Architecture et Système

Stefan Schwoon

Cours L3, 2022/2023, ENS Cachan

Interruptions

Des périphériques (clavier, réseau) n'ont pas toujours besoin de communiquer avec le processeur, mais si besoin, il faut réagir vite.

Quand un périphérique souhaite échanger des données avec le processeur, il met son signal d'interruption à 1.

Quand le processeur est prêt pour la prochaine instruction, il vérifie s'il y a reçu un tel signal d'interruption et choisit celui avec la priorité la plus élevée.

Dans ce cas le processeur continue son travail à une certaine adresse en fonction du signal choisi. Cette adresse devrait contenir du code pour gérer la communication. Ensuite on continue l'exécution normale.

Architecture: remarques

Notre petit ordinateur (voir documentation) comporte quelques simplifications et faiblesses importantes qui empêcheraient sa mise en pratique dans un ordinateur réel.

Notamment, la profondeur est trop importante :

obtenir les signaux de contrôle (17)

obtenir la valeur sur le bus (+42), dont +33 pour la RAM

calcul de l'ALU (+29)

$\rightarrow$ circuit sous-utilisé et inefficace

Cycles

Dans un premier temps, on sous-divise le circuit en plusieurs parties indépendants qui achèvent des tâches bien précises et moins complexes :

obtenir le prochain code d'instruction (IF = instruction fetch)

obtenir les signaux de contrôle (ID = instruction decode)

exécuter, p.ex. faire un calcul dans l'ALU (EX)

lire/écrire dans la mémoire (MEM)

$\rightarrow$ instructions coupées en plusieurs cycles de profondeur réduite

$\rightarrow$ permet d'adapter le temps d'exécution à la complexité de l'instruction

D’autres faiblesses

Dans notre architecture primitive, tout transfert de données passe par le CPU.

CPU surchargé de travail

la vitesse de l’ordinateur dépend du composant le plus lent

$\rightarrow$ diviser le travail, introduire des sous-composants chargés d’organiser le transfert de données à une vitesse propre

Notre architecture traite registres et mémoire avec la même vitesse.

pas réaliste, un accès mémoire peut être beaucoup plus lent qu’un accès registre

$\rightarrow$ introduire cycles d’attente

$\rightarrow$ gestion mémoire (plus tard)

Développement historique

Les premiers ordinateurs (années 50) :

- peu d'instructions et signaux $\rightarrow$ architecture câblée

Les années 60-80 : âge de la microprogramation

- jeux d'instructions et signaux toujours plus complexe :
  microprogrammation plus facile à construire et gerer

Exemple: Jeu d'instructions Intel x86

- instructions complexes (p.ex. recopier une chaîne d'octets)
- opérandes de 8/16/32 bits (pour compatibilité en arrière)
- registres partiels ou complets (AL/AH, AX, EAX)
- longueur d'instructions variable (1 à 7 octets), décodage compliqué

L'architecture RISC

RISC = reduced instruction set computing (à partir des années 80)

Facteurs technologiques :

miniaturisation rend possible des circuits plus complexes

outils pour automatiquement concevoir et arranger des circuits (CAD)
$\rightarrow$ construction des circuits complexes devient plus facile

apparition de l'architecture RISC qui permet des optimisations

Idée en général : uniformiser les instructions pour exécution plus efficace

simplifier le *décodage* : toute code d'instruction a la même longeur,
opérands toujours dans la même place de l'opcode.

pas d'opérations complexes, toute instruction s'exécute en un seul cycle

Architecture RISC

D’autres caracteristiques typiques :

- mémoire rapide integrée dans le processeur (ou proche)
- pipelining (exécution parallèle) : le processeur s’occupe de plusieurs instructions à la fois : une instruction en phase “instruction fetch”, une en phase “decodage”, encore une en “exécution”
- plus de registres pour minimiser les transferts vers la mémoire

Inconveniences:

- Incompatible avec des architectures existantes (notamment x86)
- Plus de travail pour les compilateurs
- Mots d’instructions très grands, code machine peu compact

Usage dans les architectures modernes

Combinaison des deux architectures:

- (pré-)processeur qui traduit les opérations complexes vers instructions RISC
- derrière les scènes, une autre couche opère sur ces instructions RISC

Plusieurs unités d'exécution travaillant en parallèle (superscalaire). Problèmes:

- dépendances : le résultat d'une instruction porte sur la suivante
- branchements : quelle instruction sera la suivante ?

Solutions :

- analyse des dépendences, exécution "out of order"
- exécution spéculative (sur un autre jeu de registres), prédiction des branches

Représentation de données

Entiers

Les entiers (tels qu'on les utilise dans les langages de programmation habituels) sont stockés dans un mot de taille fixe (typiquement 8, 16, 32 bits).

Types en C: `char`, `short int`, `int`, `long int`, `long long int`

En C, les tailles de ces types ne sont pas précisement définis par le langage de C, seulement leurs tailles minmales : 8, 16, 16, 32, 32, où `int` est un mot de registre.

On peut obtenir les valeurs concrètes avec `sizeof(char)` etc.

Big vs little endian

Un mot de taille > 8 s'étend sur plusieurs octets, p.ex. avec 32 bits:

12	34	56	78

Sous l'interprétation big-endian, ceci représente $(12345678)_{16}$ (les bits les plus significatifs sont dans le premier octet).

Little-endian: c'est l'inverse, on interpréte cela comme $(78563412)_{16}$.

Le mode d'interprétation devient important lors des échanges des données binaires (fichiers, réseau). P.ex., l'Internet protocol (IP) définit cet ordre comme big-endian.

