

Architecture des Ordinateurs
L2-S4 CLM

Cécile Germain-Renaud

Chapitre 1

Introduction

1.1 Le modèle de Von Neumann

L'ordinateur est une machine électronique, qui traite l'information dans une unité centrale (UC, ou CPU pour Central Processing Unit), selon un programme qui est enregistré en mémoire. Les données fournies en entrée par un organe d'entrée (par exemple de type clavier) sont traitées par l'unité centrale en fonction du programme pour délivrer les résultats en sortie, via un organe de sortie (par exemple un écran).

Cette définition très générale implique de définir ce que sont l'*information* et le *traitement*. L'information est numérisée, c'est à dire limitée à des valeurs discrètes, en l'occurrence binaires ; les détails du codage binaire de l'information ont été traités dans la première partie. Par ailleurs, la définition implique que le comportement d'un programme qui ne fait pas d'entrées-sorties ne peut être spécifié.

Le traitement suit le *modèle d'exécution de Von Neumann*.

- La mémoire contient les instructions et les données.
- La mémoire est formée d'un ensemble de mots de longueur fixe, chaque mot contenant une information codée en binaire. Chaque mot de la mémoire est accessible par l'intermédiaire de l'adresse mémoire. Le temps d'accès à un mot est le même quelle que soit la place du mot en mémoire : ce type d'accès est appelé aléatoire, et les mémoires sont appelées RAM (random access memory).
- Les instructions sont exécutées en séquence. Le CPU conserve l'adresse de la prochaine instruction à exécuter dans un registre, appelé PC (Program Counter) ; pour chaque instruction, le CPU effectue les actions suivantes :
 - lire l'instruction à l'adresse PC et incrémenter PC ;
 - exécuter l'instruction.

Le premier ordinateur a été la machine ENIAC, construite à l'université de Pennsylvanie (Moore School) pendant la seconde guerre mondiale, par les ingénieurs J. P. Eckert et J. Mauchly. Elle avait été commandée par l'armée américaine pour le calcul des tables de tir pour l'artillerie. L'ENIAC a été la première machine *programmable*, c'est à

dire dont la séquence d'opérations n'était pas prédéfinie, contrairement à un automate du type caisse enregistreuse. Mais la nouveauté radicale de cette approche n'était pas complètement perçue, ce dont témoigne le nom de la machine : ENIAC signifie *Electronic Numerical Integrator And Calculator*. C'est le mathématicien John Von Neuman, intégré en 1944 au projet ENIAC, qui formalisa les concepts présents dans ce premier ordinateur, et les développa sous le nom de projet EDVAC : *Electronic Discrete Variable Automatic Computer*. La création d'un nouveau terme, Computer et non Calculator, correspond à une rupture théorique fondamentale. Comme cela se reproduira souvent dans l'histoire de l'informatique, ce progrès théorique majeur aboutit dans le très court terme à un semi-échec pratique : l'équipe se dispersa, et l'EDVAC ne fut opérationnel qu'en 1952. Von Neumann contribua à la construction d'une machine prototype universitaire à Princeton, L'IAS. Eckert et J. Mauchly fondèrent une entreprise qui réalisa le premier ordinateur commercial, l'UNIVAC-1, dont 48 exemplaires furent vendus.

1.2 L'unité centrale

Pour exécuter un programme, l'unité centrale appelle les instructions depuis la mémoire, et les données que traitent ces instructions. En effet, dans l'état actuel de la technologie, le CPU ne peut travailler que sur des informations physiquement stockées "près" des organes de calcul qu'elle contient, précisément dans des registres placés sur le même circuit intégré que les unités de calcul.

L'unité centrale se décompose en une *partie opérative*, ou *chemin de données* et une *partie contrôle*.

Le chemin de données est organisé autour d'une unité arithmétique et logique (UAL) et d'opérateurs arithmétiques flottants qui effectuent les opérations sur les données. Les opérandes, initialement lus en mémoire, et les résultats des opérations, sont stockés dans des organes de mémorisations internes au CPU, les *registres*. Certains de ces registres sont visibles : leur nom (ou leur numéro) est présent dans l'instruction, et le programmeur en langage machine peut décider quel registre lire ou écrire ; par exemple, l'instruction ADD R1, R2, R3 nomme les trois registres R1, R2 et R3. D'autres registres ne sont pas visibles, et servent à des besoins de stockage temporaire. Le chemin de données détermine les deux caractéristiques fondamentales d'une unité centrale, le *temps de cycle* et la *largeur du chemin de données*.

