Sécurité matérielle

L'interface matériel/logiciel

Ronan Lashermes et Hélène Le Bouder

Ce document utilise un style dérivé de Legrand orange book.
Édition 7 janvier 2025

Nous remercions Guillaume Didier pour ses relectures et ses commentaires.

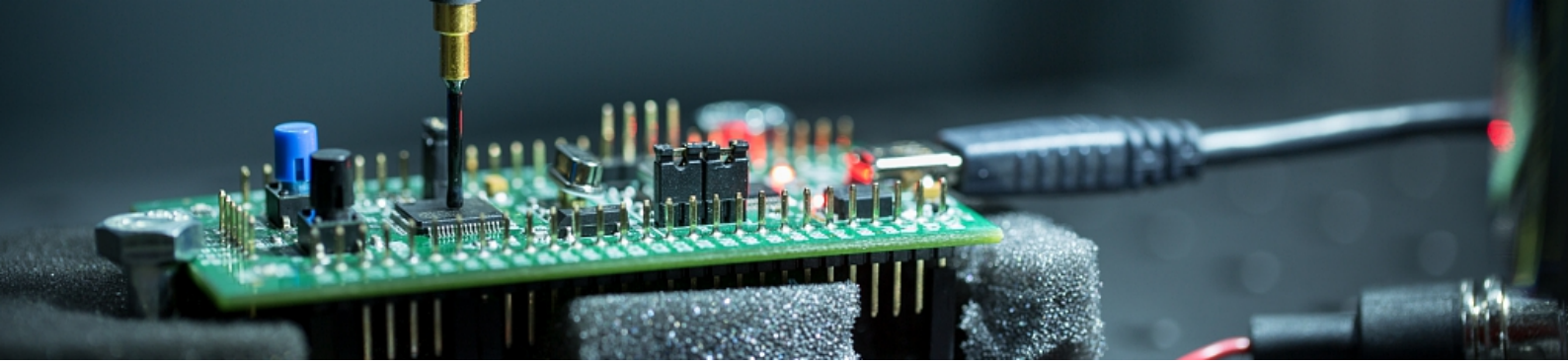


Table des matières

II Architectures et microarchitectures

Annexes

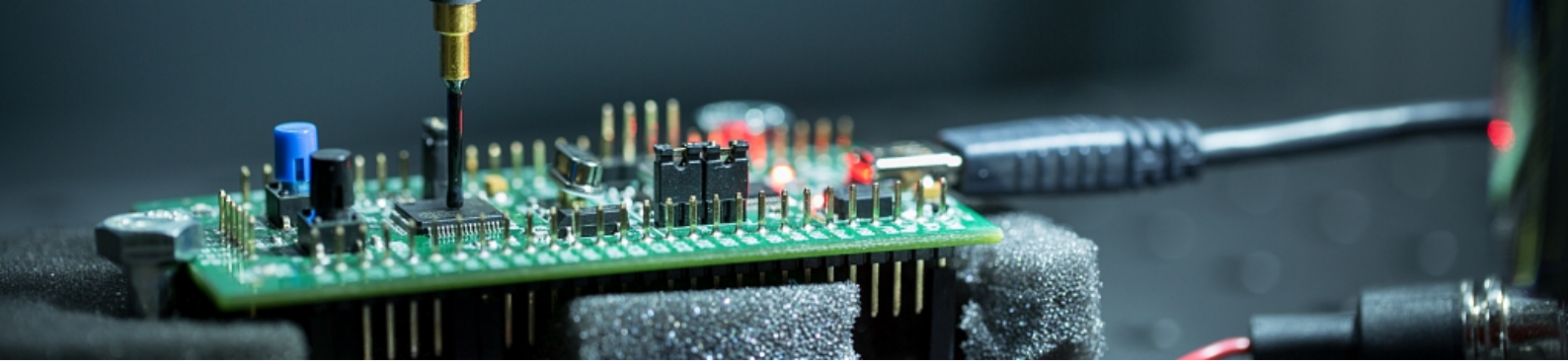
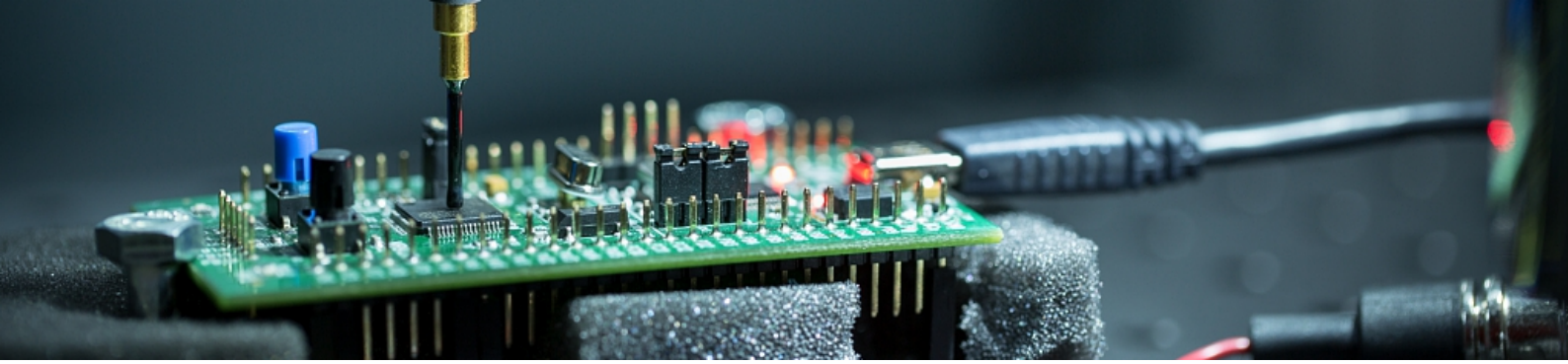
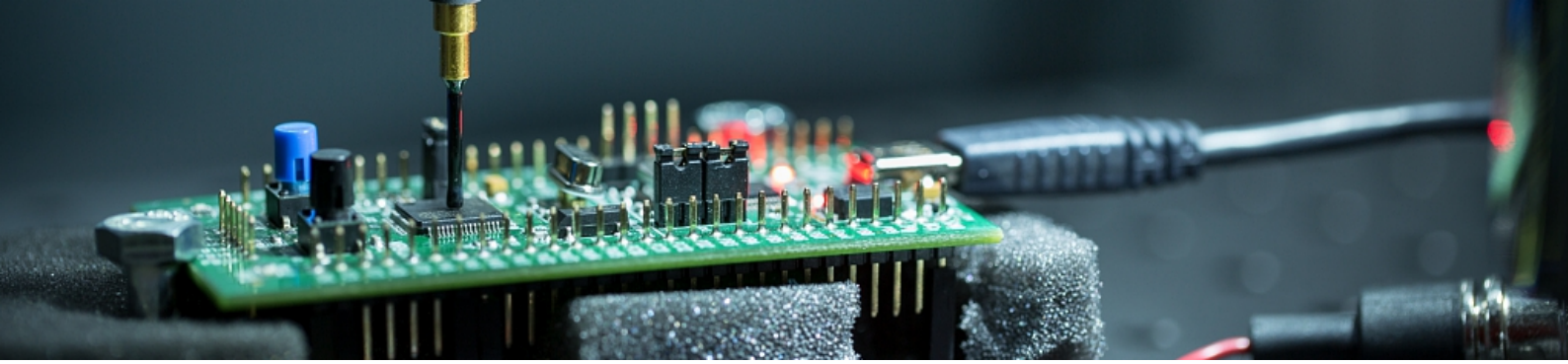


Table des figures



Liste des tableaux



Liste des exercices

0.1 Introduction générale

Un système informatique moderne repose sur une tour d'abstraction, illustrée sur la figure 1. Qui désire devoir réfléchir aux signaux électriques des transistors lors du développement d'une application pour téléphone mobile ?

Ces abstractions sont des modèles mentaux pour les humains qui les utilisent. Sans cela, nous serions dépassés par la complexité de nos systèmes.

Ces abstractions permettent également une méthode efficace pour faire collaborer des personnes et des organisations. En ce sens, elles sont une émanation de la loi de Conway.

Définition 0.1 - *Loi de Conway (1967)*

La structure des systèmes reproduit la structure des communications de l'organisation qui les produisent.

Mais ces abstractions ne sont pas parfaites, elles « fuient ». D'une part, un attaquant peut donc tenter d'exploiter ces failles. D'autre part, il est difficile de s'en protéger : il faut raisonner simultanément avec plusieurs abstractions.

Nous allons considérer 3 grandes abstractions dans ce cours :

1. Algorithmique.
2. Logiciel.
3. Matériel.

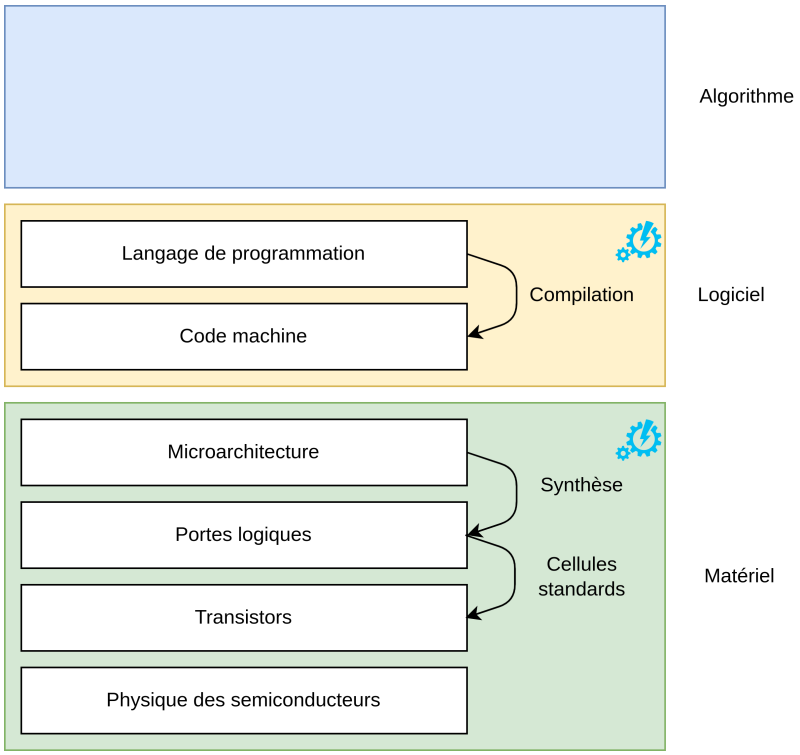


FIGURE 1 – La tour d'abstraction des systèmes informatiques.

0.2 Teaser

Après les grands principes, voici quelques problèmes concrets illustrant certaines difficultés liées aux abstractions.

```
STORE x1, 0(x4)
LOAD  x2, 0(x4)
```

Est-ce que `x1`==`x2` nécessairement après ces deux instructions ?

```
LOAD x3, 0(x5)
LOAD x4, 0(x5)
```

Est-ce que `x3`==`x4` nécessairement après ces deux instructions ?

```
uint32_t val1 = *address;
uint32_t val2 = *address;
```

Est-ce que `val1`==`val2` nécessairement ?

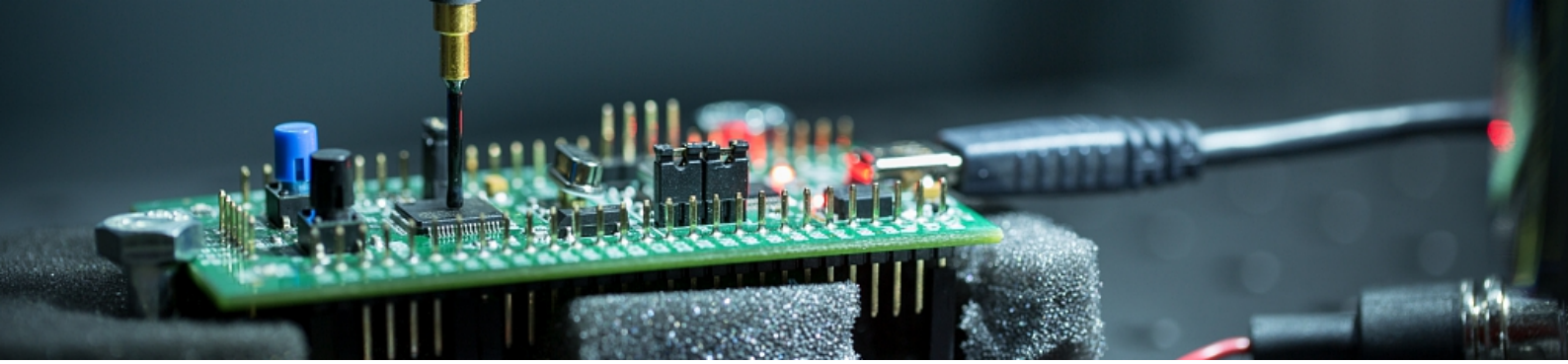
Ces questions ont des réponses étonnamment complexes, car les effets des abstractions inférieures peuvent se manifester. Ce cours a pour objectif de comprendre comment ces abstractions intéragissent entre elles, et comment elles peuvent nuire à la sécurité des systèmes.

Nous nous concentrerons sur les interfaces algorithme/logiciel et logiciel/matériel.

Technologies des circuits intégrés

«We do not choose to be created. Nor do we choose how we are made.» Sandman, série de Allan Heinberg, basée sur la série de romans graphiques de Neil Gaiman.

Ce cours se concentre sur la sécurité des mémoires et des processeurs. Mais avant cela, il nous faut quelques bases sur le fonctionnement des circuits intégrés.

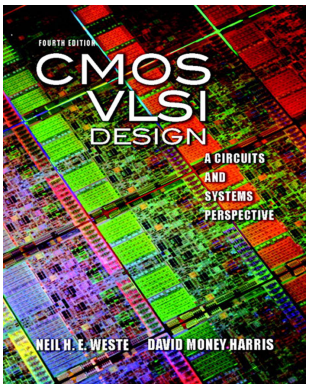


1. Les technologies des semiconducteurs

Ce chapitre est un rappel des différents concepts se rapportant aux circuits intégrés dans le but d'établir un vocabulaire commun. Il ne rentrera pas dans le détail et sera très vulgarisé, préférez des ressources dédiées si vous voulez vous initier au sujet.

> **!** Les symboles utilisés pour les composants électroniques et les portes logiques diffèrent suivant les cultures. Dans ce document, je n'utiliserai pas les symboles français, mais américains : ce sont eux que vous trouverez dans les documentations réelles.

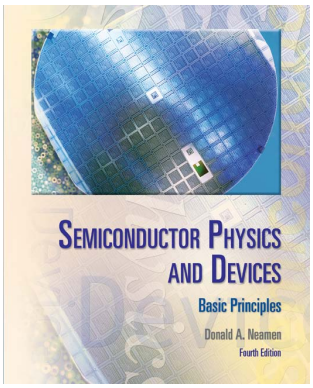
Suggestions de lecture



CMOS VLSI Design : A Circuits and Systems Perspective [47]

Par *Neil Weste, David Harris*

ISBN : 0321547748



Semiconductor Physics And Devices [44]

Par *Donald A Neamen*

ISBN : 0073529583

1.1 La jonction PN

1.1.1 Dopage

Les circuits intégrés qui nous intéressent sont essentiellement basés sur les propriétés physiques semiconductrices du silicium. À partir d'un cristal de silicium pur, il est possible d'ajouter certaines impuretés pour en modifier les caractéristiques électriques : c'est le dopage.

— Le dopage de type N, consiste à insérer des atomes comme le phosphore, donneurs d'électron. C'est à dire contenant un excès d'électrons. Le matériau en résultant est un cristal incluant des électrons libres et donc négativement chargé (d'où le N).
— Le dopage de type P au contraire, consiste à insérer des atomes comme le bore, accepteurs d'électron. On dit que l'on a ajouté des trous, une particule fictive chargée positivement (d'où le P) qui correspond au déficit d'un électron.

Il se passe des choses intéressantes lorsque l'on met en contact ces deux matériaux, c'est la jonction PN qui permet de réaliser des diodes en électronique.

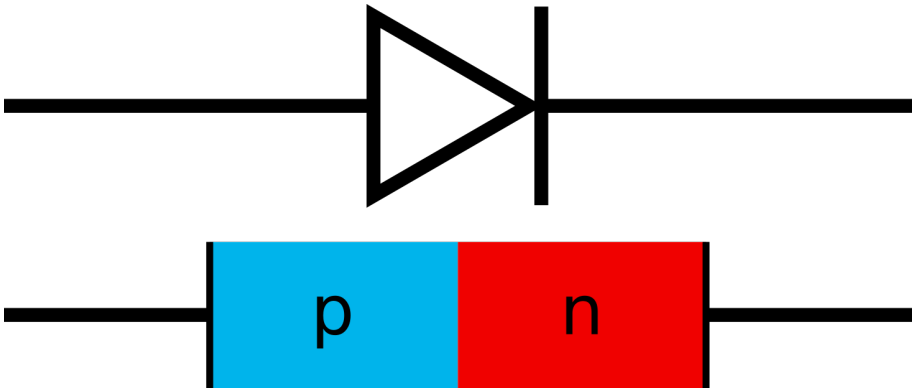


FIGURE 1.1 – La jonction PN permet de réaliser une diode.

1.1.2 Jonction PN au repos

Au repos, la jonction génère un champ électrique en son sein du fait de la diffusion des charges, jusqu'à obtenir un équilibre comme illustré sur la figure 1.2.

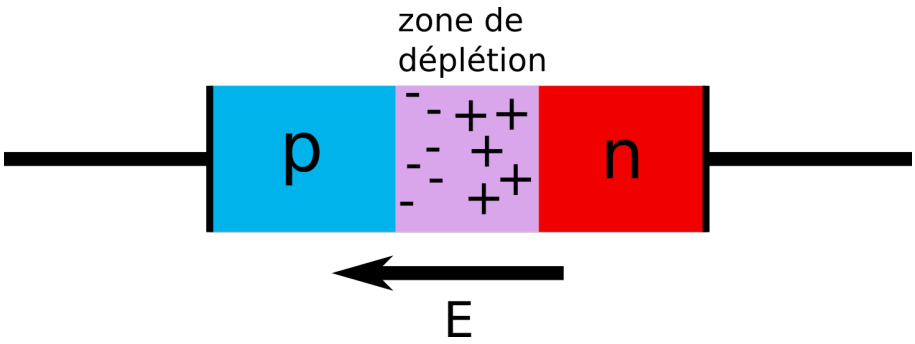


FIGURE 1.2 – Charges et champ électrique au repos.

Dans une jonction PN au repos, c'est-à-dire sans tension externe appliquée, il y a un équilibre entre deux forces opposées : la force de diffusion et la force du champ électrique.

Force de diffusion : Lors de la formation de la jonction PN, les électrons du côté N (en excès) ont tendance à diffuser vers le côté P, où il y a moins d'électrons. Simultanément, les trous du côté P (en excès) ont tendance à diffuser vers le côté N, où il y a moins de trous. Ce mouvement de diffusion des porteurs de charge résulte de la différence de concentration des charges entre les deux côtés de la jonction.

Force du champ électrique : À mesure que les électrons et les trous diffusent de part et d'autre de la jonction, ils laissent derrière eux des charges immobiles opposées (atomes donneurs positifs côté N et atomes accepteurs négatifs côté P). Ces charges immobiles, qui sont dessinés sur la figure 1.2, créent un champ électrique qui s'oppose à la diffusion des porteurs de charge (les électrons et les trous).

Au fur et à mesure que la diffusion des porteurs de charge se poursuit, le champ électrique créé par les charges immobiles devient de plus en plus fort, jusqu'à ce qu'il s'oppose exactement à la force de diffusion. À ce stade, un équilibre est atteint, et il n'y a plus de mouvement net des porteurs de charge à travers la jonction.

Cet équilibre crée une **zone de déplétion** à la jonction, où il y a très peu de porteurs de charge mobiles. La zone de déplétion sépare les régions P et N et constitue une barrière de potentiel qui empêche le mouvement des porteurs de charge à travers la jonction lorsque la jonction est au repos.

Maintenant que se passe-t-il si l'on applique une tension entre les contacts, à l'extrémité extérieure des régions dopées ?

1.1.3 Application d'une tension positive

Si l'on applique une tension positive en direct : potentiel élevé du côté P et faible du côté N, le champ électrique interne de la diode est réduit par cette différence de potentiel (cf. figure 1.3). Cette baisse du champ E permet aux électrons d'atteindre la région P et aux trous d'atteindre la région N : il y a alors recombinaison. C'est-à-dire que si un électron et un trou se rencontrent, ils s'annulent. Ainsi, au niveau du déplacement de charges, il est possible d'observer un flot continu d'électrons (de charge négative) se dirigeant vers le potentiel élevé et un flot continu de trous (de charge positive) se dirigeant vers le potentiel faible. Il y a un courant de charge présent, la diode est passante.

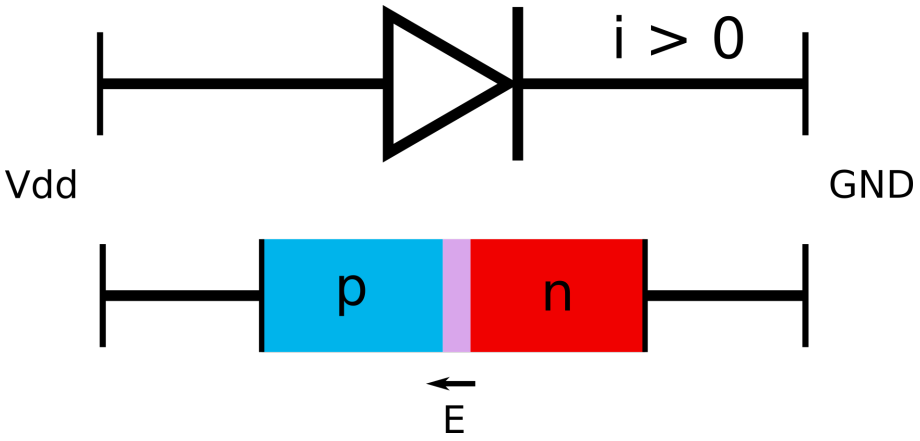


FIGURE 1.3 – Tension positive appliquée sur la jonction PN.

1.1.4 Application d'une tension négative

Si l'on applique au contraire une tension négative, le champ électrique est renforcé comme illustré figure 1.4. Comme le champ électrique est à l'origine de la force s'opposant au déplacement des charges mobiles, les électrons et les trous ne se recombinent plus. En l'absence de courant, la diode est bloquante.

1.2 Les transistors

Le transistor est considéré comme le composant de base de tout circuit intégré. Il en existe de nombreuses sortes et nous allons ici plutôt rappeler le fonctionnement basique du transistor à effet de champ à grille isolée (*metal oxyde semiconductor field effect transistor (MOSFET)*).

Il existe deux types de transistors suivant la disposition des régions P et N, le transistor à canal N ou à canal P. Le transistor est connecté au reste du circuit par 4 contacts : le drain (D), la source (S), la grille (G) et le substrat (B pour *body*).

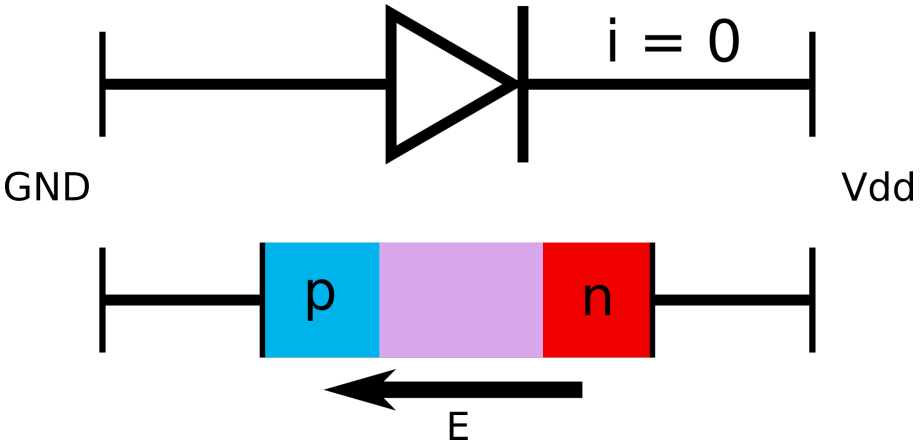


FIGURE 1.4 – Tension négative appliquée sur la jonction PN.

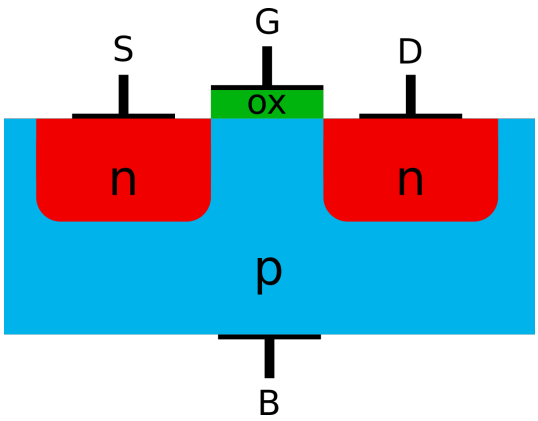


FIGURE 1.5 – Plan de coupe d'un MOSFET à canal N (NMOS) à enrichissement.

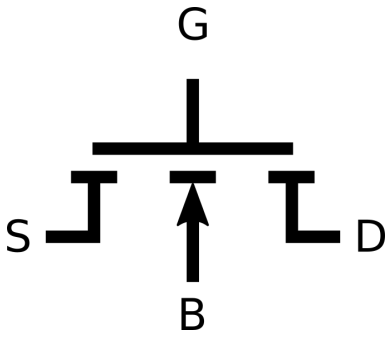


FIGURE 1.6 – Symbole d'un NMOS à enrichissement.

Transistor à canal N Le MOSFET sur la figure 1.5 peut se lire comme un ensemble de structures : on observe deux jonctions PN, entre S et B et entre D et B. Ces jonctions sont polarisées en inverse, les diodes correspondantes sont bloquantes. Autrement dit, lors du fonctionnement normal du transistor, le courant ne fuit pas de la source et du drain vers le substrat. Une autre structure intéressante est le condensateur MOS entre la grille et le substrat dopé P. On rappelle qu'un condensateur est créé par la proximité de deux conducteurs séparés par une couche d'isolant. Ici, c'est l'oxyde qui joue ce rôle d'isolant. Il s'agit en général de SiO^2.

Lorsque la tension de grille V_{GS} est inférieure à la tension de seuil du transistor, le courant I_D passant par le drain est nul puisque les 2 diodes sont bloquantes (cf. figure 1.7). Toutefois, au-delà de cette tension de seuil, il y a accumulation de charges positives sous l'oxyde, du fait du champ électrique généré par le condensateur MOS. Cela crée une zone de continuité des charges sous l'oxyde (cf. figure 1.8) permettant le déplacement des charges et donc l'apparition d'un courant I_D.

Transistor à canal P Les transistors à canal P sont très semblables à ceux à canal N mais inversent les dopages N et P. Ils sont passant lorsque la tension V_{GS} est en dessous de la tension de seuil comme illustré sur les figures 1.11 et 1.12.

Le MOSFET, une vision un peu dépassée Bien que l'exposé traditionnel des transistors MOSFET reste pertinent dans son approche conceptuelle, il est important de souligner que l'implémentation pratique de ces dispositifs a beaucoup évolué avec le temps. Dans le contexte

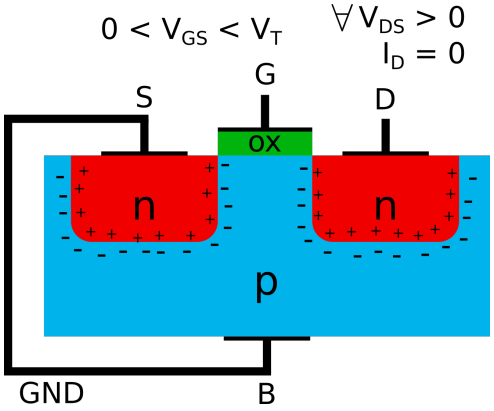


FIGURE 1.7 – NMOS, tension de grille sous le seuil : bloquant.

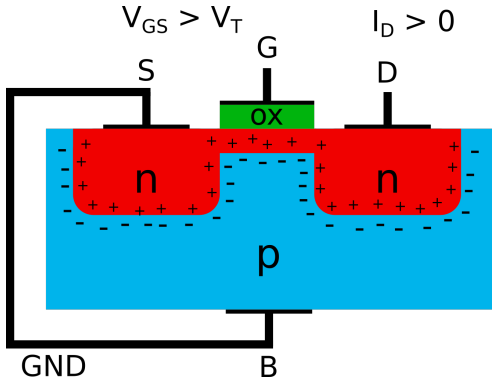


FIGURE 1.8 – NMOS, tension de grille au-dessus du seuil : passant.

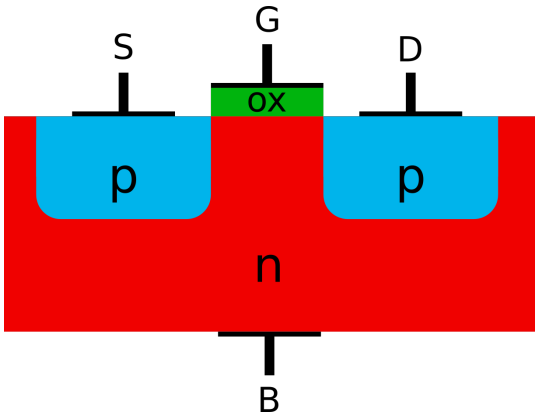


FIGURE 1.9 – Plan de coupe d'un MOSFET à canal P (PMOS) à enrichissement.

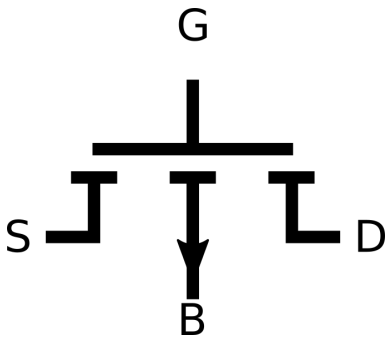


FIGURE 1.10 – Symbole d'un PMOS à enrichissement.

des technologies de fabrication modernes, les MOSFET ont été progressivement remplacés par des architectures plus avancées, telles que les FinFET, les Gate All Around, etc. L'objectif étant de miniaturiser le transistor en diminuant notamment la distance entre drain et source. Une réduction naïve de cette distance empêche le bon fonctionnement du transistor, car le canal n'a plus la bonne longueur pour permettre une bonne isolation entre drain et source. Les FinFET et let Gate All Around ont une structure tridimensionnelle pour permettre de contourner ce souci.

Il existe énormément de variations sur la conception des transistors, et vous ne pouvez supposer du type de transistor utilisé dans une puce sans plus d'information.

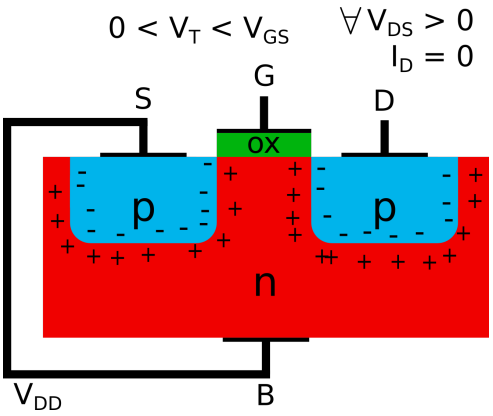


FIGURE 1.11 – PMOS, tension de grille au-dessus du seuil : bloquant.

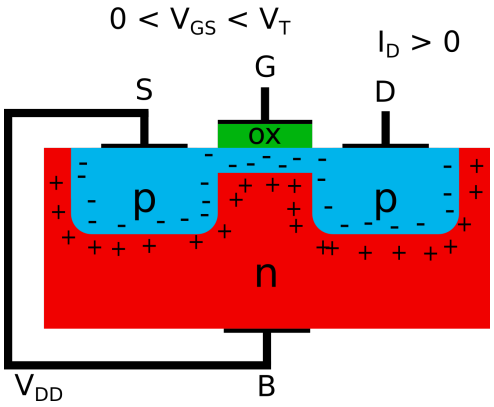
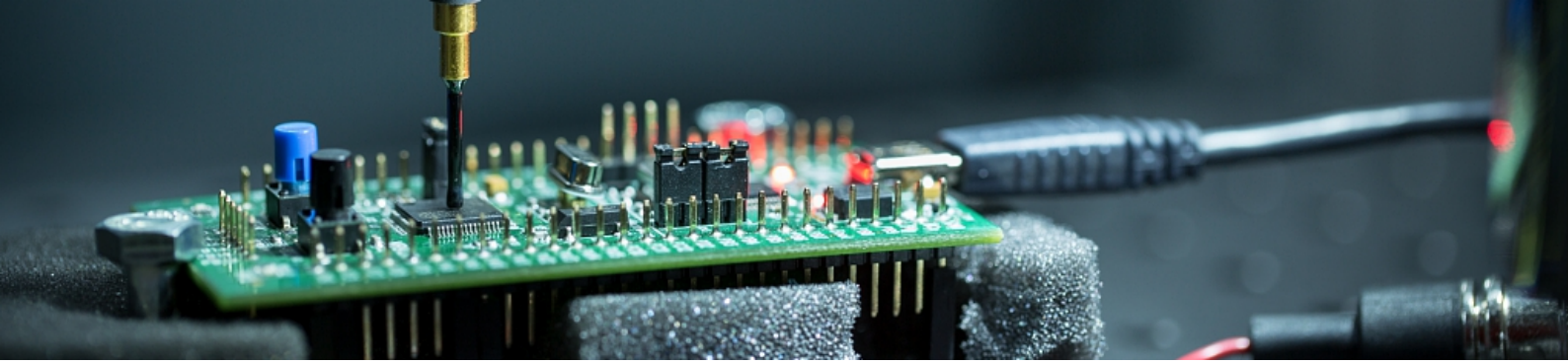


FIGURE 1.12 – PMOS, tension de grille sous le seuil : passant.



2. Les circuits logiques

2.1 Portes logiques

Une porte logique est un circuit utilisant une combinaison de transistor pour réaliser une fonction logique. Dans cette optique, nous considérons qu'un signal à V_{DD} prend une valeur logique 1 est un signal à GND une valeur logique 0.

Dans les schémas suivants, nous utiliserons les symboles simplifiés des transistors MOSFET comme sur les figures 2.1 et 2.2.

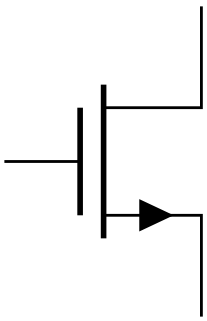


FIGURE 2.1 – Symbole simplifié d'un NMOS. La flêche est parfois omise si le sens du transistor est définis par le contexte (source connectée à la masse notamment).

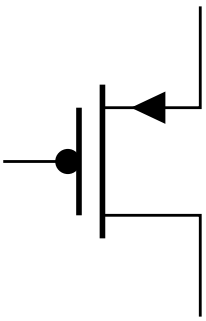


FIGURE 2.2 – Symbole simplifié d'un PMOS. La flêche est parfois omise si le sens du transistor est définis par le contexte (source connectée à l'alimentation notamment).

2.1.1 L'exemple de l'inverseur

Un inverseur, également appelé porte NOT, inverse la valeur logique, comme son nom l'indique. Une valeur 0 en entrée dans un 1 en sortie, et inversement.

Nous pouvons comprendre son fonctionnement à partir de la figure 2.3. Lorsque l'entrée I est à 0, le PMOS est passant et le NMOS bloquant. Ainsi la sortie O est reliée à V_{DD}, sa valeur est à 1. À l'inverse si l'entrée est à 1, le PMOS est bloquant et le NMOS est passant. La sortie est alors à 0, puisque reliée à GND.

Le plan de coupe de la figure 2.4 montre comment sont disposés les régions P et N ainsi que les différentes connections. La porte logique est ainsi une structure comprenant de multiples

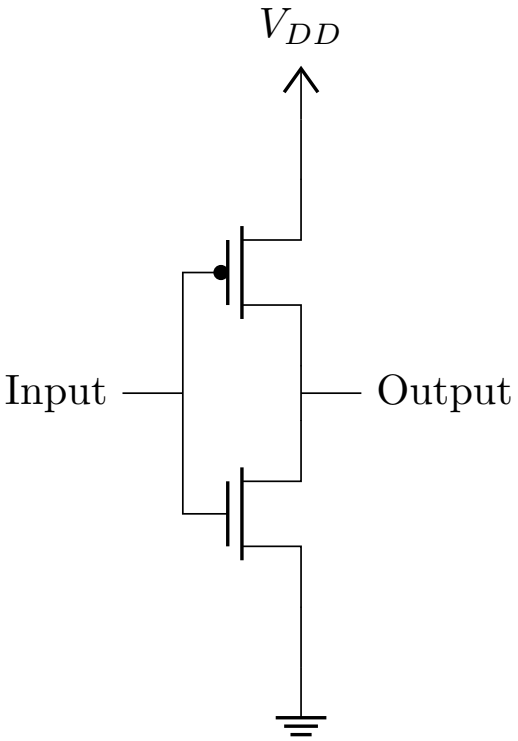


FIGURE 2.3 – La porte NOT est une combinaison de 2 transistors : un PMOS et un NMOS.

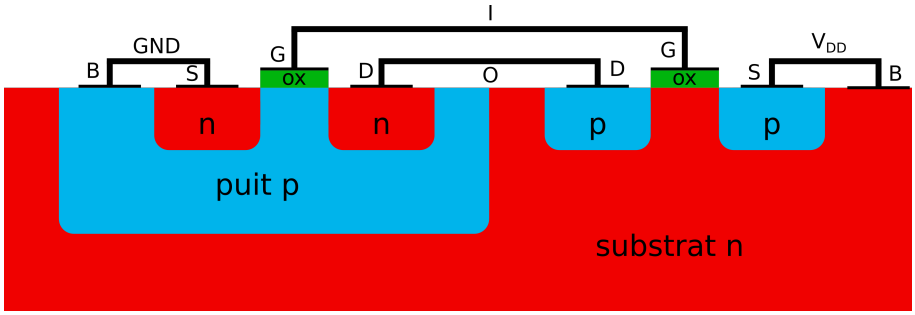


FIGURE 2.4 – Plan de coupe d'une porte NOT, montrant la disposition des matériaux.

jonctions PN et condensateurs MOS.

2.1.2 Les portes logiques courantes

Les autres portes logiques les plus couramment utilisées sont les portes ET (*AND*), OU (*OR*), OU-EXCLUSIF (*XOR*), NON-ET (*NAND*), NON-OU (*NOR*), etc. Leurs symboles et tables de vérité sont illustrés sur la figure 2.5. Certaines portes sont universelles, par exemple la porte NON-ET (*NAND*) : toutes les autres portes logiques peuvent être conçues à l'aide de portes NON-ET.

2.2 Circuits combinatoires

Les portes logiques constituent les fonctions élémentaires à la base de circuits plus complexes : les circuits combinatoires.

Définition 2.1 - *Circuit combinatoire*

Un circuit combinatoire est constitué d'un ensemble de portes logiques reliées entre elles selon les règles suivantes :

- Chaque porte logique n'a qu'un et un seul fil pour chaque entrée logique. Une porte ET a deux entrées logiques par exemple.
- Chaque fil de sortie d'une porte peut être librement dupliqué. Une seule sortie d'une porte peut être reliée à plusieurs entrées logiques. Le nombre d'entrées pour une sortie est appelé le *fan out*.
- Les boucles de rétroaction sont interdites. Un circuit combinatoire est un graphe dirigé acyclique.
- Il n'y a pas de mémoire des états précédents du circuit : pour un ensemble de valeurs d'entrée fixé, les valeurs de sortie sont déterministes.

Exercice 2.1 - *NAND to XOR*

Dessiner le circuit combinatoire, constitué uniquement de portes NAND, ayant la fonctionnalité d'une porte OU-EXCLUSIF. Annoter le chemin critique.

Un circuit combinatoire ne peut pas mémoriser des valeurs dans le temps. Toutefois la traversée d'un circuit combinatoire, c'est à dire la propagation de la modification de la valeur d'une entrée jusqu'à toutes les sorties, prend du temps. La traversée d'une porte logique n'est pas immédiate ni la propagation le long d'un fil. Par approximation, on considère que la vitesse d'un signal électrique est de $c/2$: en 1 ns, le signal parcours 15 cm.

Tout circuit combinatoire possède un chemin de donnée le plus long, prenant en compte les portes logiques traversées ainsi que le temps de propagation du signal sur les fils métalliques. Ce chemin est le **chemin critique**.

Si l'on change une entrée du circuit combinatoire, il faut attendre le temps de propagation du signal le long du chemin critique avant de considérer que la sortie est valide. Avant cela, la sortie peut être instable : elle alterne entre prendre la valeur 0 et 1 au gré des propagations des signaux le long des différents chemins possibles.

2.3 Les circuits séquentiels

Un circuit combinatoire n'évolue que lorsque ses entrées changent. On n'aimerait pourtant souvent que nos données évoluent en fonction d'un résultat précédent. Il faut donc introduire des éléments permettant de sauvegarder les données dans le temps. Ce sont les bascules.

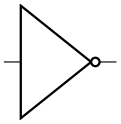
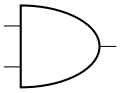
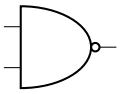
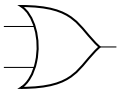
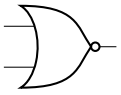
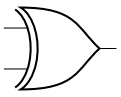
Nom	Nom anglais	Table de vérité	Symbole (américain)
NON	NOT	I | O 0 | 1 1 | 0	
ET	AND	I_1 I_2 | O 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1	
NON-ET	NAND	I_1 I_2 | O 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0	
OU	OR	I_1 I_2 | O 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1	
NON-OU	NOR	I_1 I_2 | O 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0	
OU-EXCLUSIF	XOR	I_1 I_2 | O 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	

FIGURE 2.5 – Les portes logiques, symboles et tables de vérité.

Définition 2.2 - *Circuit séquentiel*

Un circuit séquentiel est un type de circuit logique dont la sortie dépend non seulement des entrées actuelles, comme dans un circuit combinatoire, mais aussi des états passés du circuit.

La mémorisation des états passés est réalisée à l'aide de portes logiques spécifiques, appelées bascules. Un circuit séquentiel peut être **synchrone** ou **asynchrone**. Dans un circuit synchrone, le changement d'état du circuit est déclenché par un signal spécifique appelé *signal d'horloge*. En revanche, dans un circuit asynchrone, il n'y a pas de tel signal, et l'évolution de l'état se fait progressivement à mesure que les signaux se propagent.

2.3.1 Les bascules

Bascules asynchrones Une bascule asynchrone, aussi appelée verrou (*latch* en anglais), est un élément dont la sortie ne dépend uniquement que de la valeur des entrées, mais permettant de faire perdurer un état dans le temps. Par exemple la bascule RS NON-OU est définie sur la figure 2.6. Si les deux signaux S (*set*) et R (*reset*) sont à 0, la sortie perdure dans le temps.

S	R	Q	$\bar{Q}$
0	0	q	$\bar{q}$
0	1	0	1
1	0	1	0
1	1	U	U

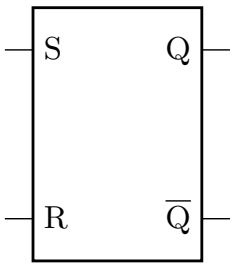


FIGURE 2.6 – Table de vérité et symbole du verrou RS NON-OU. U pour "undefined".

Une telle porte logique a un inconvénient majeur si l'on veut l'utiliser dans un circuit complexe. En effet, les signaux en sortie d'un circuit combinatoire sont instables pour toute durée inférieure au chemin critique. Ainsi si R ou S est la sortie d'un tel circuit combinatoire, la sortie de la bascule RS est déterminée par ces oscillations. Les bascules synchrones sont la réponse courante à ce problème.

Bascules synchrones Les bascules synchrones ont en commun l'utilisation d'un signal spécial : l'**horloge**. Il s'agit d'un signal carré cadençant l'évolution des signaux dans le circuit. Les bascules synchrones mémorisent leur entrée sur le front montant, noté $\uparrow$, du signal d'horloge. La bascule synchrone la plus courante est la bascule D (*D flipflop* en anglais), D pour Data, illustrée sur la figure 2.7.

D	clk	Q	$\bar{Q}$
0	$\uparrow$	0	1
1	$\uparrow$	1	0
X	$\neq\uparrow$	q	$\bar{q}$

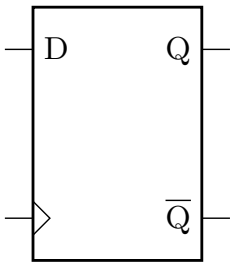


FIGURE 2.7 – Table de vérité et symbole d'une bascule D. X pour "don't care".

Son principe de fonctionnement est simple : l'entrée de la bascule, le signal D, est mémorisée sur le front montant de l'horloge. Cette valeur est maintenue jusqu'au front montant suivant. En pratique, comme nous le verrons dans la sous-section 11.1.2, une bascule n'est pas un composant purement abstrait et sa réalisation physique rajoute des contraintes pour un bon fonctionnement.

2.3.2 La structure des circuits synchrones

Un circuit combinant circuits combinatoires et bascules synchrones est un circuit synchrone. La plupart des puces de la vie courante : microcontrôleurs, processeurs, etc. sont des circuits synchrones.

Ainsi un circuit synchrone a une structure classique, illustrée sur la figure 2.8. En entrée de chaque bascule, nous avons un circuit combinatoire réalisant une fonctionnalité.

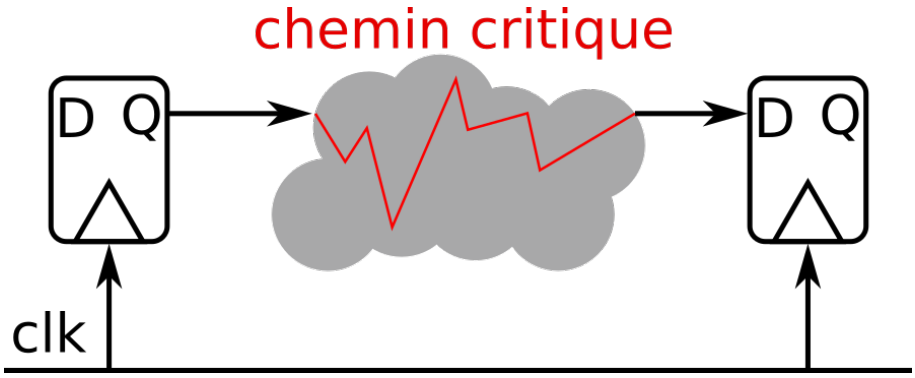
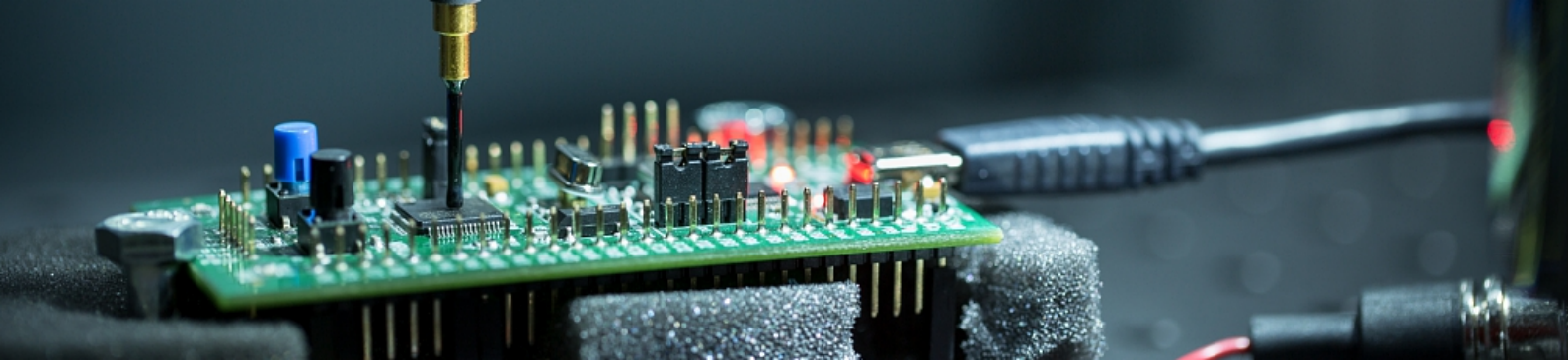


FIGURE 2.8 – Une représentation schématisée d'un circuit synchrone. Le circuit combinatoire est représenté par le nuage gris, et son chemin critique en rouge.



3. La fabrication des circuits intégrés

3.1 Processus de fabrication

La fabrication d'un circuit est un processus extrêmement complexe qui nécessite un environnement contrôlé et exempt de poussière : la salle blanche.

Un grand nombre d'étapes de fabrication ont lieu, permettant de graver une structure tridimensionnelle composée de silicium (dopé ou non), d'oxyde, de métaux, etc. Beaucoup de ces structures sont créées à l'aide de la **photolithographie**.

Le principe est de recouvrir la plaquette de silicium, appelée wafer, d'une couche de résine photosensible, un matériau qui réagit à la lumière. Ensuite, on expose le wafer à une lumière à travers un masque qui contrôle les zones illuminées. Dans le cas de la résine photosensible positive, les zones éclairées deviennent solubles dans le révélateur (un produit chimique spécifique). Cela permet de découvrir certaines régions et pas d'autres. Il est ainsi possible, par exemple, de déposer des matériaux dans les zones découvertes par dépôt chimique en phase vapeur (Chemical Vapor Deposition ou CVD).

Exemple 3.1

Pour une introduction simple au processus de fabrication, voir https://blog.robertelder.org/how-to-make-a-cpu/ ou cette excellente vidéo de Branch Education.

3.2 Le layout

On désigne la disposition spatiale des matériaux sous le terme générique de layout de la puce. La disposition spatiale, combinée avec la technologie de fabrication, est très importante et détermine les performances (vitesse, consommation énergétique ...) de la puce. Le layout est toujours tridimensionnel, et peut être très complexe.

Sur la figure 3.1, est illustré le layout d'une porte NAND. C'est à dire la disposition physique, dans le plan vue du dessus, des différents matériaux. Ces rectangles servent à définir à quels endroits déposer tel ou tel matériau lors du processus de fabrication.

- En bleu les pistes métalliques : VDD, masse en bas, et sortie. Il manque les pistes métalliques des deux entrées, omises car superposées à l'oxyde.
- En vert, l'oxyde isolant la grille.
- En jaune, les zones dopées P.
- En rouge les zones dopées N.

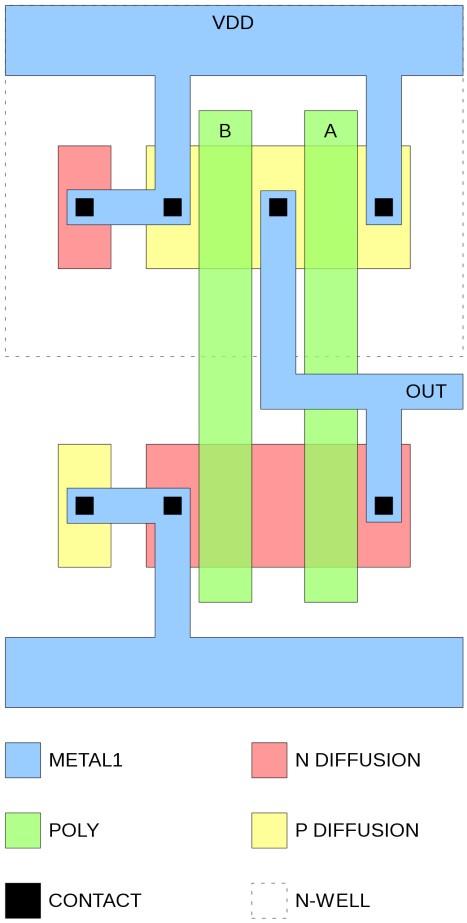


FIGURE 3.1 – Layout d'une porte NAND. *Crédit Jamesm76 sur Wikimedia commons.*

— En noir, les vias établissant un contant entre les différentes couches.

On peut noter que les 2 transistors de type N sont directement connectés en une structure conjointe, une sorte de transistor à deux grilles. Via ce genre de techniques, il est possible de rendre les portes logiques plus compact.

Ce layout correspond au circuit de la figure 3.2.

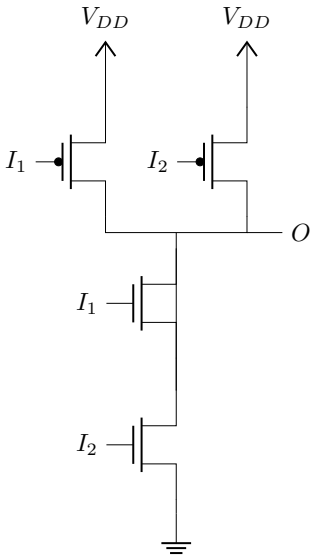


FIGURE 3.2 – Porte NAND à partir de transitors.

II Architectures et microarchitectures

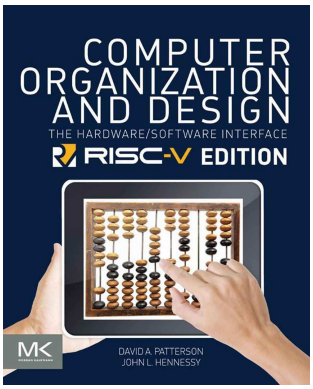
« Je suis l'Architecte. Je suis celui qui a créé la Matrice. Je t'attendais.» L'Architecte dans Matrix Reloaded.

Le jeu d'instructions (ISA) est le standard à l'interface entre le logiciel et le matériel. L'architecture est constituée des éléments définis dans ce standard. La microarchitecture est l'implémentation, sous forme de circuit, du ISA.

Ainsi, par exemple, les instructions `load` et `store`, définies par l'ISA font parties de l'architecture. Les mémoires caches sont un détail de l'implémentation de ces instructions : elles font partie de la microarchitecture.

Après avoir décris en détail les principes derrière le ISA RISC-V, nous verrons différents types de microarchitectures, et de certains composantes essentiels de celles-ci comme la prédiction de branchement.

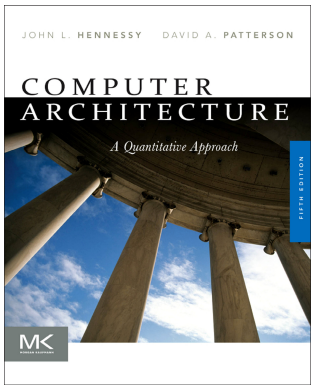
Suggestions de lecture



Computer Organization and Design RISC-V Edition : The Hardware Software Interface [45]

Par *David Patterson, John L. Hennessy*

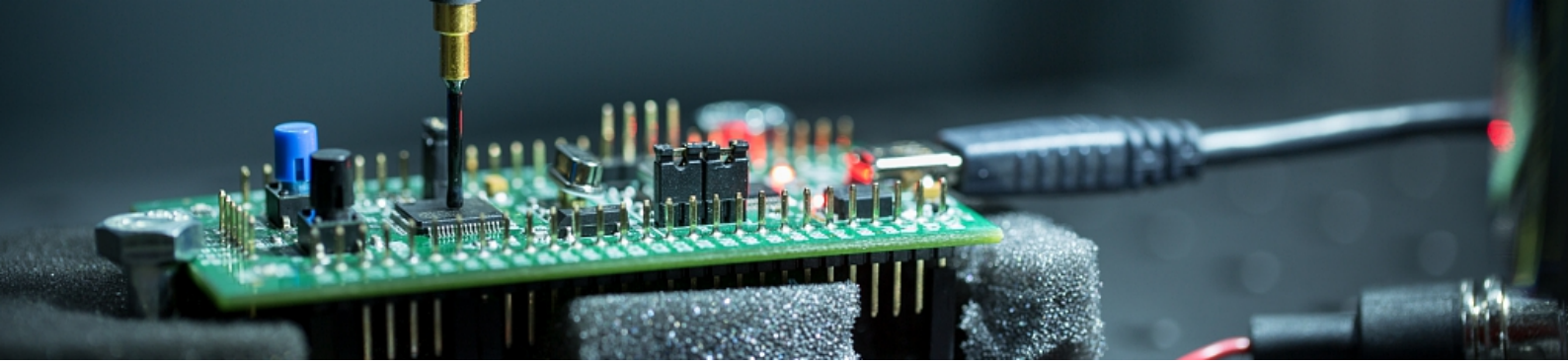
ISBN : 0128122757



Computer Architecture : A Quantitative Approach [43]

Par *John L. Hennessy, David Patterson*

ISBN : 012383872X



4. Le jeu d'instructions RISC-V

Le jeu d'instructions RISC-V est un standard libre, édité par la fondation RISC-V, trouvable simplement en ligne : https://riscv.org/technical/specifications/.

À l'heure de l'écriture de ces lignes, ce standard comprend environ 840 pages et il n'est pas question ici de les paraphrasées. Nous allons décrire l'essentiel, suffisamment pour comprendre le fonctionnement d'un microcontrôleur RISC-V.

Un ISA définit des instructions : des opérations élémentaires, simples, qui seront exécutée par le circuit matériel. Le logiciel écrit par un développeur est compilé en séquences de ces instructions.

RISC-V est un ISA modulaire : il est constitué d'une base RV32I, RV64I ou RV32E et d'un ensemble d'extensions rajoutant des instructions spécifiques.

Pour obtenir un ensemble cohérent de jeu de base plus un ensemble d'extension, RISC-V définit des profils applicatifs (RVA) ou embarqués (RVM) qui peuvent ainsi être ciblés par les compilateurs.

4.1 Les registres

Le jeu d'instructions commence par la définition des registres. Les registres sont de petites mémoires directement utilisables par les instructions.

General purpose registers RISC-V définit 32 general purpose registers (GPRs), noté `x0`, `x1`, $\cdots$, `x31`. Suivant la variante de jeux d'instruction RV32I pour le 32-bit ou RV64I pour le 64-bit, ces registres contiennent 32 ou 64 bits. On nomme `XLEN` cette taille de registre. `XLEN` définit ainsi la taille d'une adresse mémoire et donc la taille de l'espace adressable : 2^{32} octets pour $\texttt{XLEN} = 32$ et 2^{64} octets pour $\texttt{XLEN} = 64$.

Le registres `x0` est particulier : il vaut zéro à la lecture, et l'écriture n'a pas d'effet en général.

Ces registres ont souvent des rôles spécifiques, par convention. Ces rôles sont donnés par le compilateur, et permettent certaines optimisations matérielles. On parle de convention d'appel, ou d'application binary interface (ABI).

Exemple 4.1 - *Return Address*

Le registre `x1` est aussi appelé `ra` pour `return address`, ces deux noms sont donc synonymes et désignent la même mémoire.
L'adresse de retour est l'adresse ou le flot d'exécution doit retourner à la fin de l'exécu-

tion d'une fonction, cf sous-section 4.2.2. Du coup, le matériel peut optimiser la rapidité des retours de fonction à l'aide d'un composant dédié le RSB (cf section 8.2), lorsque le registre utilisé est `ra`.

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–7	t0–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

TABLE 4.1 – RISC-V ABI Register Convention.

Program Counter En plus des 32 registres génériques 'GPRs', nous avons un registre spécial : le program counter (PC), parfois noté `pc` dans ce document. Le program counter (PC) contient l'adresse de la prochaine instruction a exécuter.

Ce registre est initialisé à une adresse qui dépend de l'implémentation matérielle. Lors de l'exécution de chaque instruction, le `pc` est mis à jours avec la bonne valeur : soit l'adresse de l'instruction suivante en mémoire, soit l'adresse de l'instruction destination d'un saut ou d'un branchement. Il n'est pas possible d'écrire directement dans ce registre en RISC-V, `pc` ne peut pas être un argument pour une instruction. Il ne peut être modifié que comme conséquence de l'exécution d'une instruction.

4.2 Les instructions

RISC-V est suffisamment simple pour que nous puissions lister exhaustivement les instructions de base. Celles-ci sont toute de taille 32 bits pour les jeux d'instructions de base RV32I et RV64I.

Les instructions ont, au plus 2 registres sources en entrée (`rs1` et `rs2`) et 1 registre destination (`rd`) en sortie. Certaines instructions permettent en plus de sauvegarder des constantes dans l'instruction elle-même : on parle de valeur **immédiate**.

Par convention, dans la liste des registres en argument, on commence par le registre destination qui recevra le résultat.

Exemple 4.2 - *Instructions RV32I vs RV64I*

Une addition RISC-V est définie par l'instruction `add rd, rs1, rs2`. Cette instruction prends 32 bits en mémoire, quelque soit la taille des registres 32-bit ou 64-bit.

Les bits de poids faibles de l'instruction codent la taille de l'instruction : 11 en poids faible si c'est une instruction 32-bit, la taille de base. Les 5 bits suivants codent l'opération (arithmétique, branchement, etc.), on appelle ces 5 bits l'**opcode**.

