
MASTER RESEARCH INTERNSHIP

INTERNSHIP REPORT

# Synthèse automatique de micro-architectures pour jeux d'instructions CISC, une étude de cas : le WebAssembly

**Hardware Architecture**

*Author:*
Dylan LEOTHAUD

*Supervisor:*
Steven DERRIEN
Jean-Michel GORIUS
TARAN

**Abstract:** Les systèmes embarqués de type IoT reposent généralement sur l'utilisation de microcontrôleurs à faible consommation d'énergie dont la micro-architecture a été spécialisée. Ces processeurs sont donc conçus manuellement en utilisant des langages de description matériel. La synthèse de haut-niveau permet de concevoir des micro-architectures à partir d'une description comportementale dans un langage de haut-niveau comme le C ou le C++. Néanmoins, les outils de synthèse de haut-niveau actuels ne permettent pas d'obtenir aisément des micro-architectures de processeurs pipelinés à exécution spéculative. SpecHLS est alors une extension aux outils de synthèses de haut-niveau permettant de concevoir de telles micro-architectures à partir d'un simulateur de jeu d'instructions en C ou C++. Cependant, l'évaluation de SpecHLS a été réalisée exclusivement pour le jeu d'instructions RISC-V. Nous souhaitons alors étudier la possibilité d'utiliser SpecHLS pour concevoir automatiquement des processeurs CISC. Nous faisons cela au travers de l'exemple de la synthèse de processeurs WebAssembly afin de faire émerger les enjeux de la conception des processeurs CISC. Nous proposons ensuite des solutions aux défis liés à la conception de processeurs WebAssembly, notamment en proposant des approches de piles matérielles à mettre en œuvre, ainsi qu'une transformation, que nous avons appelé le *unpipeline*, nous permettant de contrôler à une granularité fine l'exécution d'instructions complexes qui semble à première vue ne pas pouvoir s'insérer dans un pipeline classique de processeur.

# Table des matières

# 1 Introduction

Les systèmes embarqués de type IoT (*Internet of Things*) reposent généralement sur l'utilisation de microcontrôleurs à faible consommation d'énergie. La micro-architecture et les jeux d'instructions de ces microcontrôleurs sont spécifiques et doivent donc être conçus manuellement à l'aide de langages de description matériel (HDL, ou *Hardware Description Languages*). La conception d'un processeur à ce niveau d'abstraction est un processus laborieux et complexe. Elle nécessite, de plus, une comprégension préliminaire de la micro-architecture à concevoir.

Dans le but de rendre plus aisé et efficace la conception de processeurs, des outils ont été développés afin d'augmenter le niveau d'abstraction auquel s'effectue la conception. C'est notamment le cas de la synthèse de haut-niveau (HLS, ou *High-Level Synthesis*). Néanmoins, les outils de HLS actuels ne permettent pas d'obtenir aisément des micro-architectures de processeurs pipelinés spéculatifs efficaces. SpecHLS [1] est alors une extension des outils de HLS offrant la capacité à concevoir de tels processeurs à partir d'une description comportementale du processeur. SpecHLS permet de synthétiser automatiquement des processeurs pipelinés spéculatifs à jeu d'instructions RISC. Les performances et le coût en surface de ces processeurs sont comparables aux conceptions manuelles.

Cependant, il convient de souligner que l'évaluation de SpecHLS a été réalisée exclusivement avec la conception de processeurs RISC-V [1]. Nous souhaitons alors concevoir également des processeurs CISC, dont les instructions sont plus complexes. Pour atteindre ce but, nous devrons alors identifier les mécanismes avec lesquels étendre SpecHLS. Nous avons décidé d'étudier la faisabilité de conception de processeurs CISC au travers de l'exemple de la conception de processeurs WebAssembly.

Les contributions de ce stage sont alors :

— Une implémentation de processeur dont le jeu d'instructions est un sous-ensemble du WebAssembly. Cette implémentation est fonctionnelle et peut-être utilisé sur FPGA, mais elle représente un processeur non spéculatif et le programme permettant la synthèse de ce processeur a été manuellement conçu et optimisé dans ce but. Nous avons utilisé ce processeur afin d'effectuer une étude de faisabilité de la conception de processeurs CISC en HLS, ainsi qu'une mise en évidence des enjeux de la conception de ces processeurs.

— WebAssembly étant un langage utilisant une pile d'opérande au lieu d'une file de registre, nous proposons des approches de mise en œuvre de piles matérielles adaptées à un processeur pipeliné en HLS.

— Une transformation de programme appliquée en amont de SpecHLS, le *unpipeline*, nous permettant d'obtenir un contrôle fin de l'exécution des instructions, et ains nous permet par conséquence d'ajouter, aux processeurs conçus par SpecHLS, des instructions complexes, ne s'insérant pas dans un pipeline classique de processeurs.

La Section 2 évoque les motivations et le contexte du stage. La Section 3 présente la micro-architecture des processeurs utilisés dans l'IoT, ainsi que la conception automatique de micro-architectures pipelinées efficaces au travers d'outils de synthèses de haut-niveau. La Section 4 étudie les particularités du langage WebAssembly afin d'identifier les défis liés à la conception de processeurs WebAssembly. Enfin, la Section 5 explique la contribution principale de ce stage : la transformation du *unpipeline*.

# 2 Motivation et contexte

Les systèmes embarqués sont généralement soumis à des contraintes de performances, de ressources et de consommation d'énergie. À cause de ces contraintes, les approches classiques permettant d'exécuter un programme (interprétation, compilation et compilation à la volée) ne sont pas toujours adaptées. Nous pouvons alors se tourner vers l'utilisation d'un support matériel spéci-

fique pour l'exécution de programmes. Ce support matériel peut prendre la forme d'un processeur dont le jeu d'instructions a été choisi spécifiquement pour permettre au système de respecter les contraintes. Ce processeur peut également être spécialisé à l'exécution de programmes écrits dans un langage de haut niveau afin de permettre l'écriture plus efficace de programmes.

La Section 2.1 rappelle les différentes techniques utilisées pour exécuter des programmes. La Section 2.2 présente les différents types de jeux d'instructions. La Section 2.3 présente un exemple de processeurs exécutants un langage de haut niveau, des processeurs Java. Enfin, la Section 2.4 évoque les objectifs du stage.

## 2.1 Exécution d'un programme

L'exécution d'un programme peut s'effectuer au travers de différentes approches. Le choix de celle-ci a une influence sur le temps d'exécution et un impact sur l'usage de la mémoire ou encore la consommation d'énergie d'un programme. De manière classique, un programme peut être interprété, compilé ou compilé à la volée. Mais il peut également être exécuté à l'aide d'un support matériel spécifique. Ces différentes approches sont détaillées dans cette section.

### Interprétation

L'interprétation repose sur un programme, appelé *interpréteur*, qui analyse la syntaxe d'un programme afin de le comprendre et de l'exécuter. Le seul programme qui s'exécute sur l'ordinateur est l'interpréteur, mais le comportement observé est celui du programme interprété. Cette approche est simple à mettre en œuvre mais généralement peu efficace en temps d'exécution [2].

### Compilation

La compilation (AoT, ou *Ahead-of-Time*) consiste à transformer le programme avant l'exécution vers un exécutable dit *natif*, qui pourra être exécuté par l'ordinateur de l'utilisateur. Cette méthode permet d'obtenir un exécutable optimisé efficace, mais il perd alors sa portabilité.

### Compilation à la volée

La compilation à la volée (JiT, ou *Just-in-Time*) consiste à compiler le code source vers du code machine exécutable nativement pendant l'exécution. Cette méthode permet d'exécuter du code optimisé et efficace comme pour la compilation, et préserve la portabilité. Cette approche offre un bon niveau de performance, mais n'est pas adaptée à des systèmes très contraints en mémoire (par exemple l'IoT [3]).

### Support matériel spécifique

L'utilisation d'un support matériel spécifique pour l'exécution d'un programme lui permet de s'exécuter très efficacement. Pour obtenir ce résultat, une phase préalable de conception de la micro-architecture spécialisée est alors nécessaire avant l'exécution. Cette micro-architecture spécialisée peut être conçue par exemple à l'aide d'un langage de description matériel (HDL, ou *Hardware Description Language*), ou de la synthèse de haut-niveau (HLS, ou *High-Level Synthesis*) qui permet la création de circuits matériels à partir de descriptions écrites en C ou en C++.

Ainsi, dans le cas de l'IoT, soumis à des contraintes d'efficacité et d'usage mémoire, les approches d'exécution classiques peuvent ne pas convenir. Dans ce cas, l'utilisation d'un support matériel spécifique peut être la seule solution possible. Ce support matériel peut alors prendre la forme d'un processeur avec un jeu d'instructions choisi spécifiquement pour rendre ce programme plus efficace en mémoire ou en temps.

## 2.2 Jeux d'instructions

La micro-architecture d'un processeur dépend du jeu d'instructions qui lui est associée. Pour limiter l'influence du jeu d'instructions, un processeur peut être micro-codé, c'est-à-dire qu'il traduis les instructions vers un micro-code plus simple avant l'exécution. Cependant, la micro-codage n'est pas toujours adapté, car il nécessite un usage plus important en ressources matérielles et donc en coût et en consommation d'énergie. Dans le cas de l'IoT, le choix du jeu d'instructions a donc un impact important sur la micro-architecture du processeur. Il existe deux familles de jeux d'instructions couramment utilisées pour les processeurs modernes : les processeurs CISC (*Complex Instruction Set Computer*) et les processeurs RISC (*Reduced Instruction Set Computer*).

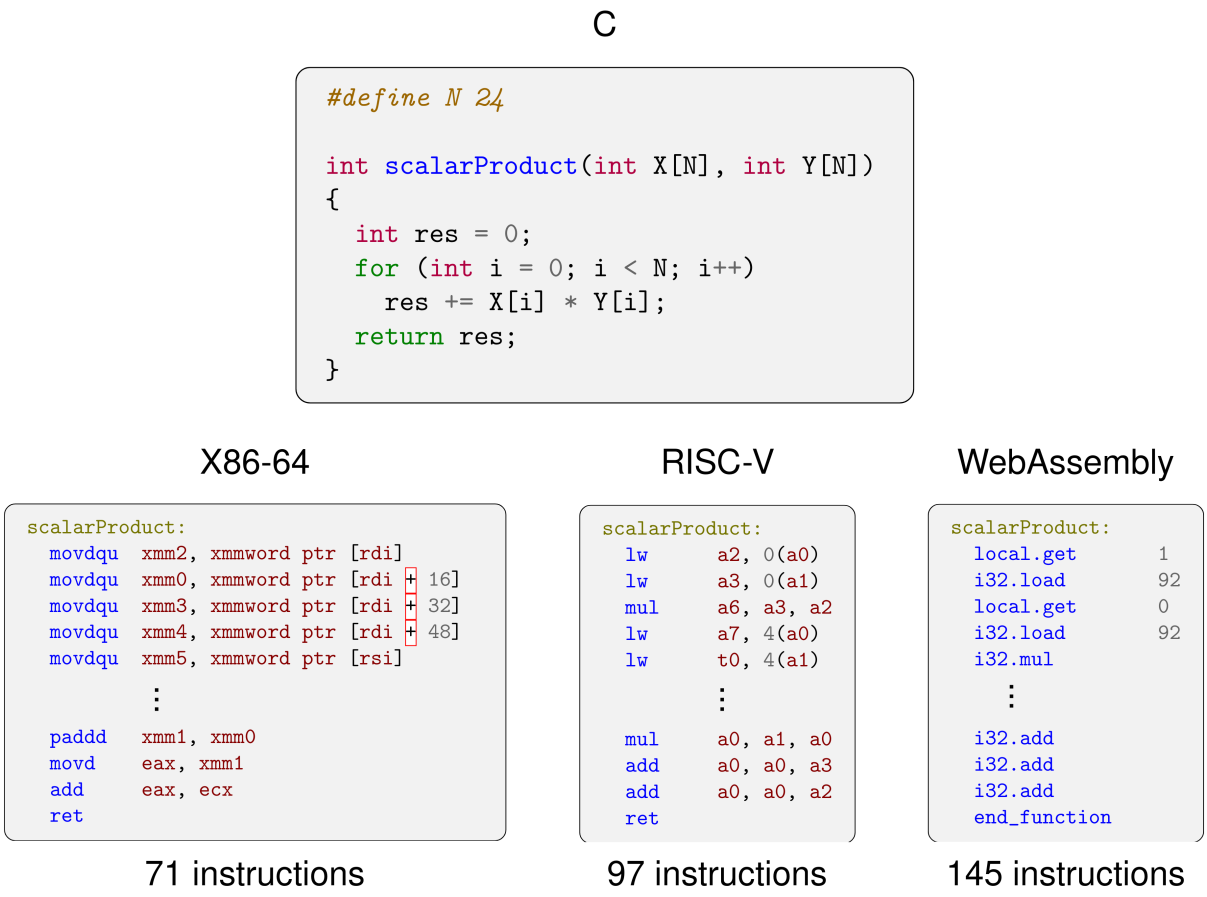



FIGURE 1 – Exemple de programme compilé vers différentes cibles de compilation.