- Le temps de cycle T_c est essentiellement le temps d'une opération de l'UAL ; dans les processeurs les plus récents, le temps de cycle est également déterminé par le temps de propagation des signaux dans les connexions. L'inverse du temps de cycle est la fréquence ; si l'unité de temps de cycle est la seconde, l'unité de fréquence est le Hertz. Par exemple, un processeur cadencé à 500MHz a un temps de cycle de $\frac{1}{500 \times 10^6} = 2ns = 2 \times 10^{-9}s$.
- La largeur du chemin de données est la taille de l'information traitée par la partie opérative. Les premiers microprocesseurs étaient des 8 bits, les processeurs généralistes

actuels sont 32 ou 64 bits, le processeur graphique de la Playstation 2 est un 128 bits.

La partie contrôle effectue le séquençage des différentes instructions en fonction des résultats des opérations et actionne les circuits logiques de base pour réaliser les transferts et traitements de données. Elle contrôle également la synchronisation avec les autres composants.

A cheval entre le chemin de données et la partie contrôle, on trouve deux registres spécialisés : le compteur de programme PC, dont on a déjà parlé, et le *registre instruction* RI. Comme les autres informations, l'instruction ne peut être traitée par le CPU que si elle y est physiquement présente. La lecture de l'instruction consiste donc à copier l'instruction depuis la mémoire vers le registre RI. Comme le registre RI peut contenir une constante opérande, il appartient au chemin de données ; comme il contient l'instruction, que la partie contrôle décode pour activer les circuits logiques qui exécutent l'instruction, il appartient aussi à la partie contrôle. PC contrôle la lecture de l'instruction, mais il peut aussi être modifié par les instructions qui réalisent des branchements.

1.3 Architecture logicielle et architecture matérielle

Le fonctionnement d'un ordinateur peut s'envisager suivant la hiérarchie des niveaux suivante L'utilisateur final voit une application plus ou moins interactive, par exemple un jeu, ou un logiciel de traitement de texte. Cette couche est réalisée essentiellement suivant un modèle de calcul figé dans un langage de haut niveau, qui utilise des mécanismes génériques. Par exemple, tous les langages connaissent la notion de conditionnelle, qui requiert sur les processeurs un mécanisme de rupture de séquence (ceci sera expliqué plus loin).

Chaque processeur dispose d'un langage spécifique, son *jeu d'instructions*. Le jeu d'instruction est également appelé langage-machine, ou encore architecture logicielle. De façon précise, le jeu d'instruction est une abstraction programmable (i.e. utilisable pour la programmation) du processeur. Les niveaux suivants sont figés dans le matériel, et ne sont pas reprogrammables. Le jeu d'instruction est donc l'interface de programmation, l'API, du processeur.

Le jeu d'instruction est implémenté par divers circuits logiques : registres, UAL etc. L'organisation et la coordination de ces organes constitue l'architecture matérielle du processeur. De même qu'une API est destinée à recevoir plusieurs implémentations, ou une voiture plusieurs motorisations, un jeu d'instruction correspond en général à plusieurs architectures matérielles différentes. Finalement, les circuits logiques abstraits sont instanciés dans une technologie donnée.

L'interaction entre les niveaux de la hiérarchie n'est pas simplement descendante. Une architecture logicielle est d'une part conçue pour permettre une compilation commode et efficace des langages de haut niveau, mais aussi en fonction des possibilités d'implantation matérielle. L'exemple le plus élémentaire est la taille des mots que le processeur peut

traiter : 8 bits dans les années 70, 32 et plus souvent 64 maintenant.

