République Algérienne démocratique et Populaire

Ministre de l’Enseignement Supérieure et de la Recherche Scientifique

Université Ziane Achour Djelfa

Faculté : Sciences Exactes et Informatique

Département : Mathématiques et Informatique

Filière : Informatique

Spécialité : Systèmes Informatiques (SI)

Intitulé du cours : Architecture de l’ordinateur

Chapitre 03 : Notions sur les instructions de l’ordinateur

Unité d’enseignement fondamentale : UEF1

Crédits : 5

Coefficient : 3

Enseignant : Dr. RABEHI Ratiba

Cours, TD et TP : Dr. RABEHI Ratiba

Contact : rabehiratiba@yahoo.fr

Chapitre 03 : Notions sur les instructions de l’ordinateur

1- Langage de haut niveau, assembleur, langage machine

Langage machine

Le terme « langage machine » ou « code machine » représente la suite de bits qui est interprétée par le processeur de l’ordinateur lors de l’exécution d’un programme. C’est le langage natif du processeur, et le seul qui soit reconnu par celui-ci.

Les premiers ordinateurs ont été programmés en reliant physiquement par des câbles leurs différents composants et en positionnant manuellement des commutateurs sur les éléments fonctionnels. Suivant la nature et l’ordre de ces liaisons, la machine effectuait les opérations voulues par l’utilisateur.

L’inconvénient majeur de cette technique était que le passage à une autre tâche faisait perdre l’ensemble des connexions installées. C’est la raison pour laquelle Von Neumann a pensé à décrire la notion de programme enregistré comme solution de ce problème.

Le principe de cette technique est ne pas laisser la suite des opérations à effectuer sur la machine comme une composante matérielle de celle-ci mais comme une série d’instructions à lui donner. Ces instructions devaient se mettre sous forme de codes numériques que l’on allait pouvoir laisser dans la mémoire de l’ordinateur où le processeur irait les chercher pour les traiter.

Assembleur

Les premiers programmes étaient écrits directement en code machine, c’est-à-dire que les codes numériques correspondant aux instructions étaient entrés un par un en mémoire par l’utilisateur. Très rapidement, on est passé à la programmation en langage assembleur, dans laquelle des expressions symboliques, appelées « mnémoniques », remplaçaient les codes numériques.

Par exemple, au lieu de saisir le code $(5ef4)_{16}$ en hexadécimal (ou 0101111011110100) en binaire, on écrivait « ADD A,B ». Cette dernière formulation était beaucoup plus compréhensible pour le programmeur.

Mais les ordinateurs ne comprennent que le langage machine, ce qui oblige ce programmeur de faire une traduction du langage d'assemblage au langage machine à l'aide d'un « assembleur ».

Langage de haut niveau ou évolué

Contrairement aux deux autres types de langage vus précédemment, les langages de haut niveau sont totalement indépendants de l'architecture de la machine et du processeur. Par ailleurs, ils offrent un pouvoir d'expression plus riche et plus proche de la pensée humaine, rendant ainsi plus aisée la traduction des algorithmes établis pour résoudre un problème.

Toujours lors de la phase d'exécution, Un programme écrit en langage haut niveau doit être converti vers son équivalent en langage machine. C'est le rôle du traducteur de langage, qui est spécifique à chaque langage évolué utilisé.

Les traducteurs sont divisés en deux catégories : les *compilateur*s et les *interpréteurs*.

- un compilateur traduit une fois pour toutes le langage évolué en langage machine et construit ainsi un programme qualifié de programme objet qui est stocké sur un support de masse tel qu'un disque ;
- un interpréteur lit une à une les instructions du langage évolué, puis il les convertit immédiatement en langage machine avant qu'elles ne soient exécutées au fur et à mesure. Il n'y a pas de génération d'un fichier objet conservant la traduction des instructions en langage évolué vers le langage machine et la traduction doit donc être refaite à chaque nouvelle demande d'exécution du programme.

Editeur de liens

Le compilateur génère ce que l'on appelle un fichier objet, qui n'est pas immédiatement exécutable. Ce fichier est bien constitué d'instructions machine, mais ne constitue pas forcément l'intégralité du programme. Celui-ci est souvent découpé en modules séparés, chacun sous la forme de code source. Une fois chacun d'entre eux compilé (sous forme de fichier objet), il faut les réunir en une seule application : c'est le rôle de l'éditeur de liens.