### Jeux d'instructions CISC et RISC

Les processeurs CISC exécutent des instructions complexes, leur micro-architecture peut donc l'être également. La micro-architecture de ces processeurs nécessite l'utilisation d'un grand nombre de ressources matérielles. De plus, la latence d'exécution des instructions est élevée puisque les instructions peuvent effectuer un grand nombre d'opérations élémentaires. Le nombre de ces instructions peut également être grand, complexifiant d'autant plus la micro-architecture. Les instructions RISC sont plus simples et n'effectuent toutes que peu d'opérations élémentaires par instructions. Cela rend le pipeline de processeurs (comme expliqué dans la Section 3.1) RISC plus simple et les latences des instructions sont plus faibles.

D'autres types de jeux d'instructions existent. Notamment des bytecodes ayant pour but d'être

exécutés sur une machine à pile tel que les bytecodes Java et WebAssembly. Une différence notable entre les jeux d'instructions RISC, CISC et le bytecode WebAssembly est que, en tirant partie des instructions effectuant des tâches complexes, les programmes assembleurs CISC sont généralement plus simple, tandis que les programmes assembleurs RISC sont plus long, car doivent détailler les tâches complexes en une suite d'instructions simples. En revanche, un programme WebAssembly peut tirer parti des instructions complexes également, mais sera tout de même long, car des instructions de manipulation de la pile d'opérandes doivent être ajouté au programme. Un exemple de programme compiler vers des jeux d'instructions CISC (x86-64), RISC (RISC-V) et WebAssembly sont présenté sur la Figure 1.

#### Bytecode WebAssembly

Le bytecode WebAssembly possède des instructions de taille variable. Il inclut par exemple des instructions d'opérations vectorielles, et d'autres servants à la gestion du flot de contrôle. Il a pour but de s'exécuter sur une machine virtuelle à pile, isolé de tout autre programme, en s'exécutant dans un *bac à sable*, pour la sécurité.

#### Bytecode Java

Le bytecode Java quant à lui est un jeu d'instructions dont les instructions sont toutes codées sur un octet. Les instructions Java peuvent par exemple servir à la gestion des types dynamiques Java. Bien que ce jeu d'instructions soit prévu pour s'exécuter sur une machine virtuelle, des supports matériels spécifiques existent, dont des processeurs Java.

### 2.3 Processeurs Java

Le langage Java est depuis longtemps plutôt populaire auprès des programmeurs. De plus, certains aspects de ce langage, tel que le *garbage collector*, ou les références, qui sont sûres comparées à l'utilisation de pointeur en C, lui donnent le potentiel de rendre simple et sûr la conception de programme pour les systèmes embarqués [4]. Cependant, son exécution basée sur une machine virtuelle impact fortement son utilisation mémoire et la rapidité d'exécution des programmes.

Afin d'améliorer cela, une solution est d'utiliser des supports matériels spécifiques à l'exécution d'une machine virtuelle Java. Ces supports matériels prennent la forme de processeur exécutant nativement du bytecode Java. La conception de processeur Java n'est pas une idée récente, Sun Microelectronics a proposé dès 1997 un processeur appelé picojava [2]. D'autres essaies de processeurs Java ont été effectués, avec JOP en 2007 [4] ou encore JAIP en 2015 [5]. Certains processeurs ARM étaient également pourvus d'une extension appelée Jazelle ayant pour but d'exécuter en matériel certaines instructions Java.

Le but de ces processeurs Java est de permettre l'implémentation de machine virtuelles plus efficaces, notamment en accélérant le *garbage collector* et en ajoutant un support matériel pour le multi-threading. En revanche, certaines instructions très complexes n'arrivent en pratique que rarement. Par exemple, l'instruction `new` n'arrive que 0,4% du temps [2]. Ces instructions peuvent alors être traitées en logiciel, permettant l'utilisation de micro-architectures de processeur plus simple tout en impactant que faiblement les performances.

Ainsi, la conception de processeurs dont le jeu d'instructions est supposé s'exécuter sur une machine virtuelle à pile n'est pas nouvelle, mais ils ont tous été conçus manuellement, contraignant les concepteurs à devoir connaître avec exactitude la micro-architecture au moment de la conception. La capacité de pouvoir concevoir automatiquement de tels processeurs pourrait rendre la conception plus efficace.

### 2.4 Objectifs du stage

Nous souhaitons permettre la conception automatisée de processeurs CISC grâce à SpecHLS et la HLS. Cependant, SpecHLS permet de concevoir des processeurs RISC efficaces mais ne permet pas de concevoir de processeurs CISC. Nous allons alors, au travers d'un étude de faisabilités de processeurs WebAssembly en HLS, délimiter précisément les enjeux de conceptions automatique de processeurs CISC.

Les objectifs de ce stage sont d'identifier les transformations qui sont nécessaire à appliquer à un simulateur de jeu d'instructions afin d'être capable de concevoir automatiquement des processeurs CISC, voir WebAssembly, grâce à SpecHLS. En effet, les jeux d'instructions CISC possèdent des instructions complexes, c'est-à-dire qui ne rentre pas dans un pipeline classique de processeur. Pour atteindre ce but, nous avons fait le choix d'effectuer une étude de faisabilité d'un processeur WebAssembly en utilisant SpecHLS, nous avons donc implémenter un processeur dont le jeu d'instruction est un sous-ensemble du WebAssembly afin de comprendre quelles sont les transformations nécessaire pour obtenir le programme permettant la synthèse de ce processeur à partir d'un simulateur de jeu d'instructions.

Pour arriver à notre but, il a tout d'abords été nécessaire de comprendre la micro-architecture des processeurs pouvant être conçu grâce à SpecHLS, ainsi que le fonctionnement des outils permettant leurs synthèses (Section 3). Puis, nous avons étudier le langage WebAssembly afin de mettre en avant les différences fondamentales entre le WebAssembly et les jeux d'instructions permettant la synthèse de processeurs avec SpecHLS (Section 4). Finalement, nous proposons une transformation, s'effectuant au niveau source, permettant d'inclure des instructions complexes dans des processeurs conçu avec SpecHLS (Section 5).

## 3 État de l'art

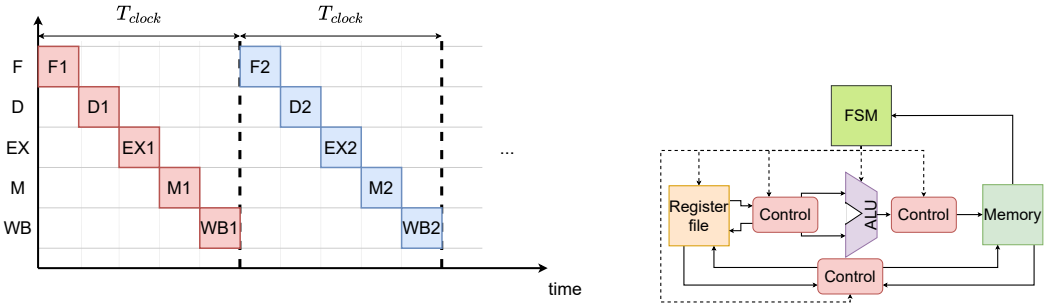
Afin de permettre la synthèse automatique de micro-architecture pipelinées de processeurs pour l'IoT, il est nécessaire de comprendre le fonctionnement des micro-architectures de tels processeurs (Section 3.1), ainsi que les enjeux de conception de ces micro-architectures (Section 3.2).

### 3.1 Micro-architectures de processeurs pour l'IoT

Dans les systèmes embarqués, notamment pour l'IoT, des contraintes fortes restreignent les processeurs utilisés. Les systèmes peuvent par exemple avoir une limite de consommation d'énergie ou de coût. Cela impose à la micro-architecture des processeurs d'être simple, mais efficace. Les micro-architectures traditionnellement utilisées pour ce type de produit sont des processeurs *in-order* spéculatif, qui offre un compromis entre efficacité et coût intéressant.

Un processeur exécute un programme, qui est une suite d'instructions contenues en mémoire. Pour cela, différentes approches existent, nous commençons par détailler les processeurs micro-codés qui sont simples, puis les processeurs pipelinés à exécution dans l'ordre qui sont les processeurs produit par SpecHLS.

### Processeurs micro-codés




(a) Trace d'exécution sur un processeur non pipeliné.

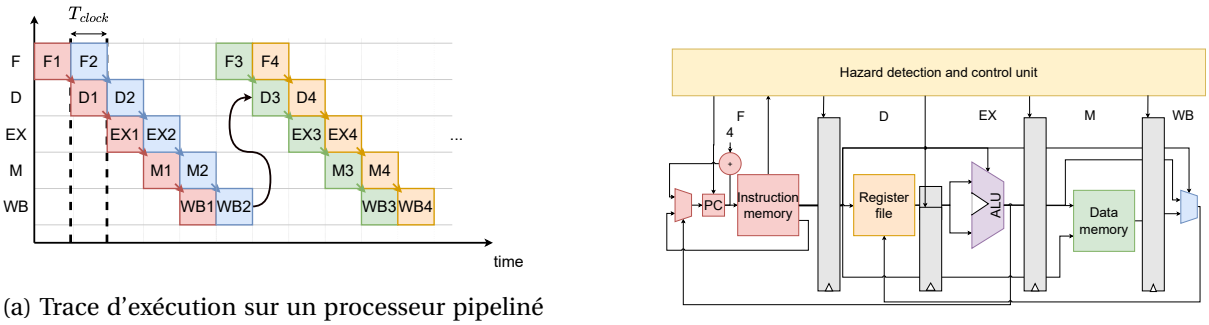
(b) Processeur micro-codé.

FIGURE 2 – Trace d'exécution et chemin de données d'un processeur micro-codé.

Lors de son fonctionnement, un processeur récupère dans la mémoire une instruction à exécuter et la décode pour obtenir l'opération à effectuer et les opérandes. Une fois cette instruction connue, elle peut être micro-codée, c'est-à-dire transformée en une suite d'instructions simples. Afin de micro-coder une instruction, les instructions sont lues par une machine à états (FSM, ou *Finite-State Machine*), qui peut alors contrôler les différents composants du processeur pour effectuer la suite d'opérations élémentaires nécessaire à l'exécution de l'instruction. Le comportement temporel de l'exécution des instructions peut alors être représenté par la trace de la Figure 2a. Un chemin de données correspondant à un processeur micro-codé est représenté sur la Figure 2b.

Pour améliorer l'efficacité d'un processeur, il est souvent possible de commencer l'instruction suivante avant la fin de l'instruction en cours. On parle alors de processeurs à exécution pipeliné [6].

### Processeurs *in-oder* pipelinés




(a) Trace d'exécution sur un processeur pipeliné avec dépendances de données.

(b) Processeur pipeliné.

FIGURE 3 – Trace d'exécution et chemin de données d'un processeur pipeliné.

Dans un processeur pipeliné, le chemin de données exécutant les instructions est séparé en différentes parties distinctes séparées par des registres. Ces parties de chemin de données correspondent aux différents étages du pipeline.

Ainsi, à chaque cycle d'horloge, les différents étages du pipeline peuvent s'exécuter en parallèle sur des données différentes, et donc, cela permet de commencer une instruction avant que la précédente ne soit terminée. Pour permettre cela, le processeur *spécule* que les instructions ne sont pas des branchements, c'est à dire qu'il commence l'instruction suivante en mémoire avant de savoir si elle doit réellement s'exécuter, puis annule son exécution si elle n'aurait pas dû s'exécuter. Cependant,

il existe des dépendances entre certaines instructions et la sémantique du programme peut ne pas être respecté si l'exécution des instructions se superpose toujours. Il est donc nécessaire d'ajouter au processeur une unité de contrôle et de détection des aléas permettant aux instructions d'attendre la résolution des aléas avant de s'exécuter. Par exemple, si une instruction utilise le résultat de la précédente, elle ne peut pas s'effectuer avant la résolution de ses opérandes.

Un chemin de donnée d'un processeur pipeliné est représenté sur la Figure 3b. L'exécution d'une instruction s'effectue en plusieurs étapes, les données transitent par des registres entre les différents étages du pipeline. Ainsi, lorsqu'une instruction passe à l'étape suivante, une autre instruction peut commencer à s'exécuter. Une trace d'exécution est présentée sur la Figure 3a, où la troisième instruction dépend de la deuxième. Un tel processeur respecte la sémantique séquentielle des programmes, on parle de processeur à exécution dans l'ordre, ou de processeur *in-order*.

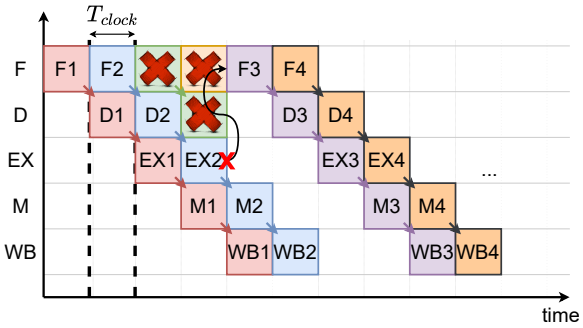



FIGURE 4 – Trace d'exécution sur un processeur pipeliné spéculatif avec misprédiction.

