

Cahier des charges pour la conception d'un processeur RISC-V 2023 Polytech Grenoble – E2I5

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	4
2.2.2	Format du rendu	5
3	Architecture du processeur et son environnement	5
3.1	Partie Contrôle	5
3.2	Partie Opérative	5
3.2.1	La gestion des sauts	7
3.2.2	La gestion des interruptions	7
3.3	Système complet	8
4	Méthode de conception	8
4.1	Identification de l'instruction	8
4.2	Projection sur la PO	9
4.3	Ajout d'états dans la PC	9
4.4	Mise en œuvre de l'instruction	9
4.5	Ecriture d'un programme de test en langage d'assemblage	10
4.6	Validation : simulation et carte	10
4.7	Programmes de test	11
5	Spécification des périphériques	12
5.1	Les périphériques	12
5.2	Organisation de la mémoire	12
5.3	Le bus	13

A	Gestion du dépôt et validations automatiques	15
A.1	Dépôt et sources initiales	15
A.2	Validations et travail à distance	15
B	Notations	16
C	Organisation du projet	17
C.1	Répertoires et fichiers	17
C.2	Interface à la PO	17
C.3	Interface aux registres de contrôle/statut (CSR)	21
C.4	Signaux d'états de la PO	22
D	Environnement de conception	23
D.1	Utilisation du Makefile	23
D.1.1	La commande make	23
D.1.2	Simulation avec make	23
D.2	Programmation du FPGA avec make	23
D.3	Fonctionnement de l'autotest	24
D.3.1	Principe	24
D.3.2	Syntaxe des commentaires à ajouter	24
D.3.3	Base de test et regression	25
D.4	Le simulateur VHDL	25
E	Documentation	26
F	Les instructions RISC-V (RV32IM)	27
F.1	Format des instructions	27
F.2	Construction des constantes immédiates	27
F.3	Encodage des instructions	27
F.4	Description des instructions	29

1 Introduction

Le but de ce projet est de construire un processeur RISC-V en VHDL, avec pour objectif de pouvoir l'intégrer dans un système complet pour exécuter une application qui effectue un affichage sur écran. Le processeur est conçu en deux parties, Partie Contrôle (PC) et Partie Opérative (PO), et sera construit progressivement au cours du projet, en lui ajoutant des instructions. Les concepts fondamentaux des familles d'instructions 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

L'objectif du projet est de concevoir un processeur RISC-V capable d'exécuter une partie du jeu d'instructions. Le départ du projet est un canevas d'architecture composé d'une partie opérative (CPU_PO) et d'une partie contrôle (CPU_PC). Vous aurez à compléter la partie contrôle en ajoutant les états nécessaires à l'exécution des instructions du processeur et la partie opérative par deux composants (CPU_CND et CPU_CSR) permettant de gérer la condition de saut du processeur et les interruptions (voir sections [3.2.1](#) et [3.2.2](#)).

Les notations utilisées dans la suite sont données en annexe [B](#); la description des noms de fichiers, composants ainsi que des signaux sont en annexe [C](#); et l'environnement de conception est décrit en section [D](#). Assurez-vous de lire complètement ces annexes avant de continuer.

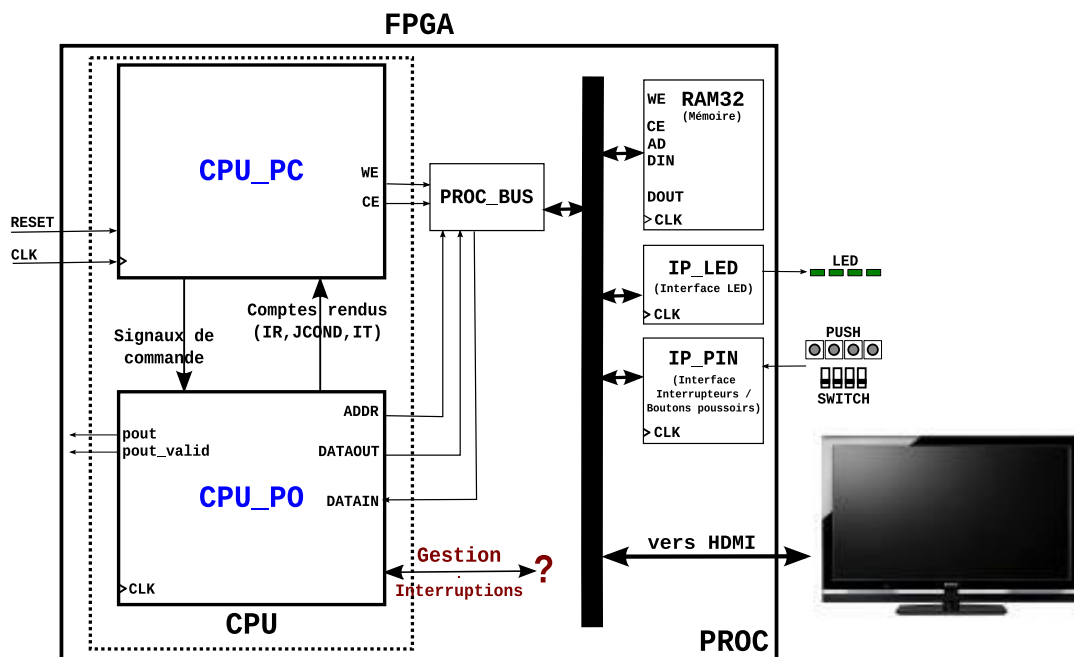


FIGURE 1 – Système complet avec les périphériques sur le bus

Les aspects techniques du cahier des charges sont détaillés en section [3](#). Vous pourrez utiliser votre processeur dans un environnement enrichie de périphériques : affichage sur LED, interfaces à des boutons poussoirs, interrupteurs et écran HDMI. Cet environnement, illustré sur la figure [1](#) vous permettra de faire fonctionner des applications graphiques.

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. L'exécution de certaines instructions permettra d'allumer et d'éteindre les LEDs (dans un premier temps en changeant la valeur du registre x31)

Par défaut, l'IP_LED renvoie la valeur du registre x31 (directement accessible via la sortie pout du processeur qui est branchée sur IP_LED), et les interrupteurs (SW0 à SW2) permettent de sélectionner parmi les 32 bits, les 4 bits de x31 visualisés sur les leds de la cartes. (ex : interrupteur (SW2 SW1 SW0) à 001 permettent de visualiser x31[7:4]). Pour plus de détails voir le fichier IP_LED.vhd.

Chaque instruction est à valider par simulation (étape 1) puis sur carte (étape 2) via l'écriture d'un programme de test en langage d'assemblage spécifique à cette instruction. Les deux étapes de validation seront prises en compte pour la notation. Lorsqu'il y aura assez d'instructions, vous pourrez valider le processeur par de petites applications graphiques écrites en langage C.

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 au moins 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 à écrire
1 (4h)	Lecture du CdC, tutoriel GIT en autonomie, Analyse du code + décodage initial lui , addi		allumer_leds.s
2-3 (8h)	add sll auipc	and , or , ori , andi , xor , xori , sub srl , sra , srai , slli , srli	compteur.s chenillard_minimaliste.s
4-5 (8h)	PO : composant CPU_CND (JCOND/SLT) beq slt	bne , blt , bge , bltu , bgeu slti , sltu , sltiu	chenillard_rotation.s
6 (4h)	lw , sw jal	jalr	Programmes donnés : blue , pong , snake , brick , invader
Fin du projet	Jouer avec les applications, extensions	Modifier les apps	Soyez créatif!!

2.2 Evaluation

La note 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.3](#). Il vous aidera à définir votre base de test au cours du projet et nous permettra de l'évaluer.

2.2.2 Format du rendu

Vous devez rendre votre réalisation dans votre dépôt Git (voir section A). Vous devez bien veiller à ce que la bonne version soit disponible dans la branche `master`. 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 elles sont réalisées (dans quels fichiers) 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 (à l'aide du Makefile donnée) ;
- 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. `chenillard_minimaliste.s`). Un exemple est donné pour le test de l'instruction `lui`. Explorez le répertoire `program/autotest` pour trouver ce test.

Attention : l'équipe d'enseignants 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 PC/PO. 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 Contrôle

Le graphe de contrôle de la PC initiale peut être représenté par la figure 2 :

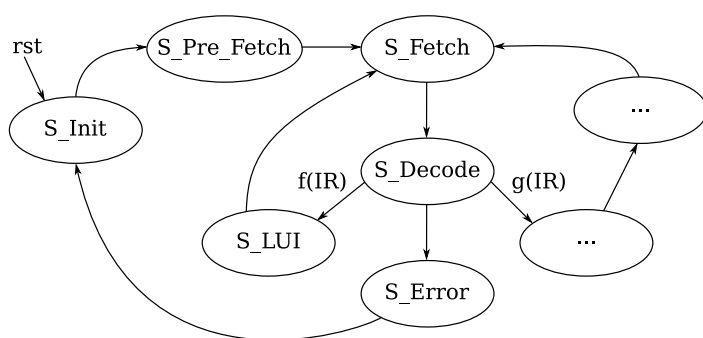


FIGURE 2 – Partie contrôle

États	Opérations entre registres
S_Init	$PC \leftarrow 0x1000$
S_Pre_Fetch	$mem_addr \leftarrow mem[PC]^a$
S_Fetch	$IR \leftarrow mem_datain$
S_Decode	$PC \leftarrow PC + 4^b$
S_LUI	$RD \leftarrow IR_{31..12} 0^{12}$; $mem_addr \leftarrow mem[PC]$
...	...

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

^bAttention, les instructions de branchement et `auipc` n'incrémentent pas PC dans cet état.

Par défaut le décodage d'une instruction non implantée conduit à un état d'erreur qui relance l'exécution de l'instruction à l'adresse `0x1000`. Cela permet d'obtenir un comportement de type boucle infinie, sans avoir à implanter d'instruction de saut.

3.2 Partie Opérative

La figure 3 décrit l'architecture de la partie opérative fournie. Les blocs en jaune sont les deux composants (`CPU_CND` et `CPU_CSR`) à ajouter pour la gestion des sauts et des interruptions.