2- Les instructions

2-1- Cycle d'instruction

L'exécution d'une instruction peut se décomposer en quelques étapes composant le cycle d'instruction :

- récupération de l'instruction et mise à jour de PC (*Fetch 1*) ;
- décodage de l'instruction (*Decode*) ;
- récupération des données (*Fetch 2*) ;
- exécution de l'instruction (*Execute*) ;
- écriture du résultat et modification des bits conditions (*Write*).

Chaque étape est gérée par le séquenceur et transformée en ordres internes.

2-2- Les types des instructions usuelles

Transfert de données

Mnémonique	Opération	Commentaire
MOV rX,rY	$rX \leftarrow rY$	Le registre *rX* reçoit la valeur sur 32 bits contenue dans le registre *rY*.
MVI rX,#i	$rX \leftarrow$ valeur i	Le registre *rX* reçoit sur 32 bits la valeur immédiate *i* indiquée dans l'instruction (MVI, *MoVe Immediate*).
LDB rX,(rY)	$rX \leftarrow Mem[rY]_8$	Le registre *rX* reçoit l'octet mémoire dont l'adresse est dans *rY* (*LoaD Byte*).
LDH rX,(rY)	$rX \leftarrow Mem[rY]_{16}$	Le registre *rX* reçoit les 2 octets mémoire dont l'adresse est dans *rY* (*LoaD Half-word*).
LDW rX,(rY)	$rX \leftarrow Mem[rY]_{32}$	Le registre *rX* reçoit les 4 octets mémoire dont l'adresse est dans *rY* (*LoaD Word*).
STB (rX),rY	$Mem[rX]_8 \leftarrow rY$	L'octet de poids faible de *rY* est stocké en mémoire à l'adresse contenue dans *rX* (*STore Byte*).
STH (rX),rY	$Mem[rX]_{16} \leftarrow rY$	Les 2 octets de poids faible de *rY* sont stockés en mémoire à l'adresse contenue dans *rX* (*STore Half-word*).
STW (rX),rY	$Mem[rX]_{32} \leftarrow rY$	Les 4 octets de *rY* sont stockés en mémoire à l'adresse contenue dans *rX* (*STore Word*).

Opérations arithmétiques et logiques

Mnémonique	Opération	Commentaire
`ADD rX,rY,rZ` `ADD rX,rY,#i`	*rX* ← *rY* + *rZ* *rX* ← *rY* + valeur *i*	Le registre *rX* reçoit la somme sur 32 bits des valeurs contenues dans les registres *rY* et *rZ* (ou la somme de *rY* et d'une valeur immédiate *i*).
`SUB rX,rY,rZ` `SUB rX,rY,#i`	*rX* ← *rY* − *rZ* *rX* ← *rY* − valeur *i*	Le registre *rX* reçoit la différence sur 32 bits des valeurs contenues dans les registres *rY* et *rZ* (ou la différence de *rY* et d'une valeur *i*).
`MUL rX,rY,rZ` `MUL rX,rY,#i`	*rX* ← *rY* × *rZ* *rX* ← *rY* × valeur *i*	Le registre *rX* reçoit sur 32 bits le produit des 16 bits de poids faible des registres *rY* et *rZ* (ou le produit des 16 bits de poids faible de *rY* et d'une valeur *i*).
`DIV rX,rY,rZ` `DIV rX,rY,#i`	*rX* ← *rY*/*rZ* *rX* ← *rY*/valeur *i*	Le registre *rX* reçoit sur 32 bits le quotient de la division entière des registres *rY* et *rZ* (ou le quotient de la division de *rY* et d'une valeur *i*).
`AND rX,rY,rZ` `AND rX,rY,#i`	*rX* ← *rY* ET *rZ* *rX* ← *rY* ET valeur *i*	Le registre *rX* reçoit le ET logique sur 32 bits des valeurs contenues dans les registres *rY* et *rZ* (ou le ET logique de *rY* et d'une valeur *i*).
`OR rX,rY,rZ` `OR rX,rY,#i`	*rX* ← *rY* OU *rZ* *rX* ← *rY* OU valeur *i*	Le registre *rX* reçoit le OU logique sur 32 bits des valeurs contenues dans les registres *rY* et *rZ* (ou le OU logique de *rY* et d'une valeur *i*).
`XOR rX,rY,rZ` `XOR rX,rY,#i`	*rX* ← *rY* ⊕ *rZ* *rX* ← *rY* ⊕ valeur *i*	Le registre *rX* reçoit le OU-exclusif sur 32 bits des valeurs contenues dans les registres *rY* et *rZ* (ou le OU-exclusif de *rY* et d'une valeur *i*).
`NOT rX,rZ` `NOT rX,#i`	*rX* ← NON(*rZ*) *rX* ← NON(valeur *i*)	Le registre *rX* reçoit le complémentaire sur 32 bits de la valeur contenue dans le registre *rZ* (ou le complémentaire d'une valeur *i*).
`NEG rX,rZ` `NEG rX,#i`	*rX* ← −*rZ* *rX* ← −valeur *i*	Le registre *rX* reçoit l'opposé arithmétique sur 32 bits de la valeur contenue dans le registre *rZ* (ou de la valeur *i*), en complément à 2.