Le résultat de l'encodage complet d'une instruction est appelé **code machine** de cette instruction et sa représentation textuelle à destination des humains (e.g. `add t0, a0, a1`) est un **mnémonique**. Ainsi l'opcode est la partie du code machine qui encode la fonctionnalité de cette instruction, mais pas ses arguments.

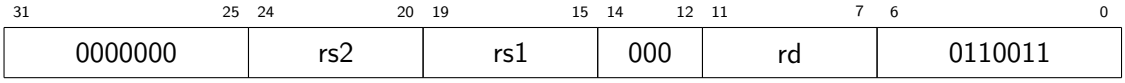
4.2.1 Arithmétique

Les instructions arithmétiques de base du jeu d'instructions RV32I incluent des opérations avec ou sans valeur immédiate. Ces instructions permettent d'effectuer des opérations comme l'addition, la soustraction, et des opérations logiques entre des registres ou entre un registre et une constante immédiate.

4.2.1.1 ADD (Addition)

L'instruction `add rd, rs1, rs2` additionne le contenu de `rs1` et `rs2`, et stocke le résultat dans `rd`.

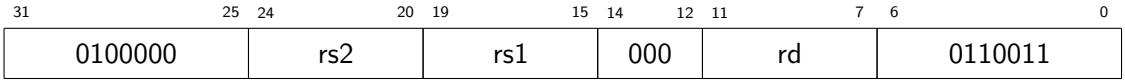
$$\texttt{rd} \leftarrow \texttt{rs1} + \texttt{rs2}$$



4.2.1.2 SUB (Soustraction)

L'instruction `sub rd, rs1, rs2` soustrait le contenu de `rs2` de celui de `rs1`, et stocke le résultat dans `rd`.

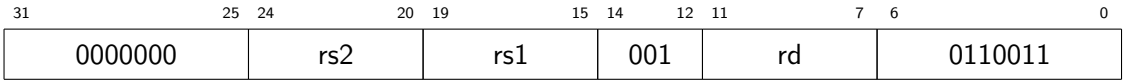
$$\texttt{rd} \leftarrow \texttt{rs1} - \texttt{rs2}$$



4.2.1.3 SLL (Décalage logique à gauche)

L'instruction `sll rd, rs1, rs2` décale le contenu de `rs1` à gauche d'un nombre de bits spécifié par `rs2` et stocke le résultat dans `rd`.

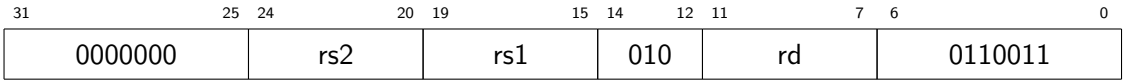
$$\texttt{rd} \leftarrow \texttt{rs1} \ll \texttt{rs2}$$



4.2.1.4 SLT (Set Less Than)

L'instruction `slt rd, rs1, rs2` place 1 dans `rd` si `rs1` est inférieur à `rs2` (interprétés comme des nombres signés), sinon 0.

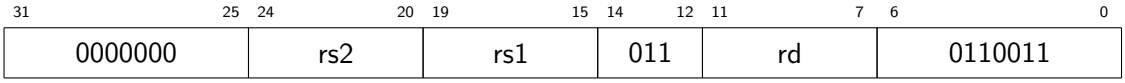
$$\texttt{rd} \leftarrow (\texttt{rs1} < \texttt{rs2})$$



4.2.1.5 SLTU (Set Less Than Unsigned)

L'instruction `sltu rd, rs1, rs2` place 1 dans `rd` si `rs1` est inférieur à `rs2` (interprétés comme des nombres non signés), sinon 0.

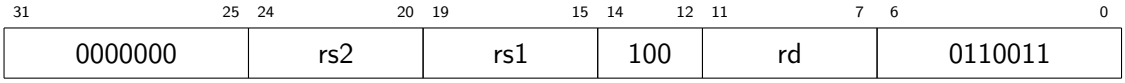
`rd` $\leftarrow$ (`rs1` $<$ `rs2`) (non signé)



4.2.1.6 XOR (Ou exclusif)

L'instruction `xor rd, rs1, rs2` effectue une opération XOR bit à bit entre `rs1` et `rs2` et stocke le résultat dans `rd`.

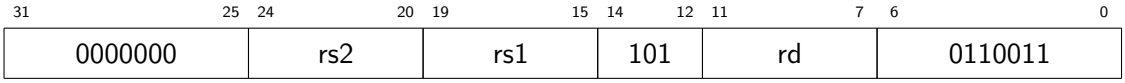
`rd` $\leftarrow$ `rs1` $\oplus$ `rs2`



4.2.1.7 SRL (Décalage logique à droite)

L'instruction `srl rd, rs1, rs2` décale le contenu de `rs1` à droite d'un nombre de bits spécifié par `rs2`, sans extension du signe, et stocke le résultat dans `rd`.

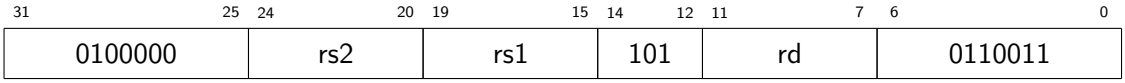
`rd` $\leftarrow$ `rs1` $\gg$ `rs2`



4.2.1.8 SRA (Décalage arithmétique à droite)

L'instruction `sra rd, rs1, rs2` décale le contenu de `rs1` à droite d'un nombre de bits spécifié par `rs2`, en conservant le signe, et stocke le résultat dans `rd`.

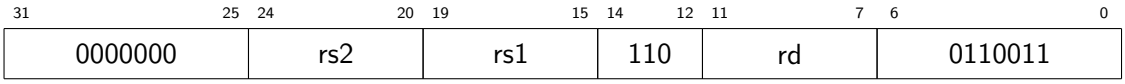
`rd` $\leftarrow$ `rs1` $\gg_s$ `rs2`



4.2.1.9 OR (Ou inclusif)

L'instruction `or rd, rs1, rs2` effectue une opération OR bit à bit entre `rs1` et `rs2`, et stocke le résultat dans `rd`.

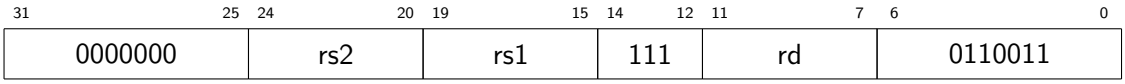
`rd` $\leftarrow$ `rs1` | `rs2`



4.2.1.10 AND (Et)

L'instruction `and rd, rs1, rs2` effectue une opération AND bit à bit entre `rs1` et `rs2`, et stocke le résultat dans `rd`.

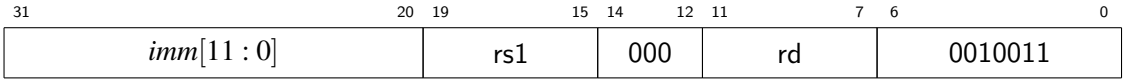
`rd` $\leftarrow$ `rs1`&`rs2`



4.2.1.11 ADDI (Addition immédiate)

L'instruction `addi rd, rs1, imm` ajoute une valeur immédiate sur 12 bits ($imm[11:0]$) au contenu du registre `rs1` et stocke le résultat dans `rd`.

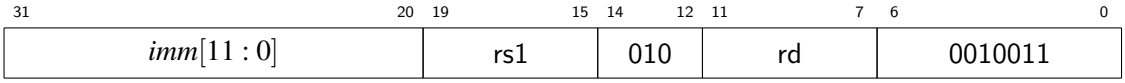
`rd` $\leftarrow$ `rs1` $+ imm$



4.2.1.12 SLTI (Set Less Than immédiate)

L'instruction `slti rd, rs1, imm` compare `rs1` avec l'immédiat interprété comme une valeur signée. Si `rs1` est inférieur à l'immédiat, `rd` est mis à 1, sinon à 0.

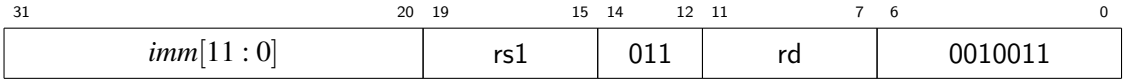
`rd` $\leftarrow$ (`rs1` $< imm$)



4.2.1.13 SLTIU (Set Less Than Unsigned immédiate)

L'instruction `sltiu rd, rs1, imm` fonctionne de la même manière que `slti`, sauf que les registres et l'immédiat sont traités comme des entiers non signés.

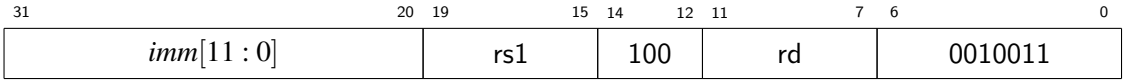
`rd` $\leftarrow$ (`rs1` $< imm$) (non signé)



4.2.1.14 XORI (Ou exclusif immédiate)

L'instruction `xori rd, rs1, imm` effectue une opération XOR entre le contenu de `rs1` et l'immédiat ($imm[11:0]$), et stocke le résultat dans `rd`.

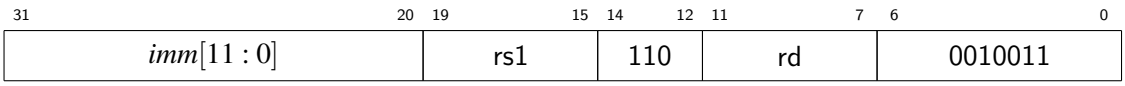
`rd` $\leftarrow$ `rs1` $\oplus imm$



4.2.1.15 ORI (Ou inclusif immédiate)

L'instruction `ori rd, rs1, imm` effectue une opération OR entre `rs1` et l'immédiat ($imm[11:0]$), et stocke le résultat dans `rd`.

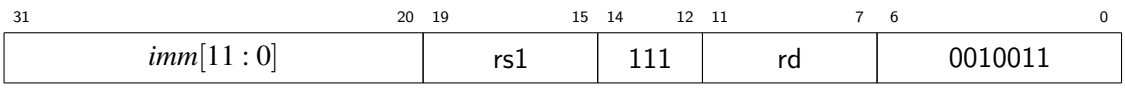
`rd` $\leftarrow$ `rs1` $| imm$



4.2.1.16 ANDI (Et immédiate)

L'instruction `andi rd, rs1, imm` effectue une opération AND bit à bit entre `rs1` et l'immédiat ($imm[11:0]$), et stocke le résultat dans `rd`.

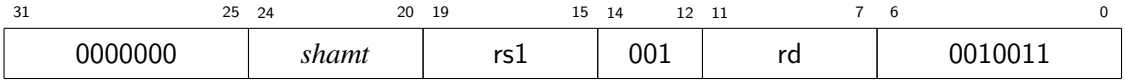
`rd` $\leftarrow$ `rs1` $\& imm$



4.2.1.17 SLLI (Décalage logique à gauche immédiate)

L'instruction `slli rd, rs1, shamt` décale à gauche le contenu de `rs1` par un nombre de bits spécifié par `shamt` ($imm[4:0]$) et stocke le résultat dans `rd`.

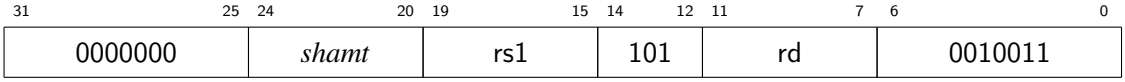
$$\texttt{rd} \leftarrow \texttt{rs1} \ll shamt$$



4.2.1.18 SRLI (Décalage logique à droite immédiate)

L'instruction `srli rd, rs1, shamt` décale logiquement à droite le contenu de `rs1` par un nombre de bits spécifié par `shamt` ($imm[4:0]$) et stocke le résultat dans `rd`.

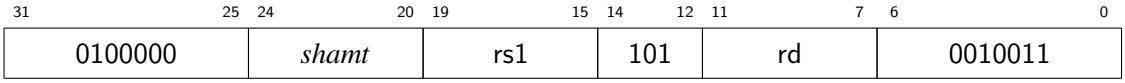
$$\texttt{rd} \leftarrow \texttt{rs1} \gg shamt$$



4.2.1.19 SRAI (Décalage arithmétique à droite immédiate)

L'instruction `srai rd, rs1, shamt` décale arithmétiquement à droite le contenu de `rs1` par un nombre de bits spécifié par `shamt` ($imm[4:0]$) et conserve le signe, puis stocke le résultat dans `rd`.

$$\texttt{rd} \leftarrow \texttt{rs1} \gg_s shamt$$



4.2.2 Sauts

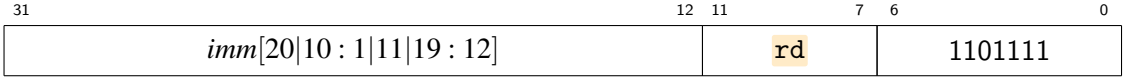
Les instructions de sauts permettent de modifier la séquence d'exécution des instructions en changeant la valeur du PC. Ces instructions incluent les sauts inconditionnels avec et sans mise à jour de l'adresse de retour.

4.2.2.1 JAL (Jump and Link)

L'instruction `jal rd, imm` effectue un saut vers l'adresse cible calculée en ajoutant l'immédiat `imm` au PC actuel. Elle stocke l'adresse de retour dans `rd`. Si toutes les instructions sont de taille 32-bit, l'adresse de retour est décalée de 4 octets. Si les instructions peuvent avoir des tailles variables (avec l'extension C notamment), ce décalage doit pointer vers l'instruction suivante. Si l'on a pas besoin de sauvegarder l'adresse de retour, il suffit de mettre `x0` en `rd`.

$$\texttt{rd} \leftarrow \texttt{pc} + 4$$

$$\texttt{pc} \leftarrow \texttt{pc} + imm$$

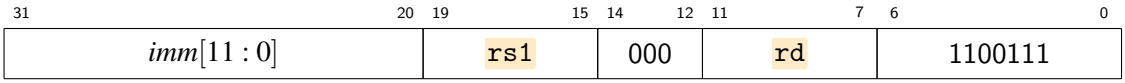


4.2.2.2 JALR (Jump and Link Register)

L'instruction `jalr rd, rs1, imm` calcule l'adresse de saut en ajoutant l'immédiat `imm` au contenu du registre `rs1`, puis saute à cette nouvelle adresse, tout en mettant à jour le registre `rd` avec `pc` + 4.

$$\texttt{rd} \leftarrow \texttt{pc} + 4$$

$$\texttt{pc} \leftarrow (\texttt{rs1} + imm) \ \& \ 0xFFFFFFFE$$



4.2.3 Branchements

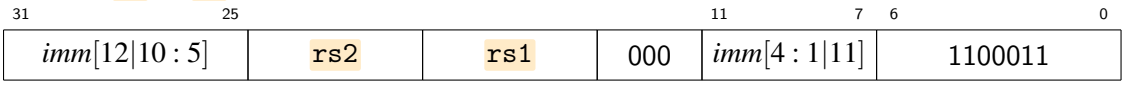
Les instructions de branchements conditionnels permettent de changer la séquence d'exécution en fonction de la comparaison entre deux registres. Si la condition est remplie, le saut s'effectue vers l'adresse calculée, sinon l'exécution continue normalement. On remarque que la valeur immédiate n'a pas de bit de poids faible $imm[0]$, en effet cette valeur est forcément égale à 0 puisque la taille minimale d'instruction est de 2 octets avec un alignement de 2 octets obligatoire.

4.2.3.1 BEQ (Branch if Equal)

L'instruction `beq rs1, rs2, imm` effectue un saut si les valeurs dans `rs1` et `rs2` sont égales.

$$\texttt{si } \texttt{rs1} = \texttt{rs2}$$

$$\texttt{pc} \leftarrow \texttt{pc} + imm$$

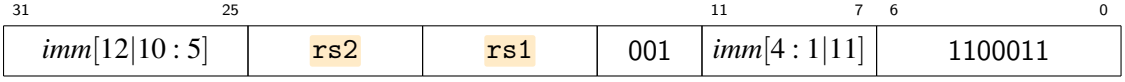


4.2.3.2 BNE (Branch if Not Equal)

L'instruction `bne rs1, rs2, imm` effectue un saut si les valeurs dans `rs1` et `rs2` ne sont pas égales.

$$\texttt{si } \texttt{rs1} \neq \texttt{rs2}$$

$$\texttt{pc} \leftarrow \texttt{pc} + imm$$

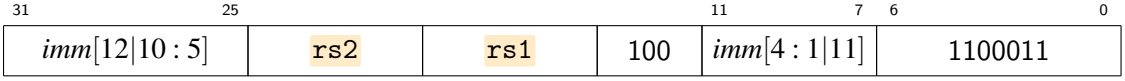


4.2.3.3 BLT (Branch if Less Than)

L'instruction `blt rs1, rs2, imm` effectue un saut si la valeur de `rs1` est inférieure à celle de `rs2` en interprétant les valeurs comme des entiers signés.

$$\texttt{si } \texttt{rs1} < \texttt{rs2}$$

$$\texttt{pc} \leftarrow \texttt{pc} + imm$$

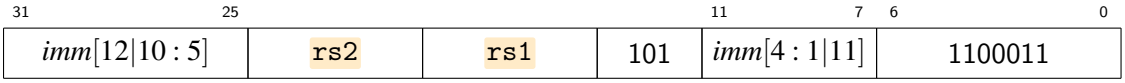


4.2.3.4 BGE (Branch if Greater or Equal)

L'instruction `bge rs1, rs2, imm` effectue un saut si la valeur de `rs1` est supérieure ou égale à celle de `rs2` en interprétant les valeurs comme des entiers signés.

si `rs1` $\geq$ `rs2`

`pc` $\leftarrow$ `pc` $+ imm$

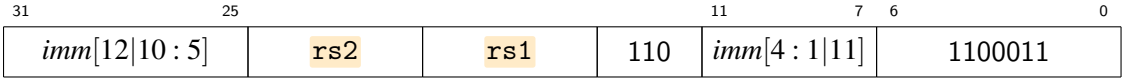


4.2.3.5 BLTU (Branch if Less Than Unsigned)

L'instruction `bltu rs1, rs2, imm` effectue un saut si la valeur de `rs1` est inférieure à celle de `rs2` en les interprétant comme des entiers non signés.

si `rs1` $<$ `rs2` (non signé)

`pc` $\leftarrow$ `pc` $+ imm$

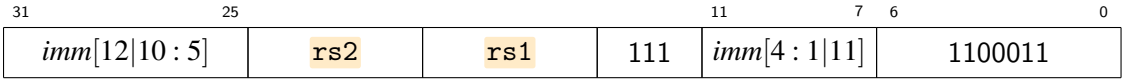


4.2.3.6 BGEU (Branch if Greater or Equal Unsigned)

L'instruction `bgeu rs1, rs2, imm` effectue un saut si la valeur de `rs1` est supérieure ou égale à celle de `rs2`, en les interprétant comme des entiers non signés.

si `rs1` $\geq$ `rs2` (non signé)

`pc` $\leftarrow$ `pc` $+ imm$



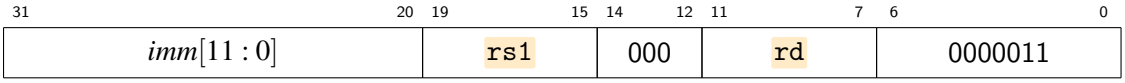
4.2.4 Accès mémoires

Les instructions d'accès mémoire permettent de charger des données depuis la mémoire random-access memory (RAM) vers un registre (`load`) ou de stocker des données d'un registre vers la mémoire (`store`). Les adresses sont calculées en ajoutant un décalage immédiat à un registre source, généralement un pointeur.

4.2.4.1 LB (Load Byte)

L'instruction `lb rd, imm(rs1)` charge un octet signé à partir de la mémoire et l'étend en un mot signé avant de le stocker dans `rd`.

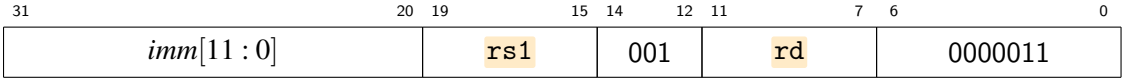
`rd` $\leftarrow$ `mem`$[$`rs1` $+ imm]$



4.2.4.2 LH (Load Halfword)

L'instruction `lh rd, imm(rs1)` charge un demi-mot signé (16 bits) à partir de la mémoire et l'étend en un mot signé avant de le stocker dans `rd`.

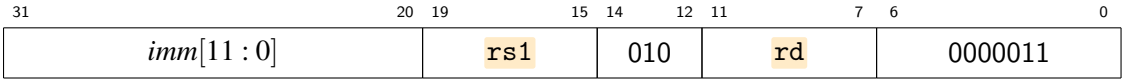
$\texttt{rd} \leftarrow \texttt{mem}[\texttt{rs1} + imm]$



4.2.4.3 LW (Load Word)

L'instruction `lw rd, imm(rs1)` charge un mot signé (32 bits) à partir de la mémoire et le stocke dans `rd`.

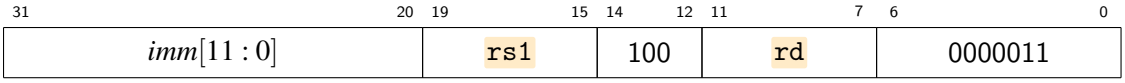
$\texttt{rd} \leftarrow \texttt{mem}[\texttt{rs1} + imm]$



4.2.4.4 LBU (Load Byte Unsigned)

L'instruction `lbu rd, imm(rs1)` charge un octet non signé à partir de la mémoire, l'étend en un mot non signé, puis le stocke dans `rd`.

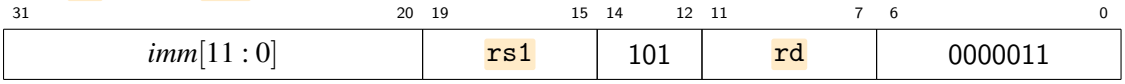
$\texttt{rd} \leftarrow \texttt{mem}[\texttt{rs1} + imm]$



4.2.4.5 LHU (Load Halfword Unsigned)

L'instruction `lhu rd, imm(rs1)` charge un demi-mot non signé (16 bits) à partir de la mémoire, l'étend en un mot non signé, puis le stocke dans `rd`.

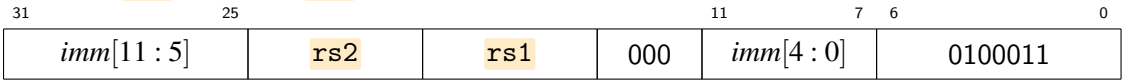
$\texttt{rd} \leftarrow \texttt{mem}[\texttt{rs1} + imm]$



4.2.4.6 SB (Store Byte)

L'instruction `sb rs2, imm(rs1)` stocke un octet provenant de `rs2` dans la mémoire à l'adresse calculée en ajoutant l'immédiat `imm` à `rs1`.

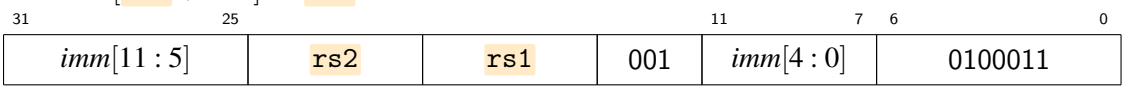
$\texttt{mem}[\texttt{rs1} + imm] \leftarrow \texttt{rs2}$



4.2.4.7 SH (Store Halfword)

L'instruction `sh rs2, imm(rs1)` stocke un demi-mot provenant de `rs2` dans la mémoire à l'adresse calculée en ajoutant l'immédiat `imm` à `rs1`.

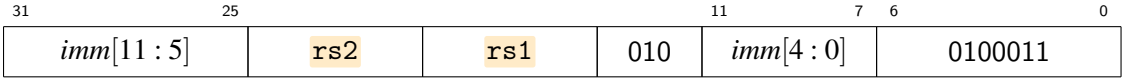
$\texttt{mem}[\texttt{rs1} + imm] \leftarrow \texttt{rs2}$



4.2.4.8 SW (Store Word)

L'instruction `sw rs2, imm(rs1)` stocke un mot provenant de `rs2` dans la mémoire à l'adresse calculée en ajoutant l'immédiat `imm` à `rs1`.

$\texttt{mem}[\texttt{rs1} + imm] \leftarrow \texttt{rs2}$



4.2.5 Accès aux CSRs

Les control and status registers (CSRs) sont des registres spéciaux utilisés pour configurer et interagir avec la configuration du matériel dans RISC-V. Ils permettent de contrôler des fonctions telles que le mode utilisateur ou superviseur, de vérifier l'état des interruptions, ou de gérer d'autres paramètres système. Les instructions qui manipulent ces registres permettent de lire et d'écrire dans les CSRs.

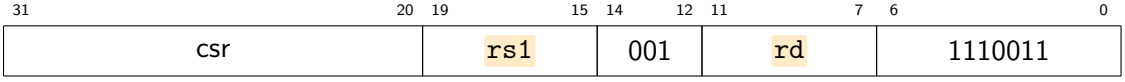
Les instructions d'accès aux CSRs incluent des opérations pour lire, écrire et modifier ces registres de manière atomique, soit avec des registres source (`rs1`), soit avec une valeur immédiate (`uimm`).

4.2.5.1 CSRRW (Atomic Read/Write CSR)

L'instruction `csrrw rd, csr, rs1` lit le contenu d'un CSR spécifié et l'écrit dans `rd`. Ensuite, elle met à jour le CSR avec la valeur de `rs1`.

$\texttt{rd} \leftarrow \texttt{csr}$

$\texttt{csr} \leftarrow \texttt{rs1}$

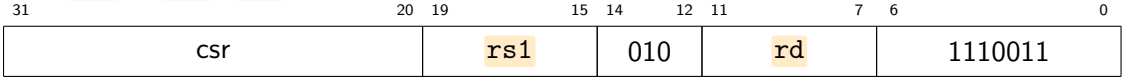


4.2.5.2 CSRRS (Atomic Read and Set CSR)

L'instruction `csrrs rd, csr, rs1` lit le contenu d'un CSR spécifié et l'écrit dans `rd`. Ensuite, elle met à jour le CSR en effectuant un OR bit à bit avec la valeur de `rs1`. Si `rs1` est zéro, le CSR n'est pas modifié.

$\texttt{rd} \leftarrow \texttt{csr}$

$\texttt{csr} \leftarrow \texttt{csr} \mid \texttt{rs1}$

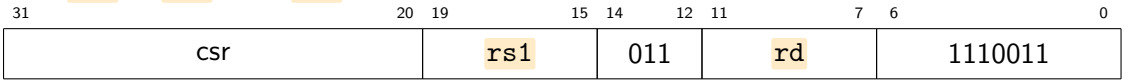


4.2.5.3 CSRRC (Atomic Read and Clear CSR)

L'instruction `csrrc rd, csr, rs1` lit le contenu d'un CSR spécifié et l'écrit dans `rd`. Ensuite, elle met à jour le CSR en effectuant un AND avec l'inverse bit à bit de `rs1`. Si `rs1` est zéro, le CSR n'est pas modifié.

$\texttt{rd} \leftarrow \texttt{csr}$

$\texttt{csr} \leftarrow \texttt{csr} \ \& \ \sim \texttt{rs1}$

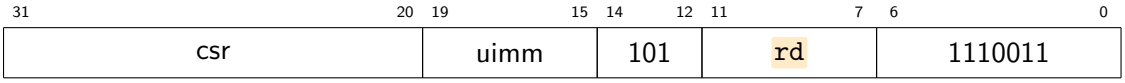


4.2.5.4 CSRRWI (Atomic Read/Write Immediate)

L'instruction `csrrwi rd, csr, uimm` lit le contenu d'un CSR spécifié et l'écrit dans `rd`. Ensuite, elle met à jour le CSR avec une valeur immédiate non signée (`uimm`).

`rd` ← `csr`

`csr` ← `uimm`

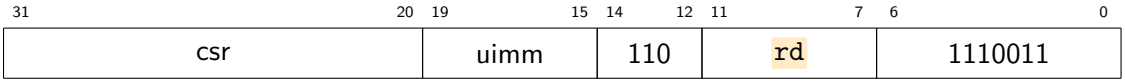


4.2.5.5 CSRRSI (Atomic Read and Set Immediate)

L'instruction `csrrsi rd, csr, uimm` lit le contenu d'un CSR spécifié et l'écrit dans `rd`. Ensuite, elle met à jour le CSR en effectuant un OR bit à bit avec la valeur immédiate non signée `uimm`. Si `uimm` est zéro, le CSR n'est pas modifié.

`rd` ← `csr`

`csr` ← `csr` | `uimm`

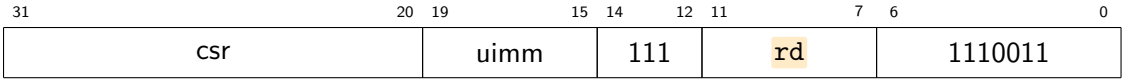


4.2.5.6 CSRRCI (Atomic Read and Clear Immediate)

L'instruction `csrrci rd, csr, uimm` lit le contenu d'un CSR spécifié et l'écrit dans `rd`. Ensuite, elle met à jour le CSR en effectuant un AND avec l'inverse bit à bit de la valeur immédiate non signée `uimm`. Si `uimm` est zéro, le CSR n'est pas modifié.

`rd` ← `csr`

`csr` ← `csr` & ∼ `uimm`



4.2.6 Autres

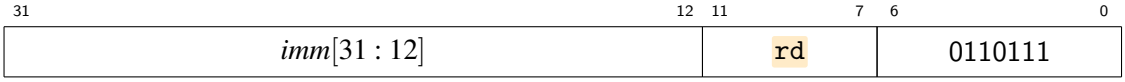
En plus des instructions arithmétiques, de branchements, de sauts, d'accès mémoire et de CSRs, le jeu d'instructions RV32I inclut diverses autres instructions utiles pour le contrôle de programme, la gestion d'exceptions et l'accès au matériel.

4.2.6.1 LUI (Load Upper Immediate)

L'instruction `lui rd, imm` charge une constante immédiate dans les 20 bits de poids fort du registre de destination `rd`, les 12 bits de poids faible sont mis à zéro.

$$\texttt{rd} \leftarrow imm \ll 12$$

Cette instruction est souvent utilisée pour charger une adresse mémoire ou une grande constante dans un registre en combinaison avec une autre instruction d'addition immédiate.

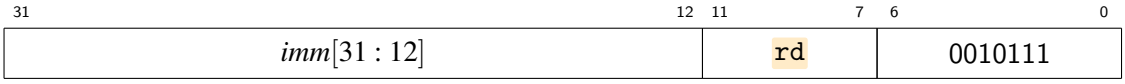


4.2.6.2 AUIPC (Add Upper Immediate to PC)

L'instruction `auipc rd, imm` ajoute une constante immédiate à la valeur actuelle du compteur de programme (`pc`) et stocke le résultat dans `rd`.

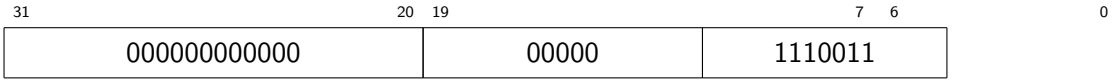
$$\texttt{rd} \leftarrow \texttt{pc} + imm$$

Cette instruction est principalement utilisée pour calculer des adresses relatives au `pc`, notamment pour les sauts ou les accès mémoire.



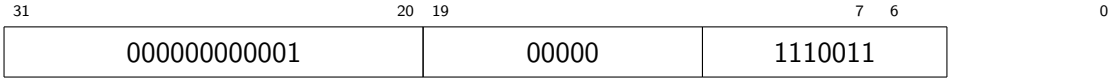
4.2.6.3 ECALL (Environment Call)

L'instruction `ecall` est utilisée pour effectuer un appel au système d'exploitation ou à un environnement d'exécution, en déclenchant une exception. Elle est généralement utilisée pour demander des services systèmes comme les entrées/sorties ou la gestion de la mémoire.



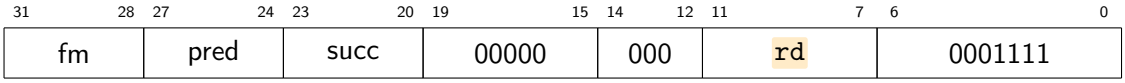
4.2.6.4 EBREAK (Environment Break)

L'instruction `ebreak` déclenche une exception pour indiquer un point d'arrêt. Cette instruction est généralement utilisée par les débogueurs pour suspendre l'exécution d'un programme et analyser l'état du système.



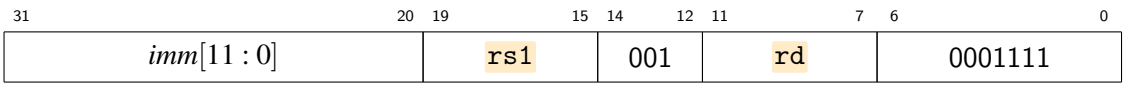
4.2.6.5 FENCE (Data Memory Fence)

L'instruction `fence` est utilisée pour imposer un ordre de synchronisation entre les opérations de mémoire. Elle garantit que les opérations mémoire précédant l'instruction `fence` dans le programme sont complétées avant que celles la suivant ne commencent. Cette instruction est particulièrement utile dans les environnements multi-cœurs ou multi-thread.



4.2.6.6 FENCE.I (Instruction Fence)

L'instruction `fence.i` garantit que toutes les instructions qui suivent cette barrière de synchronisation dans le programme sont récupérées après que toutes les opérations d'écriture dans la mémoire d'instructions précédentes soient complétées. Elle est utilisée pour assurer la cohérence des caches d'instructions après la modification du code en mémoire.



4.2.6.7 Pseudo-instructions RISC-V

Les pseudo-instructions sont des instructions qui simplifient la programmation en assembleur, ou sa lisibilité, mais qui ne correspondent pas directement à une seule instruction machine. Elles sont généralement traduites en une ou plusieurs instructions machine par l'assembleur.

Voici un liste non-exhaustive des pseudo-instructions les plus courantes, et leur traduction en véritable instruction, ou séquence d'instruction.

Pseudo-instruction	Séquence d'instructions	Description

`li rd, imm`	`addi rd, x0, imm`
	(ou, si `imm` est trop grand)
	`lui rd, upper_20(imm)`
	`addi rd, rd, lower_12(imm)`
		Charge un immédiat dans `rd`.
		
		Utilise deux instructions
		si l'immédiat est sur 32 bits.

`mv rd, rs1`	`addi rd, rs1, 0`	Copie le contenu de `rs1` dans `rd`.
`nop`	`addi x0, x0, 0`	Ne fait rien, instruction vide.
`neg rd, rs1`	`sub rd, x0, rs1`	Met dans `rd` l'opposé de `rs1`.
`not rd, rs1`	`xori rd, rs1, -1`	Inverse chaque bit du contenu de `rs1`.
`seqz rd, rs1`	`sltiu rd, rs1, 1`	Met à 1 si `rs1` est égal à zéro, sinon 0.
`blez rs1, label`	`bge x0, rs1, label`	Branche si `rs1` est inférieur ou égal à 0.
`bgtz rs1, label`	`blt x0, rs1, label`	Branche si `rs1` est strictement supérieur à 0.
`la rd, label`	`auipc rd, upper_20(label)` `addi rd, rd, lower_12(label)`	Charge l'adresse d'une étiquette dans `rd`.
`j label`	`jal x0, label`	Saut inconditionnel à une étiquette.
`ret`	`jalr x0, ra`	Retour d'une fonction à l'adresse sauvegardée dans `ra`.

4.3 Du programme à l'exécution : comment ça marche ?

Après cette présentation un peu abstraite, regardons un exemple un peu plus concret. Nous allons ici voir comment coder, compiler et exécuter une fonction en C qui retourne la valeur de l'élément maximum d'un tableau, ou une valeur d'erreur en cas de tableau vide.

```c
#include <stddef.h>
#include <limits.h>

int find_max(int *array, size_t size) {
    if (size == 0) return INT_MIN;
    int max = array[0];
    for (size_t i = 1; i < size; ++i) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

FIGURE 4.1 – Code C permettant retourner l'élément maximum.

Cette fonction commence par vérifier que le tableau n'est pas vide, ou bien elle retourn

un code d'erreur `INT_MIN`. Puis elle initie le max à la valeur du premier élément, qui existe puisque le tableau n'est pas vide. Dans une boucle, elle va ensuite parcourir tous les éléments du tableau. Un élément prends la place du max s'il lui est supérieur. Enfin la valeur max est retournée.

Ce code C est compilé à l'aide du compilateur `clang` dans sa version 18.1.0 avec les optimisations `O3` activées. Le résultat de cette compilation est présenté figure 4.2.

```
1   find_max:
2           beqz    a1, .LBB0_6
3           lw      a3, 0(a0)
4           addi    a1, a1, -1
5           beqz    a1, .LBB0_8
6           addi    a2, a0, 4
7           j       .LBB0_4
8   .LBB0_3:
9           addi    a1, a1, -1
10          addi    a2, a2, 4
11          mv      a3, a0
12          beqz    a1, .LBB0_7
13  .LBB0_4:
14          lw      a0, 0(a2)
15          blt     a3, a0, .LBB0_3
16          mv      a0, a3
17          j       .LBB0_3
18  .LBB0_6:
19          lui     a0, 524288
20  .LBB0_7:
21          ret
22  .LBB0_8:
23          mv      a0, a3
24          ret
```

FIGURE 4.2 – Code assembleur RISC-V correspondant à la recherche de l'élément maximum, issu de la compilation du code de la figure 4.1 par `clang`.

Analysons ensemble le code assembleur optimisé de la figure 4.2, en le déroulant comme la machine le ferait.

Les labels sont les identifiants suivis de " :", ils permettent de nommer une adresse mémoire. Mais ce n'est pas une instruction ! Ainsi `.LBB0_3` est le nom de l'adresse de l'instruction 8. Le branchement à l'instruction 15 branche donc à l'instruction 8 s'il est pris.

Les arguments de la fonction, c'est-à-dire l'adresse du tableau `array` ainsi que sa taille, sont sauvegardés dans les registres arguments de la fonction `a0` et `a1` respectivement.

4.3.1 Analyse du code

Voici une analyse ligne par ligne du code assembleur de la figure 4.2.

2. L'instruction, ligne 2, `beqz a1, .LBB0_6` est une branche conditionnelle qui vérifie si `a1` (qui contient la taille du tableau `size`) est égale à zéro. Si c'est le cas, elle branche vers l'adresse `.LBB0_6`, ce qui correspond au cas où la taille est 0, et la fonction renverra la plus petite valeur possible (`INT_MIN`).
3. À la ligne 3, l'instruction `lw a3, 0(a0)` charge la première valeur du tableau (`a0` contient l'adresse du tableau `array`) dans `a3`, qui est utilisée pour stocker la valeur maximale (`max`) lors de la comparaison des éléments.

4. À la ligne 4, `addi a1, a1, -1` décrémente `a1`, ce qui correspond à la diminution du compteur de la taille du tableau. Cela prépare la boucle de comparaison pour qu'elle traite les éléments restants. On va comparer les éléments en partant de la fin.
5. L'instruction suivante, ligne 5, `beqz a1, .LBB0_8`, vérifie si `a1` est égal à zéro après décrémentation. Si c'est le cas, cela signifie qu'il n'y a plus d'éléments à comparer, et la branche redirige vers `.LBB0_8` pour retourner la valeur maximale trouvée.
6. À la ligne 6, l'instruction `addi a2, a0, 4` incrémente l'adresse de base du tableau dans `a2` pour pointer vers l'élément suivant du tableau. Cela prépare les comparaisons suivantes, en parcourant le tableau élément par élément.
7. À la ligne 7, `j .LBB0_4` est une instruction de saut inconditionnel qui redirige l'exécution vers l'adresse `.LBB0_4`, où la boucle de comparaison commence.
9. À la ligne 9, on décrémente le compteur qui parcours tous les éléments.
10. L'instruction suivante ligne 10, elle, incrémente de 4 l'adresse de l'élément à lire.
11. À la ligne 11, `mv a3, a0` sauvegarde l'adresse du tableau dans `a3`.
12. L'instruction suivante ligne 12, teste si `a1` la valeur du compteur est égale à 0. Si c'est le cas, tous les éléments ont été parcourus, et on saute à la fin du programme `.LBB0_7`.
14. On lit en mémoire la valeur du tableau à tester à la ligne 14 à l'aide de `lw` et met le résultat dans `a0`.
15. L'instruction ligne 15 test si la valeur dans `a3` (le max courant) est strictement plus petite que la valeur dans `a0`. Si c'est le cas, il n'y a rien à faire et on recommence la boucle à `.LBB0_3`. Sinon on continue pour mettre à jour le max.
16. Ligne 16, la valeur de l'élément qui vient d'être lu est mise dans `a3` qui contient la valeur max.
17. Saut inconditionnel vers une nouvelle itération de la boucle à `.LBB0_3`.
19. À la ligne 19, l'instruction `lui a0, 524288` dans `.LBB0_6` est exécutée si la taille du tableau est 0. Elle charge la constante `INT_MIN` dans `a0`, car il n'y a aucun élément dans le tableau à comparer.
21. L'instruction suivante, ligne 21, `ret` dans `.LBB0_7` termine la fonction et retourne la valeur dans `a0`, qui contient soit la valeur maximale trouvée, soit `INT_MIN` si la taille du tableau est 0.
23. Enfin, à la ligne 23, `mv a0, a3` dans `.LBB0_8` copie la valeur maximale trouvée (`a3`) dans `a0`, juste avant que la fonction ne retourne cette valeur comme résultat à la ligne 24.

Exercice 4.1 - *Assembleur RISC-V*

Écrire le code assembleur RISC-V qui correspond au code C suivant :

```c
void swap(int* a, int* b) {
    int c = *a;
    *a = *b;
    *b = c;
}
```

4.4 Lire un listing

Un listing est le code décompilé (ou désassemblé) à partir d'un fichier binaire (comme un fichier .elf sous Linux, ou .dll ou .exe sous Windows).

Exemple 4.3 - *objdump*

Pour désassembler un binaire .elf, on utilise l'outil `objdump` de la suite GCC :

```
> objdump -S -d binary.elf > binary.listing
```

Attention, il est important d'utiliser la bonne chaîne de compilation dans le cas d'une compilation croisée. Par exemple, pour RISC-V :

```
> riscv64-none-elf-objdump -S -d binary.elf > binary.listing
```

Un extrait de code désassemblé est présenté en figure 4.3.

```
0000000080000368 <linear_position>:
80000368:   0055579b      srliw   a5,a0,0x5
8000036c:   7f87f793      andi    a5,a5,2040
80000370:   1141          addi    sp,sp,-16
80000372:   953e          add     a0,a0,a5
80000374:   0ff57513      zext.b  a0,a0
80000378:   0141          addi    sp,sp,16
8000037a:   8082          ret
```

FIGURE 4.3 – Extrait d'un listing issu de la décompilation d'un binaire (.elf) RISC-V.

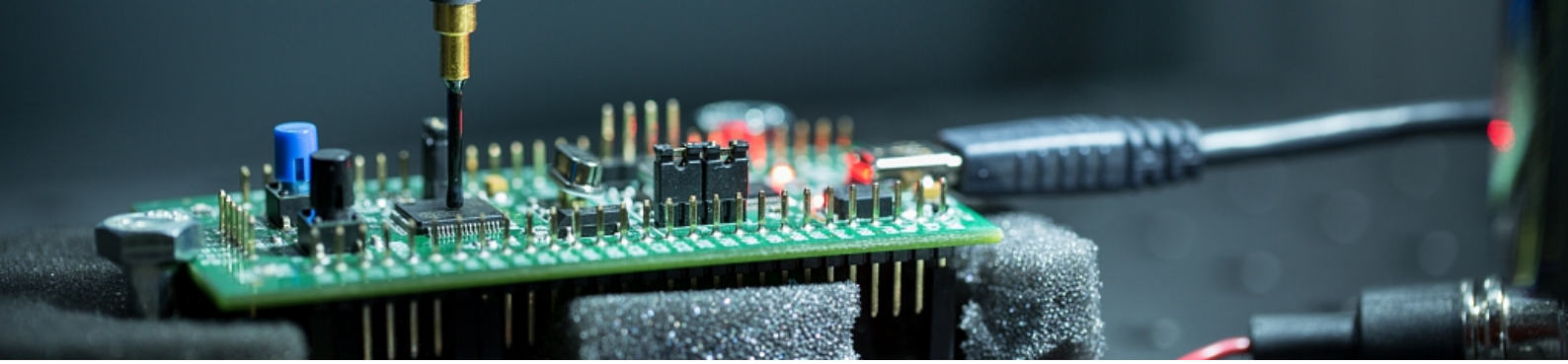
Le listing montre le code assembleur correspondant, mais pas uniquement.

— `0000000080000368 <linear_position>:` indique la présence d'un label entre chevrons, avec son nom `linear_position` et son adresse mémoire 80000368.

— Ensuite viennent les instructions. Pour chaque instruction, on a son adresse, son code machine et le mnémonique associé. Par exemple, 0055579b est le codage de l'instruction `srliw a5,a0,0x5`.

Il est important de noter que dans cet extrait de la figure 4.3, certaines instructions sont encodées en utilisant l'extension C, appelée "compressed", qui permet de coder certaines instructions sur 16 bits au lieu de 32.

Le label `linear_position` n'étant pas une instruction, il n'occupe pas d'espace mémoire : son adresse est la même que celle de l'instruction qui le suit. Ce nom correspond à celui de la fonction C à partir de laquelle ce code a été généré. Si le nom de ce label apparaît dans le binaire, c'est parce qu'il est présent en tant que symbole dans celui-ci ! On parle de symbole de débogage, car ces symboles ne sont réellement nécessaires que pour déboguer un binaire. Il est souvent possible de réduire la taille d'un binaire en supprimant ces symboles, ce que l'on appelle le *strip*. Dans ce cas, le désassembleur ne peut pas reconstituer les labels.



5. La microarchitecture des cœurs à exécution dans l'ordre *in-order*

Il existe plein de circuits différents, mais nous nous intéresserons ici à un type de circuit en particulier : les processeurs.

Un coeur de processeur moderne est en effet un circuit synchrone permettant l'exécution d'un ISA. L'implémentation matérielle d'un jeu d'instructions s'appelle la **microarchitecture**.

Il existe autant de microarchitectures que de coeurs, il n'existe pas une implémentation standardisée. D'ailleurs aujourd'hui, nous considérons que la microarchitecture est plus importante que l'ISA pour déterminer les qualités d'un processeur. Oubliez les débats ARM vs x86, RISC vs CISC, c'est la microarchitecture qui compte vraiment.

Dans les coeurs central processing unit (CPU), deux types de microarchitectures se partagent la plupart des implémentations :

— Les coeurs à exécution dans l'ordre (*in-order cores* en anglais) ont la microarchitecture la plus simple, avec une consommation énergétique plus faible, mais des performances limitées.

— Les coeurs à exécution dans le désordre (*out-of-order cores* en anglais) visent à obtenir les meilleures performances au prix d'une forte consommation énergétique et d'une grande complexité.

Beaucoup de processeurs modernes ont des coeurs des deux types en même temps. En fonction de l'intensité du calcul, ils peuvent activer les cœurs le plus énergivores, ou non. On parle alors d'**architecture hétérogène**.

En plus, certains cœurs (dans l'ordre ou dans le désordre) sont capables d'exécuter plusieurs instructions à la fois, dans le même cycle d'horloge. Il s'agit des cœurs **superscalaires**.

5.1 Description d'un cœur à exécution dans l'ordre

Nous allons ici décrire le fonctionnement d'un cœur à exécution dans l'ordre selon le modèle établi dans le livre de Patterson et Hennessy [45].

Ce cœur est constitué de 5 étages pipelinés, ayant tous une fonction spécifique.

Fetch L'instruction de départ est à une adresse fixe. Un registre spécifique, le program counter (PC), contient l'adresse de l'instruction suivante. Le rôle de l'étage de **Fetch** est d'aller chercher l'instruction suivante, en interrogeant le cache instruction (I$).

En sortie de cet étage, nous obtenons une instruction à exécuter.

Decode Il faut maintenant décoder l'instruction. À partir de son code, il faut identifier les registres utilisés, quelle unité d'exécution sera utilisée, etc. Les index des registres consommés par l'instruction, en particulier, sont envoyés au **Register file**, la mémoire contenant les registres.

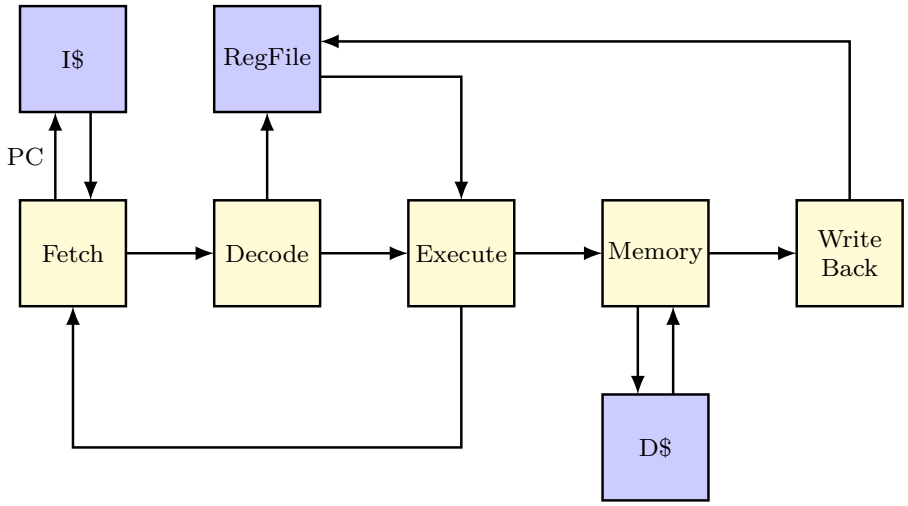


FIGURE 5.1 – Schéma simplifié du pipeline d'un cœur in-order à 5 étages.

Execute Il s'agit de l'étape où le processeur réalise les opérations arithmétiques. À partir des valeurs des registres d'entrée et du code de l'opération, cet étage produit la valeur résultat de l'opération. C'est également là que l'on calcule si un branchement est pris ou non.

Memory L'étage **Memory** est dédié aux échanges avec la mémoire, pour les instructions `load` et `store`. La raison de son positionnement après *Execute* est que, généralement, l'instruction permet de calculer l'adresse comme un décalage par rapport à la valeur d'un registre. Ainsi *Execute* calcule la valeur de l'adresse finale qui nourrit l'étage *Memory*.

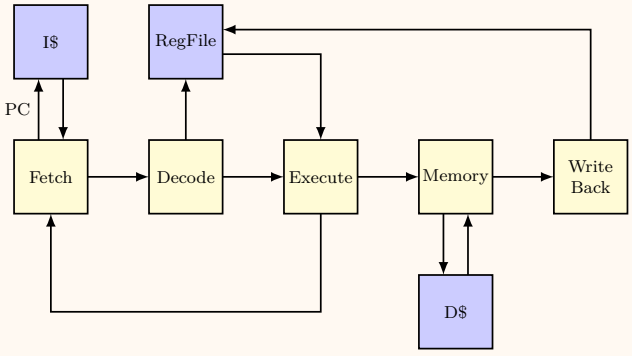
Cet étage échange en général avec un cache de données dédié. Il est à noter que les caches instructions et données de premier niveau sont généralement considérés comme faisant partie du cœur (cf chapitre 15).

Write-Back Enfin le dernier étage est responsable pour l'écriture du résultat de l'instruction dans le *Register file*.

Comme nous pouvons le voir, dans la plupart des cas, pour les instructions n'impliquant pas de lire ou écrire en mémoire, l'étage *Memory* fait perdre du temps. Pour optimiser le fonctionnement du cœur et redistribuer les données aux bons endroits dès qu'elles sont disponibles, on met en place le **forwarding**. Cela permet par exemple de relier la sortie de *Execute* à l'entrée du même étage pour pouvoir exécuter une instruction qui consomme le résultat de l'instruction précédente.

Exercice 5.1 - *Le voyage de l'instruction BEQ*

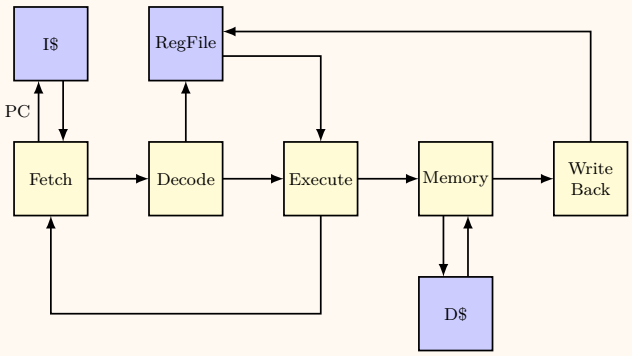
L'instruction `BEQ rs1, rs2, offset`, *branch if equal*, saute à l'instruction définie par l'*offset* ssi `rs1` == `rs2`. Sinon, on continue avec l'instruction suivante.



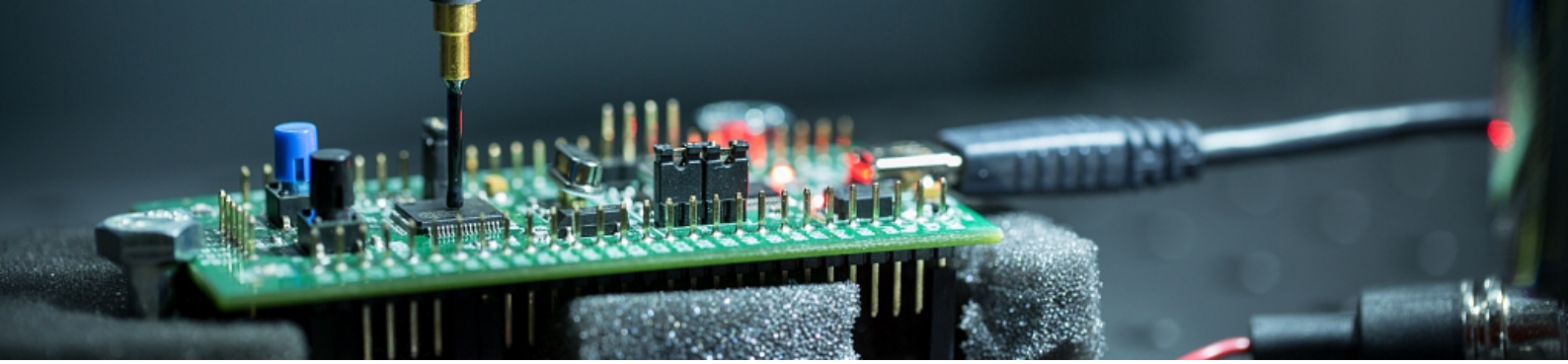
Dans le schéma du cœur à 5 étages, décrivez les opérations élémentaires effectuées à chaque étage.

Exercice 5.2 - *Le voyage de l'instruction LOAD*

L'instruction `LOAD rd, off(rs1)`, charge la valeur définie par l'adresse $rs1 + off$ et la met dans `rd`.



Dans le schéma du cœur à 5 étages, décrivez les opérations élémentaires effectuées à chaque étage.



6. Fonctionnement d'un processeur à exécution dans le désordre (*out-of-order*)

Les processeurs modernes utilisent différentes techniques pour améliorer les performances. L'une de ces techniques est l'exécution dans le désordre (*out-of-order execution*). Cette technique permet au processeur d'exécuter les instructions dans un ordre différent de celui dans lequel elles apparaissent dans le programme, tout en garantissant que le résultat final est le même que si les instructions avaient été exécutées dans l'ordre.

Nous allons ici décrire les principes généraux, chaque processeur ayant sa propre implémentation.

6.1 Principes de Base

L'exécution dans le désordre est basée sur l'idée que certaines instructions ne dépendent pas des résultats d'autres instructions qui les précèdent dans le programme. Par exemple, considérez les instructions suivantes :

