Ressource R2.04
Communication et fonctionnement bas niveau

Cours n°1
Organisation d'un ordinateur
Exécution d'un programme

---

version du 21 janvier 2024

D’après le programme natiional du BUT INFO

Descriptif

L’objectif de cette ressource est de comprendre le fonctionnement des couches systèmes et réseaux bas niveau. Cette ressource permet de découvrir les multiples technologies et fonctions mises en œuvre dans un réseau informatique et de comprendre les rôles et structures des mécanismes bas niveau mis en œuvre pour leur fonctionnement.

Savoirs de référence étudiés

- Étude d’un système à microprocesseur ou microcontrôleur avec ses composants (mémoires, interfaces, périphériques...)
- Langages de programmation de bas niveau et mécanismes de bas niveau d’un système informatique
- Étude d’architectures de réseaux et notion de pile protocolaire
- Technologie des réseaux locaux : Ethernet, wifi, TCP/IP, routage, commutation, adressage, transport

Mots clés

Protocoles – Pointeurs – Interruptions – Langage bas niveau

D’après le programme natiional du BUT INFO

Descriptif

L’objectif de cette ressource est de comprendre le fonctionnement des couches systèmes et réseaux bas niveau. Cette ressource permet de découvrir les multiples technologies et fonctions mises en œuvre dans un réseau informatique et de comprendre les rôles et structures des mécanismes bas niveau mis en œuvre pour leur fonctionnement.

Savoirs de référence étudiés

- Étude d’un système à microprocesseur ou microcontrôleur avec ses composants (mémoires, interfaces, périphériques...)
- Langages de programmation de bas niveau et mécanismes de bas niveau d’un système informatique
- Étude d’architectures de réseaux et notion de pile protocolaire
- Technologie des réseaux locaux : Ethernet, wifi, TCP/IP, routage, commutation, adressage, transport

Mots clés

Protocoles – Pointeurs – Interruptions – Langage bas niveau

Plan du cours

Plan du cours

Exemple

```go
package main

import "fmt"

func main() {
  var hello = "Hello, BUT"
  var helloAsRuneArray = []rune(hello)
  for i := 0; i < len(helloAsRuneArray); i++ {
    fmt.Print(string(helloAsRuneArray[i]), ".")
  }
  fmt.Println()
}
```

On souhaite exécuter ce programme. Quelles étapes doit-on suivre ?

La commande `go build` enchaîne plusieurs étapes :

1. Compilation (langage de haut niveau → code objet)
    1.1 Analyse lexicale
    1.2 Analyse syntaxique
    1.3 Analyse sémantique
    1.4 Traduction en code intermédiaire
    1.5 Optimisations
    1.6 Génération du code objet et assemblage
2. Édition des liens (codes objets → exécutable)
    2.1 Résolution des liens
    2.2 Optimisations
    2.3 Génération du code exécutable

Quelques commandes pour voir ce qui se passe :

Afficher toutes les commandes enchaînées

```
go build -x -work
```

Obtenir le code avant assemblage

```
go build -gcflags="-S" 2> main.S
```

Désassembler le code exécutable

```
objdump -d hello > hello.d
```

→ Dans le code désassemblé, en plus du code correspondant à la traduction du code source, on trouve :

- du code spécifique au système d'exploitation,
- du code venant du *runtime* go.

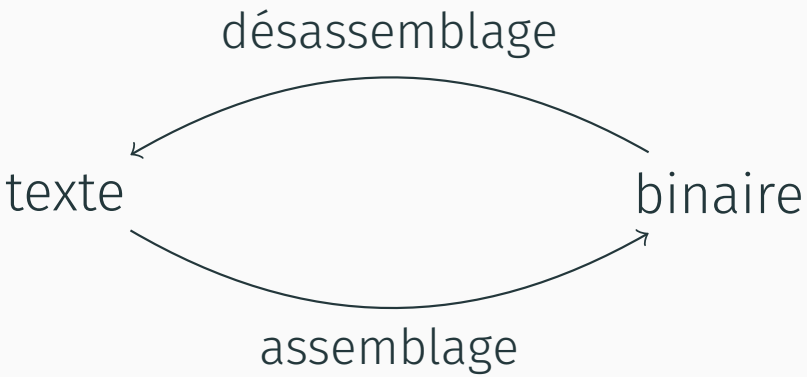
Langage machine (1)

Un exécutable contient un programme exprimé en *langage machine*.