Décalage et rotation :

Mnémonique	Opération	Commentaire
`LRT rX,rY,rZ` `LRT rX,rY,#i`	*rX* ← *rY* décalé (par rotation) vers la gauche *rZ* fois (ou *i* fois)	Le registre *rX* reçoit la valeur de *rY* décalée (par rotation) *rZ* fois (ou *i* fois) vers la gauche (LRT, *Left RotaTe*).
`LLS rX,rY,rZ` `LLS rX,rY,#i`	*rX* ← *rY* décalé vers la gauche *rZ* fois (ou *i* fois)	Le registre *rX* reçoit la valeur de *rY* décalée *rZ* fois (ou *i* fois) vers la gauche ; les nouveaux bits sont remplacés par des zéros (LLS, *Logical Left Shift*).
`ALS rX,rY,rZ` `ALS rX,rY,#i`	*rX* ← *rY* décalé vers la gauche *rZ* fois (ou *i* fois)	Le registre *rX* reçoit la valeur de *rY* décalée *rZ* fois (ou *i* fois) vers la gauche ; les nouveaux bits sont remplacés par des zéros lors d'un décalage vers la gauche ou par le bit de poids fort lors d'une décalage à droite (ALS, *Arithmetical Left Shift*).

Sauts :

Mnémonique	Opération	Commentaire
JMP Adr	PC ← *Adr*	Saut inconditionnel à l'adresse *Adr* (JMP, *JuMP*).
JZ rX,Adr	PC ← *Adr* si *rX* = 0	Saut conditionnel à l'adresse *Adr* si le registre *rX* est nul (JZ, *Jump if Zero*).
JNZ rX,Adr	PC ← *Adr* si *rX* ≠ 0	Saut conditionnel à l'adresse *Adr* si le registre *rX* est non nul (JNZ, *Jump if Not Zero*).
JGT rX,Adr	PC ← *Adr* si *rX* > 0	Saut conditionnel à l'adresse *Adr* si le registre *rX* est strictement positif (JGT, *Jump if Greater Than zero*).
JLT rX,Adr	PC ← *Adr* si *rX* < 0	Saut conditionnel à l'adresse *Adr* si le registre *rX* est strictement négatif (JLT, *Jump if Less Than zero*).
JGE rX,Adr	PC ← *Adr* si *rX* ≥ 0	Saut conditionnel à l'adresse *Adr* si le registre *rX* est positif ou nul (JGE, *Jump if Greater or Equal to zero*).
JLE rX,Adr	PC ← *Adr* si *rX* ≤ 0	Saut conditionnel à l'adresse *Adr* si le registre *rX* est négatif ou nul (JLE, *Jump if Less or Equal to zero*).
CCR rX	*rX* ← bit C	Le bit C du registre d'état est mis dans le bit de poids faible de *rX* et ses autres bits sont mis à 0 (CCR, *Copy bit C into Register*).
CZR rX	*rX* ← bit Z	Le bit Z du registre d'état est mis dans le bit de poids faible de *rX* et ses autres bits sont mis à 0 (CZR, *Copy bit Z into Register*).
CNR rX	*rX* ← bit N	Le bit N du registre d'état est mis dans le bit de poids faible de *rX* et ses autres bits sont mis à 0 (CNR, *Copy bit N into Register*).
CVR rX	*rX* ← bit V	Le bit V du registre d'état est mis dans le bit de poids faible de *rX* et ses autres bits sont mis à 0 (CVR, *Copy bit V into Register*).