Le terme d'architecture a été initialement défini comme équivalent à celui de jeu d'instructions. La première "architecture" en ce sens est l'IBM 360, dont le cycle de vie s'est étendu de 64 à 86. De même, l'architecture 8086 est incluse dans toutes les architectures Intel qui lui ont succédé. Auparavant, les niveaux de spécification (jeu d'instruction) et de réalisation n'étaient pas distingués. L'intérêt évident est d'assurer la compatibilité totale sans recompilation des applications. C'est un aspect supplémentaire, et très important, de la fonctionnalité. En particulier, l'architecture logicielle x86 possède de nombreuses caractéristiques (nombre de registres, modes d'adressage) qui expriment les contraintes technologiques des années 70, et ne sont plus du tout adaptées à la technologie actuelle. Cependant, Intel a maintenu la compatibilité binaire jusqu'au Pentium IV, à cause de l'énorme base installée.

Les architectures logicielles et matérielles visent à réaliser un compromis entre des exigences en partie contradictoires.

- Expressivité. Le jeu d'instruction doit fournir des instructions qu'un compilateur peut exploiter commodément.
- Performance. Le jeu d'instruction doit pouvoir être réalisé par une microarchitecture rapide.
- Coût. Le marché est dominé par les ordinateurs personnels ; les ordinateurs de type serveur sont basés sur les mêmes CPU que les ordinateurs personnels. Les contraintes de coût sont donc très fortes.

Les compromis entre ces différentes exigences ont grandement varié au cours des vingt dernières années. Cette activité de recherche-développement a conduit à l'explosion de la capacité informatique que nous connaissons actuellement. L'étude de ces compromis dépasse très largement le cadre de ce cours ; elle correspond aux cours "Architecture" de Licence, Maîtrise et DEA d'informatique. La meilleure référence actuelle est : J. Hennessy, D. Patterson. *Architecture des ordinateurs : une approche quantitative, Deuxième édition*. Thompson Publishing France, 96. Traduction de *Computer Architecture. A Quantitative Approach*. McGrawHill 96.

Plus modestement, nous définirons un jeu d'instructions simple, mais typique des microprocesseurs actuels, l'architecture *logicielle* DIDE. Cette architecture logicielle est décrite complètement dans le chapitre suivant, et spécifiée dans l'annexe. Puis, nous montrerons comment cette architecture *logicielle* peut être implémenté par une *microarchitecture*, DIDE/00.

Cette microarchitecture est réaliste : on peut facilement la spécifier complètement, par exemple avec DIGLOG, donc l'implémenter en circuits. Sa relation avec les microprocesseurs actuels est, par exemple, celle d'une 2CV, avec une Ferrari : une version très simplifiée, mais fondée sur les mêmes principes.

Chapitre 2

Architecture logicielle

L'architecture logicielle est la *spécification* d'un processeur. Elle décrit les unités fonctionnelles (UAL, Unités de calcul flottant etc.), les registres visibles, les types de données manipulés et les opérations le processeur effectue sur ces données. Cette spécification est décrite par le *jeu d'instructions*. C'est la machine que voit un compilateur, ou un programmeur en langage machine.

2.1 Un exemple

On considère l'instruction $A = B + C$, où A , B et C sont des variables entières. Elle est exécutée par une suite de quatre instructions machine : chargement du premier opérande dans un premier registre, chargement du deuxième opérande dans un deuxième registre, addition sur registres, et rangement du résultat en mémoire.

```
LOAD R1, "adresse de B"  
LOAD R2, "adresse de C"  
ADD R3, R1, R2  
STORE R3, "adresse de A"
```

2.2 Langage de haut niveau et langage machine

La fig. 2.1 résume les principales structures des LHN impératifs (Pascal, C ou Fortran). Les caractéristiques des langages plus évolués (fonctionnels, objets etc.) n'étant pas significatives dans le contexte des architectures logicielles, ne seront pas considérées ici.

Les constantes des LHN sont intégrées au programme, on verra ensuite comment. Sinon, on pourrait les modifier par erreur. Le code n'est en principe pas modifiable... Les variables des LHN sont ultimement réalisées par un ensemble de cases mémoire.