Il est possible d'améliorer l'efficacité des processeurs pipelinés en permettant au processeur de prédire le comportement le plus probable en cas de branchement. Ainsi, en cas de bonne prédiction, cette prédiction aura permis au processeur d'anticiper les instructions à exécuter et donc à commencer leur exécution au plus tôt. En cas de mauvaise prédiction, les instructions prédites en cours d'exécution devront être annulées afin d'obtenir le même comportement que si elles ne s'étaient pas exécutées. Une trace d'exécution spéculative est présentée sur la Figure 4.

D'autres versions de processeurs existent également, tels que des processeurs exécutant les instructions dans le désordre, dit processeurs *out-of-order*, ou encore exécutant plusieurs instructions en parallèle, dit processeurs *superscalaires*. Ces processeurs sont généralement plus performants, mais leurs chemins de données sont plus complexes et sont moins efficace en énergie.

## 3.2 Synthèse automatique de circuits matériels

La conception de micro-architectures peut être complexe et nécessiter beaucoup de temps et d'efforts. Des solutions sont alors développées pour faciliter et raccourcir la conception de micro-architectures, telle que la synthèse de haut-niveau (Section 3.2.1). La synthèse de haut-niveau permet d'abstraire certains détails de la conception, en reposant sur un langage de programmation utilisé également pour la conception de logiciels, comme le C ou le C++. Cependant, celle-ci ne permet pas toujours d'obtenir des résultats aussi performant que ceux obtenue avec une conception manuelle. Par exemple, l'ordonnancement est statique et donc nécessairement pessimiste. Lors du pipeline de boucle, la latence de la boucle est alors constante. La latence du corps de la boucle doit donc être au moins aussi grande que la latence du plus long chemin d'exécution possible. Des micro-architectures spéculatives peuvent tout de même s'adapter aux latences réelles lors de l'exécution et donc permettre une exécution généralement plus efficace. Des solutions ont alors été développées pour permettre automatiquement le pipeline de boucle spéculatif (Section 3.2.2) en utilisant la syn-

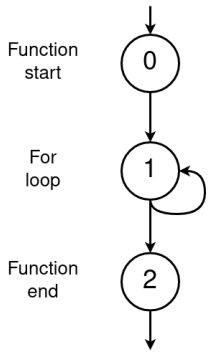
thèse de haut-niveau. Ces solutions peuvent alors également servir à la conception automatique de processeurs pipelinés à partir d'un simulateur de jeu d'instructions (Section 3.2.3).

### 3.2.1 Synthèse de haut-niveau

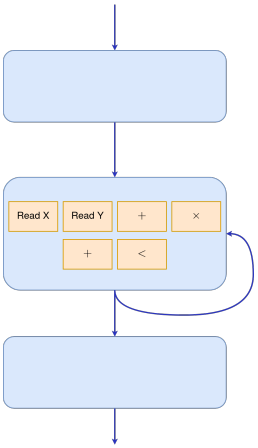
La description de micro-architecture est habituellement effectuée en utilisant des langages de description matérielle (HDL, ou *Hardware Description Languages*). Ces langages permettent de décrire le comportement des composants de la micro-architecture voulue. La micro-architecture décrite pourra ainsi par la suite être synthétisée et implémentée sur un circuit logique programmable (FPGA, ou *Field-Programmable Gate Array*) ou permettre à la création d'un circuit imprimé. La conception d'une micro-architecture en utilisant un langage de type HDL n'est pas une tâche facile. Les HDL sont des langages permettant de décrire une micro-architecture qui doit être fixée au préalable. L'exploration de différente micro-architectures avec des langages HDL requiert alors l'implémentation de la description de chacune de ces micro-architectures.

```
int scalarProduct(int X[N], int Y[N])
{
  int result = 0;
  for (int i = 0; i < N; i++)
  {
    result += X[i] * Y[i];
  }
  return result;
}
```

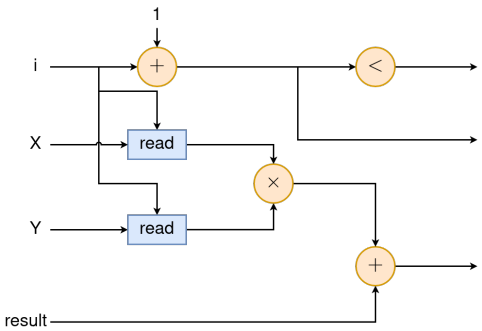
(a) Code d'une fonction effectuant un produit scalaire.



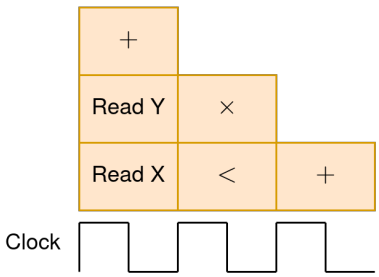

(b) Machine à états finis extrait du code de la Figure 5a.




(c) Machine à états finis annotés avec les opérations extrait du code de la Figure 5a.




(d) Chemin de données du code de la boucle du code de la Figure 5a.




(e) Ordonnancement possible des opérations de la boucle du code de la Figure 5a.

FIGURE 5 – Différentes étapes de la synthèse de haut-niveau d'un produit scalaire.

Pour aider à la conception de micro-architectures, des outils augmentant le niveau d'abstraction des langages utilisés existent. Ces outils permettent ainsi de faciliter l'écriture de code. La synthèse de haut-niveau (HLS, ou *High-Level Synthesis*) permet la conception de micro-architectures en utilisant

des langages comme le C ou le C++ auxquels il est possible d'ajouter des directives afin d'aider à la synthèse. La connaissance du circuit n'est alors plus un prérequis à la conception d'une micro-architecture puisque celle-ci est inférée à partir du code C.

Pour transformer un code comme celui présenté sur la Figure 5a, la HLS effectue plusieurs étapes [7]. Une machine à états est extraite du code (Figure 5b). Les états de cette machine à états correspondent aux différents comportements possibles du code, tels que l'exécution d'une boucle par exemple. Par la suite, les différentes opérations à effectuer sont associées à chaque état de la machine à états (Figure 5c). Ces opérations peuvent correspondre à des opérations arithmétiques ou à des accès à des blocs mémoires par exemple. Les différentes opérations doivent ensuite être ordonnancées entre elles. Pour cela, il est nécessaire de connaître les dépendances de données. Ces dépendances sont visibles sur un chemin de données par exemple (Figure 5d). Les opérations sont alors ordonnancées, la HLS choisit quelles opérations s'effectuent à quel cycle d'horloge (Figure 5e). La latence du circuit est alors connue à cette étape. Dans notre exemple, la latence de la boucle est de 3 cycles. Différents ordonnancements sont possibles pour le même chemin de données, le choix de celui-ci a une influence sur le nombre de ressources à utiliser ou la fréquence maximale par exemple. En effet, en cassant un chemin de données en deux étapes, le chemin critique devient plus court, permettant ainsi d'augmenter la fréquence, ou peut aussi être possible de partager un opérateur entre plusieurs opérations, cependant le nombre de cycles d'horloge nécessaire au même calcul devient plus grand. Des contraintes peuvent limiter le choix de l'ordonnancement, comme le nombre de ports d'accès à une mémoire ou encore le nombre d'opérateur disponible. Finalement, les opérations sont placées sur les opérateurs réels, cette étape permet de partager un opérateur entre plusieurs opérations si celles-ci ont été ordonnancées à des cycles différents, ou alors de permettre le pipeline de boucle, s'il n'y a pas de dépendances et si les opérateurs ont été allouées correctement. Le nombre de cycles d'horloge nécessaires à l'exécution d'une itération est appelée la *latence*, le nombre de cycle d'horloge nécessaire avant de débuter l'itération suivante est appelée l'*intervalle d'initiation* (II, ou *Initiation Interval*) de la boucle. Un ordonnancement pipeliné de cette boucle est présenté sur la Figure 6.

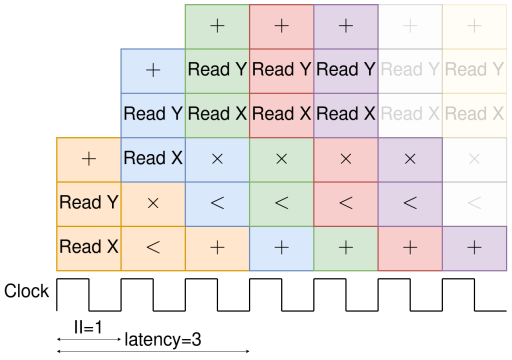



FIGURE 6 – Ordonnancement pipeliné de la boucle du code de la Figure 5a.

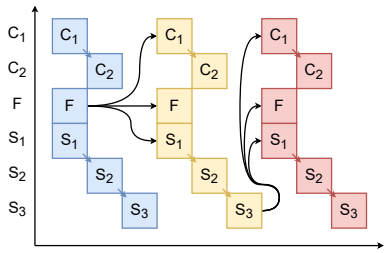
La synthèse de haut-niveau permet de transformer la description d'un comportement en C ou C++ vers une micro-architecture décrite en HDL. La HLS impose des contraintes sur la description comportementale à transformer. Il est par exemple impossible d'utiliser un allocateur dynamique de mémoire puisque le résultat de la HLS est un circuit matériel. La mémoire est allouée sur des composants physiques tels que des blocs mémoire ou encore des registres.

Un programme C ou C++ décrivant un comportement voulu peut donc permettre d'obtenir une micro-architecture fonctionnelle en utilisant la HLS. Cependant, obtenir une micro-architecture optimisée n'est pas évident. Par exemple, l'ordonnancement est statique, c'est-à-dire totalement décidé au moment de la compilation. Ainsi, si l'exécution peut avoir deux comportements possibles, la latence prise en compte sera donc toujours la plus grande latence possible sans s'adapter à ce qui

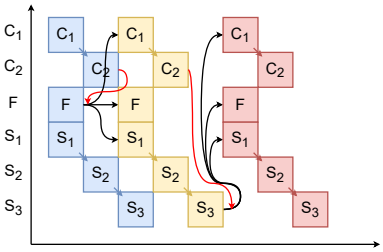
s'exécute réellement.

```
while (1) {
  if (C(x)) // 2 cycles
    x = F(x); // 1 cycle
  else
    x = S(x); // 3 cycles
}
```

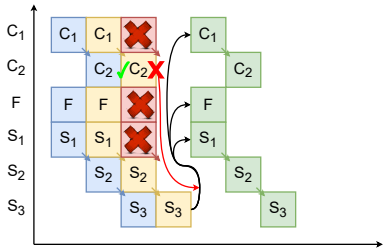
(a) Exemple de code de traitement de données.




(b) Ordonnancement statique.




(c) Ordonnancement dynamique.




(d) Ordonnancement spéculatif.

FIGURE 7 – Code de traitement de données et ordonnancements possibles.

L'ordonnancement de différentes opérations pourrait cependant s'effectuer pendant l'exécution d'un programme en s'adaptant à l'exécution en cours, on parle d'*ordonnancement dynamique*. Dans le code de la Figure 7a, la boucle exécutée peut appeler soit la fonction *F*, qui est rapide, soit appeler la fonction *S*, qui est lente. La décision du chemin pris par l'exécution dépend des données et ne peut pas être choisi pendant la compilation. La HLS utilise un ordonnanceur statique, le temps entre deux itérations est toujours celui du chemin le plus long. Même si seul le chemin court est utile, la micro-architecture obtenue grâce à la HLS devra toujours attendre la fin du calcul de la fonction *S*.

En revanche, avec un ordonnancement dynamique, le temps avant le début de l'itération suivante dépend du calcul de la condition *C*, connu au moment de l'exécution. Les différences entre les ordonnancements statique et dynamique sont présentées sur les Figures 7b et 7c.

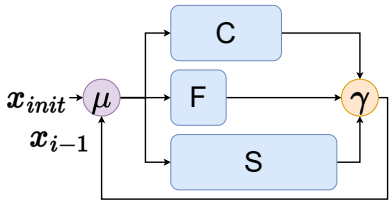
Il est possible d'aller plus loin et de tirer parti d'un pipeline de boucle spéculatif dont l'exécution consiste à présumer de quel sera le résultat du calcul de la condition afin de commencer au plus tôt les prochaines itérations. Si l'hypothèse consistant à choisir le chemin court était fausse, le circuit devra revenir dans le dernier état correct. Une trace d'exécution spéculative de la boucle est présentée sur la Figure 7d.

