Attaques matérielles et sécurisation

1. Architecture matérielle

Eric Lacombe, Anaïs Gantet, Ludovic Tyack, Guillaume Baud-Berthier, Benoît Morgan
TLS-SEC

Introduction

Introduction

Qu'est-ce qu'est la sécurité matérielle

L'ordinateur élémentaire

Les principaux composants

Microarchitecture des microprocesseurs

Exécution dans le désordre

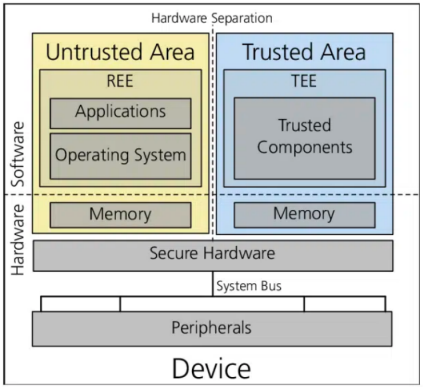
Exécution spéculative

Mémoires cache et *cache attacks*

Hardware vs software

La sécurité des logiciels considère souvent que :

- Le matériel exécutant le programme est fiable
  - Réalise les instructions constituant le programme, y compris ses bugs
- L'environnement est maitrisé
  - Architecture physique connue
  - Interfaces non exposées considérées de confiances
- L'accès physique est impossible
  - Cycle de vie du produit maitrisé
  - Sous-traitants de confiances
  - Utilisateurs consciencieux



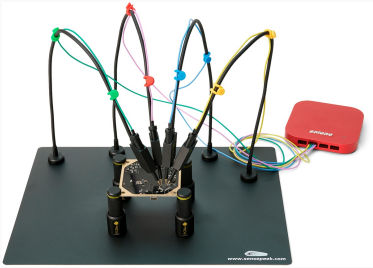
Trusted Execution Environment

(Source : Javier González)

Hardware vs software

Pourtant le matériel n'est pas parfait :

- L'exécution / les données peuvent être erronées :
  - Une instruction corrompue
  - Un registre corrompu
  - Une mémoire corrompue
  - Isolation vulnérable
- Des données sensibles peuvent fuiter :
  - Pendant un stockage de la donnée (ex. Dump hors ligne)
  - Pendant le transfert de la donnée (ex. Sniff)
  - Pendant l'utilisation de la donnée (ex. Attaques par canaux auxiliaires)
- L'architecture interne peut être altérée :
  - Remplacement / suppression des composants (ex. dégradation de la sécurité)
  - Ajouts de composants (ex. piégeage, attaque de l'homme du milieu, version du matériel)
- Les interfaces internes peuvent être accédées directement :
  - Interfaces de débogages
  - Mémoires internes
  - Communications inter-composants



Analyse de signaux logiques

(Saleae Pro 16 + PCBite probes)

Plan du cours

Composition de l'UE 'Attaques matérielles et sécurisation' :

- 2*CM Side Channel & Fault Injection
- 1*CM Introduction
- 1*CM Virtualisation
- 1*TD Virtualisation
- 1*TD Démarrage sécurisé
- 1*TD Mécanismes de protections
- 2*TD Attaques micro-architecturales
- 1*TP Virtualisation
- 2*TP Attaque d'un module ESP
- Examen

Introduction

- Qu'est-ce qu'est la sécurité matérielle
- L'ordinateur élémentaire
- Les principaux composants

Microarchitecture des microprocesseurs

- Exécution dans le désordre
- Exécution spéculative
- Mémoires cache et *cache attacks*

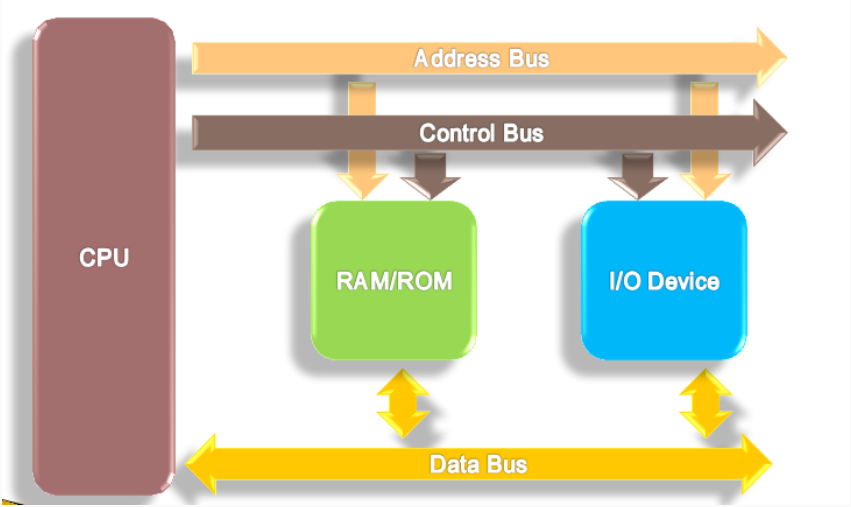
Définitions

Définition :