-
- Structures de données : Variables et constantes.
 - scalaire : entiers, caractères, flottants etc.
 - tableaux
 - enregistrements
 - pointeurs
 - Instructions
 - Affectation : variable := expression
 - Contrôle interne
 - Séquence : debut Inst1 ; Inst2 ; . . . ; Instn fin
 - Conditionnelle : Si ExpBool alors Inst_vrai sinon Inst_faux
 - Repetition :
 - Tant que ExpBool faire Inst
 - Répéter Inst jusqu'à ExpBool
 - Contrôle externe : appel de fonction
-

FIG. 2.1 – Les composantes des LHN impératifs

2.3 Format des instructions

Le format des instructions est leur codage binaire. Il est absolument spécifique d'une architecture logicielle.

Le format doit décrire l'opération et fournir le moyen d'accéder aux opérandes. L'annexe présente les formats de DIDE. On trouve plusieurs champs : le code opération (CODOP), qui décrit l'opération effectuée par l'instruction (addition, soustraction, chargement, etc.) et des champs qui décrivent les opérandes. On voit comment le format décrit l'architecture logicielle : la largeur des champs numéros de registres est 5 bits ; le compilateur dispose donc de $2^5 = 32$ registres.

On appelle *format fixe* un format où toutes les instructions sont codées sur un mot de taille fixe, de la taille d'un mot mémoire (32 ou 64 bits). L'architecture DIDE utilise un format fixe 32 bits.

On a mentionné ci-dessus "l'instruction ADD R1, R2, R3", et d'autres instructions alors que les instructions sont des mots binaires. En fait, le codage binaire des instructions n'étant pas très pratique pour l'utilisateur humain, les instructions peuvent être décrites par un langage rudimentaire, le langage d'assemblage, ou assembleur. Celui-ci comporte des mnémoniques, qui décrivent l'opération, et une description des opérandes. C'est aussi le langage qu'utilise un programmeur humain. Le langage d'assemblage n'a aucune réalité au niveau de la machine ; la traduction en binaire est réalisée par un utilitaire, *l'assembleur*. La correspondance entre instruction binaire et langage d'assemblage est très élémentaire, du type "traduction mot-à-mot" (contrairement à la compilation). Le codage est si immédiat qu'il est réversible : les débogueurs (dbx, gdb) contiennent des désassembleurs, qui effectuent le codage inverse, du binaire vers le langage d'assemblage. La syntaxe du langage d'assemblage de DIDE est donnée dans l'annexe.

2.4 Type de données

Un type décrit en général, à la fois l'encombrement mémoire et les opérateurs admissibles. la question est donc : quels sont les types connus dans le langage-machine. A ce niveau, les opérateurs sont matériels, donc les types sont peu nombreux : entiers signés et non signés, dont la taille est typiquement 32 bits, demi-mots (16 bits), octets (8 bits), et flottants simple (32 bits) ou double (64 bits) précision. Dans ce cours, on ne considèrera que les entiers signés et non signés.

2.5 Instructions arithmétiques et logiques

Ce sont les instructions d'addition (ADD) et de soustraction (SUB). On trouve également des instructions logiques, qui effectuent bit à bit les opérations logiques Et (AND), Ou (OR), Ou exclusif (XOR), etc. et des instructions de décalage arithmétique ou logique, à droite ou à gauche. Ces dernières instructions effectuent les multiplications et divisions par 2, en signé et non signé. On trouve enfin les instruction CMP et CMPI, qu'on étudiera plus loin.

2.5.1 Opérandes

Un opérande peut être indiqué par un numéro de registre, ou bien être contenu directement dans l'instruction (mode *immédiat*).

Mode registre

Dans ce mode, l'opérande est situé dans un registre général. Par exemple, les trois opérandes de l'instruction

$$\text{ADD Rd, Ra, Rb; Rd} < - \text{Ra} + \text{Rb}$$

sont adressés en mode registre. Plus généralement, dans les instructions en format F1, les deux opérandes sont adressés en mode registre.

Mode immédiat

Dans le mode immédiat, l'opérande est contenu dans l'instruction, par exemple ADD R1, R2, 12. Plus généralement, dans les instructions en format F2, le deuxième opérande est adressé en mode immédiat. Dans la syntaxe assembleur, l'immédiat (12 dans l'exemple) peut être noté en hexadécimal, avec le préfixe 0x, ou en décimal, sans préfixe. Certaines instruction admettent des opérandes signés, qui seront représentés par leur codage en complément à 2 en hexadécimal, et par signe et valeur absolue en décimal. Exemple : ADD R1, R2, -1 ou bien ADD R1, R2, 0xFFFF.