```
A: x = y + z
B: u = v * w
C: t = x + u
D: u = y - v
```

L'instruction B ne dépend pas du résultat de l'instruction A, donc ces deux instructions peuvent être exécutées en parallèle ou dans un ordre différent. L'instruction C, elle, devra attendre la complétion des deux instructions précédentes.

Il est donc possible de traiter les instructions A et B en même temps, dans des unités d'exécution différentes. L'instruction D réutilise u, mais il ne s'agit que d'une collision de nommage : il suffirait de renommer le registre pour ne plus avoir de dépendance.

6.2 Microarchitecture typique

Ce que l'on appelle le frontend est la partie du processeur qui est responsable de la récupération des instructions, de leur décodage et de leur envoi aux unités d'exécution. Le frontend s'exécute dans l'ordre. Le backend est la partie qui se charge de l'exécution des instructions dans le désordre tout en gardant trace de l'ordre du programme pour une gestion correcte des exceptions.

Fetch et Decode Le processeur doit être capable de récupérer et de décoder un nombre d'instructions important : le fetch et le decode sont en général faits par paquets de plusieurs instructions. Ces opérations sont trivialement parallélisables si la taille des instructions est fixe.

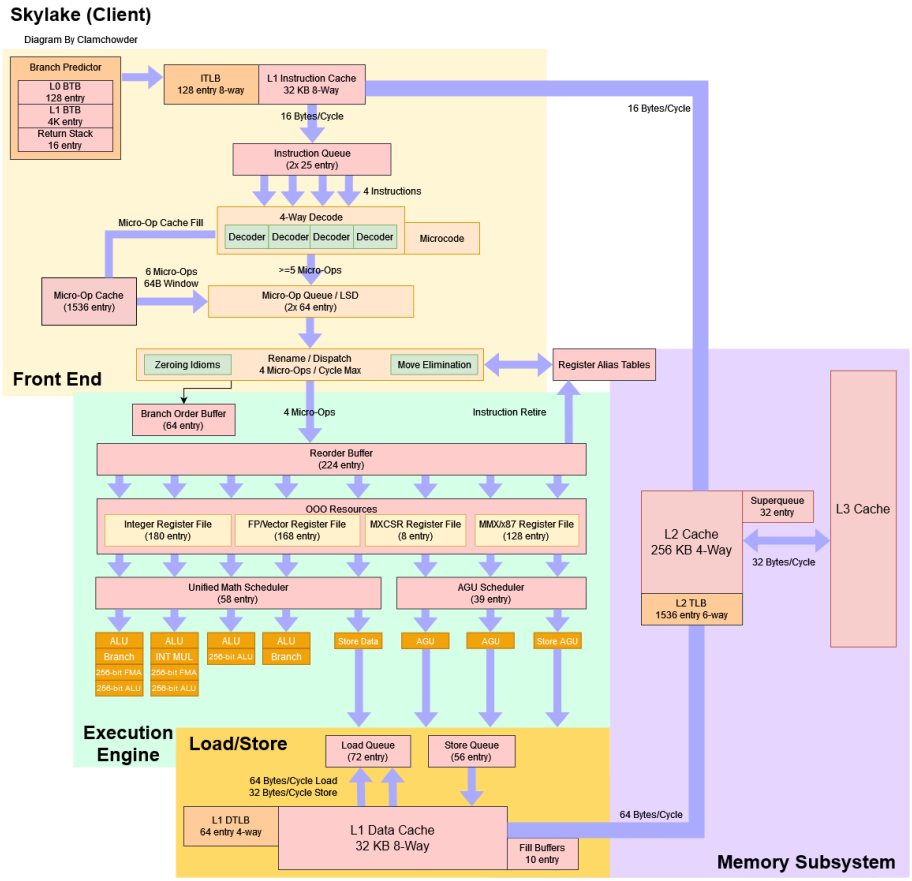


FIGURE 6.1 – La microarchitecture des cœurs Skylake d'Intel. Source : https://chipsandcheese.com/.

Dépendances entre les instructions Pour obtenir une exécution dans le désordre, il faut déterminer les dépendances entre les instructions. Ces dépendances sont de plusieurs types :

- Read-after-Write (RAW) : une instruction lit un registre qui a été écrit par une instruction précédente (cf instruction C). Ceci est une vraie dépendance qui ne peut pas être évitée.
- Write-after-Read (WAR) : une instruction écrit un registre qui a été lu par une instruction précédente. Ceci est une fausse dépendance, due au fait que le nombre de registres est fini et limité. Il est possible de la résoudre en renommant les registres.
- Write-after-Write (WAW) : une instruction écrit un registre qui a été écrit par une instruction précédente. Ceci est une autre fausse dépendance, et la première écriture peut souvent être ignorée, on parle de *squashing*.

Pour éliminer les fausses dépendances, il faut renommer les registres (*register renaming*). Sur un tel processeur, il y a en général plus de registres physiques (ceux existant réellement dans le processeur) que de registres architecturaux (ceux définis par le jeu d'instructions). À l'étape de renommage, on assigne un registre physique à chaque registre architectural, en prenant soin de ne tenir compte que des vraies dépendances. Les instructions sont alors exécutées sur les registres physiques, et les résultats sont écrits dans les registres physiques.

Dispatch Le dispatch est le processus par lequel les instructions sont routées vers l'*instruction queue*, une file d'attente globale, en vue de leur exécution future par les unités d'exécution du processeur. Après le renommage des registres, les instructions sont prêtes à être dispatchées, car toutes les fausses dépendances ont été éliminées, et chaque instruction est désormais associée à des registres physiques spécifiques. Le dispatch prend en compte les dépendances réelles entre les instructions pour déterminer le moment approprié de leur transfert vers la file d'attente.

Ce mécanisme joue un rôle crucial dans l'optimisation du flux d'instructions, car il permet

d'aligner l'ordre d'exécution des instructions avec la disponibilité des unités d'exécution et des données nécessaires. Il assure également que les ressources du processeur sont utilisées de manière efficiente, en évitant les situations où les unités d'exécution restent inactives faute d'instructions prêtes à être traitées. En définitive, le dispatch contribue à maximiser le débit d'instructions et à réduire le temps d'exécution global des programmes.

Issue et unités d'exécution L'étape d'issue consiste à envoyer les instructions, venant d'une file globale (*instruction queue*), aux unités d'exécution. Chaque unité possède une file d'entrées appelée *reservation station* qui contient les instructions qui doivent être exécutées par cette unité.

Les unités d'exécution sont en général spécialisées dans un type d'instruction. Typiquement, on sépare les unités arithmetic and logical unit (ALU), branch and repeat unit (BRU), load store unit (LSU), les unités pour les calculs en virgule flottante, pour la multiplication, etc.

L'issue doit donc choisir l'unité d'exécution la plus appropriée pour chaque instruction, en fonction de son type, des dépendances de l'instruction et de la disponibilité des unités d'exécution. Lorsque toutes les dépendances sont satisfaites et que l'unité d'exécution est disponible, l'instruction est transférée de la file globale vers la file dédiée à l'unité d'exécution. Une fois qu'une instruction est exécutée, elle est retirée de la file d'entrées de cette unité. Le résultat de l'instruction est stocké dans un tampon dédié, le reorder buffer (ROB), et n'est pas propagé tout de suite dans les registres. Toutefois, les instructions dépendantes peuvent utiliser ce résultat directement depuis le ROB.

Commit Notamment pour pouvoir gérer les erreurs correctement, le processeur doit donner l'illusion de s'exécuter dans l'ordre. C'est le rôle de l'étape de commit qui détermine quelles instructions peuvent être retirées du ROB et écrites dans les registres, suivant l'ordre des instructions du programme.

6.3 L'exécution spéculative

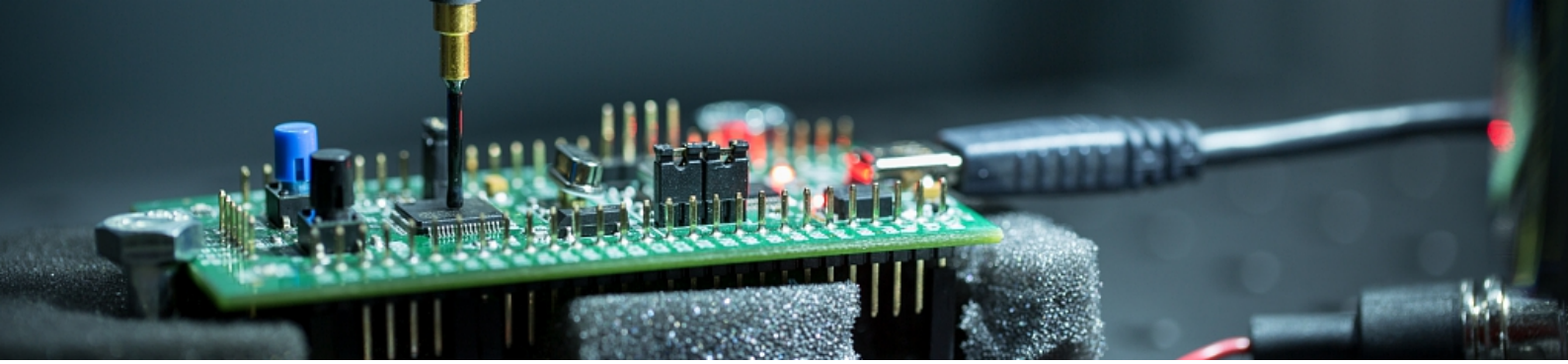
La spéculation, ou exécution spéculative, est la possibilité pour les cœurs complexes d'exécuter des instructions sans être certain qu'elles feront partie du flot d'exécution final. Dans les cœurs simples, la fenêtre de spéculation (le nombre d'instructions spéculatives) est en général inférieure à la taille du pipeline. Ainsi un mauvais branchement ne nécessite que de vider le pipeline.

Un cœur capable d'exécution spéculative est capable de considérer des fenêtres de spéculation beaucoup plus grandes que le pipeline. Ce dernier doit donc être modifié en profondeur pour gérer ce mode d'exécution.

Il existe de multiples sources de spéculation :

— Prédiction de la direction de branchement : le processeur doit prédire si un branchement sera pris ou non (en utilisant une BHT par exemple).
— Prédiction de la cible d'un branchement : le processeur doit prédire la cible d'un branchement (en utilisant un BTB ou le RSB par exemple).
— Prédiction de l'aliasing mémoire (ou *store-to-load forwarding*) : le processeur doit prédire si une instruction `load` peut être exécutée avant une instruction `store` qui lui est antérieure en supposant que les deux instructions accèdent à des adresses différentes.

Il n'est pas impossible que d'autres mécanismes de spéculation existent, mais ceux-ci sont les plus courants.



7. La prédiction de branchement

On appelle branchements conditionnels les instructions qui peuvent rediriger le flux de contrôle en fonction d'une condition. Par exemple, `blt rs1, rs2, offset` est l'instruction RISC-V qui est définie avec la sémantique suivante : le flux de contrôle doit aller à l'instruction courante décalée de l'offset si le registre `rs1` contient une valeur strictement inférieure (*branch if lower than*) à la valeur du registre `rs2`. Sinon, le flux de contrôle continue avec l'instruction suivante.

Dans ce cours, un branchement est toujours un branchement conditionnel, sinon on parle de sauts.

Dans une architecture pipelinée, les branchements représentent une difficulté car ils créent une discontinuité dans le flux de contrôle. Une instruction qui permet un branchement a deux successeurs possibles, suivant si le branchement est pris (*taken* / T) ou non (*not taken* / N). C'est ce que l'on appelle la direction du branchement. La décision de la direction de branchement est prise tardivement dans le pipeline, puisqu'il faut réaliser une opération sur des registres. Mais on a besoin de cette direction rapidement, pour récupérer la bonne instruction lors du *fetch*. Pour éviter une trop grande perte de performance, la grande majorité des cœurs, y compris les petits microcontrôleurs, implémentent une forme de prédiction de cette direction. Cette prédiction peut être associée à de la spéculation (cf section 6.3), où le branchement prédit entre dans le pipeline. En cas de mauvaise prédiction, les instructions incorrectes sont évacuées du pipeline et remplacées par les bonnes.

Aujourd'hui, il existe des prédictions de direction de branchement efficaces, avec des taux de moins de 3 misp/KI (mispredictions per kilo instructions) facilement atteints.

> **!** Le comportement spéculatif lié à la prédiction de branchement dans un cœur simple in-order n'est pas ce qu'on appelle l'exécution spéculative, que l'on verra dans la section 6.3. La différence principale est que la fenêtre de spéculation, dans ce cas, est inférieure à la taille du pipeline : ce mécanisme est beaucoup plus simple.

Nous allons voir en détail dans cette section comment fonctionnent certains mécanismes de prédiction de direction : BHT, pattern history table (PHT) et GShare.

Exercice 7.1 - *De l'importance de la prédiction de direction de branchement*

Soit un cœur fictif avec les caractéristiques suivantes :
- Pipeline de profondeur 10.
- Superscalaire, capable d'exécuter 2 instructions par cycle d'horloge dans le cas idéal.
- Les instructions du programme en cours d'exécution sont à 20% des branchements. Autrement formulé, 1 instruction sur 5 est un branchement.

On suppose que, de manière simplifiée, tant que les prédictions de branchements sont bonnes, le cœur exécute 2 instructions par cycle. Si un branchement est mal prédit, le pipeline entier doit être invalidé : coût de 10 cycles en cas de mauvaise prédiction.
Questions :
1. Quel est le nombre d'instructions per cycle (IPC) du cœur en cas de prédiction parfaite (100% de bonnes prédictions) ?
2. Quel est l'instructions per cycle (IPC) du cœur pour 50% de bonnes prédictions ?
3. Quel taux de succès doit atteindre le prédicteur pour obtenir un IPC de 1.5 ?
4. Complexifier, ajouter des transistors sur un prédicteur de branchement pour l'améliorer, augmente-t-il ou diminue-t-il la consommation énergétique de la puce ?

Il s'agit ici d'obtenir des ordres de grandeur.

7.1 Prédicteurs statiques

7.1.1 Branchement pris par défaut

Le mécanisme le plus simple de prédiction de branchement est de supposer que le branchement sera pris par défaut. À cause de la présence de boucles dans les programmes, cette prédiction sera juste dans la majorité des cas. On parle ici de prédiction statique car cela ne dépend pas de l'exécution du programme.

Ce mécanisme est intéressant s'il est facile de savoir si une instruction est un branchement, ce qui est le cas avec RISC-V. Dans le cas de x86, il peut être nécessaire de décoder l'instruction pour déterminer qu'il s'agit d'un branchement, ce qui réduit l'intérêt de cette méthode.

Selon [51], environ 60% à 70% des branchements exécutés sont pris.

7.1.2 Selon l'orientation du branchement

Il est possible d'améliorer légèrement le prédicteur précédent en regardant l'orientation du saut :
- Vers l'avant : si le branchement est pris, on saute à une adresse supérieure à celle du branchement.
- Vers l'arrière : dans ce cas, on saute à une adresse inférieure.

Comme la plupart des boucles ont leur condition placée à la fin de celle-ci, le prédicteur *backward taken, forward not taken* suppose que le branchement ne sera pris que si l'offset du branchement est négatif (celui-ci saute à une adresse inférieure). Dans l'autre cas, on prédit *not taken*.

7.1.3 Selon les conditions de branchement

Avec RISC-V, il y a plusieurs instructions de branchement, suivant la condition à appliquer aux registres (égaux, supérieurs à, etc.). Cette distinction nous permet d'affiner notre prédicteur statique : pour chaque condition de branchement, et chaque orientation, nous choisissons la direction la plus probable. Cela permet d'atteindre un taux de succès d'environ 80%.

7.2 Prédicteurs dynamiques

Les prédicteurs statiques ne peuvent pas s'adapter à l'exécution du programme. Peut-être que selon les fonctions de votre programme, toutes n'ont pas les mêmes caractéristiques de branchement. Il faut donc un prédicteur capable de mesurer l'activité du programme lors de son exécution. On parle alors de prédicteurs dynamiques.

7.2.1 Répéter la dernière direction (*last branch predictor*)

Le choix le plus simple est de sauvegarder la décision précédente *taken* / *not taken* (T/N). On peut supposer que rejouer la décision précédente serait efficace. Et c'est le cas pour des séquences longues où les branchements sont dans la même direction.

Pour l'implémenter, on crée une table à 2^n entrées, où chaque entrée contient 1 bit définissant la direction du dernier branchement à l'adresse a dont les n bits de poids faible désignent l'entrée dans le tableau (voir figure 7.1 pour l'illustration d'une structure similaire). Le problème est que pour une séquence de branchements $\mathrm{T}/\mathrm{N}/\mathrm{T}/\mathrm{N}/\mathrm{T}/\mathrm{N}$..., le taux de bonnes prédictions est de 0%, ce qui détruit les performances du cœur, et rend ce prédicteur inapplicable en pratique.

7.2.2 BHT

Pour contrer cette instabilité, il suffit de rajouter un bit d'hystérésis, c'est le principe du branch history table (BHT).

Source	Counter (2-bit)
b0000	b00
b0001	b10
...	...
b1111	b11

FIGURE 7.1 – Un exemple de BHT à 16 entrées de 2 bits. Les sources sont les indices du tableau.

Le 2-bit BHT, illustré sur la figure 7.1, sauvegarde un compteur sur 2 bits du nombre de décisions T prises sur les 3 derniers branchements. Le bit de poids fort de ce compteur indique la prédiction T/N pour le branchement suivant à la même adresse. Plus précisément à la même adresse partageant les n bits de poids faibles pour une table à 2^n entrées ($n = 4$ bits dans la figure 7.1). On parle ici de corrélation de branche locale : on ne considère l'historique que d'une seule branche.

Il y a toutefois un risque de collision si n est trop petit : deux branchements avec le même index dans le BHT vont être en concurrence sur le même compteur, ce qui peut engendrer de mauvaises prédictions.

2 bits est considéré comme la meilleure taille pour les compteurs : un 1-bit BHT est en

fait un *last branch predictor*, et plus de bits n'améliorent pas significativement la performance du prédicteur qui peut atteindre 90% avec 2 bits.

7.2.3 PHT

La direction de certains branchements peut être corrélée à ceux d'autres branchements ; on parle alors de corrélation de branche globale.

```
if (x) { // branch 1
    a = 0;
}

if (a % 2 == 0) { // branch 2

}
```

Dans l'exemple ci-dessus, le deuxième branchement est T si le premier l'est.

Ou la direction est corrélée à des branchements anciens à la même adresse, on parle de corrélation de branche locale. Ce cas est notamment représenté par les boucles.

C'est le principe derrière la pattern history table (PHT), où l'on utilise un registre global, le global history register (GHR), qui sauvegarde l'historique des décisions T/N. Par exemple, un registre sur n bits enregistre les décisions des n derniers branchements, avec un 1 pour un branchement pris (T) et 0 pour un branchement non pris (N). La valeur du GHR est utilisée comme index dans la PHT : pour chaque index, nous avons un compteur sur 2 bits comme pour la BHT prédisant la décision T/N.

7.2.4 GShare

Le GShare est un prédicteur de branchement qui combine les deux approches précédentes : BHT et PHT. Il s'agit d'un prédicteur de branchement à corrélation de branche globale mais qui tient compte de l'adresse du branchement.

L'index de la table de compteurs est obtenu en combinant l'adresse du branchement et le GHR par un XOR (cf figure 7.2).

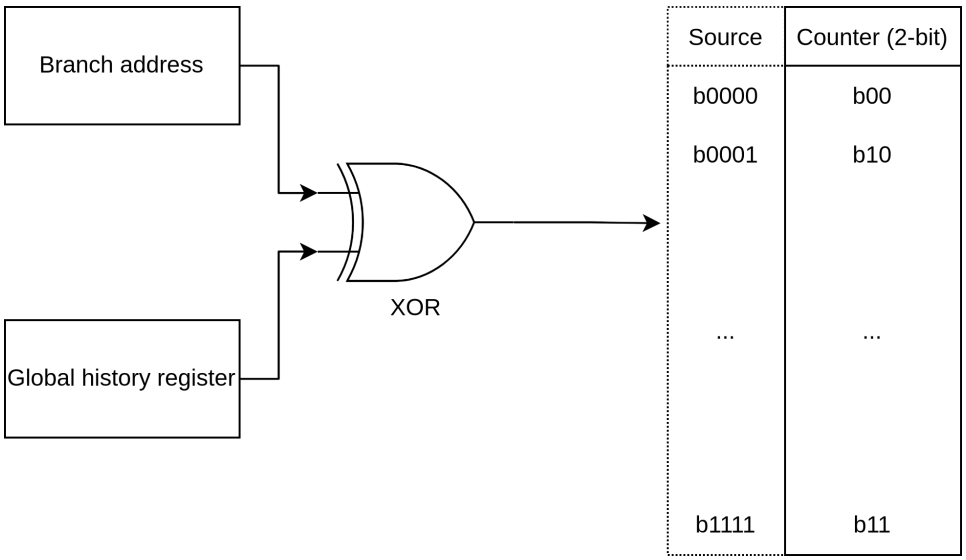


FIGURE 7.2 – Un exemple de GShare.

On notera que la taille du GHR doit être adaptée au nombre d'entrées de la table de compteurs. Sinon, certains bits sont ignorés.

Utilisés naïvement, les prédicteurs PHT et GShare souffrent d'un défaut gênant : ils ont un temps de chauffe (*warmup*) long. Autrement dit, si le GHR est de 32 bits, il faut que l'historique sur 32 bits soit établi pour avoir une bonne prédiction. Dans le cas d'une boucle toujours T, il n'y a pas d'entrée dans le PHT correspondante avant la 33^{e} itération.

7.2.5 Tournament predictors

On parle de tournoi de prédicteurs lorsque l'on combine plusieurs prédicteurs différents. Le choix du prédicteur à utiliser est alors lui-même le résultat d'un entraînement à partir de l'historique des branchements. Autrement dit, on entraîne un prédicteur à prédire quel prédicteur choisir.

7.2.6 TAGE

Le prédicteur tagged geometric history length branch predictor (TAGE) [35] améliore significativement le défaut du GShare : suivant le branchement, la longueur de l'historique à considérer n'est pas toujours la même. L'idée du prédicteur TAGE est d'avoir plusieurs sous-prédicteurs T_i pour différentes tailles d'historique de branchement. La taille de ces historiques suit une progression géométrique en i (d'où le nom de TA**GE**).

Dans le cas du GShare, l'entrée de la table de décision, constituée du XOR entre l'adresse et l'historique, est toujours valide et donne une décision. Il serait cependant intéressant de savoir si ce couple (adresse de branchement, historique de branchements), parfois appelé scénario de branchements, a réellement été observé. C'est le concept du tag (**TA**GE) : chaque sous-prédicteur T_i est constitué de lignes avec (compteur de branchements $pred$, tag, compteur d'utilité u), comme illustré sur la figure 7.3.

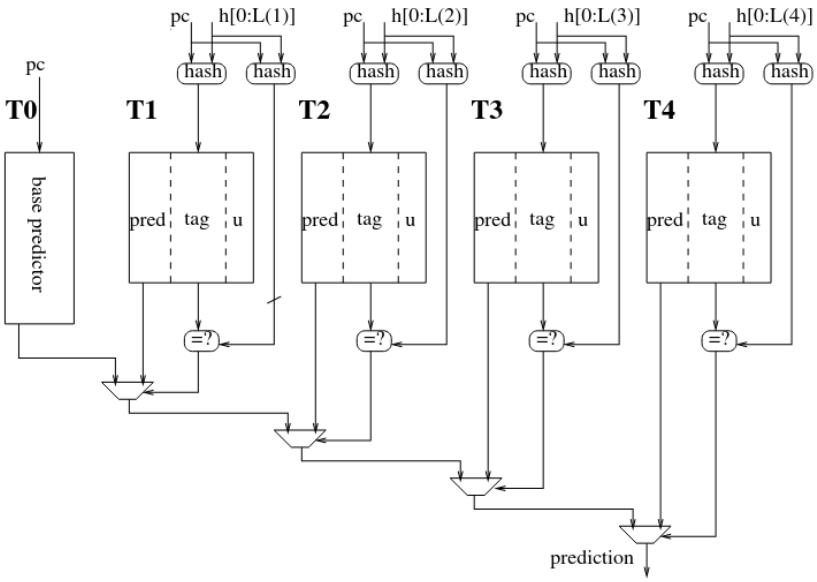


FIGURE 7.3 – La structure du prédicteur TAGE, tiré de [35].

Le tag est calculé comme le hash du scénario de branchement, il doit être si possible unique pour chaque scénario. Le compteur de branchement compte si les branchements correspondant au scénario sont T ou N. Le compteur d'utilité sert à savoir si cette entrée est toujours utile, toujours utilisée ou s'il ne serait pas mieux de laisser la place à un autre scénario. Pour cette raison, les bits de ce compteur d'utilité sont mis à zéro périodiquement. Ces compteurs servent à allouer ou désallouer les entrées dans les prédicteurs lors de nouveaux scénarios de branchement.

Le plus long scénario valide est toujours utilisé pour prédire un branchement. Autrement dit, si le scénario a deux entrées valides, dont le tag correspond, dans deux T_i et T_j, alors on choisit $T_{\max(i,j)}$ pour la prédiction.

Si aucun scénario n'est valide, alors on utilise T_0 qui n'utilise pas l'historique : il s'agit d'un simple BHT.

Les prédicteurs de la famille du TAGE sont maintenant les plus courants dans les processeurs performants, notamment les processeurs Intel.

7.2.7 Perceptron predictors

Les prédicteurs à base de perceptron sont une technique moderne et très efficace. Les processeurs AMD utilisent des prédicteurs de cette famille, bien que les détails ne soient pas publics.

La prédiction de la direction de branchement est un problème d'apprentissage supervisé : selon l'historique de l'exécution, nous avons le contexte (adresse et historique) et le résultat des branchements. Les perceptrons sont une des toutes premières formes de réseau de neurones, et peuvent être appliqués dans notre cas d'usage.

Le perceptron désigne un vecteur de poids $W_0, W_1, \ldots, W_{n-1}$; ces poids sont des compteurs entiers signés, souvent sur 8 bits ($\in [\![-128, 127]\!]$). Ces poids sont appliqués sur une entrée constituée de l'historique des branchements $H_0 = 1, H_1, H_2, \ldots, H_{n-1}$ (la première entrée est bien 1, nous en détaillerons l'intuition par la suite). Ces H_i prennent deux valeurs : 1 pour T, -1 pour N.

La sortie du prédicteur P est la somme pondérée :

$$P = \sum_{i=0}^{n-1} H_i \cdot W_i \tag{7.1}$$

La prédiction T/Nest déterminée par le signe de P.

L'entraînement se fait à l'aide d'un paramètre arbitraire θ qui définit le seuil d'erreur où la mise à jour est nécessaire. Soit b le résultat réalisé du branchement d'intérêt ($b = 1$ si T, $b = -1$ si N).

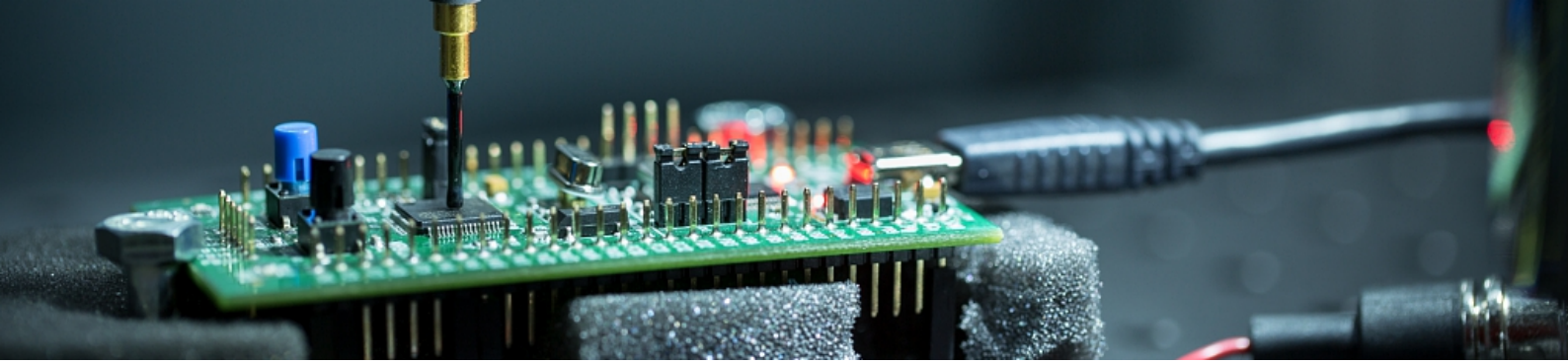
Algorithm 1 Entraînement du perceptron en cas de mauvaise prédiction, ou de trop d'incertitude sur cette prédiction.

if $sign(P) \neq b$ or $|P| < \theta$ **then**
  for all $i \in [\![0, n-1]\!]$ in parallel **do**
    $W_i \leftarrow W_i + b \cdot H_i$
  end for
end if

Intuitivement, le perceptron associe une importance, à travers le poids W_i, à chaque direction de branchement dans l'historique. Si ce branchement historique particulier est fortement corrélé à notre décision, le poids sera positif et élevé. S'il est anticorrélé, il sera négatif et élevé en valeur absolue. Si ce branchement n'influe pas sur notre décision, son poids sera faible ou nul.

Similairement à la transition du PHT au GShare, il est intéressant d'avoir une prédiction différente suivant l'adresse du branchement à prédire. Pour cela, il suffit de sauvegarder et mettre à jour un vecteur $W = W_0, \ldots, W_{n-1}$ pour chaque adresse de branchement.

Ce prédicteur est coûteux à implémenter mais peut être très performant.



8. Les prédicteurs de destination de saut

En plus de prédire la direction d'un branchement, il est possible de prédire la destination, c'est à dire l'adresse, des sauts. Les branchements étant des sauts conditionnels, ils peuvent également être concernés.

Il est possible de prédire la destination d'un saut, à partir de l'adresse de l'instruction. Ainsi cette prédiction peut se faire très tôt dans le pipeline, dès que l'on connait l'adresse de l'instruction de saut.

8.1 BTB

Le branch target buffer (BTB) est un buffer sauvegardant les adresses de destination des branchements ou sauts précédents. On peut le voir comme une fonction

$$BTB : \text{adresse de l'instruction}_n \mapsto adresse$$

Évidemment, on ne va pas utiliser un tableau de taille 2^{32} ou 2^{64}, on ne considère en entrée que les n bits de poids faible pour obtenir un tableau de 2^n adresses.

Source	Destination
b0000	h80001245
b0001	h800a4788
...	...
b1111	invalid_target handler

FIGURE 8.1 – Un exemple de BTB à 16 entrées. Bien sûr, les sources n'ont pas besoin d'être sauvegardées, il s'agit des indices dans le tableau.

Selon la figure 8.1, pour une instruction dont les 4 bits de poids faible de l'adresse sont 0001, la destination prédite est $h800a4788$. C'est-à-dire que cette adresse est prédite comme étant celle de la prochaine instruction a exécuter. Cette prédiction est issue du branchement ou saut précédent dont l'adresse avait 0001 en bit de poids faible. Le BTB est mis à jour à chaque branchement ou saut réellement pris.

En pratique, le BTB est surtout utilisé pour les sauts indirects, les branchements sont plus efficacement traités par une prédiction de la direction *taken* / *not taken* (T/N) lorsque la destination du branchement est calculable à partir de la valeur de l'instruction (destination $=$ PC $+$ offset).

8.2 RSB ou RAS

Une autre prédiction simple à réaliser est celle d'un retour de fonction. L'adresse de retour est celle de l'instruction qui suit l'appel de la fonction. Pour accélérer ce saut indirect, il est possible d'utiliser la return address stack (RAS), aussi parfois appelée return stack buffer (RSB). Il s'agit d'une pile matérielle, poussant une adresse à chaque instruction `jalr x1, rs1` et en tirant une à chaque `jalr x0, x1`. Selon la spécification RISC-V, ce mécanisme peut être utilisé uniquement pour les registres `x1` (comme sur l'exemple précédent) et `x5`. L'adresse prédite du prochain retour est toujours en haut de cette pile. Dans la figure 8.2, cette pile est implémentée à l'aide d'une mémoire tampon circulaire, accompagnée d'un registre (ici sur 3 bits) pointant vers la dernière entrée valide du tampon.

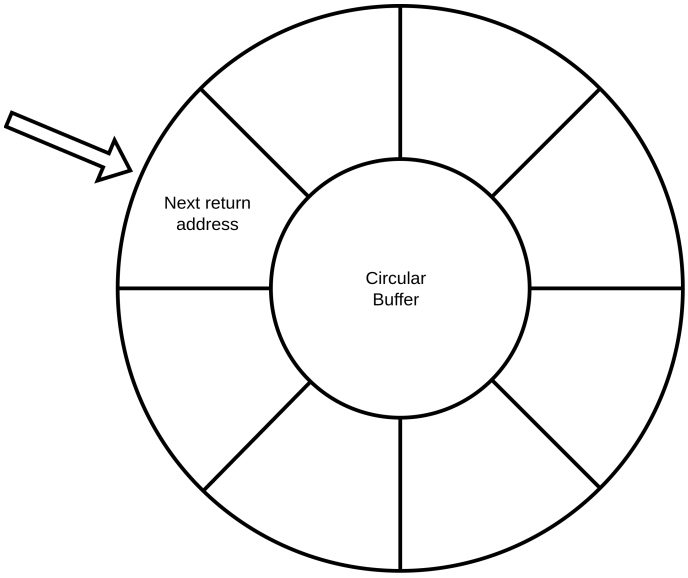
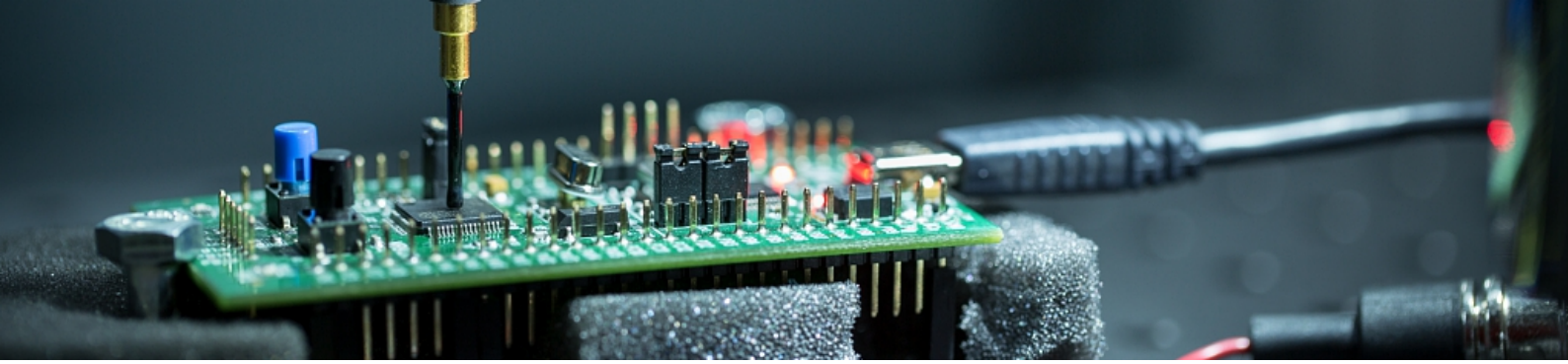


FIGURE 8.2 – Un exemple de RSB à 8 entrées. Il s'agit d'une mémoire tampon circulaire, plus un registre qui permet de pointer vers la dernière entrée valide dans le tampon.



9. Les prefetchers

Un prefetcher est un mécanisme de prédiction de valeur. Ils sont principalement utilisés dans deux contextes. D'abord pour prédire les adresses des accès mémoires, par exemple pour prédire l'adresse de la prochaine ligne de cache à récupérer. Mais également pour spéculer sur la valeur de retour d'un `LOAD` avant que l'accès mémoire ne soit réellement utilisé. Nous nous concentrerons ici sur les prefetchs mémoires.

9.1 Next Line Prefetching

La contrepartie de la prédiction de branchement : quelle instruction faut-il charger lorsque l'on prédit NT ? Il faut l'instruction suivante en mémoire. D'où le prefetch le plus simple, qui prédit que la prochaine ligne est celle qui suit en mémoire la ligne de l'instruction en cours de fetch.

Ce prefetcher est particulièrement utilisé pour le cache instruction.

9.2 Stream and stride prefetching

Les prefetchers de données sont plus variés, et peuvent différer selon leur niveau dans la hiérarchie : un prefetcher pour L2 ou L3 n'utiliseront probablement pas la même technique. Un stream prefetcher est l'équivalent du *next line prefetcher* pour les données.

Le principe du stream prefetcher est de détecter et anticiper des accès à des adresses consécutives.

1. Pour un accès à une adresse a.
2. Si un accès à l'adresse $a+1$ peu après,
3. prédire un prochain accès à $a+2$.
4. La confiance dans la prédiction augmente si le motif se répète.

Le stride prefetcher c'est la même chose pour des accès séparés par une constante S. Les accès à a, $a+S$, $a+2\cdot S$ nous permettent de prédire un prochain accès à $a+3\cdot S$.

9.3 Global history buffer [27]

Le GHB applique les principes des BHT et PHT vu dans les sections 7.2.2 et 7.2.3 aux prefetchers. La solution avec GHB utilise 2 tables :

1. Une première, appelée GHB, sauvegarde l'historique des accès mémoires *missed*. Dans cette même table, pour chaque entrée contentant une adresse *missed*, nous associons l'entrée du miss précédent correspondant à la même adresse.

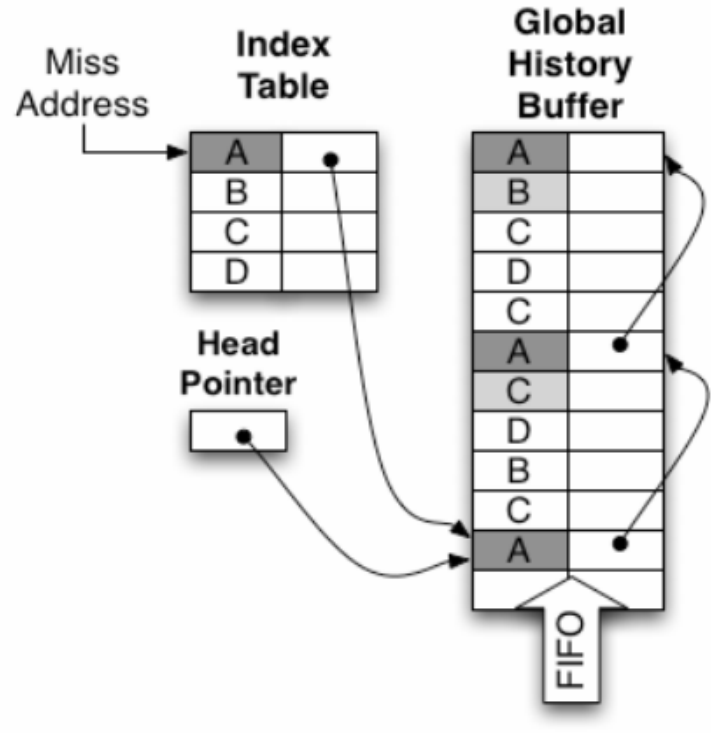


FIGURE 9.1 – Les tables du GHB, et leur relations, tiré de [27].

2. Une deuxième table permet d'associer l'entrée la plus récente dans le GHB à une adresse *missed*.

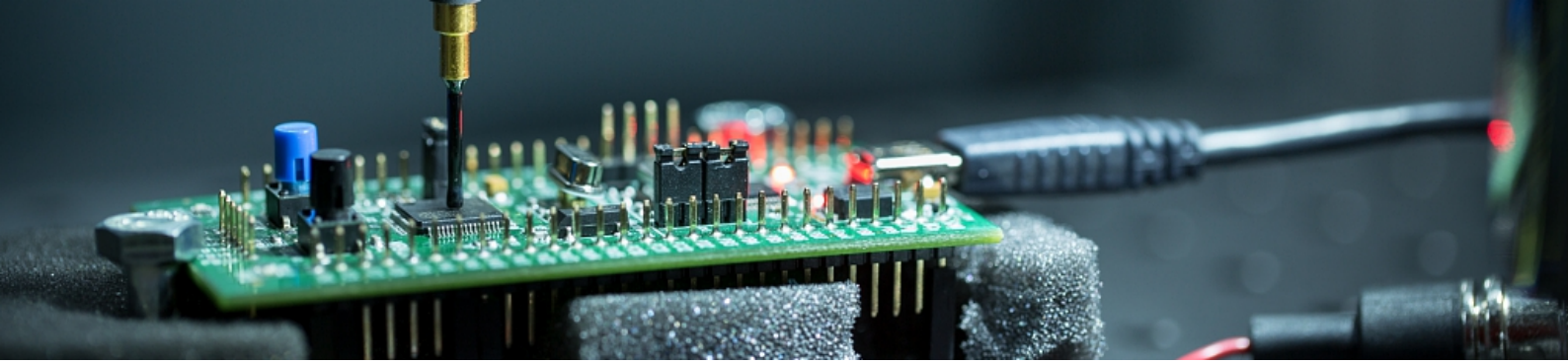
Ainsi nous avons facilement l'information de quelles adresses sont susceptibles de suivre un accès *missed*.

II Les attaques physiques

Introduction

La sécurité des systèmes embarqué est obligé de considérer un modèle d'attaquant particulier. Celui-ci est en effet capable de récupérer le système et de l'emmener dans son laboratoire. Il est alors capable de mesurer ou d'influer sur l'environnement physique du système : ce sont les attaques physiques.

Ce qui nous intéresse dans cette partie, ce sont les intéractions entre un circuit intégré et son environnement. Ainsi après un rapide rappel des concepts liés aux circuits intégrés dans le chapitre 1, nous verrons les deux familles d'attaques physiques. Les attaques par observation dans le chapitre 10 et les attaques par injection de faute dans le chapitre 11.



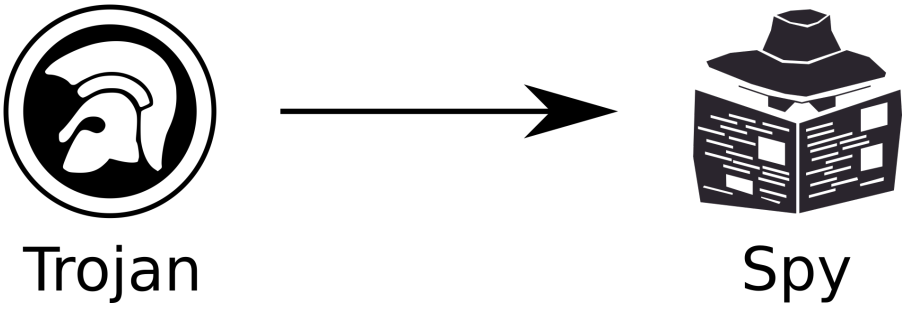
10. Les attaques par observation

Les attaques par observation sont des attaques où l'attaquant exploite l'observation de l'environnement physique du système pour atteindre son objectif. Le temps, la temperature, la consommation de courant, le rayonnement électromagnétique sont des exemples de quantités physiques mesurées. Cette observation nécessite le plus souvent que l'attaquant ait le système à sa disposition dans son laboratoire. Mais dans certains cas, il est possible de mesurer des paramètres physiques à partir du logiciel, notamment pour la mesure du temps. Comme dans toutes mesures expérimentales, il faut prendre en compte un bruit environnant et des erreurs de mesures.

10.1 Les canaux de communication : modèle de la menace

L'objectif de la plupart des attaques par observation est d'exfiltrer un secret, auquel l'attaquant ne devrait pas avoir accès. On distingue généralement deux schémas d'attaques : les canaux cachés et les canaux auxiliaires.

Canaux cachés Pour exfiltrer un secret, il faut établir un canal de communication entre une entité ayant accès au secret, entité que l'on nomme le Troyen. À l'autre bout, une autre entité que l'on nomme l'Espion tente de recevoir la communication alors qu'elle n'a normalement pas accès au secret.



Dans le cas des canaux cachés, l'attaquant contrôle à la fois le Troyen et l'Espion. Les difficultés sont alors de détourner des ressources pour permettre cette communication et de ne pas se faire repérer. Les canaux cachés sont plutôt utilisés dans un contexte de sécurité de la microarchitecture, comme nous le verrons dans la partie VI.

Canaux auxiliaires On appelle les canaux auxiliaires, les canaux de communication où l'attaquant ne contrôle que l'Espion. L'émission du secret se fait naturellement, à l'insu de la cible.

Les canaux auxiliaires sont particulièrement utilisés lors de l'extraction de secrets du fait de la consommation de courant par exemple.

10.2 Ce que l'on peut mesurer

L'environnement physique d'un système comprends de nombreux paramètres, et plusieurs dimensions physiques peuvent être mesurées et exploitées.

10.2.1 Le temps

Le concept d'attaque temporelle a été introduit par Kocher[17] en 1996. L'idée est que le temps d'exécution d'un algorithme donne de l'information sur ce calcul et les données manipulées.

Exemple 10.1

Je donne à une personne le calcul $2+3=?$ et à une autre personne le calcul $\sqrt{2^4-3*1.3}=?$. La personne A donne la première sa réponse. Quel calcul avait-elle à réaliser ?

Comment exploiter cette information dépend de l'application précise qui est ciblée. Par exemple dans le chapitre 22, nous verrons des attaques très efficaces pour lire des données secrètes dans un cœur moderne.

L'exploitation du temps pour retrouver un code PIN Un code PIN est un code confidentiel destiné à authontifier le titulaire d'une carte à puce. Il s'agit d'un couple (identifiant, mot de passe). La carte est l'identifiant, le code PIN le mot de passe.

Examinons le cas de la vérification de code PIN. La comparaison compare deux tableaux de 4 nombres allant de 0 à 9 : d'un côté le code candidat entré dans le terminal, de l'autre le bon code qui est sauvegardé dans la carte à puce. L'attaquant va alors réaliser l'attaque suivante :

1. Il essaie le code PIN 0000, tout en mesurant le temps de réponse de la comparaison. S'il est rapide, le premier chiffre est bien 0, sinon il réessaye avec 1000 puis 2000 etc.
2. Une fois le premier chiffre trouvé, il cherche le deuxième en mesurant le temps de réponse avec le premier chiffre correct. Si le premier chiffre est 1, il essaie successivement 1000, 1100, 1200, etc.
3. Et ainsi de suite jusqu'à retrouver les 4 chiffres.

Exercice 10.1 - *Nombre d'essais de code PIN*

Combien d'essais sont nécessaires au maximum sans et avec la fuite d'information temporelle dans le cas du code PIN à 4 chiffres ?

La question se pose donc de comment réaliser la comparaison sécurisée de deux tableaux de données, par exemple dans le but de comparer le code PIN candidat, entré par l'utilisateur, et le bon code PIN. Lorsque l'on compare deux tableaux, il faut comparer les éléments du tableau deux à deux. Une implémentation naïve retourne que les tableaux ne sont pas identiques dès que deux éléments différents sont trouvés. C'est fonctionnellement correct mais alors un attaquant est capable de savoir le premier indice tel quel les éléments du tableau sont différents : une information qu'il n'avait pas auparavant.

Exercice 10.2 - *Comparaison de tableaux*

Trouvez la faiblesse de cet algorithme de comparaison de code PIN. Corrigez-le.

```
bool compare_arrays(const uint8_t* a, const uint8_t* b, size_t len) {
    for(size_t i = 0; i < len; i++) {
        if (a[i] != b[i]) {
            return false;
        }
    }
    return true;
}
```

40 essais restent supérieurs à la limite usuelle de 3 essais permises par la carte à puce. Toutefois, il faut penser à l'échelle d'un réseau criminel : s'ils ont volé 10 000 cartes, avec 3 chances sur 10 000, ils pourront déverrouiller 3 cartes. Mais avec 3 chances sur 40, ils pourront déverrouiller 750 cartes. Un bien meilleur retour sur investissement pour les criminels.

D'autre part, sur des implémentations insuffisamment protégées, il est possible de mener une attaque par **arrachage**. Si l'attaquant détecte au cours de la comparaison, d'une quelconque manière, que le code PIN est incorrect, il peut arracher la carte de son lecteur. Ou plus vraisemblablement, couper l'alimentation de la carte, pour que le compteur ne soit pas décrémenté. Une bonne implémentation de la vérification du code PIN ne permet pas cette attaque.

Les attaques par dépendance temporelle dans la microarchitecture Les attaques microarchitecturales ont une partie dédiée. Une description de l'exploitation de la dépendance temporelle peut être lue dans le chapitre 22.

10.2.2 L'échantillonnage matériel

L'observation des données peut être réalisée physiquement par échantillonnage matériel (*probing* en anglais) : l'attaquant connecte un dispositif de mesure directement sur le fil ciblé. Si le fil est à l'extérieur de la puce, c'est relativement facile. Si le fil est une piste métallique du circuit intégré, la difficulté augmente. Toutefois, il est possible de se connecter, voire d'éditer un circuit intégré à l'aide d'un focused ion beam (FIB) (le terme français, sonde ionique focalisée, n'est jamais utilisé). Cet outil génère des ions de gallium et les focalise à l'aide d'un champ électromagnétique sur la cible de manière extrêmement précise. La collision permet d'enlever les atomes en surface de la cible, avec une résolution de l'ordre de 1 nm.

L'échantillonnage matériel est une technique invasive : on crée un trou jusqu'au fil cible, que l'on remplit d'un matériau conducteur, lui-même connecté au dispositif de mesure. Il est donc possible de lire directement un signal, par exemple celui qui fait transiter la valeur d'une clé cryptographique.

Un FIB est un instrument lourd, hors de portée d'un particulier, mais utilisé régulièrement par des organisations : agences, mafia, centre d'évaluation, etc.

10.2.3 La consommation de courant

10.2.3.1 Modèles de fuite

Une porte logique CMOS n'a pas une consommation de courant constante dans le temps, on considère par approximation le modèle suivant.

$$conso(porte) = sortie \cdot c_1 + \overline{sortie} \cdot c_0 = sortie \cdot (c_1 - c_0) + c_0. \tag{10.1}$$

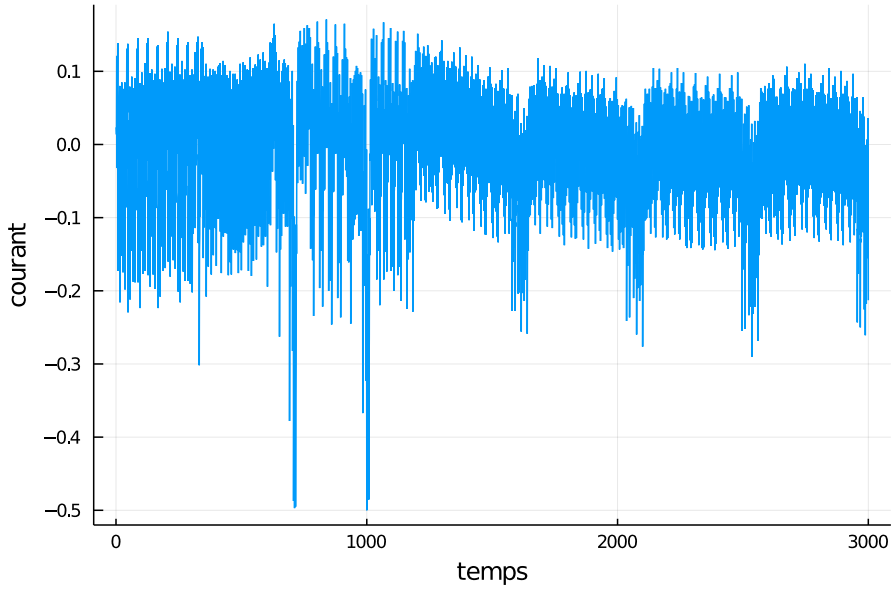


FIGURE 10.1 – Un exemple de mesure de consommation de courant : une trace.

L'équation 10.1 établi deux valeurs de consommation, si l'état de la sortie est à 0 ou si l'état de la sortie est à 1. Il s'agit d'un **modèle de consommation statique** : ce modèle considère que la porte consomme de l'énergie au repos.

Un autre modèle est celui de la **consommation dynamique** : la porte ne consomme de l'énergie que lorsqu'elle change d'état.

$$conso(porte) = \overline{sortie_{t-1}} \cdot sortie_t \cdot c_{\uparrow} + sortie_{t-1} \cdot \overline{sortie_t} \cdot c_{\downarrow}. \tag{10.2}$$

Ces deux modèles sont couramment utilisés. Normalement, un circuit CMOS est conçu pour minimiser les courants de fuite. Ainsi, suivant la technologie de la puce, le modèle statique devrait être négligeable par rapport au modèle dynamique. Ceci est moins vrai pour des circuits récents avec des technologies très fines.

Toutefois, pour améliorer la vitesse d'un circuit intégré, la technique de la précharge est souvent utilisée pour lutter contre les capacités parasites dans le circuit. Cette technique consiste à remettre le potentiel d'un fil à une valeur prédéterminée, souvent à l'état 1 ou à une valeur intermédiaire, avant le front montant. Ainsi si la sortie de la porte logique reste théoriquement à 0 d'un cycle sur l'autre, en pratique il peut y avoir des changements d'état à cause de la précharge.

Dans ce scénario, même si le modèle dynamique est celui qui décrit précisément la consommation de la porte logique, nous observerons en pratique un modèle statique du fait de la précharge.

Définition 10.1 - *Poids de Hamming*

Le poids de Hamming d'une valeur est le nombre de bits à 1 dans sa représentation binaire. Par exemple $HW(5) = 2$ puisque $5 = 101_2$.

Ce modèle de consommation statique est affine. En l'approximant par un modèle linéaire ($c_0 = 0$, possible, car on analyse la variation de consommation de courant et non sa valeur absolue), on peut facilement additionner la consommation de plusieurs portes logiques. Ainsi la consommation de courant d'une valeur v présente dans le circuit, portée par un ensemble de

portes logiques, est proportionnelle à $HW(v)$ où HW est le poids de Hamming (cf. définition 10.1).

Dans le cas où le modèle de consommation dynamique prévaut, on le modélisera par $HD(x_t, x_{t-1})$ la distance de Hamming entre la donnée au temps t et sa valeur au temps $t-1$. La distance de Hamming est le nombre de bits qui ont variés entre ces deux valeurs ($HD(x_t, x_{t-1}) = HW(x_t \oplus x_{t-1})$).

D'autres modèles de consommation peuvent être utilisés, et même être plus précis dans certains cas, mais le modèle de fuite le plus courant reste le poids de Hamming.

10.2.4 Les émissions électromagnétiques

Mesurer la consommation électrique du circuit pose un problème : nous cherchons à mesure la consommation de quelques transistors, mais nous obtenons une mesure de l'ensemble du circuit. La corrélation est efficace, elle marche tout de même dans de nombreux cas.

Toutefois, parfois, nous avons besoin de la consommation que d'une partie restreinte du circuit, car la partie qui nous intéresse est noyée dans le bruit. Ou plus simplement, nous n'avons pas accès au fil qui alimente la puce cible en énergie.

Dans ce cas, nous pouvons mesurer les émissions électromagnétiques à la place de la consommation de courant.

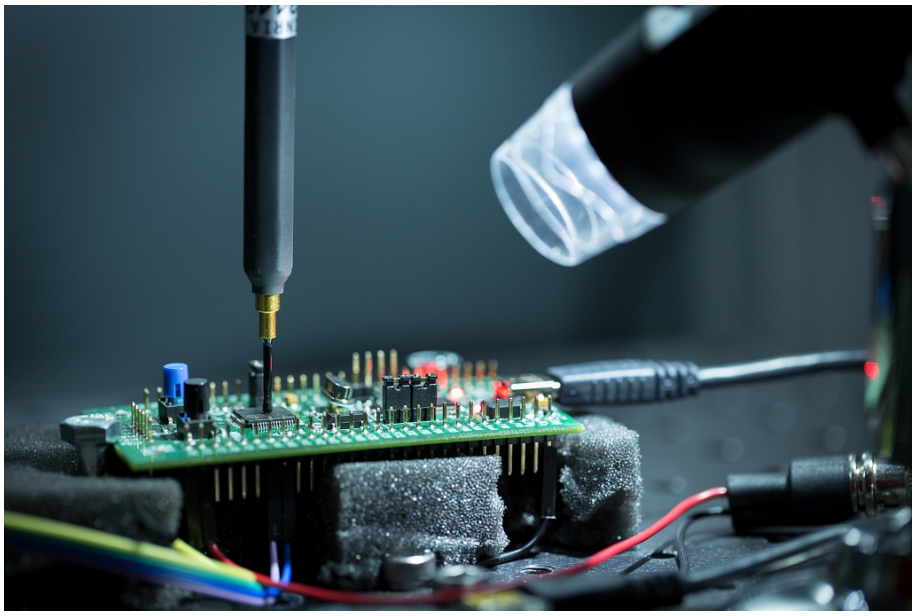


FIGURE 10.2 – Une antenne EM positionnée au-dessus de la puce cible. *© Inria / Photo C. Morel.*

La signature électromagnétique des logiciels De même que la consommation de courant d'une puce, le rayonnement électromagnétique permet de caractériser le logiciel en cours d'exécution. En effet, le signal dépend des données manipulées, mais aussi du fonctionnement de la microarchitecture : instructions, signaux de contrôle, etc. De plus, des motifs d'instructions sont régulièrement répétés, par exemple les itérations d'une boucle.

La conséquence est que la structure du logiciel peut être observée avec l'apparition de signaux ayant des fréquences plus faibles que celle de l'horloge.

Un spectrogramme, comme sur la figure 10.3, permet donc de caractériser le logiciel en cours d'exécution. Le rayonnement électromagnétique est une signature du logiciel exécuté.

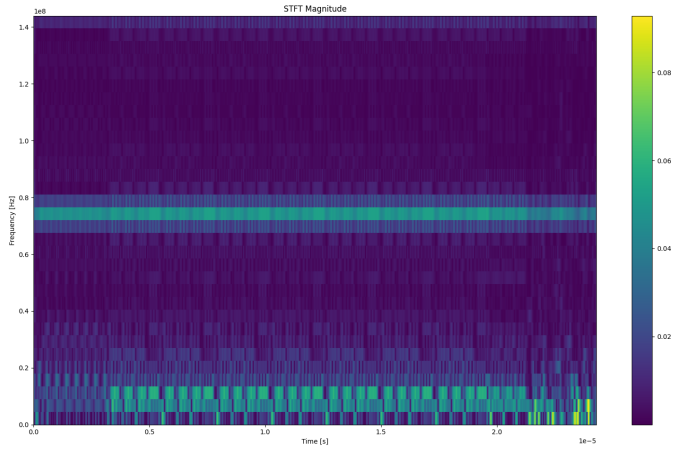


FIGURE 10.3 – Le spectrogramme des émissions électromagnétiques d'un Raspberry Pi 3 en cours d'exécution. Il est possible de détecter l'activité logicielle sur la puce à partir de ses émissions. Crédit Damien Marion et Duy-Phuc Pham.

10.2.5 Autres méthodes moins courantes

Les méthodes précédentes sont les plus courantes mais pas les seuls.

Par exemple, le son peut être utilisé comme source d'information. En particulier, certaines technologies de condensateurs ou d'inducteurs voient les composants vibrer en fonction des variations de tension [18]. Ces vibrations peuvent ensuite être mesurées à l'aide d'un microphone, si le signal est dans la bonne plage de fréquence (ce qui est en pratique rarement le cas).

Une technique plus utile, mais rare car difficile à mettre en œuvre, est l'analyse de la photoemission. Les variations d'énergie (courant et tension) dans les semiconducteurs peuvet mener à l'émission de photons. Ceux-ci peuvent être capté, offrant ainsi une image des signaux dans la puce [33].

10.3 Exploitation des mesures

10.3.1 Simple Power Analysis sur une exponentiation modulaire

L'analyse simple d'un paramètre physique [23], notée SPA, vise à déterminer directement, à partir d'une observation, par exemple de la consommation de courant, lors d'une exécution normale de l'algorithme, des informations sur le calcul effectué ou les données manipulées [11, 52, 23].

La SPA de Kocher vise l'algorithme `Square and Multiply` (voir FIGURE 10.4) très utilisé en cryptographie. Le but de cet algorithme est de calculer une exponentiation modulaire en réduisant au maximum le nombre de calculs possibles. Dans l'exemple suivant, e est l'exposant et n le modulo :

$$x^e \mod n$$

Pour cela l'exposant est écrit en binaire. Dans l'écriture si l'on a un 1 une multiplication et une élévation au carré sont calculées. Si l'on a un 0 seule une élévation au carré est calculée. Aussi cette suite de calculs est très caractéristique et peut s'observer directement sur une trace de consommation de courant comme l'illustre la FIGURE 10.5.