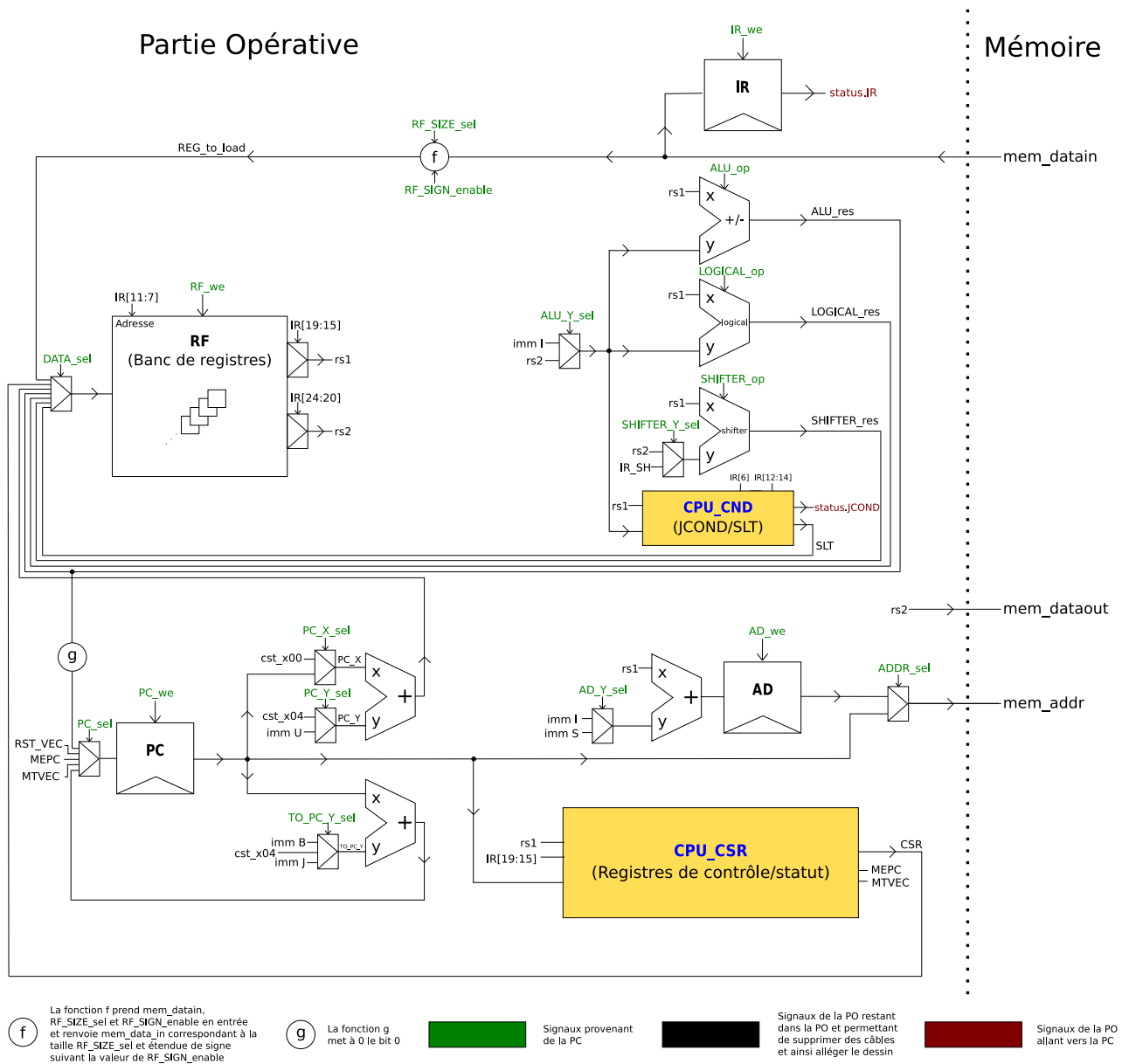


FIGURE 3 – Partie opérative et son interface avec la mémoire

3.2.1 La gestion des sauts

La figure 4 décrit le bloc responsable de la gestion des conditions de saut du processeur, dans le cas des instructions de branchement conditionnel (beq, bne, blt, bge, bltu, bgeu) différenciées entre elles par les bits IR[14:12] et des instructions de comparaison signée ou non-signée (slt/slti, sltu/sltiu) différenciées des précédentes par le bit IR[6].

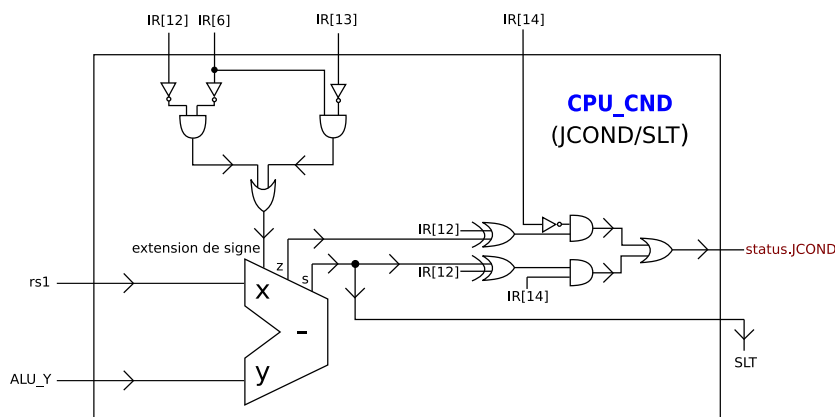


FIGURE 4 – Bloc pour la gestion des sauts dans le processeur

3.2.2 La gestion des interruptions

La figure 5 décrit le bloc responsable de la gestion de lecture/écriture des registres de contrôle/statut du processeur (CSR). Ces registres sont nécessaires pour l'implantation du support des interruptions par votre processeur. Ce bloc ne doit être mis en œuvre qu'une fois toutes les autres instructions du processeur (celles qui n'en font pas usage) sont codées. Malheureusement cette année ce module ne sera pas codé dans le cadre du projet.

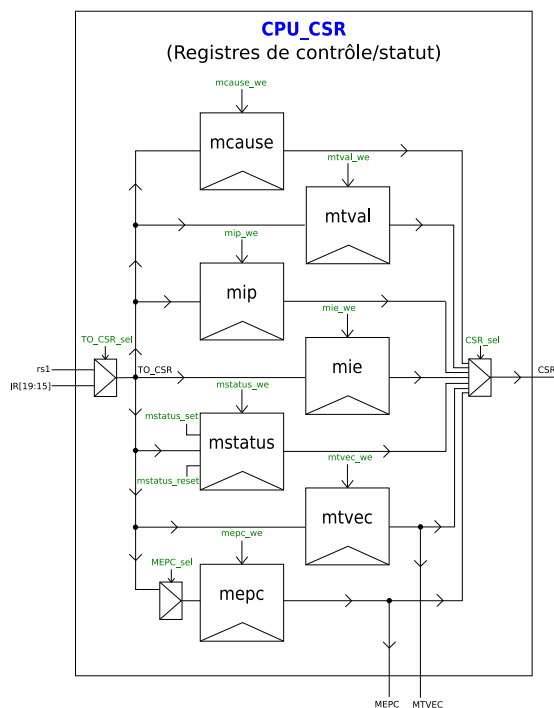


FIGURE 5 – Registres de contrôle/statut (CSR) pour la gestion d'interruptions

Il est connecté à la PO (signaux en noir, sauf `it` qui va vers la PC), à la PC (signaux de commande en vert et signal de test `it`) mais également aux périphériques¹ (signaux soulignés et en fushia). Les registres `mip` et `mcause` sont mis à jour par les périphériques : `mip` à chaque cycle (d'où l'absence de signal `mip_we`) et `mcause` lorsqu'une interruption est reçue (signal `irq`).

3.3 Système complet

La figure 1 représente le système complet. Ce dernier intègre une mémoire, ainsi que des périphériques d'entrée sorties. Ces périphériques sont des connections aux LEDs, interrupteurs et boutons poussoirs de la carte et au HDMI. Ce système est implanté dans le fichier `PROC.vhd`.

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 de lecture d'un mot mémoire vers un registre.

Ces périphériques sont interfacés avec le processeur par un *bus* 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 `PROC.vhd` et les explications dans la section 5).

Tant que les instructions d'accès mémoire ne sont pas implantées, un mécanisme de debug vous permet d'utiliser les périphériques et, en particulier, l'affichage sur les LEDs. 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 `x31` : une écriture dans ce registre met la valeur écrite sur le signal `pout`. Comme 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.

<code>int₂</code>	<code>int₁</code>	<code>int₀</code>	<code>LED_{3...0}</code>
0	0	0	<code>pout_{3...0}</code>
0	0	1	<code>pout_{7...4}</code>
0	1	0	<code>pout_{11...8}</code>
0	1	1	<code>pout_{15...12}</code>
1	0	0	<code>pout_{19...16}</code>
1	0	1	<code>pout_{23...20}</code>
1	1	0	<code>pout_{27...24}</code>
1	1	1	<code>pout_{31...28}</code>

TABLE 1 – Configuration des interrupteurs pour affichage sur LEDs

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 F.4, par exemple en utilisant les hyperliens du tableau de la section F.3. 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 que, pour implémenter cette instruction, il faut mettre les 20 bits de poids fort du registre `IR` (champ `imm[31:12]` dans le format U) dans les 20 bits de poids fort du registre `rd` et mettre à zéro ses bits de poids faible. Il faut aussi effectuer $mem_addr \leftarrow mem[PC]$ pour commencer le chargement de l'instruction suivante.

¹Plus précisément, au PLIC (Platform Local Interrupt Controller) et au CLINT (Core Local Interruptor)

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 (voir 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 ajouter 4 à `pc`, il faut identifier un 4 à l'entrée d'un multiplexeur qui passe par un additionneur dont une entrée sera `pc` et dont la sortie se retrouve sur l'entrée du registre `pc`. On voit sur la figure 3 qu'il s'agit de l'additionneur du bas, et qu'il faut sélectionner la valeur `cst_x04` grâce à `T0_PC_Y_sel`, et diriger ce qui sort de l'additionneur sur le registre grâce à `PC_sel`. Pour le comportement de l'instruction `lui` en tant que telle, on peut repérer l'immédiat au format U (signal `immU` dans la figure 3) et le banc de registres. Le seul chemin les reliant passe par un additionneur. Par une addition du champ immédiat avec la constante 0, cet additionneur peut fournir le résultat escompté. L'annexe C.2 permet de valider ce choix, car l'opérande Y de cet additionneur (`PC_Y_select`) peut prendre la valeur `immU` (`PC_Y_immU`) et l'opérande (`PC_X_select`) peut prendre la valeur 0 (`PC_X_cst_x00`). De plus, on peut aussi vérifier que le signal sortant du même additionneur (`PC_X + PC_Y`) est bien une source valable du multiplexeur en entrée du banc de registre RF. L'écriture dans RF, de la donnée sélectionnée selon la valeur du type `DATA_select`, est toujours faite dans le registre destination `rd`. L'opération RTL identifiée est donc $rd \leftarrow (immU) + 0$, équivalent à $rd \leftarrow (IR_{31...12} \parallel 0^{12}) + 0$.

À noter que les constantes extraites de IR (notées `immX` où X est le format de l'instruction) peuvent apparaître plusieurs fois dans la PO, il faudra choisir le bon point d'entrée selon l'utilisation souhaitée.

4.3 Ajout d'états dans la PC

Chaque opération RTL est réalisée en un cycle et doit donc être associée 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 (figure 2), cette suite d'états sera connectée à l'état `S_Decode` et rebouclera vers un état existant, qui permettra de charger correctement l'instruction suivante. Le passage de l'état `S_Decode` au premier état de l'instruction est décidé en fonction de l'encodage des instructions. L'annexe F.3 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 de `lui`, il faut tout d'abord faire l'incrémement de `pc` dans l'état `S_Decode`².

Ensuite, on n'a besoin que d'un seul état pour réaliser l'opération RTL identifiée. La détection du code 0110111 sur les 7 bits de poids faible de IR suffit pour entrer dans cet état. De cet état, on peut passer à l'état `S_Pre_Fetch`, 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 aussi réalisée sans conflit dans notre état `S_LUI` en validant une transaction de lecture vers la mémoire (voir `mem_ce` dans la section C.2). Cette opération devra récupérer la prochaine instruction à exécuter (pointée par PC). En faisant ce choix, on peut directement passer à l'état `S_Fetch` comme proposé dans la section 3.1, au lieu de passer par l'état `S_Pre_Fetch`.

4.4 Mise en œuvre de l'instruction

Il s'agit de décrire dans le fichier VHDL `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 `S_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 commandes qui feront réaliser à la PO les opérations RTL concernées.

²Attention, si cela est vrai pour `lui` et une vaste majorité d'instructions, ce n'est pas le cas pour les instructions de branchement et `auipc`.