2-3- Format d'instruction

Une instruction processeur peut être découpée en deux parties :

Code opération : correspond à la mnémonique utilisée par l'assembleur (ADD, MOV, SUB…), qui va se traduire en une valeur numérique dont les bits spécifient quelles actions (transfert de données, calcul…) doivent intervenir à l'intérieur du processeur pour l'exécution de l'instruction. Ces bits du code opération sont utilisés par le séquenceur pour l'envoi des commandes aux différents composants internes du processeur.

Nombre d'opérandes : il s'agit des données sir lesquelles l'instruction porte. Elle doit être spécifiée *via* ***des modes d'adressage*** qui permettent d'indiquer où sont les données (en mémoire, dans un registre, mises explicitement dans l'instruction…).

La plupart des opérations arithmétiques nécessite de spécifier trois données : les deux sur lesquelles portent l'opération (addition, soustraction…) ainsi que l'emplacement où ranger le résultat (qui, sinon, est perdu).

Exemple :

Le concepteur d'un jeu d'instructions peut décider que ADD r1, r2, r3 additionne les valeurs se trouvant dans les registres *r2* et *r3*, et place le résultat dans le registre *r1*. C'est une convention assez classique d'indiquer, comme ici, la destination en premier dans la liste des données d'une instruction.

2-4- Taille de l'instruction :

Les instructions n'ont pas le même nombre de données, par exemple l'addition nécessite 3 données, l'opposition ou le ET logique ne nécessite que 2 données ainsi que le saut a besoin juste d'une seule ! Ça va rendre le taille de l'instruction variable et pose un problème lors de l'exécution du programme où le séquenceur ne peut pas prédire cette taille à l'avance ce qui le rend plus lent.

Une solution adéquate pour ce problème conduit à une simplification radicale du jeu d'instructions de la plupart des sous la forme d'instructions de taille fixe, souvent 32 bits. En rationalisant l'expression des données des instructions, on a simplifié le séquenceur et donc permis son optimisation.

3- Modes d'adressage

Le mode d'adressage représente la technique de localisation de l'opérande. Il y a plusieurs types d'adressage comme :

3-1- L'adressage immédiat

C'est le mode le plus facile, dans ce cas on déclare directement la valeur de l'opérande dans l'instruction et pas son adresse (utilisé pour les constantes).

Ex **: MOV R1 , #6 (chargement de la valeur 6 dans le registre R1)**

Il est nécessaire de noter qu'il faut ajouter le caractère # avant la valeur constante pour indiquer qu'il ne s'agit pas d'une adresse.

3-2- L'adressage direct

Dans ce mode, on écrit l'adresse effective dans le champ adresse dans l'instruction.

Ex **: MOV R1 , adr1 (chargement de la valeur contenu dans l'adresse adr1 dans le registre R1)**

Malheureusement, certains processeurs (plus précisément l'UAL) ne peuvent être alimentés que par des registres, c'est pourquoi on ne peut pas utiliser ce mode d'adressage dans les instructions arithmétiques, mais seulement pour des instructions de transfert mémoire vers un registre, ou l'inverse.

3-3- L'adressage par registre

Il est semblable au mode précédent. La seule différence entre eux c'est que l'adresse de l'opérande réfère à registre (pas un mot mémoire).

Ex **: ADD r1, r2, r3 (additionnez les contenus des registres r3 et r2 et mettez le résultat dans r1).**

3-4- L'adressage indirecte par registre

Au lieu d'utiliser directement une adresse mémoire dans l'instruction, on peut indiquer où trouver cette adresse : dans un registre. Alors l'opérande se situe dans la mémoire mais dans l'instruction, on trouve seulement le registre portant son adresse.