```
1:  procedure SQUARE AND MULTIPLY(x,e,n)
2:      r = 1
3:      b = (e)_2
4:      for i bit de poids fort au bit de poids faible do
5:          if b[i] = 1 then
6:              r = r^2 · x  mod n
7:          else
8:              r = r^2  mod n
9:          end if
10:     end for
11:     return r
12: end procedure
```

FIGURE 10.4 – Algorithme Square and Multiply

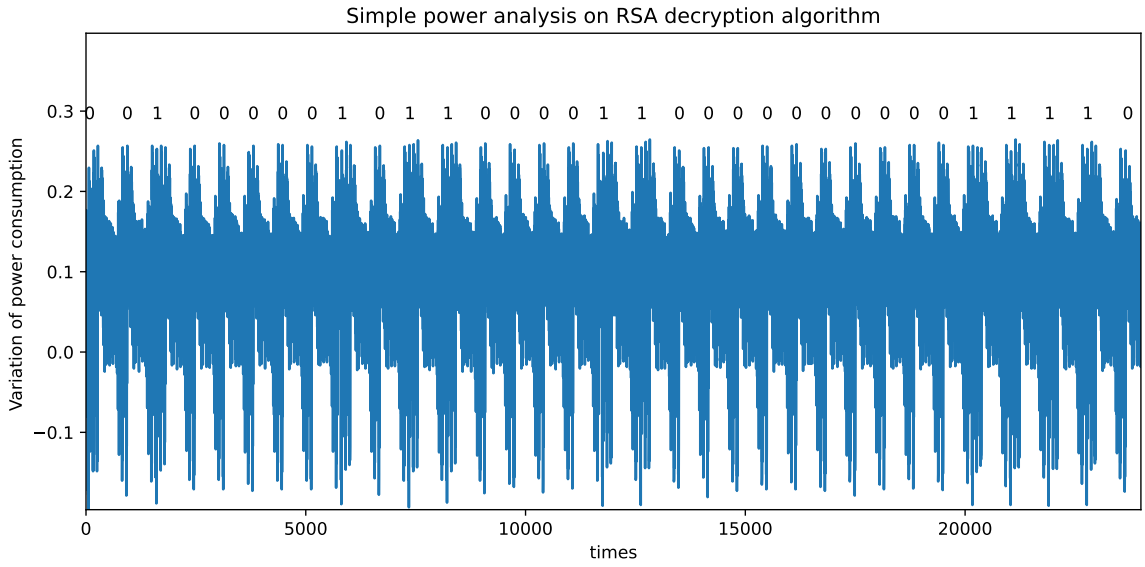


FIGURE 10.5 – SPA contre l'algorithme Square and Multiply utilisé dans algorithme RSA (illustration réalisée en projet par Jonathan Amatu, Maël Leproust, Salim Sama Mola et Alexis Prou étudiants à IMT-Atlantique, année 2024)

10.3.2 Correlation Power Analysis sur AES

Le principe de la correlation power analysis (CPA) [5] consiste à corréler la consommation de courant ou l'émission électromagnétique avec des valeurs intermédiaires qui dépendent du secret recherché.

L'advanced encryption standard (AES) est un algorithme cryptographique couramment utilisé pour le chiffrement symétrique. Il s'agit d'un réseau de substitution-permutation travaillant par blocs de 16 octets.
Comme illustré dans la figure 10.6, l'algorithme est constitué de 4 sous-fonctions appelées plusieurs fois, en rondes. Ces 4 sous-fonctions sont `AddRoundKey`, `SubBytes`, `ShiftRows` et `MixColumns`. Elles opèrent sur le `State`, qui est l'état interne sous forme de matrice d'octets 4×4, progressivement transformée.

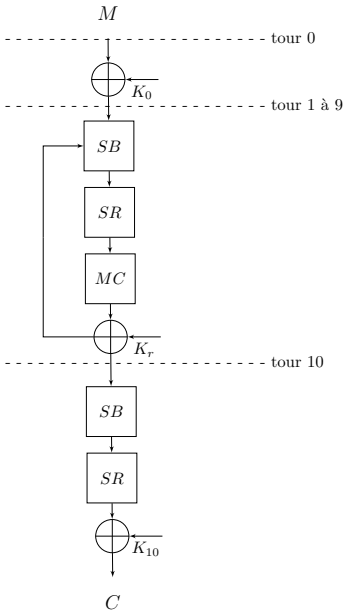


FIGURE 10.6 – La structure de l'AES.

10.3.2.1 Campagne de mesures

L'objectif de l'attaquant est de retrouver la clé. Cette clé est réutilisée à plusieurs endroits dans l'algorithme : elle passe par un algorithme dédié de génération de clés (*key schedule* en anglais) qui dérive des sous-clés à partir de la clé maître. Cette dérivation est une bijection publiquement connue. Si l'attaquant retrouve une sous-clé, il peut retrouver la clé maître.

Nous allons donc cibler une sous-clé en particulier. Prenons le cas de la CPA sur le premier tour en exemple. Au début de l'algorithme AES, le texte clair (*plaintext* en anglais) est d'abord xoré avec la clé maître, qui est la clé du tour 0. Ensuite, le `State` passe dans la fonction `SubBytes`.

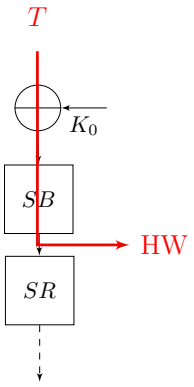


FIGURE 10.7 – Le début de l'AES est la partie qui nous intéresse. Nous voulons relier le texte clair avec la mesure prise en sortie de `SubBytes`.

Ceci constitue le chemin d'attaque : nous comparons les données mesurées à l'oscilloscope avec la valeur intermédiaire prédite après la fonction `SubBytes`.

Nous mesurons un certain nombre n de traces, une trace étant un vecteur de données mesurées à l'oscilloscope correspondant à la consommation de courant évoluant au cours du temps lors du calcul pour un texte clair donné.

En particulier, le moment où la valeur intermédiaire d'intérêt est manipulée par le processeur

doit être présent dans cette trace. Lors de ces n exécutions de l'algorithme, le texte clair est tiré aléatoirement pour chaque trace. Nous obtenons une matrice de mesures $(M_{e,t})$, où e ($e \in [\![0; n-1]\!]$) est l'indice du texte clair, et t représente le temps dans chaque trace.

10.3.2.2 Prédictions

La fonction `SubBytes` est composée de 16 fonctions `SBox` (notée parfois SB) identiques, appliquées en parallèle à chaque octet du `State`. Nous pouvons donc réaliser notre attaque octet par octet, ce qui permet de retrouver chaque octet de la clé indépendamment des autres. On parle de la stratégie **diviser pour régner**. C'est ce que nous indique l'équation 10.3, qui s'applique pour chaque octet i du `State`, avec T le texte clair, X la valeur intermédiaire après `SubBytes` et K_0 la clé maître.

$$\forall i \in [\![0,15]\!], X^i = SB(T^i \oplus K_0^i) \tag{10.3}$$

Concentrons-nous sur le premier octet : $X^0 = SB(T^0 \oplus K_0^0)$ relie notre texte clair connu, la clé secrète recherchée et la valeur intermédiaire dont nous avons mesuré une fuite d'information.

Pour prédire la valeur mesurée à l'oscilloscope, nous devons établir un **modèle de fuite**, une fonction modélisant la fuite d'information à partir de la valeur intermédiaire. Pour les raisons évoquées dans la sous-sous-section 10.2.3.1, nous utilisons généralement le poids de Hamming HW.

En combinant ces éléments, nous établissons la matrice P^0 des prédictions pour l'octet 0.

$$P^0_{e,k} = HW\left(SB(T^0_e \oplus k)\right), \tag{10.4}$$

pour l'exécution d'indice e et l'hypothèse de clé k ($k \in [\![0;255]\!]$).

10.3.2.3 Confrontation

Il faut maintenant confronter les données mesurées à l'oscilloscope avec nos prédictions.

Pour un couple (t,k) donné correspondant au temps où la valeur intermédiaire ciblée est calculée par le processeur et à la bonne hypothèse de clé, la colonne $P_{*,k}$ dans la matrice de prédiction et la colonne $M_{*,t}$ dans la matrice de mesures doivent correspondre.

Ainsi, pour chaque couple, nous calculons la corrélation de Pearson comme dans l'équation 10.5.

$$C_{t,k} = corr(P_{*,k}, M_{*,t}) \tag{10.5}$$

Le résultat de ces calculs peut être visualisé comme dans la figure 10.8.

Il suffit de répliquer cette procédure pour chaque octet de clé pour reconstituer celle-ci intégralement.

Exercice 10.3 - *CPA au dernier tour*

Il est possible de réaliser une CPA sur le dernier tour de l'AES. Donner l'équation permettant de générer la matrice de prédiction.

Ici, la corrélation de Pearson est utilisée comme **distingueur**, permettant de distinguer la bonne hypothèse de clé parmi les mauvaises.

D'autres distingueurs sont possibles, par exemple :

— la corrélation [5],
— l'information mutuelle [14],
— la composante principale [41],
— le discriminant linéaire [37].

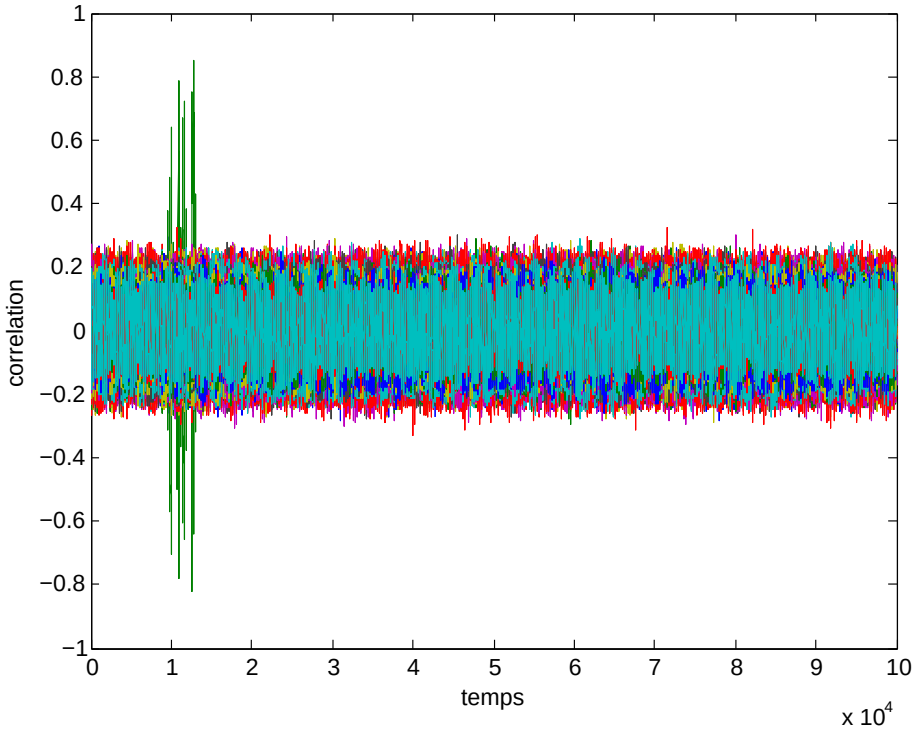


FIGURE 10.8 – 256 courbes correspondant à autant d'hypothèses de clé. Pour certains instants précis, la corrélation ressort du lot pour une seule hypothèse : la bonne.

10.3.3 Les attaques par caractérisation

Les **attaques par caractérisations**, appelées aussi attaques profilées ou attaques par apprentissage (Template attacks [49, 50, 8, 13, 53, 55]) sont les plus efficaces en termes d'information. En effet il n'y a pas de modèle de fuite choisi à prioris par l'attaquant (comme le poids de Hamming dans la CPA) mais un apprentissage de la fuite physique, et donc une modélisation optimale de cette fuite. Pour réaliser une telle attaque, on suppose que l'attaquant possède un circuit identique à celui qu'il souhaite attaquer, dont il est l'utilisateur légitime.
Il y a donc deux circuits :

1. un circuit qui est la cible de l'attaque,
2. un circuit de référence identique sur lequel l'attaquant peut effectuer des modifications.

Il y a deux grandes étapes à une attaque par apprentissage.

1. **La caractérisation** ou phase d'apprentissage est la première étape. Il s'agit de caractériser le circuit de référence par rapport à une fuite d'information. Les mesures obtenues sont en fait une "signature" de l'exécution du programme au sein du circuit (instructions exécutées, données manipulées, ...). La caractérisation permet d'apprendre la fuite significative et les caractéristiques du bruit. Sans être capable d'identifier exactement une instruction ou des données manipulées, le biais statistique présent dans un signal peut être suffisant pour une exploitation. Beaucoup de mesures doivent être obtenues, le nombre dépendant du bruit présent dans le signal.
2. L'**attaque** en soit est la deuxième étape. L'attaquant réalise des mesures sur le circuit cible et va les confronter à sa caractérisation pour retrouver un secret. Selon les résultats expérimentaux, il est possible de distinguer l'exécution d'une instruction à la place d'une autre. Il est même possible de retrouver la valeur d'une donnée chargée dans un registre avec une probabilité non négligeable.

Les algorithmes cryptographiques sont très vulnérables à ce type d'attaque, mais ils ne sont pas les seuls. Une attaque d'une vérification de code PIN par caractérisation a été publiée [20].

10.4 Contremesures

Prévenir les fuites d'information, que ce soit la variation temporelle ou les émissions électromagnétiques, peut être réalisé avec différentes techniques.

10.4.1 Exécution en temps constant

La variation temporelle peut être contrée en exécutant le code en temps constant. Par exemple, la comparaison de 2 tableaux en temps constant peut être réalisée comme suit.

```
bool compare_arrays(uint8_t* arr1, uint8_t* arr2, int len) {
    uint8_t result = 0;
    for(int i = 0; i < len; i++) {
        result |= arr1[i] ^ arr2[i];
    }
    return result == 0;
}
```

Dans cet exemple, nous avons enlevé tous les branchements. Tous les octets des tableaux sont lus, même si les premiers octets sont différents. Une exécution en temps constant implique de toujours exécuter une fonction avec son temps worst case execution time (WCET).

Mais même dans cet exemple, quelles **garanties** avons-nous que cette fonction est en temps constant ? Pouvez-vous être certain que l'exécution du xor ne se fasse pas en un temps variable dépendant des données en entrée ? Aujourd'hui, nous nous reposons sur des considérations ad hoc : si cela semble en temps constant, on se dit que c'est suffisant ; surtout parce que l'on ne peut pas faire mieux.

De plus en plus, les contraintes de l'exécution en temps constant se diffusent au matériel. Par exemple, le jeu d'instructions RISC-V a introduit l'extension Zkt qui est une garantie que l'exécution de certaines instructions se fasse en temps constant. Charge au matériel d'implémenter cette garantie.

10.4.2 Bouclier métallique

Pour limiter les émissions électromagnétiques, il est possible de rajouter un bouclier métallique : les pistes métalliques en surfaces sont alors utilisées pour faire écran au rayonnement. Toutefois, il n'est pas encore possible de réaliser une cage de Faraday en utilisant ces pistes. La protection offerte par leur présence est limitée.

10.4.3 Génération de bruit et désynchronisation

Pour améliorer l'intérêt d'un bouclier métallique, il ne faut pas le laisser passif. Une meilleure possibilité est d'envoyer des signaux dans ces pistes dans le but de créer du bruit. Un rapport signal sur bruit plus faible complique la tâche de l'attaquant.

Cette fonctionnalité se combine bien avec l'utilisation du bouclier contre les attaques par injection de fautes (cf. sous-sous-section 11.4.2.1).

Cette génération de bruit peut également avoir lieu au sein de la puce elle-même pour compliquer les mesures.

Une variation de ce principe est la désynchronisation : rendre le signal d'horloge volontairement chaotique, en respectant les contraintes de temps dans le circuit. Cela rends plus difficile la synchronisation des traces par l'attaquant.

10.4.4 Masquage

Une autre contremesure efficace est le masquage, appliquée au niveau algorithmique.

Le principe est de ne jamais réaliser un calcul à partir du secret, mais de le randomiser d'abord.

Cas des fonctions linéaires Par exemple, soit f une fonction linéaire. Au lieu de calculer la fonction appliquée au secret s directement, c'est-à-dire de calculer $o = f(s)$. Il faut

1. Générer un masque aléatoire m, de même taille que s.
2. Le secret est alors réparti en deux parts : $s_1 = m$, $s_2 = s \oplus m$. On a bien $s = s_1 \oplus s_2$. Les deux parts sont randomisées.
3. On calcule la fonction f deux fois : $o_1 = f(s_1) = f(m)$ et $o_2 = f(s_2) = f(s \oplus m)$. La fonction n'est donc jamais appliquée au secret directement.
4. Si nous avons besoin de réassembler o, il suffit de calculer $o = o_1 \oplus o_2$.

Dans ce cas, le masquage marche grâce à la linéarité de f :

$$o = f(s_1) \oplus f(s_2) = f(m) \oplus f(s \oplus m) = f(m) \oplus f(s) \oplus f(m) = f(s)$$

Dans le cas de l' AES, ce cas s'applique `ShiftRows` et `MixColumns` qui sont toutes des fonctions linéaires. La solution est beaucoup moins élégante pour `SubBytes` qui est non linéaire.

Cas des fonctions non linéaires Pour une fonction non linéaire, il n'y a pas de méthode universelle : il faut adapter le schéma de masquage à la fonction à protéger. Dans le cas de la fonction `SubBytes` de l'AES, il est possible d'adapter le calcul d'une SBox. Il y a en fait de nombreuses façons de faire, voir par exemple [31].

Pour protéger une SBox, il est possible de précalculer la table $T[x,y]$, à deux dimensions où x et y sont deux octets.

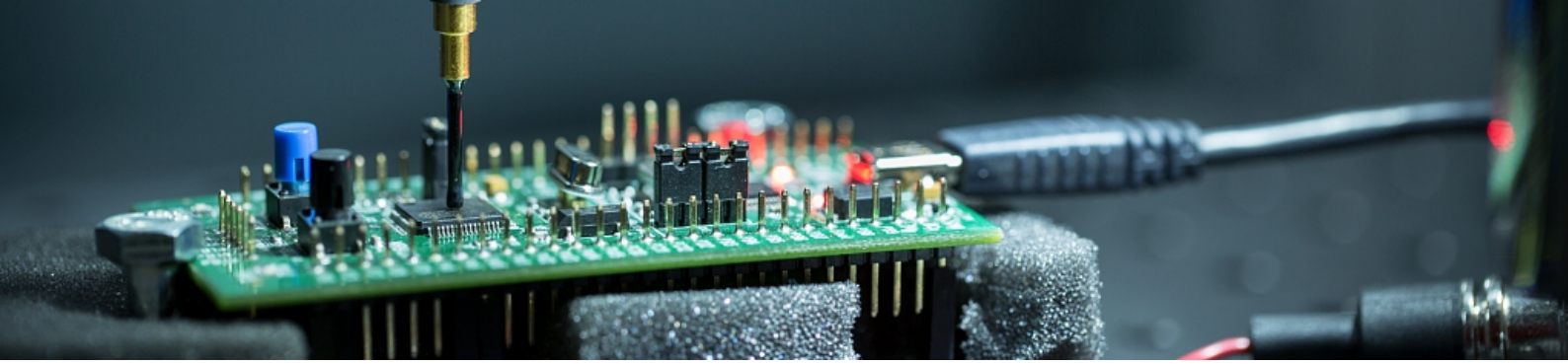
$$T[x,y] = SB(x \oplus y) \oplus y$$

Cette table prend $256 * 256 = 64$ Kio, soit une place trop grande pour un petit microcontrôleur, mais acceptable pour de nombreux systèmes. On utilise la table avec les bonnes parts en x et y :

$$o_1 = s_1$$
$$o_2 = T[s_2, s_1] = SB(s_2 \oplus s_1) \oplus s_1 = SB(s \oplus m \oplus m) \oplus m = SB(s) \oplus m$$

Attaques d'ordre supérieur Mais malheureusement, l'histoire ne s'arrête pas là. Il est possible pour l'attaquant de surmonter le masquage en combinant les fuites en deux points distincts. Par exemple si l'attaquant est capable de mesurer les fuites pour $f(s_1)$ ET $f(s_2)$, il peut dans certains cas les recombiner pour retrouver s.

Il faut alors partager s en 3 parts (avec deux masques aléatoires indépendants), on parle alors d'un schéma de masquage d'ordre 2. L'attaquant peut surmonter ce schéma en combinant 3 fuites, etc.



11. Les injections de fautes

En plus des difficultés de conception qui peuvent entrainer des défauts de fonctionnement, tout circuit intégré est susceptible d'être fauté par des évènements extérieurs, principalement de 3 natures : photonique, électromagnétique et du fait des particules chargées. Chacun de ces phénomènes a différentes conséquences.

Dans ce chapitre, nous verrons les fautes du point de la sécurité informatique. Attention, les fautes sur les circuits intégrés sont aussi très étudié dans le contexte de la défaillance accidentelle. Dans cet autre contexte, le vocabulaire n'est pas toujours le même que lorsque l'on discute de sécurité.

11.1 La physique des fautes

Commençons par comprendre comment les fautes apparaissent. Par définition, il s'agit d'un évènement anormal qui peut avoir de nombreuses origines.

> **!** Le sens d'une erreur, d'une faute et d'une défaillance dépend du contexte dans lequel on l'utilise. Pour les attaques matérielles, l'erreur est l'effet transitoire et la faute est la matérialisation de l'erreur dans un registre. Une défaillance dans ce contexte désigne plutôt un système ou un sous-système qui ne répond plus.

11.1.1 Les évènements

Historiquement, les fautes étaient avant tout des perturbations environnementales, avant de devenir un outil pour les attaquants. On parle d'évènements uniques lorsque l'on rencontre une perturbation temporaire, que son effet soit lui-même temporaire ou non.

Un single-event upset (SEU) est une perturbation du circuit par une particule isolée dont l'effet ne dure pas dans le temps. Un exemple est une charge électrique se diffusant dans un fil. Le courant en est temporairement modifié, mais se rétablit très vite à sa valeur nominale.

À l'inverse par exemple une perturbation single-event latchup (SEL) peut provoquer une perturbation ne se dissipant pas d'elle-même : la valeur reste erronée dans le temps, parfois le circuit est détruit, ou bien il est nécessaire d'éteindre l'alimentation pour arrêter l'effet de verrou.

11.1.2 Perturbation d'alimentation et perturbation d'horloge

En plus des perturbations environnementales que nous verrons par la suite, il est possible d'injecter des perturbations directement dans le circuit, volontairement par un attaquant ou non. Par exemple lorsque le réseau électrique est instable, des perturbations de l'alimentation apparaissent.

Pour que le circuit soit fonctionnel, il est nécessaire que la période de l'horloge du circuit synchrone soit supérieure au temps de propagation du signal sur le chemin critique. Dans le cas contraire, il y a un risque qu'il soit mal mémorisé puisqu'il n'est pas stabilisé à l'entrée du registre suivant. De plus, le signal doit être stable durant une durée fixe, dépendante de la technologie, avant le front montant : c'est le temps de setup. De même, il doit également rester stable après le front montant : c'est le temps de hold. Les contraintes sur la période de l'horloge peuvent se traduire par les équations 11.1, illustrée sur la figure 11.1, en négligeant le temps de propagation entre l'horloge et la sortie Q de la bascule D.

$$t_{\text{clock}} \geq t_{\text{critical path}} + t_{\text{setup}} \tag{11.1}$$

$$t_{\text{shortest path}} \geq t_{\text{hold}}$$

$$t_{\text{clock high}} \geq t_{\text{min width}} \tag{11.2}$$

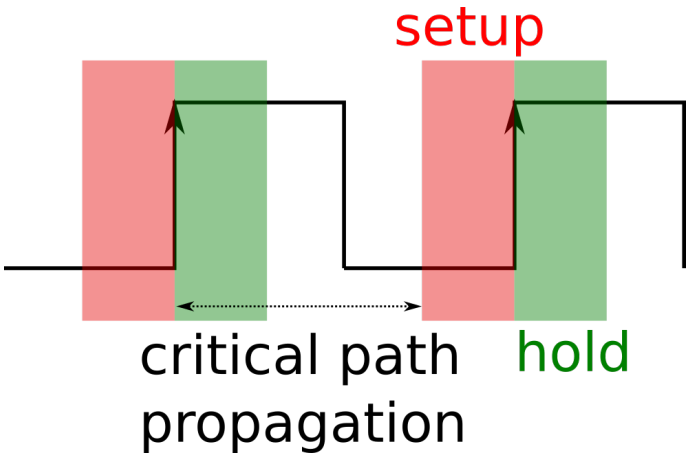


FIGURE 11.1 – Les contraintes sur la durée de la période de l'horloge. Après la propagation du signal sur tout le chemin critique, le signal doit rester stable durant toute la durée du temps de setup et du temps de hold en entrée du registre.

Il est possible de générer des erreurs en ne respectant pas les contraintes temporelles sur l'horloge énoncées par les équations 11.1. En particulier, lorsque le signal D varie en entrée d'un registre durant la durée du setup, on obtient de la **métastabilité**. La valeur réellement mémorisée est non-déterministe.

Exemple 11.1

Ce non-déterminisme peut être utilisé pour générer des valeurs aléatoires, notamment via des physically unclonable function (PUF).

Quelques exemples La tension d'alimentation a un effet direct sur la rapidité de changement d'état d'un transistor, ce qui entraine un changement de vitesse d'une porte logique. En effet, lorsqu'un signal en entrée d'une porte logique est modifié, il y a une certaine latence avant que cela impacte la sortie. Une tension d'alimentation plus élevée permet des circuits plus rapides, au contraire une tension plus basse augmente la latence de la propagation du signal.

Ainsi, réduire la tension d'alimentation d'un circuit ralentit la propagation du signal le long du chemin critique. Si elle est suffisamment réduite, il y a violation des contraintes temporelles et une faute peut apparaitre, via le timing fault model. De même, si on surcadence (*overclock* en anglais) l'horloge, le front montant peut arriver avant la stabilisation du signal sur D.

Il est également possible de manipuler directement le signal d'horloge, par exemple en générant un front montant inachevé. C'est à dire que le signal monte à une certaine tension intermédiaire et redescend tout de suite. Avec le bon niveau de front montant, lors de sa propagation dans le circuit, certains registres « verront » le front, d'autres ne le verront pas. Ce décalage peut créer des fautes, il s'agit ici de l'energy-threshold fault model.

Exemple 11.2 - *TRAITOR*

TRAITOR [10] est un injecteur de faute par perturbation d'horloge avec un modèle de faute physique spécifique. À l'aide d'un field programmable gate array (FPGA) à bas cout, une horloge malveillante est générée et transmise à la cible. Cela permet de générer des trains de fautes, jusqu'à plusieurs centaines, avec des taux de succès $> 99\%$.

11.1.3 Injection électromagnétique

Il est possible d'altérer les signaux d'alimentation, de masse localement à l'aide de l'injection EM.

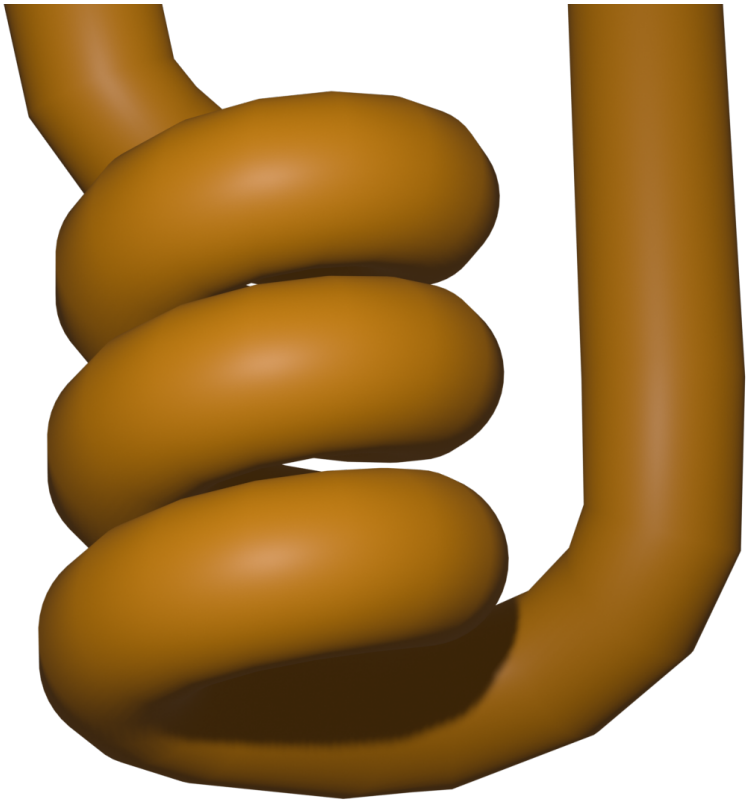


FIGURE 11.2 – Une sonde EM est un simple solénoïde, formé par quelques tours d'un fil de cuivre, parfois entourant un cœur en ferrite.

Le principe est d'utiliser un solénoïde, comme sur la figure 11.2, dans lequel nous faisons circuler un courant intense, mais très court. Le but est de ne perturber qu'une seule instruction, il faut donc obtenir une résolution temporelle de l'ordre du cycle d'horloge (1 ns pour un cœur tournant à 1 GHz).

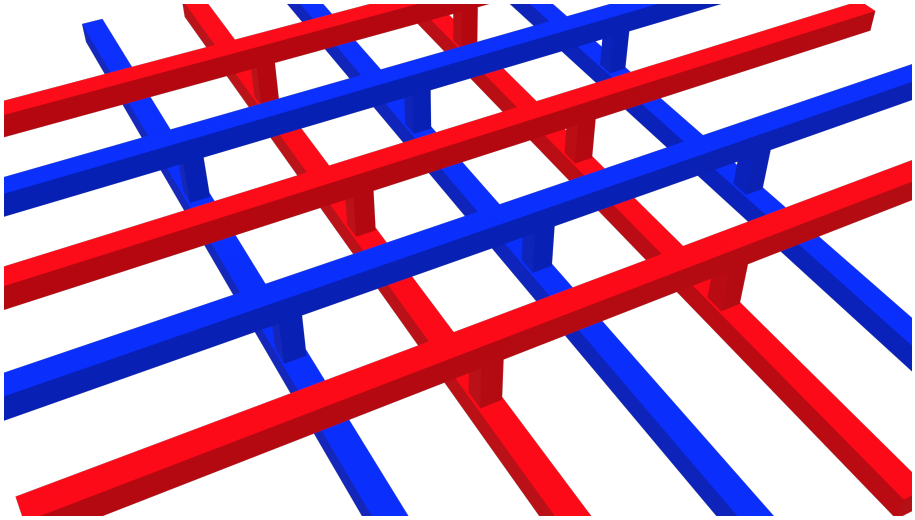


FIGURE 11.3 – Les grilles métalliques servant à fournir l'alimentation (VDD) et la masse forment des boucles du fait de leur structure tridimensionnelle. Ces boucles sont donc sensibles à l'induction électromagnétique.

La sonde EM permet d'initier un phénomène d'induction entre elle-même et les boucles formées par les signaux d'alimentation et de masse dans leur réseau de distribution respectif, illustré figure 11.3. L'effet de l'induction sera différent sur les deux signaux du fait des décalages spatiaux entre les boucles de VDD et les boucles de masse.

Le phénomène d'induction, induction de Neumann dans notre contexte, est la conséquence de l'équation de Maxwell-Faraday (équation 11.3).

$$\vec{rot}\vec{E} = -\frac{\partial \vec{B}}{\partial t}. \tag{11.3}$$

À partir de cette équation, l'intégration sur une boucle métallique C, on obtient la force électromotrice e via l'équation 11.4.

$$e = \oint_C \vec{E} \cdot d\vec{l} = -\frac{d\Phi}{dt} \tag{11.4}$$

Enfin le courant induit i est

$$i = \frac{e}{R}, \tag{11.5}$$

où R est la résistance de notre boucle.

En termes simples, la variation d'un champ magnétique à travers une boucle crée un courant en son sein.

Ces courants induits par l'injection EM sont nécessairement dissymétriques entre VDD et la masse, puisque la disposition des boucles correspondantes est différente. Ainsi l'injection EM altère la différence de potentiel entre VDD et la masse, pouvant entrainer une faute selon plusieurs modèles discutés dans la sous-section 11.2.1.

11.1.4 Perturbation au contact du substrat : body biasing

Pour que l'injection EM soit possible, il faut que les pistes métalliques soient tournées vers l'extérieur. Toutefois, certaines techniques de packaging comme le « flip chip » retournent la puce, n'offrant que le substrat à l'attaquant.

La technique de body biaising consiste à envoyer une impulsion électrique dans le substrat au niveau des transistors ciblés. Pour cela, on place une micropointe au contact du substrat, nécessitant le décapsulage de la puce. Comme nous l'avons vu dans la section 1.2, le substrat est normalement forcé à un potentiel déterminé pour le bon fonctionnement du transistor. En modifiant localement ce potentiel, le comportement des transistors à proximité est modifié.

11.1.5 Perturbation photonique

Lorsque l'on illumine un circuit non packagé avec certaines lumières, il est possible d'obtenir un effet photoélectrique induisant un courant dans certaines jonctions des transistors. Deux sources lumineuses sont particulièrement efficaces : les flashs au xénon de certains appareils photo, et l'utilisation de laser dans le contexte d'une attaque contre un circuit. Il suffit d'utiliser un packaging opaque à ces lumières pour s'en protéger, en supposant que l'attaquant ne l'enlève pas...

Exemple 11.3

Les premiers Raspberry Pi 2 pouvaient crasher lorsqu'on les prenait en photo, avec un flash au xénon. La cause est une puce mémoire dont le die n'était pas protégé de l'exposition lumineuse.

L'injection laser L'injection de fautes par laser est courante, car très efficace. Elle consiste à illuminer une partie du circuit cible avec un laser, après décapsulation pour mettre à nu le circuit. Comme illustré sur la figure 11.4, les photons du laser créent des paires électron-trou par effet photoélectrique. La recombinaison de ces paires, menant à un retour à l'équilibre, entraine un déplacement de charges, et donc à l'apparition d'un courant au niveau de la jonction.

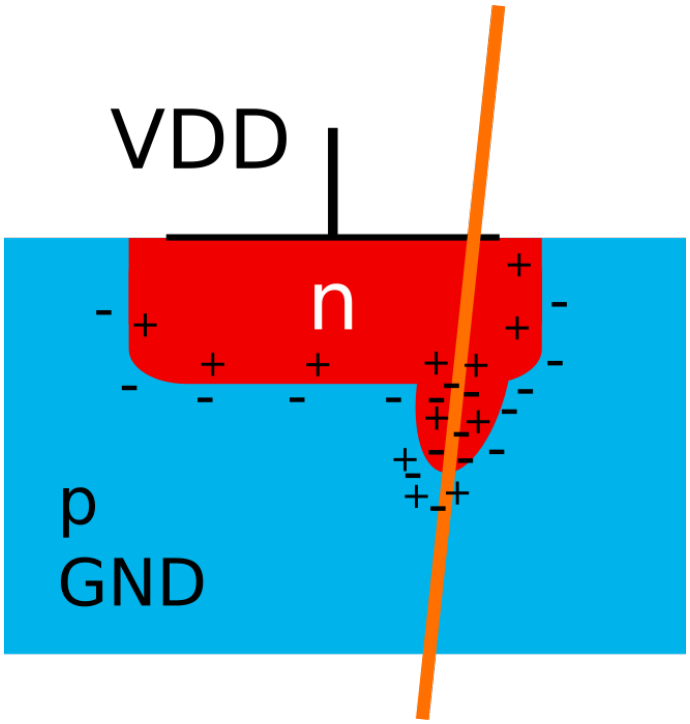
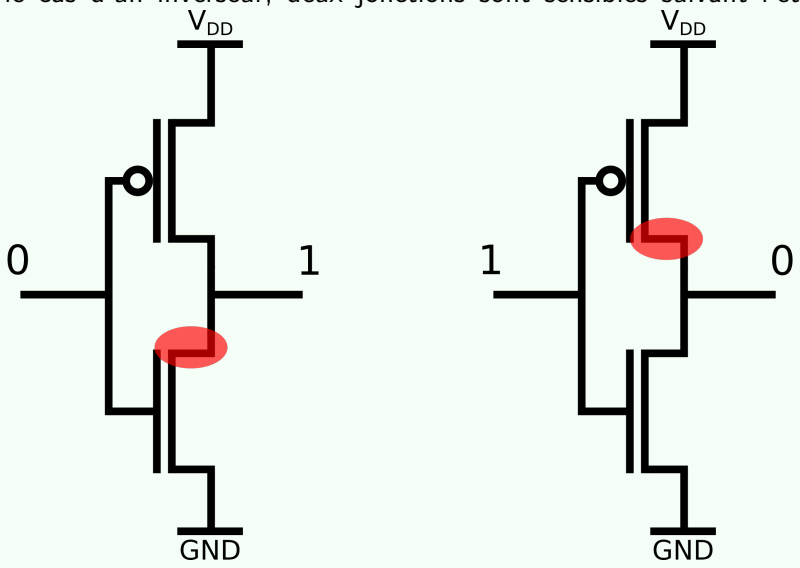


FIGURE 11.4 – Le rayon laser, en orange, crée des paires électron-trou au niveau de la jonction.

Suivant l'état du transistor et donc de la jonction, et le rôle du transistor, ce courant peut être mémorisé et donc mener à une faute.

Exemple 11.4 - *Injection laser sur un inverseur*

Dans le cas d'un inverseur, deux jonctions sont sensibles suivant l'état de la porte.



Si l'entrée de l'inverseur est à 0, il faut cibler le drain du NMOS : le courant induit par l'injection circule entre le drain, relié à la sortie, et le substrat relié à la masse. Nous avons alors relié la sortie, normalement à 1, à la masse. Ce qui crée une perturbation en mettant temporairement la sortie à 0.

De manière symétrique, le drain du PMOS est également sensible, car il relie la sortie au substrat, à V_{DD} dans ce cas-là.

En particulier, le laser est autant capable d'injecter une faute dans un circuit, séquentiel ou combinatoire, en fonctionnement, que dans une cellule mémoire au repos. Il est notamment possible d'altérer les données d'une cellule SRAM.

11.1.6 Perturbation avec des particules chargées

Tout environnement est traversé à tout moment de particules chargées. Une particule chargée qui entre en contact avec le circuit peut induire un courant menant à une faute. Le packaging est primordial pour s'en protéger.

Il existe de multiples causes de telles perturbations, mais certaines sont prépondérantes. Le rayonnement cosmique est principalement dû à des protons de haute énergie voyageant dans l'espace, qui se décomposes en différentes particules au contact de l'atmosphère. Ces particules sont particulièrement dommageables pour les circuits intégrés. Heureusement, l'atmosphère terrestre nous en protège pour la plupart. Mais, d'une part elles restent très présentes en altitude et dans l'espace, et d'autre part certaines peuvent quand même nous atteindre. Notamment les muons, qui bien que de durée de vie très faible (2,2 µs), peuvent atteindre le niveau de la mer du fait de la dilatation du temps du à leur vitesse relativiste. Les muons pénètrent les matériaux en profondeur et sont donc susceptibles de traverser le packaging de la puce. Ainsi un circuit destiné à être embarqué dans un avion, ou pour des trajets en haute montagne, devra être particulirèment durci. Même un circuit même au niveau du sol sera la cible d'erreurs du à ce rayonnement, quoi que de manière anecdotique.

La deuxième source importante de perturbation est le rayonnement alpha dû à une réaction nucléaire. C'est particules sont présentes en permanence dans notre environnement du fait de la radioactivité naturelle. Leur prépondérance est toutefois très variable selon les endroits. Les caves bretonnes, à cause de radon émis par les roches granitiques, sont particulièrement émettrices de ce rayonnement.

Exemple 11.5 - *Starfish Prime*

Le 9 juillet 1962, une bombe H d'une puissance de 1,4 Mt explose à une altitude de 400 km au dessus du Pacifique. Il s'agit de l'essai américain nommé **Starfish Prime**. Ayant eu lieu dans l'espace, il n'y a pas d'onde de choc détruisant tout sur son passage : cet essai a permis d'étudier les radiations émises par une telle explosion et son effet sur les circuits électriques.
Sur les 24 satellites présent en orbite à cette période, 8 furent détruits par les radiations dues à l'explosion. Les effets radiatifs durèrent jusqu'à 5 ans en altitude. L'explosion entraina des dommages sur les infrastructures électriques à Hawaii, à $\approx$ 1500 km de là. Entre 1% et 3% des lampadaires furent notamment détruits malgré des circuits électriques beaucoup moins sensibles qu'aujourd'hui. Les conséquences furent beaucoup plus lourdes qu'attendu, et conduisirent à l'annulation des essais spatiaux suivants déjà planifiés : ils risquaient d'interdire l'accès à l'espace pour l'espèce humaine. Pour cette raison, les résultats de cet essai ont été en grande partie déclassifiés.

L'explosion de Starfish Prime telle qu'observée depuis Honolulu.

L'impulsion électromagnétique (EMP pour *electromagnetic pulse*) créée par une détonation atomique est complexe et constituée de plusieurs composantes. Elles sont au nombre de 3 et appelé E1, E2 et E3.

— E1 : c'est la composante la plus rapide. Un champ électromagnétique bref et intense est émis entrainant des courants induits dans les éléments conducteurs et la destruction potentielle des éléments semiconducteurs. Il est à son apogée après 200 ns et dure de l'ordre de 1 µs. Il est créé par les rayons gammas issus de la fission ou fusion initiale qui ionise les atomes rencontrés, c'est à dire qui en éjecte les électrons. C'est ce mouvement d'électrons qui crée le champ électromagnétique.
— E2 : cette composante dure de 1 µs à 1 s. Similaire à E1 en moins intense et plus long, elle est également créée par des rayons gamma, issus cette fois-ci des neutrons présents dans la bombe. Les caractéristiques de E2 se rapprochent de celles de la foudre. Il est donc plus facile de s'en protéger. Toutefois, les mécanismes de protection ont pu être détruits par E1.
— E3 : cette dernière composante dure de 10 s à plusieurs centaines de secondes. Cette impulsion est la conséquence de l'interaction de l'explosion avec l'atmosphere et le champ magnétique terrestre naturel. Le radiations gamma précédentes ionisant l'atmosphère à une échelle suffisante pour perturber le champ magnétique terrestre. Les caractéristiques sont similaires à une tempête solaire. De par sa très

longue longueur d'onde, E3 interagit plutôt avec les très longs conducteurs comme les lignes électriques (ou télégraphiques, cf Évènement de Carrington).
On peut voir à travers cet exemple que concevoir un circuit résistant à toutes ces composantes très différentes n'est pas chose aisée. De plus, il existe de nombreuses conceptions de bombes atomiques, avec des profils de radiations différents.

11.2 Les modèles de fautes

Nous venons de voir comment injecter des fautes expérimentalement. Caractériser l'effet des fautes sur le circuit est complexe, et dépends très largement de là où l'on regarde. Ainsi nous pouvons modéliser la faute au niveau physique, au niveau register-transfer level (RTL) ou via son effet sur la microarchitecture.

11.2.1 Modélisation physique

Nous parlons de **timing fault model** [12] lorsqu'une faute est générée dans un registre à cause de la violation des contraintes temporelles comme présenté dans la sous-section 11.1.2.

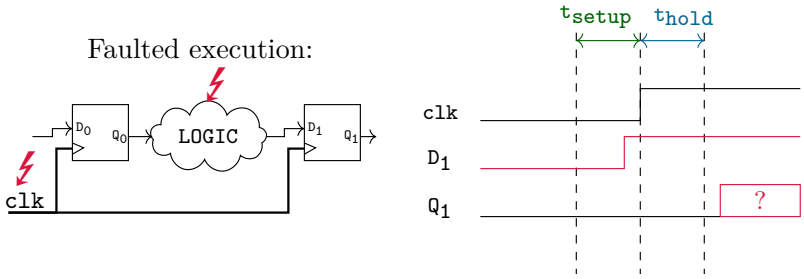


FIGURE 11.5 – Illustration du **timing fault model** : une variation de l'entrée de la bascule pendant le temps de setup même à une faute. Crédit Amélie Marotta.

Il est également possible d'influer directement sur de processus d'échantillonnage d'un registre. Il s'agit alors du **sampling fault model** [28]. C'est un modèle correspondant à priori plutôt à l'injection EM qui se couple avec les lignes d'alimentation et de masse, dans lequel la perturbation EM va mettre les signaux dans un état intermédiaire. À la fin de la perturbation, tous les signaux vont commencer à se rétablir. Si le signal d'horloge se rétabli avant l'entrée D de la flip-flop, une mauvaise valeur peut être mémorisée.

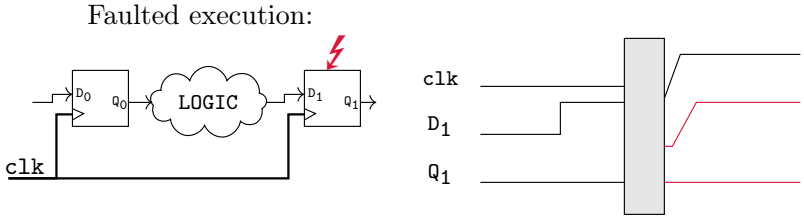


FIGURE 11.6 – Illustration du **sampling fault model**. Crédit Amélie Marotta.

Enfin, il est nécessaire d'avoir une énergie minimale sur le front montant de l'horloge pour un échantillonnage correct (signal au dessus d'un seuil pendant un certain temps). Et cela

indépendamment des autres signaux. S'il n'y a pas cette énergie minimale, l'échantillonnage ne se fait pas. On parle d'**energy-threshold fault model** [22]. Ce modèle est celui correspondant à une injection EM qui se couple avec l'arbre d'horloge, ou un clock glitch à l'aide de TRAITOR.

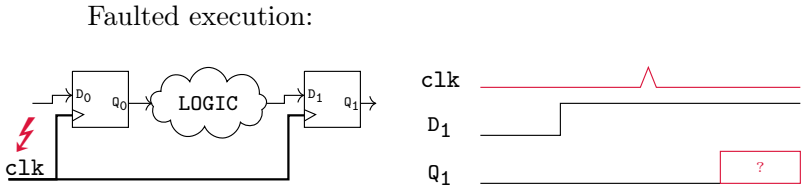


FIGURE 11.7 – Illustration du **energy-threshold fault model**. Crédit Amélie Marotta.

En réalité, une injection de faute par perturbation ou EM est susceptible de causer l'un des trois modèles, ou plusieurs en même temps, dépendant du banc et de la cible (notamment sa fréquence de fonctionnement). Il faut donc toujours caractériser son banc d'injection pour comprendre son effet sur une cible particulière.

11.2.2 Modélisation RTL

L'effet du point de vue du circuit, au niveau RTL, correspond à son effet sur les valeurs logiques des signaux.

— Un retournement de bit (*bitflip* en anglais) est l'inversion de la valeur d'un bit sur un fil. Suivant l'emplacement du fil dans le circuit, cette erreur peut être propagée jusqu'à un registre, menant à une faute c'est-à-dire un résultat erroné. Toutefois, il est également possible que l'erreur soit logiquement absorbée par un circuit combinatoire : aucune faute ne se manifeste alors.

— Un collage à 0, ou à 1 (*stuck-at 0, 1* en anglais) est la conséquence d'un courant induit ayant un effet mémoire par rétroaction : c'est un verrou (*latch-up* en anglais). Le fil ne bouge alors plus durant un certain temps, restant à la valeur 0 ou 1. Ce temps peut être réduit ou infini : il faut alors arrêter l'alimentation du circuit pour effacer la perturbation.

11.2.3 Modèles de fautes dans la microarchitecture

Nous venons de voir l'injection de faute d'un point de vue cryptanalytique, c'est-à-dire que la faute est modélisée au niveau de l'algorithme directement. Dans cette section, nous regarderons la faute du point de vue du système : quels sont les effets de l'injection sur le fonctionnement du processeur et donc du logiciel tournant dessus.

Nous devons garder à l'esprit que la plupart des travaux de caractérisation des modèles de faute sur la microarchitecture ciblent les microcontrôleurs. Ainsi les détails des effets d'une injection de faute sur une architecture complexe sont très largement inconnus.

11.2.3.1 La corruption du cache

Il est reporté dans [40] qu'il est possible de corrompre les mémoires caches sur un System-on-Chip (SoC), ici un Raspberry Pi 3. Plusieurs caches de la hiérarchie mémoire (cf chapitre 15) sont ciblés indépendamment.

Il semble que l'injection de faute électromagnétique permet de se coupler aux lignes de bus et donc de corrompre les données en transit. Ainsi une faute dans une mémoire cache sera persistance tant que celle-ci n'aura pas été invalidée. Si la mémoire cache ciblée contient des instructions, le programme en train d'être exécuté est modifié durablement.

La corruption semble aléatoire, l'attaquant ne choisit pas la nouvelle instruction remplaçant l'ancienne.

11.2.3.2 Le NOP virtuel et le saut d'instruction

Qu'elle est la conséquence d'une corruption aléatoire d'une instruction ? La réponse dépend du jeu d'instruction utilisé. La densité de l'encodage des instructions fait qu'il est hautement probable que la nouvelle instruction, obtenue après la faute, soit valide. Mais d'expérience, cette nouvelle instruction a peu de chance de créer des effets de bords par rapport au programme exécuté.

> Avec une grande probabilité, une instruction corrompue n'a pas d'effet. Elle agit comme une instruction NOP (No-Operation).

On parle alors de NOP virtuel, l'instruction agit comme un NOP [25].

Saut d'instruction Le modèle de faute logiciel le plus couramment utilisé est le « saut d'instruction ». Ce modèle est le terme générique désignant soit la création d'un NOP virtuel, soit le décalage du PC par un effet de l'injection sur le fonctionnement de la microarchitecture. Fonctionnellement, on ne différencie généralement pas l'exécution d'un NOP ou la non-exécution d'une instruction.

Les possibilités d'exploit à partir d'un simple saut d'instruction sont nombreuses [6]. Un saut sur une instruction manipulant des données peut permettre la cryptanalyse, par exemple sur RSA (cf sous-section 11.3.1) ou sur AES (cf sous-section 11.3.2).

Un saut sur une instruction gérant le flot de contrôle permet de dérouter un programme. Cela peut permettre de valider un code PIN erroné, de déclencher une porte dérobée ou d'initier un return-oriented programming (ROP).

11.2.3.3 Les arêtes fantômes

Les arêtes fantômes (*phantom edges* en anglais) désignent le fait que même si le NOP virtuel est la conséquence la plus probable d'une corruption d'instruction, il ne s'agit pas de la seule possibilité. Un autre effet observé est l'apparition d'arêtes fantômes, c'est à dire de branchement ou de saut à des destinations non contrôlées. Cet effet est par exemple obtenu par la corruption d'une petite partie d'une instruction de branchement, qui ne modifierait que le décalage (*offset* en anglais).

Le nom d'arêtes fantômes vient du fait que cet effet de l'injection ajoute une arête, normalement non présente, dans le graphe de flot de contrôle (CFG).

11.2.3.4 Le rejeu

Une variante du saut d'instruction est le modèle de rejeu [32]. Ce modèle est issu d'une injection de faute qui détourne le comportement de l'étage de `fetch` du pipeline. Ainsi une instruction est écrasée par l'instruction précédente. Le corruption de l'instruction n'est plus aléatoire, la probabilité d'obtenir un NOP virtuel est augmentée, car il est probable que l'instruction rejouée soit idempotente.

11.2.3.5 L'injection de fautes nombreuses

Injecter une seule faute se réalise assez facilement, c'est une technique bien connue maintenant. Face à des contremesures qui se focalisent sur ce scénario, de nouveaux travaux [29] montrent que réaliser des fautes nombreuses est possible dans certains cas.

Si réaliser une faute unique permet de détourner le flot de contrôle, des fautes nombreuses permettent d'éditer un programme lors de son exécution. Il suffit pour l'attaquant de supprimer les instructions qui ne l'intéressent pas, pour exécuter le programme qu'il a choisi lui-même.

Exercice 11.1 - *Fault-activated backdoor*

Vous travaillez pour EvilCorp et vous devez placer une porte dérobée, indétectable de préférence, et nécessitant une injection de faute pour être activée. Votre cible est un microcontrôleur à base de cœur ARM Cortex-M3. Comment faites-vous ? Comment vous en protéger ?

11.3 Exploiter l'injection de fautes

Les fautes sont très utilisées pour la cryptanalyse, nous parlons alors de fault injection analysis (FIA). Même si un algorithme est prouvé sûr d'un point de vue cryptographique, cela ne présage pas de sa sécurité si certaines instructions ne sont pas réellement exécutées.

11.3.1 L'attaque de Bellcore sur le RSA-CRT

RSA-CRT (pour *Chinese Remainder Theorem*, soit théorème des restes chinois ou théorème de Sun Zi) est une implémentation optimisée de RSA.

11.3.1.1 Rappel : RSA

RSA est un algorithme de cryptographie à clé publique, qui peut être utilisé pour du chiffrement ou de la signature. Nous nous intéressons ici au schéma de signature classique (pas CRT donc).

Soit m un message que Alice désire signer et que Bob désire vérifier.

Génération des paramètres Alice génère p et q deux grands entiers premiers et calcul $n = p \cdot q$, le modulo public. Elle calcule l'indicatrice de Carmichael $\lambda(n) = ppcm(p-1, q-1)$. Seule Alice peut calculer cette indicatrice, secrète, car elle seule connait p et q. L'indicatrice $\lambda(n)$ est le plus petit m tel que $a^m = 1 \mod n$ pour tout a coprime avec n. Elle choisit l'exposant e tel que $1 < e < \lambda(n)$ et $pgcd(e, \lambda(n)) = 1$. e est un paramètre public.

Finalement la clé secrète d est déterminée : $d = e^{-1} \mod \lambda(n)$. Autrement dit, $d \cdot e = 1 \mod \lambda(n)$.

Alice peut maintenant publier les paramètres publics n et e.

Signature et vérification Puisque m peut être de taille variable, Alice va d'abord calculer le condensat (*hash* en anglais), $h = hash(m)$.

La signature est

$$s = h^d \mod n,$$

que seule Alice peut calculer puisque d est la clé secrète. Bob peut vérifier que la signature vient bien d'Alice à l'aide de la clé publique de cette dernière.

Bob calcule et vérifie $s^e = \left(h^d\right)^e = h^{de} \stackrel{?}{=} h \mod n$.

11.3.1.2 RSA-CRT

Calculer l'exponentiation sur de grands entiers peut être couteux en ressources de calculs. Il est par exemple possible d'utiliser l'algorithme d'exponentiation rapide (cf algorithme 10.4). Il est possible d'obtenir des calculs plus rapides avec RSA-CRT.

Dans ce cas, la clé secrète est partagée en deux parts :

$$d_p = d \mod (p-1) \tag{11.6}$$
$$d_q = d \mod (q-1), \tag{11.7}$$

et on précalcule

$$p_{inv} = p^{-1} \mod q.$$

La signature est alors calculée en deux parts puis recombinée avec le théorème de Sun Zi.

$$s_p = h^{d_p} \mod p \tag{11.8}$$
$$s_q = h^{d_q} \mod q \tag{11.9}$$
$$s = (((s_q - s_p) \cdot p_{inv}) \mod q) \cdot p + s_p. \tag{11.10}$$

Nous pouvons vérifier que

$$s = s_p \mod p \text{ (la vérification est triviale).} \tag{11.11}$$
$$s = ((s_q - s_p) \cdot p_{inv} \cdot p) \mod q + s_p \mod q, \tag{11.12}$$
$$s = s_q \mod q - s_p \mod q + s_p \mod q, \tag{11.13}$$
$$s = s_q \mod q. \tag{11.14}$$

11.3.1.3 L'attaque de Bellcore [4]

Nous supposons maintenant un adversaire capable d'injecter une faute aléatoire lors du calcul de s_p, (cela marche aussi avec s_q). Nous avons donc, pour une valeur d'erreur err

$$s'_p = h^{d_p} + err \mod p \tag{11.15}$$
$$s' = \left(\left(\left(s_q - s'_p\right) \cdot p_{inv}\right) \mod q\right) \cdot p + s'_p \tag{11.16}$$
$$s' \mod q = s_q \mod q - s'_p \mod q + s'_p \mod q. \tag{11.17}$$

Dans ce cas, $s' \neq s \mod p$, mais $s' = s \mod q$. Autrement dit, la différence $s' - s$ est un multiple de q mais pas de p. Nous pouvons alors calculer

$$pgcd(s' - s, n) = q.$$

Cette attaque illustre bien le pouvoir cryptanalytique de l'injection de faute. Une simple faute, totalement aléatoire, permet de retrouver l'intégrité de la clé secrète.

11.3.2 L'attaque de Piret-Quisquater sur l'AES [30] : Differential Fault Analysis

Il n'y a pas toujours, comme dans l'attaque de Bellcore, la possibilité de retrouver toute la clé avec une seule faute. De manière générale, des attaques par fautes ont été conçues contre toutes sortes d'algorithmes. Un motif que l'on retrouve souvent est la differential fault analysis (DFA). La différence entre le chiffré fauté et celui correct permet de gagner de l'information sur le secret, à partir d'un modèle de faute supposé.

L'attaque de Piret-Quisquater suppose une faute aléatoire sur un octet du State de l'AES en entrée du dernier `MixColumns` (cf figure 10.6). Nous pouvons alors suivre sa propagation, illustrée sur la figure 11.8.

— Après `MixColumns`, 4 octets sont fautés avec un motif spécifique. En effet, `MixColumns` est une multiplication matricielle sur une colonne.

$$\begin{pmatrix} o_0 \\ o_1 \\ o_2 \\ o_3 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} i_0 \\ i_1 \\ i_2 \\ i_3 \end{pmatrix}$$

Comme cette opération est linéaire, l'erreur en sortie de `MixColumns` peut être calculée en applicant la même multiplication matricielle à l'erreur en entrée. Puisqu'il n'existe que $1020 = 4 \cdot 255$ erreurs possibles en entrée, il y en a autant possible en sortie. Nous précalculons donc Δ, l'ensemble des vecteurs de 4 octets de fautes possibles en sortie de `MixColumns`.

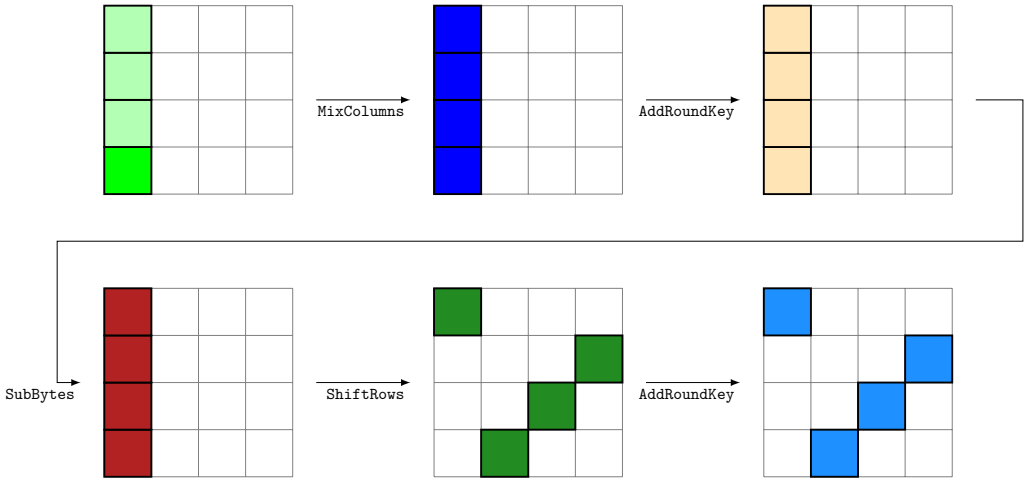


FIGURE 11.8 – Propagation de la faute dans le `State` au cours de l'attaque Piret-Quisquater [30].

— Toujours les mêmes 4 octets fautés après l'application de `AddRoundKey` et dde `SubBytes`.
— Après `ShiftRows`, les 4 octets fautés sont décalés.
— Ce décalage est conservé à travers le `AddRoundKey` final.

Nous travaillons sur une seule colonne du `State` à la fois. En tant qu'attaquant, nous sauvegardons le texte chiffré correct et le texte chiffré fauté pour un même texte clair et une même clé, celle que nous recherchons.

L'attaque se déroule comme suit :

1. Nous appliquons `InvShiftRows`, la réciproque de `ShiftRows` sur les textes chiffrés (correct et fauté) pour ne plus avoir à gérer le décalage.
2. Nous faisons une supposition sur une colonne de clé. C'est-à-dire que nous choisissons un vecteur de 4 octets, noté K_{10}. Cette attaque nécessite donc de réaliser 2^{32} tests d'hypothèse.
3. Grâce à l'hypothèse sur la clé, nous calculons successivement les états intermédiaires en entrée du dernier `AddRoundKey` et en entrée de `SubBytes`, dans les deux cas : à partir du chiffré correct et du chiffré fauté.
4. Nous calculons δ la différentielle des deux colonnes (correct et fautée) en entrée de `SubBytes`.
5. Si $\delta \in \Delta$, alors l'hypothèse de colonne de clé K_{10} est conforme à nos observations.
6. En général, pour un couple (chiffré correct, chiffré fauté), plusieurs hypothèses de clé sont valides. Il faut alors renouveler l'opération avec un deuxième couple de textes chiffrés. L'intersection des ensembles d'hypothèses valides donne l'unique hypothèse correspondant à la colonne de clé véritable.

Cette attaque est parallélisable, en injectant une faute sur un octet par colonne en entrée du dernier `MixColumns` (au tour 9). Ou, de manière équivalente, en injectant une faute sur un seul octet en entrée du `MixColumns` du tour 8.

L'attaque nécessite de réaliser 2 injections de fautes et $2^{34} = 2^{32} \cdot 4$ hypothèses pour retrouver la clé complètement.

11.3.3 L'attaque NUEVA (Non-Uniform Error Value Analysis) [19]

L'attaque de Piret-Quisquater présuppose un modèle de faute précis. Comment faire si nous ne connaissons pas le modèle de faute, ou qu'il ne correspond pas ? Il faut alors choisir une autre attaque comme NUEVA [19].

L'intuition est ici un peu différente. Le but de l'attaquant est de calculer la valeur de

la faute injectée, paramétrisée par une hypothèse sur la clé. S'il peut reconnaitre les bonnes valeurs de fautes, il peut alors retrouver la clé.

Plus précisément, on suppose que l'attaquant peut injecter une faute avant le dernier `SubBytes` et que la distribution des valeurs de fautes est non uniforme. L'analyse se fait octet par octet, nous ne discuterons ici que du premier octet de la première colonne du `State`, mais l'attaque est trivialement généralisable.

L'attaque se déroule comme suit :

On répète n injections de fautes, et à chaque fois on sauvegarde c le chiffré correct et c' le chiffré fauté (1 seul octet chaque). On peut alors calculer e la valeur de la faute injectée (1 octet), en utilisant une hypothèse k sur la clé (1 octet), selon l'équation 11.18.

$$e = SB^{-1}(c \oplus k) \oplus SB^{-1}(c' \oplus k) \tag{11.18}$$

Parmi les colonnes de la table 11.1, une seule est correcte. Le but de l'attaquant est de trouver laquelle, ce qui lui permet de déduire la valeur de l'octet de clé.

TABLE 11.1 – Table des valeurs d'erreurs

# injection \ hypothèse clé	0	1	⋯	255
1	$e_{1,0}$	$e_{1,1}$	⋯	$e_{1,255}$
2	$e_{2,0}$	$e_{2,1}$	⋯	$e_{2,255}$
⋮	⋮	⋮	⋱	⋮
n	$e_{n,0}$	$e_{n,1}$	⋯	$e_{n,255}$

Pour cela on se base sur une propriété de l'équation 11.18 : si l'hypothèse de clé est incorrecte, alors la distribution des valeurs d'erreurs est « plus » uniforme. Par exemple, si l'attaquant est capable d'injecter toujours la même valeur d'erreur, seule la bonne colonne est constante. Les autres colonnes ne seront pas constantes du fait de l'équation 11.18.

L'attaquant peut donc mesurer le caractère uniforme de la distribution des valeurs d'erreurs, par exemple avec l'entropie de Shannon, et ainsi déterminer quelle est la bonne hypothèse de clé.

11.3.4 Le cas du code PIN

La vérification de code PIN est un cas d'usage typique vulnérable à une attaque par injection de faute. Le scénario est simple et n'implique pas (nécessairement) de cryptographie :

Un terminal envoie un code, entré par l'utilisateur à un microcontrôleur, par exemple sur une carte à puce. La carte doit vérifier que ce code est correct, identique à celui sauvegardé en interne, et ne laisse qu'un nombre limité d'essais en cas de code erroné.

Lorsque l'on compile le code de la figure 11.9, nous avons probablement (sauf inlining), un appel à la fonction `compare_arrays` dans la fonction `verify_pin`. Voici un extrait de listing résultant du code de la figure 11.9.

```
8000120: 2204       movs    r2, #4
8000122: 490e       ldr     r1, [pc, #56]
8000124: 4628       mov     r0, r5
8000126: f7ff ffd5  bl      80000d4 <compare_arrays>
```

Si l'on saute cet appel à l'adresse $0x08000126$, nous ne mettons pas à jour le registre `r0` décidant si le code PIN est valide ou non. Suivant les détails de l'application, cela peut suffire à outrepasser la vérification du code.