Pour [lui](#), après avoir déclaré `S_LUI` au type *State_type*, on pourra insérer le code VHDL ci-dessous dans le processus qui décrit les fonctions de transition et de sortie :

```
when S_LUI =>
  -- rd <- ImmU + 0
  cmd.PC_X_sel <= PC_X_cst_x00;
  cmd.PC_Y_sel <= PC_Y_immU;
  cmd.RF_we <= '1';
  cmd.DATA_sel <= DATA_from_pc;
  -- lecture mem[PC]
  cmd.ADDR_sel <= ADDR_from_pc;
  cmd.mem_ce <= '1';
  cmd.mem_we <= '0';
  -- next state
  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, celle de l'opérande Y, l'opération réalisée par l'additionneur, la sélection du signal en entrée du 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.

Attention : avant de coder vos instructions il est indispensable de sélectionner les valeurs par défaut qui seront affectées aux signaux de commande (`cmd`). Au début du projet, ces signaux sont fixés à une valeur indéfinie (U ou UNDEFINED). Une fois vos valeurs par défaut choisies, vous pouvez coder par exemple votre état `S_LUI` qui mettra à jour seulement les signaux de commande spécifiés précédemment.

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 x31, qui est connecté sur `pout` dans votre 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.3](#)).

Le fichier `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 racine de votre projet*, exécutez :

```
make simulation TOP=PROC PROG=lui
```

Vous pourrez alors comparer dans le simulateur les valeurs obtenues à celles espérées. En cas d'erreur, le simulateur vous permettra d'inspecter les signaux et de remonter à la source de l'erreur. En utilisant le mécanisme d'autotest, vous n'aurez à utiliser le simulateur que pour débusquer des erreurs dans votre code.

Vous pouvez peut être remarquer que parmi les signaux en simulation un grand ensemble d'entre eux sont "undefined". N'oubliez pas que dans le *process* `FSM_comb` de la PC ces signaux n'ont pas été initialisés. Vous devez leur choisir des valeurs par défaut dès maintenant ou au fur et à mesure du projet, en fonction des besoins.

Pour tester sur la carte, exécutez :

```
make fpga TOP=PROC PROG=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.

4.7 Programmes de test

Les programmes suivants sont à traduire en assembleur en affectant les variables à des registres et en utilisant les opérations arithmétiques adéquates. Les programmes à implémenter sont à tester sur carte !

Compteur sur LED

Un registre est incrémenté de 1 puis affiché sur les LED. La sélection des poids forts par les boutons poussoirs permet d'observer l'incrémentation du compteur.

Chenillard minimaliste sur LED

Une chenille s'agrandit sur les LED. L'algorithme est le suivant :

```
procedure chenillard_minimaliste is

index, mask, increment, compteur: Positive;
motif, motif_suivant : Positive;
bit_extraite , bit_extraite_precedent , change : Positive;

begin

  /*Boucle infinie: la derniere instr. doit etre invalide pour revenir ici*/
  while true
  loop
    index := 24;
    mask := 1 << index;
    compteur := compteur+1;
    /* extrait bit numero index */
    bit_extraite := compteur & mask;
    /* detecte passage de 0 a 1 et 1 a 0 du bit 24 du compteur*/
    change := bit_extraite xor bit_extraite_precedent;
    bit_extraite_precedent := bit_extraite;
    /* pour que change soit 0 ou 1 */
    change := change >> index;
    /* le motif n'est decale que tous les 2^24 passages */
    motif := motif << change;
    motif := motif or 0x1;
    afficher(motif); /* sur LEDs */
  end loop; /* a remplacer par une instruction invalide */

end;
```

Chenillard à rotation de motif

Ce chenillard nécessite des branchements inconditionnels et conditionnels.

```

procedure chenillard_rotation is

motif,i : Positive;

begin

    motif := 0x03;
    /* boucle infinie */
    while true
    loop
        /* boucle d'attente */
        for i in 0 to 10000000
        loop
            end loop;
            afficher(motif); /* sur LEDs */
            /* decale le motif vers la gauche */
            motif := motif << 1;
            /* s'il débordé */
            if motif and 0x10 = true then
                /* limite le nouveau motif sur 4 bits */
                motif := motif and 0xF;
                /* met le bit de poids faible a 1 */
                motif := motif or 0x1;
            end if;
        end loop;
    end;

```

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 PROC_bus est ajouté au système. Ce bus intercepte les accès mémoire et sélectionne le périphérique concerné selon l'adresse de l'accès. Pour éviter les conflits, chaque périphérique se voit 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
RAM32	RW	0x00001000 - 0x00008FFF	Mémoire RAM pour les programmes, données et traitant d'interruption (optionnel).
IP_LED	W	0x30000000 - 0x30000003	Mot de 32 bits à afficher sur les LED
IP_PIN (interrupteurs + boutons poussoirs)	R	0x30000008	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
IP_PLIC	RW	0x0C000000 - 0x10000000 0x0C001000 0x0C002000 0x0C200004	Contrôleur d'interruptions au niveau de la plateforme Génère le signal <code>meip</code> . Bit 2 est à 1 si une interruption de IP_PIN est en attente Bit 2 doit être mis à 1 pour autoriser les interruptions provenant de IP_PIN Doit être lu pour acquitter la demande d'interruption en cours (plus de détails dans <code>IP_PLIC.vhd</code>)
IP_CLINT	RW	0x02000000 - 0x0200C000	Contrôleur d'interruptions local : timer Génère le signal <code>mtip</code> . (plus de détails dans <code>IP_CLINT.vhd</code>)
PS_Link	W	0x80000000 - 0x8FFFFFFF	DDR
RAM vidéo	RW	0x40000000 - 0x4004AFFF	RAM Vidéo double port pour VGA

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. Tous les périphériques doivent donc avoir l'interface suivante :

Port	Sens	Type	Description
clk	In	std_logic	Horloge
rst	In	std_logic	Reset
addr	In	waddr	Adresse en provenance du bus
size	In	RF_size_select	Taille de la donnée (mot, demi-mot, octet)
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'
ce	In	std_logic	Habilite une écriture ou une lecture

Paramétrage

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

- La constante `BUS_N_SLAVE` correspond au nombre de périphériques
- La plage d'adresses a été configurée sur le bus. C'est-à-dire les adresses de base et haute ont été ajoutées aux tableaux `base` et `high` du bus.
- Les périphériques ont été instanciés et connectés aux signaux internes : connexion à un périphérique via les signaux `bus_datai`, `bus_ce`, et `bus_we`, d'indice adéquat, ainsi que les signaux `mem_addr`, `mem_d_size` et `mem_dataout` en provenance du processeur.

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

A Gestion du dépôt et validations automatiques

A.1 Dépôt et sources initiales

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

<https://gricad-gitlab.univ-grenoble-alpes.fr/>

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

```
git clone https://gricad-gitlab.univ-grenoble-alpes.fr/fpga-e2i/fpga-e2i5-XX.git
```

dont le suffixe **XX** correspond au numéro du binôme qui vous a été attribué.

Attention : remplacez **XX** par votre numéro de binôme.

Cette commande crée un sous-répertoire **fpga-e2i5-XX** dans lequel vous trouverez les sources de départ.

Attention : vous pouvez accéder à la documentation du logiciel de gestion de versions GIT dans :

<https://git-scm.com/doc>

A.2 Validations et travail à distance

Pour vous aider à mettre au point votre processeur, ainsi que pour nous permettre d'évaluer votre progression et votre travail, on vous propose à la fin de chaque séance, de nous fournir vos avancements (implémentation VHDL et tests en assembleur) à travers du dépôt Git.

Attention : Avant d'utiliser ce mécanisme, il est important que vous ayez fait tous les tests de votre côté (voir mécanisme d'autotest dans la section D.3) .

L'idée globale est de déposer vos fichiers dans le dépôt Git³. Pour cela, vous utiliserez les commandes suivantes (dans la racine de votre projet) :

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

La commande `make realclean` permet de nettoyer votre répertoire avant de faire un `commit`, la commande `git add` indique quels sont les fichiers qui seront ajoutés dans le dépôt, la commande `git commit` enregistre vos modifications dans votre dépôt de travail local (i.e. sur le PC sur lequel vous travaillez) alors que la commande `git push` envoie ces modifications sur le dépôt `git` distant.

Attention : Seuls les fichiers `vhdl` présents dans le répertoire `vhd`, les programmes de tests présents dans le répertoire `program/autotest`, et le fichier `program/sequence_tag` sont analysés par l'outil de validation. Mais vous pouvez déposer tous vos fichiers dans le dépôt tout de même car cela permettra d'en garder une sauvegarde. Comme précisé plus haut, l'interface du composant `CPU` ne doit pas être modifiée.

³<https://gricad-gitlab.univ-grenoble-alpes.fr/>

B Notations

=	test d'égalité
+	addition entière en complément à deux
−	soustraction entière en complément à deux
×	multiplication entière en complément à deux
÷	division entière en complément à deux
mod	reste de la division entière en complément à deux
and	opérateur et bit-à-bit
or	opérateur ou bit-à-bit
nor	opérateur non-ou bit-à-bit
xor	opérateur ou-exclusif bit-à-bit
mem[<i>a</i>]	contenu de la mémoire à l'adresse <i>a</i>
←	assignation
⇒	implication
	concaténation de chaînes de bits
x^n	réplication du bit <i>x</i> dans une chaîne de <i>n</i> bits. Notons que <i>x</i> est un unique bit
$x_{p...q}$	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 $\overset{15}{0} \overset{14}{0} \overset{13}{0} \overset{12}{1} \overset{11}{1} \overset{10}{0} \overset{9}{1} \overset{8}{1} \overset{7}{0} \overset{6}{1} \overset{5}{0} \overset{4}{0} \overset{3}{1} \overset{2}{0} \overset{1}{0} \overset{0}{0}$ 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. $x_{15}^{16} || x_{15...0}$ est la valeur 32 bits avec extension de signe de l'immédiat en complément à deux de 16 bits *x*.