L'immédiat est disponible dans l'instruction, donc dans le registre RI, mais il n'est pas de la taille requise : le format de l'instruction est 32 bits, la même largeur que le chemin de

données, et l'immédiat ne peut donc l'occuper tout entière. L'opérande est obtenu soit en ajoutant des 0 en tête de l'immédiat (instructions ANDI, ORI, XORI), soit par extension de signe (instructions ADDI et CMPI).

En effet, considérons le problème suivant : un entier est codé sur p bits, et on veut récupérer son codage sur n bits, avec $n > p$. Si le codage est du type entier naturel, il suffit de rajouter des 0. Si le codage est du type entiers relatifs en complément à 2, il faut **étendre le signe** : les bits de p à $n - 1$ sont positionnés par le bit $p - 1$.

Le mode d'adressage immédiat permet, entre autres, de compiler les constantes.

```
x := y + 0x12,
```

se compile en

```
...; charger y dans R2
ADD R1, R2, 0x12.
...; ranger R1 dans x
```

Mais, pour compiler

```
x = y + 0x12348765,
```

on ne peut PAS écrire ADD R1, R2, 0x12348765. En effet, l'immédiat est limité à moins que la taille du plus grand entier représentable.

Les deux opérandes source ne peuvent pas être tous deux des immédiats : la taille de l'instruction est trop petite pour en accepter deux. En outre, une telle instruction serait inutile : le compilateur peut calculer directement le résultat d'une affectation du type $x : = \text{cst1} + \text{cst2}$.

2.5.2 Exemple

L'addition et les décalages sont évidemment utilisées pour les compiler les instructions de calcul. Elles sont aussi fortement utilisés pour les calculs d'adresse (fig. 2.2).

2.5.3 Instructions de Test

Pour les instructions de rupture de séquence qu'on verra par la suite, on a besoin d'évaluer un prédicat, c'est à dire une fonction à valeur booléenne. Au niveau du langage-machine, et en se limitant aux types entiers, les prédicats sont calculés à partir du résultat des opérations arithmétiques dans l'UAL. Les prédicats élémentaires sont :

- Zero
- Négatif
- Retenue
- Overflow

```

int v [N], k;
...
temp = v[k];
v[k] = v [k+1];
v[k+1] = temp;

```

Initialement, R1 = @v[0], R2 = k

SLL	R2, 4, R2	; déplacement $k \leftarrow k * 4$
ADD	R2, R2, R1	; $R2 \leftarrow @ v[k]$
LOAD	R3, 0(R2)	; $temp = R3 \leftarrow v[k]$
LOAD	R4, 4(R2)	; $R4 \leftarrow v[k+1]$
LOAD	R4, 0(R2)	; $v[k] \leftarrow R4$
STORE	R3, 4(R2)	; $v[k+1] \leftarrow temp$

FIG. 2.2 – Utilisation des instructions arithmétiques pour le calcul d'adresse

Ces booléens sont rangés dans un registre implicite, le Registre Code Condition (RCC).

On peut souhaiter calculer un prédicat complexe (par exemple l'égalité de deux registres), sans en polluer un autre. Dans l'architecture DIDE, cela est possible en utilisant les instructions CMP et CMPI.

2.6 Instructions d'accès mémoire

On ne traitera ici que les accès à des variables 32 bits.

La taille des registres entiers d'un processeur est fixe (32 bits). Il existe donc des instructions de chargement et de rangement, qui copient un mot de la mémoire vers le registre, ou l'inverse : ce sont les instructions LOAD et STORE.

La principale question est le mode de description de l'adresse, que l'on appelle *mode d'adressage*.