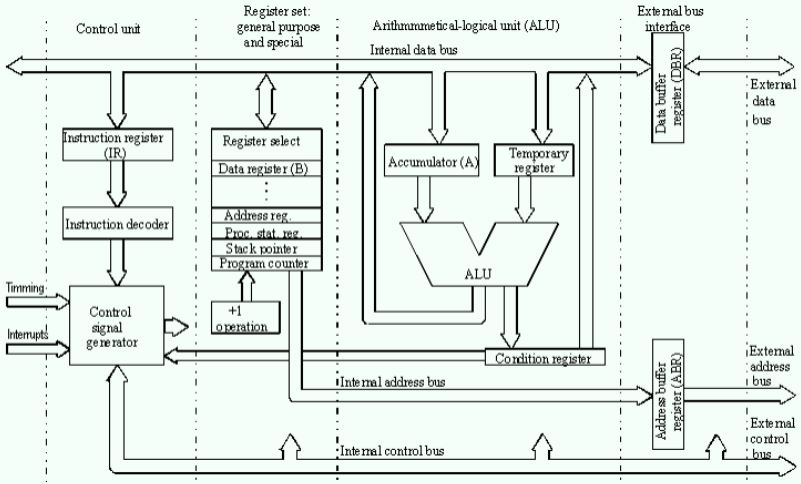
*Wikipedia : Un ordinateur est un système de traitement de l'information **programmable** tel que défini par Alan Turing et qui fonctionne par la lecture séquentielle d'un ensemble d'instructions, organisées en programmes, qui lui font exécuter des opérations logiques et arithmétiques.*

- **Le microprocesseur (CPU)** : Composant principal d'un ordinateur (unité(s) arithmétique(s), registres, pipelines, caches, ...) qui expose son bus d'adresse, de données et de contrôle.
- **La mémoire volatile (RAM)** : mémoire non persistante, rapide, utilisé pour les piles d'exécution (stack, heap) ou le mapping mémoire (mmap).
- **La mémoire non-volatile (NVM / ROM / Flash / ...)** : mémoire persistante, plus lente. Contient les logiciels, les configuration et les données.
- **Les périphériques** : entrées/sorties de l'ordinateur (clavier, souris, carte graphique, carte son, capteurs, actionneurs, etc...)

Architecture simplifié d'un ordinateur :

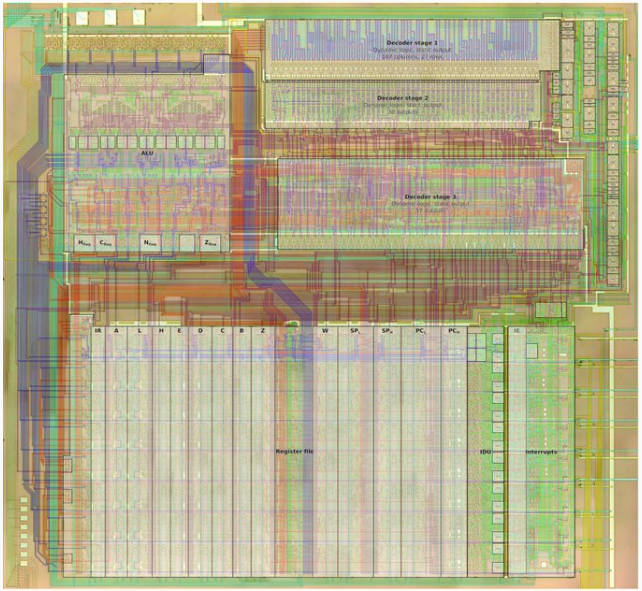


Exemple d'architecture CPU simple :



Basic block diagram of a 8-bit microprocessor

Exemple d'architecture CPU simple (CPU gameboy SM83) :

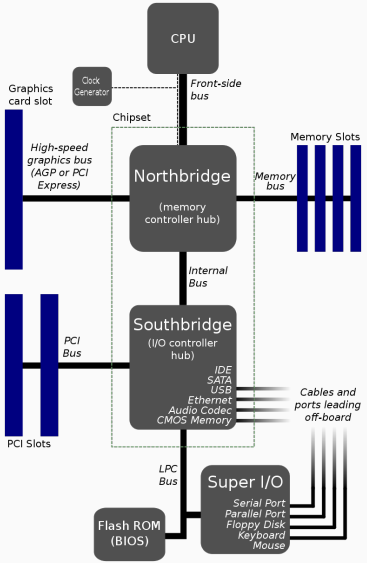


L'ordinateur contenporain : exemple du PC

Les premières cartes mères :

- CPU : exécute les instructions
- Northbridge : interconnexion haute-vitesse
  - les barrettes de mémoires RAM
  - la carte graphique
- Southbridge : interconnexion basse vitesse
  - Le(s) disque(s) dur(s)
  - Les ports USBs / Ethernet / Audio
  - les cartes PCIs
  - le bus "LPC" (Bios / TPM)
  - etc...

