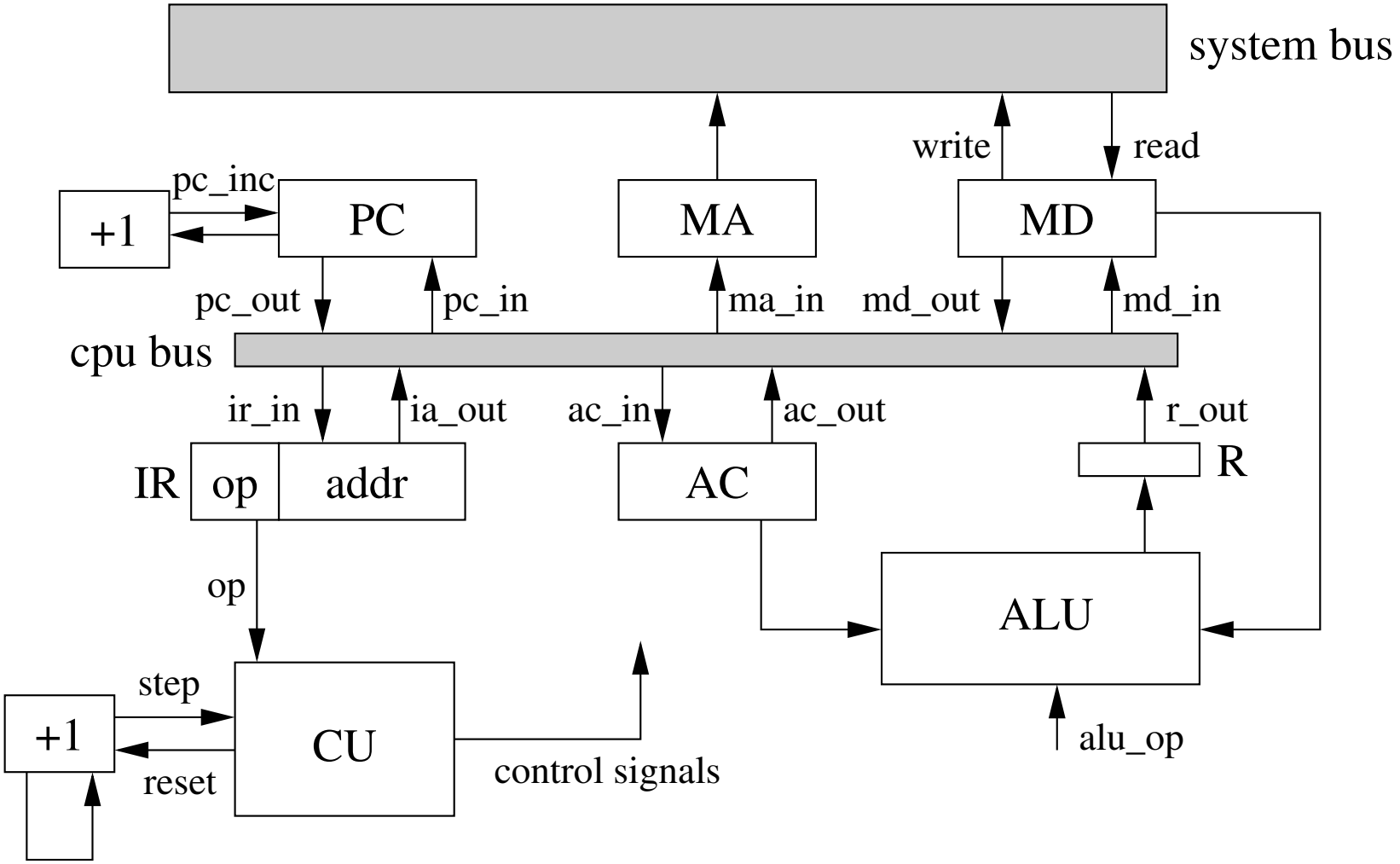
Architecture et Système

Stefan Schwoon

Cours L3, 2016/17, ENS Cachan

Rappel : architecture simple



Control signals

Les transferts de données se font sur un bus (multiplexeur/décodeur).

On appelle signaux de contrôle ceux qui sélectionnent parmi les entrées et sorties.

Comment 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 de contrôle.

Câblé: Construire un grand circuit spécialisé 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 adapté aux besoins différents

microprogrammation utilisateur sur certaines machines

Exemple: Jeu d'instructions Intel x86

Jeu d'instructions complexe, avec quelques instructions assez puissants (boucles pour traiter des blocs de mémoire)

Opérandes de 8/16/32 bits (pour compatibilité en arrière)

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 :

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

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, juste load/store/op.arithmétiques

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 "instruction fetch", une autre en "exécution"

plus de registres pour minimiser les transferts vers la mémoire

Inconveniences de RISC

Incompatible avec des architectures existantes (notamment x86)

Plus de travail pour les compilateurs

Mots d'instructions très grands, code machine 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 nécessaire 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

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 octets en mémoire.

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

Little-endian: c'est l'inverse, on stocke les octets 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`