Inclure une adresse mémoire explicite dans une instruction est exclu : l'adresse est de la taille du chemin de données, donc doublerait la taille au moins des instructions mémoire. Les adresses mémoires sont donc calculées comme somme d'un registre et d'un immédiat ou d'un registre. Comme on peut le voir dans l'annexe, l'architecture DIDE n'a qu'un mode d'adressage, le premier. Ce mode est appelé *Base + Déplacement*; l'adresse est la somme d'un registre (*registre de base*) et d'un immédiat, le *déplacement*, positif ou négatif :

$$\text{adresse} = Ra + ES(\text{Imm})$$

Le format des instructions mémoire est identique à ceux des instructions UAL en format F2, car les champs sont identiques.

```

    <calcul de la condition>
    Bcond lvrai
    <code de B2>
    BA lsuite
lvrai: <code de B1>
lsuite: ...

```

FIG. 2.3 – Schéma de compilation d’une conditionnelle

2.7 Instructions de branchement

2.7.1 Définition

Les instructions en langage machine s’exécutent en séquence. Des instructions particulières, les *branchements*, permettent d’implémenter toutes les structures de contrôle interne des LHN, conditionnelles et boucles. L’effet d’un branchement est $PC \leftarrow$ nouvelle valeur.

DIDE n’a qu’une classe d’instructions de branchement, *Bcond* dep. La valeur de PC à laquelle s’ajoute le déplacement est celle de l’instruction qui suit le branchement, et non celle du branchement : PC est incrémenté en même temps que l’instruction est lue. Pour éviter des calculs fastidieux, les programmes en langage d’assemblage utilisent des étiquettes symboliques.

Les instructions *Bcond* n’exécutent la rupture de séquence que si la condition booléenne testée est vraie. Dans le cas contraire, l’exécution des instructions continue en séquence. Les instructions *Bcond* testent les code-conditions contenus dans ce registre, qui est implicite dans l’instruction. Le suffixe *cond* décrit le prédicat testé. La sémantique exacte de ces mnémoniques est fournie par la combinaison des codes conditions qu’ils testent, par exemple E correspond au code condition $Z = 1$, mais leur interprétation est intuitive ; typiquement, la séquence :

```

    CMP R1, R2
    BGT cible

```

branche si $R1 > R2$, lorsque R1 et R2 sont interprétés comme des entiers relatifs.

2.7.2 Compilation des conditionnelles

L’instruction conditionnelle

Si condition alors Bloc B1 sinon Bloc B2

se compile classiquement par le schéma de la fig. 2.3.

La fig. 2.4 donne un exemple de de réalisation des conditionnelles.

CMP R1, R2	; 1 effectue R1 - R2 et positionne RCC
BGT vrai	; 2 Saute les deux instructions suivantes si R1>R2
ADD R3, R0, R2	; 3 Transfere le contenu de R2 dans R3
BA suite	; 4 Saute l'instruction suivante
vrai : ADD R3, R0, R1	; 5 Transfere le contenu de R1 dans R3
suite :	; 6 Suite du programme

FIG. 2.4 – Code de la conditionnelle *Si R1 > R2, alors R3 = R1 sinon R3 = R2* sur une architecture à RCC. Si $R1 > R2$, la suite d'instructions exécutées est 1-2-5-6; sinon 1-2-3-4-6.

Annexe A

L'architecture logicielle DIDE

L'architecture est 32bits, possède 32 registres 32 bits. La mémoire est organisée par octets. Dans la table suivant, ES désigne l'extension de signe, et || la concaténation des chaînes de bits.

A.1 Liste des instructions

- ADD
Format : F1
Syntaxe : ADD rd, ra, rb
Action : $rd \leftarrow ra + rb$, RCC modifié
- ADDI
Format : F2
Syntaxe : ADDI rd, ra, imm₁₆, RCC modifié
Action : $rd \leftarrow ra + ES(imm_{16})$
- SUB
Format : F1
Syntaxe : SUB rd, ra, rb
Action : $rd \leftarrow ra - rb$, RCC modifié
- CMP
Format : F1, le champ Rd n'est pas significatif
Syntaxe : CMP ra, rb
Action : $ra - rb$, RCC modifié
- CMPI
Format : F2, le champ Rd n'est pas significatif
Syntaxe : CMPI ra, imm₁₆
Action : $ra - ES(imm_{16})$, RCC modifié
- AND
Format : F1