### 3.2.2 Pipeline de boucle spéculatif

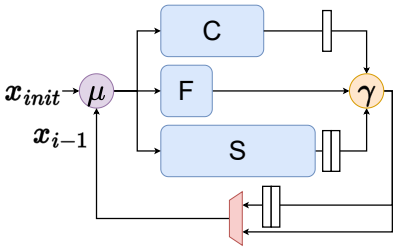
Afin d'obtenir une micro-architecture efficace, nous voulons obtenir un pipeline de boucle spéculatif, comme présenté sur la Figure 7d, lors de la synthèse des programmes conçu avec SpecHLS par la HLS.

```
#pragma HLS distance mis_x=3
#pragma HLS distance ctrl=2
while (1) {
#pragma HLS pipeline II=1
    ctrl[t] = C(s_x[t-1]);
    mis_x[t] = S(s_x[t-1]);
    cs = nextstate(cs, ctrl[t-2]);
    s_x[t] = cs.selSlow ? mis_x[t-3] : F(s_x[t-1]);
    if (cs.rollback)
        s_x[t] = s_x[t-3];
    if (cs.commit)
        x = s_x[t-3];
    t++;
}
```

(a) Code transformé pour permettre la spéculation.




(b) Représentation sous la forme Gated-SSA.




(c) Représentation sous la forme Gated-SSA transformé pour permettre la spéculation.

FIGURE 8 – Différentes représentations du code de la Figure 7a.

**Pipeline spéculatif**

Malgré l'ordonnancement statique de la HLS, il est possible d'obtenir un pipeline de boucle spéculatif efficace en modifiant la description comportementale du processeur [8]. Pour obtenir ce résultat, il est nécessaire d'ajouter à la boucle le contrôle explicite du pipeline. Pour cela, il faut parcourir les chemins court et long en parallèle, en spéculant que le chemin choisi est le plus court afin de commencer les itérations suivantes avant la résolution du choix du chemin. Si le chemin devait être le long, il suffit alors de retourner dans le dernier état correct. Le code de la Figure 7a, transformé afin de permettre la spéculation, est présenté sur la Figure 8a. L'ajout du contrôle du pipeline de boucle spéculatif au code permet d'obtenir un pipeline spéculatif efficace, mais nécessite de modifier la description de chaque boucle pour y ajouter ce contrôle. Cela nécessite alors un travail important pour obtenir ce résultat. Ces transformations de code peuvent être automatisées afin de faciliter la conception de micro-architectures pipelinées, notamment grâce à l'outil SpecHLS [9, 10].

**Représentation de programme Gated-SSA**

Pour automatiser le pipeline de boucle spéculatif, SpecHLS transforme la description du comportement du processeur sous la forme *Gated-SSA* [11], qu'il peut alors transformer. Il s'agit d'une représentation des programmes où les variables sont à affectation unique (SSA, ou *Static single assignment form*). Dans la forme SSA, le choix entre deux variables venant de chemins différents est représenté par des nœuds $\varphi$. Dans la forme Gated-SSA, ces nœuds $\varphi$ sont explicités en nœuds $\mu$ ou $\gamma$. Les nœuds $\mu$ sont utilisés dans les têtes de boucles pour choisir entre la valeur initiale d'une variable et celle calculée pendant l'exécution de la boucle. Les nœuds $\gamma$ sont utilisés pour la jonction conditionnelle de deux chemins de données différents. Une représentation sous la forme Gated-SSA du code de la Figure 7a est représenté dans la Figure 8b. Une version étendue de Gated-SSA est utilisée, ajoutant des nœuds $\alpha$ utilisé lors de l'affectation d'une donnée dans un tableau. Les tableaux

sont considérés comme des valeurs à part entière et affecter une donnée dans un tableau par un nœud $\alpha$ crée un nouveau tableau modifié.

#### Mise en œuvre de spéculation

Grâce à la forme Gated-SSA utilisée pour le code, la spéculation peut automatiquement être mise en œuvre. Cette spéculation s'effectue au niveau des nœuds $\gamma$, car ces nœuds correspondent à un choix entre plusieurs chemins de données. De plus, les nœuds $\gamma$ permettant à la spéculation de rendre la micro-architecture plus efficace sont ceux fusionnant des chemins de longueurs déséquilibrées, tels que le chemin le plus court aies la plus grande probabilité de s'exécuter. Les étapes permettant la spéculation sont :

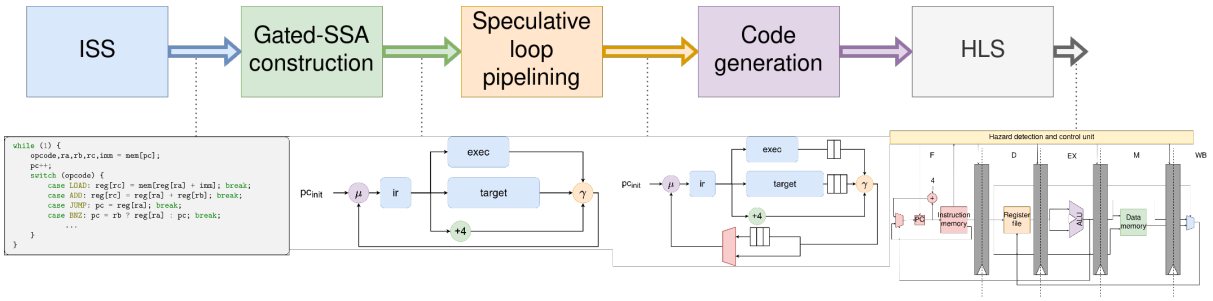
— L'ajout de tampons de données permettant l'accès à des valeurs calculées plusieurs cycles auparavant.
— L'ajout de composants permettant la restauration des valeurs en cas de mauvaise spéculation. Ces systèmes consistent à l'affectation conditionnelle à une variable de : soit la valeur actuelle, soit une valeur précédente qui a été stockée dans un tampon de données.
— L'affectation des valeurs dans les variables présentes dans la description initiales du comportement après validation que la valeur soit correcte.
— L'ajout d'une machine à état contrôlant les restaurations d'états et les validations de l'affectation des variables.

Il est alors possible de générer du code C ou C++ correspondant au circuit après ces transformations qui pourra ensuite générer, grâce à la HLS, une micro-architecture pipelinée spéculative.

### 3.2.3 Synthèse automatique de micro-architectures pipelinées spéculatives

```
while (1) {
    opcode,ra,rb,rc,imm = mem[pc];
    pc++;
    switch (opcode) {
        case LOAD: reg[rc] = mem[reg[ra] + imm]; break;
        case ADD: reg[rc] = reg[ra] + reg[rb]; break;
        case JUMP: pc = reg[ra]; break;
        case BNZ: pc = rb ? reg[ra] : pc; break;
            ...
    }
}
```

(a) Simulateur de jeu d'instruction.




(b) Flot de compilation de processeur pipeliné spéculatif.

FIGURE 9 – Simulateur de jeu d'instructions et flot de compilation permettant la conception de processeur pipeliné spéculatif.

SpecHLS permet la transformation d'une description du comportement d'un processeur afin de permettre la spéculation. En explorant ainsi les opportunités de spéculation, SpecHLS permet ainsi la conception de multiple micro-architectures de processeur à partir d'une description du comportement de ce dernier.

#### Flot de compilation de micro-architectures spéculatives

En utilisant SpecHLS et un outil de HLS, il est donc possible de générer automatiquement une micro-architecture spéculative efficace à partir d'une description comportementale d'un processeur dans un code C ou C++. Le comportement d'un processeur est similaire à celui d'un interpréteur permettant d'exécuter son jeu d'instruction. Le code représentant le comportement d'un processeur s'appelle un *simulateur de jeu d'instruction* (ISS, ou *Instruction Set Simulator*). Un exemple simple de simulateur de jeu d'instruction est présenté sur la Figure 9a. Cet ISS peut alors être transformé sous la forme Gated-SSA pour permettre l'exploration des différentes opportunités de spéculation. Le code transformé est ensuite regénéré, permettant ainsi à un outil de HLS d'obtenir diverses micro-architectures de processeur à partir de l'ISS initial. Ce flot de compilation permettant la compilation de micro-architectures de processeur à partir d'un ISS est représenté sur la Figure 9b.

### Les limites de SpecHLS

SpecHLS a été évalué grâce à la conception de processeurs RISC-V [1], et les résultats obtenus sont comparables aux processeurs conçus manuellement. Cependant, les instructions RISC-V se décomposent toutes vers un nombre d'opérations élémentaires du même ordre de grandeur. En effet, le langage RISC-V a été conçu dans le but de facilité la mise en œuvre matérielle de processeur. Par conception, la granularité des opérations RISC-V est le cycle d'horloge. En revanche, pour les instructions CISC les opérations élémentaires nécessaires à l'exécution de chaque instruction ne sont pas forcément bornées. Par exemple, l'instruction WebAssembly *memory.copy* copie une portion de la mémoire et les opérations nécessaire pour effectuer cette instruction dépendent des opérandes. De plus, la longueur de l'encodage des instructions CISC n'est pas toujours constante comme pour les instructions RISC. Par exemple, les instructions WebAssembly sont encodées sur un à trois octets, suivies des opérandes dont le nombre et la longueur des encodages sont également variables. Le flot de compilation utilisant SpecHLS n'est donc à priori pas capable de produire des résultats satisfaisants pour des instructions CISC.

## 4 Étude de cas du WebAssembly

Afin de comprendre comment adapter un outil tel que SpecHLS pour permettre l'obtention de résultats efficaces pour un processeur WebAssembly, il est nécessaire de comprendre les caractéristiques de ce langage, comme la structure de ses programmes (Section 4.1), ou encore comment sont exécutées les instructions WebAssembly (Section 4.2). Nous connaîtrons alors quels sont les aspects du WebAssembly qui semblent ne pas être adaptées à la conception de processeurs pipelinés spéculatifs (Section 4.3).

### 4.1 Structure d'un programme WebAssembly

Contrairement à des jeux d'instructions classiques comme RISC-V, le WebAssembly ne définit pas uniquement une liste d'instruction, il définit également le format du fichier exécutable. Il existe deux versions du format de fichiers WebAssembly, une première textuelle (fichiers *.wat*) et compréhensible par un humain et une seconde binaire (fichiers *.wasm*) utilisée pour l'exécution. Nous nous

intéressons ici à la version binaire puisque notre but est de concevoir un processeur exécutant du WebAssembly.

#### Modules et sections

Un fichier binaire WebAssembly représente un *module* WebAssembly. Un module est composé d'une suite de sections. Il existe douze catégories de sections auxquelles sont attachées des identifiants uniques. Ces sections peuvent contenir des déclarations de types, des déclarations de fonctions ou le code de ces fonctions par exemple. Chaque section apparaît au plus une fois, par ordre croissant d'identifiant, la seule exception est la catégorie d'identifiant 0 qui peut être présente plusieurs fois et n'importe où, elle représente une section personnalisée dont la sémantique n'est pas définie par le WebAssembly. Une section est composée d'un octet représentant son identifiant, suivi d'un entier représentant la longueur en octets du contenue de la section, suivi du contenu de la section.

Les sections sont interdépendantes entre elles et il est nécessaire de pouvoir naviguer dans ces sections pour exécuter le code WebAssembly. Par exemple, pour effectuer un appel de fonction, l'instruction `call` est suivie d'un entier 32 bits dont la valeur est la position de la fonction dans la section *fonction*. La section *fonction* contient un vecteur d'entiers 32 bits dont les valeurs sont la position du type de la fonction correspondante dans la section type, la section type contenant un vecteur de type. Le code de la fonction se situe dans la section code contenant un vecteur de code, où un code est représenté par un entier 32bits représentant la taille en octets des instructions, un vecteur de déclarations de variable locale suivie du corps de la fonction terminée par une instruction `end` (`0x0b`). Une représentation des indirections dans l'exécutable à effectuer pour l'exécution d'un appel de fonction est présenté sur la Figure 10.

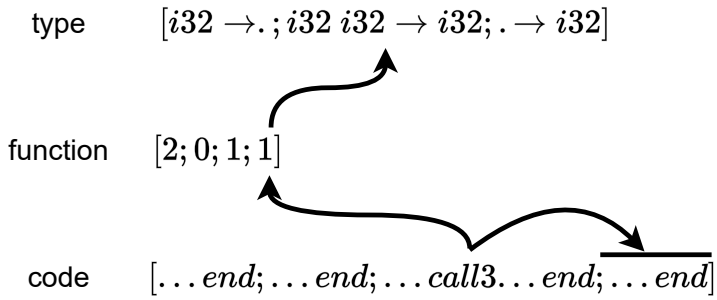



FIGURE 10 – Indirection pour la résolution d'un appel de fonction en WebAssembly.

### 4.2 Exécution des instructions WebAssembly

Comme pour la résolution des appels de fonction, il existe d'autres cas où l'exécution des instructions WebAssembly nécessite de parcourir le fichier binaire, c'est notamment le cas lors de l'exécution des instructions de flot de contrôle.

#### Flot de contrôle

Contrairement à un jeu d'instructions classique, le WebAssembly définit également des instructions de flot de contrôle faisant généralement partie de langages de haut-niveau. En effet, dans un programme WebAssembly, il n'y a aucune adresse. En revanche, il y a des instructions telles que `block` et `loop`, qui définissent le début d'une structure qui se termine par une instruction `end`. Les instructions de branchements telles que `br` ou `br_if` sont alors suivi d'un entier 32 bits correspondant au nombre de structures imbriquées desquelles il faut sortir pour réaliser le branchement. Il y a alors deux possibilités :

— Le branchement arrive sur une instruction `block`, l'instruction suivante est alors celle qui se situe après la structure, il faut donc parcourir le fichier binaire pour trouver l'instruction `end` correspondante. Les instructions `end` peuvent terminer soit un `block`, soit un `loop`, soit un `if`.
— Le branchement arrive sur une instruction `loop`, l'instruction suivante est alors celle qui se situe au début de la structure.