```c
#define PIN_SIZE 4
uint8_t secret_pin[PIN_SIZE] = { /* ... */ };

bool compare_arrays(uint8_t* arr1, uint8_t* arr2) {
    uint8_t acc = 0;
    for (int i = 0; i < PIN_SIZE; i++) {
        acc |= arr1[i] ^ arr2[i];
    }
    return (acc == 0);
}

bool verify_pin(uint8_t* candidate) {
    if (compare_arrays(candiate, secret_pin) == true) {
        return VALID;
    }
    else {
        return INVALID;
    }
}
```

FIGURE 11.9 – Exemple de code simplifié pour une vérification de code PIN. Il manque notamment la vérification du nombre d'essais.

11.4 Les contremesures à l'injection de faute

En présence de fautes, le durcissement de l'application peut se faire de nombreuses manières. Toutefois la technique principale est la duplication, déclinée en plusieurs variantes.

11.4.1 Les contremesures logicielles

11.4.1.1 Protéger les données avec des codes détecteurs d'erreurs

Pour un attaquant uniquement capable de corrompre des données, il est possible de protéger l'application par l'ajout de codes détecteurs d'erreurs. Le schéma dépend de l'application spécifiquement.

Pour l'AES par exemple, une technique simple contre un attaquant réalisant une faute aléatoire sur un octet (modèle de faute pour l'attaque Piret-Quisquater, sous-section 11.3.2), il est possible de rajouter des sommes de contrôle sur le `State`. Pour chaque colonne et chaque ligne du `State`, calculer la somme (opérateur xor) de la colonne ou de la ligne. Au cours du calcul, il faut calculer indépendamment, quand c'est possible, la mise à jour du `State` et la mise à jour des sommes de contrôle. Puis, la validité des sommes mises à jour est vérifiée par rapport à la nouvelle valeur du `State`. En cas d'erreur, l'injection de faute est détectée. La puce peut réagir, par exemple en supprimant la clé secrète s'il n'y a pas de contraintes de disponibilité.

11.4.1.2 La duplication d'instructions

Au niveau logiciel, il est possible de créer un programme tel que la suppression de n'importe quelle instruction ne change pas son comportement [26]. Cette tâche peut être réalisée manuellement, ou à l'aide d'un compilateur spécialisé.

Le principe est simple, si l'instruction est idempotente, il suffit de la dupliquer immédiatement. Les instructions *séparables* sont les instructions non idempotentes qui peuvent être remplacées par une ou des instructions idempotentes.

```
    add r1, r1, r3
    # peut être remplacé par
    # (rx est un registre disponible)
    mov rx, r1
    mov rx, r1
    add r1, rx, r3
    add r1, rx, r3
```

Enfin certaines instructions spécifiques ne sont ni idempotentes ni séparables, il faut trouver une séquence de remplacement pour chacune. Par exemple l'instruction `bl <function>` qui branche vers la fonction spécifiée et sauvegarde l'adresse de retour dans le registre `lr` (*link register*) dédié peut être remplacé par la séquence suivante.

```
    bl <function>
    # peut être remplacé par
    # (ry est un registre disponible)
    adr ry, <return_label>
    adr ry, <return_label>
    b <function>
    b <function>
return_label:
```

Cette séquence, pour peu que la sous-fonction soit elle-même durcie, s'assure que cette dernière est toujours exécutée une seule fois, même en présence de fautes.

Cette protection est relativement couteuse : le programme est bien ralenti et sa taille est multipliée par ≈ 2. Surtout, le programme n'est protégé que contre des injections simples. Les fautes multiples rendent cette protection caduque.

11.4.2 Les contremesures matérielles

Puisque les fautes multiples rendent quasiment impossible une protection purement logicielle, il est temps de se tourner vers les protections se focalisant sur le matériel. Il est possible d'ajouter des durcissements à plusieurs niveaux. Au niveau physique, au niveau combinatoire, au niveau de l'architecture du processeur, etc.

Selon le principe de la défense en profondeur, une seule contremesure n'est pas considérée comme suffisante. On combinera donc les techniques présentées ici, en s'adaptant au modèle de menaces considéré.

11.4.2.1 Les boucliers

Le premier niveau de protection est d'empêcher l'attaquant d'interagir avec le circuit. Il est possible d'ajouter un bouclier [9], maillage de fils métalliques en surface du circuit. On parle de bouclier (*shield ou mesh* en anglais).

Le maillage est positionné pour empêcher si possible, mais surtout détecter des perturbations. Évidemment, le maillage est lui-même la cible d'attaques et il faut se prémunir de toute modification de celui-ci.

— Pour empêcher que les pistes ne soient simplement enlevées, des signaux doivent parcourir ces fils. Lorsque ces signaux ne sont plus reçus, ou corrompus, l'attaque est détectée.
— Pour prévenir une attaque par man in the middle, les signaux doivent être constitués par un protocole cryptographique choisi pour l'attaquant ne puisse l'altérer sans être détecté.
— Pour prévenir un reroutage des signaux (possible avec un FIB par exemple), le temps de vol, c'est-à-dire le temps que mettent les signaux entre l'émetteur et le récepteur, doit être mesuré et validé.

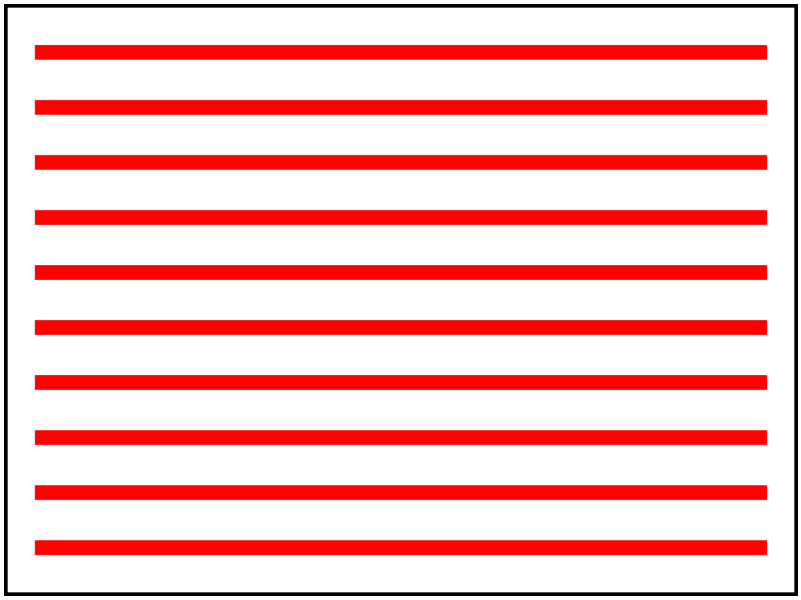


FIGURE 11.10 – Illustration d'un bouclier constitué de lignes métalliques en surface de la puce. Les lignes ont une direction, car des composants matériels, sur la puce en dessous, génèrent des données dont l'intégrité est mesurée en sortie.

Les boucliers doivent être systématiquement utilisés pour un système embarqué devant résister à une injection de faute. C'est le cas, par exemple, de la puce de votre carte bancaire.

Malgré leur intégration simple au processus de fabrication, il suffit de rajouter des pistes métalliques, les boucliers ne peuvent dans ce cas que protéger la surface de la puce. Il reste possible pour l'attaquant d'injecter des fautes par le dos de la puce.

Exemple 11.6

Vous pouvez voir le décorticage d'un microcontrôleur protégé avec un bouclier sur la vidéo Que vaut l'électronique d'un portefeuille de cryptomonnaie LEDGER NANO S ? de la chaine Deus Ex Silicium sur YouTube.

11.4.2.2 Les capteurs

De nombreuses techniques d'injection (perturbations d'alimentation, d'horloge, injection électromagnétique) partagent des modèles électriques similaires. En ciblant ces modèles, il est possible de mesurer les effets de l'injection [42].

Il existe plusieurs moyens de faire ces mesures. Une technique consiste à mesurer le temps de propagation d'un signal électrique à l'aide de lignes à retard (*delay lines* en anglais) et permet de détecter une faute suivant le *timing fault model*, voir la figure 11.11.

Un front montant arrivant trop tôt, ou un retard de la propagation du signal, est détecté par des valeurs erronées présentes dans les registres du détecteur.

Ces injections, y compris les perturbations d'horloge et d'alimentation, n'ont pas un effet homogène sur la totalité de la surface du circuit. En conséquence, un seul capteur pourrait ne pas détecter une injection, car son effet ne serait pas détectable au niveau du capteur. Toutefois, comme il n'y a besoin que d'un faible nombre de transistors par capteur, il est possible de dupliquer les capteurs et de les répartir sur la surface de la puce à protéger. Une puce durcie inclut donc des capteurs d'injection pour détecter une attaque.

Ces capteurs ne sont pas suffisants seuls. En particulier, l'injection laser induit des effets différents et extrêmement limités dans l'espace.

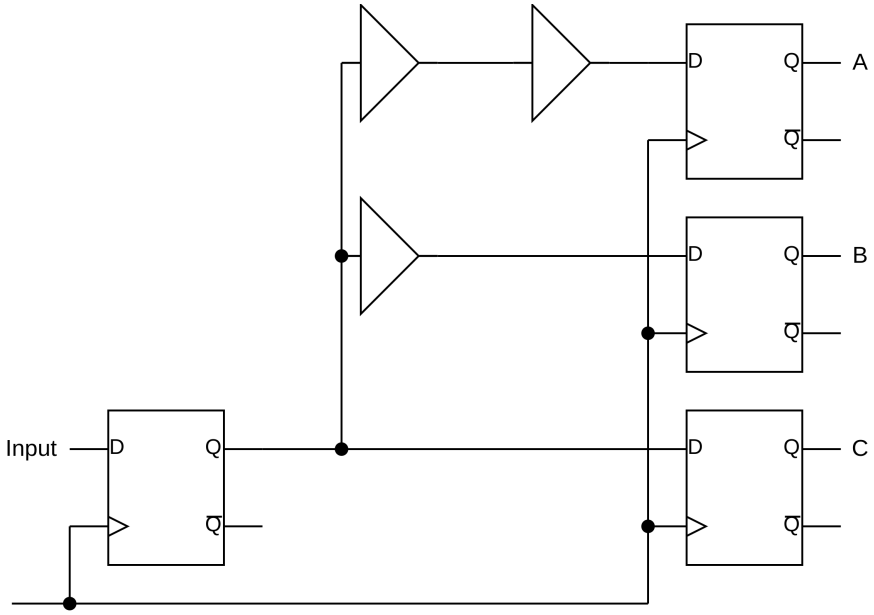


FIGURE 11.11 – Les lignes à retard permettent de mesurer le temps de propagation d'un signal à travers des portes logiques, des portes identité, par rapport à deux fronts d'horloge consécutifs. Si par exemple, la période d'horloge nominale permet de ne traverser qu'une seule porte logique, lorsque `input` passe à 1, on s'attend à lire $\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$. Toute autre valeur dénote une injection en cours.

11.4.2.3 Le durcissement des circuits combinatoires

Absorption d'erreur À un autre niveau, il est possible de durcir un circuit combinatoire pour qu'il résiste à un certain nombre de fautes. Il faut pour cela transformer le circuit pour que tout retournement de bit sur un fil sensible à l'injection soit détecté ou corrigé. On parle d'absorption de l'erreur lorsque le circuit est capable de la corriger.

Lorsque l'on se contente de la détection, un signal d'alarme est ajouté en sortie comme sur la figure 11.12. Bien sûr, ce signal pourrait à son tour servir de cible pour l'attaquant dans le but de supprimer l'alarme.

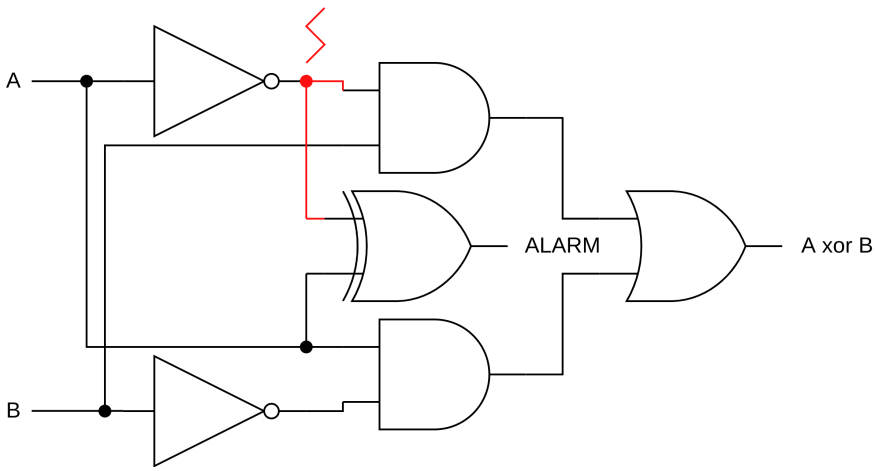


FIGURE 11.12 – Le fil rouge en sortie d'inverseur est protégé par une alarme.

Dual rail Le dual rail (ou triple rail, etc.) est la systémisation de la redondance dans le circuit. Tout signal logique est porté par deux fils avec un codage spécifique. Par exemple $00 \rightarrow 0$, $11 \rightarrow 1$, $01,10 \rightarrow$ alarm. Dans le circuit combinatoire, il ne doit pas y avoir de fil « simple », comme illustré sur la figure 11.13.

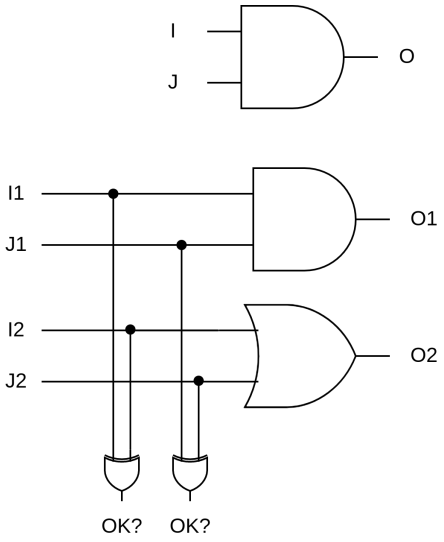


FIGURE 11.13 – Exemple de circuit dual rail remplaçant une porte AND avec de la détection d'erreur (et pas de propagation d'erreur).

La création de ce type de circuits est facilement automatisable, et des bibliothèques de cellules dédiées peuvent être créées. Toutefois, le surcout de ce durcissement est très élevé : doublement des fils, doublement au moins du nombre de transistors. Ce qui implique un circuit plus lent, prenant plus de silicium et consommant plus d'énergie.

Détection d'erreur au niveau des registres Selon le circuit, il est possible de réaliser la détection d'erreur en se focalisant sur les registres. Par exemple, pour un circuit réalisant une fonction $y = f(x)$ appliquée au vecteur d'entrée x, nous pouvons ajouter un registre sauvegardant la parité $x^p \bigoplus_i x_i$. Puis nous ajoutons le circuit combinatoire $y^p = p(x^p)$, qui calcule l'évolution de la parité. Au niveau des registres sauvegardant y, on vérifie que $y^p \stackrel{?}{=} \bigoplus_i y_i$.

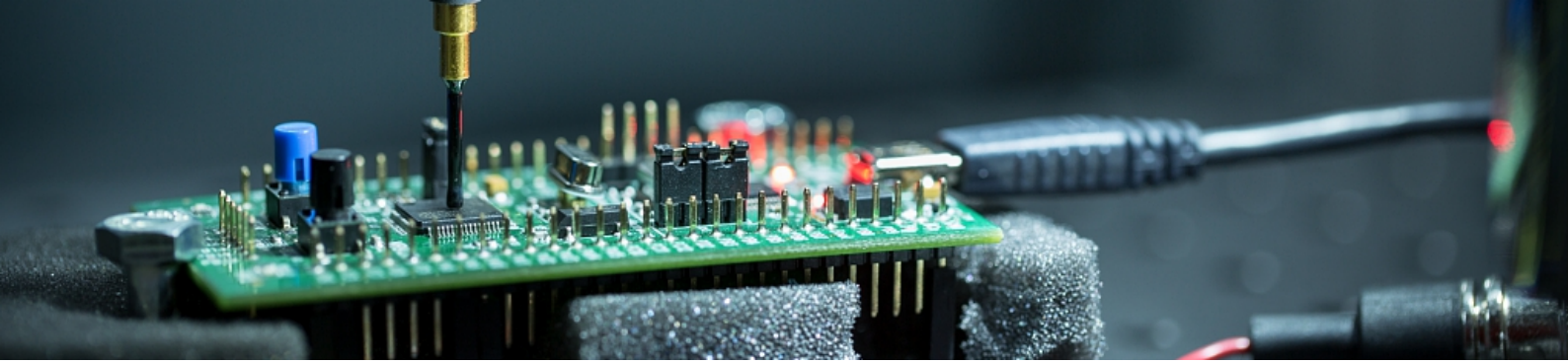
Un calcul permettant la mise à jour de la parité indépendamment des données elles-mêmes n'est pas toujours possible. Ce genre de schéma doit s'adapter au circuit combinatoire que l'on cherche à protéger.

11.4.2.4 Duplication, triplication, et processeurs lockstep

Une autre technique pour protéger un processeur est de dupliquer celui-ci. On place alors deux cœurs ensemble, qui doivent exécuter la même chose de manière synchronisée. On parle de processeurs en lockstep. Toute déviation entre le comportement des deux processeurs est le signe d'une injection de faute.

Une variante de ce système est la triplication. Avec les processeurs, un système de vote est chargé de sélectionner la sortie majoritaire parmi les 3 cœurs. Ainsi il est possible de corriger une erreur en plus de la détecter.

La triplication des processeurs est la norme pour les systèmes critiques, dans l'aérospatial notamment.



12. Certification

La **certification** est une évaluation de sécurité pour évaluer la robustesse d'un produit.

La certification dite tierce partie est la certification de plus haut niveau, qui permet à un client de s'assurer par l'intervention d'un professionnel indépendant, compétent et contrôlé, appelé organisme certificateur, de la conformité d'un produit à un cahier des charges ou à une spécification technique. [1]

Cette évaluation est faite par un Centre d'évaluation de la sécurité des technologies de l'information (CESTI), sous l'autorité de l'Agence nationale de la sécurité des systèmes d'information (ANSSI). La certification est utile :

— à l'industriel fabriquant le produit, de garantir un niveau de sécurité à son produit afin de pouvoir accéder à des marchés spécifiques ;
— à l'utilisateur final du produit, de choisir et d'utiliser un produit garanti à un niveau de sécurité certifié par une agence gouvernementale.

La certification implique alors 3 partis :

1. le fabricant du produit,
2. un CESTI,
3. l'ANSSI.

En France, l'Agence nationale de la sécurité des systèmes d'information (ANSSI) est une agence étatique qui a un rôle de défense et uniquement de défense des systèmes d'information. Elle n'est par conséquent ni rattachée au ministère de la Défense ni au ministère de l'Intérieur. Elle dépend directement du Premier ministre. La mission de l'ANSSI est d'assister au développement et à la sécurité des développements de services numériques en France afin d'élever le niveau de sécurité globale. Elle n' a pas un rôle offensif. Cette séparation des rôles attaque / défense est primordiale. Cela permet de garantir une confiance que l'on peut avoir en l'ANSSI.

Les rôles de l'ANSSI sont multiples mais peuvent être regroupés en trois axes principaux :

1. la **prévention**, au travers de la formation par exemple,
2. la **sensibilisation**, faire prendre conscience des risques de sécurité (exemple rédaction de guide [48])
3. la **Défense**, lorsqu'il y a une cyberattaque.

Dans le cadre de ce cours le rôle de l'ANSSI qui nous intéresse est ce lui d'émetteur de certificat de sécurité de produits.

1. Définition officielle de l'ANSSI.https://cyber.gouv.fr/

Un Centre d'évaluation de la sécurité des technologies de l'information (CESTI) est un acteur du schéma de la certification. Il est agréé par l'ANSSI. Son rôle est l'évaluation de produits d'un point de vue sécuritaire. Il existe des CESTI matériel et logiciel. Dans le contexte de ce cours, nous parlons de CESTI matériel, qui évalue des composants électroniques.

Voici son mode de fonctionnement.

1. Le fabricant/développeur demande un contrat pour l'évaluation du couple produit et cas d'usage spécifique. Un exemple peut être une carte à puce à l'usage de paiement.
2. Le CESTI propose donc un contrat pour un niveau de certification proposé par l'ANSSI.
3. Le fabricant envoie le contrat à l'ANSSI.
4. l'ANSSI analyse le contrat et le valide si l'évaluation est conforme au niveau de sécurité exigée. Les normes et les standards d'évaluation et de certification pour chaque cas d'usage sont spécifiés par l'ANSSI et publiés sur leur site. Le fabricant peut alors avoir confiance dans le contrat du CESTI puisqu'il a été validé par l'ANSSI.
5. Le fabricant et le CESTI se mettent d'accord sur le devis de l'évaluation.
6. Le CESTI réalise son évaluation. Cela consiste à tester le produit face à une liste d'attaques définies par l'ANSSI, auxquelles le produit doit résister pour obtenir le niveau de certification demandé. Une attaque est toujours bornée dans le temps, l'évaluateur ne peut dépasser un temps déterminé sur chaque attaque. Si l'attaque n'a pas réussi dans le temps imparti, le produit est considéré comme résistant. Tous les résultats sont inscrits dans un rapport. En pratique, l'ANSSI va suivre l'évaluation effectuée par le CESTI étape par étape.
7. L'évaluation est terminée, le CESTI envoie son rapport final à l'ANSSI. Celui-ci contient un avis favorable ou non sur la certification du produit.
8. l'ANSSI vérifie la conformité de l'évaluation. Si tout va bien, l'ANSSI édite le certificat du niveau évalué au fabricant et l'envoie à ce dernier. Le certificat émis par l'ANSSI atteste que les produits certifiés sont conformes à un cas d'usage. Le certificat est daté.

Pour prolonger le certificat d'un produit, le fabricant devra demander une mise à jour au CESTI qui réalisera une évaluation partielle contenant les nouveautés. Évidemment seule l'ANSSI peut valider la prolongation du certificat.

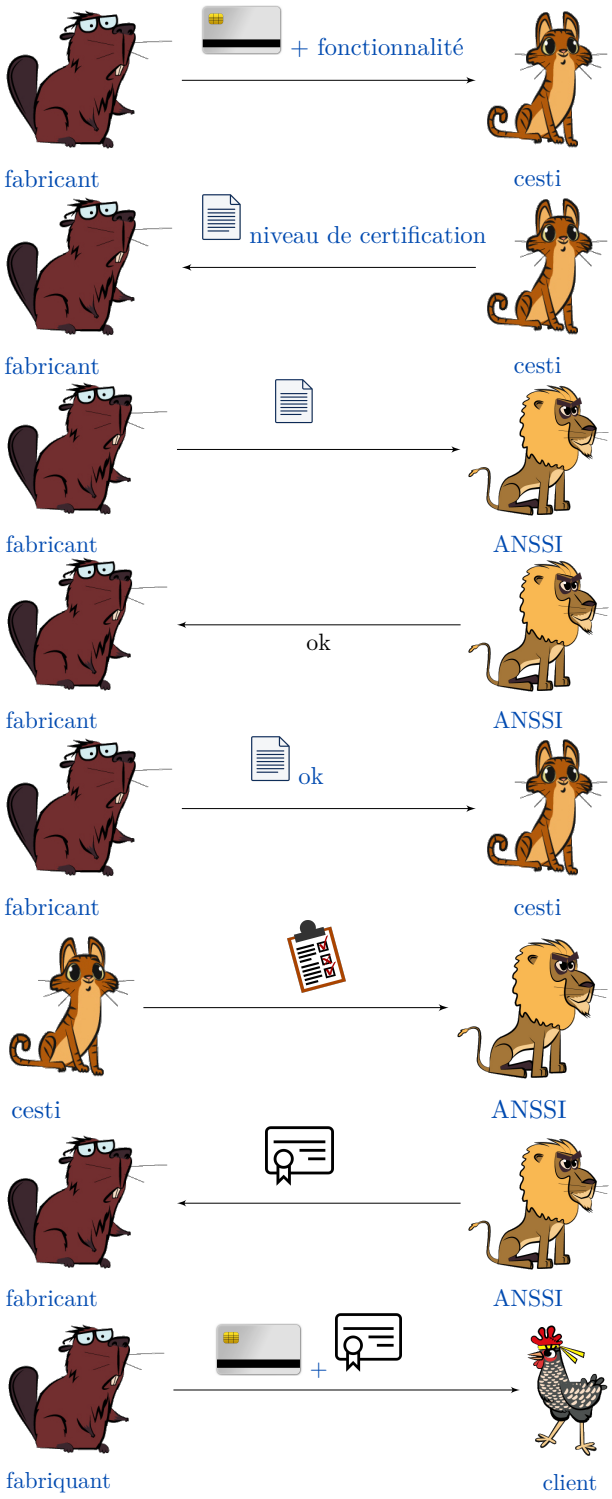
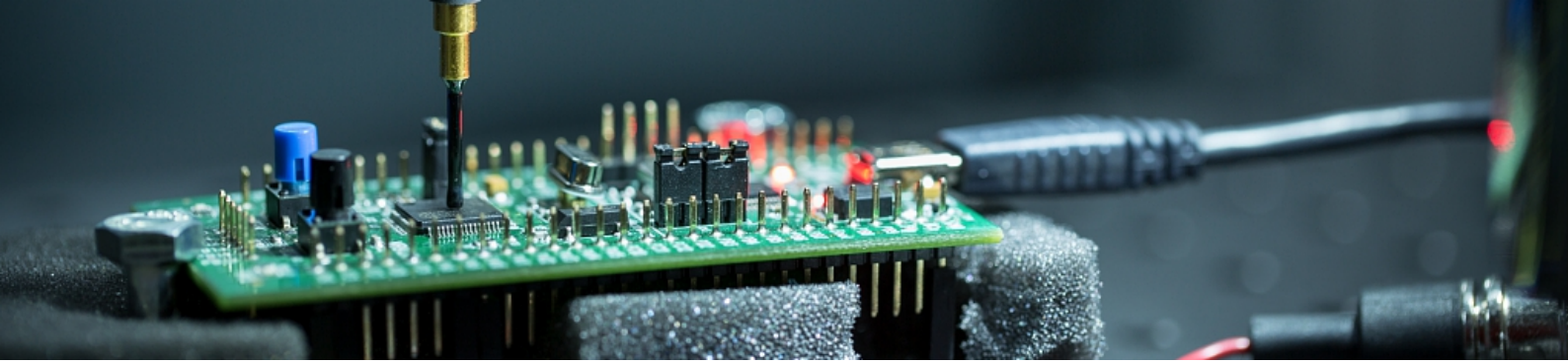


FIGURE 12.1 – Schéma de la certification *illustration : studio MoonCat.*

IV Les mémoires

«Je sens que ma mémoire me lâche, Dave.» HAL 9000 dans le film 2001 l'Odyssée de l'espace.



13. Les technologies de mémoire

Il n'y a pas qu'une seule manière de concevoir des mémoires. Dans un ordinateur moderne, de nombreuses technologies cohabitent pour répondre aux contraintes de vitesse, densité, cout, consommation énergétique, etc. Dans ce chapitre nous allons explorer les technologies les plus courantes. En comprendre le fonctionnement permet également d'en comprendre les limites et les faiblesses. Certaines de ces faiblesses, liées à la technologie, affectent la sécurité du système comme nous le verrons dans le chapitre 27.

13.1 SRAM

La mémoire static random-access memory (SRAM) est conçue à partir de transistors, de préférence durant le même processus de fabrication que le reste de la puce.

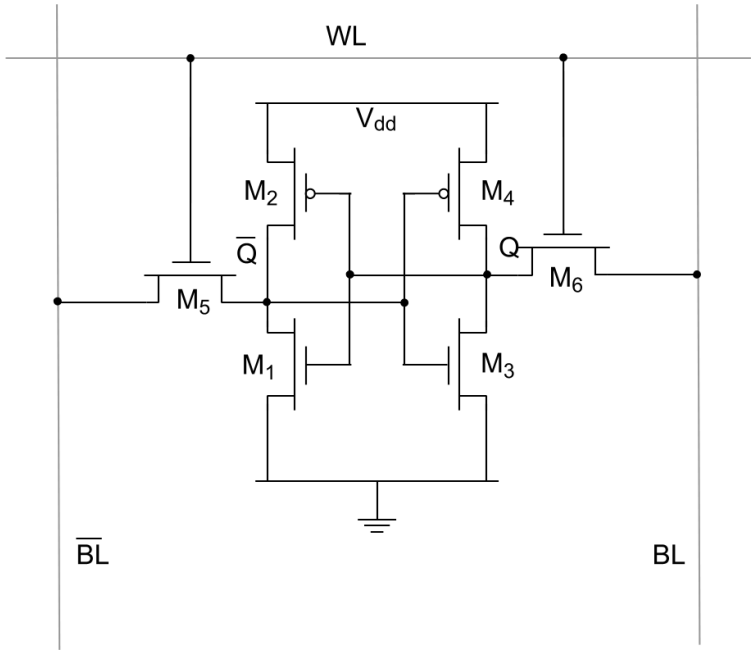


FIGURE 13.1 – Une cellule SRAM à 6 transistors peut sauvegarder un bit de données. WL est la Write Line, BL la Bit Line.

Une cellule mémoire SRAM à 6 transistors est constituée de deux inverseurs bouclés l'un

sur l'autre (cf. figure 13.1). Cette cellule mémoire a trois états.

— Au **repos**, la ligne d'écriture `WL` (write line) n'est pas active. La boucle de rétroaction constituée par les deux inverseurs permet de garder un état stable ($Q = 0$ ou $Q = 1$).
— En **lecture**, on commence par précharger la BL (bit line) **et** $\overline{BL}$ à la valeur logique 1. Puis on active la ligne WL. Puisque soit Q, soit $\overline{Q}$ est à 0, une des deux lignes BL ou $\overline{BL}$ va descendre plus rapidement vers 0. On place donc un capteur analogique qui mesure la différence de tension entre BL et $\overline{BL}$. Suivant si la tension est positive ou négative, nous avons lu la valeur de notre cellule mémoire.
— En **écriture**, on active la ligne WL et on choisit les valeurs de BL et $\overline{BL}$ selon la valeur que l'on veut écrire. Il y a donc une compétition entre la stabilité de la cellule mémoire, et l'instabilité due à l'écriture. Il suffit de dimensionner correctement les transistors $M5$ et $M6$ par rapport à $M1 - M4$ pour que l'écriture soit possible.

Les mémoires SRAM sont rapides, mais peu denses. Elles sont généralement utilisées pour les mémoires les plus proches du pipeline : fichier de registres et mémoires caches (cf. chapitre 15).

Il existe d'autres types de cellules SRAM avec plus ou moins de transistors suivant les compromis que l'on recherche.

13.2 DRAM

La mémoire dynamic random-accesss memory (DRAM) utilise une technologie dont la cellule mémoire ne contient qu'un seul transistor, offrant ainsi une forte densité. Ce transistor commande la charge ou la décharge d'un condensateur. La tension aux bornes du condensateur définit l'état de la mémoire. Évidemment, un condensateur se décharge naturellement avec le temps. Ainsi, il est nécessaire de rafraichir régulièrement chaque cellule mémoire pour garder la valeur en mémoire. En cas d'arrêt de l'alimentation, les données en mémoire sont perdues.

Plusieurs opérations peuvent être réalisées :

— La **lecture** est encore basée sur une mesure analogique. On précharge les Bit Lines avec une même valeur intermédiaire, puis on active la Word Line pour la ligne que l'on désire lire. Les transistors reliés deviennent passants, on parle d'« ouverture de ligne ». Ainsi si la capacité de la cellule est chargée (état 1), des charges sont transférées dans la Bit Line. Au contraire si le condensateur est déchargé (état 0), les charges vont de la Bit Line vers le condensateur. La tension des Bit Lines est ainsi très légèrement modifiée. Elle est ensuite analogiquement comparée avec la Bit Line adjacente, qui n'a pas été modifiée, à l'aide d'une bascule à rétroaction. La rétroaction permet d'amplifier la tension sur la Bit Line et donc de « rafraichir » le condensateur dont la charge a été légèrement altérée.
— L'**écriture** se fait en ouvrant une ligne, puis en forçant l'état de la bascule amplificatrice. Après un certain temps, le condensateur est chargé correctement.
— Une des propriétés fondamentales de toute capacité est qu'elle se décharge avec le temps. Ainsi en l'absence de lecture sur une ligne particulière durant un certain temps, tous les condensateurs se retrouvent déchargés. Le **rafraichissement** est l'opération permettant de contrer ce phénomène. À intervalle de temps régulier, on force une lecture sur l'ensemble des cellules mémoires.

La durée de sauvegarde des charges dans le condensateur dépend de nombreux paramètres, ce qui fixe ensuite l'intervalle entre deux rafraichissements. La valeur typique de référence est un rafraichissement toutes les 64 ms. Les trois paramètres les plus importants influençant la décharge sont : l'intensité du courant de fuite, dépendant surtout du processus de fabrication des transistors, la capacité du condensateur et de la température. Ce dernier paramètre en particulier est utilisé pour les attaques par démarrage à froid, « cold boot attacks », où l'attaquant utilise de l'azote ou de l'hélium liquide pour refroidir suffisamment la RAM de sorte que la décharge des condensateurs soit quasiment stoppée. Ainsi l'attaquant peut extraire physique-

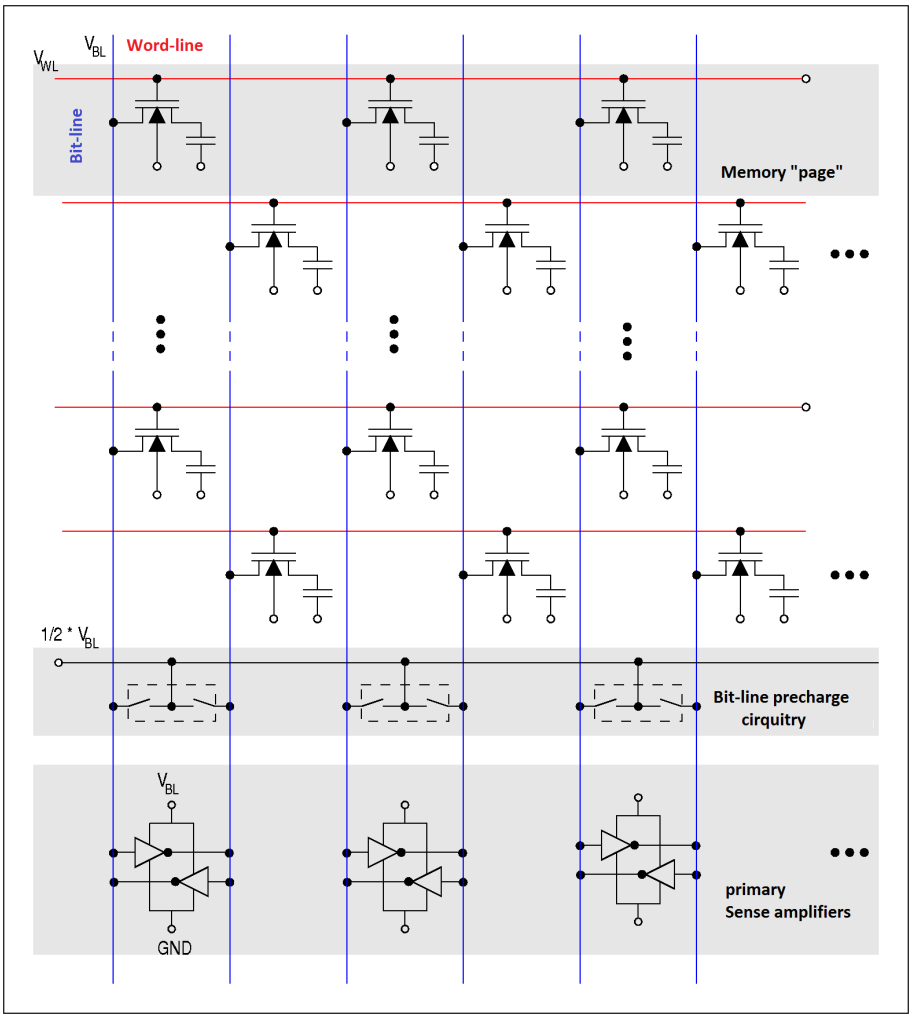


FIGURE 13.2 – Une mémoire DRAM composé de nombreuses cellules.

ment la mémoire RAM contenant encore toutes ses données, pour la lire à l'aide d'équipements dédiés.

13.3 Mémoire flash

La mémoire flash est une mémoire non volatile : à l'extinction du courant, les données restent en mémoire. C'est la mémoire que l'on retrouve dans les solid-state disks (SSDs). Il existe différents types de mémoire flash (NAND et NOR) et de nombreuses variantes, se basant sur une même technologie de transistor à grille flottante, cf figure 13.3.

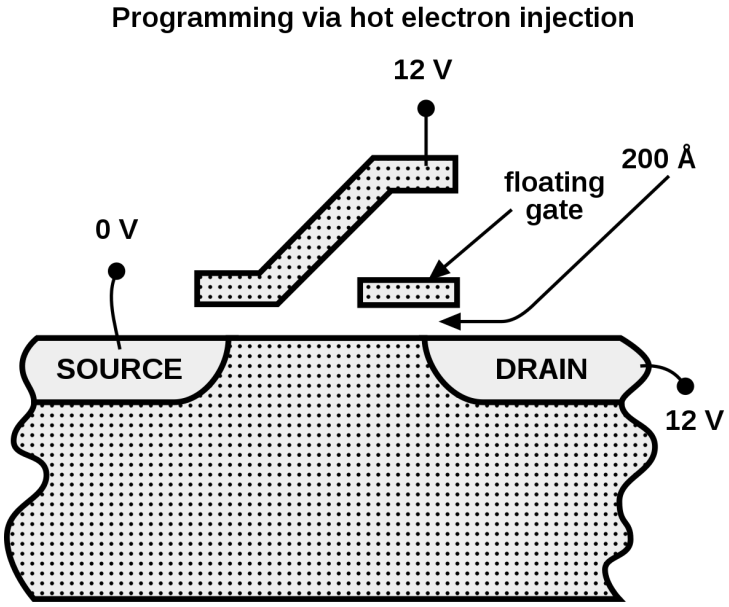
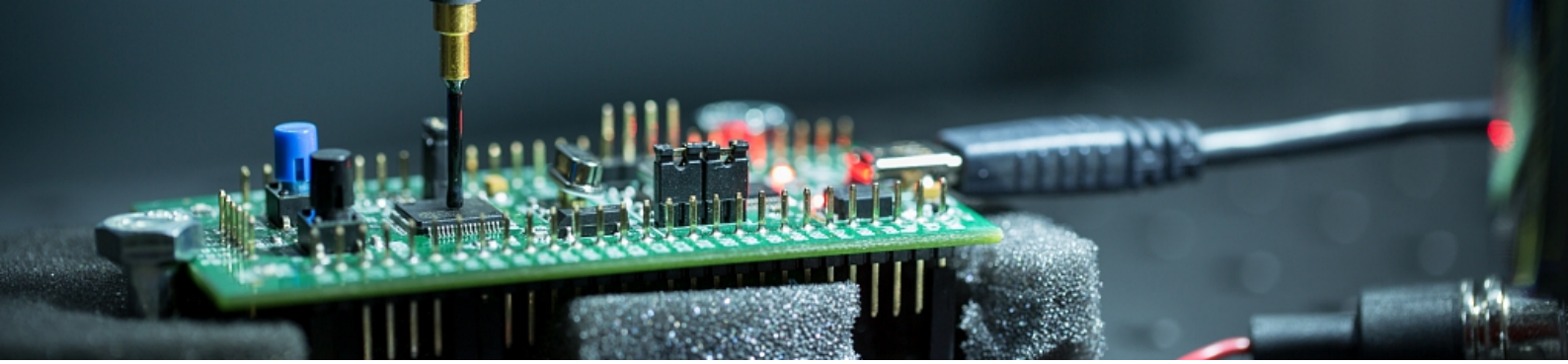


FIGURE 13.3 – Une cellule de mémoire flash.

Un transistor à grille flottante possède deux grilles : la grille flottante immédiatement au-dessus du canal entre le drain et la source et la grille de contrôle au-dessus de la grille flottante. Les opérations sont les suivantes :

— Pour la **lecture**, pour une tension de référence appliquée entre la source et le drain, on mesure le courant qui circule entre ces deux points. C'est la charge emprisonnée dans la grille flottante qui détermine ce courant. Dans le cas d'une cellule single-level cell (SLC), on ne discerne que deux états : passant ou bloquant, un seul bit est sauvegardé par cellule. Dans le cas d'une cellule multi-levels cell (MLC), on discerne des états intermédiaires et on peut obtenir plusieurs bits par cellule.

— L' **écriture** peut être réalisée à l'aide de l'injection d'électrons chauds et l'effet tunnel Fowler-Nordheim. L'injection d'électrons chauds consiste à faire passer un courant d'électrons entre la source et le drain à l'aide d'une différence de potentiel suffisante. L'application d'un fort potentiel positif sur la grille de contrôle permet d'attirer une partie de ces électrons, circulant entre la source et le drain, dans la grille flottante. L'injection d'électrons a tendance à altérer l'oxyde : le nombre de cycles est ainsi limité. En appliquant un potentiel nul sur la grille de contrôle et un fort potentiel sur la source, le drain et le substrat, on chasse les charges de la grille flottante vers le substrat. Comme le substrat est polarisé, cette opération est réalisée sur plusieurs cellules à la fois : c'est le "flash" !



14. La fiabilité des mémoires

Selon une étude [34] de Google en 2009, 8% des modules mémoires DIMM considérés subissent une erreur corrigible par l'ECC par an.

Il n'est pas possible de considérer les mémoires comme étant des composants parfaitement fiables. L'étude de la fiabilité des mémoires nous permet d'appréhender les facteurs qui peuvent influer sur le dysfonctionnement d'une mémoire.

14.1 Le processus de fabrication

Le processus de fabrication d'un circuit intégré est extrêmement complexe. On essaye en général d'obtenir la finesse de gravure la plus fine possible pour avoir la plus forte densité de transistors.

Or ce processus n'est pas parfait : certains transistors sont défectueux dès leur fabrication.

À l'échelle d'une puce, un transistor défectueux peut impliquer le dysfonctionnement de la puce complète ou non. Par exemple, un transistor défectueux dans un seul cœur d'un processeur multicœur peut être contourné en désactivant le cœur en question et en vendant le processeur moins cher. Le taux de puces qui peuvent être utilisées sur un wafer s'appelle le **yield**. Dans une mémoire, il peut être possible de désactiver certaines parties de la mémoire et considérer le circuit final comme une mémoire de plus faible capacité.

D'autre part le processus de fabrication n'est jamais parfaitement homogène à l'échelle du wafer. Les caractéristiques électriques des transistors varient selon leur emplacement physique. Ceci est particulièrement problématique pour les circuits analogiques que sont les mémoires. Si les transistors ne sont pas bien équilibrés, les capteurs analogiques pourraient se tromper et mésinterpréter l'état 0 ou 1. Pire, les éléments aux alentour d'un transistor peuvent affecter ses caractéristiques. Ainsi, les mémoires sont implémentées avec un motif régulier, et l'on rajoute parfois des rangées de cellules inutiles (*dummy cells* en anglais) en bordure pour augmenter la régularité des transistors utiles.

La conception et le dimensionnement des mémoires sont donc un processus qui ne peut se faire qu'en concertation avec les ingénieurs backend.

14.2 Le vieillissement

En plus de la variabilité spatiale liée au processus de fabrication, il faut également faire avec la variabilité temporelle. Avec le temps, les dopants introduits lors de la fabrication ont tendance à se diffuser, processus qui s'accélère avec la température. Ainsi, tout transistor voit ses propriétés électriques se dégrader avec le temps. Ceci peut mener à des cellules mémoires

défectueuses.

Autre phénomène de vieillissement, l'électromigration est le déplacement de particules métalliques lorsqu'elles sont traversées par un courant. Cela va amaincir certaines portions des pistes métalliques qui sont susceptibles de claquées, comme un fusible, lorsque trop minces par rapport au courant.

Certaines opérations particulières comme l'injection d'électrons chauds dans les cellules de mémoires flash (cf section 13.3), peuvent dégrader rapidement une cellule. On donne généralement 10 000 à 100 000 écritures possibles sur une cellule de mémoire flash.

Il faut donc être capable de répartir les écritures sur toutes les cellules, ou de désactiver des parties de la mémoire dynamiquement pour s'adapter au vieillissement.

14.3 Les mémoires sont des circuits intégrés

Comme tout circuit intégré, les mémoires sont également susceptibles aux injections de fautes, vues en détail dans le chapitre 11. Avec toutefois quelques spécificités.

La plupart des cellules mémoires sont "statiques", leurs valeurs ne sont pas mises à jour souvent. À l'exception de l'opération *Refresh* de la DRAM qui est exploitée par Rowhammer par exemple (cf chapitre 27). Ainsi la plupart des perturbations ont peu d'effets sur les données au repos.

Exemple 14.1

Une injection de faute par impulsion électromagnétique sur le SoC BCM2837 du Raspberry Pi 3 n'a pas de conséquences sur les données au repos, mais peut corrompre les transferts entre les mémoires caches [40].

Par contre l'utilisation d'un laser a été démontrée efficace pour altérer la valeur de cellules mémoires SRAM [1].

14.4 Comment durcir les mémoires ?

Le durcissement est ici le processus de modifier un système pour le rendre résilient face aux perturbations ou aux autres défauts.

14.4.1 ECC

On appelle généralement mémoire error correcting code (ECC) des DRAM intégrant un dispositif de correction d'erreur. De manière générale, l'ECC permet de récupérer de fautes accidentelles.

L'ECC seule n'est pas adaptée pour contrer un attaquant actif, même si elle peut lui compliquer la vie.

Le principe est de rajouter de la redondance sous la forme de n bits de redondance pour chaque bloc de l bits de données.

Exemple 14.2 - *Codes de Hamming*

Les codes de Hamming sont une méthode générique pour encoder et décoder des données avec de la redondance. Chaque message de l bits est sauvegardé dans un vecteur de $l + n$ bits, où les n bits de redondance sont choisis avec un schéma spécifique.
Prenons l'exemple des codes de Hamming (7,4), avec $l = 4$ et $n = 3$; c'est à dire 3

bits de redondance pour chaque mot de 4 bits. Ce schéma permet de détecter 2 bits d'erreurs et d'en corriger 1 bit.
Chaque donnée sera donc codée sur 7 bits $(c_0, c_1, c_2, c_3, c_4, c_5, c_6)$. Nous pouvons définir les équations d'encodage suivantes, permettant de corriger 1 bit d'erreur :

$$(c_0, c_1, c_2, c_3) = \text{message de 4 bits} \tag{14.1}$$
$$c_4 = c_0 \oplus c_1 \oplus c_2 \tag{14.2}$$
$$c_5 = c_0 \oplus c_2 \oplus c_3 \tag{14.3}$$
$$c_6 = c_0 \oplus c_1 \oplus c_3 \tag{14.4}$$

Voici ce qui se passe si l'on vérifie quelques mots :
- 1010001 : c_4, c_5, c_6 sont corrects, le mot est bon.
- 0010001 : c_4, c_5 et c_6 sont incorrects. Donc c_0 est erroné.
- 1110001 : c_4, c_6 incorrects, c_5 correct. Donc c_1 est erroné.
- 1010101 : c_4 incorrect, c_5, c_6 corrects. Donc c_4 est erroné.

La redondance rajoutée diminue la capacité utile de la mémoire.

14.4.2 Cellules mémoires durcies

Au delà de la cellule mémoire SRAM à 6 transistors vue dans la section 13.1, il est possible de concevoir des cellules mémoires durcies. Un example classique est la cellule DICE [7], pour *dual interlocked storage cell*. C'est une cellule à 12 transistors.

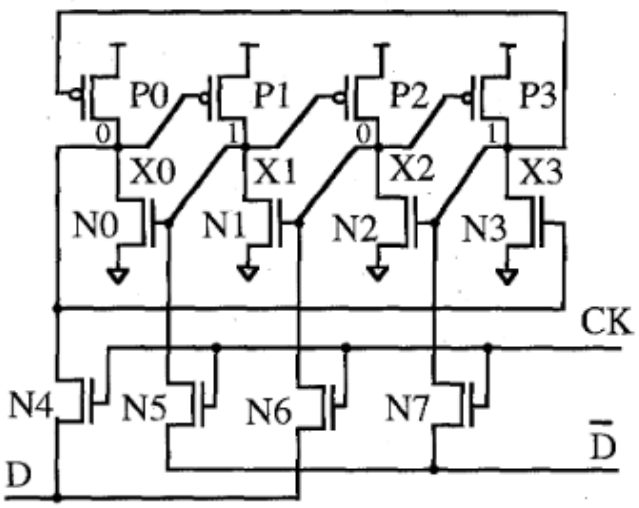


FIGURE 14.1 – Schéma électrique de la cellule mémoire DICE (extrait de [7]).

Le principe est d'introduire de la redondance dans la boucle de rétroaction. Une SEU ne peut plus changer l'état de la cellule mémoire en entier.

Les cellules mémoires durcies sont très utilsées dans les systèmes critiques (défense, aérospace, etc.).

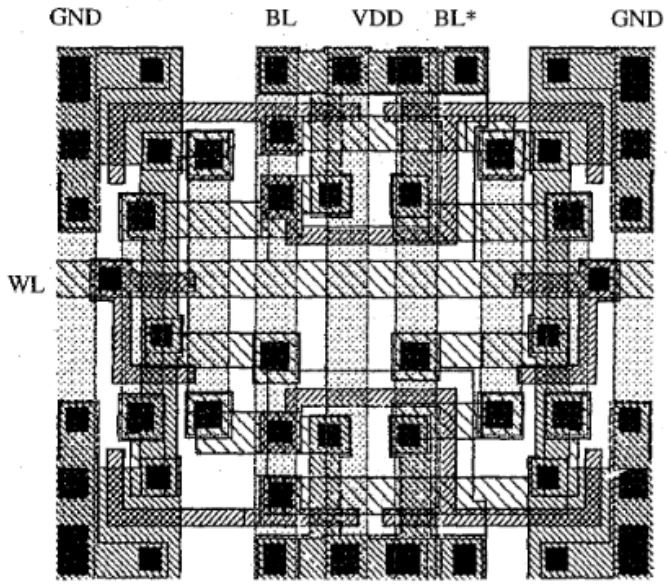
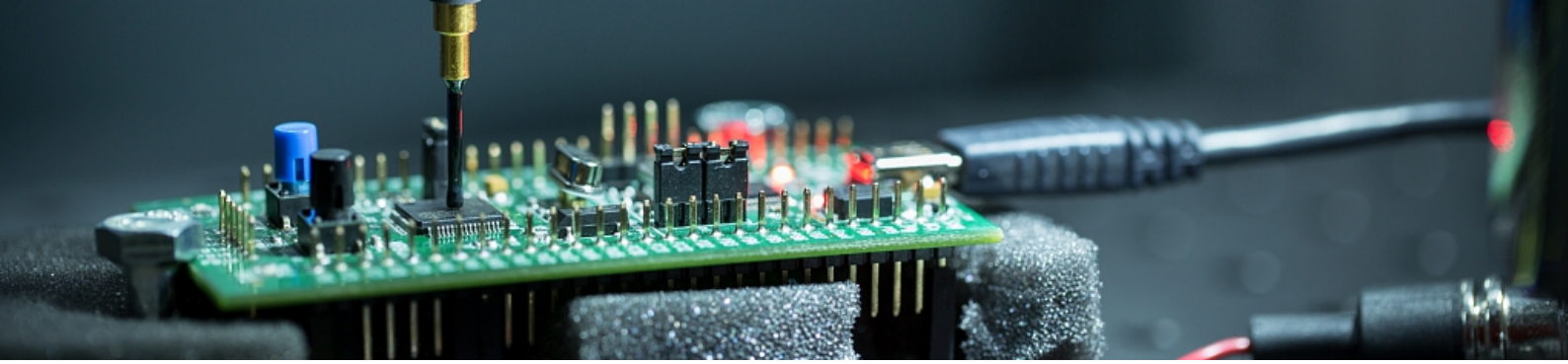


FIGURE 14.2 – Layout de la cellule mémoire DICE (extrait de [7]).



15. La hiérarchie mémoire

Nous avons vu que la mémoire flash permet la meilleure densité et à l'avantage d'être non volatile. Pourquoi est-ce que toutes les mémoires ne sont donc pas des mémoires flash ?

15.1 La latence d'un accès mémoire

15.1.1 Définition

Une propriété très importante d'une mémoire que nous n'avons pas encore abordée est sa latence.

Définition 15.1

La latence est le temps nécessaire pour un accès à une information ou un transfert (une lecture, une écriture, sur un réseau, dans une mémoire, ...).

! La latence ne doit pas être confondue avec le débit qui est la quantité d'information transférée durant une période définie.

Exercice 15.1 - *Calcul de latence et débit*

Pour transférer une grosse quantité de données d'un point A à un point B, nous décidons d'utiliser un camion transportant 1000 disques durs (HDDs) de 5 To. Le camion met 12 h pour faire le trajet. **Quel sont le débit et la latence de ce transfert d'information ?** Quel serait le temps nécessaire pour transférer la même quantité de données sur une connection ethernet standard de 1 Gbit s^{-1} ?

15.1.2 Quelques latences à avoir en tête

Tous les accès mémoires ne se valent pas. Les ordres de grandeur des différentes latences dans un système sont radicalement différents.

Exercice 15.2 - *HDD vs SSD*

Vaut-il mieux utiliser un disque dur HDD ou un SSD présent sur le réseau local ? Discutez des différents avantages et inconvénients de ces solutions.

TABLE 15.1 – Ordres de grandeur des latences dans un ordinateur.

Description	Latence
Période de l'horloge	1 ns
Accès RAM	100 ns
Lire un bloc 4K au hasard sur un SSD	100 µs
Aller-retour ethernet	100 µs
Aller-retour wifi	1 ms
Positionnement de tête de lecture HDD	10 ms
Transmission réseau longue distance	100 ms

15.1.3 Les limites physiques de la latence

Serait-il possible d'obtenir une mémoire parfaite, où augmenter la taille de la mémoire n'affecte pas négativement la latence ? La réponse est **non** : asymptotiquement la latence croit en $O(\sqrt{n})$ de la quantité mémoire n. Cette propriété se retrouve à la fois en théorie et en pratique.

En théorie, il suffit de supposer que la densité spatiale de la mémoire a une valeur maximale d, en bit m^{-2}. Nous savons que l'information a une vitesse maximale c, la vitesse de la lumière. Ainsi les cellules mémoires accessibles en un temps (la latence) t couvre un disque de surface $S = \pi \cdot (t \cdot c)^2$: les cellules mémoires sont disposées sur la surface du wafer. Ce qui donne une quantité de données accessibles (en bits) de $Q = d \cdot S = d \cdot \pi \cdot (t \cdot c)^2$. En retournant l'équation, on voit que $t \propto \sqrt{Q}$: la latence est proportionnelle à la racine carrée de la quantité de mémoire adressée.

En pratique, voici, sur la figure 15.1 en rouge, les temps nécessaires pour parcourir les éléments d'une liste chainée suivant sa taille.

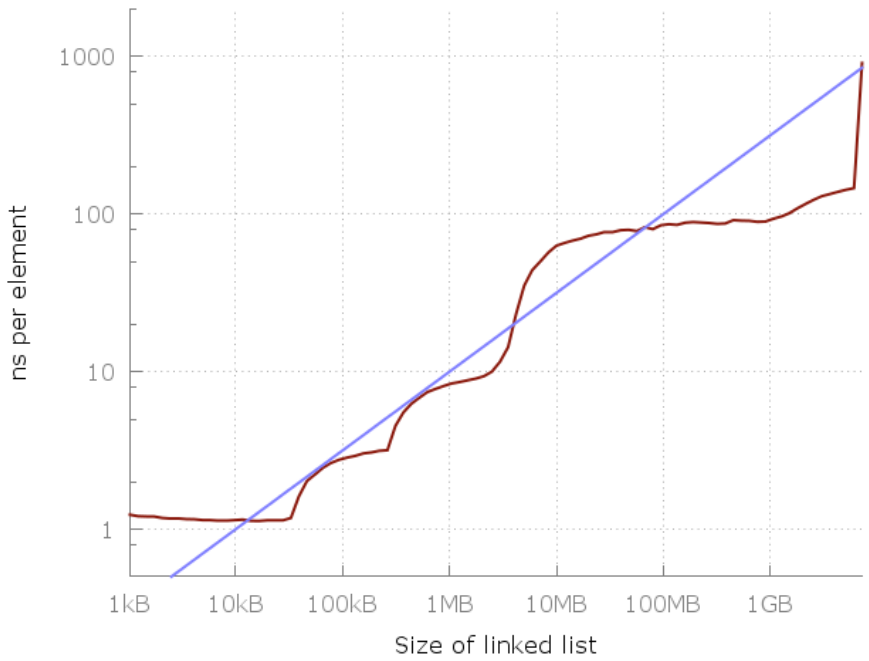


FIGURE 15.1 – Temps nécessaire pour parcourir tous les éléments d'une liste chainée selon sa taille. La ligne bleue représente $O(\sqrt{n})$. Source : https://github.com/emilk/ram_bench.

Est affiché en bleu, la progression attendue de la latence si elle est en $O(\sqrt{n})$. À noter que la correspondance n'est peut-être qu'une coïncidence. Ce que l'on observe surtout, c'est que

la latence n'a pas une progression régulière en fonction de la taille de la liste. On peut voir l'impact des différentes mémoires (latence et capacité) de la hiérarchie mémoire.

Cette limite est à prendre en compte lors de la conception et l'implémentation de vos algorithmes : si de grandes quantités de données sont en jeu, il est faux de considérer qu'un accès mémoire prend un nombre de cycles en $O(1)$, la réalité est que cet accès est en $O(\sqrt{n})$ où n est la taille de la mémoire en jeu.

Sur la figure 15.1, nous pouvons observer l'existence de petites mémoires, plus rapides pour les petites tailles de données. Il s'agit de mémoires caches.

15.2 Les mémoires caches

Définition 15.2

Une mémoire cache est une mémoire qui duplique une partie de l'espace mémoire, dans le but de permettre des accès rapides aux données les plus couramment utilisées.

L'efficacité d'une mémoire cache repose sur les principes de localité spatiale et temporelle. Si une donnée est accédée, il est fortement probable qu'un nouvel accès à cette même donnée ait lieu rapidement, c'est la localité temporelle. Il est également fortement probable qu'un accès à une donnée à une adresse contigüe ait lieu rapidement : c'est la localité spatiale. Ainsi, il est possible d'accélérer efficacement les accès mémoires en ne dupliquant qu'une fraction de la mémoire.

Le fonctionnement d'une mémoire cache moderne est très complexe, mais nous allons voir ici une version simplifiée.

15.2.1 Principes généraux

Une mémoire cache reçoit des requêtes mémoires et tente d'y répondre si possible. Pour cela elle va sauvegarder un certain nombre de données, par bloc que l'on appelle une ligne de cache, alignée selon leur taille (cf section 16.2).

Exemple 15.1

Une mémoire cache avec 4 lignes de 16 o contient au total $4 \cdot 16 = 64$ o.

Pour chaque ligne de cache, il faut également sauvegarder l'adresse correspondante, appelée le « tag », et un bit de validité pour savoir si cette ligne correspond réellement à des données valides.

Si l'on part d'une mémoire cache vide, une requête en lecture ne trouvera forcément pas la donnée correspondante dans le cache. On parle d'un **miss**. Dans ce cas, la mémoire cache transmet une requête en lecture au niveau supérieur, qui peut être une autre mémoire cache ou la mémoire principale. La réponse à cette requête est obtenue, la ligne de cache correspondante est sauvegardée et la réponse à la requête initiale est envoyée.

Si la mémoire cache reçoit à nouveau une requête en lecture pour une donnée dans cette ligne de cache, l'adresse de la requête est comparée avec succès aux tags des lignes valides : la réponse peut être renvoyée immédiatement. Il s'agit d'un **hit**.

Pour une requête en écriture, il est possible d'écrire sur une ligne de cache non valide et de continuer le calcul. Charge pour la mémoire cache de récupérer le reste de la ligne, et de propager l'écriture. S'il n'y a pas de ligne de cache non valide, il faut procéder à l'éviction d'une ligne, opération précisée dans la sous-section 15.2.3.

Les mémoires caches sont nommées selon leur niveau (*level* en anglais). La mémoire L1 est la plus rapide, la plus petite et donc la plus proche du pipeline. La mémoire cache L2 est

un peu plus lente et un peu plus grosse. Etc. pour les mémoires L3, L4, ...Le dernier niveau de cache avant la RAM est le « cache de dernier niveau » (last level cache (LLC)).

Une notation alternative est d'utiliser le symbole $ pour désigner une mémoire cache ($ = cash). I$ désigne parfois le cache instruction, aussi noté L1I.

15.2.2 Associativité

Pour certaines mémoires caches, les données de n'importe quelle adresse de la mémoire principale peuvent être sauvegardées dans n'importe quelle ligne de cache. On parle alors de mémoire complètement associative (*fully associative* en anglais). Malheureusement, cela implique de nombreuses portes logiques pour gérer le routage des données, via des multiplexeurs.

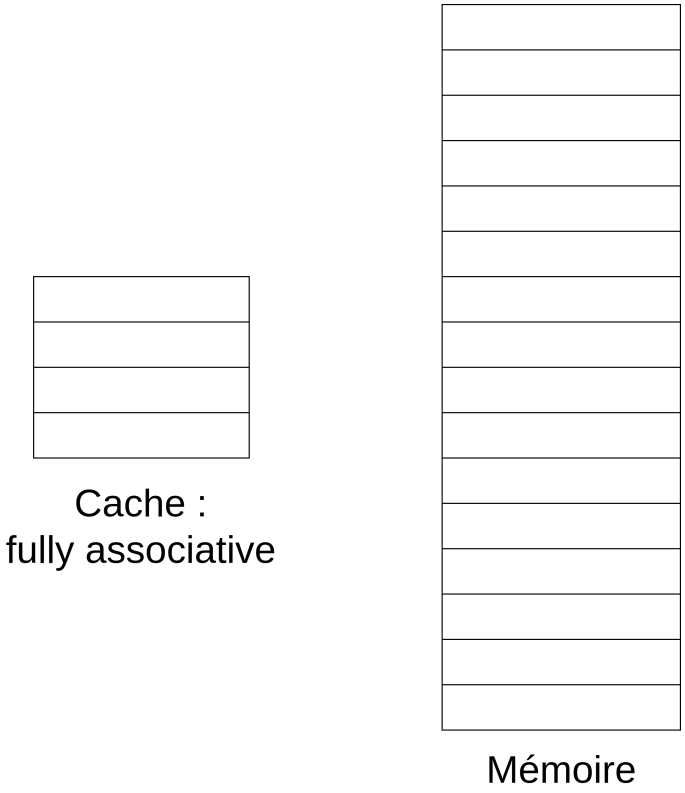
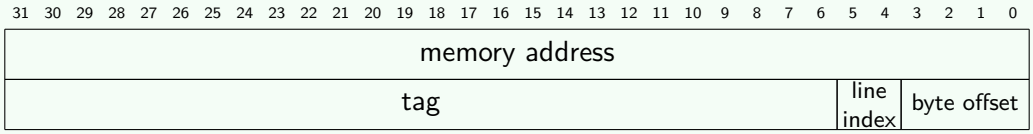


FIGURE 15.2 – Toute ligne de cache peux contenir n'importe quelle adresse mémoire.

Il est possible également d'éviter toute logique de routage à l'aide d'une **correspondance directe** (*direct mapping*). Dans ce cas, toute adresse mémoire ne peut être sauvegardée que dans une seule ligne de cache prédéterminée.

Exemple 15.2

Reprenons notre mémoire cache avec 4 lignes de 16 o. Nous divisons notre espace mémoire en 4 parties sur les bits de poids faible de l'adresse, chaque partie est assignée à une seule ligne de cache. Le rôle des bits de l'adresse est donc le suivant :



Ainsi, les bits $5-4$ de l'adresse définissent la ligne de cache à utiliser.

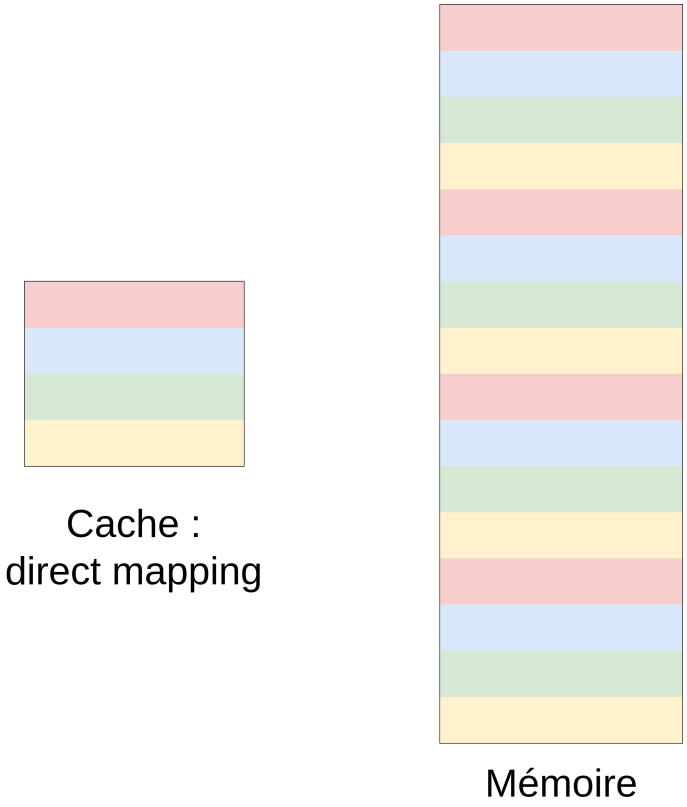
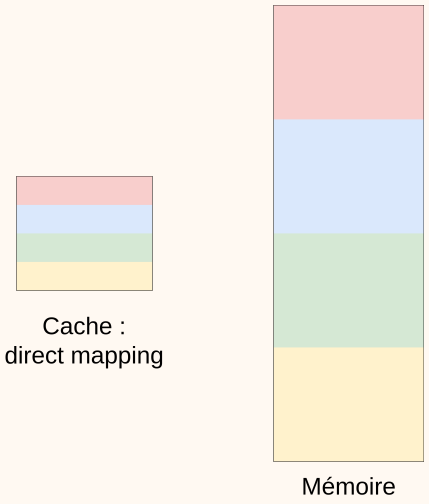


FIGURE 15.3 – Assignation, par couleur, des 4 lignes de cache dans un cache à correspondance directe.

Exercice 15.3 - *Pourquoi une correspondance sur les bits de poids faibles ?*

À votre avis, pourquoi la correspondance est faite sur les bits de poids faibles et non ceux de poids forts ?



Le souci avec la correspondance directe est qu'il y a une forte chance que certaines lignes de caches soient surutilisées, s'il y a collision des adresses, et donc une chute des performances.

Le compromis est le cache à **associativité par ensemble** (*set associative cache*). Dans ce cas l'associativité existe, mais est limitée. Un cache 4-way associatif signifie que les données à une même adresse mémoire peuvent être potentiellement sauvegardées dans 4 lignes de caches

possibles.

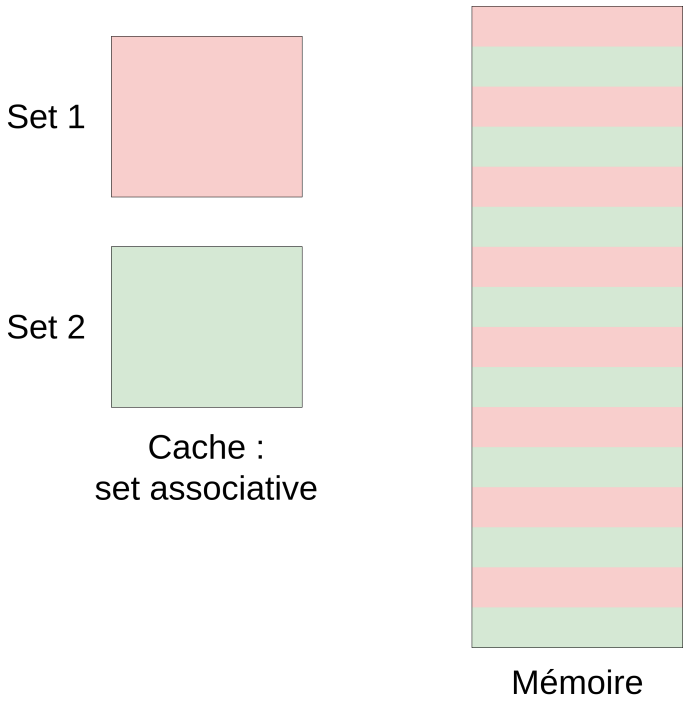
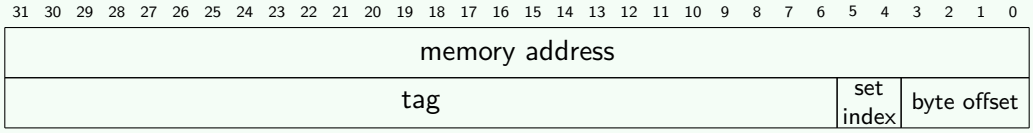


FIGURE 15.4 – Assignation, par couleur, des 2 ensembles de ligne de cache dans un cache à associativité par ensemble.

Exemple 15.3

Modifions notre mémoire cache d'exemple pour avoir maintenant 8 lignes de 16 o réparties en 4 ensembles. Ainsi chaque ensemble possède 2 *ways*. Le rôle des bits de l'adresse est donc le suivant :



L'associativité est toujours limitée : les nombres de ways couramment rencontrés sont 4, 8 et 16.

15.2.3 Éviction

> *" There are two hard things in computer science : cache invalidation, naming things and off-by-one errors. "*
>
> **– Phil Karlton et Jeff Atwood**

Lorsque le cache est plein, et qu'il faut remplacer une ligne de cache existante par une nouvelle, il devient nécessaire d'évincer une ligne de cache valide. Quelle ligne de cache choisir ? C'est le problème de l'invalidation ou de l'éviction de lignes de caches.

Plusieurs méthodes sont couramment implémentées :

— La politique d'éviction **aléatoire** : on choisit une ligne au hasard dans l'ensemble ciblé. Le risque étant d'enlever la ligne de cache qui sera prochainement utilisée. Un vrai aléa étant généralement couteux à générer, il s'agit en fait d'une éviction pseudo aléatoire.
— La politique **least recently used (LRU)** : on choisit la ligne de cache qui n'a pas été touchée depuis le plus longtemps. Cette politique, qui est l'une des plus efficaces, est aussi très couteuse à implémenter.
— La politique **pseudo-LRU** : on approxime la politique, LRU mais avec des heuristiques moins couteuses. La politique de cette famille la plus courante est la **bit-PLRU**. Elle fonctionne en ajoutant un seul bit d'état pour chaque ligne de cache, initialement à 0. À chaque fois qu'une ligne est touchée, on met ce bit à 1. Quand tous les bits de l'ensemble sont à 1, on les réinitialise à 0 sauf celui du dernier accès. On évince en priorité les lignes dont le bit est à 0. La performance de cette politique est très proche de la politique LRU.

Une propriété importante de l'éviction est qu'elle est déterministe. En connaissant l'ordre des accès mémoire et les tags associés, un attaquant peut savoir l'ordre des évictions pour n'importe quelle séquence d'adresse.

15.2.4 Propriétés spécifiques liées aux caches

Il existe d'autres propriétés des caches qui peuvent avoir leur importance lors d'une attaque. Ces propriétés peuvent souvent être configurées soit au niveau du cœur à l'aide de registres de configuration dédiés, soit au niveau de la page mémoire (cf chapitre 17).

Cached/uncached Les adresses mémoires sont aussi utilisées pour accéder à des périphériques, ce que l'on verra plus en détail dans le chapitre 16. Mais l'existence des mémoires caches peut perturber l'utilisation de ces périphériques. Il est donc nécessaire de pouvoir définir des zones mémoires qui ne sont pas mises en cache. La memory management unit (MMU) se charge de cela via la configuration des pages mémoires sur les systèmes complexes. Sur des systèmes simples, seule la zone d'adressage des mémoires est mise en cache, quand il y en a. À chaque adresse, on peut associer la propriété *cached* si elle peut être mise en cache, ou *uncached* sinon.

Une variante est la possibilité d'accéder à un cache en mode *non allocating* : si la donnée n'est pas trouvée, elle est récupérée depuis les étages supérieurs sans modification du cache interrogé. La donnée ne sera donc toujours pas présente pour une future requête.

Inclusive/exclusive Lorsqu'il y a plusieurs mémoires caches en cascade dans un système (cf section 15.3), se pose la question de la redondance des données. Est-ce que les données du cache le plus grand contiennent les données du cache le plus petit ? Si c'est le cas, on parle de cache *inclusive*, sinon de cache *exclusive*.

Write-back/write-through Lorsque le cœur envoie des requêtes en écriture à la mémoire cache, celle-ci à le choix : doit-elle répliquer avec une requête en écriture sur la mémoire de niveau supérieur ? Si la requête est envoyée immédiatement au niveau supérieur, on parle de *write-through*. Sinon, le cache peut attendre l'invalidation d'une ligne avant d'envoyer cette requête, lors de l'éviction ou à l'aide d'une instruction dédiée par exemple. On parle alors de *write-back*.

15.3 L'organisation mémoire dans un SoC

Suivant la complexité du système, il n'y a souvent pas qu'une seule mémoire cache, mais plusieurs. C'est cet ensemble de mémoires, en conjonction avec la mémoire principale, que l'on nomme la hiérarchie mémoire. Il n'existe pas une norme unique ni une architecture canon. La hiérarchie mémoire s'adapte au système dans sa topologie et sa configuration. Chaque concepteur fait ses propres choix technologiques.

Toutefois, pour optimiser la microarchitecture, certaines caractéristiques se retrouvent sou-

vent.

Séparation des caches L1 données et instruction En violation du principe de l'architecture de Von Neumann, il est plus efficace de considérer que les instructions ne soient pas lues ni écrites par la logique du programme, mais uniquement dans le but d'exécuter celui-ci. Ainsi il est efficace de séparer les caches instructions et données pour le niveau L1, celui relié au pipeline. En effets, les instructions seront uniquement lues lors de l'étage de fetch. Les données peuvent, elles, être lues ou écrites lors de l'étage mémoire.

Le débat entre les architectures Von Neumann et Harvard n'est plus vraiment d'actualité : les machines modernes sont un hybride des deux !

Le cache L2 unifié Conceptuellement, il ne devrait y avoir qu'une seule mémoire dans un système moderne, selon une architecture Von Neumann. Les données et les instructions sont généralement fusionnées au niveau L2. On parle alors de mémoire L2 unifiée (ou *unified*).

> **!** Attention, une L2 unifiée ne veut pas dire que la mémoire soit nécessairement partagée (*shared*) entre plusieurs cœurs.

Cache de dernier niveau Autre cache notable, le cache de dernier niveau (*LLC*) est en général le point ou la mémoire est partagée dans le système complet. Cette mémoire est alors partagée entre les cœurs, parfois le graphics processing unit (GPU), etc. On parle parfois de point de cohérence, tous les éléments du système voient la même donnée à ce point là. Le LLC est souvent la L3, mais peux être la L4 voir directement la L2.

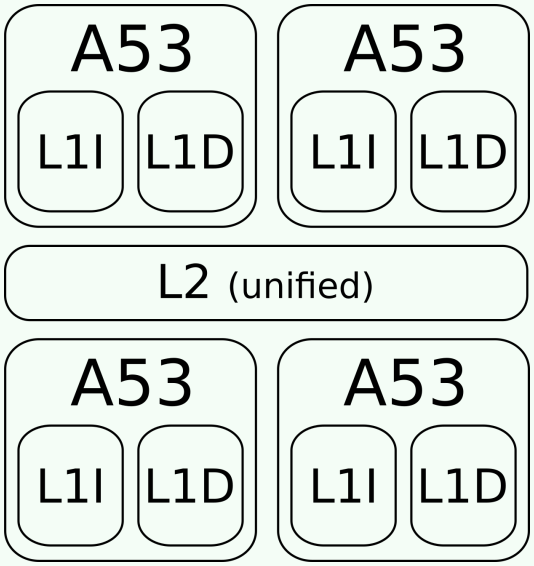
Exemple 15.4 - *Apple M2 SoC, jeu d'instruction ARMv8*

Dans ce SoC cohabitent deux types de cœurs : ceux à hautes **p**erformances, et ceux à haute efficacité **é**nergétique. Le GPU partage la mémoire L3 avec tous les cœurs, d'où la dénomination « système ».

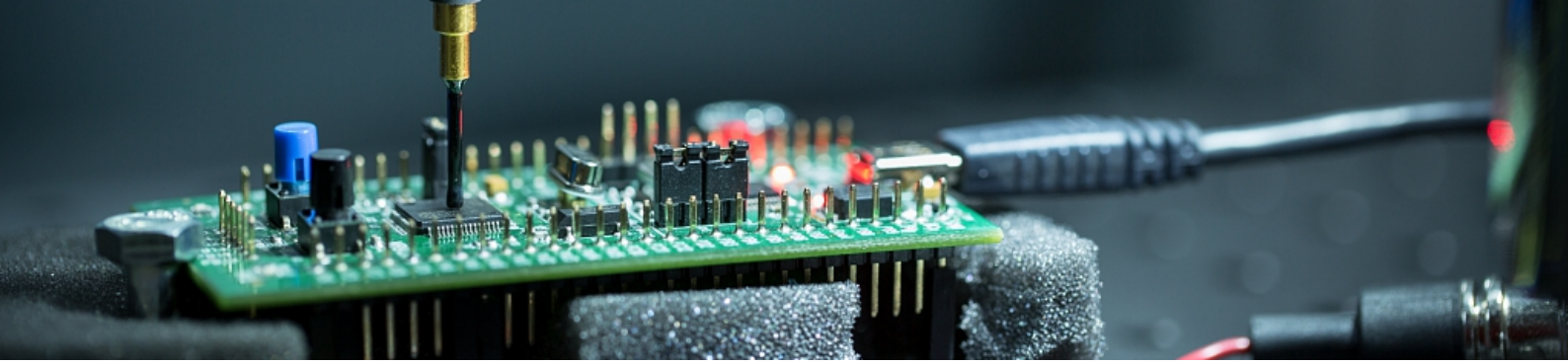
Dénomination	Taille
pL1-instruction	192 ko
pL1-data	128 ko
pL2-shared	16 Mo
eL1-instruction	128 ko
eL1-data	64 ko
eL2-shared	4 Mo
L3-system	8 Mo

Exemple 15.5 - *Description de la hiérarchie mémoire du BCM2837 (SoC du Raspberry Pi 3)*

Le BCM2837 possède 4 cœurs A53, mais seulement deux niveaux de cache. Le L2 est unifié et partagé.



Les caches de niveau L1 sont généralement considérées comme faisant partie du cœur lui-même. À l'inverse des autres caches.



16. L'adressage comme interface

Les mémoires sont parfois directement lues ou écrites à l'aide des instructions `load` ou `store`. Certaines mémoires, les mémoires caches notamment, ne sont pas directement accessibles : elles sont transparentes pour l'architecture définie par le jeu d'instructions.

Définition 16.1 - *Sémantique d'accès mémoire*

On appelle ici la sémantique d'accès mémoire, la sémantique définie par les deux instructions suivantes :

- `load rd, 0(rs1)` : on lit une donnée présente en mémoire à l'adresse contenue dans le registre `rs1` (avec un offset, ici 0). Cette donnée est alors placée dans le registre `rd`.
- `store rs1, 0(rs2)` : on écrit la donnée présente dans le registre `rs1` en mémoire à l'adresse contenue dans le registre `rs2` (avec un offset, ici 0).

Nous considérons que la taille des registres est constante (typiquement 32-bit ou 64-bit). On peut rajouter des variantes pour gérer plusieurs tailles de données, mais il ne s'agit alors que d'une optimisation. Dans cette sémantique, on suppose l'existence d'une seule et unique mémoire, les instructions ne référencent pas plusieurs mémoires.

Contre-intuitivement, cette sémantique d'accès mémoire n'est pas utilisée uniquement pour interagir avec des mémoires.

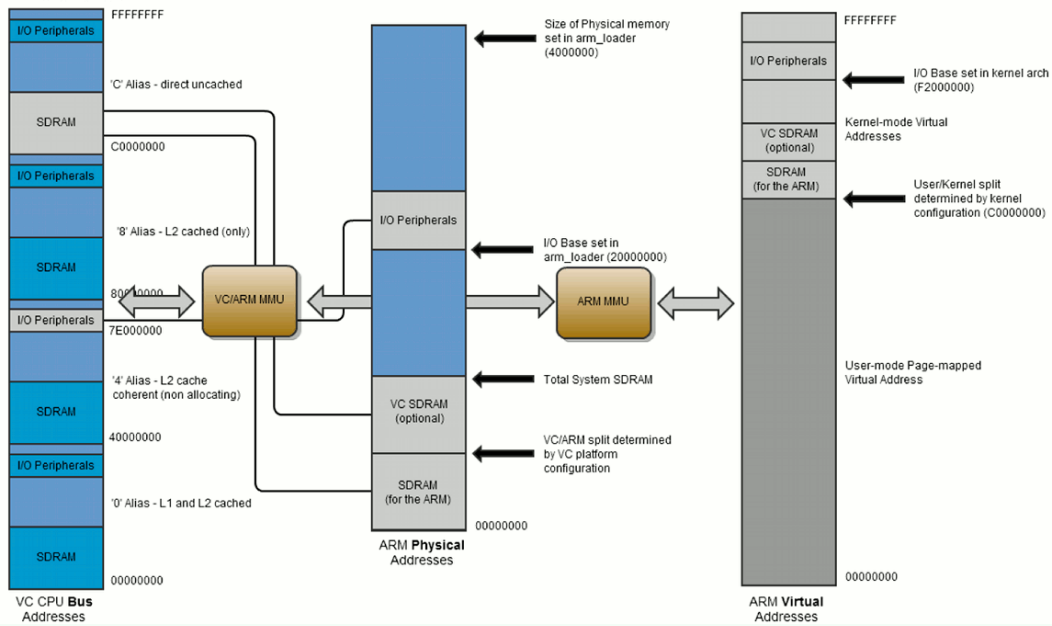
16.1 L'accès aux périphériques via le bus mémoire

La sémantique d'accès mémoire est utilisée pour accéder aux périphériques en général, pas que la mémoire. Ces périphériques incluent ceux pour la communication (UART, USB, ethernet, etc.), pour l'audio (I2S, PCM, etc.), les GPIOs et plein d'autres choses. En général, les périphériques permettent à votre processeur d'interagir avec l'environnement.

Chaque périphérique dispose d'une interface via la mémoire : en écrivant à certaines adresses dédiées, en lisant aux mêmes ou à d'autres, nous contrôlons le périphérique. En général, l'application ne va pas directement interagir avec ces adresses mémoires. Cette tâche est déléguée à un programme dédié, proposant une application programming interface (API) à l'application : c'est le pilote.

Exemple 16.1 - *L'organisation mémoire du BCM2835 (SoC du Raspberry Pi 1)*

La documentation de l'organisation mémoire du BCM2835 du Raspberry Pi 1, qui s'applique approximativement pour les Raspberry Pi 2 et 3 propose le schéma suivant :



Nous pouvons voir 3 visions différentes de la mémoire : la vision du bus, utilisée notamment pour la direct memory access (DMA), les adresses physiques vues par le cœur et une suggestion d'organisation mémoire pour la mémoire virtuelle du noyau Linux (cf. chapitre 17). Remarquez que la même SDRAM, qui est une DRAM, est présente 4 fois à des adresses différentes sur le bus. Il s'agit de la même mémoire, accédée avec des configurations différentes de la hiérarchie mémoire : *uncached*, *L2 cached only*, *L2 cache coherent (non allocating)* et *L1 and L2 cached*

Exercice 16.1 - *Les GPIOs du BCM2835*

Voici un extrait de ce que la documentation du BCM2835 a à dire sur la gestion des general purpose input/output (GPIO).

6.1 Register View

The GPIO has 41 registers. All accesses are assumed to be 32-bit.

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W

De cette description succincte, quelle est selon vous la procédure (par exemple le programme C) pour mettre le pin numéro 31 à un niveau haut, pin désigné par le bit de poids fort du Set 0 ?

La fin de l'idempotence Dans un système avec un seul cœur, on pourrait penser que lire deux fois la même adresse à la suite est inutile : on obtiendrait systématiquement la même valeur.