C Organisation du projet

C.1 Répertoires et fichiers

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

Répertoires utiles	
vhd	Les sources VHDL du processeur RISC-V et ses périphériques
vhd/axi	Les sources VHDL du protocole AXI utilisé pour la communication
vhd/bench	Les sources VHDL des environnements de simulation
vhd/hdmi	Les sources VHDL du contrôleur de la sortie vidéo HDMI
program	Les sources des programmes en langage d'assemblage
logiciel	Les sources des applications graphiques en langage C
Divers (à 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
PKG.vhd			Bibliothèque contenant les déclarations des types utilisés dans le projet
CPU.vhd			Assemblage PC+PO
CPU_PC.vhd		A compléter	Partie contrôle
CPU_PO.vhd		A compléter	Partie opérative
CPU_CND.vhd		A compléter	Entité gérant la condition de saut du processeur
CPU_CSR.vhd		(hors projet)	Entité gérant le registre de control/status du processeur (CSR) et les interruptions.
PROC.vhd	top		Processeur RISC-V + Périphériques
PROC_bus.vhd			Gère le décodage d'adresse pour les périphériques
Compléments/Périphériques (à ne pas modifier)			
RAM32.vhd			Mémoire RAM (block mémoire Xilinx)
IP_LED.vhd			Périphériques sortie LED
IP_PIN.vhd			Périphérique bouton poussoir + interrupteurs
IP_CLINT.vhd			Contrôleur d'interruptions local au processeur
IP_PLIC.vhd			Contrôleur d'interruptions au niveau de la plateforme complète
IP_VGA.vhd			Contrôleur de la sortie VGA

Le fichier 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 (figure 3) sont regroupés dans une structure de type PO_cmd, définie dans le fichier PKG.vhd. Voici les différents champs de cette structure :

Champ	Type VHDL	Valeurs possibles	Rôle
ALU_op	ALU_op_type	ALU_plus, ALU_minus	Sélection de l'opération arithmétique effectuée par l'ALU
LOGICAL_op	LOGICAL_op_type	LOGICAL_and, LOGICAL_or, LOGICAL_xor	Sélection de l'opération logique effectuée par l'ALU
ALU_Y_sel	ALU_Y_select	ALU_Y_rf_rs2, ALU_Y_immI	Sélection de l'opérande Y (arithmétique/logique) sur l'ALU
SHIFTER_op	SHIFTER_op_type	SHIFT_rl, SHIFT_ra, SHIFT_ll	Sélection de l'opération de décalage effectuée par l'ALU
SHIFTER_Y_sel	SHIFTER_Y_select	SHIFTER_Y_rs2, SHIFTER_Y_ir_sh	Sélection de l'opérande Y (de décalage) sur l'ALU
RF_we	boolean	true, false	Valide l'écriture dans RF
RF_SIGN_enable	boolean	true, false	Valide l'extension de signe pendant un accès au banc de registres
RF_SIZE_sel	RF_SIZE_select	RF_SIZE_word, RF_SIZE_half, RF_SIZE_byte	Sélection du mot, demi-mot ou octet à écrire dans le banc de registres ou à écrire en mémoire
DATA_sel	DATA_select	DATA_from_alu, DATA_from_logical, DATA_from_mem, DATA_from_pc, DATA_from_slt, DATA_from_shifter, DATA_from_csr	Sélection de la provenance de la donnée à écrire dans le banc de registres
PC_we	boolean	true, false	Valide l'écriture dans PC
PC_sel	PC_select	PC_from_alu, PC_mtvec, PC_rstvec, PC_from_pc, PC_from_mepc	Sélection de la provenance de la donnée à écrire dans PC
PC_X_sel	PC_X_select	PC_X_cst_x00, PC_X_pc	Sélection de l'opérande X sur l'additionneur vers le banc de registres
PC_Y_sel	PC_Y_select	PC_Y_cst_x04, PC_Y_immU	Sélection de l'opérande Y sur l'additionneur vers le banc de registres
TO_PC_Y_sel	TO_PC_Y_select	TO_PC_Y_immB, TO_PC_Y_immJ, TO_PC_Y_cst_x04	Sélection de l'opérande Y sur l'additionneur de PC
AD_we	boolean	true, false	Valide l'écriture dans AD
AD_Y_sel	AD_Y_select	AD_Y_immI, AD_Y_immS	Sélection de l'opérande Y sur l'additionneur de AD
IR_we	boolean	true, false	Valide l'écriture dans IR
ADDR_sel	ADDR_select	ADDR_from_pc, ADDR_from_ad	Sélection de l'adresse vers la mémoire
mem_we	boolean	true, false	Valide une écriture dans la mémoire
mem_ce	boolean	true, false	Valide une transaction vers la mémoire (lecture ou écriture)
cs	PO_cs_cmd	voir détail	Interface aux registres de contrôle/statut (CSR "Control Status Register")

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

- `ALU_op_type` est le type énuméré utilisé pour sélectionner l'opération arithmétique à réaliser par l'ALU.

Valeur	Sémantique
ALU_plus	$ALU_res \leftarrow X + Y$
ALU_minus	$ALU_res \leftarrow X - Y$

- L'opérateur X prend toujours la valeur du registre `rs1`.
- L'opérateur Y peut prendre la valeur du registre `rs2` ou la valeur d'une constante immédiate de type `I` selon le signal `ALU_Y_sel`.
- `LOGICAL_op_type` est le type énuméré utilisé pour sélectionner l'opération logique à réaliser par l'ALU.

Valeur	Sémantique
LOGICAL_and	$LOGICAL_res \leftarrow X \text{ and } Y$
LOGICAL_or	$LOGICAL_res \leftarrow X \text{ or } Y$
LOGICAL_xor	$LOGICAL_res \leftarrow X \oplus Y$

- L'opérateur X prend toujours la valeur du registre `rs1`.
- L'opérateur Y peut prendre la valeur du registre `rs2` ou la valeur d'une constante immédiate de type `I` selon le signal `ALU_Y_sel`.
- `ALU_Y_select` est le type énuméré utilisé pour sélectionner la valeur à fournir sur l'opérande Y de l'ALU dans le cas d'une opération arithmétique/logique.

Valeur	Sémantique
ALU_Y_rf_rs2	Port B du banc de registre pointé par $IR_{24...20}$ (<code>rs2</code>)
ALU_Y_immI	$IR_{31}^{20} \parallel IR_{31...20}$ (constante immédiate I)

- `SHIFTER_op_type` est le type énuméré utilisé pour sélectionner l'opération de décalage à réaliser par l'ALU.

Valeur	Sémantique
SHIFT_r1	$SHIFTER_res \leftarrow X \gg Y_{4...0}$ (logique)
SHIFT_ra	$SHIFTER_res \leftarrow X \ggg Y_{4...0}$ (arithmétique)
SHIFT_l1	$SHIFTER_res \leftarrow X \ll Y_{4...0}$ (logique)

- L'opérateur X prend toujours la valeur du registre `rs1`.
- L'opérateur Y prend la valeur d'une constante de décalage selon le signal `SHIFTER_Y_sel`.
- `SHIFTER_Y_select` est le type énuméré utilisé pour sélectionner la valeur à fournir sur l'opérande Y de l'ALU dans le cas d'une opération de décalage.

Valeur	Sémantique
SHIFTER_Y_rs2	Les 5 bits de poids faible du registre <code>rs2</code>
SHIFTER_Y_ir_sh	$IR_{24...20}$ (<code>shamt</code>)

- `RF_SIZE_select` est le type énuméré utilisé pour sélectionner le mot, le demi-mot ou l'octet à écrire dans le banc de registres. Cette sélection n'est nécessaire que lorsque la donnée provient de la mémoire.

Valeur	Sémantique
RF_SIZE_word	Le mot disponible sur <code>mem_datain</code>
RF_SIZE_half	Un demi-mot choisi sur <code>mem_datain</code> en fonction de la valeur du registre <code>AD₁</code>
RF_SIZE_byte	Un octet choisi sur <code>mem_datain</code> en fonction de la valeur du registre <code>AD_{1...0}</code>

- Dans le cas d'une écriture d'un demi-mot ou d'un octet, une extension de signe est faite en fonction de la valeur du `RF_SIGN_enable`.
- Cette commande est également utilisée lors d'une écriture en mémoire (instructions `sb`, `sh`, `sw`) pour indiquer le nombre de bits à écrire effectivement en mémoire.

- DATA_select est le type énuméré utilisé pour sélectionner la provenance de la donnée à écrire dans le banc de registres.

Valeur	Sémantique
DATA_from_alu	$RF \leftarrow ALU_res$
DATA_from_logical	$RF \leftarrow LOGICAL_res$
DATA_from_mem	$RF \leftarrow REG_to_load$
DATA_from_pc	$RF \leftarrow PC_X + PC_Y$
DATA_from_slt	$RF \leftarrow 0^{31} \parallel SLT$
DATA_from_shifter	$RF \leftarrow SHIFTER_res$
DATA_from_csr	$RF \leftarrow CSR$

- PC_select est le type énuméré utilisé pour sélectionner la provenance de la donnée à écrire dans PC.

Valeur	Sémantique
PC_from_alu	$PC \leftarrow ALU_res_{31...1} \parallel 0$
PC_mtvec	$PC \leftarrow MTVEC$
PC_rstvec	$PC \leftarrow RST_VEC$
PC_from_pc	$PC \leftarrow PC + TO_PC_Y$
PC_from_mepc	$PC \leftarrow MEPC$

- PC_X_select est le type énuméré utilisé pour sélectionner la valeur à fournir sur l'opérande X de l'additionneur vers le banc de registres.

Valeur	Sémantique
PC_X_cst_x00	Constante 0x00000000
PC_X_pc	PC

- PC_Y_select est le type énuméré utilisé pour sélectionner la valeur à fournir sur l'opérande Y de l'additionneur vers le banc de registres.

Valeur	Sémantique
PC_Y_cst_x04	Constante 0x00000004
PC_Y_immU	$IR_{31...12} \parallel 0^{12}$

- TO_PC_Y_select est le type énuméré utilisé pour sélectionner l'opérande Y sur l'additionneur de PC.

Valeur	Sémantique
TO_PC_Y_immB	$IR_{31}^{20} \parallel IR_7 \parallel IR_{30...25} \parallel IR_{11...8} \parallel 0$
TO_PC_Y_immJ	$IR_{31}^{12} \parallel IR_{19...12} \parallel IR_{20} \parallel IR_{30...25} \parallel IR_{24...21} \parallel 0$
TO_PC_Y_cst_x04	Constante 0x00000004

- AD_Y_sel est le type énuméré utilisé pour sélectionner l'opérande Y de l'additionneur vers le registre AD.

Valeur	Sémantique
AD_Y_immI	$IR_{31}^{20} \parallel IR_{31...20}$
AD_Y_immS	$IR_{31}^{20} \parallel IR_{31...25} \parallel IR_{11...7}$

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

Valeur	Sémantique
ADDR_from_pc	$mem_addr \leftarrow PC$
ADDR_from_ad	$mem_addr \leftarrow AD$

C.3 Interface aux registres de contrôle/statut (CSR)

Les signaux de commandes vers les CSR (figure 5) sont regroupés dans une structure de type `P0_cs_cmd`, définie dans le fichier `PKG.vhd`. Voici les différents champs de cette structure :

Champ	Type VHDL	Rôle
CSR_we	CSR_write_enable	Valide l'écriture sur l'un des registres de contrôle/statut
TO_CSR_Sel	TO_CSR_select	Sélection de la provenance de la donnée à écrire dans l'un des registres de contrôle/statut
CSR_sel	CSR_select	Sélection du registre de contrôle/statut à envoyer au banc de registres
MEPC_sel	MEPC_select	Sélection de la provenance de la donnée à écrire dans le registre mepc
MSTATUS_mie_set	boolean	Valide l'écriture de la valeur 1 dans le bit 3 du registre mstatus (même sans avoir CSR_we = CSR_mstatus)
MSTATUS_mie_reset	boolean	Valide l'écriture de la valeur 0 dans le bit 3 du registre mstatus (même sans avoir CSR_we = CSR_mstatus)
CSR_WRITE_mode	CSR_WRITE_mode_type	Sélection du mode d'écriture dans l'un des registres de contrôle/statut

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

- `CSR_write_enable` est le type énuméré utilisé pour valider l'écriture sur l'un des registres de contrôle/statut.

Valeur	Sémantique
CSR_mie	Valide l'écriture sur mie ("Machine Interrupt-Enable Register")
CSR_mstatus	Valide l'écriture sur mstatus ("Machine Status Register")
CSR_mtvec	Valide l'écriture sur mtvec ("Machine Trap-Vector Base-Address Register")
CSR_mepc	Valide l'écriture sur mepc ("Machine Exception Program Counter")

- `TO_CSR_select` est le type énuméré utilisé pour sélectionner la provenance de la donnée à écrire dans l'un des registres de contrôle/statut.

Valeur	Sémantique
TO_CSR_from_rs1	Port A du banc de registre pointé par $IR_{19...15}$ (rs1)
TO_CSR_from_imm	$0^{27} \parallel IR_{19...15}$

- `CSR_select` est le type énuméré utilisé pour sélectionner le registre de contrôle/statut à envoyer au banc de registres.