Remarque : Le CPU est parfois amovible (montée sur un socket)



Source : Kimon Berlin

Introduction

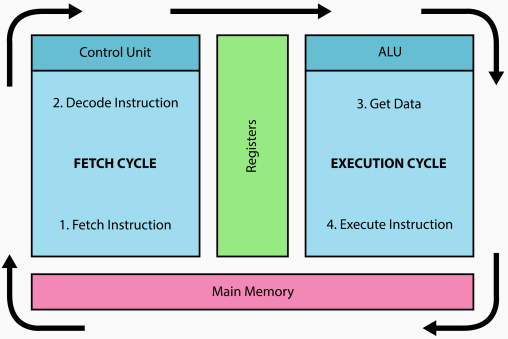
- Qu'est-ce qu'est la sécurité matérielle
- L'ordinateur élémentaire
- Les principaux composants

Microarchitecture des microprocesseurs

- Exécution dans le désordre
- Exécution spéculative
- Mémoires cache et *cache attacks*

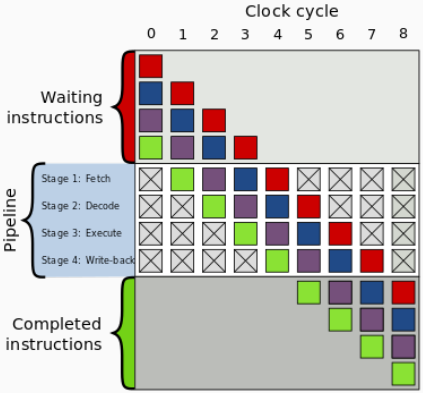
Le cycle Fetch/Decode/Execute des instructions :

- Mise à jour du Program Counter
- Chargement de l'instruction (FETCH)
- Décodage de l'instruction (DECODE)
- Lecture des opérandes dans les registres
- Calcul impliquant l'ALU (EXEC)
- Accès mémoire
- Ecriture du résultat dans les registres (WRITEBACK)



Les opérations sont cadences par une horloge (ex. 2GHz)

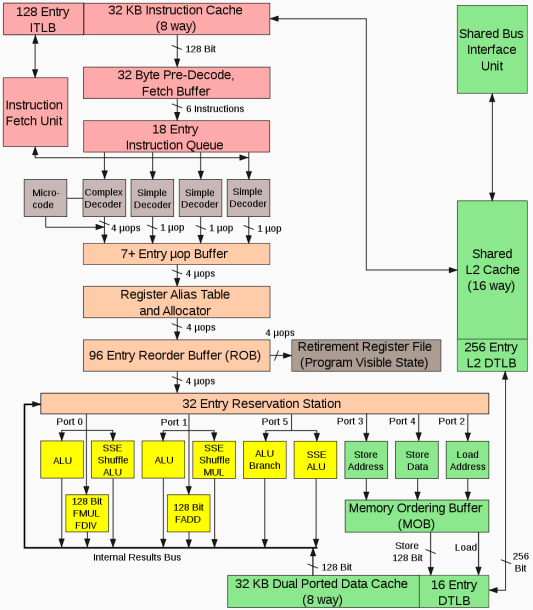
Les étapes sont parallélisées pour plus de performance (jusqu'à 1 exécution complète par cycle d'horloge)



Problèmes :

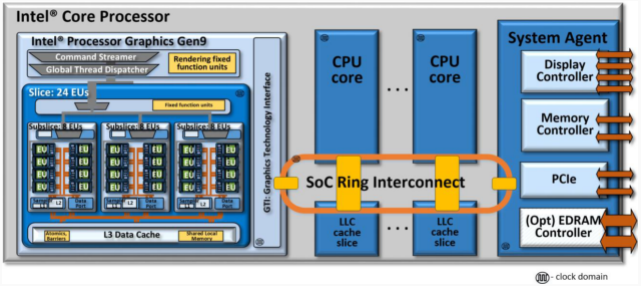
- La mauvaise instruction est parfois décodée (branches / interruption synchrone - etc)
  => Prédiction de branches (estimation de la prochaine instruction à exécuté selon certains critères)
- Complexification du pipeline (jusqu'à 20 étapes sur x86)
  => Attaques micro-architecturales (ex. meltdown)