```
load x3, &x5
load x4, &x5
```

Dans ce programme ci-dessus, est-ce que `x3==x4` nécessairement ? On sait maintenant que la réponse est **non**, `&x5` peut pointer vers un périphérique. Si c'est le buffer des données reçues par l'UART, cette valeur peut changer pour chaque donnée reçue.

Par défaut, le compilateur suppose que deux lectures consécutives à une même adresse sont idempotentes. Le compilateur enlève donc une des deux, considérée inutile. Nous verrons dans le section 21.2 comment éviter ce comportement.

16.2 L'importance de l'alignement mémoire

L'alignement mémoire d'un bloc ou d'une structure de données correspond à sa disposition à une adresse multiple d'un entier. Par exemple un alignement sur 2 octets d'une structure signifie qu'elle commence à une adresse paire (multiple de 2, commençant à 0). L'alignement a une importance critique dans de nombreux cas.

— L'accès à un périphérique doit être réalisé avec le bon alignement. En cas d'accès non alignés, deux adresses valides gérant deux périphériques différents peuvent être accédées, souvent partiellement. Ce cas n'est parfois pas géré et une exception sera levée.

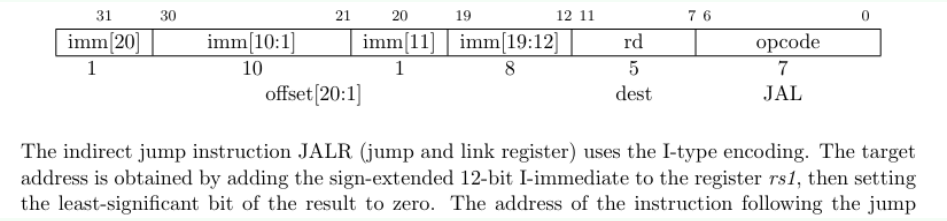
— Une structure de données critique pour les performances d'une application sera idéalement alignée avec les lignes de cache. En cas de chevauchement, il faudra deux accès mémoire au lieu d'un seul pour lire ou écrire sur cette structure.

L'alignement peut parfois être géré directement par le langage de programmation.

```
__attribute__((aligned(4096)))
char* data;
```

En C, la structure de données suivante sera alignée à 4096 o.

Exemple 16.2 - *L'alignement des sauts dans les jeux d'instructions, RISC-V*



The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register *rs1*, then setting the least-significant bit of the result to zero. The address of the instruction following the jump

extrait de la spécification RISC-V

Dans le jeu d'instruction RISC-V, les instructions font au minimum 16 bits (ou 2 octets). Il est requis selon la spécification que les instructions soient alignées sur 2 octets. Autrement dit, il est interdit d'avoir une instruction commençant à une adresse impaire.

Exemple 16.3 - *L'alignement des sauts dans les jeux d'instructions, Thumb-2*

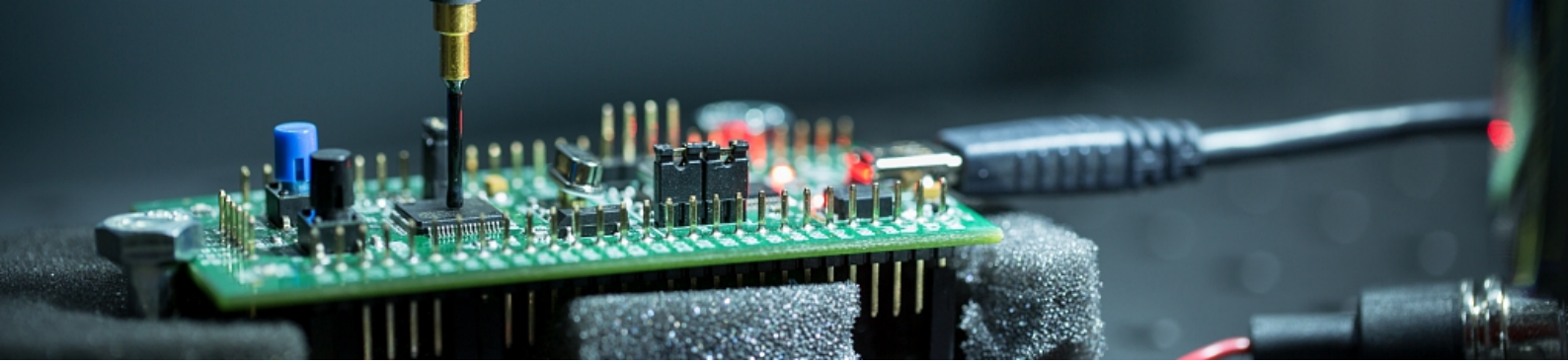
A4.1.1 Armv7-M and interworking support

Thumb interworking is held as bit [0] of an *interworking address*. Interworking addresses are used in the following instructions:

- BX or BLX.
- an LDR or LDM that loads the PC.

extrait de ARMv7-M Architecture Reference Manual

Thumb-2 est un jeu d'instruction de taille variable qui cohabite avec le jeu d'instruction ARMv7-M. On considère qu'il s'agit de deux jeux d'instructions, plutôt qu'une extension, à cause de la façon de basculer de l'un vers l'autre. Il faut en effet faire basculer l'étage de *decode* dans le nouvel état, dédié au ISA désiré, à l'aide du bit de poids faible de l'adresse pour certaines instructions : p. ex. `BX` pour les branchements, `LDR` lorsque l'on charge une valeur dans le `PC` directement (`PC` est le PC). Ce mécanisme utilisant le bit de poids faible de l'adresse, ces instructions requièrent un alignement de 2 octets pour l'adresse cible.



17. La mémoire virtuelle

Exemple 17.1 - *Gestion des périphérique*

Imaginez que chaque programme doit gérer lui-même ses interactions avec les périphériques. Programme `A` envoie une donnée sur l'universal asynchronous receiver-transmitter, et en attendant la réponse, laisse la main à programme `B`. Toutefois `B` veut savoir s'il a reçu une commande sur l'universal asynchronous receiver-transmitter et lit donc les données reçues. Ces données constituent-elles une commande pour `B` ou la réponse attendue par `A` ?
Multipliez ce conflit par le nombre de périphériques et vous obtenez un système inutilisable.

Exemple 17.2 - *Confidentialité des secrets cryptographiques*

Programme `A` veut sauvegarder une clé cryptographique secrète en mémoire, et la garder inaccessible aux yeux indiscrets. Programme `B` n'a qu'un seul but : lire toute la mémoire et l'afficher à l'écran. Ces deux rôles ne sont pas compatibles.

Ces exemples montrent qu'il faut être capable de contrôler l'accès à la mémoire, pour la confidentialité des données, mais aussi simplement pour la gestion de la complexité du système. Ce besoin se fait en particulier sentir si des processus concurrents s'exécutent sur le même système. Dans ce chapitre, nous verrons les techniques actuelles permettant cela.

17.1 Memory protection unit

Une memory protection unit (MPU) est un système de contrôle d'accès mémoire essentiellement utilisé dans les systèmes à bas couts tels que les microcontrôleurs. Une MPU ne modifie pas les adresses, mais décide si certains accès mémoires sont autorisés ou non.

Il existe de nombreuses variantes : l'implémentation de la MPU n'étant pas toujours standardisée.

Dans les STM32 Regardons en détail l'implémentation de la MPU dans les STM32 : la gamme de microcontrôleurs de la société STMicroelectronics[1]. La MPU permet ici de définir 16 régions, numérotés de 0 à 15 ou 15 est la région la plus prioritaire.

Une région est définie par une adresse de départ et une taille. Plusieurs régions peuvent se

1. Nous nous basons sur la documentation de ST : https://www.st.com/resource/en/application_note/dm00272912-managi

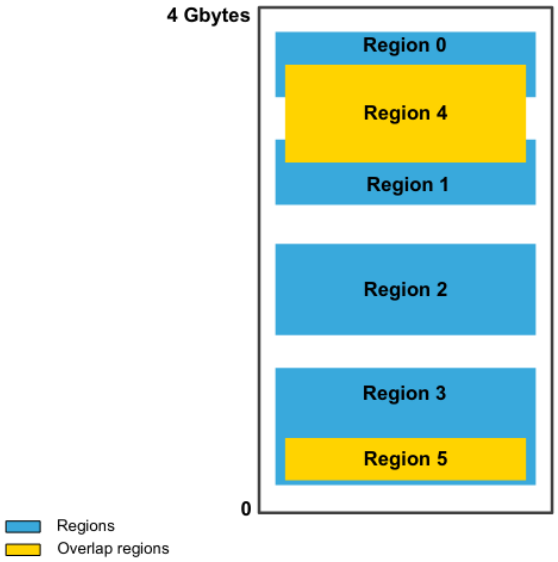


FIGURE 17.1 – Exemples de placement de régions en mémoire. Extrait tiré de la documentation de ST.

chevaucher. Chaque région est associée à un type de mémoire et à des attributs mémoires.

Le type de mémoire est utilisé pour définir le modèle mémoire (cf. chapitre 18) à utiliser : `normal` pour désigner une mémoire associée avec un modèle faible. `device` pour désigner un espace adressable utilisé pour accéder aux périphériques, conservant l'ordre des accès en lecture et écriture. `strong` pour un modèle mémoire séquentiel avec la pénalité associée en termes de performances, permet de désactiver les accès spéculatifs.

Les attributs mémoires comprennent :

— `XN` execute never, pour interdire l'exécution des données.
— `AP` data access permission, pour spécifier les droits d'accès entre RO (read-only), RW (read-write), No (pas d'accès).
— La *shareability* au sens large, c'est-à-dire la configuration des caches et du type de mémoire vu ci-dessus. La configuration est résumée sur la figure 17.2.

TEX	C	B	Memory Type	Description	Shareable
000	0	0	Strongly Ordered	Strongly Ordered	Yes
000	0	1	Device	Shared Device	Yes
000	1	0	Normal	Write through, no write allocate	S bit
000	1	1	Normal	Write-back, no write allocate	S bit
001	0	0	Normal	Non-cacheable	S bit
001	0	1	Reserved	Reserved	Reserved
001	1	0	Undefined	Undefined	Undefined
001	1	1	Normal	Write-back, write and read allocate	S bit
010	0	0	Device	Non-shareable device	No
010	0	1	Reserved	Reserved	Reserved

FIGURE 17.2 – Conséquences de la configuration de `TEX` type extension field, `C` cacheability, `S` shareability et `B` bufferability.

Une mauvaise configuration de la MPU peut n'entrainer qu'une baisse des performances. Mais elle peut déclencher également l'apparition de bugs pernicieux, par exemple lors de l'utilisation d'un modèle mémoire faible sur l'espace adressable réservé pour un périphérique, pouvant entrainer un réordonnancement des lectures et écritures par le matériel.

Exercice 17.1 - *Memory protection unit (MPU)*

Comment configurer la MPU pour assurer la confidentialité des secrets cryptographiques, scénario décrit au début du chapitre ?

17.2 Memory management unit et mémoire virtuelle

17.2.1 Généralités

La memory management unit (MMU) peut être vue comme une version étendue de la MPU. La MMU permet en plus la traduction d'adresses : l'adresse manipulée par le programme et celle vue par le matériel sont différentes. La MMU est le composant matériel mettant en place le mécanisme de mémoire virtuelle.

L'intuition derrière la mémoire virtuelle est de donner l'illusion à un programme qu'il est le seul utilisateur de la mémoire. Il ne peut pas entrer en concurrence avec un autre programme puisqu'il est le seul. En d'autres termes, la vision de la mémoire devient subjective. On appelle processus (*process* en anglais), l'entité logicielle ayant une même vision subjective de la mémoire. Deux processus différents ont des visions différentes de la mémoire.

Là encore, les implémentations varient. Nous allons regarder plus en détail l'implémentation RISC-V décrite dans la spécification *Volume 2, Privileged Spec v. 20211203* disponible en ligne.

17.2.2 La PMA pour tous

Dans la spécification RISC-V, la gestion de la mémoire virtuelle est séparée de la configuration du comportement mémoire. Les physical memory attributes (PMA) sont responsables de la déclaration des propriétés des zones mémoires (p. ex. la cacheability, l'ordre mémoire, l'idempotence des accès, etc.). Dans la logique RISC-V, les PMA sont figés au maximum : la possibilité de changer les attributs doit être limitée si possible.

17.2.3 Sv32 : 32-bit virtual memory system

RISC-V définit 4 systèmes de mémoire virtuelle (Sv32, Sv39, Sv48, Sv57) qui sont des variations d'un même principe. Nous regardons ici le Sv32, qui définit une mémoire virtuelle sur 2 niveaux, des adresses virtuelles sur 32 bits et des adresses physiques sur 34 bits. La MMU est le composant responsable de la traduction des adresses virtuelles, celles vues par le processus, en adresses physiques, celles vues par le matériel. Comme cette traduction est faite par page, c'est une traduction de virtual page number (VPN) vers physical page number (PPN).

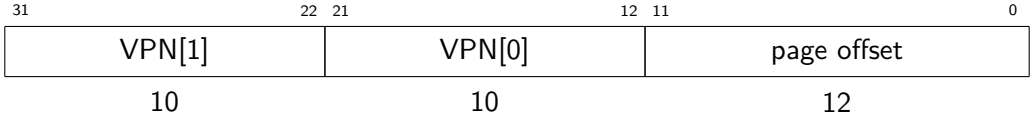


FIGURE 17.3 – Structure d'une adresse virtuelle, sur 32 bits.

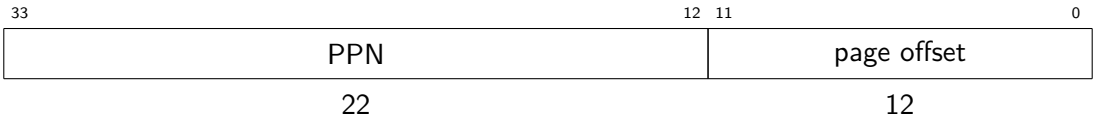


FIGURE 17.4 – Structure d'une adresse physique pour un système avec un bus sur 34 bits.

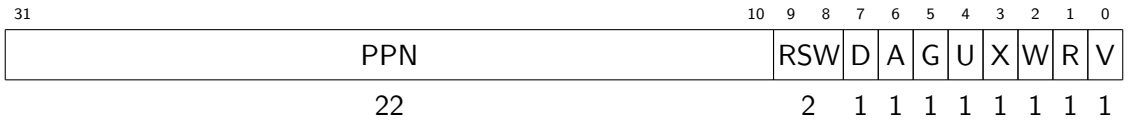


FIGURE 17.5 – page table entry (PTE)

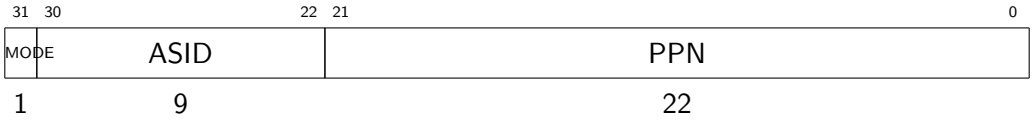


FIGURE 17.6 – La structure du registre `satp` : Supervisor Address Translation and Protection

Traduction La traduction se fait par bloc de taille fixe appelé une *page*, ici de taille 2^{12} octets soit 4 Kio. Ainsi les 12 bits de poids faible d'une adresse, offset dans une page, sont conservés entre adresses virtuelles et physiques. Les 20 bits de poids fort de l'adresse virtuelle sont traduit en 22 bits de poids fort pour constituer l'adresse physique.

Une **table des pages** contient un certain nombre de page table entry (PTE), une structure sur 4 octets décrivant la traduction pour une page. La table des pages doit elle-même être alignée sur la taille d'une page : elle commence à une adresse dont les 12 bits de poids faibles sont à 0.

Le processus de traduction (en omettant la gestion des exceptions) est le suivant :

1. Le champ `PPN` du registre `satp` (cf. figure 17.6) pointe vers l'adresse de la table des pages de premier niveau, qui est alignée sur 4 Kio. Le `MODE` est soit 0 pour désactiver la mémoire virtuelle ou 1 pour choisir le schéma Sv32.
2. À partir de la table des pages de premier niveau, le champ `VPN[1]` (sur la figure 17.3) sur 10 bits désigne une des 2^{10} entrées, une PTE, de la table ($2^{10} = 2^{12}/4$).
3. Cette PTE pointe vers une deuxième table des pages, celle de deuxième niveau si `X`, `W` et `R` sont tous à 0. Sinon il s'agit d'une PTE feuille : se reporter au point 5. Il s'agit alors d'une mégapage de taille 4 Mio. L'adresse de la table des pages de deuxième est désignée par les 22 bits du champ `PPN`.
4. Dans cette table de deuxième niveau, la PTE feuille est désignée par le champ `VPN[0]` de l'adresse virtuelle.
5. À partir d'une PTE feuille, concaténer le champ `PPN` avec l'offset génère l'adresse physique sur 34 bits.

Propriétés de la PTE Dans la PTE, certains bits (cf. figure 17.5) spécifient des attributs mémoires de la page en question. Par exemple, `X`, `W` et `R` donnent les droits en exécution, écriture et lecture respectivement. Les autres bits sont essentiellement utilisés pour faciliter la mise à jour des tables de pages et le parcours de la table des pages (*page table walk* en anglais). Ce parcours est le nom de l'exécution du processus de traduction, détaillé précédemment, par le matériel, la MMU.

17.2.4 Translation lookaside buffer

Le parcours de la table, même accéléré par le matériel, est une opération couteuse impliquant plusieurs accès mémoire. Pour accélérer la traduction d'adresse, il est possible de mettre les anciennes traductions dans une mémoire cache dédiée, c'est le translation lookaside buffer (TLB).

De la même manière qu'il existe une hiérarchie mémoire pour la mémoire principale, une hiérarchie des TLB est présente dans les SoCs modernes. Avec des caches dédiés ou non à des

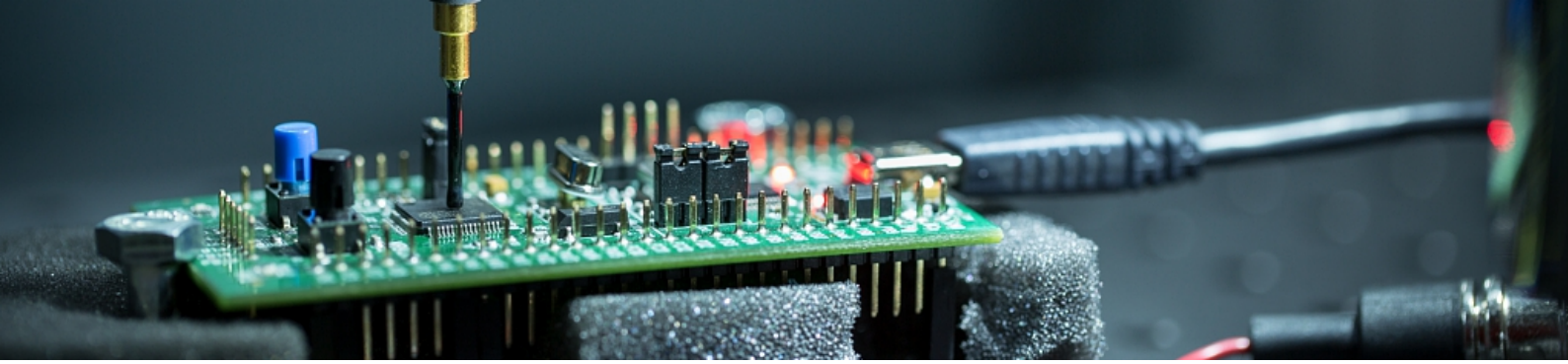
cœurs ou des types de données.

17.2.5 Address space identifier et changement de processus

Puisque la table des pages de premier niveau est défini par le registre `satp`, cf. figure 17.6, il suffit de modifier sa valeur pour pointer vers une nouvelle table et ainsi modifier la vue subjective de la mémoire par le programme en cours.

Ainsi, chaque processus possède ses propres tables des pages, et lors du basculement vers un nouveau processus le registre `satp` est modifié pour pointer vers la nouvelle table. Pour optimiser l'utilisation du TLB, il est utile d'identifier chaque espace mémoire avec un identifiant unique : c'est l'address space identifier (ASID), parfois aussi appelé le *process ID*. Ainsi le TLB peut identifier que l'espace mémoire a changé et que les entrées de traduction en cache sont maintenant obsolètes.

L'ASID est le seul élément matériel qui relie l'exécution des instructions au concept logiciel de «processus».



18. Les modèles mémoires

Les processeurs modernes sont constitués de nombreux cœurs, 8 cœurs, 16 cœurs ... jusqu'à 64 cœurs pour le grand public aujourd'hui. Encore plus pour les serveurs. Chaque cœur physique pouvant donner l'illusion de deux cœurs virtuels grâce au simultaneous multithreading (SMT).

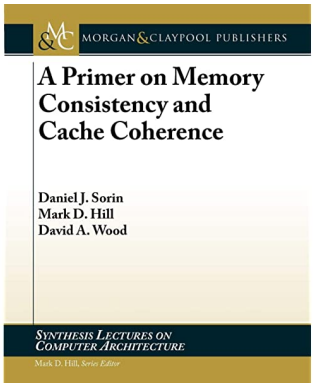
Nous pouvons donc nous poser la question de ce qui fait l'unicité d'un système. Pourquoi avons-nous l'illusion d'interagir avec un seul processeur, ou non dans le cas d'un système partagé ?

La réponse tient dans le partage mémoire. C'est parce que tous ces cœurs voient une même mémoire unique que nous pouvons dire que nous avons un seul système.

Mais est-ce vraiment le cas ? Est-ce que tous ces processeurs voient vraiment la même mémoire ? La réponse en trois mots : c'est compliqué.

Dans cette section, nous verrons les modèles mémoires, c'est-à-dire les règles régissant la bonne utilisation de la mémoire, notamment lorsqu'il y a plusieurs cœurs.

Suggestions de lecture



A Primer on Memory Consistency and Cache Coherence [36]

Par *Daniel J. Sorin, Mark D. Hill, David A. Wood*

ISBN : 1608455645

Ce chapitre a été principalement conçu à partir de ce livre.

18.1 La consistance mémoire

La consistance mémoire (ou modèle de consistance mémoire ou modèle mémoire) est l'ensemble de règles définissant les opérations mémoires correctes à un instant donné en fonction de l'état du système. Les règles concernant la mémoire partagée sont subtiles et plusieurs possibilités existent, ayant des conséquences sur les performances du système et sur ce qu'il est correct de faire.

Les modèles de consistance mémoire définissent ce qui est correct en termes d'opérations `LOAD` et `STORE` (cf définition 18.1) sans faire référence à l'implémentation de la mémoire (sa hiérarchie, etc.). Ainsi le modèle mémoire fait partie de l'architecture et doit impérativement être définie par le jeu d'instructions. Pour un processeur à un seul cœur, il y a peu d'ambigüité : l'ordre des `LOAD` et `STORE` doit être respecté lorsqu'il y a des dépendances entre eux. Pour un processeur à plusieurs cœurs, que se passe-t-il si plusieurs cœurs écrivent simultanément à une même adresse. Ce qui est possible ou impossible est décrit par le modèle de consistance mémoire.

Exercice 18.1 - *Les modèles mémoires*

Observons l'exécution des deux programmes suivants sur deux cœurs au sein d'un même processeur. Les variables ayant un scope global. *Toutes les variables sont initialisées à zéro.*

Cœur 1 :

```
C1_A: a = 1;
C1_B: x1 = b;
```

Cœur 2 :

```
C2_A: b = 1;
C2_B: x2 = a;
```

Quels sont les couples de valeurs $(x1, x2)$ autorisés, à la fin de l'exécution de ce programme ?

Intuitivement, du fait des accès concurrents (*race condition* en anglais), nous nous doutons que les valeurs suivantes sont légales :

— $(0,1)$ pour la séquence d'exécution `C1_A`, `C1_B`, `C2_A`, `C2_B`.
— $(1,0)$ pour `C2_A`, `C2_B`, `C1_A`, `C1_B`.
— $(1,1)$ pour `C1_A`, `C2_A`, `C1_B`, `C2_B`.

De manière plus étonnante, sur tous les processeurs modernes, le couple $(0,0)$ est également possible.

18.1.1 Consistance séquentielle

La consistance séquentielle (sequential consistency (SC)) correspond au modèle mémoire intuitif, qui n'est plus utilisé sur les processeurs modernes.

Soit $<_p$ la relation d'ordre dans un programme. $i_1 <_p i_2$ si l'instruction i_1 est placée avant i_2 dans le programme par rapport à l'ordre d'exécution des instructions.

Soit $<_m$ la relation d'ordre pour la mémoire partagée entre tous les cœurs. $i_1 <_m i_2$ si l'accès mémoire i_1 a lieu avant i_2 du point de vue de la mémoire.

Le modèle SC spécifie que $\forall i_1, i_2$, alors $i_1 <_p i_2 \Rightarrow i_1 <_m i_2$. Autrement formulé, le modèle SC impose que l'ordre des accès mémoires respecte l'ordre de ces mêmes accès dans les programmes. Dans notre 18.1, SC n'autorise que les 3 couples de valeur $(0,1)$, $(1,0)$ et $(1,1)$.

La consistance séquentielle n'est pas le modèle mémoire des processeurs modernes

Sur la plupart des processeurs modernes, les instructions `STORE` sont sauvegardées dans un buffer dédié, le **write buffer**, avant de les transmettre au cache de premier niveau. En effet, pour exécuter ce `STORE`, il faut que la ligne de cache correspondante soit présente dans le cache

L1. Toutefois, pour ne pas ralentir le pipeline, si un `LOAD` se présente qui tente d'accéder à une adresse en attente d'écriture par une instruction dans le *write buffer*, la donnée sera extraite du buffer et non du cache. Cela permet de garder une consistance par rapport à l'ordre du programme.

Toutefois ce comportement rend possible d'obtenir le couple $(0,0)$ dans l'18.1. Garantir SC serait extrêmement couteux en termes de performances.

18.1.2 Total store order (TSO)

Total store order (TSO) est le modèle mémoire des processeurs x86. Les contraintes sur les ordres à respecter dépendent maintenant en détail des instructions.

Pour le modèle TSO, pour n'importe quelles adresses `a` et `b` :

— `LOAD a` $<_p$ `LOAD b` $\Rightarrow$ `LOAD a` $<_m$ `LOAD b`.
— `LOAD a` $<_p$ `STORE b` $\Rightarrow$ `LOAD a` $<_m$ `STORE b`.
— `STORE a` $<_p$ `STORE b` $\Rightarrow$ `STORE a` $<_m$ `STORE b`.

Contrairement au modèle SC, il n'y a **pas** l'implication suivante :

`STORE a` $<_p$ `LOAD b` $\Rightarrow$ `STORE a` $<_m$ `LOAD b`

Nous voyons que ce cas correspond à l'usage des *write buffers*. Ce modèle mémoire est donc intrinsèquement plus performant, car il n'impose pas de resynchronisation entre tous les cœurs lorsqu'un `LOAD` dépend d'un `STORE`. Mais il peut mener à des résultats non intuitifs, comme le couple $(0,0)$ dans notre exemple.

Il faut noter que ce résultat non intuitif n'est pas si gênant. En effet, l'exemple nous montre un cas limite ou le résultat est non déterministe. En général, le logiciel n'aime pas les résultats non déterministes. C'est le rôle du logiciel de s'assurer, via l'utilisation de primitives dédiées (opérations atomiques, mutex, ...) que l'on obtient toujours le résultat correct.

18.1.3 Modèles mémoires faibles

Les modèles mémoires faibles (*relaxed memory models ou weak memory models* en anglais) sont plus permissifs pour encore plus de performances. ARM ou RISC-V spécifient de tels modèles. Il existe de nombreuses possibilités, mais nous pouvons donner un exemple de modèle faible, tiré de [36].

Pour toute adresse `a` :

— `LOAD a` $<_p$ `LOAD a` $\Rightarrow$ `LOAD a` $<_m$ `LOAD a`.
— `LOAD a` $<_p$ `STORE a` $\Rightarrow$ `LOAD a` $<_m$ `STORE a`.
— `STORE a` $<_p$ `STORE a` $\Rightarrow$ `STORE a` $<_m$ `STORE a`.

Le fait que la contrainte d'ordre ne soit maintenue que pour des instructions utilisant une même adresse, permets de réordonner dynamiquement les `LOAD` et `STORE` qui utilisent des adresses différentes. Quel intérêt y-a-t-il à maintenir la contrainte d'ordre entre, par exemple, `STORE 0x1000` et `STORE 0x2000` ? Le réordonnancement possible des instructions permet la possibilité de meilleures performances.

18.2 Les instructions supplémentaires pour les modèles mémoires

Les modèles mémoires TSO et faibles ne contraignant pas l'ordre de tous les cas de figure, il est parfois nécessaire de rajouter ces contraintes dans des cas particuliers, par exemple pour synchroniser les cœurs entre eux.

18.2.1 Les fences

Les instructions `FENCE` permettent au développeur (ou au compilateur) de réintroduire manuellement des contraintes d'ordres. Elle rajoute les contraintes suivantes au modèle mémoire. Pour toute adresse `a` :

— `LOAD a` $<_p$ `FENCE` $\Rightarrow$ `LOAD a` $<_m$ `FENCE`.
— `STORE a` $<_p$ `FENCE` $\Rightarrow$ `STORE a` $<_m$ `FENCE`.
— `FENCE` $<_p$ `LOAD a` $\Rightarrow$ `FENCE` $<_m$ `LOAD a`.
— `FENCE` $<_p$ `STORE a` $\Rightarrow$ `FENCE` $<_m$ `STORE a`.

Ainsi l'utilisation de l'instruction `FENCE` permet d'obtenir, ponctuellement, un modèle mémoire plus strict pour les cas le nécessitant.

18.2.2 Les instructions atomiques

Les instructions atomiques sont nécessaires dans un système à plusieurs cœurs, utilisant un modèle mémoire faible, lorsque l'on veut les synchroniser entre eux. Il existe plusieurs primitives atomiques, nous nous concentrerons sur l'extension « A » de RISC-V.

18.2.2.1 Load-Reserved / Store-Conditional

La combinaison des instructions `LR` (Load-Reserved) et `SC` (Store-Conditional) permettent de réaliser des primitives de type LOAD-MODIFY-WRITE en garantissant qu'aucune écriture n'a lieu sur l'adresse cible durant la primitive, par un autre cœur. `LR x1, 0(x2)` agit comme un `LOAD` en lisant la valeur en mémoire à l'adresse `x2` et en écrivant cette valeur dans `x1`. Toutefois, `LR` va en plus enregistrer l'adresse `0(x2)` dans le **reservation set**, un ensemble d'adresses « réservées ». De même `SC x1, x2, 0(x3)` a un comportement similaire à un `STORE x2, 0(x3)`, écrivant la valeur lue à l'adresse définie par `x3` dans le registre `x2`. Toutefois, cette opération ne sera réalisée que si l'adresse cible est présente dans le *reservation set*. En cas de succès de l'opération, un 0 est écrit dans `x1` et le `STORE` est réalisé. Sinon, un code d'erreur est écrit dans `x1` et le `STORE` n'est pas réalisé. Le *reservation set* est global à l'espace mémoire.

Exemple 18.1

Pour comprendre toutes les subtilités possibles, et elles sont nombreuses, vous êtes invités à lire la spécification de l'extension A de RISC-V.

18.2.2.2 Atomic Memory Operations

Les instructions atomic memory operations (AMO) permettent d'appliquer une opération atomique simple en visant une adresse mémoire. Par exemple `AMOXOR rd, 0(rs1), rs2` :

1. Lit la valeur à l'adresse définie par `rs1` et la met dans `rd`.
2. Applique l'opération XOR entre la valeur chargée et `rs2`.
3. Réécris le résultat du XOR à l'adresse définie par `rs1`.

Le caractère atomique de l'opération garantie qu'il n'y a pas de modification de la mémoire par un autre cœur entre le `LOAD` et le `STORE`. La spécification RISC-V définit les opérations atomiques suivantes : SWAP, ADD, AND, OR, XOR, MAX et MIN.

18.2.3 Les modèles mémoires des langages

Nous n'avons aucune envie de gérer à la main tous les cas limites liés aux modèles mémoires. Cela tombe bien, nous n'écrivons que très rarement le code machine directement. Ainsi les langages de programmation définissent souvent leur propre modèle mémoire. Charge au compilateur d'en garantir la sémantique sur la machine cible, par exemple en insérant des `FENCE` quand nécessaire. Cela permet également la portabilité du langage : nous apprécions que le comportement d'un même programme soit le même sur plusieurs processeurs.

Exemple 18.2 - *Modèle mémoire des langages*

Exemple tiré de https://research.swtch.com/plmm :

```
// Thread 1            // Thread 2
x = 1;                 while(done == 0) { /* loop */ }
done = 1;              print(x);
```

Le comportement d'un tel programme dépend du modèle mémoire du langage. Il est possible que la variable done corresponde à un registre et non à une valeur en mémoire. Sans accès mémoire, il n'y a donc pas de synchronisation possible entre les deux threads. Il est possible que $x = 0$ et $done = 1$ car le compilateur peut appliquer un modèle mémoire faible et réordonnancer les lectures.

Les modèles mémoires des langages ont des notions similaires aux modèles mémoires microarchitecturaux mais s'appliquent à la compilation et non à l'exécution.

18.3 Protocoles de cohérence de caches

Maintenant que nous avons vu les modèles de consistance mémoire, se pose la question de leur implémentation. La cohérence de caches est là pour assurer la bonne diffusion des informations pour faire respecter le modèle mémoire du système.

La cohérence est responsable de la mise en place des garanties d'ordres entre instructions des modèles de consistance mémoire. La cohérence de caches fait partie de la microarchitecture là ou les modèles mémoires font partie de l'architecture.

Ces protocoles de cohérence se reposent généralement sur un invariant appelé Single-writer, multiple-readers (SWMR). Pour une adresse mémoire donnée, un seul processeur peut écrire ou lire à cette adresse, ou plusieurs processeurs peuvent lire, mais pas écrire.

Contrôleurs de cache La cohérence nécessite la synchronisation de nœuds dans un réseau. Ces nœuds sont les contrôleurs de cache, responsable de l'état de la mémoire cache associée. Les messages dans ce réseau dédié sont les requêtes de cohérence, aussi appelées transactions mémoires. Ainsi, le contrôleur définit l'état de la mémoire par bloc de données, usuellement par ligne de cache. Il s'agit d'un système distribué, il n'est pas possible de connaitre exactement les valeurs de l'ensemble de la mémoire sans interroger tous les contrôleurs.

Les états de la mémoires Dans un système avec un seul cœur, pas de DMA, etc., mais avec un cache pour des accès mémoires rapides, chaque ligne de cache ne nécessite qu'un seul bit d'état. La ligne est **valide** lorsqu'elle représente l'état courant de cette portion de la mémoire ou **invalide** lorsque les données ne représentent rien. Par exemple, au démarrage, avant le premier accès mémoire, l'ensemble des lignes de cache sont invalides.

Toutefois, dans un cache *write-back*, le contrôleur a également besoin de savoir si la ligne a été modifiée et nécessitera une mise à jour de la mémoire de niveau supérieur. On parle de ligne **dirty** dans ce cas, de ligne **clean** dans le cas contraire. Les choses se compliquent lorsqu'il y a plusieurs cœurs ou simplement plusieurs caches. Nous avons parfois besoin de savoir si une ligne est **exclusive**, c'est-à-dire que c'est le seul bloc mémoire à cette adresse dans un cache dans le système. Enfin, il faut décider quel contrôleur est apte à répondre à une requête d'état de cohérence, pour éviter un conflit ou 2 caches prétendent avoir une exclusivité sur la même ligne de cache. Le contrôleur qui décide de l'état d'une ligne est le propriétaire (**owner**) de celle-ci.

On utilise en général un système inspiré par les 5 états MOESI introduits dans [38], en général les états MSI sont suffisants.

— **M**odified : le bloc est valid, exclusive, owned et potentiellement dirty. Dans cet état, il

est possible de lire et d'écrire dans le bloc.
- **O**wned : Le bloc est valid, owned, potentiellement dirty, mais pas exclusif. Le cache a une copie du bloc en mode lecture seule, mais doit répondre aux requêtes de cohérence.
- **E**xclusive : Le bloc est valid, exclusive et clean. Il s'agit d'un état rarement utilisé où seule la lecture est possible bien que l'accès soit exclusif. Dans la plupart des cas, cet état implique directement la propriété (owned) et se confond donc avec l'état owned.
- **S**hared : le bloc est valid, non-exclusive, clean, pas owned. Le cache a une copie du bloc en mode lecture seule.
- **I**nvalid : le bloc est invalid. Il n'est pas possible ni d'écrire ni de lire dans ce bloc. C'est l'état initial des lignes de cache au démarrage.

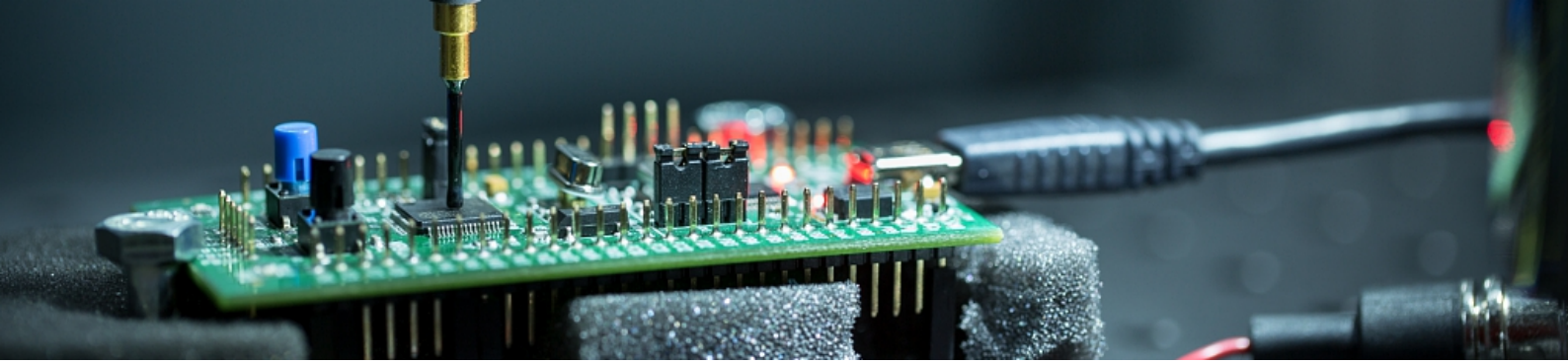
Transactions ou requêtes Pour établir les états des différents blocs, les contrôleurs doivent communiquer entre eux via un protocole dédié. On appelle **transactions** mémoires ou requêtes un message de ce protocole, celui-ci est généralement conçu spécifiquement pour une architecture donnée.

Voici quelques exemples de telles transactions :
- *GetShared* pour obtenir un bloc dans l'état *shared*.
- *GetModified* pour un bloc dans l'état *modified*.
- *Upgrade* pour élever un bloc en lecture seule à un bloc en lecture-écriture.
- *PutModified* pour évincer un bloc dans l'état *modified*.
- ...

Topologie du réseau Enfin le protocole de cohérence est également caractérisé par sa topologie. Il y en a deux principales.

Les **snooping coherence protocols** communiquent en broadcastant leurs messages. C'est-à-dire que les transactions sont envoyées à tous les contrôleurs en même temps. La simplicité de ce protocole est contrebalancée par un faible passage à l'échelle. Il est plutôt utilisé dans des petits systèmes.

Les **directory coherence protocols** associent un contrôleur hôte pour chaque bloc, qui peut être différent du propriétaire (owner). Chaque contrôleur doit donc maintenir un répertoire des états pour chaque bloc dans la hiérarchie mémoire. Les transactions ne sont transmises qu'au contrôleur hôte, évitant ainsi d'encombrer le bus de données.



19. La sémantique des pointeurs

19.1 Qu'est-ce qu'un pointeur ?

Exercice 19.1 - *Qu'est ce qu'un pointeur ?*

Proposer une définition de ce qu'est un pointeur. Considérer les cas suivant (en C++) :

1. ```
   uint32_t* addi = (uint32_t*) 0x12345678;
   void* addv = (void*) addi;
   ```

   En quoi diffèrent `addi` et `addv` ?