C'est un langage de bas niveau, très proche du matériel.

Un programme en langage machine peut être exprimé de deux façons :

- en binaire – représentation utilisée en mémoire
- en texte – représentation utilisée pour écrire / déboguer le code machine



Langage machine (2)

Le langage machine est très différents des langages de haut niveau :

- Il y a beaucoup d'instructions, construite sur le même schéma.
  - un opcode : identifie l'opération
  - des opérandes sources et cibles : registre, adresse mémoire, ou constante entière
- Le nombre, les noms, et la taille des variables sont connues à l'avance :
  - les registres ;
  - la mémoire de travail = un tableau d'octets, qui contient le code et les variables.
- Il n'y a pas de type, ou plutôt il n'y a qu'un type : les entiers de taille fixe.

Langage machine (3)

Les opérations relèvent très majoritairement des catégories suivantes :

- accès à la mémoire de travail : copie de données entre la mémoire de travail et les registres ;
- instruction arithmétique ou logique : opération sur des données présentes dans les registres, des constantes ;
- branchement (ou saut) : détournement du flot de contrôle normal, utilisés pour les appels de procédures, les conditionnels, les boucles.

Il existe d'autres opérations permettant d'interagir avec les périphériques ou de contrôler l'état de la machine.

Langage machine (4)

Le langage machine est également appelé jeu d'instruction (*instruction set architecture* en anglais, abbr. ISA).

On distingue habituellement deux catégories d'ISA :

Complex Instruction Set Computer (CISC)
Beaucoup d'opérations, dont certaines complexes, pouvant relever de plusieurs catégories. Code compact, parfois difficile à optimiser. Implémentation complexe mais permet d'optimiser fortement certaines instructions.

Reduced Instruction Set Computer (RISC)
Peu d'opérations, chacune relevant d'une unique catégorie. Code plus verbeux, mais plus simple à optimiser. Implémentation simple et souvent efficace sur le plan énergétique.

Langage machine (5)

```
func add(a, b int) (res int) {
    res = a + b
    return
}
```

```
push   %rbp
mov    %rsp,%rbp
sub    $0x8,%rsp
mov    %rax,0x18(%rsp)
mov    %rbx,0x20(%rsp)
movq   $0x0,(%rsp)
add    %rbx,%rax
mov    %rax,(%rsp)
add    $0x8,%rsp
pop    %rbp
ret
```

Remarque : quand on désactive les optimisations, le code produit par le compilateur comporte des instructions « inutiles ».

Langage machine (6)

```
func add(a, b int) (res int) {
    res = a + b
    return
}
```

```
add     %rbx,%rax
ret
```