Ex **: MOV r1 , @r2 (chargement –dans le registre r1- la valeur contenu dans l'adresse cachée dans le registre r2)**

3-5- L'adressage indirect avec déplacement

Ce mode est une combinaison du mode d'adressage indirect avec un décalage. L'adresse de la donnée est la somme de l'adresse contenue dans un registre (mode indirect) et d'un déplacement immédiat indiqué dans l'instruction.

Ex **: MOV r1 , (8)r2 (met dans *r1* la valeur dont l'adresse mémoire est obtenue en ajoutant 8 à celle contenue dans *r2*)**

3-6- L'adressage indirect indexé

Ce mode d'adressage résout le problème de l'adressage indirect, où la valeur du déplacement est statique et indiquée d'une façon dure dans l'instruction. Alors dans l'adressage indirect indexé, l'adresse mémoire de la donnée est la somme du contenu d'un registre de base (mode indirect) et d'un registre jouant le rôle d'un déplacement.

Ex **: MOV r4, (r5, r6) (met dans *r4* la valeur dont l'adresse mémoire est obtenue en ajoutant la valeur contenue dans *r5* à celle contenue dans *r6*)**

4- Pipeline

Chaque instruction peut être décomposée en plusieurs étapes (la lecture de l'instruction, son décodage, son exécution, etc.), nécessitant chacune des circuits différents. Ces derniers travaillent pendant une partie du temps d'exécution, mais restent inactifs lorsque l'instruction n'est pas à l'étape concernée.

Le principe du traitement pipeline consiste à ne pas attendre la fin de l'exécution d'une instruction pour lancer la suivante, en profitant de l'inactivité des circuits ayant déjà traité l'instruction en cours. C'est l'idée du travail à la chaîne : un circuit effectue une étape d'exécution et enchaîne immédiatement la même étape avec l'instruction suivante pendant que la première avance dans la chaîne de traitement. La figure suivante illustre ce concept en décomposant une instruction en cinq étapes. Chaque chiffre correspond à une instruction avançant dans le pipeline et dont l'exécution se termine à sa sortie.

Le gain ne se fait pas directement sur la vitesse de traitement d'une instruction (elle doit toujours passer par les cinq étapes du pipeline) mais sur le nombre d'instructions par seconde qui sont exécutées. Dans le meilleur des cas, le processeur peut traiter cinq fois plus d'instructions qu'un système non pipeliné.

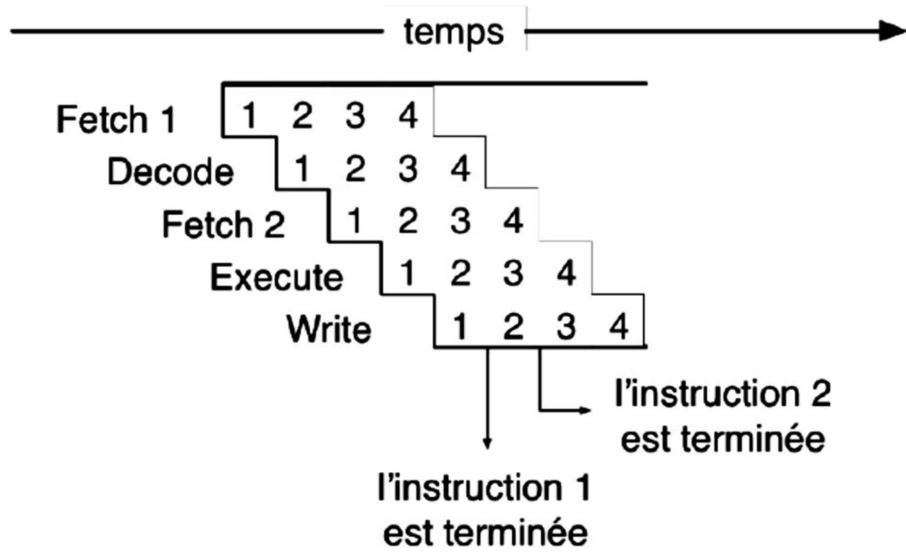

Fig.1 : Traitement Pipeline.