Valeur	Sémantique
CSR_from_mcause	$CSR \leftarrow mcause$
CSR_from_mip	$CSR \leftarrow mip$
CSR_from_mie	$CSR \leftarrow mie$
CSR_from_mstatus	$CSR \leftarrow mstatus$
CSR_from_mtvec	$CSR \leftarrow mtvec$
CSR_from_mepc	$CSR \leftarrow mepc$

- `MEPC_select` est le type énuméré utilisé pour sélectionner la provenance de la donnée à écrire dans le registre mepc.

Valeur	Sémantique
MEPC_from_pc	mepc \leftarrow PC
MEPC_from_csr	mepc \leftarrow TO_CSR

- CSR_WRITE_mode_type est le type énuméré utilisé pour sélectionner le mode d'écriture dans les registres du contrôle/statut (choisi selon la valeur du CSR_we)

Valeur	Sémantique
WRITE_mode_simple	Permet d'écrire les 32 bits dans le registre
WRITE_mode_set	Permet d'affecter un ensemble de bits
WRITE_mode_clear	Permet de masquer un ensemble de bits

C.4 Signaux d'états de la PO

La PO retourne un ensemble de signaux d'états (*status*), regroupés dans une structure de type PO_status, définie dans le fichier PKG.vhd. Les différents champs sont les suivants :

Champ	Type VHDL	Valeur
IR	w32	L'instruction en cours
JCOND	boolean	Valide un saut
IT	boolean	Valide une interruption

Le type w32 est un vecteur de 32 bits.

D Environnement de conception

D.1 Utilisation du Makefile

D.1.1 La commande make

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

```
make clean
```

permet de nettoyer votre répertoire de travail.

D.1.2 Simulation avec make

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

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

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

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

Les arguments entre [] sont optionnels :

- TOP=<top> sélectionne l'entité à simuler. Au début du projet une seule entité <top> est disponible.
 - **Valeur par défaut** : PROC
 - **Exemple** : TOP=PROC
- 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=lui
 - **Exemple** : PROG=compteur

D.2 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 et les **programmes de test** (simples) sur le FPGA, lancer :

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

Pour télécharger le fichier de configuration et les **applications graphiques** (invader, pong, snake, brick) sur le FPGA, lancer :

```
make fpga [LIB=libfemto] [TOP=<top>] [PROG=<prog>]
```

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

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

D.3 Fonctionnement de l'autotest

D.3.1 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 l'étiquette de la fonctionnalité testée, les valeurs attendues en sortie, les moments où les interruptions seront générées dans le test automatique.

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.3.2 Syntaxe des commentaires à ajouter

- Etiquette du test (INDISPENSABLE)

```
# TAG = <etiquette>
```

À mettre en début de fichier de test, elle permet d'indiquer que ce test permet de valider le cahier des charges identifié par l'étiquette <etiquette>.

Afin de tester tous les tests identifiés par une étiquette donnée, il faut également ajouter cette <etiquette> dans le fichier `program/sequence_tag`

Exemple dans le fichier `program/autotest/lui.s` :

```
# TAG = LUI
```

Exemple dans le fichier `program/sequence_tag` :

```
LUI
```

- 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 31.

Le caractère x est optionnel. Sa présence indique que la valeur peut être répétée plusieurs fois (par exemple si on écrit dans le registre 31 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.3.3 Base de test et regression