Fonctions en C : `ntohl`, `ntohs`, `htonl`, `htons`

Entiers avec/sans signe

Un mot de n bits peut représenter 2^n valeurs différentes. Les opérations arithmétiques travaillent implicitement modulo 2^n.

Deux interprétations :

`unsigned` (sans signe), le domaine est de 0 à $2^n - 1$.

`signed` (avec signe) / complément à deux : le domaine est de -2^{n-1} à $2^{n-1} - 1$

Pour les valeurs non-négatives, le bit le plus significatif (MSB) est de 0.
Pour les valeurs négatives, le MSB est de 1.

On obtient la représentation de $-i$ en prenant la négation (bit par bit) de i, puis en rajoutant 1. P.ex., $-1 \mathrel{\widehat{=}} 11\cdots1$ et $-2^{n-1} \mathrel{\widehat{=}} 10\cdots0$.

En C

Les types entiers peuvent être déclaré comme `signed` ou `unsigned` ;
par défaut ils sont `signed`.

Pour des `char` cette distinction n'est pas important lorsqu'on les interprète comme des caractères.

Attention aux opérateurs de décalage (<< et >>) :

pour les `unsigned`, l'opération est dite *logique*, le décalage se fait sur l'intégralité du mot, en inclus le bit le plus significatif;

pour les `signed`, l'opération est dite *arithmétique*, elle conserve le signe.

Valeurs réelles

Les valeurs réelles sont typiquement représentées dans un format virgule flottante, dans un mot de taille fixe, avec une précision limitée.

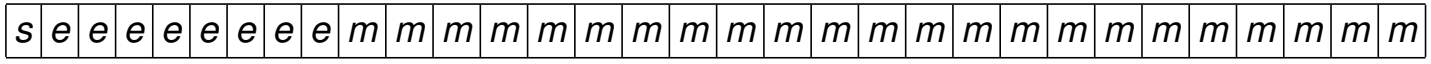
Idée en général : tuple $\langle s, m, e \rangle$ avec l'interprétation $\pm 2^e \cdot m$.

s est le signe (un seul bit, `0` non-négatif, `1` négatif);

m est la mantisse;

e est l'exposant.

$\rightarrow$ compromis entre taille et précision des valeurs représentables.



Besoin des standards

Problèmes:

- Comment repartir les bits entre taille et mantisse ?
- Représentations non uniques : $\langle s, m, e\rangle \equiv \langle s, 2m, e-1\rangle$
- Comment traiter des cas spéciaux (division par zéro), comment traiter les arrondis ?

Le standard le plus important pour règler ces questions s'appele IEEE 754.

En C : `float` en C = IEEE 754, 32 bit; `double` = IEEE 754 (64-bit).

Dans le suivant, on discutera la partie 32 bit, les autres parties étant similaires.

IEEE 754 (variante 32-bit)

IEEE 754 spécifie les conventions suivantes :

Signe : 1 bit; 0 pour positif, 1 pour négatif

Exposant : 8 bits, interprété sans signe, mais décalé de 127. P.ex., $(10000001)_2 = 129$, mais l'exposant effectif est de 2. Si tous les bits de l'exposant sont 1, voir ci-dessous.

Mantisse : 23 bits, interprété comme $1 + (m/2^{23})$, ce qui donne un résultat dans $[1, 2)$.

Remarques :

Cette interprétation de la mantisse garantit une représentation unique.

Cas spéciaux, si exposant est 1111 1111 : $\pm\infty$ (avec $m = 0$)
ou NaN (not a number, avec $m \neq 0$)

Addition dans les flottantes

Le standard IEEE définit la procédure à suivre pour effectuer des opérations arithmétiques (comment arrondir, comment traiter des cas spéciaux, . . .).

Discutons le cas d'une addition:

Soient $x = 2^{e_x} \cdot m_x$ et $y = 2^{e_y} \cdot m_y$ et $x > y$
(par simplicité on suppose qu'ils sont tous les deux positifs).

P.ex. $x = 2.5$ et $y = 0.75$, du coup,
$e_x = 1, m_x = 1.25, e_y = -1, m_y = 1.5$.

Représentation binaire :

0	1000 0000	0100 0000 0 · · · 0
0	0111 1110	1000 0000 0 · · · 0

On isole désormais la mantisse, mais en prenant compte de la “une cachée” :

$$\boxed{\begin{array}{l} 1\ 0100\ 0000\ 0\cdots\ 0 \\ 1\ 1000\ 0000\ 0\cdots\ 0 \end{array}}$$

L’exposant d’y est moins grand que celui de x de 2.
Du coup, on décale la mantisse d’y de 2 positions.

$$\boxed{\begin{array}{l} 1\ \ 0100\ \ 0000\ 0\cdots\ 0 \\ 0\ 0\ 110\ 0000\ 0\cdots\ 0 \end{array}}$$

L’addition des deux mantisses donne désormais :

$$\boxed{1\ 1010\ 0000\ 0\cdots\ 0}$$

Comme le résultat n'excède pas les 24 bits (une cachée plus 23 bits), on garde l'exposant de x et on enlève simplement l'une cachée de i_Z pour obtenir la mantisse du résultat.

0	1000 0000	1010 0000 0$\cdots$0

Si l'addition des mantisses avait débordé les 24 bits, il aurait d'abord fallu décaler i_Z à droit par une position (perdant un bit de précision).

Problèmes de virgule flottante

Précision limitée : certaines valeurs “rondes” comme 0.1 ou 2.3 ne sont pas représentables.

Erreurs d’arrondi; p.ex. $x + y$ donne x si x beaucoup plus grand qu’y.

Du coup, certaines lois comme distributivité ne sont plus valables.