- Syntaxe : AND rd, ra, rb
Action : $rd \leftarrow ra \text{ and } rb$
- ANDI Format : F2
Syntaxe : ANDI rd, ra, imm₁₆
Action : $rd \leftarrow ra \text{ and } (0x0000 \parallel imm_{16})$
- XOR
Format : F1
Syntaxe : XOR rd, ra, rb
Action : $rd \leftarrow ra \text{ xor } rb$
- XORI
Format : F2
Syntaxe : XORI rd, ra, imm₁₆
Action : $rd \leftarrow ra \text{ xor } (0x0000 \parallel imm_{16})$
- OR
Format : F1
Syntaxe : OR rd, ra, rb
Action : $rd \leftarrow ra \text{ or } rb$
- ORI
Format : F2
Syntaxe : ORI rd, ra, imm₁₆
Action : $rd \leftarrow ra \text{ or } (0x0000 \parallel imm_{16})$
- LUI
Format : F2, , le champ Ra n'est pas significatif
Syntaxe : LUI rd, imm₁₆
Action : Place imm₁₆ dans les 16 bits de poids fort de rd, et met les 16 bits de poids faible à 0
- SLL
Format : F2, les bits 5 à 15 ne sont pas significatifs
Syntaxe : SLL rd,imm₅, ra
Action : rd reçoit ra décalé à gauche de imm₅ bits. imm₅ est interprété en non signé.
- SAR
Format : F2
Syntaxe : SAR rd,imm₅, ra, les bits 5 à 15 ne sont pas significatifs
Action : rd reçoit ra décalé à droite de imm₅ bits. C'est un décalage arithmétique. imm₅ est interprété en non signé.
- SLR
Format : F2
Syntaxe : SLR rd,imm₅, ra, les bits 5 à 15 ne sont pas significatifs
Action : rd reçoit ra décalé à droite de imm₅ bits. C'est un décalage logique. imm₅ est interprété en non signé.
- LOAD
Format : F2

- Syntaxe : `LOAD rd,imm16(ra)`
Action : $rd \leftarrow \text{Mem32}[ra + ES(\text{imm}_{16})]$
- `STORE`
Format : F2
Syntaxe : `STORE rd, imm16(ra)`
Action : $\text{Mem32}[ra + ES(\text{imm}_{16})] \leftarrow rd$
- `Bcond` Format : F3
Syntaxe : `Bcond imm27`
Action : Si cond vraie $PC \leftarrow PC + ES(\text{imm}_{27} * 4)$. Les conditions sont décrites dans la table A.1.

BE	Egal
BGT	Supérieur, signé
BLE	Inférieur ou égal, signé
BGTU	Supérieur, non signé
BLEU	Inférieur ou égal, non signé
BA	Toujours (branchement inconditionnel)
BC	Retenue (code condition C = 1)
BO	Overflow (code condition O = 1)

TAB. A.1 – Mnémoniques des branchements conditionnels

A.2 Format des instructions

F1

5	1	5	5	5	11
codop	0	Rd	Ra	Rb	0000000000

F2

5	1	5	5	16
codop	1	Rd	Ra	Imm

F3

5	27
codop	Imm

TAB. A.2 – Formats de l'architecture DIDE

Instructions	Codop
arithmétiques et logiques	0xxxx voir table A.4
LOAD	10000
STORE	10001
Bcond	11xxx voir table A.5

TAB. A.3 – Les codes opérations

Instruction	Codop
ADD et ADDI	00000
SUB	00001
CMP et CMPI	00010
AND et ANDI	00100
XOR et XORI	00101
OR et ORI	00110
LUI	00111
SLL	01000
SAR	01001
SLR	01010

TAB. A.4 – Les codes opérations des instruction arithmétiques

Mnémonique	Signification	Codop
BE	Egal	11000
BGT	Supérieur, signé	11001
BLE	Inférieur ou égal, signé	11010
BGTU	Supérieur, non signé	11011
BLEU	Inférieur ou égal, non signé	11100
BA	Toujours (branchement inconditionnel)	11101
BC	Retenue (code condition $C = 1$)	11110
BO	Overflow (code condition $O = 1$)	11111

TAB. A.5 – Les codes opérations des branchements conditionnels