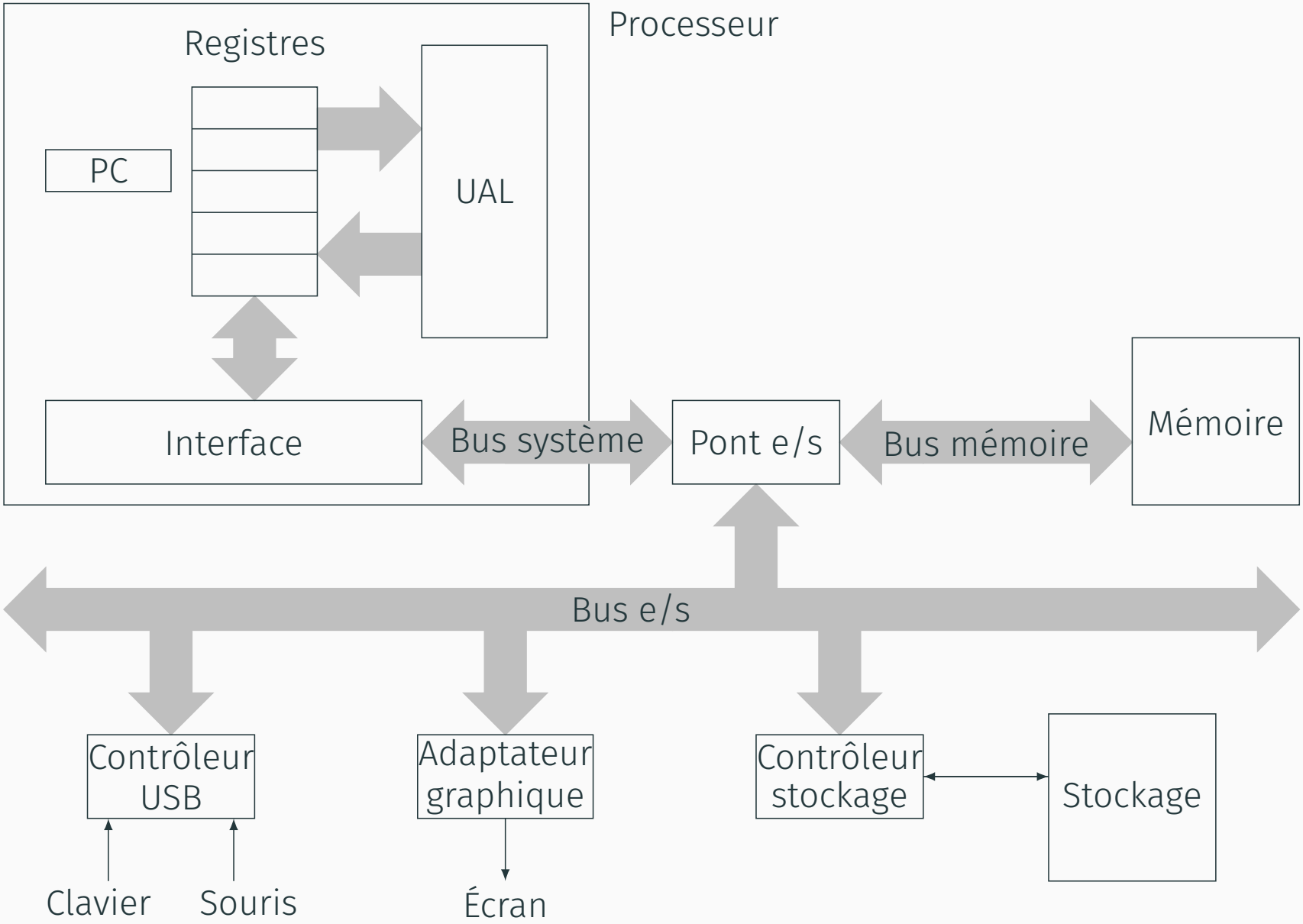
Remarque : dans le code optimisé, la fonction n'est même pas appelée (le compilateur pré-calcule le résultat et le fait directement afficher).