Un exemple montrant les indirections dans le code nécessaire à l'exécution d'un branchement est représenté sur la Figure 11.

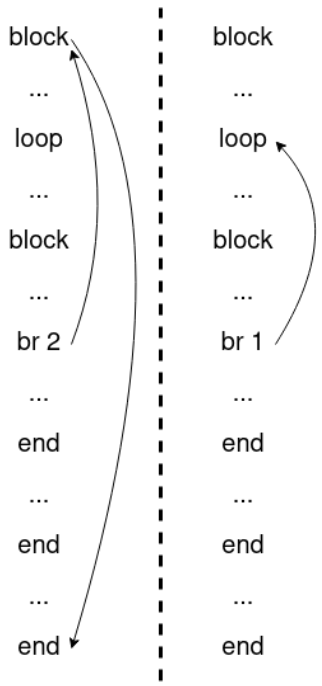



FIGURE 11 – Indirections nécessaire à l'exécution de branchement WebAssembly.

Le WebAssembly définit également des instructions `if` et `else`. La structure du code est alors `if`...`else`...`end`, telle que pour exécuter l'instruction `if`, si la valeur en sommet de pile ne vaut pas 0 alors l'exécution continue, sinon le code doit être parcouru afin de trouver l'instruction `else` correspondante et la suite de l'exécution se situe après cette instruction `else`. L'exécution de l'instruction `else` est similaire à un branchement vers l'instruction `end` correspondant au bloc `if`...`else`...`end`.

Ainsi, L'exécution d'instructions qui sont simple dans le jeu d'instructions RISC-V peut nécessiter un nombre d'opérations élémentaires importante en WebAssembly et même nécessiter de parcourir le fichier binaire. Nous pouvons donc remarquer ici que le WebAssembly se présente comme un cas extrême de jeu d'instructions CISC, c'est par exemple le seul langage ayant des instructions devant parcourir le fichier exécutable. Ces instructions complexes pouvant s'exécuter régulièrement, par exemple 7,9% des instructions sont des branchements [2], les exécuter en logiciel ne semble pas efficace. De plus, ces instructions ne rentrent pas dans un pipeline classique de processeur, il est donc nécessaire d'appliquer une transformation à ces instructions afin de leurs permettre de rentrer dans le pipeline des processeurs synthétisés.

#### Machine virtuelle à pile

Le WebAssembly contient des instructions complexes, comme la copie d'une portion de la mémoire, et sa structure impose que la gestion du flot de contrôle soit complexe également. De plus,

l'exécution du WebAssembly repose sur une machine à pile et non une file de registre. L'accès à une pile est différent de l'accès à une file de registre. En effet, tous les registres peuvent être accédés en parallèle tandis que dans une pile, seul l'élément au sommet peut être accédé. Pour obtenir une micro-architecture efficace, il faut alors choisir une implémentation de pile permettant diverses combinaisons d'accès en parallèle en fonction de l'instruction à exécuter. Si tous les accès à la pile nécessaires à l'exécution d'une instruction ne peuvent pas être effectué en parallèle, l'exécution devra alors durer plusieurs cycles d'horloge. Le pipeline devra alors être mis en pause le temps d'exécuter cette instruction.

Les accès à la pile par une instruction peuvent par exemple être :

— Une écriture sans lecture (avec l'instruction `i32.const` par exemple).
— Une écriture et deux lectures (avec l'instruction `i32.add` par exemple).
— Deux lectures sans écriture (avec l'instruction `i32.store` par exemple).
— Un nombre arbitraire de lectures sans écriture (avec l'instruction `call` par exemple).

De plus, les lectures et écritures peuvent concerner des valeurs de tailles différentes : 32, 64 ou 128 bits.

Ainsi, Il est possible que le facteur limitant lors de l'exécution d'une instruction soit l'accès à la pile. Cette problématique n'existant pas pour les processeurs à registre, il est nécessaire de choisir une implémentation permettant l'exécution la plus efficace des instructions.

## 4.3 Défi de conception de processeurs WebAssembly

Bien que le WebAssembly se définisse comme un format d'instruction binaire, SpecHLS n'a pas la capacité à concevoir des processeurs WebAssembly. En effet, la gestion de la pile d'opérandes n'est actuellement pas prise en compte et les instructions complexes, ne pouvant pas s'insérer dans un pipeline de processeur n'est pas possible.

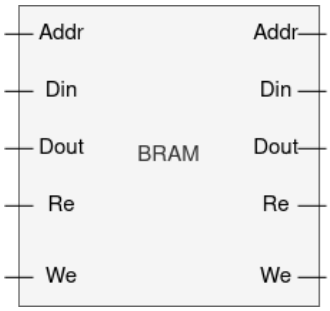
### 4.3.1 Pile d'opérandes

Afin d'obtenir un processeur WebAssembly efficace, il est nécessaire de choisir une implémentation pour la pile permettant au mieux de répondre aux contraintes du processeur. La pile doit permettre différentes combinaisons d'accès et permettre la spéculation, pour cela, il est nécessaire de pouvoir annuler les lectures destructives et les écritures sur la pile. De plus, pour les instructions les plus probables doivent pouvoir effectuer tous leurs accès en un seul cycle d'horloge pour permettre la conception d'un processeur pipeliné spéculatif commençant une instruction par cycle. Si les accès à la pile doivent s'effectuer en plusieurs cycles, alors le pipeline doit être mis en pause à cause de conflit d'accès à la pile.

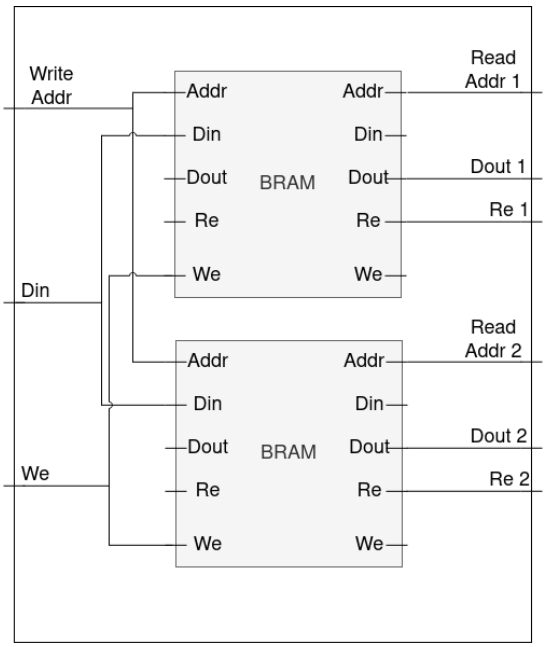
Les valeurs de la pile peuvent être stockée :

— Dans des registres, qui sont rapides mais la logique de contrôle nécessaire devient importante lorsque la taille de la pile augmente.
— Dans des blocs mémoire (BRAM, ou *Block RAM*), des mémoires agissant comme un tableau permettant d'accéder à chaque mot en mémoire en choisissant leurs adresses. Contrairement aux registres, les BRAMs possèdent un nombre limité de ports d'accès. Par exemple les BRAMs du FPGA `alveo U280` de Xilinx possèdent deux ports d'accès chacun, ces ports pouvant servir à la fois de lecture ou d'écriture.

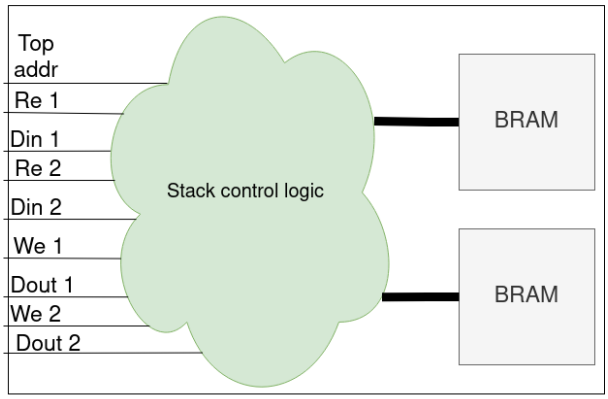
Il existe alors différentes possibilités de combinaisons de BRAMs et registres afin d'implémenter des piles, elles sont représentées sur la Figure 12 et expliquées ci-dessous.



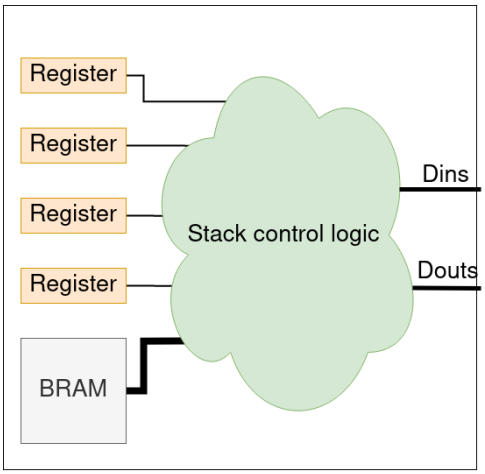

(a) Bloc RAM pouvant contenir une pile dans un tableau simple.




(b) Chemin de données représentant une pile dans un tableau dupliqué.




(c) Chemin de données représentant une pile dans un tableau partitionné. La logique de contrôle a été simplifié.




(d) Chemin de données représentant une pile dont le sommet se trouve dans des registres. La logique de contrôle a été simplifié.

FIGURE 12 – Chemins de données représentant différentes implémentations de piles matérielles.

**Tableau simple.** De manière similaire à une implémentation logicielle, une version simple d'implémentation d'une pile est d'utiliser un tableau contenant les valeurs de la pile. Ce tableau pourra être contenu dans des BRAMs, et l'adresse du sommet de la pile pourra être stockée dans des registres. Ainsi, à chaque cycle d'horloge, il sera possible d'effectuer deux opérations mémoires, que ce soit lectures ou écritures. Ce choix d'implémentation de la pile est le plus simple possible, et est celui qui nécessite le moins de logique de contrôle. Cependant, l'exécution des instructions WebAssembly ne sera pas efficace car même les instructions simples, comme les additions 32 bits, nécessiterons plusieurs cycles d'horloge pour s'exécuter. En effet, elles nécessitent la lecture des deux opérandes et l'écriture du résultat sur la pile.

**Tableau dupliqué.** Afin de lever la limitation du nombre de lecture des BRAMs, il est possible de dupliquer la mémoire. Ainsi, pour chaque BRAM, un port sera consacré à l'écriture et l'écriture s'effectuera en parallèle dans toutes les BRAMs, et le second port permettra la lecture. Il sera donc possible accéder en parallèle à une écriture et à autant de lecture que de duplication de mémoire. La duplication de la mémoire permet d'effectuer en parallèle tous les accès nécessaires aux instructions simples. Cependant, le fait du dupliquer une mémoire rend le circuit peu efficace en termes d'utilisation des ressources, car plus de ressources sont utilisées pour stocker la même information.

**Tableau partitionné.** L'accès à une pile s'effectue toujours au sommet de cette pile. Ainsi, contrairement à un tableau quelconque, il est sûr que des accès successifs au tableau s'effectueront à des adresses qui se suivent. Ainsi, à la place de dupliquer la mémoire il est également possible de partitionner le tableau en plusieurs sous-tableaux de manière cyclique, c'est-à-dire que deux adresses consécutives accèdent à des BRAMs différents, et peuvent donc être effectués en parallèle.

**Sommet de pile séparé du reste.** Si la pile est d'une taille assez réduite pour pouvoir être stockée dans un seul BRAM (36kbits pour les BRAMs de la `alveo U280`), partitionner le tableau induit une surutilisation des ressources semblable à la duplication du tableau. Une solution permettant l'accès en parallèle à plusieurs valeurs en sommet de pile est d'écrire plusieurs valeurs du sommet de la pile dans des registres et les autres valeurs dans un BRAM. Ainsi, avec `N` valeurs placées dans des registres, il est possible d'effectuer en parallèle jusqu'à `N+1` lectures et `N` écritures. Effet, il est possible de lire la valeur de chaque registre ainsi qu'une valeur présente dans le BRAM en parallèle. Cependant, pour s'effectuer en un cycle, le nombre d'écritures doit être à un près égale au nombre de lectures, car il est nécessaire d'écrire les valeurs en plus dans le BRAM en cas d'écritures et les registres vidées par une lecture doivent être remplie pour permettre les futurs accès au sommet de la pile. Cette solution utilise peu de ressources supplémentaires comparée à l'utilisation d'un unique BRAM et permet d'effectuer les opérations qui s'exécutent le plus souvent [2] (accès aux variables locales, lecture en mémoire, accès à l'ALU) en un seul cycle d'horloge.

