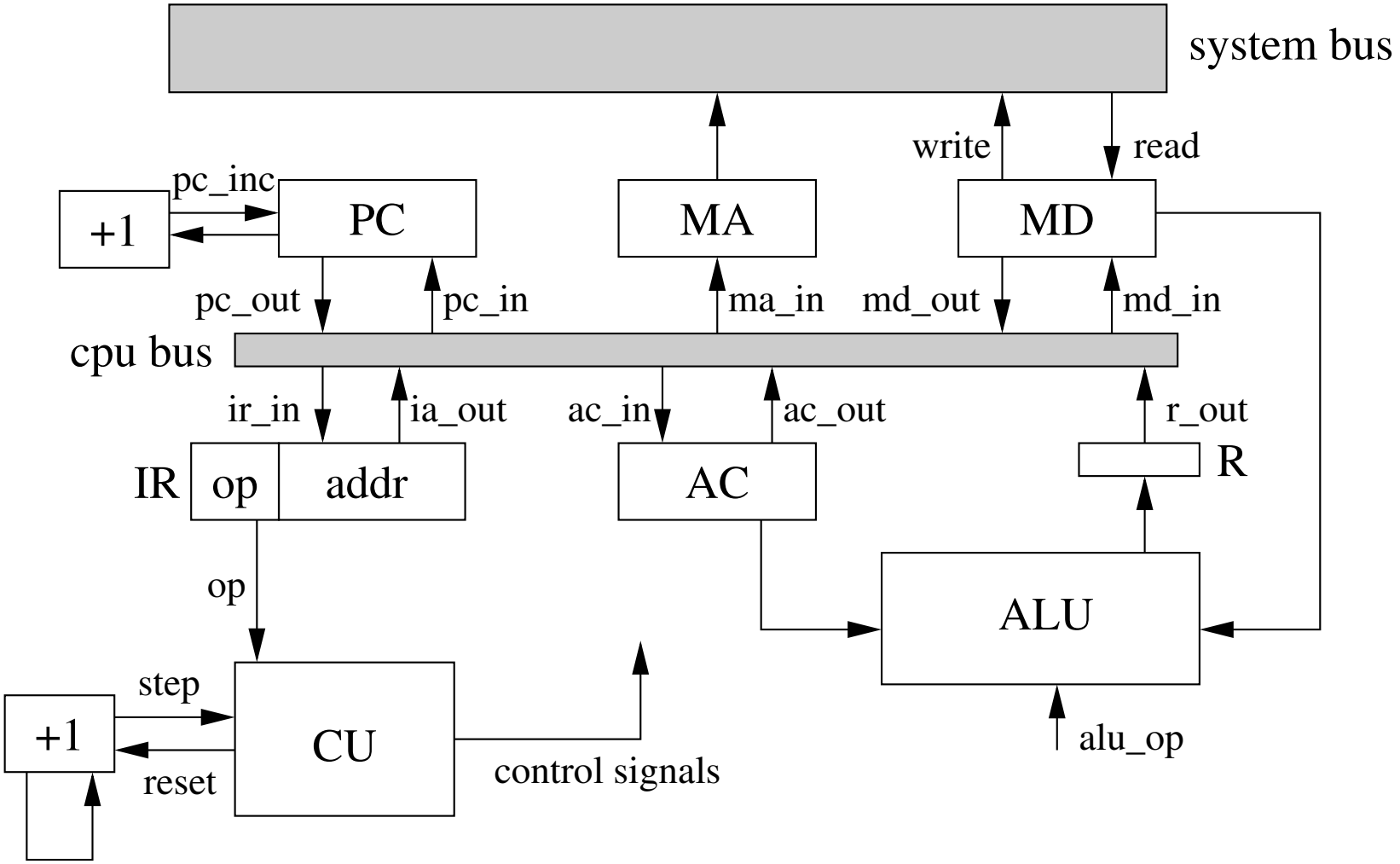
Architecture et Système

Stefan Schwoon

Cours L3, 2015/16, ENS Cachan

Rappel : architecture simple



Control signals

Comment le processeur fait-il pour fournir les bons signaux de contrôle ?

Microprogrammation :

Utiliser op et phase comme obtenir une adresse dans un ROM qui fournit les valeurs pour tous les signaux.

Câblé:

Construire un circuit logique pour tout signal qui prend en compte op et phase.

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
- une même architecture peut être adatpé à des différents besoins
- émulateurs pour être compatible avec d'anciennes machines
- microprogrammation utilisateur sur certaines machines

Exemple: Jeu d'instructions Intel x86

Jeu d'instructions complexe, avec quelques instructions assez puissants (boucles)

Opérandes de 8/16/32 bits

Instructions peuvent utiliser un registre partiel ou complet (AL/AH, AX, EAX)

Longueur d'instructions variable (1 à 7 octets), rend nécéssaire une *phase de décodage*

L'architecture RISC

A partir des annes 80 : retour au mode câblé

Facteurs technologiques :

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

miniaturisation rend possible des circuits plus complexes

apparition de l'architecture RISC qui permet des optimisations

RISC = reduced instruction set computing
(veut dire : réduire le temps pour exécuter instruction)

Architecture RISC

Idée en général : uniformiser les instructions afin de les exécuter plus efficacement.