Pour définir votre base de test, il suffit d'écrire ligne par ligne dans le fichier `program/sequence_tag` les noms des fonctionnalités (TAG), 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
```

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.

À utiliser donc sans modération avant de faire un push sur votre dépôt GIT.

D.4 Le simulateur VHDL

Lors de l'exécution de la commande permettant de lancer la simulation (voir section [D.1.2](#)), le simulateur **XSIM** de Vivado se présente sous la forme de la figure [6](#).

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, vous devez le chercher dans la hiérarchie, l'ajouter au chronogramme, redémarrer la simulation, puis la faire avancer à nouveau à l'aide des boutons montrés dans la figure [6](#).

Les boutons "Zoom" permettent de voir le chronogramme sur la durée complète de la simulation, ou bien visualiser une zone en particulier.

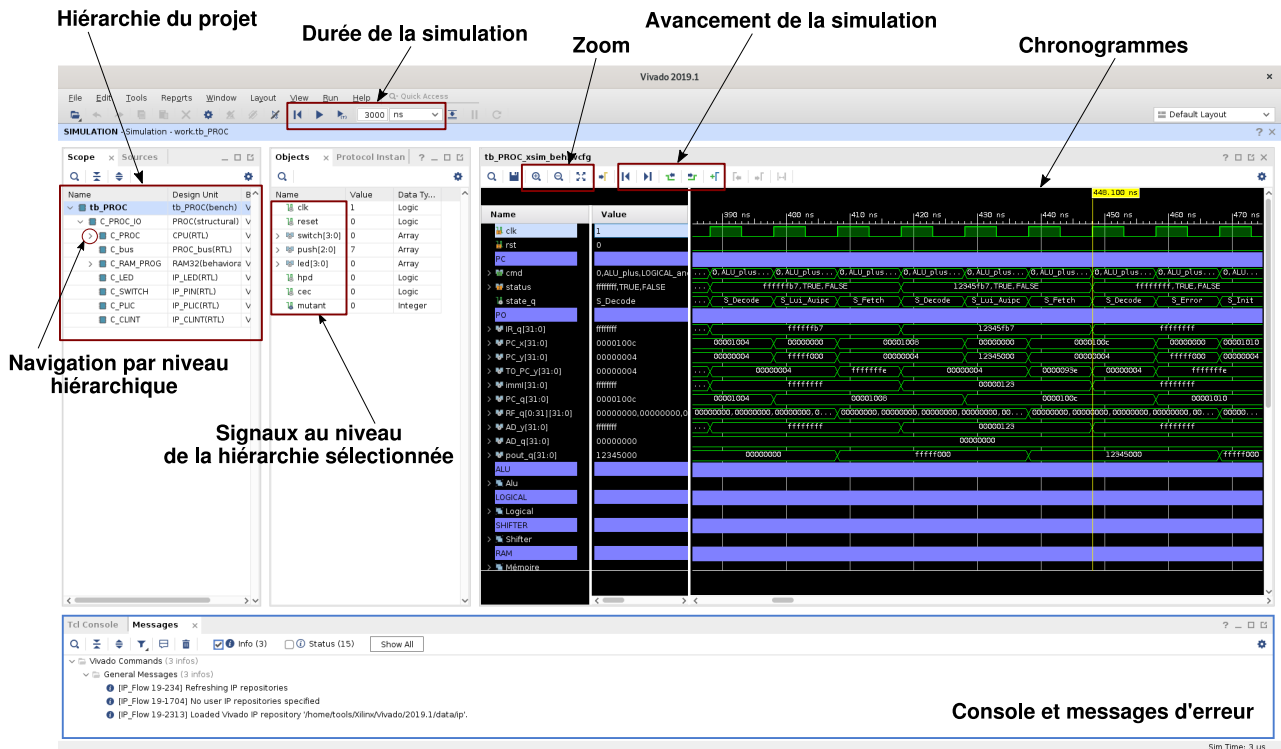


FIGURE 6 – Simulateur XSIM de Vivado (2019.1)

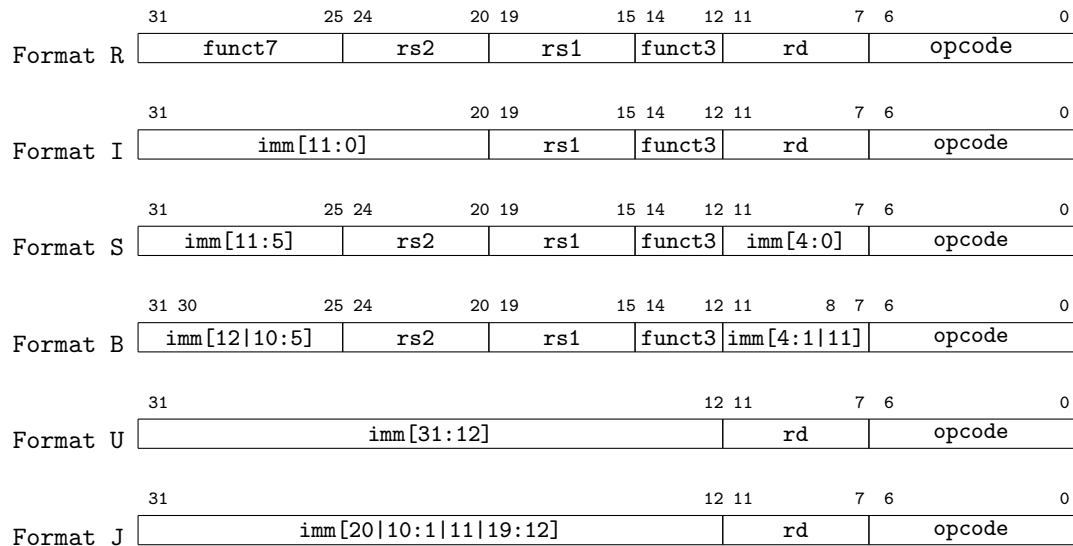
E Documentation

- L'organisation en charge du maintien du standard et de la norme VHDL : <https://www.accellera.org/downloads/ieeee>
- La norme du VHDL : [IEEE Standard VHDL LRM](#)
- Un site web très complet sur le VHDL : [Hamburg VHDL archive](#)
- La fameuse bible de la syntaxe du VHDL, le "VHDL-Cookbook" : [VHDL-Cookbook](#)
- Le site web officiel du RISC-V : <https://riscv.org/>
- Les livres autour du RISC-V : <https://riscv.org/risc-v-books/>
- Les pages Wikipédia sur les sujets : [VHDL](#), [RISC-V](#) (Attention : Wikipédia n'est pas exempte d'erreurs !)

F Les instructions RISC-V (RV32IM)

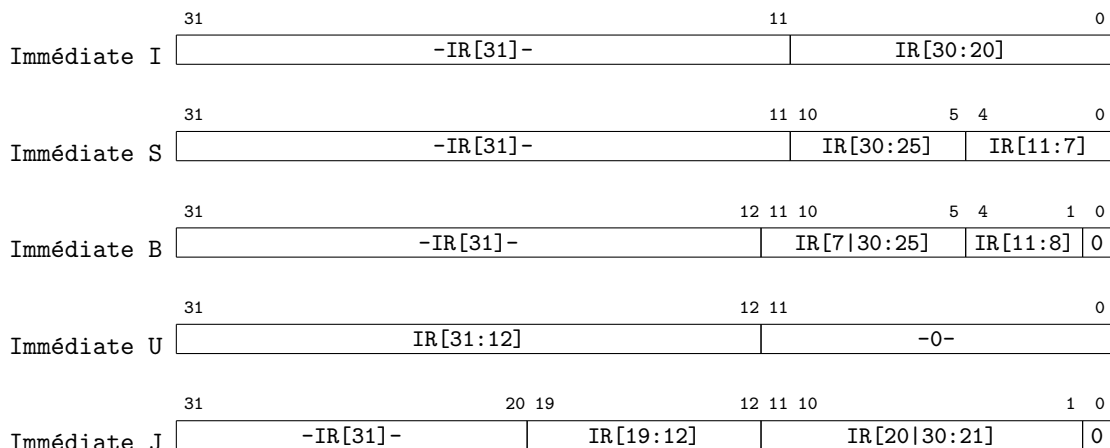
F.1 Format des instructions

Les tables suivantes présentent sous forme compacte les 6 formats des instructions rv32im codées sur 32 bits : le format R pour les opérations registre-registre, le format I pour les opérations immédiates courtes et les « loads », le format S pour les « stores », le format B pour les opérations de saut conditionnel, le format U pour les opérations immédiates longues, et le format J pour les opérations de saut inconditionnel.



F.2 Construction des constantes immédiates

Ci-dessous sont présentés sous forme compacte les 5 types de constantes immédiates produites à partir des instructions rv32im. Dans le RISC-V les constantes immédiates sont toujours étendues de signe. Certains formats construisent la valeur de la constante à partir de la valeur immédiate contenue dans l'instruction de manière assez peu conventionnelle. C'est pourquoi, dans le descriptif des instructions est introduit une constante intermédiaire (cst) pour faciliter la description.



F.3 Encodage des instructions

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

Sous-ensemble RV32I

31	25	24	20	19	15	14	12	11	7	6	0	
imm[31:12]							rd		0110111		U lui	
imm[31:12]							rd		0010111		U auipc	
imm[20 10:1 11 19:12]							rd		1101111		J jal	
imm[11:0]				rs1	000		rd		1100111		I jalr	
imm[12 10:5]	rs2		rs1	000		imm[4:1 11]	1100011		B beq			
imm[12 10:5]	rs2		rs1	001		imm[4:1 11]	1100011		B bne			
imm[12 10:5]	rs2		rs1	100		imm[4:1 11]	1100011		B blt			
imm[12 10:5]	rs2		rs1	101		imm[4:1 11]	1100011		B bge			
imm[12 10:5]	rs2		rs1	110		imm[4:1 11]	1100011		B bltu			
imm[12 10:5]	rs2		rs1	111		imm[4:1 11]	1100011		B bgeu			
imm[11:0]				rs1	000		rd		0000011		I lb	
imm[11:0]				rs1	001		rd		0000011		I lh	
imm[11:0]				rs1	010		rd		0000011		I lw	
imm[11:0]				rs1	100		rd		0000011		I lbu	
imm[11:0]				rs1	101		rd		0000011		I lhu	
imm[11:5]	rs2		rs1	000		imm[4:0]	0100011		S sb			
imm[11:5]	rs2		rs1	001		imm[4:0]	0100011		S sh			
imm[11:5]	rs2		rs1	010		imm[4:0]	0100011		S sw			
imm[11:0]				rs1	000		rd		0010011		I addi	
imm[11:0]				rs1	010		rd		0010011		I slti	
imm[11:0]				rs1	011		rd		0010011		I sltiu	
imm[11:0]				rs1	100		rd		0010011		I xori	
imm[11:0]				rs1	110		rd		0010011		I ori	
imm[11:0]				rs1	111		rd		0010011		I andi	
0000000	shamt		rs1	001		rd		0010011		R slli		
0000000	shamt		rs1	101		rd		0010011		R srli		
0100000	shamt		rs1	101		rd		0010011		R srai		
0000000	rs2		rs1	000		rd		0110011		R add		
0100000	rs2		rs1	000		rd		0110011		R sub		
0000000	rs2		rs1	001		rd		0110011		R sll		
0000000	rs2		rs1	010		rd		0110011		R slt		
0000000	rs2		rs1	011		rd		0110011		R sltu		
0000000	rs2		rs1	100		rd		0110011		R xor		
0000000	rs2		rs1	101		rd		0110011		R srl		
0100000	rs2		rs1	101		rd		0110011		R sra		
0000000	rs2		rs1	110		rd		0110011		R or		
0000000	rs2		rs1	111		rd		0110011		R and		

Sous-ensemble RV32M

0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhsu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

Sous-ensemble privilégié

0011000	00010	00000	000	00000	1110011	- mret
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

Valeur champ csr	Registre concerné	Permissions
0x300	mstatus	lecture/écriture
0x304	mie	lecture/écriture
0x305	mtvec	lecture/écriture
0x341	mepc	lecture/écriture
0x342	mcause	lecture seule
0x344	mip	lecture seule

F.4 Description des instructions

Cette section décrit les instructions du RISC-V (RV32IM) :

- le processeur possède 32 registres de 32 bits chacun, notés x0 à x31 ;
- aucune instruction n'utilise de registre implicite ;
- le registre x0 peut être écrit, mais il vaut néanmoins toujours '0'.

— add —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	000	rd	0110011	

action

Addition registre registre signée.

syntaxe

add rd, rs1, rs2

description

Les contenus des registres rs1 et rs2 sont ajoutés pour former un résultat sur 32 bits qui est placé dans le registre rd.

opération

$rd \leftarrow rs1 + rs2$

format [R](#)

— addi —

encodage

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	000	rd	0010011	

action

Addition registre immédiat signée.

syntaxe

addi rd, rs1, imm

description

Le contenu du registre rs1 est ajouté à l'immédiat sur 12 bits étendu de signe pour former un résultat sur 32 bits qui est placé dans le registre rd.

opération

$rd \leftarrow rs1 + (IR_{31}^{20} \parallel IR_{31...20})$

format [I](#)

— and —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	111	rd	0110011	

action

Et bit-à-bit registre registre

syntaxe

and rd, rs1, rs2

description

Un et bit-à-bit est effectué entre les contenus des registres rs1 et rs2. Le résultat est placé dans le registre rd.

opération

$rd \leftarrow rs1 \text{ and } rs2$

format [R](#)

— andi —

encodage

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	111	rd	0010011	

action

Et bit-à-bit registre immédiat

syntaxe

andi rd, rs1, imm

description

La valeur immédiate sur 12 bits subit une extension de zéros. Un et bit-à-bit est effectué entre cette valeur étendue et le contenu du registre rs1 pour former un résultat placé dans le registre rs2.

opération

$rd \leftarrow (IR_{31}^{20} \parallel IR_{31...20}) \text{ and } rs1$

format [I](#)

— auipc —

encodage

31	12 11	7 6	0
imm[31:12]	rd	0010111	

action

Addition d'un immédiat aux bits de poids fort de pc.

syntaxe

auipc rd, imm

description

La valeur immédiate sur 20 bits décalée à gauche de 12 bits, avec injection de zéros. La constante ainsi obtenue est ajoutée à pc et le résultat est stocké dans rd.

opération

$rd \leftarrow (IR_{31...12} \parallel 0^{12}) + pc$

format [U](#)

— beq —

encodage

31	25	24	20	19	15	14	12	11	7	6	0
imm[12 10:5]			rs2		rs1		000		imm[4:1 11]		1100011

action

Branchement si registre égal registre

syntaxe

beq rs1, rs2, label

description

Les contenus des registres `rs1` et `rs2` sont comparés. S'ils sont égaux, le programme saute à l'adresse correspondant à l'étiquette. La constante `cst` construite (de manière assez exotique) à partir de l'`imm` représente la distance, en avant ou en arrière, à laquelle il faut sauter. Cette distance est calculée par l'assembleur.

opération

$cst = (IR_{31}^{20} \parallel IR_7 \parallel IR_{30...25} \parallel IR_{11...8} \parallel 0)$

$rs1 = rs2 \Rightarrow pc \leftarrow pc + cst$

format [B](#)

— bge —

encodage

31	25	24	20	19	15	14	12	11	7	6	0
imm[12 10:5]			rs2		rs1		101		imm[4:1 11]		1100011

action

Branchement si supérieur ou égal, comparaison signée

syntaxe

bge rs1, rs2, label

description

Les valeurs contenues dans les registres `rs1` et `rs2` sont considérés comme signées. Si le contenu du registre `rs1` est supérieur ou égal à celui du registre `rs2`, le programme saute à l'adresse correspondant à l'étiquette. La constante `cst` construite (de manière assez exotique) à partir de l'`imm` représente la distance, en avant ou en arrière, à laquelle il faut sauter. Cette distance est calculée par l'assembleur.

opération

$cst = (IR_{31}^{20} \parallel IR_7 \parallel IR_{30...25} \parallel IR_{11...8} \parallel 0)$

$rs1 \geq rs2 \Rightarrow pc \leftarrow pc + cst$

format [B](#)

— bgeu —

encodage

31	25	24	20	19	15	14	12	11	7	6	0
imm[12 10:5]			rs2		rs1		111		imm[4:1 11]		1100011

action

Branchement si supérieur ou égal, comparaison non-signée

syntaxe

bgeu rs1, rs2, label

description

Les valeurs contenues dans les registres `rs1` et `rs2` sont considérés comme non-signées. Si le contenu du registre `rs1` est supérieur ou égal à celui du registre `rs2`, le programme saute à l'adresse correspondant à l'étiquette. La constante `cst` construite (de manière assez exotique) à partir de l'`imm` représente la distance, en avant ou en arrière, à laquelle il faut sauter. Cette distance est calculée par l'assembleur.

opération

$$cst = (IR_{31}^{20} \parallel IR_7 \parallel IR_{30...25} \parallel IR_{11...8} \parallel 0)$$
$$rs1 \overset{usg}{\geq} rs2 \Rightarrow pc \leftarrow pc + cst$$

format [B](#)

— blt —

encodage

31	25	24	20	19	15	14	12	11	7	6	0
imm[12 10:5]		rs2		rs1		100		imm[4:1 11]		1100011	

action

Branchement si strictement inférieur, comparaison signée

syntaxe

`blt rs1, rs2, label`

description

Les valeurs contenues dans les registres `rs1` et `rs2` sont considérés comme signées. Si le contenu du registre `rs1` est strictement inférieur à celui du registre `rs2`, le programme saute à l'adresse correspondant à l'étiquette. La constante `cst` construite (de manière assez exotique) à partir de l'`imm` représente la distance, en avant ou en arrière, à laquelle il faut sauter. Cette distance est calculée par l'assembleur.

opération

$$cst = (IR_{31}^{20} \parallel IR_7 \parallel IR_{30...25} \parallel IR_{11...8} \parallel 0)$$
$$rs1 < rs2 \Rightarrow pc \leftarrow pc + cst$$

format [B](#)

— bltu —

encodage

31	25	24	20	19	15	14	12	11	7	6	0
imm[12 10:5]		rs2		rs1		110		imm[4:1 11]		1100011	

action

Branchement si strictement inférieur, comparaison non-signée

syntaxe

`bltu rs1, rs2, label`

description

Les valeurs contenues dans les registres `rs1` et `rs2` sont considérés comme non-signées. Si le contenu du registre `rs1` est strictement inférieur à celui du registre `rs2`, le programme saute à l'adresse correspondant à l'étiquette. La constante `cst` construite (de manière assez exotique) à partir de l'`imm` représente la distance, en avant ou en arrière, à laquelle il faut sauter. Cette distance est calculée par l'assembleur.