Ainsi, Nous avons ici proposé différentes piles matérielles. Par la suite, il sera nécessaire d'implémenter les différentes versions afin de comparer leurs efficacités en fonction de l'utilisation des ressources et de la taille de la pile, puis de proposer une transformation automatique afin d'obtenir une pile matérielle efficaces à partir d'un simulateur de jeu d'instructions. Une difficulté de cette automatisation est de pouvoir différentier une pile d'un tableau. Cela peut être fait manuellement en ajoutant une annotation au code affirmant qu'un tableau représente une pile, ou automatiquement, en analysant les accès au tableau.

Cependant, pour pouvoir comparer techniquement les différentes approches, il est nécessaire de pouvoir exécuter des instructions complexes afin de mesurer les surcoûts engendrés qu'elles engendre. Ces instructions peuvent devoir s'exécuter en plusieurs cycle à cause d'une limitation des accès à la pile.

```
while (1) {
  instruction = decode(fetch(memory, pc));
  switch (instruction) {
  ...
  case MEMORY_COPY: {
    n = stack.pop();
    s = stack.pop();
    d = stack.pop();
    if ((d + n >= memory.size) || (s + n >= memory.size))
      trap();
    if (d <= s) {
      for (i = 0; i < n; i++) {
       memory[d+i] = memory[s+i];
    }
    } else {
      memory[d+n-1-i] = memory[s+n-1-i];
    }
  }
  ...
  }
}
```

FIGURE 13 – Extrait de simulateur de jeu d'instruction WebAssembly exécutant l'instruction `memory.copy`.

### 4.3.2 Micro-codage d'instructions complexes

Pour certaines instructions WebAssembly, comme `br` ou encore `memory.copy`, l'exécution de l'instruction dépend du contexte dans lequel elle se trouve. Son exécution peut dépendre d'autres instructions se trouvant dans le code pour déterminer la cible d'un branchement. L'exécution peut également dépendre d'opérandes présente sur la pile, comme pour connaître la source, la destination et le nombre d'octets à copier en mémoire. La latence d'exécution de ces instructions est donc non bornée, indéterminée et non contrôlée, ces instructions ne peuvent donc pas entrer dans un pipeline classique de processeur. Il est donc nécessaire de transformer ces instructions dans une version pouvant être pipeliné dans le processeur généré avec SpecHLS. Dans le simulateur de jeu d'instructions, la description de l'exécution de ces instructions nécessite l'utilisation de boucles, comme présenté sur la Figure 13. Cependant, la représentation Gated-SSA utilisée dans l'outil SpecHLS est générée à partir de la boucle exécutant les instructions, cette transformation ne peut pas s'appliquer en cas de présence de boucles imbriquées. Une contribution de ce stage est une transformation source-à-source à effectuer au préalable afin de permettre à SpecHLS d'exécuter de telles instructions. Nous appelons cette transformation le *unpipeline*.

## 5 Transformation des instructions CISC : le Unpipeline

Pour permettre à SpecHLS de générer un processeur pipeliné spéculatif, le programme en entrée ne doit comporter qu'un seul niveau de boucle afin de pouvoir représenter ce programme sous la forme Gated-SSA. Cependant, certaines instructions CISC du langage WebAssembly, comme memory.copy, ont une latence d'exécution imprévisible qui dépend des opérandes, la description naturelle de leur comportement utilise donc des boucles imbriquées. Il est donc nécessaire d'ajouter une pré-transformation permettant d'obtenir un seul niveau de boucle à partir d'un code quelconque. Des transformations ayant un but similaire existent dans certains cas particuliers (Section 5.1). Nous avons défini une transformation générale adaptée à la spéculation appelé le *unpipeline*, ainsi qu'une automatisation de cette transformation (Section 5.3).

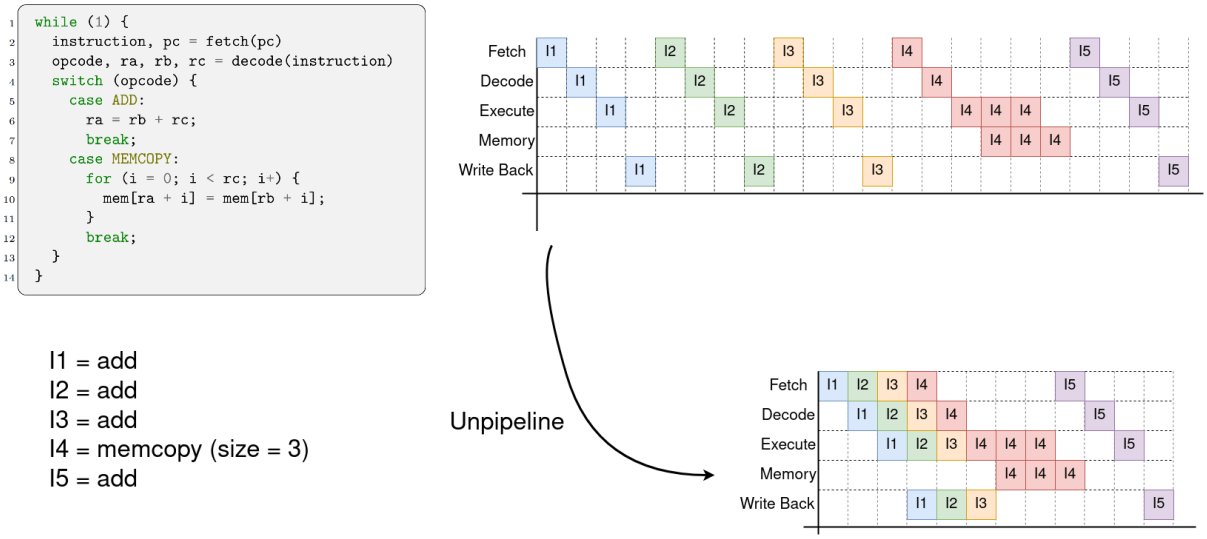



FIGURE 14 – Exemple de transformation de unpipeline.

Le *unpipeline* est une transformation qui a pour but de transformer un nid de boucles imbriquées en un seul niveau de boucle, cela dans le but de transformer le comportement temporel des micro-architectures générées avec la HLS, comme représenté sur la Figure 14. Cette transformation doit fonctionner dans le cas général et le code régénéré doit être adapté à SpecHLS. C'est-à-dire notamment qu'il doit permettre de spéculer si le corps d'une boucle est exécuté ou non. Cependant, obtenir un seul niveau de boucle à partir de boucles imbriquées n'est pas une idée nouvelle [12], certains outils de HLS, comme Vitis HLS, effectue également une transformation similaire, le *loop flattening*. Cependant, le *unpipeline* permet un contrôle plus fin de l'exécution du programme que le *loop flattening*, permettant ainsi de concevoir des processeurs plus performant, efficace en ressources ou en énergie.

## 5.1 Loop flattening

Afin de permettre à la HLS de concevoir des circuits pipelinés efficaces, il peut être nécessaire de fournir en entrée un code comprenant un seul niveau de boucle, même si le programme semble à priori nécessiter un nid de boucles imbriqués. Le *loop flattening*, ou *loop coalescing*, est une transformation permettant d'obtenir un unique niveau de boucle à partir de boucles imbriquées, mais elle ne s'applique pas dans le cas général.

```
for (i = 0; i < N; i++) {
    <body1>
    for (j = 0; j < M; j++) {
        <body2>
    }
    <body3>
}
```

(a) Nid de boucles imbriquées.

```
int i = 0;
int j = 0;
while (i < N) {
    if (j == 0) {
        <body1>
    }
    <body2>
    j++;
    if (!(j < N)) {
        <body3>
        i++;
        j=0;
    }
}
```

(b) Code de la figure 15a après *loop flattening*.

FIGURE 15 – Transformations de *loop flattening*.

Le *loop flattening* [12] consiste à remplacer un nid de boucles imbriquées par une seule boucle dont le nombre d'itérations est le nombre total d'itérations possible. Cette boucle effectue, en fonction du calcul de gardes, le contenu de certains niveaux d'imbrications. Cette boucle construite par la transformation est alors capable de simuler le comportement du nid de boucles présent avant la transformation. La Figure 15a représente un nid de boucles et la Figure 15b représente la version transformée par *loop flattening*.

Cette approche permet d'obtenir un unique niveau de boucle à partir de boucles imbriquées. Cependant, cette transformation ne peut pas s'appliquer si une logique complexe conditionne l'entrée dans la boucle interne. De plus, les programmes générés possèdent la logique nécessaire pour exécuter le corps de toutes les boucles en une seule itération, par exemple, sur la Figure 15b, une unique itération de la boucle peut effectuer `<body1>`, `<body2>` et `<body3>`. Dans le cadre de la conception de circuit matériel, cela impose que le chemin de données générer doit également être capable d'effectuer les trois parties de ce programme. En revanche, des chemins en exclusion mutuelle permettraient d'une part de fusionner les chemins de données afin d'obtenir un circuit utilisant moins de ressources matérielles, et d'autre part de permettre de spéculer sur le chemin choisit.

Ainsi, bien que le *loop flattening* soit une transformation pouvant sembler convenir à première vue, elle ne remplit pas totalement nos objectifs. Nous proposons donc une nouvelle transformation, s'appliquant sur un programme quelconque en entrée, et utilisant la structure C des switchs afin que les différents chemins possibles soient indépendants, permettant ainsi par la suite d'appliquer d'autres optimisations utilisant cette propriété. Notre transformation nous permet d'obtenir plus de contrôle sur les micro-architectures conçu avec la HLS que le *loop flattening*.

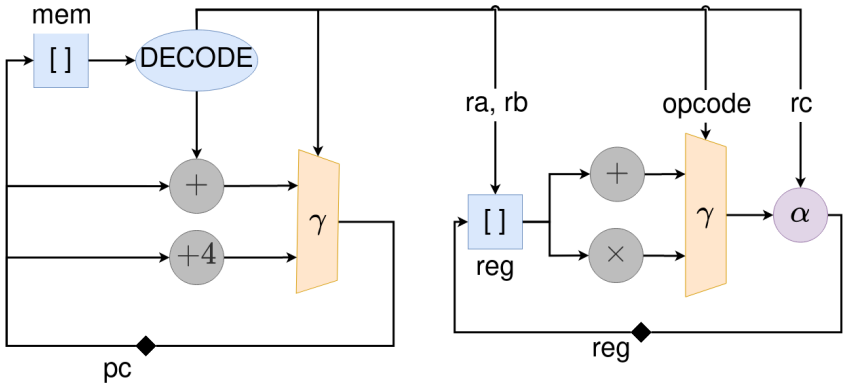
## 5.2 Transformation de unpipeline

Afin d'arriver à ce but, une machine à états (FSM, ou *Finite-State Machine*) est ajoutée et contrôle l'exécution de la boucle la plus externe, appelé par la suite "boucle principale". Dans le cadre de la conception d'un processeur WebAssembly, la boucle la plus externe correspond à la boucle du processeur exécutant les instructions ; tout le design du processeur dépend de cette boucle. La FSM contrôle l'exécution du programme, avec un état initial exécutant le comportement normal. Cependant, lorsque le programme devrait entrer dans un niveau de boucle supplémentaire, à la place, la

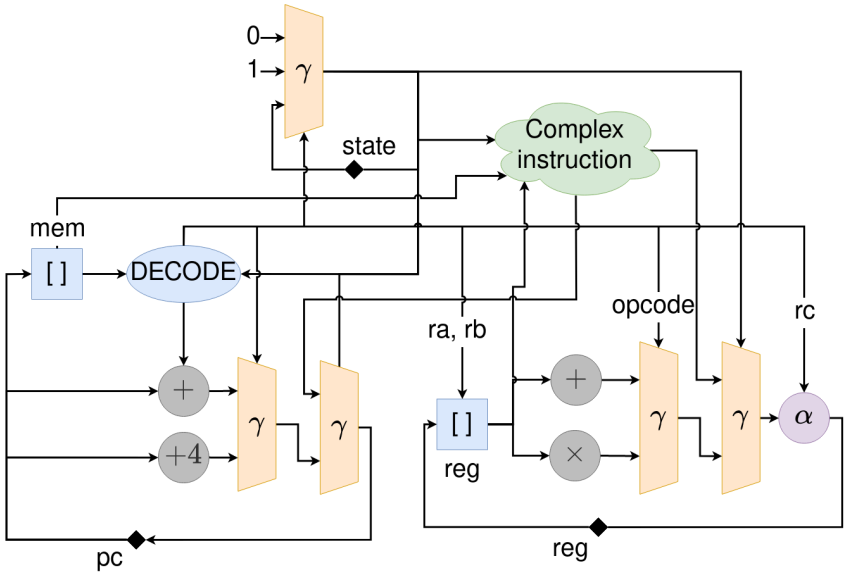
FSM effectue une transition vers un état exécutant le corps de cette boucle.

Grâce à cette méthode, l'ensemble des niveaux de boucles du programme initial se retrouvent agrégés en un seul niveau de boucle, dont l'exécution est conditionnée par le contrôle d'une FSM choisissant l'exécution du corps d'une boucle du programme initial. Dans le cadre de la conception d'un processeur WebAssembly pipeliné spéculatif, il sera alors pertinent de spéculer que l'exécution ne mène pas à un changement d'état de la FSM.