Caracteristiques typiques :

éliminer des opérations complexes (load/store)

mémoire rapide integrée dans le processeur (ou proche)

éliminer la phase *décodage* : codes opération toujours d'une même longeur, opérands toujours dans la même place de l'opcode.

exécution parallèle : exécuter plusieurs instructions à la fois : une instruction en phase IF, une autre en EX

plus de registres

Inconveniences de RISC

Incompatible avec des architectures existantes (notamment x86)

Plus de travail pour les compilateurs

Mots d'instructions plus grand, peu compact

$\rightarrow$ combinaison des deux techniques:

(pré-)processeur traduit opérations complexes vers (plusieurs) instructions RISC

une autre couche opère sur ces instructions RISC

Parallelisme dans les architectures modernes

– Le processeur exécute plusieurs instructions en parallèle, en profitant des différentes phases d'exécution (pipelining).

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

En principe, cela permet d'exécuter plus d'instructions dans une même temps. Mais il y a des problèmes :

dépendances : le résultat d'une instruction est nécéssaire pour le prochain

branchements : quelle instruction sera la prochaine ?

Solutions :

analyse des dépendences, exécution "out of order"

exécution spéculative (sur un autre jeu de registres), prédiction des branchements

Autres approches au parallelisme

VLIW = very long instruction word

EPIC = explicitly parallel instruction computing

Un processeur obtient un grand mot qui contient plusieurs instructions qui seront par la suite exécute en mode *pipeline*.

C'est le compilateur qui doit compiler ces mots, tout en respectant les dépendances.

Représentation des données

Quelquels notions de base :

bit: valeur 0 ou 1.

byte: unité la plus petite qui peut être adressé par le programmeur dans la mémoire (typiquement 8 bit = octet).

mot: vecteur de bits traité comme une seule unité.
p.ex. *mot de registre*, *mot d'instruction*, etc.
Sans qualificatif, on parle d'un mot stocké dans un registre.

Entiers

Stockés dans un mot de taille fixe (typiquement 8, 16, 32 bits)

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

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 nécéssite plusieurs bytes en mémoire.

Big-endian: on stocke le mot le plus significatif dans le premier byte
p.ex. $(12345678)_{16}$ est stocké dans quatre bytes dans l'ordrer 12, 34, 56, 78 (hexadécimal)

Little-endian: c'est l'inverse, on stocke les bytes dans l'ordre 78, 56, 34, 12

Les deux formats sont utilisés en pratique (p.ex., little-endian sur les ordinateurs Intel ou compatible)

Les raisons en sont pour la plupart historiques :

pro little-endian: un mot peut être interpreté modulo 2^8, 2^{16} etc en utilisant une même adresse, certaines opérations arithmétiques étaient (historiquement) plus facile (p.ex. addition en traitant un octet à la fois).

pro big-endian: division modulo 2^8 etc en utilisant une même adresse

Attention !

Le mode de stockage devient important quand on échange des données binaires (fichiers, réseau).

Dans ces cas, l'ordre doit être spécifié par le protocol / format de fichier.

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 octets peut être stocke les valeurs $0 \ldots 2^n - 1$.
On l'appele un entire sans signe (*unsigned*). Les opération arithmétiques travaillent implicitement modulo 2^n.

Un entier avec signe (*signed*) est typiquement stocké en format complément à deux.
Ici, un mot stocke des valeurs $-2^{n-1} \ldots 2^{n-1} - 1$.

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

Pour les valeurs négatives, le MSB est de 1: représentation de $-i$ (pour $i \geq 0$) obtenue en soustrayant 1, puis en prenant la négation (bit par bit) du résultat.
P.ex., -1 représenté par $11 \cdots 1$, -2^{n-1} par $10 \cdots 0$.

Du coup, l'addition de i and $-i$ en utilisant l'addition *sans signe* donne 0, le bon résultat.

Conclusion : sur le niveaux binaire, les opérations arithmétiques peuvent travailler comme pour les valeurs sans signe.

Sans/avec signe devient simplement une façon d'interpréter un entier !

Valeurs réelles

Valeurs réelles représentées typiquement par virgule flottante, stockées dans des mots de taille fixe.

Du coup : précision limité

Idée en général : tuple (s, m, e) 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.

Besoin des standards

Problèmes:

taille d'exposant, mantisse?

représentations pas uniques : $(s, m, e) \equiv (s, 2m, e - 1)$

Comment traiter des cas spéciaux (division par zéro), comment traiter les arrondis ?

Le standard le plus important s'appele IEEE 754. Ici, on ne discutera que la partie 32 bit. (`float` en C, `double` = 64-bit).

IEEE 754 (variante 32-bit)

IEEE 754 spécifie les conventions suivantes :

1 bit pour le signe, 8 pour l'exposant, 23 pour la mantisse;

Interpretation d'exposant: $e_u - 127$, où e_u est l'interprétation *sans signe* des 8 bits. Du coup, on représente ± 127. La valeur 128 est reservé pour quelques cas spéciaux.

Interprétation de la mantisse: $1 + (m_u/2^{23})$, ce qui donne $[1, 2)$.

Remarques :

Interprétation de la mantisse force une représentation unique.

$e = 128$ pour $\pm\infty$ (avec $m_u = 0$)
ou NaN (not a number, avec $m_u \neq 0$)