opération

$$cst = (IR_{31}^{20} \parallel IR_7 \parallel IR_{30...25} \parallel IR_{11...8} \parallel 0)$$

$rs1 \overset{usg}{<} rs2 \Rightarrow pc \leftarrow pc + cst$

format **B**

— bne —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	

action

Branchement si registre différent de registre

syntaxe

bne rs1, rs2, label

description

Les contenus des registres rs1 et rs2 sont comparés. S'ils sont différents, le programme saute à l'adresse correspondant à l'étiquette. La constante cst construite (de manière assez exotique) à partir de l'imm représente la distance, en avant ou en arrière, à laquelle il faut sauter. Cette distance est calculée par l'assembleur.

opération

$cst = (IR_{31}^{20} \parallel IR_7 \parallel IR_{30...25} \parallel IR_{11...8} \parallel 0)$
 $rs1 \neq rs2 \Rightarrow pc \leftarrow pc + cst$

format **B**

— div —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	100	rd	0110011	

action

Division entière signée

syntaxe

div rd, rs1, rs2

description

Le contenu du registre rs1 est divisé par le contenu du registre rs2, le contenu des deux registres étant considéré comme des nombres en complément à deux (signés). Le quotient resultant de la division est placé dans le registre rd.

opération

$rd \leftarrow \frac{rs1}{rs2}$

format **R**

— divu —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	101	rd	0110011	

action

Division entière non-signée

syntaxe

divu rd, rs1, rs2

description

Le contenu du registre `rs1` est divisé par le contenu du registre `rs2`, le contenu des deux registres étant considéré comme des nombres non-signés. Le quotient resultant de la division est placé dans le registre `rd`.

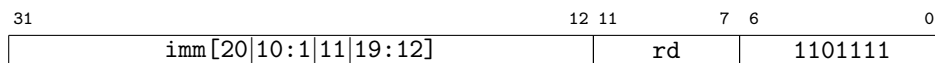
opération

$$rd \leftarrow \frac{0 \parallel rs1}{0 \parallel rs2}$$

format [R](#)

— jal —

encodage



action

Saut ou appel de fonction inconditionnel immédiat

syntaxe

jal rd, label

description

L'adresse de l'instruction suivant le `jal` est sauvée dans le registre `rd`. On effectue un simple saut (sans sauvegarder l'adresse de retour) en choisissant `x0` pour `rd`. Le programme saute inconditionnellement à l'adresse correspondant à l'étiquette. La constante `cst` construite (de manière tout aussi exotique mais cependant différente de celle des branchements) à partir de l'`imm` représente la distance, en avant ou en arrière, à laquelle il faut sauter. Cette distance est calculée par l'assembleur.

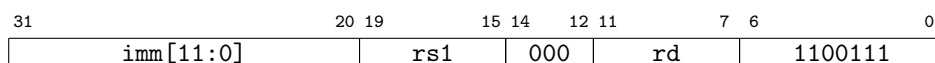
opération

$$\begin{aligned} rd &\leftarrow pc + 4 \\ cst &= (IR_{31}^{12} \parallel IR_{19...12} \parallel IR_{20} \parallel IR_{30...25} \parallel IR_{24...21} \parallel 0) \\ pc &\leftarrow pc + cst \end{aligned}$$

format [J](#)

— jalr —

encodage



action

Saut ou appel de fonction inconditionnel registre plus immédiat

syntaxe

jalr rd, imm(rs1)

description

Le programme saute à l'adresse contenue dans le registre `rs1` auquel la constante sur 12 bits étendue de signe a été ajoutée, puis le bit de poids faible mis à zéro. L'adresse de l'instruction suivant le `jalr` est sauvée dans le registre `rd`. Si `rd` est `x0`, l'instruction est un simple saut.

opération

$$\begin{aligned} rd &\leftarrow pc + 4 \\ pc &\leftarrow (rs1 + (IR_{31}^{20} \parallel IR_{31...20}))_{31...1} \parallel 0 \end{aligned}$$

format [I](#)

— lb —

encodage

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	000	rd	0000011	

action

Lecture d'un octet signé de la mémoire

syntaxe

lb rd, imm(rs1)

description

L'adresse de chargement est la somme de la valeur immédiate sur 12 bits étendue de signe, et du contenu du registre rs1. Le contenu de cette adresse subit une extension de signe et est ensuite placé dans le registre rd.

opération

$rd \leftarrow \text{mem}[(\text{IR}_{31}^{20} \parallel \text{IR}_{31\dots20}) + rs1]_7^{24} \parallel \text{mem}[(\text{IR}_{31}^{20} \parallel \text{IR}_{31\dots20}) + rs1]_{7\dots0}$

format **I**

— lbu —

encodage

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	100	rd	0000011	

action

Lecture d'un octet non-signé de la mémoire

syntaxe

lbu rd, imm(rs1)

description

L'adresse de chargement est la somme de la valeur immédiate sur 12 bits étendue de signe, et du contenu du registre rs1. Le contenu de cette adresse est étendu avec des zéros et est ensuite placé dans le registre rd.

opération

$rd \leftarrow 0^{24} \parallel \text{mem}[(\text{IR}_{31}^{20} \parallel \text{IR}_{31\dots20}) + rs1]_{7\dots0}$

format **I**

— lh —

encodage

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	001	rd	0000011	

action

Lecture d'un demi-mot signé de la mémoire

syntaxe

lh rd, imm(rs1)

description

L'adresse de chargement est la somme de la valeur immédiate sur 12 bits étendue de signe, et du contenu du registre rs1. Le demi-mot contenu à cette adresse subit une extension de signe et est ensuite placé dans le registre rd. Attention, le bit de poids faible de l'adresse résultante doit être à zéro.

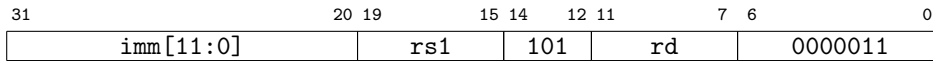
opération

$rd \leftarrow \text{mem}[(\text{IR}_{31}^{20} \parallel \text{IR}_{31\dots20}) + rs1]_{15}^{16} \parallel \text{mem}[(\text{IR}_{31}^{20} \parallel \text{IR}_{31\dots20}) + rs1]_{15\dots0}$

format **I**

— lhu —

encodage



action

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

syntaxe

lhu rd, imm(rs1)

description

L'adresse de chargement est la somme de la valeur immédiate sur 12 bits étendue de signe, et du contenu du registre rs1. Le demi-mot contenu à cette adresse est étendu de zéros et est ensuite placé dans le registre rd. Attention, le bit de poids faible de l'adresse résultante doit être à zéro.

opération

$rd \leftarrow 0^{16} \parallel \text{mem}[(IR_{31}^{20} \parallel IR_{31...20}) + rs1]_{15...0}$

format **I**

— lui —

encodage



action

Lecture d'une constante dans les poids forts

syntaxe

lui rd, imm

description

La constante immédiate de 20 bits est décalée de 12 bits à gauche, et complétée de zéros. La valeur ainsi obtenue est placée dans rd.

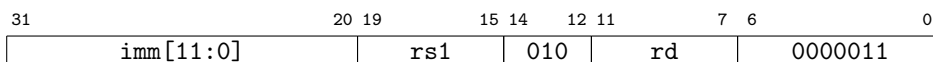
opération

$rd \leftarrow (IR_{31...12} \parallel 0^{12})$

format **U**

— lw —

encodage



action

Lecture d'un mot de la mémoire

syntaxe

lw rd, imm(rs1)

description

L'adresse de chargement est la somme de la valeur immédiate sur 12 bits étendue de signe, et du contenu du registre rs1. Le contenu de cette adresse est placé dans le registre rd. Attention, les deux bits de poids faible de l'adresse résultante doivent être à zéro.

opération

$rd \leftarrow \text{mem}[(IR_{31}^{20} \parallel IR_{31...20}) + rs1]$

format **I**

— mul —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	000	rd	0110011	

action

Multiplication registre registre, poids faibles

syntaxe

mul rd, rs1, rs2

description

Le contenu du registre rs1 est multiplié par le contenu du registre rs2, et les 32 bits de poids faible du résultat de l'opération sont placés dans rd.

opération

$rd \leftarrow [rs1 \times rs2]_{31...0}$

format **R**

— mulh —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	001	rd	0110011	

action

Multiplication registre registre, opérandes signés, poids forts

syntaxe

mulh rd, rs1, rs2

description

Le contenu du registre rs1 est multiplié par le contenu du registre rs2, tous deux considérés comme signés, et les 32 bits de poids fort du résultat de l'opération sont placés dans rd.

opération

$rd \leftarrow [rs1 \times rs2]_{63...32}$

format **R**

— mulhsu —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	010	rd	0110011	

action

Multiplication registre registre, premier opérande signé, second non signé, poids forts

syntaxe

mulhsu rd, rs1, rs2

description

Le contenu du registre rs1 considéré comme signé est multiplié par le contenu du registre rs2 considéré comme non-signé, et les 32 bits de poids fort du résultat de l'opération sont placés dans rd.

opération

$rd \leftarrow [rs1 \times (0 \parallel rs2)]_{63...32}$

format [R](#)

— mulhu —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	011	rd	0110011	

action

Multiplication registre registre, opérandes non-signés, poids forts

syntaxe

mulhu rd, rs1, rs2

description

Le contenu du registre rs1 est multiplié par le contenu du registre rs2, tous deux considérés comme non-signés, et les 32 bits de poids fort du résultat de l'opération sont placés dans rd.

opération

$rd \leftarrow [(0 \parallel rs1) \times (0 \parallel rs2)]_{63...32}$

format [R](#)

— or —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	110	rd	0110011	

action

Ou bit-à-bit registre registre

syntaxe

or rd, rs1, rs2

description

Un ou bit-à-bit est effectué entre les contenus des registres rs1 et rs2. Le résultat est placé dans le registre rd.

opération

$rd \leftarrow rs1 \text{ or } rs2$

format [R](#)

— ori —

encodage

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	110	rd	0010011	

action

Ou bit-à-bit registre immédiat

syntaxe

ori rd, rs1, imm

description

La valeur immédiate sur 12 bits est étendue de signe. Un ou bit-à-bit est effectué entre cette valeur étendue et le contenu du registre rs1 pour former un résultat placé dans le registre rd.

opération

$$rd \leftarrow (IR_{31}^{20} \parallel IR_{31...20}) \text{ or } rs1$$

format [I](#)

— rem —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	110	rd	0110011	

action

Reste de la division entière signée

syntaxe

rem rd, rs1, rs2

description

Le contenu du registre rs1 est divisé par le contenu du registre rs2, le contenu des deux registres étant considéré comme des nombres en complément à deux (signés). Le reste de la division est placé dans le registre rd.

opération

$$rd \leftarrow rs1 \bmod rs2$$

format [R](#)

— remu —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	111	rd	0110011	

action

Reste de la division entière non-signée

syntaxe

remu rd, rs1, rs2

description

Le contenu du registre rs1 est divisé par le contenu du registre rs2, le contenu des deux registres étant considéré comme des nombres non-signés. Le reste de la division est placé dans le registre rd.

opération

$$rd \leftarrow (0 \parallel rs1) \bmod (0 \parallel rs2)$$

format [R](#)

— sb —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	

action

Écriture d'un octet en mémoire

syntaxe

sb rs2, imm(rs1)

description

La constante cst construite à partir de l'imm est étendue de signe et sommée avec le contenu du registre rs1. L'octet de poids faible du registre rs2 est écrit à l'adresse ainsi calculée.

opération

$$cst = (IR_{31}^{20} \parallel IR_{31...25} \parallel IR_{11...7})$$
$$mem[cst + rs1] \leftarrow rs2_{7...0}$$

format **S**

— sh —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	

action

Écriture d'un demi-mot en mémoire

syntaxe

sh rs2, imm(rs1)

description