2. ```
   void f1(uint32_t* input, uint32_t* output) {
       if (*input > 10) {
           *output = 1;
       }
       if (*input > 5) {
           *output *= 2;
       }
   }
   void f2(uint32_t* input, uint32_t* output) {
       uint32_t ival = *input;
       if (ival > 10) {
           *output = 1;
       }
       if (ival > 5) {
           *output *= 2;
       }
   }
   ```

   `f1` et `f2` sont-elles équivalentes ? Pourquoi ? [a]

3. ```
   int test() {
       auto x = new int[8];
       auto y = new int[8];
       y[0] = 42;
       auto x_ptr = x+8; // one past the end
       if (x_ptr == &y[0])
         *x_ptr = 23;
   ```
   ```

```
        return y[0];
    }
```

Quelle valeur retourne la fonction `test` ? [b]

a. Exemple tiré de https://faultlore.com/blah/fix-rust-pointers/
b. Exemple tiré de https://www.ralfj.de/blog/2018/07/24/pointers-and-bytes.html

Définition 19.1

Un pointeur est un concept de langage de programmation permettant de manipuler des objets en mémoire. Le pointeur est en général défini par une adresse mémoire et des métadonnées le liant aux données vers lesquelles il pointe.
Le processeur ne manipule pas de pointeurs, il utilise le concept d'adresses virtuelles ou physiques (cf chapitre 17), voir de *capabilities* dans le cas de CHERI (cf sous-section 20.2.6).

Le pointeur n'est pas un concept qui existe dans tous les langages de programmation : il est surtout présent en C, C++ et autres langages systèmes (tel que le rust, etc.). Comme nous l'avons vu dans l'exercice 19.1, ce concept est en réalité très subtile, mais permet de contrôler finement l'accès du processeur à l'espace adressable.

Comme le pointeur est un concept du langage de programmation, c'est le compilateur associé qui se charge de vérifier sa bonne utilisation, dans les limites imposées par le langage. Mais attention, les standards ne sont pas toujours respectés ou interprétés de la même façon par tous les compilateurs d'un même langage. Plus vous jouez avec les subtilités, plus vous avez de chances que cela ne marche pas comme prévu.

19.2 Pointeurs synonymes (aka *'Aliasing'*)

L'*aliasing* est la possibilité pour deux pointeurs de se recouper en mémoire. C'est-à-dire que les structures de données pointées peuvent se chevaucher ou non.

Les pointeurs pouvant parfois n'être définis que dynamiquement, c'est-à-dire lors de l'exécution du programme, prouver que deux pointeurs ne sont pas synonymes implique de restreindre ce que le programme peut faire avec eux. Ou de déclarer certains comportements *undefined behaviour*.

```
int* i = /* valeur définie à l'exécution uniquement */
int tableau[10];

// fonction très loin
void test() {
    int autre_tableau[10];
}
```

Dans ce code, le pointeur `i` n'étant défini que dynamiquement, il peut chevaucher n'importe quel tableau (`tableau`, `autre_tableau`) et en fait il peut chevaucher toutes les structures de données du programme, pour autant que le compilateur le sache.

En C, depuis C-99, il est possible d'expliciter qu'un pointeur n'a pas de synonymes à l'aide du mot-clé `restrict` : `int* restrict p`.Utiliser `restrict` est une garantie donnée par le programmeur au compilateur, qu'il n'existe pas de pointeur synonyme de `p` : aucun autre pointeur dans le programme ne chevauche la structure de donnée pointée par `p`.

Exercice 19.2 - *restrict*

Si on reprend les fonctions de l'exercice 19.1.

```
void f1(uint32_t* input, uint32_t* output) {
    if (*input > 10) {
        *output = 1;
    }
    if (*input > 5) {
        *output *= 2;
    }
}
void f2(uint32_t* input, uint32_t* output) {
    uint32_t ival = *input;
    if (ival > 10) {
        *output = 1;
    }
    if (ival > 5) {
        *output *= 2;
    }
}
void f3(uint32_t* restrict input, uint32_t* restrict output) {
    if (*input > 10) {
        *output = 1;
    }
    if (*input > 5) {
        *output *= 2;
    }
}
```

`f3` est-elle équivalente à `f1` ou à `f2` ? Pourquoi ?

19.3 Provenance des pointeurs

Une autre subtilité des pointeurs utilisée pour les optimisations du compilateur est la provenance. Un pointeur défini comme pointant sur une structure de donnée est différent, pour le compilateur, d'un autre pointeur défini sur un autre objet. Cela mène à l'exemple paradoxal n°3 présenté dans l'exercice 19.1.

Les métadonnées associées à un pointeur ne sont pas seulement le type du pointeur, qui définit la structure obtenue lors du déréférencement, et l'aliasing. Elles attachent également le pointeur à un «objet» associé, c'est ce que l'on appelle la **provenance** du pointeur. [1]

Sans trop rentrer dans les détails, étudier la provenance du pointeur permet de déterminer plus facilement s'il est synonyme d'un autre pointeur : ils ne sont pas synonymes s'ils n'ont pas la même provenance.

Une conséquence du concept de provenance est que la conversion d'un pointeur vers un entier, puis à nouveau vers un pointeur détruit cette information. Cet aller-retour ne doit donc pas être optimisé par le compilateur.

Les deux programmes sur la figure 19.1 ne sont pas équivalents.

Il est facile de mal utiliser des pointeurs : leur sémantique étant subtile, une opération peut marcher sur un compilateur et pas sur un autre. Pour éviter leur mauvaise utilisation,

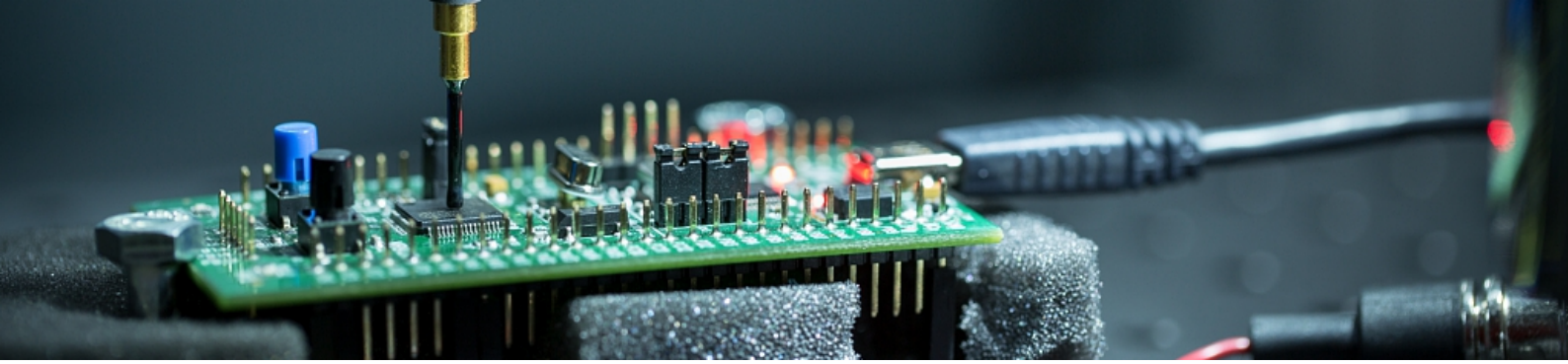
1. Pour un approfondissement du sujet : https://www.ralfj.de/blog/2020/12/14/provenance.html.

```c
char c[1] = {0};
return c;
```

```c
char c[1] = {0};
int ptr_i = (int)c;
return (char*)ptr_i;
```

FIGURE 19.1 – Le cast vers un entier détruit l'information de provenance.

les langages modernes soit les suppriment en les remplaçant par le concept de référence à un objet, soit essayent de restreindre les opérations légales à une sémantique des pointeurs plus restreinte. C'est le champ de la sécurité mémoire (*memory safety*).



20. Sécurité mémoire (aka *'memory safety'*)

Selon une étude de Matt Miller à Microsoft en 2019, 70% des vulnérabilités observées sont des problèmes de sécurité de la mémoire. En d'autres mots, 70% des vulnérabilités sont liées à une mauvaise utilisation de la mémoire, en particulier à des erreurs dans la gestion des pointeurs.

20.1 Les vulnérabilités

Ces vulnérabilités de sécurité mémoire sont multiples, et nous allons tenter d'en comprendre le fonctionnement dans cette section. Pour cette section, nous allons explorer quelques-unes des vulnérabilités de la liste Commen Weakness Enumeration. N'hésitez pas à explorer les autres vulnérabilités, elles sont toujours intéressantes !

20.1.1 Heap out-of-bounds

Cette classe de vulnérabilité désigne les accès mémoires (en lecture ou écriture) visant un objet sur le tas (*heap*), mais à une adresse ne correspondant pas à cet objet.

Exemple 20.1 - *Heap out-of-bounds read*

Voici un exemple de *heap out-of-bounds read*, tiré de la définition CWE-125 de Mitre.

```
int getValueFromArray(int *array, int len, int index) {
    int value;

    // check that the array index is less than the maximum
    // length of the array
    if (index < len) {

        // get the value at the specified index of the array
        value = array[index];
    }
    // if array index is invalid return value indicating error
    else {
        value = -1;
    }
```

```
    return value;
}
```

Dans cet exemple, nous ne vérifions pas que l'index est une valeur positive. Un index négatif permet des accès mémoires en dehors du tableau.

Exemple 20.2 - *Heap out-of-bounds write*

La définition CWE-787 concerne les accès en écriture.

```
int id_sequence[3];

id_sequence[0] = 123;
id_sequence[1] = 234;
id_sequence[2] = 345;
id_sequence[3] = 456;
```

20.1.2 Use-after-free

Cette vulnérabilité désigne l'utilisation d'une structure de donnée **après** que la fonction `free` ait été appelée sur cette structure.

Exemple 20.3 - *Use-after-free*

Il s'agit de la vulnérabilité CWE-416, dont est tiré l'exemple ci-dessous.

```
#define BUFSIZER1 512
#define BUFSIZER2 ((BUFSIZER1/2) - 8)
int main(int argc, char **argv) {
    char* buf1R1 = (char*) malloc(BUFSIZER1);
    char* buf2R1 = (char*) malloc(BUFSIZER1);;
    free(buf2R1);
    char* buf2R2 = (char *) malloc(BUFSIZER2);
    char* buf3R2 = (char *) malloc(BUFSIZER2);
    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```

Puisque le `strncpy` copie dans un tableau déjà libéré, d'autres tableaux ont pris sa place. Selon toute vraisemblance ici, ce sont les tableaux `buf2R2` et `buf3R2` qui sont écrit à la place.

20.1.3 Double-free

Que se passe-t-il si l'on appelle `free` deux fois sur le même pointeur ? La réponse exacte dépend de l'allocateur utilisé, mais en général ce dernier entre dans un état corrompu : les allocations ou les libérations suivantes risquent d'être incorrectes. L'imprévisibilité du résultat, parfois aucune erreur n'apparait, rend cette vulnérabilité particulièrement vicieuse.

Exemple 20.4 - *Double-free*

```
char* ptr = (char*)malloc (SIZE);
//...
if (abrt) {
    free(ptr);
}
//...
free(ptr);
```

20.1.4 Uninitialized use

Dans le langage C, un pointeur peut être défini, mais non initialisé. C'est-à-dire que l'adresse associée contient par défaut les dernières données écrites à cette adresse.

Exemple 20.5 - *Uninitialized use*

```
char *test_string;
if (i != err_val)
{
    test_string = "Hello World!";
}
printf("%s", test_string);
```

Cette vulnérabilité peut être éliminée par le langage de programmation, par exemple en imposant une valeur par défaut lors de la création d'une structure de données.

20.1.5 Stack corruption

La corruption de la pile apparait lorsque la sémantique de ladite pile n'est pas respectée. L'exemple classique est l'écriture out-of-bounds sur la pile, plus communément appelé **stack overflow**.

Exemple 20.6 - *Stack overflow*

Désigné sous le nom de CWE-121 dans la base de données CWE.

```
#define BUFSIZE 256
int main(int argc, char **argv) {
    char buf[BUFSIZE];
    strcpy(buf, argv[1]);
}
```

Ce code minimal essaie de copier l'argument du programme dans un buffer. Que se passe-t-il si l'argument est plus grand que la taille du buffer ?

20.1.6 Type confusion

On appelle type confusion (CWE-704), les vulnérabilités liées à la mauvaise conversion entre structures de données de types différents. Lorsqu'un cast est erroné, l'utilisation de l'objet peut mener à des bugs, ou à un exploit.

Exemple 20.7 - *Type confusion*

```
typedef struct structA {
    int something;
    int a;
} structA;

typedef struct structB {
    int a;
} structB;

int do_something(structA* arg) {
    arg.a = 2;
}

structB* objB = //...
structA* objA = (structA*) objB;

do_something(objA);
```

Dans cet exemple la fonction `do_something` modifie le deuxième champ de la structure `structA`, soit à un offset 4 du pointeur de la structure. Hors le pointeur `objB` est censé pointer vers une structure de taille 4 uniquement, nous allons donc écrire sur un autre objet.

20.2 Assurer la sécurité mémoire

Nous venons de voir un extrait de vulnérabilités mémoires possibles, mais il y en a d'autres. Sommes-nous condamnés à l'émergence de failles de sécurité toujours plus nombreuses ?

Certaines personnes pointent du doigt la médiocrité des développeurs, incapable de gérer correctement leur mémoire. Toutefois, ce constat implique que seule une poignée de développeurs sur la planète peut se voir confier la conception d'un programme. Et d'un autre côté, nous venons de voir dans cette partie que la gestion de la mémoire est en fait un édifice complexe impliquant la microarchitecture, le compilateur, le langage. La « bonne » gestion de la mémoire par le développeur n'est pas un ensemble de règles absolues, mais plutôt une construction abstraite du langage devant s'adapter à chaque système cible.

Alors que la plupart des logiciels systèmes sont développés avec le langage C, très susceptible aux erreurs, que peut-on faire ?

20.2.1 Fat pointers

Les *fat pointers* font le constat qu'un pointeur simple (*thin*), ne contenant qu'une adresse, est souvent trop limité. Nous aimerions ajouter des métadonnées dans ce pointeur, pour prévenir une mauvaise utilisation ou automatiser certaines bonnes pratiques mémoires.

Sized structs Une première possibilité intéressante est de lier indissociablement une structure de donnée, un tableau par exemple, et sa taille. Il n'y a alors plus besoin de spécifier sa taille dans les fonctions appelées, cette dernière est chargée de s'assurer qu'il n'y a pas d'erreur mémoire.

```
typedef struct SizedInt32Array {
    int32_t* array;
```

```
    size_t   array_size;
} SizedInt32Array;
```

Reference counting pointers Il est également possible d'automatiser la libération d'une structure de données avec les pointeurs à compteur de références. L'idée est que chaque pointeur vers une même structure, en plus de l'adresse de ladite structure, partage un compteur du nombre de références existantes. Ainsi, si le pointeur est dupliqué le compteur est incrémenté. De même, lorsque le pointeur devient inutile il est décrémenté, par exemple à la sortie du scope le définissant. Lorsque le compteur atteint 0, il n'existe plus de référence vers cette structure, elle peut alors être libérée.

20.2.2 Garbage collector

Le ramasse-miette (*garbage collector*) est une technique pour automatiser la gestion de la mémoire. À une fréquence déterminée, le programme est mis en pause et le ramasse-miette inventorie les objets qui peuvent être libérés avant de le faire. Il existe plusieurs techniques, certaines utilisent des compteurs de référence, cachant ainsi l'utilisation de fat pointers.

Les ramasse-miettes sont extrêmement pratiques pour le développeur puisqu'il lui enlève la responsabilité de la gestion de la mémoire. Le succès de Java repose en particulier sur son ramasse-miette intégré dans le langage lui-même. À noter que le ramasse-miette peut être intégré via une bibliothèque dans la plupart des langages, même le C.

Toutefois ils dégradent les performances de manière significative : un programme Java ne pourra jamais avoir la même vitesse qu'un même programme C optimisé.

20.2.3 Canaries

Les canaries sont une technique pour lutter contre les stack overflow, dont le nom est inspiré des oiseaux que l'on plaçait dans les mines pour détecter une atmosphère irrespirable. L'idée est de placer des valeurs spéciales, les canaries, dans la pile, et de vérifier leur intégrité avant d'utiliser la pile, pour un retour de fonction par exemple.

```
push ra
push canary // random value

// ...

pop canary_candidate
bneq canary, canary_candidate, UNDER_ATTACK_ROUTINE
pop ra
```

Idéalement, le placement et la vérification des canaries sont transparents pour le développeur, car insérés directement par le compilateur.

20.2.4 ASLR

De nombreux exploits nécessitent que l'attaquant connaisse l'adresse des données qu'il veut exfiltrer ou modifier. Une façon relativement simple de lui compliquer la tâche est de randomiser l'espace mémoire à l'aide de l'address space layout randomization (ASLR).

Lors du lancement d'un programme, le runtime choisit des adresses aléatoires à la granularité la plus fine possible : l'adresse de chargement du programme lui-même, l'adresse de chargement des bibliothèques dynamiques, et parfois les adresses de parties du programme si la relocation est possible. Le noyau lui-même peut randomisé son placement, on parle alors de kernel address space layout randomization (KASLR). Ce mécanisme repose sur la mémoire virtuelle, la randomisation n'a pas besoin d'influencer le placement physique des données.

Cette protection n'est pas parfaite et de nombreuses techniques permettent de tenter de la surmonter. Toutefois le cout de cette protection est faible et rend le travail plus difficile pour l'attaquant.

20.2.5 Pointer integrity / pointer authentication

Il existe des mécanismes pour garantir l'intégrité des pointeurs. Prenons l'exemple de l'extension *pointer authentication* (PA) du jeu d'instructions ARMv8.3-A. Cette extension rajoute des instructions pour créer et manipuler des pointer authentication codes (PACs).

L'idée est d'utiliser les bits d'adresse virtuelle inutilisés pour sauvegarder un tag d'intégrité lié à l'adresse associée. De manière simplifiée, une première instruction `pac` crée un pointeur authentifié à partir d'une adresse. Cette instruction utilise un message authentication code (MAC) basé sur une clé choisie parmi plusieurs (par exemple `pacia` utilise la clé **i**nstruction **A**). Le pointeur authentifié n'est pas utilisable directement, il faut d'abord le vérifier à l'aide de l'instruction `aut`. Si cette vérification est correcte, on obtient la valeur d'adresse attendue. Sinon, l'adresse obtenue déclenche une exception *memory translation fault*.

20.2.6 CHERI / Morello

La solution proposée par l'extension PA d'ARMv8.3-A est intéressante, mais susceptible aux attaques réutilisant les pointeurs. L'université de Cambridge travaille depuis 2010, avant l'extension d'ARM donc, sur une solution plus robuste appelée CHERI. Arm a repris et adapté cette extension sous le nom de Morello.

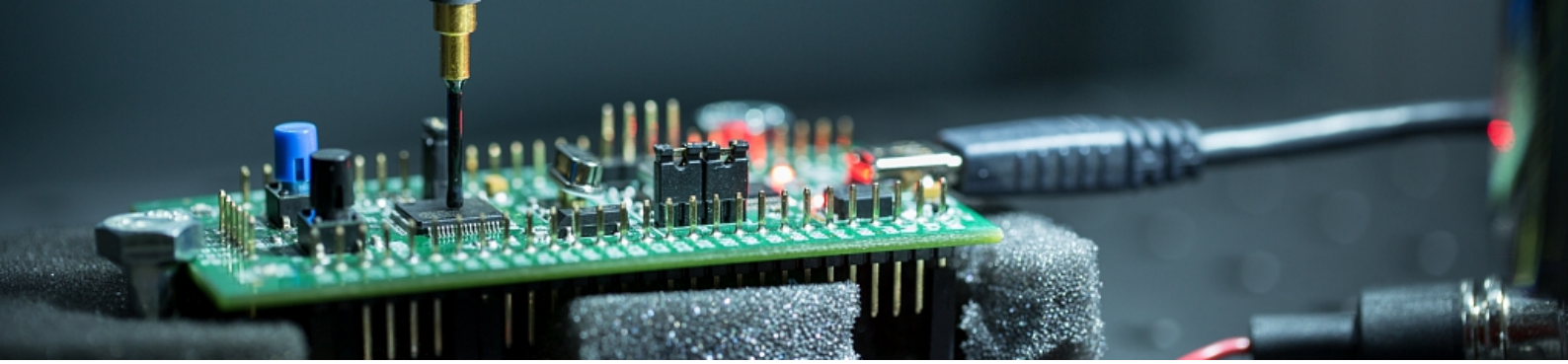
Le concept est que tout pointeur est un *fat pointer* spécifié par l'ISA, donc géré directement par le matériel. Ainsi, même pour des adresses de 64 bits, un pointeur fait 128 bits + 1 bit de validité. Les métadonnées associées contiennent les *capabilities* du pointeur, ces droits d'accès, et un interval d'adresses de validité. Ainsi si l'adresse du pointeur est en dehors de la plage de validité, il n'est pas possible d'utiliser ce pointeur.

L'utilisation d'instructions dédiée est obligatoire pour sceller un pointeur, c'est-à-dire l'authentifier, en conjonction avec le bit de validité pour empêcher un attaquant de modifier un pointeur scellé.

En résumé, CHERI introduit le concept matériel de pointeur, à l'opposé des ISAs classiques qui n'ont que la notion d'adresse. Le découplage de la taille d'un pointeur et la taille des registres n'est malheureusement pas supporté par la plupart des langages. Les logiciels doivent donc être partiellement réécrits pour tirer parti de CHERI.

De manière très intéressante, l'exécution sur une microarchitecture supportant CHERI fait apparaitre de nombreuses vulnérabilités du à la mauvaise utilisation des pointeurs.

V Concepts avancés de C



21. Sémantiques spéciales en C

21.1 `const`

Le mot-clé `const` est utilisé pour déclarer une variable comme étant constante, c'est-à-dire qu'elle ne peut pas être modifiée après sa déclaration lors de l'exécution du programme. Il peut être utilisé pour améliorer la sécurité du code en empêchant des modifications accidentelles de la variable. Le compilateur peut en effet détecter une transgression de cette sémantique.

```
const int a = 5;
// a = 6;  // Ceci génèrera une erreur de compilation
```

`const` fait parti du type de la variable, il est donc possible de déclarer des pointeurs vers des variables constantes ou des constantes pointeurs vers des variables non constantes.

```
int const * x; // Pointeur vers une variable constante
int * const x;   // Pointeur constant vers une variable non constante
int const * const x; // Pointeur constant vers une variable constante
```

La sémantique de `const` est subtile : elle ne garantit pas que la valeur de la variable ne changera pas, mais seulement que la variable ne peut pas être modifiée par le biais de son nom. Autrement dit, le programme ne peut réécrire la valeur de la variable, mais il peut toujours la modifier par d'autres moyens.

Un cas d'usage courant dans les systèmes embarqués est de qualifiés les registres read-only avec `const volatile`. En effet, ce registre ne peut pas être modifié par le programme, mais peut être modifié par le matériel. Et chaque lecture du registre peut potentiellement donner une valeur différente (cf section 21.2).

21.2 `volatile`

Le mot-clé `volatile` indique au compilateur que la variable associée n'est pas idempotente, c'est-à-dire que deux accès successifs peuvent retourner deux valeurs différentes, sans que le programme ne l'ait modifié. Il implique de ne pas optimiser l'accès à une variable, forçant ainsi chaque accès à lire à partir de la mémoire, ou du périphérique associé. C'est principalement utilisé dans les systèmes embarqués ou lors de l'interaction avec des registres de périphériques de matériel.

```c
volatile int b = 5;
while (b == 5) {
    // Boucle sans fin, sauf si 'b' est modifié
    // par quelque chose d'extérieur au programme

    // En l'absence de 'volatile', le compilateur considère
    // qu'il s'agit d'une boucle infinie
}
```

21.3 restrict

Le mot-clé `restrict`, introduit dans la norme C99, est utilisé dans les déclarations de pointeurs pour indiquer que le pointeur est le seul moyen d'accéder à la mémoire qu'il pointe pendant sa durée de vie (pas d'aliasing).

```c
int arr[100];
int *restrict rest_ptr = arr;
// Le compilateur peut maintenant optimiser l'accès à 'arr' via 'rest_ptr'
// sachant qu'aucun autre pointeur ne pointe vers 'arr'

void foo(int *restrict ptr1, int *restrict ptr2) {
    // Le compilateur sait que ptr1 et ptr2 ne pointent pas vers la même mémoire
}
```

21.4 register

Le mot-clé `register` suggère au compilateur d'essayer de stocker la variable dans un registre du processeur, **si possible**, pour accélérer l'accès à cette variable. Notez que cela est seulement une suggestion.

La sémantique associée à `register` est que la variable ne peut pas être pointée par un pointeur.

```c
register int c = 5;
// int * ptr = &c; // Ceci génèrera une erreur de compilation
```

21.5 extern

`extern` est utilisé pour déclarer une variable ou une fonction qui est définie dans un autre fichier source. Cela permet de dire au compilateur que la variable existe sans que la mémoire correspondante ne soit allouée dans le fichier objet. C'est à l'étape de l'édition de liens (*linking*) que le compilateur cherchera où est allouée la variable.

```c
extern int a; // Déclare que 'a' existe, mais ne l'alloue pas
```

21.6 static

Le mot-clé `static` a plusieurs utilisations en C. En général, il est utilisé pour contrôler la durée de vie et la visibilité d'une variable ou d'une fonction.

21.6.1 Variables statiques

Dans le contexte d'une variable, `static` fait que la variable conserve sa valeur entre les appels de fonction. Les variables statiques sont initialisées une seule fois et subsistent même après que la fonction ait retourné son contrôle, permettant ainsi les valeurs des variables statiques d'être conservées entre les différents appels de la fonction.

```
void func() {
    static int count = 0;
    count++;
    printf("count = %d\n", count);
}
```

Dans ce code, la variable `count` est conservée entre les appels de `func()`, et son incrément se poursuivra à chaque nouvel appel.

21.6.2 Fonctions statiques

Les fonctions déclarées avec le mot-clé `static` ont une portée limitée au fichier source dans lequel elles sont déclarées, ce qui les rend inaccessibles à partir d'autres fichiers. Cela peut être utilisé pour limiter l'accès à des fonctions internes qui ne devraient pas être exposées à l'extérieur du fichier source.

```
static int private_func(int x) {
    return x + 1;
}
```

Dans ce cas, `private_func` n'est accessible qu'à partir du fichier source actuel. Ce mot-clé est par définition incompatible avec `extern`.

21.7 `inline`

Le mot-clé `inline` est une suggestion au compilateur pour insérer le contenu d'une fonction au lieu d'un appel de cette fonction, évitant ainsi le surcout associé à un appel et sa gestion de pile. Cependant, il est important de noter que l'utilisation du mot-clé `inline` est simplement une suggestion, et le compilateur est libre de l'ignorer.

Utiliser `inline` peut potentiellement augmenter la vitesse d'exécution, car il supprime le surcout d'un appel de fonction. C'est surtout intéressant pour de petites fonctions. Cependant, il peut également augmenter la taille du code binaire résultant, car le même code est répété à chaque point d'appel.

Voici un exemple simple :

```
inline int max(int a, int b) {
    return a > b ? a : b;
}
```

Dans ce cas, l'utilisation de `inline` suggère au compilateur d'insérer le contenu de la fonction `max` directement dans le code source à chaque point où `max` est appelé, plutôt que de générer un appel de fonction.

21.8 Attributes

Les attributs en C permettent de définir des propriétés supplémentaires sur une variable ou une fonction, spécifiques au compilateur utilisé.

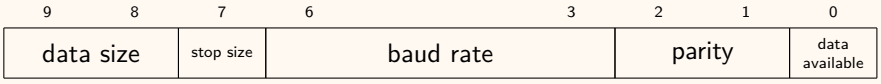
L'attribut le plus intéressant avec GCC est celui permettant de fixer l'alignement d'une variable.

```
int a __attribute__((aligned(16))) = 0;
// 'a' est aligné sur 16 octets en mémoire
```

Exercice 21.1 - *Un pilote dans l'UART*

Votre microcontrôleur définit l'interface suivante pour un périphérique UART :

— À l'adresse `0x1000` se trouve un registre de contrôle, qui permet de configurer le périphérique. Il est accessible en lecture et en écriture.



Les autres bits n'ont pas de fonctionnalité définie.

— À l'adresse `0x1004` se trouve un registre de données, qui permet uniquement de lire un octet provenant de l'UART. Il est accessible en lecture seule.

— À l'adresse `0x1008` se trouve un registre de données, qui permet uniquement d'écrire un octet à envoyer sur l'UART. Il est accessible en écriture seule. Chaque écriture dans ce registre déclenche l'envoi de l'octet correspondant sur l'UART.

Écrivez un pilote pour ce périphérique, en prenant garde à bien qualifier vos variables avec les contraintes nécessaires. Ce pilote sera défini par un fichier d'en-tête `uart.h` et un fichier source `uart.c`, définissant les fonctions suivantes :

— `void uart_init(/*...*/)` : initialise l'UART avec les paramètres du périphérique.

— `uint8_t uart_read()` : lit un octet sur l'UART et le retourne. Bloque le programme tant qu'aucune donnée n'est disponible.

— `void uart_write(const uint8_t data)` : écrit un octet sur l'UART.

VI Sécurité de la microarchitecture

Au delà des problèmes de vulnérabilité des circuits intégrés, la complexité de la microarchitecture est susceptible de crée des problèmes de sécurité. Il est en effet possible d'exploiter les comportements de la microarchitecture, pour obtenir des informations sensibles ou pour contourner des mécanismes de sécurité, tout en respectant scrupuleusement le jeu d'instructions. A l'heure où ces lignes sont écrites, les microarchitectures sont très vulnérables avec de nombreuses nouvelles failles sévères tous les ans.

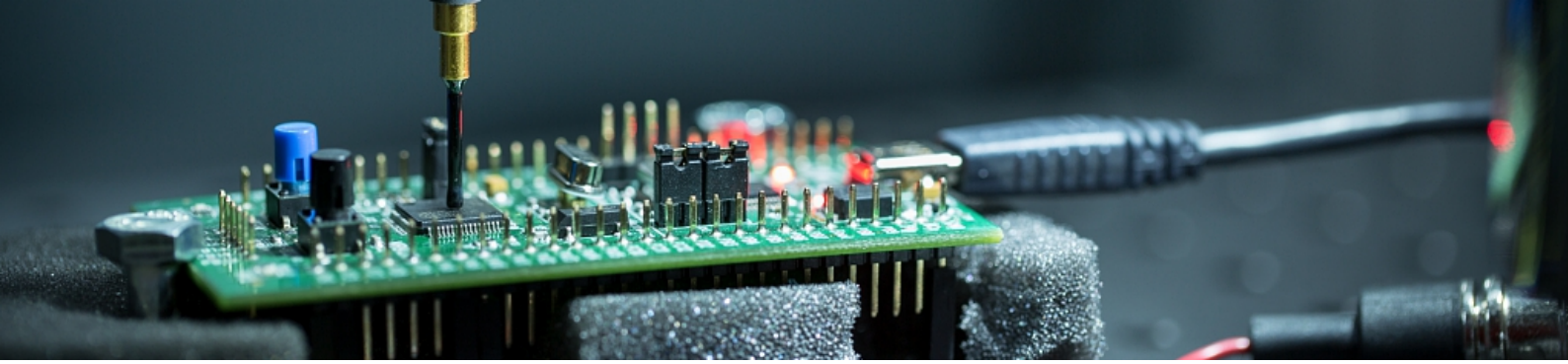
Dans cette partie, nous étudierons les différents soucis liés à la microarchitecture et donnerons des pistes pour y remédier.

Suggestions de lecture

Sécurité matérielle des systèmes - Vulnérabilité des processeurs et techniques d'exploitation [46]

Par *O. Savry, T. Hiscock, M. El Majihi*

ISBN : 210079096X



22. Attaques sur les caches

Les attaques sur les caches (*cache timing attacks*) permettent l'exfiltration de données au travers de canaux cachés ou de canaux auxiliaires. Les mémoires caches ont été décrites dans la section 15.2, aussi ne reviendrons-nous pas sur ses principes de fonctionnement.

Dans ce chapitre, nous verrons plusieurs attaques différentes s'appuyant sur les mémoires caches.

Ces attaques, maintenant anciennes puisque connues dès 2005 [54], reposent toute sur un principe similaire : suivant l'état du cache, le temps pour réaliser différentes opérations diffère. Avec la bonne stratégie, on peut utiliser ce principe pour lire un secret.

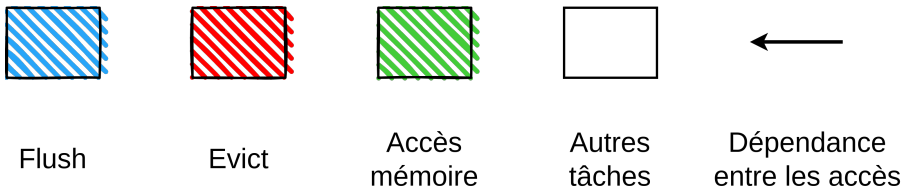


FIGURE 22.1 – Légende pour les illustrations des attaques sur les caches.

Ces attaques reposent sur la possibilité d'effacer des données du cache, de la part de l'attaquant. Il existe deux méthodes pour cela : soit en utilisant une instruction dédiée, on parle alors de *flush*.

D'autre méthodes existent en cas d'absence d'instruction dédiée, notamment en chargeant d'autres données en cache jusqu'à évincer la ligne cible. Il est possible d'optimiser cette méthode en utilisant la politique d'éviction du cache, souvent connue : on établis un ensemble d'éviction (*eviction set*), le séquence minimale de requêtes mémoires permettant de chasser la ligne cible du cache.

Le terme *evict* peut signifier deux choses suivant le papier : soit ils utilisent une éviction indirecte (via l'utilisation d'un *eviction set* par exemple), soit ils ne précisent pas la méthode utilisée (qui peut être un *flush*). Dans ce cas *evict* est un terme abstrait pour signifier que la ligne cible n'est plus en cache.

La rédaction de ce chapitre s'est appuyé sur les thèses de Mathieu Escouteloup et Thomas Rokicki.

22.1 Evict+Time

Le principe de l'attaque Evict+Time, illustré figure 22.2, est de mesurer la variation temporelle d'un programme cible lorsque l'attaquant évince certaines lignes du cache.

La boucle d'attaque, exécutée répétitivement, est la suivante :

1. Trigger : la première étape est d'obtenir un temps de référence pour l'exécution de la cible tout en remettant les caches dans leur état de base.
2. Evict : l'attaquant choisit des lignes de cache à évincer.
3. Time : la cible est réexécutée et le nouveau temps d'exécution est comparé au temps de référence.

Il faut donc que l'attaquant et la victime partage le cache. Une différence de temps entre la référence et la mesure après éviction indique que la cible utilise cette ligne de cache.

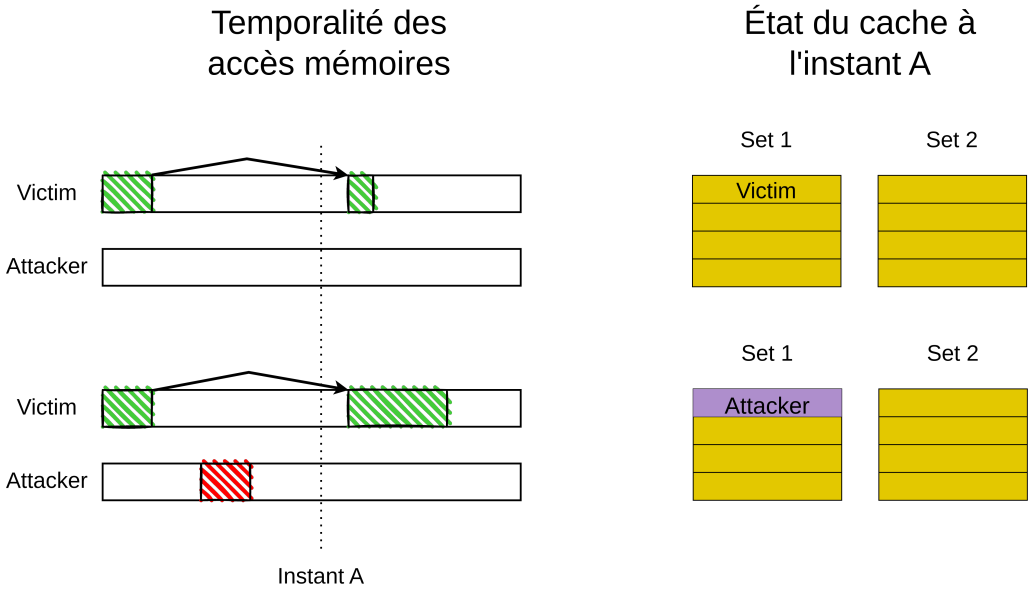


FIGURE 22.2 – Illustration de l'attaque Evict+Time. L'accès mémoire de la victime est allongé dans le cas de l'éviction d'une ligne de cache par l'attaquant.

22.2 Prime+Probe

Le principe du Prime+Probe, illustré figure 22.3, est à l'inverse de Evict+Time de retrouver les lignes évincées par la cible.

1. Prime : l'attaquant remplit la mémoire cache avec ses données.
2. Trigger : la cible est exécutée.
3. Probe : l'attaquant, en mesurant les temps d'accès aux données précédemment écrites, retrouve les lignes qui ont été évincées.

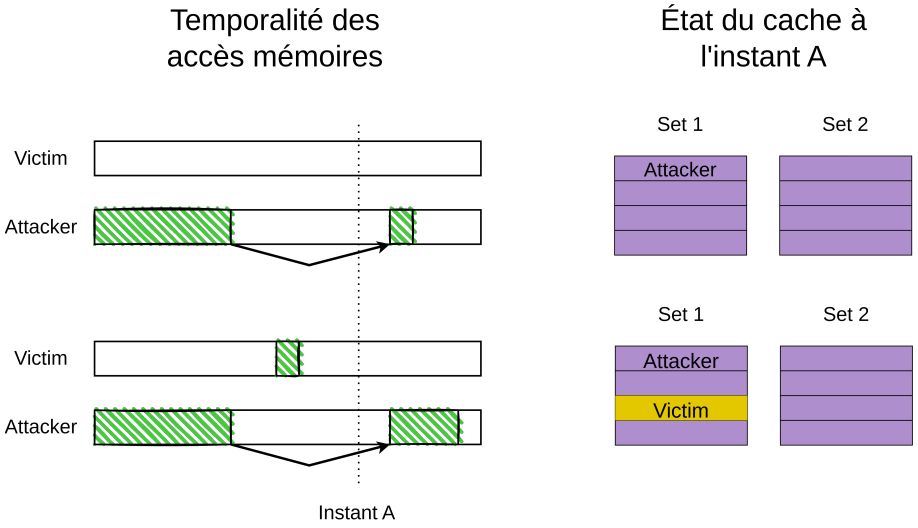


FIGURE 22.3 – Illustration de l'attaque Prime+Probe. L'accès mémoire de l'attaquant est raccourcit par l'accès mémoire de la victime. Il s'agit d'une éviction involontaire de la part de la victime.

Cette attaque ne permet pas de retrouver beaucoup d'information, essentiellement quel ensemble (*set*) est utilisé par la victime. Toutefois, pour certaines applications [3], la série temporelle des ensembles utilisés peut suffire à retrouver un secret. De plus cette attaque marche même si l'attaquant et la victime n'ont pas de mémoire partagée.

22.3 Flush+Reload

Cette attaque, illustrée figure 22.4, cible plus spécifiquement la mémoire partagée entre l'attaquant et sa victime. Par exemple, lorsqu'ils partagent une bibliothèque commune.

1. Flush : la mémoire cache est vidée à l'aide des instructions dédiées.
2. Trigger : la cible est exécutée.
3. Reload : l'attaquant utilise certaines zones mémoires partagées dans le but de détecter celles qui ont été utilisées par la cible. C'est-à-dire qu'il va réaliser une lecture à une adresse partagée, et il mesure le temps que cela prend.

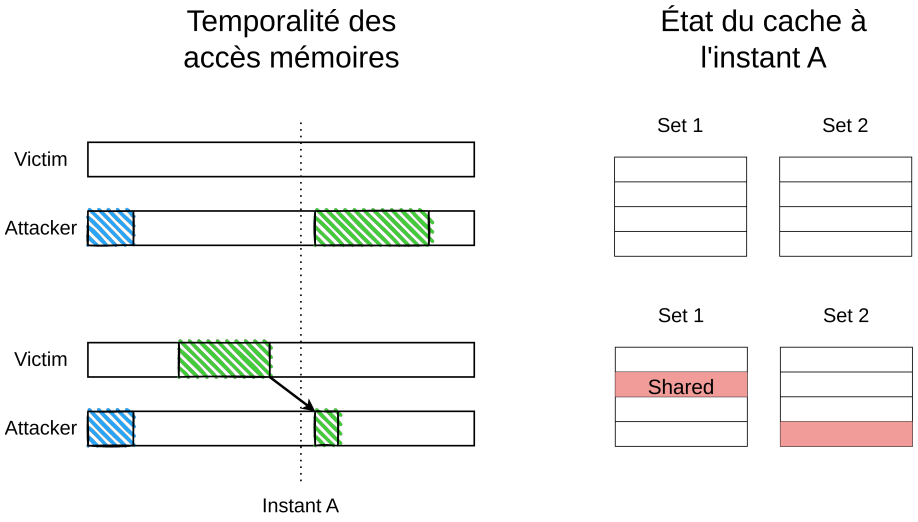


FIGURE 22.4 – Illustration de l'attaque Flush+Reload. L'accès mémoire de la victime raccourcit le temps pour l'accès dépendant suivant provenant de l'attaquant.

22.4 Flush+Flush

Flush+Flush est une variante de Flush+Reload qui remplace la dernière étape, illustrée figure 22.5.

1. Flush : la mémoire cache est vidée à l'aide des instructions dédiées.
2. Trigger : la cible est exécutée.
3. Flush : l'attaquant revide le cache tout en mesurant le temps pris pour cette opération.

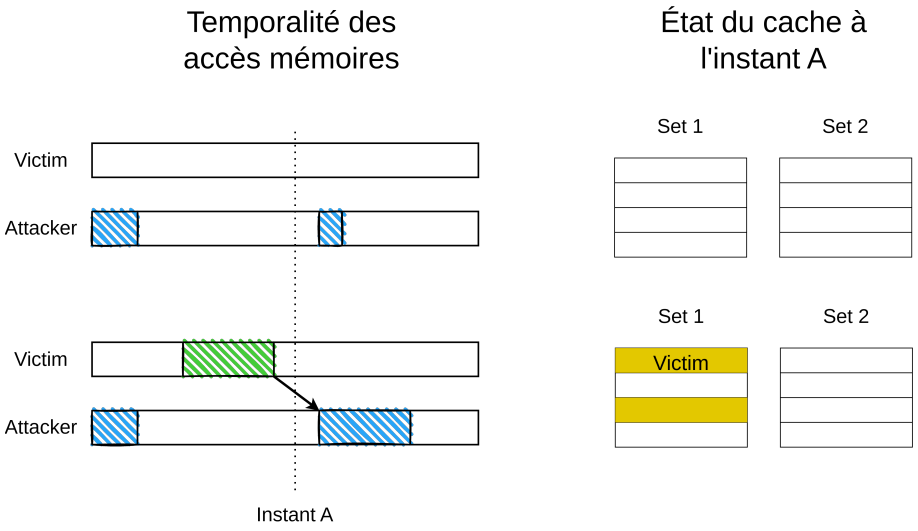


FIGURE 22.5 – Illustration de l'attaque Flush+Flush. Du fait de l'accès mémoire de la victime, un flush à une adresse dépendante sera plus long qu'à une adresse indépendante.

Cette attaque repose sur l'utilisation de l'instruction `clflush` du jeu d'instruction x86. Cette instruction de « flush » prend en argument une adresse, et elle évince la ligne de cache correspondante dans la hiérarchie mémoire. Ainsi, cette instruction ne prend pas le même temps pour s'exécuter s'il y a un *hit* ou un *miss* pour l'adresse donnée en argument.

Cette variante utilise le temps pris pour exécuter `clflush`, plutôt que le temps pris pour un accès mémoire dans Flush+Reload. C'est-à-dire que l'attaquant ne réalise aucun accès mémoire, cette variante est donc beaucoup plus discrète.

22.5 Exploitation d'une attaque sur les caches

Les différentes attaques présentées permettent d'exfiltrer de l'information qui devrait rester secrète. En un sens, il s'agit donc d'une primitive d'attaque qui doit être combinée avec d'autres éléments pour réaliser un exploit complet.

Dans cette section, nous présentons comment utiliser l'attaque d'un cache pour retrouver une clé AES. Nous nous concentrons sur une implémentation d'AES utilisant les T-boxes.

Cette implémentation, courante sur les processeurs 32-bits, combine le calcul des S-boxes et de MixColumns dans un parcours de tableau. Nous définissons 4 T-boxes,

$$T_0(x) = \begin{pmatrix} SB(x)\cdot 2 \\ SB(x) \\ SB(x) \\ SB(x)\cdot 3 \end{pmatrix}, T_1(x) = \begin{pmatrix} SB(x)\cdot 2 \\ SB(x)\cdot 3 \\ SB(x) \\ SB(x) \end{pmatrix}, T_2(x) = \begin{pmatrix} SB(x) \\ SB(x)\cdot 2 \\ SB(x)\cdot 3 \\ SB(x) \end{pmatrix}, T_3(x) = \begin{pmatrix} SB(x) \\ SB(x) \\ SB(x)\cdot 2 \\ SB(x)\cdot 3 \end{pmatrix}$$

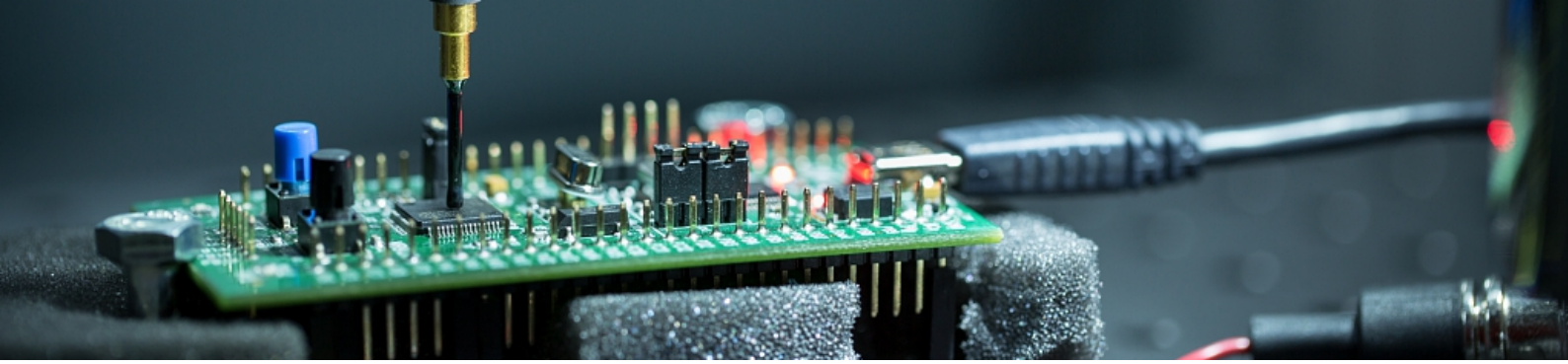
À partir de ces tables, il est possible de calculer une colonne du `STATE` avec

$$out_j = T_0(in_{0,j}) \oplus T_1(in_{1,j-1}) \oplus T_2(in_{2,j-2}) \oplus T_3(in_{3,j-3}) \oplus RK_j,$$

où $in_{i,j}$ est le `STATE` avant le tour, adressable par octet, et out_j est le `STATE` après le tour, adressable par colonne. Cette implémentation permet de calculer un tour d'AES à l'aide de 4 lectures dans un tableau. Chaque tableau fait 1 ko, avec 256 mots de 32 bits.

Ainsi, imaginons un attaquant capable de réaliser une attaque Prime+Probe. Il remplit la mémoire cache avec ses propres données. Il exécute ensuite le calcul du premier tour de l'AES avec un texte clair connu. On rappelle que l'AES commence par l'opération `AddRoundKey`, donc avant le premier tour et l'utilisation des T-boxes. Après ce premier tour, l'attaquant sonde (Probe) la mémoire cache pour retrouver les lignes de caches qui ont été évincées par l'AES. Cette opération peut être répétée si nécessaire.

Les adresses touchées par la victime correspondent aux adresses accédées dans les T-boxes, dont l'offset sont les octets du `STATE` après le premier `AddRoundKey`. Pour retrouver quelle valeur d'octet correspond à quelle position dans le `STATE`, il faut recommencer l'opération avec plusieurs textes clairs différents. Si l'on change un octet du texte clair, et qu'un seul offset a changé, on peut associer valeur et position. Enfin, connaissant les valeurs avant et après `AddRoundKey`, il est trivial de retrouver la clé.



23. Attaques sur la prédiction de branchement

23.1 Utiliser les prédicteurs pour établir des canaux cachés

Toute structure microarchitecturale utilisant des éléments de mémorisation peut servir de support à des canaux cachés. Les prédicteurs de branchement ne font pas exception et nous allons voir ici comment en construire un à partir du BHT.

L'idée est d'utiliser un gadget qui permet de forcer l'état de la table du BHT. Ce gadget est constitué d'instructions `blt a0, a1, end` qui comparent les deux registres `a0` et `a1` et branchent à l'adresse de l'étiquette `end` si le premier est strictement inférieur au second (*branch if lower than*). Il y a autant d'instructions de branchement que de compteur dans la BHT. Enfin la dernière instruction à l'adresse étiquetée `end` est un retour de fonction (`ret`).

L'initialisation de l'attaque consiste à effacer tous les compteurs en exécutant le premier branchement avec une condition $\mathrm{N}($`a0` $\not<$ `a1`$)$, comme illustré sur la figure 23.1. Cette opération est répétée 4 fois pour mettre tous les compteurs à 0.

Puis le Troyen va choisir la valeur i à encoder, une valeur correspondant à un compteur unique de la BHT. Il va exécuter l'instruction correspondante 3 fois avec la condition $\mathrm{T}($`a0` $<$ `a1`$)$, comme illustré sur la figure 23.2. À cause de la condition, seul ce branchement sera exécuté puisqu'il saute à `end` pour l'instruction suivante.

Enfin l'Espion va tester la valeur o, correspondant à un compteur, en mesurant le temps pris par ce branchement avec une condition T. S'il est rapide, alors $i = o$ sinon les valeurs sont différentes.

En mesurant les temps pour toutes les combinaisons i et o, nous obtenons une figure comme la figure 23.3 mettant en avant la présence d'un canal caché.

Nous avons maintenant un canal de communication qui pourrait permettre au Troyen d'exfiltrer un secret.

Exercice 23.1 - *Quantifier la fuite*

Imaginons un canal caché se reposant sur un BHT à 8 compteurs. Le temps du branchement est toujours de 24 cycles lorsque $i = o$ et toujours de 40 cycles sinon.

1. Quelle quantité d'information, en bits par message, pouvons-nous communiquer par ce biais, pour une distribution uniforme des messages à envoyer ?
2. Combien de messages faut-il envoyer pour transmettre une clé RSA sur 2048 bits ?

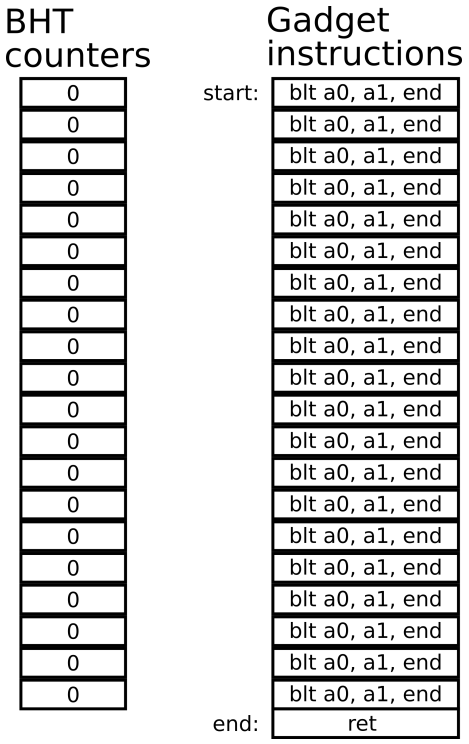


FIGURE 23.1 – Initialisation des compteurs à 0.

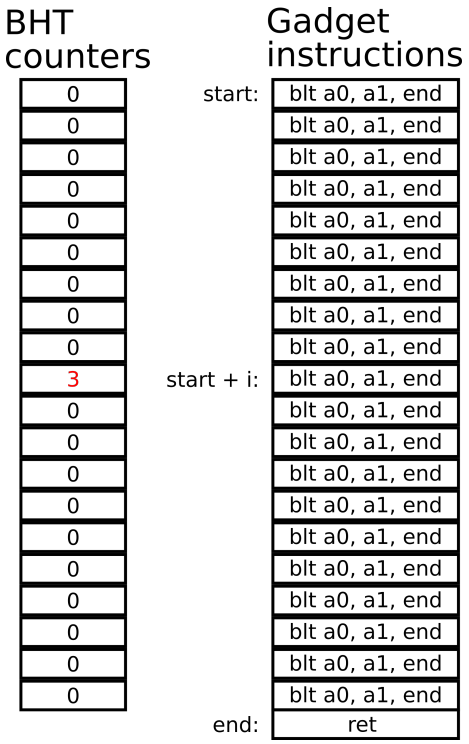


FIGURE 23.2 – Le Troyen choisit un compteur qu'il va incrémenter en exécutant le branchement avec la condition T.

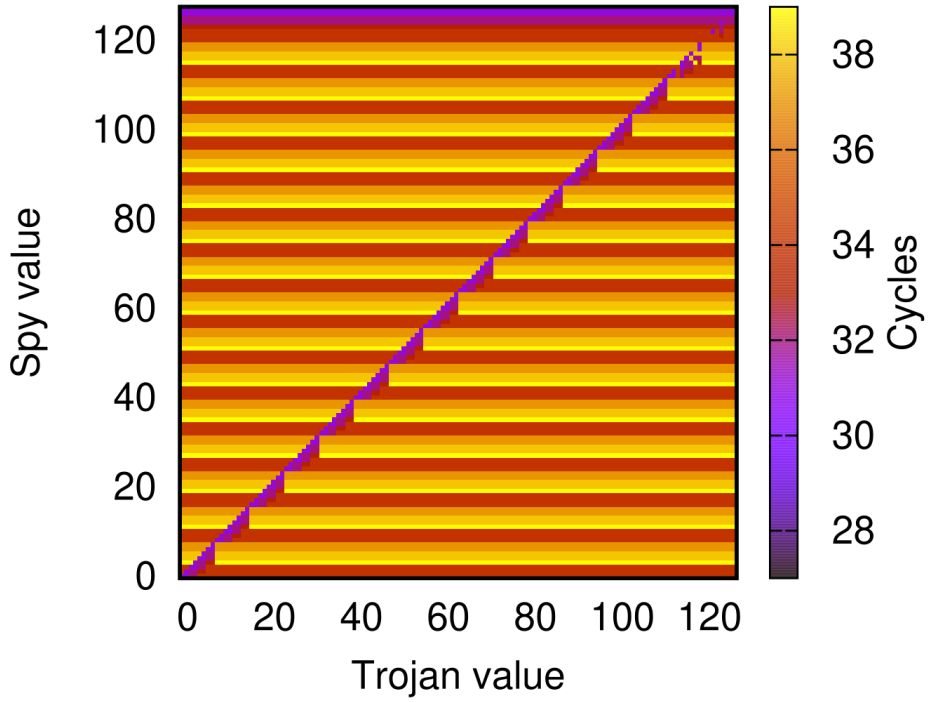


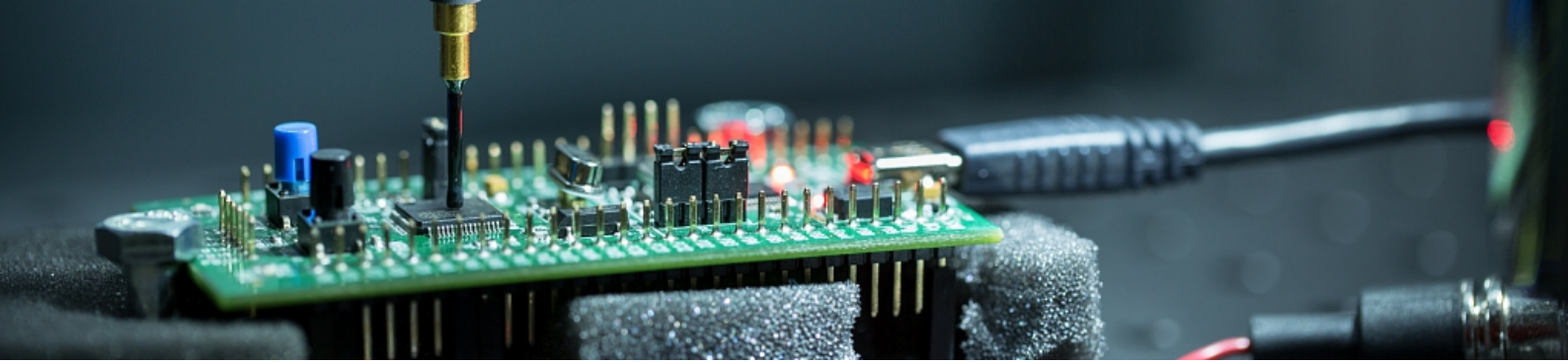
FIGURE 23.3 – La matrice des temps sur le cœur Aubrac mettant en évidence la présence d'un canal caché.

23.2 Subvertir un prédicteur pour détourner le flot de contrôle

Avec la prédiction de branchement vient la possibilité d'exécuter des *instructions transientes*, c'est-à-dire des instructions qui seront finalement abandonnées quand la prédiction se trouve erronée. La subversion du prédicteur consiste à entrainer le prédicteur pour obtenir une mauvaise prédiction.

Il s'agit d'une opération d'écriture dans les microstructures au même titre que le Troyen écrivant pour émission dans un canal caché. Mais plutôt que d'utiliser cette écriture pour exfiltrer une donnée, il est possible de s'en servir pour détourner le flot de contrôle pour des instructions transientes.

Ce cas est notamment utilisé pour l'attaque Spectre-PHT (cf. section 25.2).



24. Attaques sur les prefetchers

Similairement aux prédicteurs de branchement, les prefetchers peuvent être utilisés pour monter des exploits.