Le CPU - Exemple de micro-architecture

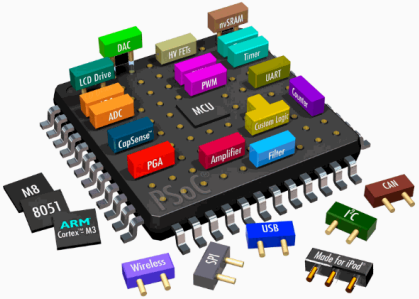


Intel Core 2 Architecture

Plusieurs CPUs interconnectés dans un même composant :



Les Périphériques

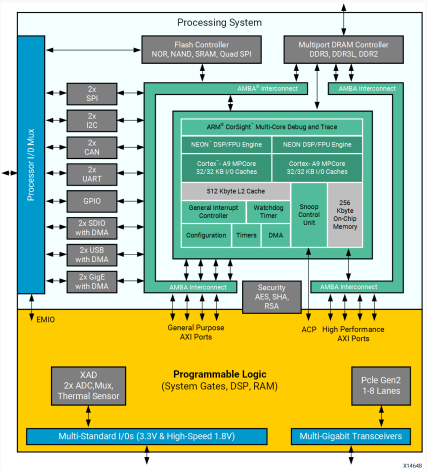


Les SoCs/ le MCUs / le FPGAs intègrent de nombreux ‘périphériques’ interconnectés :

- Des entrées/sorties logiques (GPIO)
- Des convertisseurs analogiques (ADC / DAC)
- Des horloges (ex. Timer, Watchdog timers, RTC)
- Des contrôleurs d’interfaces (USART, USB, PCIe)
- Des contrôleurs de mémoires (RAM, Flash, Cache)
- des unités de gestion mémoires (MMU, SMMU, IOMMU)
- Des contrôleurs d’interruptions (GIC, PIC)
- Des contrôleurs d’alimentation (PSCI, DVFS)
- Des accélérateurs matériels (GPU, DSP, NPU, processeurs cryptographiques/SIMD)
- Des débogueurs (ex. Corsight sur ARM)

Plusieurs interconnexions possibles avec le CPU :

- **Les connexions 'memory-mapped'** : Le CPU accède à une adresse spécifique (ex. instruction 'MOV') via un bus interne (ex. AXI)
- **Les connexions 'port-mapped'** : Le CPU accède à un registre spécifique (ex. GDTR/IDTR sur x86) via une instruction spécifique (ex. MSR/MRS sur armv8)
- **Les signaux d'interruptions** : Sur réception d'un événement externe, le CPU exécute une 'routine d'interruption'
- **Les 'DMA' (Direct Memory Access)** : Le périphérique accède directement à l'espace mémoire du CPU
- **Les accès indirects** : Le CPU accède au périphérique via un autre périphérique/contrôleur d'interfaces (ex. bus SPI, bus PCIe, bus USB)



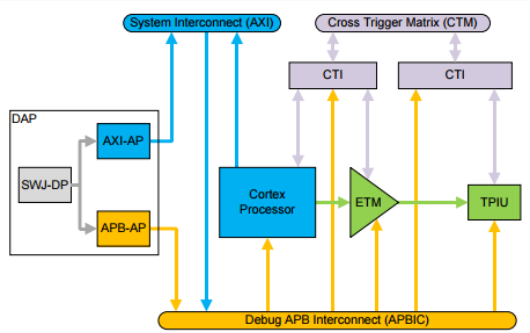
AMD Zynq-7000 Architecture

Les bits d'un registre sont associés à des fonctions particulières. Certains sont en lecture seule (ex. bits d'états), d'autre en écriture seuls (ex. trigger).

Les interfaces de débogage

Les SoCs / MCUs / FPGAs fournissent souvent une infrastructure de débogage (ex. Coresight sur ARM, Debug Module sur RISC-V) et permettent :

- un contrôle total du/des CPU et ses registres internes
- un accès aux différentes mémoires (Flash, RAM, Cache)
- Un accès aux différents périphériques internes (memory-mapped ou port-mapped)

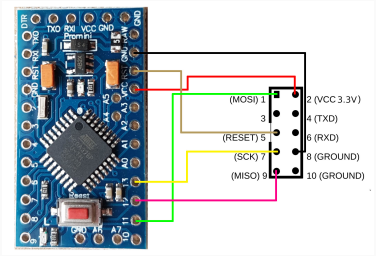


single ARM processor trace using the CoreSight infrastructure

Parmi les interfaces de débogage on trouve le JTAG et le SWD - mais peut aussi se faire via USB, PCIe ou même depuis un des cœurs (ex. attaque Nailgun)