Un exemple du comportement attendu après transformation de *unpipeline* est présenté sur la Figure 14. Cet exemple représente l'exécution d'un simulateur de jeu d'instructions minimaliste comprenant une instruction d'addition et une instruction de copie de portions de la mémoire. Puisque la boucle principale contient une boucle dont le nombre d'exécution dépend d'une variable, elle ne peut pas être pipeliné, seule la boucle effectuant la copie mémoire peut l'être. Cependant, après unpipeline, nous souhaiterions que la boucle principale soit également pipelinée, permettant ainsi à chaque cycle de démarrer soit l'itération suivante de la boucle principale soit l'itération suivante de la boucle copiant la mémoire.




(a) Forme Gated-SSA d'un processeur RISC.




(b) Forme Gated-SSA du processeur de la Figure 16a auquel a été ajouté une instruction complexe grâce au unpipeline.

FIGURE 16 – Forme Gated-SSA de processeurs RISC et CISC transformé grâce au *unpipeline*.

Le *unpipeline* permet alors à un simulateur de jeu d'instructions simulant des instructions CISC d'être transformé en un code pouvant être ensuite transformé en sous la forme Gated-SSA pour per-

mettre à la spéculation. En rajoutant une instruction complexe au processeur représenté sur la Figure 16a, il devient similaire à celui représenté sur la Figure 16b.

## 5.3 Automatisation du unpipeline

SpecHLS est un outil permettant de concevoir automatiquement des micro-architectures de processeurs pipelinés spéculatifs à partir d'un simulateur de jeu d'instruction. Grâce à la transformation de *unpipeline*, il est possible d'ajouter des instructions CISC aux processeurs conçu avec SpecHLS. Cependant, pour pouvoir concevoir un processeur possédant une instruction CISC, SpecHLS doit prendre en entrée la version *unpipelinée* du simulateur de jeu d'instructions, ce qui n'est pas le but initial de SpecHLS. Ajouter une phase appliquant automatiquement la transformation du unpipeline permettrait à SpecHLS de concevoir des processeurs CISC à partir d'un simulateur de jeu d'instructions non modifié. Nous proposons ici une automatisation du *unpipeline* s'appliquant sur l'arbre de syntaxe abstraite (AST, ou *Abstract Syntax Tree*) grâce à de la programmation dirigée par la syntaxe, c'est-à-dire ajoutant des attributs aux nœuds de l'AST.

### Définitions utiles

Dans notre AST utilisé pour automatiser la transformation du *unpipeline*, nous appelons les nœuds correspondant à une boucle while, `while`, à une boucle do...while, `doWhile` et à une boucle for, `for`. Nous désignons les différents composants de ces boucles grâce à des attributs :

- Le corps d'une boucle `l` est désigné par l'attribut `l.bodyBlock`.
- La condition d'une boucle `l` est désignée par l'attribut `l.condBlock`.
- L'instruction s'exécutant à l'initialisation d'une boucle for est désignée par l'attribut `for.initBlock`.
- L'instruction s'exécutant entre les itérations d'une boucle for est désignée par l'attribut `for.stepBlock`.

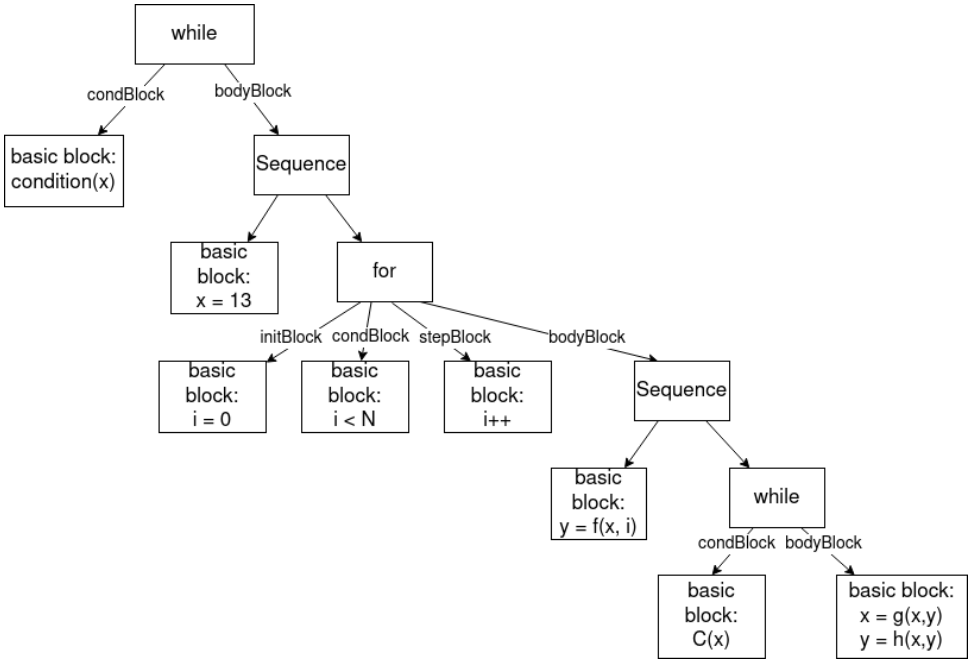
La boucle la plus externe est toujours appelée boucle principale. Nous supposons ici qu'il s'agit soit d'une boucle while, soit d'une boucle do...while, les boucles for pouvant être transformées en boucle while si besoin.

On appelle le `parent` d'une boucle interne la boucle dans laquelle elle est imbriquée. La boucle principale ne possède pas de parent.

On appelle `fallthrough` d'une boucle, l'ensemble du code pouvant s'exécuter après cette boucle.

```
mainLoop: while (condition(x)) {
  x = 13;
  intermediateLoop: for (i = 0; i < N; i++) {
    y = f(x,i);
    innerLoop: while (C(x)) {
      x = g(x,y);
      y = h(x,y);
    }
  }
}
```

(a) Exemple de code contenant des boucles.




(b) AST correspondant au code de la Figure 17a.

FIGURE 17 – Exemple de code contenant des boucles et AST correspondant.

Par exemple, dans le code de la Figure 17a, la boucle `mainLoop` est la boucle principale, le parent de la boucle `innerLoop` est la boucle `intermediateLoop` et le parent de la boucle `intermediateLoop` est la boucle `mainLoop`. L'AST correspondant à ce code est représenté sur la Figure 17b présentant le nom des différents composants des boucles annotés sur les arcs où on appelle *basic block* les blocs de code tel qu'aucun branchement ne rentre ailleurs qu'au début du bloc et aucun branchement ne sort hormis à la fin du bloc. On appelle également *sequence* un bloc contenant plusieurs blocs s'exécutant à la suite.

On appelle *Block* le type générique des nœuds de l'AST, un *Block* peut-être un *basic block*, une *sequence* ou un *while* par exemple.

### Définition des attributs

Pour permettre l'automatisation du *unpipeline* en utilisant un AST attribué, il est nécessaire de définir cinq attributs, quatre attribuant les boucles internes et le dernier attribuant également la boucle principale.

On définit donc :

— `bodyStateNumber`, un entier attribuant les boucles qui correspond à l'indice désignant l'état de la FSM exécutant le corps de la boucle.

— `fallthroughStateNumber`, un entier attribuant les boucles qui correspond à l'indice désignant l'état de la FSM exécutant le code après la fin de l'exécution de la boucle.
— `replacementBlock`, un bloc attribuant les boucles qui correspond au bloc par lequel sera remplacé la boucle après la transformation. En effet, après transformation la boucle n'existe plus en elle-même, mais elle est remplacée par une transition d'état de la FSM.
— `FSMSwitchCases`, un ensemble de cas attribuant les boucles qui correspond aux états de la FSM dont les indices sont `bodyStateNumber` et `fallthroughStateNumber`. En effet, pour mettre en place la FSM en termes de code, l'état courant sera stocké dans une variable et l'exécution sera conditionnée par un `switch` dont la condition est l'état exécutant le code correspondant à l'état courant.
— `transition`, un block, attribuant toutes les boucles et correspondant à la transition à effectuer lorsque l'exécution de la boucle est terminée.

#### Transformation de la condition de la boucle principale

Lors de la transformation du *unpipeline*, l'exécution d'une itération de la boucle principale peut se transformer en l'exécution de plusieurs itérations de la boucle transformée tout en conservant le comportement du programme. Cependant, si la condition de la boucle principale devient fausse pendant son exécution, la boucle transformée risque de s'arrêter entre deux itérations correspondant à une transition entre deux états et non à la fin de l'exécution du corps de la boucle principale. Afin que la transformation du *unpipeline* conserve le comportement du programme initial, il est alors nécessaire de changer la condition de la boucle principale afin d'avoir le contrôle sur les étapes nécessitant d'évaluer cette condition.

```
cond = condition(x) ? 1 : 0;
mainLoop: while (cond) {
  x = 13;
  intermediateLoop: for (i = 0; i < N; i++) {
    y = f(x,i);
    innerLoop: while (C(x)) {
      x = g(x,y);
      y = h(x,y);
    }
  }
  cond = condition(x) ? 1 : 0;
}
```

FIGURE 18 – Code de la Figure 17a après transformation de la condition.

Pour faire cela, on définit une nouvelle variable `cond`, non présente dans le programme initial, de type entier. Lorsque la condition de la boucle principale doit être évaluée, c'est-à-dire avant la boucle, à la fin du corps de la boucle, et avant toute instruction `continue` se situant dans le corps de la boucle principale mais pas dans une boucle imbriquée, la variable `cond` est affectée à 1 si la condition de la boucle principale est vraie et à 0 sinon. La condition de la boucle principale est alors remplacée par l'évaluation de la variable `cond`. La Figure 18 représente le code de la Figure 17a après transformation de la condition de la boucle principale.

#### Affectation des valeurs des attributs

Après la transformation de la condition de la boucle principale, on sait que même si une itération de la boucle principale avant transformation s'exécute en plusieurs itérations de la boucle du code transformé par le *unpipeline*, il n'y a pas de risque de changer le comportement du code. Il reste alors

à définir les valeurs des différents attributs définies plus tôt pour permettre à la transformation du *unpipeline*.

Dans toute la suite, on définit `cs`, une variable de type entier, de valeur initiale 0, et non présente dans le programme avant transformation. Cette variable contient alors tout au long de l'exécution l'indice de l'état actuel de la FSM.

- Les attributs `bodyStateNumber` et `fallthroughStateNumber` correspondent à des indices d'états de la FSM. Ce sont donc des entiers qui sont uniques, supérieurs ou égaux à 1 et deux-à-deux distinct. L'état 0 de la FSM correspond à l'exécution du corps de la boucle principale. Donc, pour deux boucles interne à la boucle principale différentes `l1` et `l2` :
  - `l1.bodyStateNumber` $\geq 1$.
  - `l1.fallthroughStateNumber` $\geq 1$.
  - `l1.bodyStateNumber` $\neq$ `l2.bodyStateNumber`.
  - `l1.fallthroughStateNumber` $\neq$ `l2.fallthroughStateNumber`.
  - `l1.bodyStateNumber` $\neq$ `l2.fallthroughStateNumber`.
  - `l1.bodyStateNumber` $\neq$ `l1.fallthroughStateNumber`.
- L'attribut `transition` correspond à la transition qui doit s'effectuer lorsque l'exécution d'une boucle se termine.
  - Soit `b` la boucle principale, `b.transition` est alors un basic block contenant les instructions `cs = 0` et `continue`.
  - Soit `for` une boucle for interne, `for.transition` est alors une *sequence* contenant le block `for.stepBlock`, suivie du *basic block* contenant les instructions `cs = for.condBlock ? for.bodyStateNumber : for.fallthroughStateNumber` et `continue`.
  - Soit `while` une boucle while interne, `while.transition` est alors un basic block contenant les instructions `cs = while.condBlock ? while.bodyStateNumber : while.fallthroughStateNumber` et `continue`.
  - Soit `doWhile` une boucle do...while interne, `doWhile.transition` est alors un basic block contenant les instructions `cs = doWhile.bodyStateNumber` et `continue`.
- L'attribut `replacementBlock` correspond au *block* par lequel sera remplacée la boucle dans le corps de la boucle principale après transformation de *unpipeline*. Elle consiste à l'initialisation et la transition dans la FSM vers l'état exécutant la boucle interne ou ce qui se situe après cette boucle si la condition d'entrée est fausse.
  - Soit `for` une boucle for interne, `for.replacementBlock` consiste en une sequence contenant le *block* `for.initBlock` suivie du *basic block* contenant les instructions `cs = for.condBlock ? for.bodyStateNumber : for.fallthroughStateNumber` et `continue`.
  - Soit `l` une boucle while ou do...while interne, `l.replacementBlock` consiste en le block `l.transition`.
- L'attribut `FSMSwitchCases` contient les cas du switch correspondant à la FSM associés aux états dont les indices sont `bodyStateNumber` et `fallthroughStateNumber`. Soit `l` une boucle interne, `l.FSMSwitchCases` correspond au cas qui à `l.bodyStateNumber` associe la sequence contenant `l.bodyBlock` suivie de `l.transition`, ainsi que le cas qui à `l.fallthroughStateNumber` associe la sequence contenant `l.fallthrough` suivie de `l.parent.transition`.