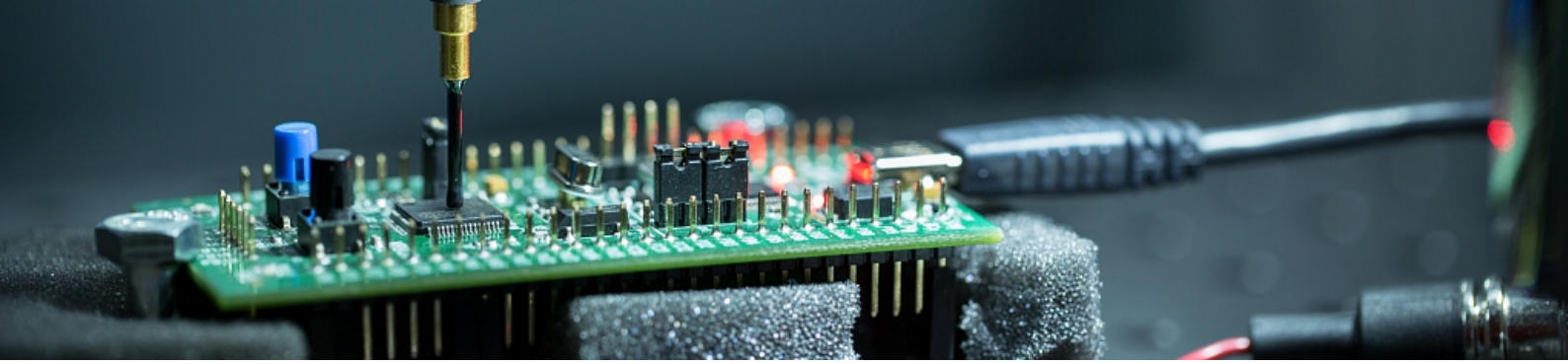
24.1 Utiliser un prefetcher pour construire un canal caché

Puisque le prefetcher influence le temps d'accès aux données, l'attaquant peut l'utiliser pour construire un canal caché. Par exemple avec un stride prefetcher.

1. L'attaquant voulant exfiltrer le secret s va générer les accès mémoires aux adresses a, $a+s$, $a+2\cdot s$, etc.
2. L'attaquant va ensuite mesurer le temps pour lire les données aux adresses a' suivi de $a'+t$ en faisant varier t (a' est une adresse synonyme à a dans la table du prefetcher). Si le deuxième accès est rapide pour un certain t_0 alors $t_0 = s$.

24.2 Utiliser un prefetcher pour détourner le flot de contrôle

Dans le cadre d'une exécution spéculative, la prédiction du prefetcher est une composante de la spéculation qui influence quelles instructions seront utilisées après celles en cours. Un attaquant capable de subvertir un prefetcher est donc capable de détourner le flot de contrôle, ce qui peut être utilisé pour exfiltrer un secret.



25. Attaques sur la spéculation

Les attaques spéculatives combinent plusieurs vulnérabilités en un exploit. L'idée est d'exploiter les traces de l'exécution **transiente** (anglicisme, une meilleure traduction serait exécution transitoire). Il s'agit de l'exécution spéculative qui est effacée après avoir été déterminée incorrecte. Toutefois, l'effacement est rarement complet, il reste une modification des états microarchitecturaux qui peut être lue par l'attaquant.

Une attaque spéculative se décompose usuellement en plusieurs parties :

1. Préparation de l'état microarchitectural pour déclencher une mauvaise spéculation.
2. Déclenchement de la spéculation, via une prédiction de branchement.
3. Contrôle de la fenêtre d'exécution spéculative si possible.
4. Lecture du secret spéculativement.
5. Encodage du secret dans la microarchitecture, c'est le Troyen.
6. Récupération du secret par l'Espion.

25.1 Meltdown

L'attaque Meltdown [21] est publiée en 2018 et met en évidence la problématique de la gestion des exceptions dans la microarchitecture.

```
uint32_t forbidden_secret = *secret_address;
uint32_t dummy = array1[forbidden_secret * 4096];
```

FIGURE 25.1 – Le code C de l'attaque Meltdown.

Le principe de l'attaque Meltdown est de juste lire le secret, puis de l'encoder dans un tag de cache. La chose extraordinaire est qu'il est possible de lire le secret !

En effet, pour gagner en performances, la totalité du noyau est mappée (traductible) en mémoire virtuelle pour tous les processus utilisateurs. Toutefois ces pages sont protégées à l'aide d'un bit qui spécifie le niveau de privilèges nécessaire pour y accéder. Ainsi, bien que le noyau soit mappé il n'est normalement pas possible d'accéder à cette mémoire sous peine d'une exception matérielle.

L'attaque Meltdown fonctionne sur les processeurs Intel des générations 3XXX à 8XXX, sur certains processeurs ARM, mais pas sur ceux d'AMD. En effet le comportement des processeurs vulnérables est particulier :

1. Pour la lecture du secret, la traduction virtuelle vers physique est effectuée même si le niveau de privilège n'est pas bon.

2. Une exception est levée du fait du mauvais niveau de privilèges.
3. Toutefois cette exception n'est traitée que lors de son commit, laissant une fenêtre d'exécution spéculative.
4. L'attaquant utilise cette fenêtre pour écrire le secret dans le tag du cache.
5. Le cœur traite l'exception et annule l'exécution spéculative. Trop tard, le secret est dans le tag du cache.

Cette attaque est simple à contrer, il suffit de vérifier et de faire appliquer les droits avant la traduction. C'est ce que faisait AMD et maintenant tout le monde, cette attaque n'est donc plus d'actualité sur les processeurs récents.

Nous pouvons également questionner la pertinence de mapper l'entièreté du noyau pour les processus utilisateurs. Le noyau Linux commence tout juste à remettre à plat cette pratique [1].

25.2 Spectre

Contrairement à Meltdown qui peut être vu comme une erreur d'implémentation, Spectre est une attaque liée au principe même de l'exécution spéculative. Il est donc beaucoup plus difficile de s'en prémunir. Aujourd'hui, et soyons honnêtes dès sa publication, il a toujours été difficile d'exploiter les attaques Spectre. Toutefois la vulnérabilité est bien réelle et toujours présente, ce qui permet à de nouvelles attaques d'apparaitre régulièrement. [2]

25.2.1 Principe de base

L'attaque Spectre [16] est publiée en 2018 et reste l'archétype de l'attaque spéculative.

```
if (x < array1_size) {
    y = array2[array1[x] * 4096];
}
```

FIGURE 25.2 – Le code C de l'attaque Spectre-PHT.

Le code de la figure 25.2 peut être décomposé comme suit :

— L'attaquant exécute ce même code de nombreuses fois avec la condition de branchement respectée. Ou, suivant le scénario d'attaque, l'historique des branchements est contrôlé par un autre processus pour faire croire que ce branchement sera toujours pris.
— Une nouvelle exécution est lancée qui ne respecte pas la condition de branchement (index dans les limites du tableau).
— À cause de la prédiction de branchement, la branche « if » est tout de même exécutée même si l'index n'est pas dans les limites du tableau.
— Un `LOAD` spéculatif est exécuté à l'adresse arbitraire `array1 + x`, pour lire le secret s.
— Un deuxième `LOAD` spéculatif est exécuté pour exfiltrer le secret dans le tag d'une mémoire cache. Après que le secret ait été décalé vers la gauche ($\cdot * 4096$) pour tenir compte de la troncation du tag.

L'Espion réalisera alors une attaque par temps d'accès au cache, par exemple Flush+Reload, pour lire le secret.

Une particularité intéressante de cette attaque est que le gadget figure 25.2 peut être du code noyau, il suffit à l'attaquant de contrôler deux éléments :

— L'offset x, par exemple si c'est un argument d'un appel système.
— De l'historique des branchements, qui déterminera la direction de la prédiction. **Si le prédicteur de branchement est partagé entre le noyau et l'utilisateur, il suffit**

1. https://lwn.net/Articles/803823/
2. Pour aller plus loin, voir le papier review https://arxiv.org/abs/2309.03376.

à l'attaquant de l'entrainer en mode utilisateur. Ce dernier point est la principale contremesure ajoutée sur les processeurs récents.

Cette variante présentée est communément appelée Spectre-PHT, car elle s'appuie sur une corruption de la prédiction de direction de branchement (ce à quoi sert la PHT). On voit le problème d'avoir une ressource partagée, l'historique des branchements, entre différents domaines de sécurité.

Toutefois, il ne suffit pas de supprimer la PHT pour contrer l'exploit, des variantes existent pour tous les mécanismes permettant l'exécution spéculative.

25.2.2 Variantes

Il existe de très nombreuses variantes de Spectre, selon deux principales dimensions : la source de la spéculation, et le canal caché utilisé pour exfiltrer le secret. D'autres éléments peuvent varier, comme la façon de créer une grande fenêtre spéculative.

Sources de la spéculation Les variantes de Spectre sont souvent nommées d'après la source de la spéculation. D'autres noms peuvent exister, notamment les «_Variantes 1, 2, 3, 4_», mais il s'agit des mêmes attaques.

— Spectre-PHT : spéculation due à la prédiction de la direction de branchement (cf sous-section 7.2.3).
— Spectre-BTB : spéculation due à la prédiction de la cible d'un branchement (cf section 8.1).
— Spectre-RSB : spéculation due à la prédiction de la cible d'un branchement indirect de type retour (cf section 8.2).
— Spectre-STL : spéculation due à la prédiction de l'aliasing mémoire (cf section 6.3).

Pour ces variantes, il s'agit d'utiliser un mécanisme différent de spéculation, entrainant des différences dans la méthode d'induire le processeur en erreur, et plus ou moins facile à exploiter et à contrer.

Canal caché Même si la plupart des attaques spéculatives utilisent le cache comme canal caché, un peu par défaut, il existe d'autres possibilités. On rappelle que tout élément de mémoire peut être le support d'un canal caché.

Les variations de canal caché ne donnent traditionnellement pas lieu à un nom de variante Spectre, mais il est important de bien identifier le canal utilisé pour comprendre, et contrer, chaque attaque.

Pour donner un ordre de grandeur, le papier review https://arxiv.org/abs/2309.03376 classe les attaques selon 13 canaux cachés différents.

25.2.3 Contremesures

Les contremesures contre Spectre ciblent les différents éléments de l'attaque :

— La possibilité pour l'attaquant de contrôler le prédicteur de branchement en mode noyau.
— Le déclenchement de la spéculation.
— Le fait de pouvoir lire un secret en mode spéculatif.
— Le fait de pouvoir exfiltrer un secret en mode spéculatif, c'est-à-dire qu'il est possible d'interdire l'utilisation de certaines instructions, comme le deuxième `LOAD`, dans ce mode.

Retpoline Une des toutes premières contremesures mises en œuvre est une contremesure logicielle qui pouvait donc être utilisée sur des processeurs vulnérables. L'idée est d'empêcher le comportement spéculatif.

Le fonctionnement de Retpoline est de tromper les mécanismes de prédiction de destination de branchement. Dans la version Retpoline le saut indirect est réalisé par un saut avec retour et doit donc mobiliser la prédiction de retour (cf section 8.2). Sauf que l'on trompe la microarchitecture sur la destination de ce retour, en lui faisant croire que c'est la boucle `capture_spec:` qui est la destination du retour.

```
                                jal set_up_target; # ra <- (capture_spec)
                                capture_spec:
                                    pause;
jr a0;                    →         j capture_spec;
                                set_up_target:
                                    mv ra, a0; # ra <- a0
                                    ret; # jr ra
```

FIGURE 25.3 – Le code assembleur de la contremesure Retpoline, un gadget remplaçant un saut indirect.

Cette contremesure est une manière astucieuse de rendre la spéculation inopérante sur les sauts indirects depuis le logiciel. Mais elle souffre de défauts rédhibitoires. Quand, sur quels sauts, doit-elle être utilisée ? De par le fonctionnement de l'attaque Spectre, elle est nécessaire sur tous les sauts indirects dans le code qui peut accéder au secret, donc au moins tout le noyau. Donc on perd l'avantage de la spéculation (environ 30% de performances), en en payant le prix matériel. De plus cette contremesure n'est pas toujours efficace, notamment contre Spectre-PHT et Spetre-STL.

Indirect branch restricted speculation (IBRS) La contremesure IBRS est une contremesure matérielle, mise en œuvre sur les processeurs récents ciblant les sauts indirects (et donc Spectre-BTB et Spectre-RSB). IBRS est le nom donné par Intel, mais il y a des équivalents chez d'autres concepteurs. Cette contremesure permet de garantir que les prédictions de destination de saut indirect ne peuvent pas être influencées par un programme exécuté avec un niveau de privilège inférieur.

Autrement dit, le code utilisateur ne peut plus contrôler la prédiction de destination en mode noyau. Cette contremesure n'empêche pas la spéculation ni à un `LOAD` spéculatif de lire un secret, mais elle empêche l'attaquant de contrôler la prédiction de destination de branchement ce qui limite beaucoup l'exploitabilité de l'attaque.

Cette protection ne s'applique que pour des attaques entre niveaux de privilège. Elle ne fait rien contre un attaquant en mode utilisateur qui attaque un autre processus en mode utilisateur.

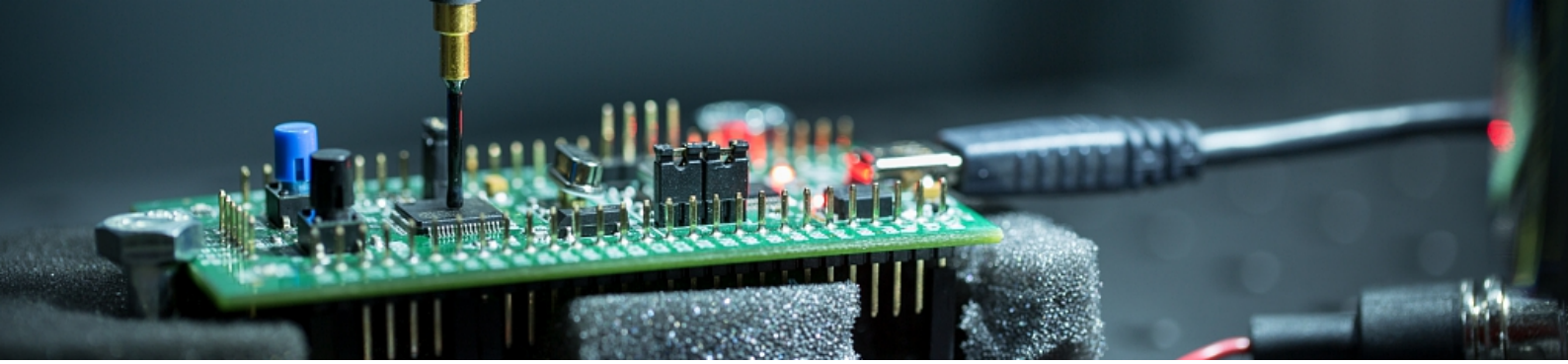
Indirect branch prediction barrier (IBPB) Contremesure à coupler avec l'IBRS pour traiter les cas manquants. Il s'agit d'une instruction barrière avec la sémantique suivante : les instructions avant la barrière ne peuvent pas influencer les prédictions de destination de branchement après la barrière.

Insérer ces barrières dans un programme peuvent permettre de protéger un processus utilisateur contre un autre processus utilisateur par exemple.

LFENCE L'instruction `LFENCE` pour *load fence* est une instruction de sérialisation : cette instruction ne s'exécute que lorsque tous les `LOAD` précédents l'instruction ont été complétés. Elle est donc un frein à la spéculation, car elle implique un point unique de passages des files d'exécution. Cette instruction a longtemps été recommandée par AMD comme leur solution à Spectre jusqu'à ce qu'un papier [24] (publié par des chercheurs d'Intel) montre bien qu'il s'agit d'un frein et non d'un arrêt total de la spéculation.

Les barrières spéculatives S'il n'était pas possible de stopper la spéculation quand les attaques ont été publiées pour la première fois, il est possible de rajouter des instructions dédiées dans les nouveaux processeurs. La barrière spéculative, comme `SB` dans le jeu d'instruction ARM, est une instruction qui ne peut pas être exécutée spéculativement. Elle permet donc d'arrêter le comportement spéculatif au moment désiré.

La question restante est de savoir quand utiliser ces instructions. Sans réponse faisant consensus aujourd'hui.



26. Simultaneous multithreading

Le multithreading matériel est un ensemble de technologies rendu connues sous le nom de Hyper-Threading dans les Pentium 4 de Intel en 2002. Nous allons ici préciser les différents types de multithreading et les vulnérabilités associées.

26.1 Le multithreading

Il n'existe pas une seule sorte de multithreading mais au moins trois.

— Le multithreading à grain grossier (*coarse grained multithreading*) consiste à exécuter un certain nombre d'instructions d'un programme, puis lors de l'exécution d'une instruction à forte latence (L3 miss, accès à un périphérique ...), basculer **matériellement** vers un autre programme. Dans ce cas, le processeur possède plusieurs registres PC et bascule de l'un à l'autre avec la bonne gestion des contextes associés entre plusieurs programmes. Ainsi si une instruction dure plusieurs centaines de cycles, nous ne perdons pas de temps utile en l'utilisant pour un autre processus.

— Le multithreading à grain fin (*fine grained multithreading*) consiste à alterner à chaque cycle d'horloge entre plusieurs processus. Là encore, cela permet de cacher la latence des instructions en occupant le processeur.

— Enfin le multithreading simultané (*simultaneous multithreading (SMT)*) est la technique utilisée dans les processeurs modernes. Il s'agit de dupliquer le front end du processeur (fetch, decode, registre PC) pour donner l'illusion d'avoir plusieurs cœurs physiques. Toutefois ces cœurs partageront un même backend, notamment les unités d'exécution, permettant d'augmenter le taux d'utilisation de celles-ci. Il est généralement reconnu que le SMT augmente les performances d'un cœur de $\approx 30\%$, donc loin de l'augmentation attendue par la duplication du cœur. La taille supplémentaire nécessaire pour le SMT est loin d'être équivalente à l'ajout d'un nouveau cœur.

Le SMT est une alternative à la spéculation pour occuper les ressources d'un cœur. En effet, par défaut un cœur passe son temps à attendre la mémoire. Une première solution est la spéculation : on continue l'exécution sans attendre la mémoire. Le SMT est une autre solution, on exécute un autre processus en parallèle sur les mêmes ressources matérielles.

Il y a donc compétition pour les ressources du processeur par ces deux processus. Dans la nomenclature RISC-V, un thread matériel est appelé un hart. En général, le SMT n'utilise que 2 harts par cœur, mais certains processeurs d'IBM utilisent 8 harts par cœur par exemple.

26.2 La contention de ports

Comme les harts partagent les mêmes ressources matérielles, il est très facile pour l'un d'espionner l'autre. Ce qui peut poser de gros soucis lorsqu'il s'agit d'un serveur gérant plusieurs clients différents par exemple.

La contention de port est une attaque [2] qui permet d'exfiltrer des données en surveillant l'utilisation des unités d'exécutions. L'ordonnancement à l'exécution des instructions sur un cœur out-of-order utilise des ports : par exemple les ports $0, 1$ et 2 sont reliés à un ALU. Le port 3 est relié à un ALU et un multiplieur. Les ports 4 et 5 sont des LSUs pour gérer les accès mémoires, etc.

Une instruction est ordonnancée sur un port en fonction de la disponibilité de celui-ci. À cause du SMT, le temps d'exécution d'une instruction dépend donc de ce qui se passe sur l'autre hart.

26.2.1 Canal caché

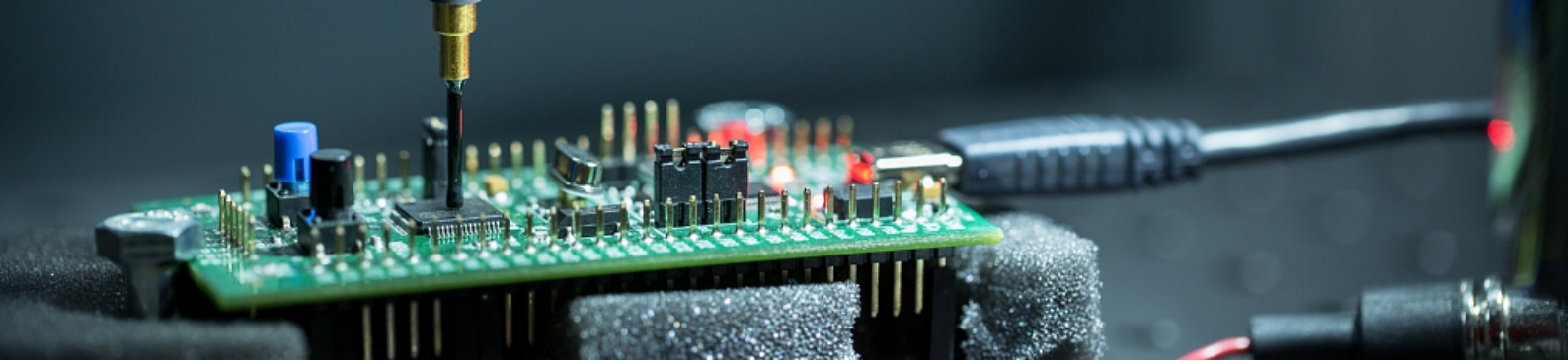
Pour créer un canal caché entre les deux harts, le Troyen va exécuter de manière continue des `add` pour envoyer un 0 et des `mul` pour envoyer un 1. L'Espion va mesurer le temps nécessaire pour exécuter une série de quelques `mul`. Si le Troyen envoie un 1, il exécute aussi des multiplications. Mais puisqu'il n'y a qu'une seule unité d'exécution le permettant, ces opérations seront plus lentes du point de vue de l'Espion. Il y a contention de port.

26.2.2 Canal auxiliaire

Dans un contexte de canal auxiliaire, la contention de port est utilisée pour espionner les opérations effectuées par l'autre hart. Avec une séquence savamment choisie par l'attaquant (voir [2] pour les détails), il est possible de construire une série temporelle de l'utilisation des ports par l'autre hart. Pour certains algorithmes, surtout de cryptographie asymétrique comme l'exponentiation rapide (cf Algorithme 10.4) et avec la bonne résolution temporelle, l'attaquant peut ainsi lire la clé cryptographique utilisée lors du calcul.

26.3 Contremesures

Il est considéré aujourd'hui que le niveau de partage de ressources lié à l'utilisation du SMT n'est pas compatible avec la sécurité. Seuls deux processus se faisant mutuellement confiance doivent être ordonnancés sur le même cœur. La distribution OpenBSD a pris la décision en 2018 de désactiver par défaut le SMT.



27. Rowhammer

L'attaque Rowhammer [15] est une attaque sur la mémoire DRAM.

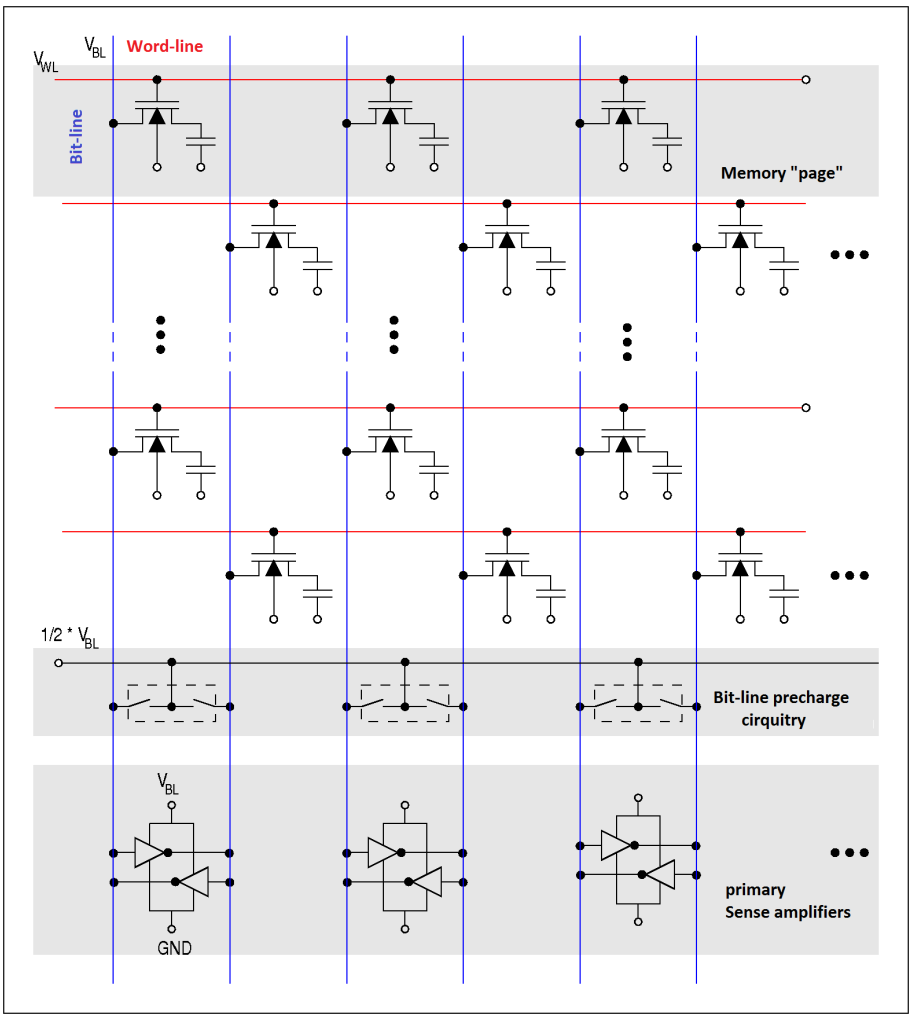


FIGURE 27.1 – Une mémoire DRAM composé de nombreuses cellules.

Comme détaillé dans la section 13.2, cette mémoire fonctionne avec des charges stockées dans un condensateur. Nous avons vu que la lecture ou l'écriture entraine un déplacement de

charges, qui doit être contré avec une rétroaction remettant le bon niveau de charge dans le condensateur.

Toutefois, avec la miniaturisation des mémoires, des phénomènes de fuite apparaissent : lors de l'ouverture d'une ligne pour la lecture ou l'écriture, les cellules mémoires adjacentes fuient des charges. La mémoire est dimensionnée pour que l'« utilisation normale » de la mémoire n'entraine pas des fuites qui pourraient changer l'état d'une cellule avant le prochain rafraichissement.

Mais rien ne contraint l'attaquant à une utilisation normale de la mémoire. En accédant de manière répétée à une même ligne, dans la fenêtre temporelle entre deux rafraichissements successifs, il est possible d'induire des fuites de charge qui change l'état d'une cellule. C'est ce que l'on appelle le martèlement de la mémoire.

27.1 Motifs d'accès

En général, l'attaquant désire fauter des bits, et donc des cellules mémoires spécifiques. Mais il n'est capable que d'altérer, avec une probabilité non contrôlée, les cellules adjacentes à la ligne martelée.

Un martèlement simple n'accède qu'à la même ligne et faute de manière non contrôlée une ou les deux lignes adjacentes.

Un martèlement double permet d'améliorer le taux de succès. Il consiste à marteler successivement les deux lignes adjacentes à celle ciblée.

Dans les deux cas, cela nécessite que l'attaquant ait un accès mémoire aux lignes adjacentes physiquement sur la puce DRAM.

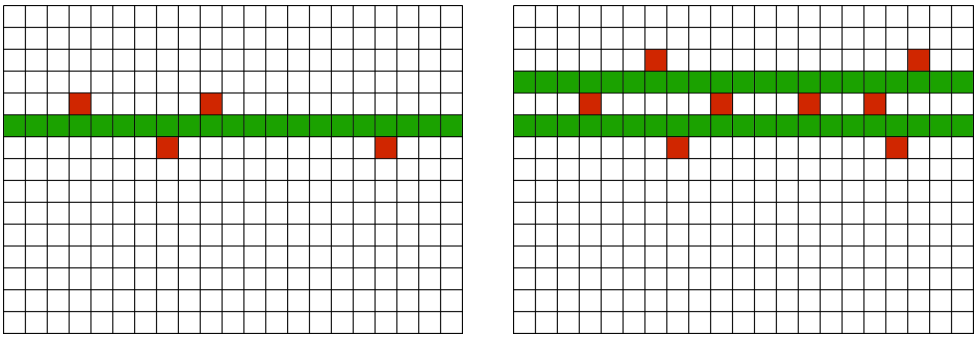


FIGURE 27.2 – Martèlement simple et double sur la ou les lignes vertes. Les cellules rouges sont celles observées fautées. Nous pouvons voir que le martèlement double permet d'être plus précis par rapport à la ligne cible choisie.

27.2 Les difficultés de mise en œuvre

Le principe derrière l'attaque est simple, la mise en œuvre est plus difficile. Comment réaliser des requêtes répétées à des lignes adjacentes ?

Placement en DRAM Le premier problème est de trouver la correspondance entre une adresse physique et sa localisation physique sur une même puce de DRAM. Pour cela, on va s'appuyer sur la documentation disponible : celle du SoC tout d'abord qui indique quelles adresses physiques sont reliées à la DRAM. Ou plus probablement, parce que cette documentation n'est pas toujours disponible, on va utiliser des techniques de rétro-ingénierie pour trouver ces informations, à adapter à chaque cas.

Ensuite, celle de la DRAM elle-même qui peut permettre de comprendre combien il y a de puces sur une même barrette et comment les lignes sont disposées physiquement.

Avec ces infos, l'attaquant va généralement réserver une grande partie de la mémoire pour avoir une bonne chance de contrôler une ligne adjacente à la cible.

MMU L'adresse physique d'un accès mémoire n'est généralement pas directement accessible par l'application. La MMU est responsable de la traduction des adresses virtuelles, celles vues par l'application, en adresses physiques. C'est normalement le système d'exploitation qui est responsable de la création et la maintenance des tables de correspondance, donc hors de portée de l'attaquant en général. Toutefois, si l'attaquant est capable de réserver une grande quantité de mémoire, en allouant de grands tableaux de données sur le tas par exemple, il récupère l'accès à une grande quantité de mémoire physique. Suivant l'ISA, il existe des instructions dédiées pour connaitre l'adresse physique pour une adresse virtuelle donnée.

Le cache Une autre difficulté est que le système est normalement conçu pour éviter des requêtes répétées à un même emplacement en RAM. C'est là tout l'objectif de la hiérarchie mémoire. Il faut donc être capable d'éviter les caches pour obtenir des accès répétés en RAM.

Une première solution est d'utiliser les propriétés mémoires d'une page, propriétés gérées par la MMU, pour spécifier que cette page ne doit pas être mise en cache. Mais cette possibilité est normalement sous la responsabilité du système d'exploitation.

Une autre solution est d'utiliser des *ensembles d'éviction* : en connaissant la politique d'éviction des différents niveaux de cache, il est possible de crée une séquence de requêtes mémoires qui maximisent le taux de miss dans les caches, entrainant des requêtes en DRAM.

27.3 Contremesures

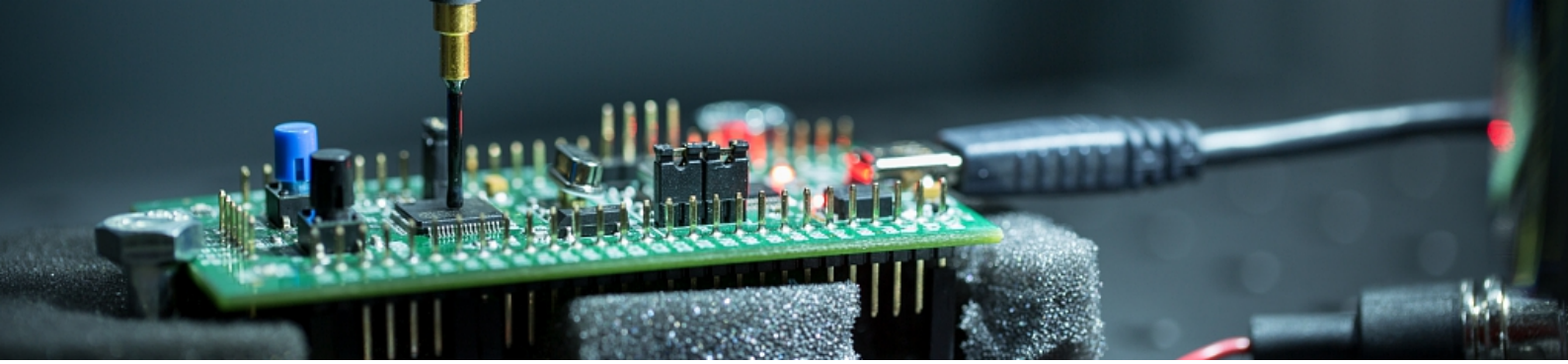
Cette attaque permet d'injection des fautes à distance, même sur un serveur dans un datacenter difficile d'accès.

Se protéger de Rowhammer implique de restreindre la possibilité d'avoir des requêtes sur des lignes de cellules proches spatialement dans la fenêtre entre deux rafraichissements.

Une première possibilité est donc de réduire cette fenêtre en rafraichissant la mémoire plus souvent. Mais cela implique une consommation énergétique supérieure et une réduction des performances puisque la lecture et l'écriture ne sont pas possibles pendant le rafraichissement. On préfèrera donc déclencher un rafraichissement que s'il y a eu de nombreux accès à des cellules proches physiquement. Nous pouvons également concevoir un circuit plus robuste en augmentant la taille des cellules, ce qui limitera la présence de fuites de charge au prix d'une densité moindre.

Ou au contraire, il est possible de réduire le débit des requêtes à des cellules proches en les mettant en mémoire tampon jusqu'au prochain rafraichissement.

Enfin, la correction d'erreur (ECC) peut être mise en place mais nécessite d'y allouer un certain nombre de cellules mémoires.



28. DVFS

28.1 Principe

Sur un SoC moderne, la tension et la fréquence des cœurs sont gérées dynamiquement par le SoC lui-même. Celui-ci embarque un petit microcontrôleur dont c'est souvent l'unique rôle. Ce mécanisme s'appelle le dynamic voltage and frequency scaling (DVFS). Le but est d'optimiser la performance et la consommation énergétique. Lorsqu'une tâche gourmande en ressources est utilisée, la fréquence du cœur est augmentée ainsi que la tension d'alimentation pour que le circuit fonctionne correctement à cette vitesse plus élevée. Cela entraine une consommation énergétique plus importante et nécessite d'évacuer la chaleur excédentaire ainsi générée.

Aussi, lorsque la tâche est terminée et qu'il y a moins de besoins, la fréquence et la tension sont abaissées, permettant à la puce de refroidir. Ce mécanisme explique pourquoi, lorsque vous voulez évaluer les performances d'un processeur, il est primordial de le tester avec une tâche lourde qui dure dans le temps pour évaluer le régime de croisière et non le pic de performances permit par le dynamic voltage and frequency scaling (DVFS).

28.2 CLKSCREW

CLKSCREW [39] est une attaque sur le DVFS publiée en 2017, qui exploite le DVFS pour injecter des fautes dans un cœur sans moyens d'injection matériels.

L'attaque consiste à identifier les zones où la paire tension, fréquence ne permet plus le fonctionnement normal de la puce, comme illustrée sur la figure 28.1. Il suffit alors d'appliquer ces paramètres pendant un court instant pour injecter une ou des fautes.

Cette attaque ne nécessite pas un accès physique à la puce et contourne les contremesures physiques telles que le bouclier.

Toutefois, n'importe quel processus utilisateur ne peut pas modifier les réglages énergétiques de la puce, le processeur doit être en mode de privilège superviseur. Cela limite l'attaque à un attaquant ayant déjà un accès privilégié à la machine. C'est pourquoi le but de l'attaque est ici de compromettre une enclave Intel SGX qui est censée être isolée du reste du système, même d'un attaquant en mode superviseur.

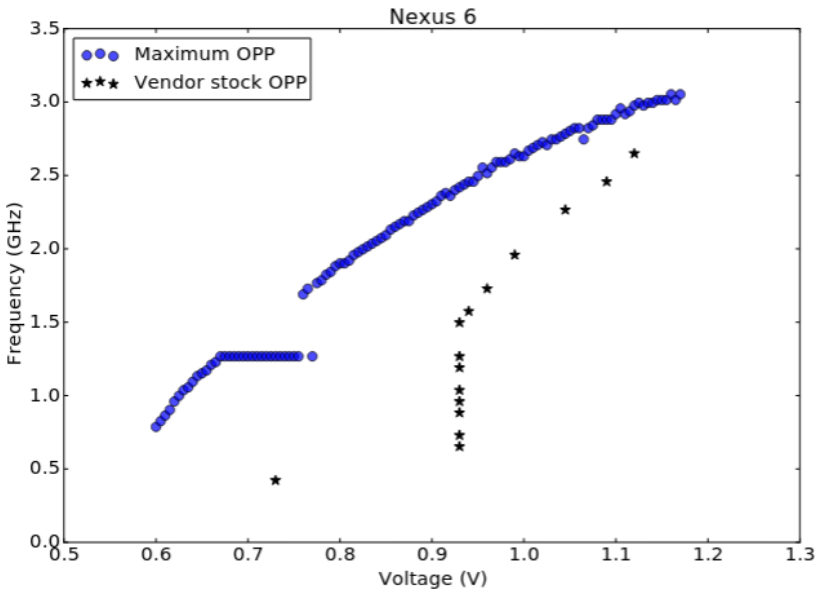


FIGURE 28.1 – Fréquence et tension fonctionnelles (Operating Performance Points OPP) selon le fabricant et selon les mesures. Tiré de [39]

VI Annexes

Bibliography

Articles

[1] Michel Agoyan et al. « How to flip a bit ? » In : **16th IEEE International On-Line Testing Symposium (IOLTS 2010), 5-7 July, 2010, Corfu, Greece**. IEEE Computer Society, 2010, pages 235-239. doi : 10.1109/IOLTS.2010.5560194. url : https://doi.org/10.1109/IOLTS.2010.5560194 (cf. page 110).

[2] Alejandro Cabrera Aldaya et al. « Port Contention for Fun and Profit ». In : **2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019**. IEEE, 2019, pages 870-887. doi : 10.1109/SP.2019.00066. url : https://doi.org/10.1109/SP.2019.00066 (cf. page 169).

[3] Daniel J Bernstein. « Cache-timing attacks on AES ». In : (2005) (cf. page 156).

[4] Dan Boneh, Richard A. DeMillo et Richard J. Lipton. « On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract) ». In : **Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding**. Sous la direction de Walter Fumy. Tome 1233. Lecture Notes in Computer Science. Springer, 1997, pages 37-51. doi : 10.1007/3-540-69053-0_4. url : https://doi.org/10.1007/3-540-69053-0%5C_4 (cf. page 93).

[5] Eric Brier, Christophe Clavier et Francis Olivier. « Correlation Power Analysis with a Leakage Model ». In : **Cryptographic Hardware and Embedded Systems - CHES 2004 : 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings**. Sous la direction de Marc Joye et Jean-Jacques Quisquater. Tome 3156. Lecture Notes in Computer Science. Springer, 2004, pages 16-29. doi : 10.1007/978-3-540-28632-5_2. url : https://doi.org/10.1007/978-3-540-28632-5%5C_2 (cf. pages 76, 78).

[6] Sébanjila Kevin Bukasa et al. « Let's shock our IoT's heart : ARMv7-M under (fault) attacks ». In : **Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018**. Sous la direction de Sebastian Doerr et al. ACM, 2018, 33 :1-33 :6. doi : 10.1145/3230833.3230842. url : https://doi.org/10.1145/3230833.3230842 (cf. page 91).

[7] Teodor Calin, Michael Nicolaidis et Raoul Velazco. « Upset hardened memory design for submicron CMOS technology ». In : **IEEE Transactions on nuclear science** 43.6 (1996), pages 2874-2878 (cf. pages 111, 112).

[8] Omar Choudary et Markus G Kuhn. « Efficient template attacks ». In : **International Conference on Smart Card Research and Advanced Applications**. Springer. 2013, pages 253-270 (cf. page 79).

[9] Jean-Michel Cioranesco et al. « Cryptographically secure shields ». In : **2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014, Arlington, VA, USA, May 6-7, 2014**. IEEE Computer Society, 2014, pages 25-31. doi : 10.1109/HST.2014.6855563. url : https://doi.org/10.1109/HST.2014.6855563 (cf. page 97).

[10] Ludovic Claudepierre et al. « TRAITOR : A Low-Cost Evaluation Platform for Multi-fault Injection ». In : **ASSS '21 : Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems, Virtual Event, Hong Kong, 7 June, 2021**. Sous la direction de Weizhi Meng et Li Li. ACM, 2021, pages 51-56. DOI : 10.1145/3457340.3458303. URL : https://doi.org/10.1145/3457340.3458303 (cf. page 84).

[11] Christophe Clavier et Marc Joye. « Universal exponentiation algorithm a first step towards provable SPA-resistance ». In : **International Workshop on Cryptographic Hardware and Embedded Systems**. Springer. 2001, pages 300-308 (cf. page 75).

[12] Amine Dehbaoui et al. « Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES ». In : **2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012**. Sous la direction de Guido Bertoni et Benedikt Gierlichs. IEEE Computer Society, 2012, pages 7-15. DOI : 10.1109/FDTC.2012.15. URL : https://doi.org/10.1109/FDTC.2012.15 (cf. page 89).

[13] M Abdelaziz Elaabid, Sylvain Guilley et Philippe Hoogvorst. « Template Attacks with a Power Model. » In : **IACR Cryptology ePrint Archive** 2007 (2007), page 443 (cf. page 79).

[14] Benedikt Gierlichs et al. « Mutual Information Analysis ». In : **CHES, Proceedings**. Springer, 2008 (cf. page 78).

[15] Yoongu Kim et al. « Flipping bits in memory without accessing them : An experimental study of DRAM disturbance errors ». In : **ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014**. IEEE Computer Society, 2014, pages 361-372. DOI : 10.1109/ISCA.2014.6853210. URL : https://doi.org/10.1109/ISCA.2014.6853210 (cf. page 170).

[16] Paul Kocher et al. « Spectre attacks : Exploiting speculative execution ». In : **Communications of the ACM** 63.7 (2020), pages 93-101 (cf. page 165).

[17] Paul C. Kocher. « Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems ». In : **Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings**. Sous la direction de Neal Koblitz. Tome 1109. Lecture Notes in Computer Science. Springer, 1996, pages 104-113. DOI : 10.1007/3-540-68697-5_9. URL : https://doi.org/10.1007/3-540-68697-5%5C_9 (cf. page 71).

[18] Mark Laps et al. « Capacitors for reduced microphonics and sound emission ». In : **CARTS-CONFERENCE-**. Tome 27. COMPONENTS TECHNOLOGY INSTITUTE INC. 2007, page 207 (cf. page 75).

[19] Ronan Lashermes et al. « A DFA on AES Based on the Entropy of Error Distributions ». In : **2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012**. Sous la direction de Guido Bertoni et Benedikt Gierlichs. IEEE Computer Society, 2012, pages 34-43. DOI : 10.1109/FDTC.2012.18. URL : https://doi.org/10.1109/FDTC.2012.18 (cf. page 94).

[20] Hélène Le Bouder et al. « A template attack against VERIFY PIN algorithms ». In : **SECRYPT 2016**. 2016, pages 231-238 (cf. page 80).

[21] Moritz Lipp et al. « Meltdown : Reading Kernel Memory from User Space ». In : **27th USENIX Security Symposium (USENIX Security 18)**. 2018 (cf. page 164).

[22] Amélie MAROTTA et al. « Characterizing and Modeling Synchronous Clock-Glitch Fault Injection ». In : **Constructive Side-Channel Analysis and Secure Design - 15th International Workshop, COSADE 2024, Gardanne, France, April 9-10, 2024, Proceedings**. Sous la direction de Romain WACQUEZ et Naofumi HOMMA. Tome 14595. Lecture Notes in Computer Science. Springer, 2024, pages 3-21. DOI : 10.1007/978-3-031-57543-3_1. URL : https://doi.org/10.1007/978-3-031-57543-3%5C_1 (cf. page 90).

[23] Thomas S MESSERGES, Ezzy A DABBISH et Robert H SLOAN. « Power analysis attacks of modular exponentiation in smartcards ». In : **International Workshop on Cryptographic Hardware and Embedded Systems**. Springer. 1999, pages 144-157 (cf. page 75).

[24] Alyssa MILBURN, Ke SUN et Henrique KAWAKAMI. « You Cannot Always Win the Race : Analyzing the LFENCE/JMP Mitigation for Branch Target Injection ». In : **CoRR** abs/2203.04277 (2022). DOI : 10.48550/arXiv.2203.04277. arXiv : 2203.04277. URL : https://doi.org/10.48550/arXiv.2203.04277 (cf. page 167).

[25] Nicolas MORO et al. « Electromagnetic fault injection : towards a fault model on a 32-bit microcontroller ». In : **CoRR** abs/1402.6421 (2014). arXiv : 1402.6421. URL : http://arxiv.org/abs/1402.6421 (cf. page 91).

[26] Nicolas MORO et al. « Formal verification of a software countermeasure against instruction skip attacks ». In : **J. Cryptogr. Eng.** 4.3 (2014), pages 145-156. DOI : 10.1007/s13389-014-0077-7. URL : https://doi.org/10.1007/s13389-014-0077-7 (cf. page 96).

[27] Kyle J NESBIT et James E SMITH. « Data cache prefetching using a global history buffer ». In : **10th International Symposium on High Performance Computer Architecture (HPCA'04)**. IEEE. 2004, pages 96-96 (cf. pages 66, 67).

[28] Sébastien ORDAS, Ludovic GUILLAUME-SAGE et Philippe MAURINE. « Electromagnetic fault injection : the curse of flip-flops ». In : **J. Cryptogr. Eng.** 7.3 (2017), pages 183-197. DOI : 10.1007/s13389-016-0128-3. URL : https://doi.org/10.1007/s13389-016-0128-3 (cf. page 89).

[29] Pierre-Yves PÉNEAU et al. « NOP-Oriented Programming : Should we Care ? » In : **IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2020, Genoa, Italy, September 7-11, 2020**. IEEE, 2020, pages 694-703. DOI : 10.1109/EuroSPW51379.2020.00100. URL : https://doi.org/10.1109/EuroSPW51379.2020.00100 (cf. page 91).

[30] Gilles PIRET et Jean-Jacques QUISQUATER. « A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD ». In : **Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings**. Sous la direction de Colin D. WALTER, Çetin Kaya KOÇ et Christof PAAR. Tome 2779. Lecture Notes in Computer Science. Springer, 2003, pages 77-88. DOI : 10.1007/978-3-540-45238-6_7. URL : https://doi.org/10.1007/978-3-540-45238-6%5C_7 (cf. pages 93, 94).

[31] Emmanuel PROUFF et Matthieu RIVAIN. « A Generic Method for Secure SBox Implementation ». In : **Information Security Applications, 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers**. Sous la direction de Sehun KIM, Moti YUNG et Hyung-Woo LEE. Tome 4867. Lecture Notes in Computer Science. Springer, 2007, pages 227-244. DOI : 10.1007/978-3-540-77535-5_17. URL : https://doi.org/10.1007/978-3-540-77535-5%5C_17 (cf. page 81).

[32] Lionel Rivière et al. « High precision fault injections on the instruction cache of ARMv7-M architectures ». In : **IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015**. IEEE Computer Society, 2015, pages 62-67. DOI : 10.1109/HST.2015.7140238. URL : https://doi.org/10.1109/HST.2015.7140238 (cf. page 91).

[33] Alexander Schlösser et al. « Simple Photonic Emission Analysis of AES - Photonic Side Channel Analysis for the Rest of Us ». In : **Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings**. Sous la direction d'Emmanuel Prouff et Patrick Schaumont. Tome 7428. Lecture Notes in Computer Science. Springer, 2012, pages 41-57. DOI : 10.1007/978-3-642-33027-8_3. URL : https://doi.org/10.1007/978-3-642-33027-8%5C_3 (cf. page 75).

[34] Bianca Schroeder, Eduardo Pinheiro et Wolf-Dietrich Weber. « DRAM errors in the wild : a large-scale field study ». In : **Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2009, Seattle, WA, USA, June 15-19, 2009**. Sous la direction de John R. Douceur et al. ACM, 2009, pages 193-204. DOI : 10.1145/1555349.1555372. URL : https://doi.org/10.1145/1555349.1555372 (cf. page 109).

[35] Andre Seznec et Pierre Michaud. « A case for (partially)-tagged geometric history length predictors ». In : **Journal of InstructionLevel Parallelism** (2006) (cf. page 62).

[36] Daniel J Sorin, Mark D Hill et David A Wood. « A primer on memory consistency and cache coherence ». In : **Synthesis lectures on computer architecture** 6.3 (2011), pages 1-212 (cf. pages 131, 133).

[37] Suresh Balakrishnama and Aravind Ganapathiraju. « Linear Discriminant Analysis - A Brief Tutorial ». In : **Institute for Signal and Information Processing, Mississippi State University** (1998) (cf. page 78).

[38] Paul Sweazey et Alan Jay Smith. « A class of compatible cache consistency protocols and their support by the IEEE futurebus ». In : **ACM SIGARCH Computer Architecture News** 14.2 (1986), pages 414-423 (cf. page 135).

[39] Adrian Tang, Simha Sethumadhavan et Salvatore J. Stolfo. « CLKSCREW : Exposing the Perils of Security-Oblivious Energy Management ». In : **26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017**. Sous la direction d'Engin Kirda et Thomas Ristenpart. USENIX Association, 2017, pages 1057-1074. URL : https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang (cf. pages 173, 174).

[40] Thomas Trouchkine et al. « Electromagnetic fault injection against a complex CPU, toward new micro-architectural fault models ». In : **J. Cryptogr. Eng.** 11.4 (2021), pages 353-367. DOI : 10.1007/s13389-021-00259-6. URL : https://doi.org/10.1007/s13389-021-00259-6 (cf. pages 90, 110).

[41] Youssef Souissi, Maxime Nassar, Sylvain Guilley, Jean-Luc Danger and Florent Flament. « First Principal Components Analysis : A New Side Channel Distinguisher ». In : **ICISC**. 2010 (cf. page 78).

[42] Loïc Zussa et al. « Efficiency of a glitch detector against electromagnetic fault injection ». In : **Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014**. Sous la direction de Gerhard P. Fettweis et Wolfgang Nebel. European Design et Automation Association, 2014, pages 1-6. doi : 10.7873/DATE.2014.216. url : https://doi.org/10.7873/DATE.2014.216 (cf. page 98).

Livres

[43] John L Hennessy et David A Patterson. **Computer architecture : a quantitative approach**. Elsevier, 2011 (cf. page 35).

[44] Donald A Neamen. **Semiconductor physics and devices : basic principles**. McGraw-hill, 2003 (cf. page 19).

[45] David A Patterson et John L Hennessy. **Computer organization and design risc-v edition : The hardware software interface**. Elsevier Science & Technology Books, 2017 (cf. pages 35, 52).

[46] Olivier Savry, Thomas Hiscock et Mustapha El Majihi. **Sécurité matérielle des systèmes : Vulnérabilité des processeurs et techniques d'exploitation**. Dunod, 2019 (cf. page 153).

[47] Neil HE Weste et David Harris. **CMOS VLSI design : a circuits and systems perspective**. Pearson Education India, 2015 (cf. page 19).

Autres publications

[48] ANSSI. **Recommandations relatives à la sécurité des (systèmes d') objets connectÉs**. https://www.ssi.gouv.fr/uploads/2021/09/anssi-guide-securite_des_systemes_objets_connectes_iot-v1.0.pdf. 2021 (cf. page 101).

[49] Cédric Archambeau et al. « Template attacks in principal subspaces ». In : **Cryptographic Hardware and Embedded Systems-CHES**. Springer, 2006, pages 1-14 (cf. page 79).

[50] Suresh Chari, Josyula R Rao et Pankaj Rohatgi. « Template attacks ». In : **Cryptographic Hardware and Embedded Systems-CHES 2002**. Springer, 2003 (cf. page 79).

[51] **Cours de Onur Mutlu à l'ETZH**. url : https://youtu.be/h6l9yYSyZHM?feature=shared (cf. page 59).

[52] Stefan Mangard. « A simple power-analysis (SPA) attack on implementations of the AES key expansion ». In : **ICISC 2002**. Springer (cf. page 75).

[53] Elisabeth Oswald et Stefan Mangard. « Template attacks on masking—resistance is futile ». In : **Topics in Cryptology–CT-RSA 2007**. Springer, 2006, pages 243-256 (cf. page 79).

[54] Colin Percival. **Cache missing for fun and profit**. 2005 (cf. page 154).

[55] Christian Rechberger et Elisabeth Oswald. « Practical template attacks ». In : **Information Security Applications**. Springer, 2005, pages 440-456 (cf. page 79).

[56] Robert NM Watson et al. **An introduction to CHERI**. Rapport technique. University of Cambridge, Computer Laboratory, 2019. url : https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf (cf. page 184).

Acronymes

ABI application binary interface. 36
AES advanced encryption standard. 77, 81, 93
ALU arithmetic and logical unit. 57, 169
AMO atomic memory operations. 134
ANSSI Agence nationale de la sécurité des systèmes d'information. 101, 102
API application programming interface. 122
ASID address space identifier. 130
ASLR address space layout randomization. 145

BHT branch history table. 10, 57, 58, 60, 61, 63, 66, 160
BRU branch and repeat unit. 57
BTB branch target buffer. 10, 57, 64, 65

CESTI Centre d'évaluation de la sécurité des technologies de l'information. 101, 102
CFG graphe de flot de contrôle. 91
CPA correlation power analysis. 76, 77
CPU central processing unit. 52, 184
CSR control and status register. 45, 46

DFA differential fault analysis. 93
DMA direct memory access. 123, 135, **Glossaire :** direct memory access
DRAM dynamic random-accesss memory. 106, 110, 123, 170–172
DVFS dynamic voltage and frequency scaling. 173

ECC error correcting code. 110, 172
EM électromagnétique. 10, 84, 85, 89, 90

FIA fault injection analysis. 92
FIB focused ion beam. 72, 97
FPGA field programmable gate array. 84

GHB global history buffer. 10, 66, 67
GHR global history register. 61, 62
GPIO general purpose input/output. 123, **Glossaire :** general purpose input/output
GPR general purpose register. 36, 37

GPU graphics processing unit. 120
GShare globally shared history buffer. 10, 58, 61–63

HDD disque dur. 113, 114, **Glossaire :** disque dur

IBPB indirect branch prediction barrier. 167
IBRS indirect branch restricted speculation. 167
IPC instructions per cycle. 59, **Glossaire :** instructions per cycle
ISA jeu d'instructions. 35, 36, 52, 56, 80, 122, 125, 132, 146, 153, 172, 184, 185, **Glossaire :** jeu d'instructions

KASLR kernel address space layout randomization. 145

LLC last level cache. 116, 120
LRU least recently used. 119
LSU load store unit. 57, 169

MAC message authentication code. 146
MLC multi-levels cell. 108
MMU memory management unit. 119, 128, 129, 172, 185, **Glossaire :** memory management unit
MOSFET metal oxyde semiconductor field effect transistor. 21
MPU memory protection unit. 126–128

PAC pointer authentication code. 146
PC program counter. 37, 41, 52, 65, 91, 125, 168, **Glossaire :** program counter
PHT pattern history table. 58, 61–63, 66, 166
PMA physical memory attributes. 128
PPN physical page number. 128
PTE page table entry. 11, 129
PUF physically unclonable function. 83

RAII ressource acquisition is initialization. **Glossaire :** ressource acquisition is initialization
RAM random-access memory. 43
RAS return address stack. 65
ROB reorder buffer. 57
ROP return-oriented programming. 91
RSB return stack buffer. 10, 37, 57, 65
RTL register-transfer level. 89, 90

SC sequential consistency. 132, 133
SEL single-event latchup. 82
SEU single-event upset. 82, 111
SLC single-level cell. 108
SMT simultaneous multithreading. 131, 168, 169, 185
SoC System-on-Chip. 90, 120, 129, 171, 173
SPA simple power analysis. 10, 75, 76
SRAM static random-access memory. 105, 106, 111
SSD solid-state disk. 108, 113, 114
SWMR single-writer, multiple-readers. 135

TAGE tagged geometric history length branch predictor. 62, 63
TLB translation lookaside buffer. 129, 130, **Glossaire :** translation lookaside buffer

TSO total store order. 133

UART universal asynchronous receiver-transmitter. **Glossaire :** universal asynchronous receiver-transmitter

VPN virtual page number. 128

WCET worst case execution time. 80, **Glossaire :** worst case execution time

Glossaire

alignement L'alignement mémoire décrit la position d'un bloc de donnée en mémoire. Un alignement de 16 octets, par exemple, implique que les données considérées commencent à une adresse qui est un multiple de 16.. 115, 124

architecture de Hardvard L'architecture de Hardvard désigne un système de calcul dont les mémoires pour les données d'une application et celle pour ses instructions sont séparées.. 120

architecture Von Neumann L'architecture de Von Neumann désigne un système de calcul avec une mémoire unique : données et instructions sont mélangées.. 120

backend Dans le contexte de la microarchitecture, le backend désigne la partie du pipeline une fois les instructions *dispatchées*, potentiellement dans le désordre. . 55, 109

CHERI CHERI est une extension de jeu d'instructions, nécessitant une modification du matériel, dédié à la sécurité de la mémoire. Vu du matériel, adresse et pointeur ne sont plus la même chose. Un pointeur, vu par CHERI, est un concept appelé *capabilities* combinant adresse et métadonnées, avec une vérification matérielle de validité. Lire [56] et la sous-section 20.2.6 pour plus de détails.. 138, 146

die Un die est le circuit intégré nu, sans package, après découpe depuis le wafer.. 86, 185

direct memory access L'accès direct à la mémoire (DMA pour *direct memory access*) est un mécanisme permettant des transferts de données entre périphériques, souvent une mémoire et un périphérique de communication, directement sans être gérés par le CPU.. 123, 135

disque dur Le disque dur (*hard-drive disk (HDD)* en anglais) est un dispositif de stockage d'information utilisant une tête de lecture lisant l'information sur des disques encodée sous forme magnétique.. 113

durcissement Le durcissement est le processus de renforcer un système avec des contremesures, un circuit intégré par exemple, ou un programme, pour qu'il résiste à un modèle d'attaquant donné.. 96, 97, 110

general purpose input/output Un GPIO désigne un pin, un fil, directement contrôlable par le processeur, permettant de le mettre à un niveau haut ou bas. Il permet ainsi d'intérargir avec l'environnement en utilisant, par exemple, un protocole non initialement anticipé lors de la conception de la puce.. 123

hart Un *hart* est un *hardware thread*. Les cœurs supportent en général un seul hart, mais certains implémentant le SMT peuvent en avoir plusieurs.. 168, 169

idempotent Une opération idempotente est une opération qui donne le même résultat qu'elle soit exécutée une seule fois ou deux fois successivement : $f = f \circ f$. 91, 96, 124

instructions per cycle Le nombre moyen d'instructions exécutées par cycle est une mesure de la capacité de parallélisme d'un cœur.. 59

jeu d'instructions Le jeu d'instructions est l'interface entre le logiciel et le matériel. Il comprends la sémantiques des instructions supportée par le processeur, ainsi que la description de l'architecture : nombre et taille des registres, configurations possibles du processeurs, ...Ce qui peut être contrôlé par le jeu d'instruction relève de l'architecture. Les éléments matériels qui ne peuvent être contrôlés par le jeu d'instruction relèvent de la microarchitecture. En anglais on parle d'*instruction set architecture* (ISA).. 35, 36, 52, 125, 146

memory management unit La memory management unit (MMU) est le composant matériel en charge de la tradution des adresses virtuelles en adresses physiques.. 119, 128, 129, 172, 185

pilote Un pilote (*driver* en anglais) est un programme dédié au contrôle d'un composant bas niveau : un périphérique, la carte graphique, etc. Le pilote est chargé d'intérargir directement avec le périphérique, via l'écriture et la lecture aux adresses correspondantes. Pour éviter que les applications n'entrent en compétition pour la gestion d'un périphérique, les pilotes sont en général gérés par le noyau du système d'exploitation et isolés des applications à l'aide de la mémoire virtuelle. . 122

program counter Le compteur de programme *program counter*, terme utilisé en général tel quel en anglais, est le registre contenant l'adresse de la prochaine instruction a exécuté. Suivant le jeu d'instructions, ce registre est directement accessible (e.g. ARM), ou non (e.g. RISC-V).. 37, 41, 65, 91, 125, 168

translation lookaside buffer Le translation lookaside buffer (TLB) est une mémoire cache pour la MMU dédiée à la traduction d'adresses virtuelles vers physiques. Sur certains systèmes, il y a une hiérarchie de TLB.. 129, 130

universal asynchronous receiver-transmitter L'*universal asynchronous receiver-transmitter* (UART) est un protocole de communication série, soit un périphérique responsable de l'envoi et de la réception des données pour ce protocole.. 126

wafer Un wafer est un large disque de substrat silicium sur lequel on grave les circuits intégrés. Par extension, on appelle également wafer le disque une fois gravé. Il suffit alors de découper les puces nues, appelées die, puis de les mettre dans un *package*. . 31, 109, 184

worst case execution time Le pire cas de temps d'exécution est le temps maximal pris pour exécuter une certaine fonctionalité. Savoir calculer le WCET est un enjeu important dans les systèmes temps réels.. 80