Parfois associée à l'interface de débogage, l'interface de programmation permet de lire et/ou écrire les mémoires persistantes :

- La programmation 'in-situ' (In-System Programming) : interface permettant la (re)programmation du composant une fois intégré sur le PCB final
- La programmation via le logiciel (ex. bootloader, bootrom en mode DFU) : Mode particulier permettant la (re)programmation de ses mémoires via une ou plusieurs interfaces standard (ex. USB, UART).
- La programmation 'chip-off" : La mémoire nécessite d'être retiré pour (re)programmation (configuration de programmation particulière, tension de programmation élevés, etc...)

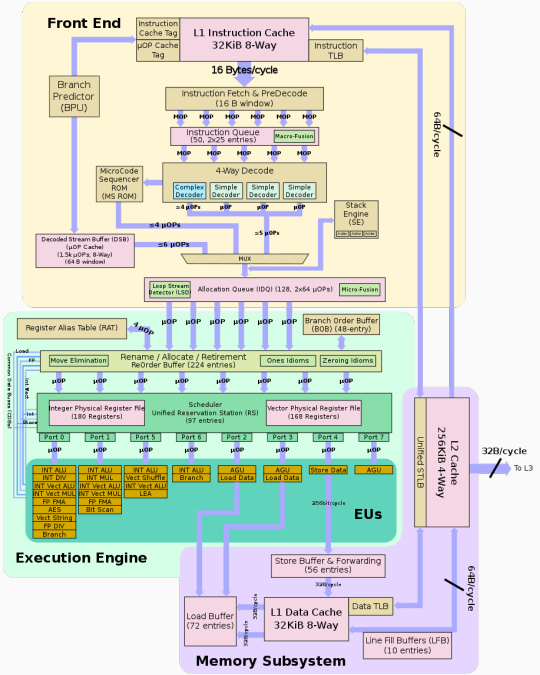


Arduino In-System Programming interface

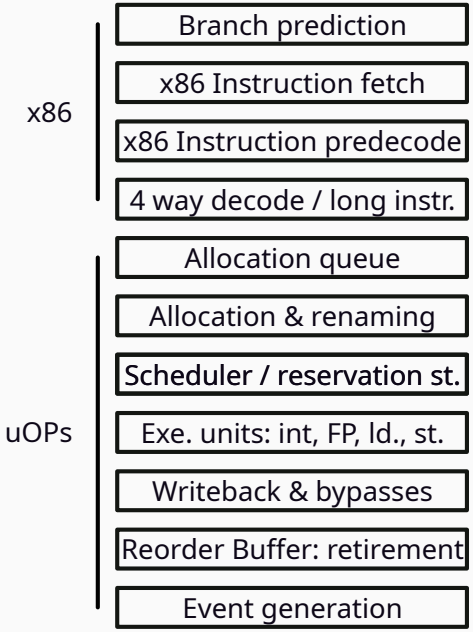
Selon les cas, les mémoires internes et/ou externes peuvent être (re)programmés.

Microarchitecture des microprocesseurs

Cœur de processeur Intel x86 - architecture Skylake



Vie d'une instruction dans le pipeline



WTF is happening ???

Back to back `rdtsc`

```
1  rdtsc
2  shl $32, %rdx
3  or %rdx, %rax
4  mov %rax, tsc_start
5
6  rdtsc
7  shl $32, %rdx
8  or %rdx, %rax
9  mov %rax, tsc_end
```

...with slow computation in the middle

```
1  rdtsc
2  shl $32, %rdx
3  or %rdx, %rax
4  mov %rax, tsc_start
5
6  xor %rax, %rax
7  add (%rcx, %rax), %rax // rcx <-> &d
8
9  rdtsc
10 shl $32, %rdx
11 or %rdx, %rax
12 mov %rax, tsc_end
```

```
1  $ taskset -c 0 lst/ooo_rdtsc_naked.elf
2  TSC = 0x0000000000000010
```

- Semantique du C intuitivement prend du temps a cause de la dépendance des operandes (R after W dependency)
- La variable d n'est pas dans le cache
- Chargemenent beaucoup plus rapide qu'attendu ...

```
1  $ taskset -c 0 lst/ooo_rdtsc_no_lfence.elf
2  TSC = 0x0000000000000010
```

```
1  $ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq | sudo tee /sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq
2  $ echo performance | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

WTF is happening ???

Serialize pipeline with `lfence`