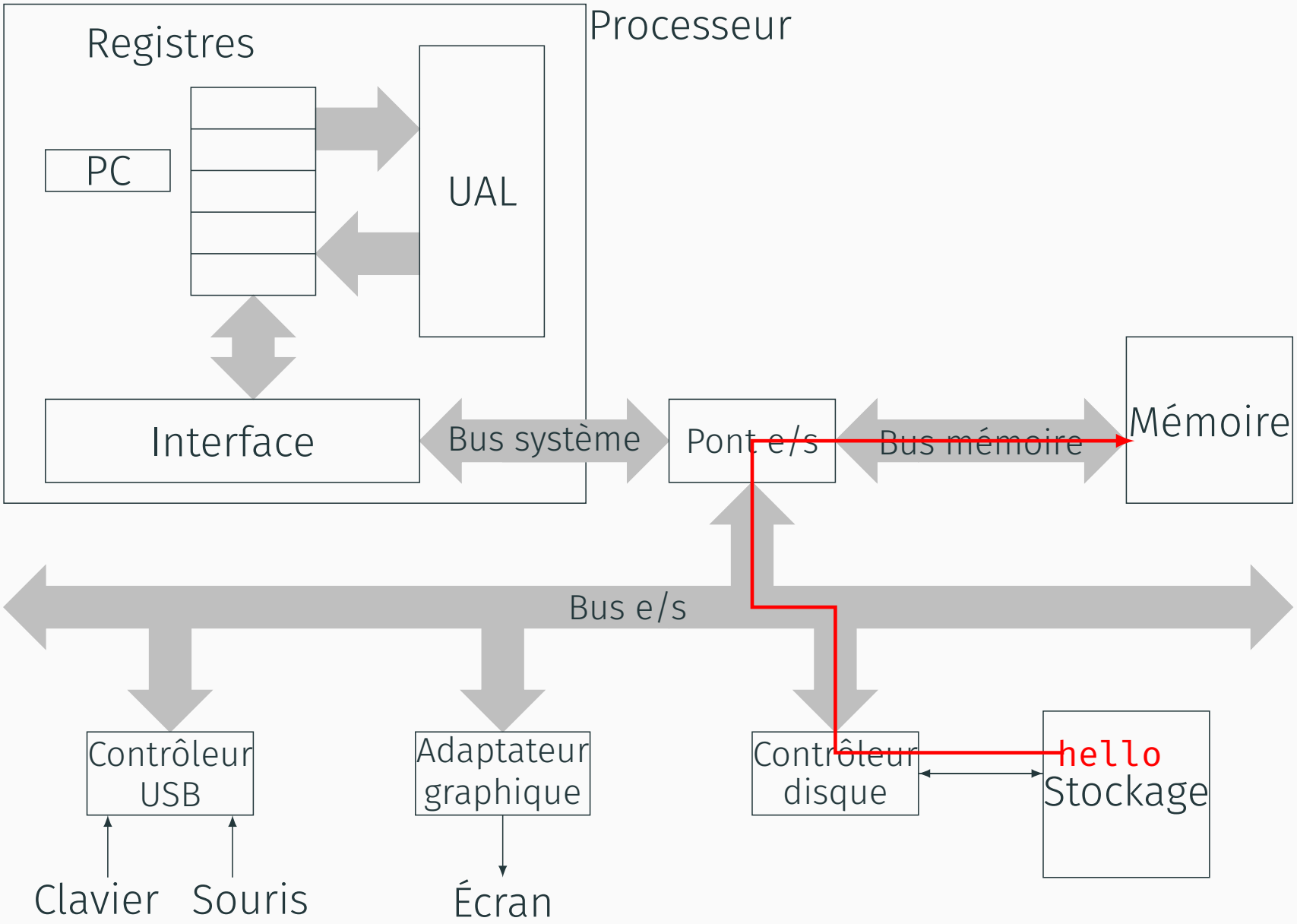
Plan du cours

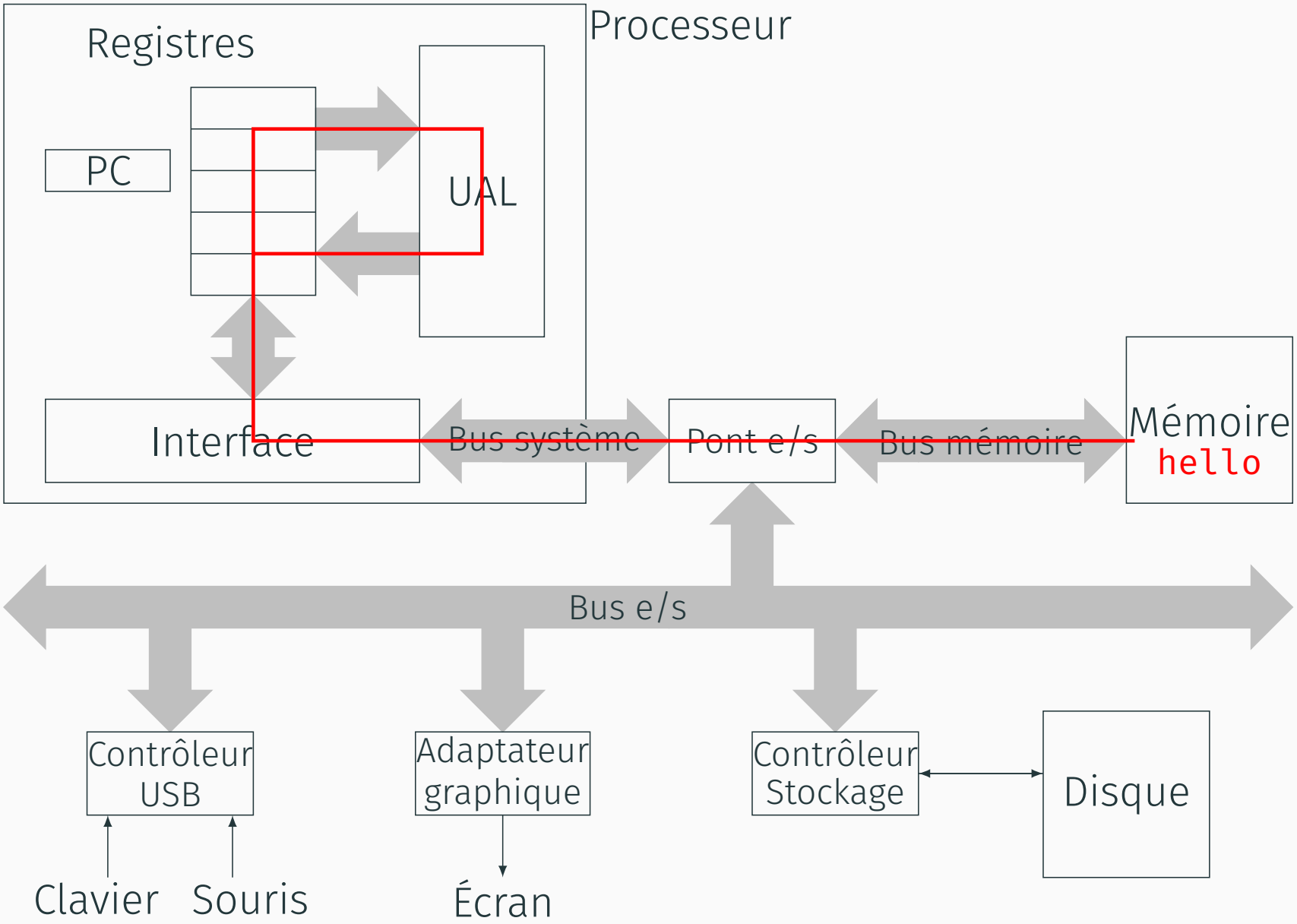
Pour « lancer le programme », le système d'exploitation doit réaliser plusieurs opérations, dont :

