sllv \$rd, \$rt, \$rs

description

Le registre \$rt est décalé à gauche du nombre de bits spécifiés dans les 5 bits de poids faible du registre \$rs, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre \$rd.

opération

$\text{rd} \leftarrow \text{rt}_{31-rs_{4\dots0}\dots0} \parallel 0^{rs_{4\dots0}}$

format R

slt

action

Comparaison signée registre registre

syntaxe

slt \$rd, \$rs, \$rt

description

Le contenu du registre \$rs est comparé au contenu du registre \$rt, les deux valeurs étant considérées comme des quantités signées. Si la valeur contenue dans \$rs est inférieure à celle contenue dans \$rt, alors \$rd prend la valeur un, sinon il prend la valeur zéro.

opération

$rs < rt \Rightarrow \text{rd} \leftarrow 0^{31} \parallel 1$

$rs \geq rt \Rightarrow \text{rd} \leftarrow 0^{32}$

format R

slti

action

Comparaison signée registre immédiat

syntaxe

slti \$rt, \$rs, imm

description

Le contenu du registre \$rs est comparé à la valeur immédiate sur 16 bits qui a subit une extension de signe. Les deux valeurs étant considérées comme des quantités signées, si la valeur contenue dans \$rs est inférieure à celle de l'immédiat étendu, alors \$rt prend la valeur un, sinon il prend la valeur zéro.

opération

$$\begin{aligned} rs < (IMM16_{15}^{16} \parallel IMM16_{15...0}) &\Rightarrow rt \leftarrow 0^{31} \parallel 1 \\ rs \geq (IMM16_{15}^{16} \parallel IMM16_{15...0}) &\Rightarrow rt \leftarrow 0^{32} \end{aligned}$$

format I

sltiu

action

Comparaison non-signée registre immédiat

syntaxe

sltiu \$rt, \$rs, imm

description

Le contenu du registre \$rs est comparé à la valeur immédiate sur 16 bits qui a subit une extension de signe. Les deux valeurs étant considérées comme des quantités non-signées, si la valeur contenue dans \$rs est inférieure à celle de l'immédiat étendu, alors \$rt prend la valeur un, sinon il prend la valeur zéro.

opération

$$\begin{aligned} (0 \parallel rs) < (0 \parallel IMM16_{15}^{16} \parallel IMM16_{15...0}) &\Rightarrow rt \leftarrow 0^{31} \parallel 1 \\ (0 \parallel rs) \geq (0 \parallel IMM16_{15}^{16} \parallel IMM16_{15...0}) &\Rightarrow rt \leftarrow 0^{32} \end{aligned}$$

format I

sltu

action

Comparaison non-signée registre registre

syntaxe

sltu \$rd, \$rs, \$rt

description

Le contenu du registre \$rs est comparé au contenu du registre \$rt, les deux valeurs étant considérés comme des quantités non-signées. Si la valeur contenue dans \$rs est inférieure à celle contenue dans \$rt, alors \$rd prend la valeur un, sinon il prend la valeur zéro.

opération

$$\begin{aligned} (0 \parallel rs) < (0 \parallel rt) &\Rightarrow rd \leftarrow 0^{31} \parallel 1 \\ (0 \parallel rs) \geq (0 \parallel rt) &\Rightarrow rd \leftarrow 0^{32} \end{aligned}$$

format R

sra

action

Décalage à droite arithmétique immédiat

syntaxe

sra \$rd, \$rt, sh

description

Le registre \$rt est décalé à droite de la valeur immédiate codée dans les 5 bits du champ SH, le bit de signe du registre étant introduit dans les bits de poids fort. Le résultat est placé dans le registre \$rd.

opération

$$rd \leftarrow rt_{31}^{sh} \parallel rt_{31...sh}$$

format R

srav

action

Décalage à droite arithmétique registre

syntaxe

srav \$rd, \$rt, \$rs

description

Le registre \$rt est décalé à droite du nombre de bits spécifiés dans les 5 bits de poids faible du registre \$rs, le signe de \$rt étant introduit dans les bits de poids fort ainsi libérés. Le résultat est placé dans le registre \$rd.

opération

$$rd \leftarrow rt_{31}^{rs_{4...0}} \parallel rt_{31...rs_{4...0}...0}$$

format R

srl

action

Décalage à droite logique immédiat

syntaxe

srl \$rd, \$rt, sh

description

Le registre \$rt est décalé à droite de la valeur immédiate codée dans les 5 bits du champ SH, des zéros étant introduits dans les bits de poids fort. Le résultat est placé dans le registre \$rd.

opération

$$rd \leftarrow 0^{sh} \parallel rt_{31...sh}$$

format R

srlv

action

Décalage à droite logique registre

syntaxe

srlv \$rd, \$rt, \$rs

description

Le registre \$rt est décalé à droite du nombre de bits spécifiés dans les 5 bits de poids faible du registre \$rs, des zéros étant introduits dans les bits de poids fort ainsi libérés. Le résultat est placé dans le registre \$rd.

opération

$$rd \leftarrow 0^{rs_{4...0}} \parallel rt_{31...rs_{4...0}}$$

format R

sub/subu

action

Soustraction registre registre signée ²

syntaxe

sub \$rd, \$rs, \$rt

description

Le contenu du registre \$rs est soustrait du contenu du registre \$rt pour former un résultat sur 32 bits qui est placé dans le registre \$rd.

opération

$$rd \leftarrow rs - rt$$

format R

sw

action

Écriture d'un mot en mémoire

syntaxe

sw \$rt, imm(\$rs)

description

L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre \$rs. Le contenu du registre \$rt est écrit à l'adresse ainsi calculée. Les deux bits de poids faible de cette adresse doivent être à zéro.

opération

$\text{mem}[(\text{IMM16}_{15}^{16} \parallel \text{IMM16}_{15\dots0}) + \text{rs}] \leftarrow \text{rt}$

exception

Adresse non alignée sur une frontière de mot.

format

I

syscall

action

Appel à une fonction du système (en mode noyau).

syntaxe

syscall

description

Un appel système est effectué, transférant immédiatement, et inconditionnellement le contrôle au gestionnaire d'exception. Note : par convention, le numéro de l'appel système, c.-à-d. le code de la fonction système à effectuer, est placé dans le registre \$2.

opération

$\text{pc} \leftarrow 0x00000080$

exception

Déclenchement d'une exception de type appel système.

format

R

xor

action

Ou-exclusif bit-à-bit registre registre

syntaxe

xor \$rd, \$rs, \$rt

description

Un ou-exclusif bit-à-bit est effectué entre les contenus des registres \$rs et \$rt. Le résultat est placé dans le registre \$rd.

opération

$\text{rd} \leftarrow \text{rs} \text{ xor } \text{rt}$

format

R

xori

action

Ou-exclusif bit-à-bit registre immédiat

syntaxe

xori \$rt, \$rs, imm

description

La valeur immédiate sur 16 bits subit une extension de zéros. Un ou-exclusif bit-à-bit est effectué entre cette valeur étendue et le contenu du registre \$rs pour former un résultat placé dans le registre \$rt.

opération

$\text{rt} \leftarrow (0^{16} \parallel \text{IMM16}) \text{ xor } \text{rs}$

format

I