```
1  rdtsc
2  shl $32, %rdx
3  or %rdx, %rax
4  mov %rax, tsc_start
5
6  xor %rax, %rax
7  add (%rcx, %rax), %rax // rcx <-> %d
8
9  lfence
10
11 rdtsc
12 shl $32, %rdx
13 or %rdx, %rax
14 mov %rax, tsc_end
```

```
1 $ taskset -c 0 lst/ooo_rdtsc_lfence.elf
2 TSC = 0x0000000000000132
```

- Serialisation de la machine avec `lfence`
- On attend la fin de l'exécution des instructions précédentes avant d'exécuter `rdtsc`
- La durée d'exécution est plus importante

→ Observation de l'effet réel de l'exécution de l'instruction `load` du `add` avec opérande mémoire.

Serialize pipeline with `lfence`

```
1  rdtsc
2  shl $32, %rdx
3  or %rdx, %rax
4  mov %rax, tsc_start
5
6  xor %rax, %rax
7  add (%rcx, %rax), %rax // rcx <-> &d
8
9  lfence
10
11 rdtsc
12 shl $32, %rdx
13 or %rdx, %rax
14 mov %rax, tsc_end
```

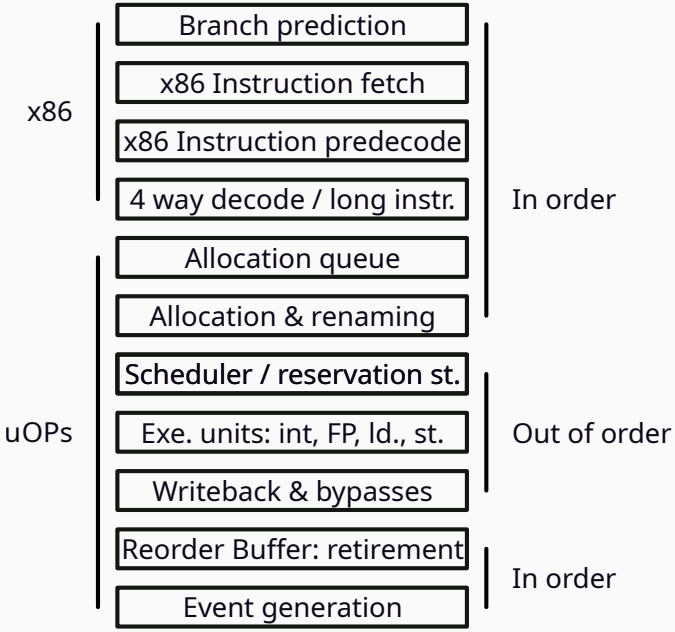
```
1 $ taskset -c 0 lst/ooo_rdtsc_lfence.elf
2 TSC = 0x0000000000000132
```

- Serialisation de la machine avec `lfence`
- On attend la fin de l'exécution des instructions précédentes avant d'exécuter `rdtsc`
- La durée d'exécution est plus importante

→ Observation de l'effet réel de l'exécution de l'instruction `load` du `add` avec opérande mémoire.

This is OOO black magic folks !

Vie d'une instruction dans le pipeline : OOO



Introduction

- Qu'est-ce qu'est la sécurité matérielle
- L'ordinateur élémentaire
- Les principaux composants

Microarchitecture des microprocesseurs

- Exécution dans le désordre
- Exécution spéculative
- Mémoires cache et *cache attacks*

Principe simplifié de l'exécution dans le désordre

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```

Principe simplifié de l'exécution dans le désordre

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```

Principe simplifié de l'exécution dans le désordre

```
add %rax, %rcx
add %rdx, %rsi
```

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```

Principe simplifié de l'exécution dans le désordre

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```

```
add %rax, %rcx
add %rdx, %rsi
```

```
mov $1, %rax ✓
mov $2, %rbx ✓
mov $3, %rdx
```

Principe simplifié de l'exécution dans le désordre

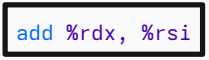
```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```

```
add %rax, %rcx
add %rdx, %rsi
```

```
mov $3, %rdx
```

Principe simplifié de l'exécution dans le désordre

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```



```
mov $3, %rdx
```

Principe simplifié de l'exécution dans le désordre

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```

```
add %rax, %rcx
add %rdx, %rsi
mov $3, %rdx
```

Principe simplifié de l'exécution dans le désordre

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```

```
add %rax, %rcx
add %rdx, %rsi
mov $3, %rdx
```

Principe simplifié de l'exécution dans le désordre

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rdx
add %rdx, %rsi
```

Cas plus complexe d'exécution dans le désordre

```
mov  $1,  %rax
mov  $2,  %rbx
add %rax, %rcx
mov $3, %rax
add %rax, %rsi
```

Affectation de concurrente de `rax`

Cas plus complexe d'exécution dans le désordre

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rax
add %rax, %rsi
```

Corruption de `rax`

Cas plus complexe d'exécution dans le désordre

```
mov $1, %rax
mov $2, %rbx
add %rax, %rcx
mov $3, %rax
add %rax, %rsi
```

Rennomage de registres !