- créér un nouveau processus et lui associer un espace d'adressage virtuel ;
- charger dans cet espace d'adressage le code et les données du programme (copie disque → mémoire de travail) ;
- initialiser les autres zones de l'espace d'adressage (en particulier : pile et tas) ;
- initaliser les descripteurs de ressources associés à ce processus (entre autres : fichiers ouverts, dont entrée standard et sortie standard) ;
- charger dans le pointeur d'instruction l'adresse du point d'entrée du programme.

Schéma de principe d'un ordinateur




Processeur
Registres
PC
UAL
Interface
Bus système
Pont e/s
Bus mémoire
Mémoire
Bus e/s
Contrôleur USB
Adaptateur graphique
Contrôleur disque
hello
Stockage
Clavier
Souris
Écran


Processeur
Registres
PC
UAL
Interface
Bus système
Pont e/s
Bus mémoire
Mémoire
hello
Bus e/s
Contrôleur USB
Adaptateur graphique
Contrôleur Stockage
Disque
Clavier
Souris
Écran

Plan du cours

Microarchitecture MIPS

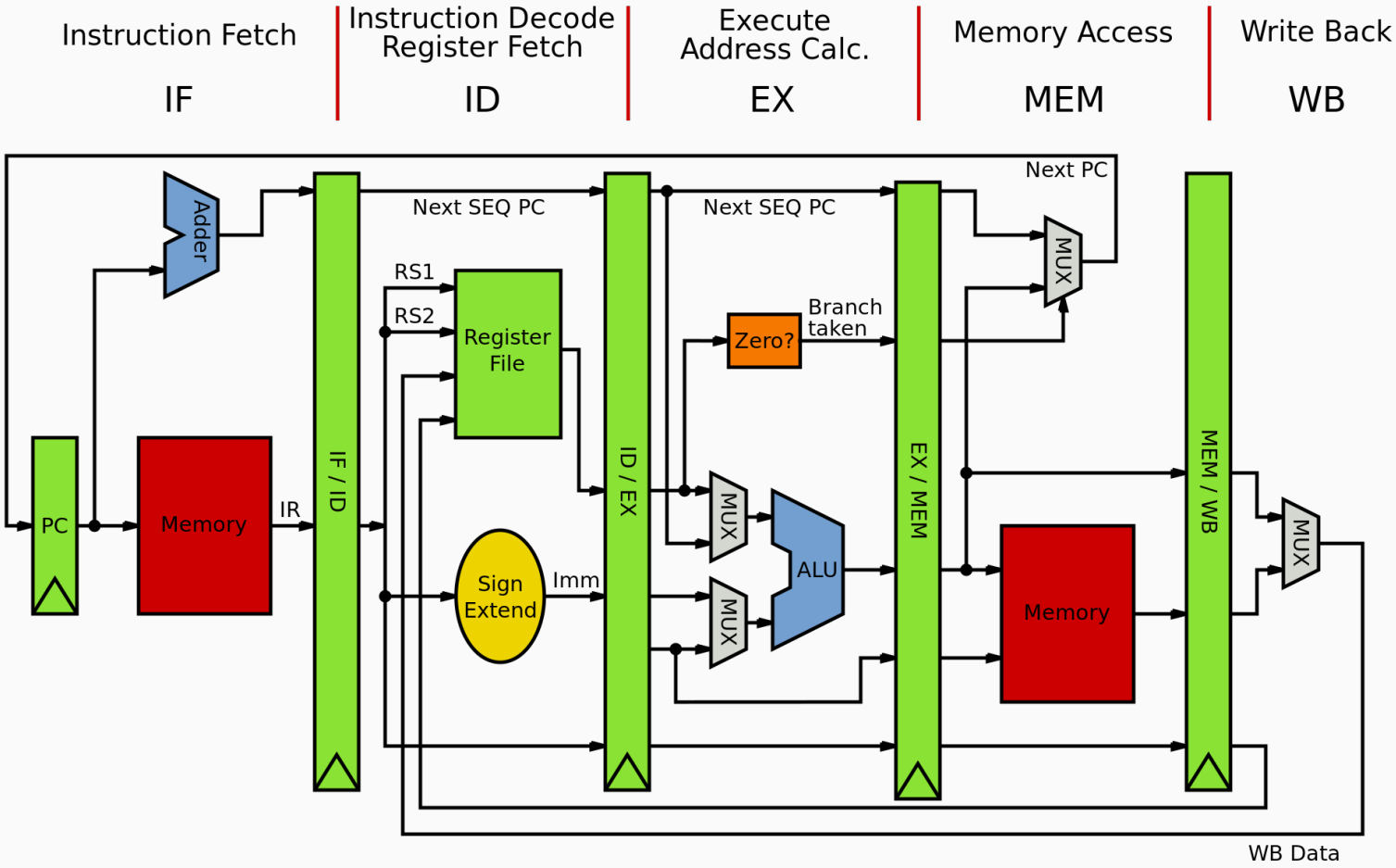


Schéma de principe d’un processeur simple.