La constante cst construite à partir de l'imm est étendue de signe et sommée avec le contenu du registre rs1. Les deux octets de poids faible du registre rs2 sont écrit à l'adresse ainsi calculée. Le bit de poids faible de cette adresse doit être à zéro.

opération

$$cst = (IR_{31}^{20} \parallel IR_{31...25} \parallel IR_{11...7})$$
$$mem[cst + rs1] \leftarrow rs2_{15...0}$$

format **S**

— sll —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	001	rd	0110011	

action

Décalage à gauche

syntaxe

sll rd, rs1, rs2

description

Le registre rs1 est décalé à gauche de la valeur immédiate codée dans les 5 bits du rs2, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre rd.

opération

$$rd \leftarrow rs1_{31-rs2_{4...0}...0} \parallel 0^{rs2_{4...0}}$$

format **R**

— slli —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	shamt	rs1	001	rd	0010011	

action

Décalage à gauche immédiat

syntaxe

slli rd, rs1, imm

description

Le registre `rs1` est décalé à gauche du nombre de bits spécifiés par l'immédiate `shamt`, des zéros étant introduits dans les bits de poids faibles. L'immédiate `shamt` occupe les bits habituellement utilisés pour coder `rs2`. Le résultat est placé dans le registre `rd`.

opération

$$rd \leftarrow rs2_{31-shamt...0} \parallel 0^{shamt}$$

format **R**

— slt —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	010	rd	0110011	

action

Comparaison signée registre registre

syntaxe

`slt rd, rs1, rs2`

description

Le contenu du registre `rs1` est comparé au contenu du registre `rs2`, les deux valeurs étant considérées comme des quantités signées. Si la valeur contenue dans `rs1` est inférieure à celle contenue dans `rs2`, alors `rd` prend la valeur '1', sinon il prend la valeur '0'.

opération

$$rs1 < rs2 \Rightarrow rd \leftarrow 0^{31} \parallel 1$$

$$rs1 \geq rs2 \Rightarrow rd \leftarrow 0^{32}$$

format **R**

— slti —

encodage

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	010	rd	0010011	

action

Comparaison signée registre immédiat

syntaxe

`slti rd, rs1, imm`

description

Le contenu du registre `rs1` est comparé à la valeur immédiate sur 12 bits qui a subi une extension de signe, les deux valeurs étant considérées comme des quantités signées. Si la valeur contenue dans `rs1` est inférieure à celle de l'immédiate étendu, alors `rd` prend la valeur '1', sinon il prend la valeur '0'.

opération

$$rs1 < (IR_{31}^{20} \parallel IR_{31...20}) \Rightarrow rd \leftarrow 0^{31} \parallel 1$$

$$rs1 \geq (IR_{31}^{20} \parallel IR_{31...20}) \Rightarrow rd \leftarrow 0^{32}$$

format **I**

— sltiu —

encodage

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	011	rd	0010011	

action

Comparaison non-signée registre immédiat

syntaxe

`sltiu rd, rs1, imm`

description

Le contenu du registre `rs1` est comparé à la valeur immédiate sur 12 bits qui a subi une extension de signe. Les deux valeurs sont considérées comme des quantités non-signées. Si la valeur contenue dans `rs1` est inférieure à celle de l'immédiat étendu, alors `rd` prend la valeur '1', sinon il prend la valeur '0'.

opération

$$(0 \parallel rs1) < (0 \parallel (IR_{31}^{20} \parallel IR_{31...20})) \Rightarrow rd \leftarrow 0^{31} \parallel 1$$
$$(0 \parallel rs1) \geq 0 \parallel (IR_{31}^{20} \parallel IR_{31...20}) \Rightarrow rd \leftarrow 0^{32}$$

format [I](#)

— sltu —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	011	rd	0110011	

action

Comparaison non-signée registre registre

syntaxe

`sltu rd, rs1, rs2`

description

Le contenu du registre `rs1` est comparé au contenu du registre `rs2`, les deux valeurs étant considérés comme des quantités non-signées. Si la valeur contenue dans `rs1` est inférieure à celle contenue dans `rs2`, alors `rd` prend la valeur '1', sinon il prend la valeur '0'.

opération

$$(0 \parallel rs1) < (0 \parallel rs2) \Rightarrow rd \leftarrow 0^{31} \parallel 1$$
$$(0 \parallel rs1) \geq (0 \parallel rs2) \Rightarrow rd \leftarrow 0^{32}$$

format [R](#)

— sra —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0100000	rs2	rs1	101	rd	0110011	

action

Décalage à droite arithmétique registre

syntaxe

`sra rd, rs1, rs2`

description

Le registre `rs1` est décalé à droite du nombre de bits spécifiés dans les 5 bits de poids faible du registre `rs2`, le signe de `rs1` étant introduit dans les bits de poids fort ainsi libérés. Le résultat est placé dans le registre `rd`.

opération

$$rd \leftarrow rs1_{31}^{rs2_{24...0}} \parallel rs1_{31...rs2_{4...0}...0}$$

format [R](#)

— srai —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0100000	shamt	rs1	101	rd	0010011	

action

Décalage à droite arithmétique immédiat

syntaxe

srai rd, rs1, shamt

description

Le registre rs1 est décalé à droite de la valeur immédiate codée dans les 5 bits du champ shamt, le bit de signe du registre étant introduit dans les bits de poids fort. Le résultat est placé dans le registre rd.

opération

$rd \leftarrow rs1_{31}^{shamt} \parallel rs1_{31...shamt}$

format [R](#)

— srl —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	101	rd	0110011	

action

Décalage à droite logique registre

syntaxe

srl rd, rs1, rs2

description

Le registre rs1 est décalé à droite du nombre de bits spécifiés dans les 5 bits de poids faible du registre rs2, des zéros étant introduits dans les bits de poids fort ainsi libérés. Le résultat est placé dans le registre rd.

opération

$rd \leftarrow 0^{rs2_{4...0}} \parallel rs1_{31...rs2_{4...0}}$

format [R](#)

— srli —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	shamt	rs1	101	rd	0010011	

action

Décalage à droite logique immédiat

syntaxe

srli rd, rs1, shamt

description

Le registre rs1 est décalé à droite de la valeur immédiate codée dans les 5 bits du champ shamt, des zéros étant introduits dans les bits de poids fort. Le résultat est placé dans le registre rd.

opération

$rd \leftarrow 0^{shamt} \parallel rs1_{31...shamt}$

format [R](#)

— sub —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0100000	rs2	rs1	000	rd	0110011	

action

Soustraction registre registre signée

syntaxe

sub rd, rs1, rs2

description

Le contenu du registre rs2 est soustrait du contenu du registre rs1 pour former un résultat sur 32 bits qui est placé dans le registre rd.

opération

$rd \leftarrow rs1 - rs2$

format [R](#)

— sw —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	

action

Écriture d'un mot en mémoire

syntaxe

sw rs2, imm(rs1)

description

La constante cst construite à partir de l'imm est étendue de signe et sommée avec le contenu du registre rs1. La mot contenu dans le registre rs2 est écrit à l'adresse ainsi calculée.

opération

$cst = (IR_{31}^{20} \parallel IR_{31...25} \parallel IR_{11...7})$
 $mem[cst + rs1] \leftarrow rs2$

format [S](#)

— xor —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	100	rd	0110011	

action

Ou-exclusif bit-à-bit registre registre

syntaxe

xor rd, rs1, rs2

description

Un ou-exclusif bit-à-bit est effectué entre les contenus des registres rs1 et rs2. Le résultat est placé dans le registre rd.

opération

$rd \leftarrow rs1 \text{ xor } rs2$

format [R](#)

— xori —

encodage

31	20 19	15 14	12 11	7 6	0
imm[11:0]		rs1	100	rd	0010011

action

Ou-exclusif bit-à-bit registre immédiat

syntaxe

xori rd, rs1, imm

description

La valeur immédiate sur 12 bits subit une extension de signe. Un ou-exclusif bit-à-bit est effectué entre cette valeur étendue et le contenu du registre rs1 pour former un résultat placé dans le registre rs2.

opération

$rd \leftarrow (IR_{31}^{20} \parallel IR_{31...20}) \text{ xor } rs1$

format I

— mret —

encodage

31	25 24	20 19	15 14	12 11	7 6	0
0011000		00010	00000	000	00000	1110011

action

Retour d'exception (ou d'interruption)

syntaxe

mret

description pour les extensions

Le programme saute à l'adresse stockée dans le registre mepc. Le bit mie du registre mstatus prend la valeur '1' pour réautoriser les interruptions⁴.

opération

$pc \leftarrow mepc$
 $mstatus_3 \leftarrow 1$

format -

— csrrw —

encodage

31	20 19	15 14	12 11	7 6	0
csr		rs1	001	rd	1110011

action

Ecriture et récupération d'un registre csr (Control and Status Register)

syntaxe

csrrw rd, csr, rs1

description

La valeur présente dans le registre csr est enregistrée dans rd. La valeur présente dans le registre rs1 est enregistrée dans csr.

⁴On utilisera cette simplification, en vrai la valeur du bit mie prend la valeur d'un bit mpie du même registre, et c'est le bit mpie qui passe à 1

opération

$rd \leftarrow csr$

$csr \leftarrow rs1$

format spécifique CSR proche de I

— csrrs —

encodage

31	20	19	15	14	12	11	7	6	0
csr			rs1		010	rd		1110011	

action

Mise à 1 de certains bits et récupération d'un registre `csr` (Control and Status Register)

syntaxe

`csrrs rd,csr,rs1`

description

La valeur présente dans le registre `csr` est enregistrée dans `rd`. Les bits à 1 du registre `rs1` sont mis à 1 dans `csr`.

opération

$rd \leftarrow csr$

$csr \leftarrow csr \text{ or } rs1$

format spécifique CSR proche de I

— csrrc —

encodage

31	20	19	15	14	12	11	7	6	0	
csr				rs1		011	rd		1110011	

action

Mise à 0 de certains bits et récupération d'un registre `csr` (Control and Status Register)

syntaxe

`csrrc rd,csr,rs1`

description

La valeur présente dans le registre `csr` est enregistrée dans `rd`. Les bits à 1 du registre `rs1` sont mis à 0 dans `csr`.

opération

$rd \leftarrow csr$

$csr \leftarrow csr \text{ and } \overline{rs1}$

format spécifique CSR proche de I

— csrrwi —

encodage

31	20	19	15	14	12	11	7	6	0
csr				zimm		101	rd		1110011

action

Ecriture et récupération d'un registre `csr` (Control and Status Register)

syntaxe

`csrrwi rd,csr,zimm`

description

La valeur présente dans le registre `csr` est enregistrée dans `rd`. Les 5 bits `zimm` sont mis dans les 5 bits de poids faible `csr`, les autres sont mis à 0.

opération

$rd \leftarrow csr$

$csr \leftarrow 0^{27} \parallel zimm$

format spécifique CSR proche de [I](#)

— csrrsi —

encodage

31	20 19	15 14	12 11	7 6	0
csr		zimm	110	rd	1110011

action

Mise à 1 de certains bits et récupération d'un registre `csr` (Control and Status Register)

syntaxe

`csrrsi rd,csr,zimm`

description

La valeur présente dans le registre `csr` est enregistrée dans `rd`. Pour les 5 bits de poids faible du registre `csr` les bits à 1 de `zimm` sont mis à 1, les autres bits sont inchangés.

opération

$rd \leftarrow csr$

$csr \leftarrow csr \text{ or } (0^{27} \parallel zimm)$

format spécifique CSR proche de [I](#)

— csrrci —

encodage

31	20 19	15 14	12 11	7 6	0
csr		zimm	111	rd	1110011

action

Mise à 0 de certains bits et récupération d'un registre `csr` (Control and Status Register)

syntaxe

`csrrci rd,csr,zimm`

description

La valeur présente dans le registre `csr` est enregistrée dans `rd`. Pour les 5 bits de poids faible du registre `csr` les bits à 1 de `zimm` sont mis à 0, les autres bits sont inchangés.

opération

$rd \leftarrow csr$

$csr \leftarrow csr \text{ and } \overline{(0^{27} \parallel zimm)}$

format spécifique CSR proche de [I](#)