Exécution dans le désordre

Exigences pour la mise en oeuvre de l'exécution dans le désordre efficace :

1. Renommage des registres
2. Unités d'exécution multiples :
3. **Un `reorder buffer` assez grand**

Exécution dans le désordre

Exigences pour la mise en oeuvre de l'exécution dans le désordre efficace :

1. Renommage des registres
2. Unités d'exécution multiples : **OK !**
3. **Un** `reorder buffer` **assez grand**

Register renaming

Separate slide deck

Introduction

- Qu'est-ce qu'est la sécurité matérielle
- L'ordinateur élémentaire
- Les principaux composants

Microarchitecture des microprocesseurs

- Exécution dans le désordre
- Exécution spéculative
- Mémoires cache et *cache attacks*

WTF is happening (reloaded) ???

```
1   call _3
2
3  _1:
4
5   inc d // Loads d in cache
6   lfence
7
8  _2:
9
10  rdtsc
11  shl $32, %rdx
12  or %rdx, %rax
13  mov %rax, tsc_start
14
15  mov d, %rax // That should be fast
16  lfence
17
18  rdtsc
19  shl $32, %rdx
20  or %rdx, %rax
21  mov %rax, tsc_end
22
23  jmp _4
24
25 _3:
26
27  ret
28
29 _4:
```

```
1 $ taskset -c 0 lst/spec_rsb.elf
2 TSC = 0x0000000000000014, d = 1
```

1. On appelle _3 qui retourne qui retourne directement
2. On precharge la variable d qui n'est pas dans le cache
3. On load d à nouveau en mesurant le temps. On s'apperçoit que la variable est bien dans le cache

WTF is happening (reloaded) ???

```
1   call _3
2
3 _1:
4
5   inc d // Loads d in cache
6   lfence
7
8 _2:
9
10  rdtsc
11  shl $32, %rdx
12  or %rdx, %rax
13  mov %rax, tsc_start
14
15  mov d, %rax // That should be slow now...
16  lfence
17
18  rdtsc
19  shl $32, %rdx
20  or %rdx, %rax
21  mov %rax, tsc_end
22
23  jmp _4
24
25 _3:
26  clflush (%rsp)
27  mfence // Makes sure clflush is complete before retiring
28  lfence // Serializes execution before executing ret
29  addq $_2-_1, (%rsp)
30  ret
31
32 _4:
```

```
1 $ taskset -c 0 lst/spec_rsb.elf
2 TSC = 0x0000000000000014, d = 0
```

1. On appelle _3 qui qui modifie son adresse de retour de pour aller directement à _2
2. On **ne precharge pas** la variable d qui reste en dehors du cache
3. On load d en mesurant le temps. On s'apperçoit que **la variable est tout de même dans le cache !!!**

WTF is happening (reloaded) ???

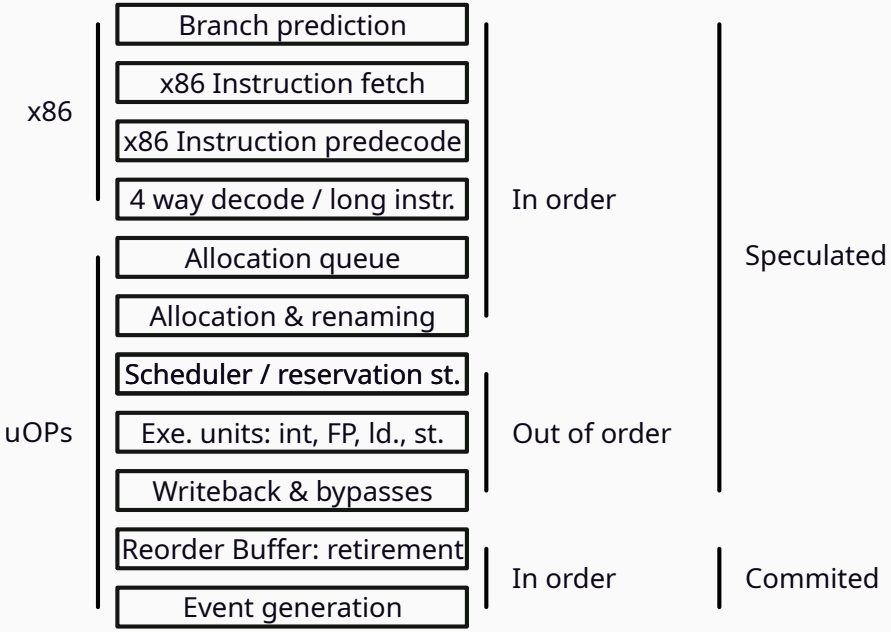
```
1   call _3
2
3 _1:
4
5   inc d // Loads d in cache
6   lfence
7
8 _2:
9
10  rdtsc
11  shl $32, %rdx
12  or %rdx, %rax
13  mov %rax, tsc_start
14
15  mov d, %rax // That should be slow now...
16  lfence
17
18  rdtsc
19  shl $32, %rdx
20  or %rdx, %rax
21  mov %rax, tsc_end
22
23  jmp _4
24
25 _3:
26  clflush (%rsp)
27  mfence // Makes sure clflush is complete before retiring
28  lfence // Serializes execution before executing ret
29  addq $_2-_1, (%rsp)
30  ret
31
32 _4:
```

```
1 $ taskset -c 0 lst/spec_rsb.elf
2 TSC = 0x0000000000000014, d = 0
```