Microarchitecture CVA6

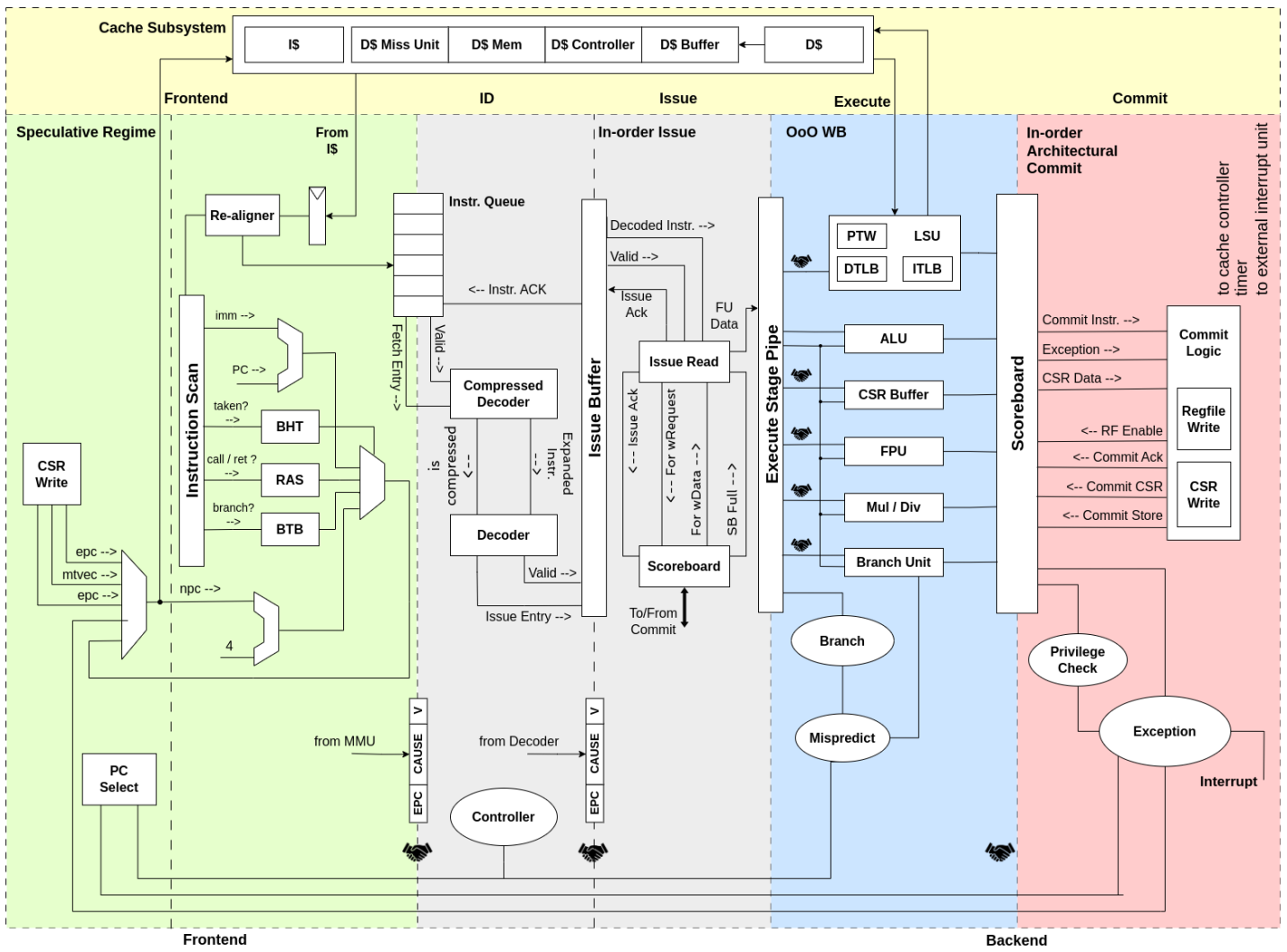


Schéma de principe d’un processeur un peu plus complexe intégrant des mécanismes de spéculation.

Les registres

Fichier de registre : mémoire très petite mais accès quasi-immédiat (dans le processeur). Elle contient :

- l'état du processeur ;
- l'état du processus ;
- les valeurs en cours d'utilisation par le programme.

Taille d'un registre = taille d'une adresse mémoire (dépend de l'architecture, 32 ou 64 bits pour les architectures modernes).

Chaque registre porte un nom utilisé pour le désigner comme opérande.

Les registres (2)

L'ISA spécifie un jeu de registre accessibles par les programmes

- soit comme opérande explicite ;
- soit comme opérande de sortie implicite.

La microarchitecture contient souvent plus de registres :

- registres internes au pipeline ;
- ou étage d'allocation des registres ISA aux registres physiques.

Plan du cours

Espace d'adressage virtuel

- abstraction de la hiérarchie mémoire du système ;
- au début : contient le code, les globales initialisées, les constantes, les paramètres, l'environnement ;
- vue du programme : un tableau de mots
  - taille d'un mot : dépend de l'architecture → typiquement 4 ou 8 octets.
  - taille du tableau : dépend de l'architecture → typiquement 2^{32} ou 2^{64} octets.
- organisation en sections par le compilateur : pile, tas, programme, données.

Les sections

La pile (`stack`)
Utilisée pour gérer les appels de procédure :

- À chaque appel, ajout d'un cadre au sommet de la pile pour les paramètres et variables locales.
- Au retour, le cadre est supprimé

Pointeur de pile : registre dédié qui contient l'adresse du sommet de la pile.

Le tas (`heap`)
Utilisée pour les structures de données dynamiques et certaines variables locales. Allocation et désallocation pilotée par le programme.

Le programme (`text`)
Utilisée pour le programme, section en lecture seule.

Les données (`data`, `bss`, `rodata`, ...)
Utilisées pour les données dont la durée de vie est celle du programme, et pour les constantes. Sur certaines archi., `rodata` et

Plan du cours

Observation d'une exécution à l'aide de **gdb**.