1. On appelle _3 qui qui modifie son adresse de retour de pour aller directement à _2
2. On **ne precharge pas** la variable d qui reste en dehors du cache
3. On load d en mesurant le temps. On s'apperçoit que **la variable est tout de même dans le cache !!!**

Le prédicteur de branchement a prédit de revenir à l'adresse de retour, avant de se raviser

Vie d'une instruction dans le pipeline : speculation



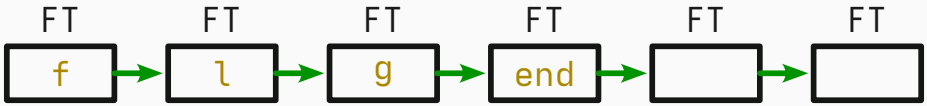
Exemple decoupage programme


```
f:
  mov $1, %rax
  cmp $0, %rax
  jge g  // rax >= $0 ?
  .align 0x20
l:
  mov $"WTF", %rbx
  jmp end
  .align 0x20
g:
  mov $"n00bz", %rbx
  .align 0x20
end:
  ret
```

Exemple decoupage programme

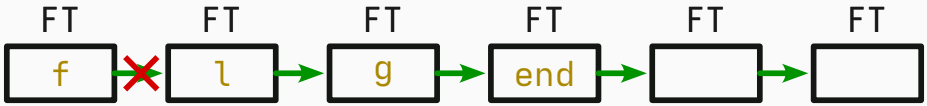

```
f:
  mov $1, %rax
  cmp $0, %rax
  jge g  // rax >= $0 ?
  .align 0x20
l:
  mov $"WTF", %rbx
  jmp end
  .align 0x20
g:
  mov $"n00bz", %rbx
  .align 0x20
end:
  ret
```

Exemple decoupage programme



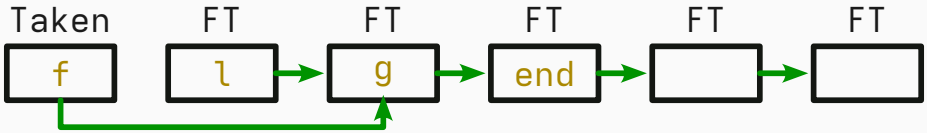

```
f:
  mov $1, %rax
  cmp $0, %rax
  jge g  // rax >= $0 ?
  .align 0x20
l:
  mov $"WTF", %rbx
  jmp end
  .align 0x20
g:
  mov $"n00bz", %rbx
  .align 0x20
end:
  ret
```

Exemple decoupage programme




```
f:
  mov $1, %rax
  cmp $0, %rax
  jge g  // rax >= $0 ?
  .align 0x20
l:
mov $"WTF", %rbx
jmp end
  .align 0x20
g:
mov $"n00bz", %rbx
  .align 0x20
end:
ret
```

Cancels execution




```
f:
  mov $1, %rax
  cmp $0, %rax
  jge g  // rax >= $0 ?
  .align 0x20
l:
  mov $"WTF", %rbx
  jmp end
  .align 0x20
g:
  mov $"n00bz", %rbx
  .align 0x20
end:
  ret
```

Corrects branch type

Introduction

- Qu'est-ce qu'est la sécurité matérielle
- L'ordinateur élémentaire
- Les principaux composants

Microarchitecture des microprocesseurs

- Exécution dans le désordre
- Exécution spéculative
- Mémoires cache et *cache attacks*

Mémoires cache et *cache attacks*

Separate slide deck