**Mise en œuvre de la transformation**

Après avoir défini les attributs ainsi que leurs valeurs, la transformation du *unpipeline* devient directe. Il suffit de remplacer certaines parties du code en fonction des différents attributs.

Pour chaque boucle interne `l`, elle est remplacée par `l.replacementBlock`. De plus, la boucle principale est remplacée par un switch dont la condition est la variable `cs` et dont les cas sont l'ensemble des `l.cases` pour toutes boucles internes `l` auxquels est ajouté, comme cas par défaut, le cas qui à 0 associe `mainLoop.bodyBlock` où `mainLoop` est la boucle principale.

On obtient alors la version *unpipelinée* du code initial.

**Exemple détaillé**

| loop | bodyStateNumber | fallthroughStateNumber |
|---|---|---|
| mainLoop | | |
| intermediateLoop | 1 | 2 |
| innerLoop | 3 | 4 |

| loop | transition |
|---|---|
| mainLoop | `cs = 0; continue;` |
| intermediateLoop | `i++; cs= (i<N) ? 1 : 2; continue;` |
| innerLoop | `cs = C(x) ? 3 : 4; continue;` |

| loop | replacementBlock |
|---|---|
| mainLoop | |
| intermediateLoop | `i=0; cs = (i<N) ? 1 : 2; continue;` |
| innerLoop | `cs = C(x) ? 3 : 4; continue;` |

| loop | FSMSwitchCases |
|---|---|
| mainLoop | |
| intermediateLoop | `1` $\rightsquigarrow$ `y = f(x,i); cs = C(x) ? 3 : 4; continue;`<br>`2` $\rightsquigarrow$ `cond = condition(x) ? 1 : 0; cs = 0; continue;` |
| innerLoop | `3` $\rightsquigarrow$ `x = g(x,y); y = h(x,y); cs = C(x) ? 3 : 4; continue;`<br>`4` $\rightsquigarrow$ `i++; cs = (i<N) ? 1 : 2; continue;` |

FIGURE 19 – Valeurs des attributs associés aux boucles du code de la Figure 18.

Considérons le code de la Figure 18, dont la condition a déjà été transformée. La boucle principale est la boucle `mainLoop` et deux boucles internes sont `intermediateLoop` et `innerLoop`.

Les attributs `bodyStateNumber` et `fallthroughStateNumber` des boucles `intermediateLoop` et `innerLoop` sont des entiers supérieur ou égaux à 1, deux à deux distincts. Les valeurs 1, 2, 3 et 4 leurs sont donc associées.

Ensuite, donnons les valeurs de l'attribut `transition`. La boucle principale (`mainLoop`) a comme `transition` `cs = 0; continue`, `intermediateLoop` est une boucle for, dont le `stepBlock` vaut `i++` et le `condBlock` vaut `i<N`, sa `transition` vaut alors `i++; cs = (i<N) ? 1 : 2; continue`. La boucle `innerLoop` est une boucle while dont le `condBlock` vaut `C(x)`, sa `transition` vaut alors `cs = C(x) ? 3 : 4; continue;`.

L'attribut `replacementBlock` de la boucle `intermediateLoop` vaut `i=0; cs = (i<N) ? 1 : 2; continue;` car son `initBlock` vaut `i=0`. Celui de la boucle `innerLoop` vaut `cs = C(x) ? 3 : 4; continue;`.

L'attribut `FSMSwitchCase` contient un ensemble de cas, on note `n` $\rightsquigarrow$ `c` le cas qui à `n` associe le code `c`. Par anticipation, nous remplaçons dès cette étape le code éventuel des boucles par `replacementBlock`. En effet, le corps (`bodyBlock`) de la boucle `intermediateLoop` continent la

boucle `innerLoop`. Ainsi, pour éviter de surcharger en recopiant la boucle puis en la remplaçant, nous remplaçons directement la boucle.

La valeur de l'attribut `FSMSwitchCase` de la boucle `intermediateLoop` est l'ensemble contenant les cas :

— `1` ⇝ `y = f(x,i); cs = C(x) ? 3 : 4; continue;`. C'est le cas qui à son `bodyStateNumber` associe son `bodyBlock` suivie de sa `transition`. Cependant, puisque son `bodyBlock` contient une instruction continue, la `transition` fait partie du code mort et a donc été omise ici.

— `2` ⇝ `cond = condition(x) ? 1 : 0; cs = 0; continue;`. C'est le cas qui a son `fallthroughStateNumber` associe le code qui doit s'exécuter après cette boucle suivie de la transition de sa boucle parent : la boucle principale.

```
1  cs = 0;
2  cond = condition(x) ? 1 : 0;
3  while (condition(x)) {
4    switch (cs) {
5    case 1:
6      y = f(x, i);
7      cs = C(x) ? 3 : 4;
8      continue;
9    case 2:
10     cond = condition(x) ? 1 : 0;
11     cs = 0;
12     continue;
13   case 3:
14     x = g(x, y);
15     y = h(x, y);
16     cs = C(x) ? 3 : 4;
17     continue;
18   case 4:
19     i++;
20     cs = (i<N) ? 1 : 2;
21     continue;
22   default:
23     x = 13;
24     i = 0;
25     cs = (i<N) ? 1 : 2;
26     continue;
27   }
28 }
```

FIGURE 20 – Code de la Figure 18 après transformation de *unpipeline*.

La valeur de l'attribut `FSMSwitchCase` de la boucle `innerLoop` est l'ensemble contenant les cas :

— `3` ⇝ `x = g(x,y); y=h(x,y); cs = C(x) ? 3 : 4; continue;`. C'est le cas qui à son `bodyStateNumber` associe son `bodyBlock` suivie de sa `transition`.

— `4 ⇝ i++; cs = (i<N) ? 1 : 2; continue;`. C'est le cas qui à son `fallthroughStateNumber` associe le code qui doit s'exécuter après cette boucle suivie de la transition de sa boucle parent, `intermediateLoop`. Aucun code ne se trouve dans la boucle `intermediateLoop` après la boucle `innerLoop`, ce cas contient donc uniquement la transition de la FSM vers l'état exécutant l'étape suivante de la boucle ligne 4.

Nous avons donc défini les valeurs des différents attributs nécessaires à la transformation du *unpipeline*. Ces valeurs sont résumées dans les tableaux de la Figure 19. Pour effectuer la transformation, il reste alors à remplacer le code des différentes boucles. La version *unpipelinée* du code de la Figure 18 est alors obtenue, et présentée sur la Figure 20.

#### Autres remarques sur le unpipeline

Après avoir défini le *unpipeline* et proposé une méthode pour l'automatiser, La marche à suivre pour obtenir la version transformée est indépendante du contenu réel du programme. Les effets de cette transformation sont globaux mais chaque étape s'effectue localement sur les nœuds de l'AST. De plus, cette transformation peut s'appliquer à un code contenant un nid de boucle quelconque, et les différents états de la FSM s'exécutent en exclusion mutuelle, c'est-à-dire qu'une itération de la boucle principale exécute le code d'un unique état de la FSM. Cela permet d'obtenir une version spéculative spéculant sur l'état de la FSM à exécuter ou encore de profiter de la fusion de chemins de données pour obtenir un circuit utilisant moins de ressources tout en ayant le même comportement.

## 6 État d'avancement des contributions

Lors de ce stage, le principal objectif était d'étudier la faisabilité de processeur CISC grâce au flot de compilation utilisant SpecHLS. Pour ce faire nous avons étudier le langage WebAssembly, puis nous avons synthétisé un processeur dont le jeu d'instructions est un sous-ensemble du bytecode WebAssembly. Grâce à cela nous avons pu identifier les deux limites principales à la conception de tels processeurs grâce à SpecHLS à partir d'un simulateur de jeu d'instructions. La première limite concerne les processeurs à pile, le choix de l'implémentation de la pile d'opérande a un impact important sur les performances du processeur, et quelques solutions ont été étudiés ici de manière uniquement théorique. Cependant, lever la seconde limitation semble avoir des conséquences bénéfiques sur des enjeux plus généraux que la conception de processeurs CISC. Cette limitation concerne les instructions complexes ne pouvant pas entrer dans un pipeline classique de processeur, il nous a fallu trouver une transformation permettant de les transformer afin de les insérer dans le pipeline des processeurs. La transformation que nous avons proposée ici, que nous avons appelé le *unpipeline*, permet alors à un simulateur de jeu d'instructions CISC d'être transformé en un programme pouvant être transformé à son tour grâce à SpecHLS. De plus, cette transformation est indépendante de SpecHLS. En effet, nous appliquons le *unpipeline* en amont de SpecHLS, et donc nous pouvons l'utiliser de manière indépendante. Nous pouvons donc l'utiliser pour des applications autres que la conception automatique de processeurs, voire pour permettre à l'ajout de fonctionnalités supplémentaire aux processeurs. Par exemple, il est possible d'utiliser le *unpipeline* afin d'ajouter un cache de données et ainsi améliorer les accès mémoires des circuits, que ce soit pour un processeur ou même pour un accélérateur matériel, qui sont généralement conçu grâce à la HLS.

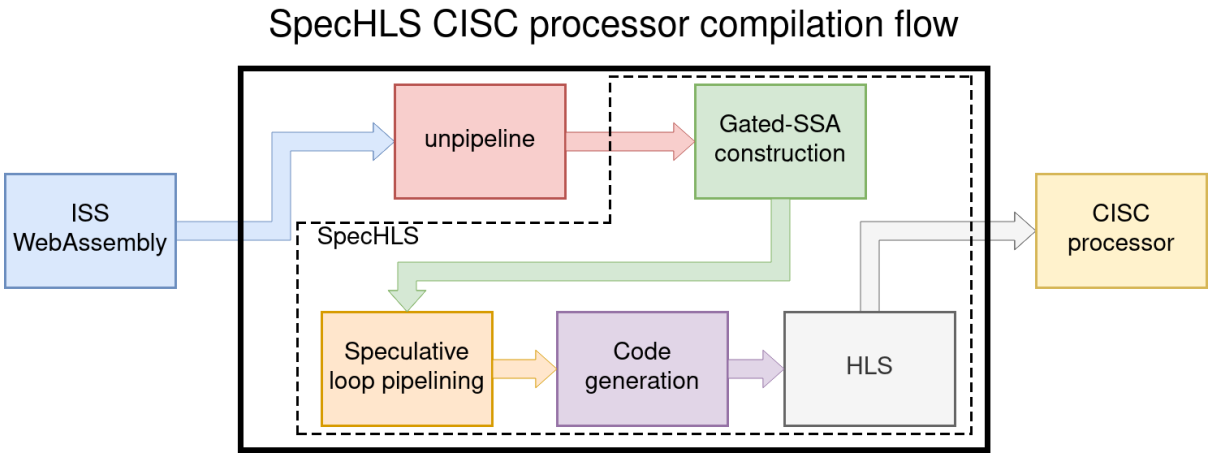



FIGURE 21 – Flot de compilation de processeurs CISC à partir d'un simulateur de jeu d'instructions.

L'avancement actuel de ce stage est donc que la version initiale du *unpipeline*, présentée ici, et son automatisation. Ainsi que des idées, de fonctionnalités supplémentaires pouvant être ajouté dans le futur en modifiant légèrement le *unpipeline*. Par exemple, l'ajout de la FSM permet d'obtenir un contrôle temporel plus fin de la micro-architecture générée par la HLS, il est donc possible d'imposer à la micro-architecture d'exécuter le corps de la boucle sur plus de cycles d'horloge différents et ainsi d'espérer obtenir une fréquence plus élevée et d'améliorer le partage des opérateurs matériels. Dans la suite de ce stage, nous devrons mesurer l'impact du *unpipeline* sur les micro-architectures générée par la HLS puis concevoir un processeur WebAssembly similaire à celui déjà obtenue, mais en utilisant le flot de compilation utilisant le *unpipeline* et SpecHLS. Ce flot de compilation à utiliser pour concevoir automatiquement un processeur WebAssembly est présenté sur la Figure 21.

## 7 Conclusion et travaux futurs

L'IoT s'appuie sur des microcontrôleurs personnalisés qui peuvent prendre la forme d'un processeur dont le jeu d'instructions a été spécialisé. La conception de processeurs personnalisés s'effectue principalement dans des langages de type HDL, mais est une tâche longue et complexe. Permettre la conception automatique de ces processeurs à partir d'une description avec un plus haut niveau d'abstraction rendrait cette tâche plus efficace. Cependant, les outils de HLS actuels ne permettent pas d'obtenir facilement des micro-architectures de processeurs efficaces. Des outils, tels que SpecHLS, permettent en revanche de générer à partir d'un simulateur de jeu d'instruction, une description en C++ du comportement du processeur pouvant, grâce à la HLS, être synthétisé vers un processeur pipeliné spéculatif efficace. Cependant, ces outils étant prévu pour concevoir des processeurs RISC à registres, il est nécessaire d'ajouter des étapes supplémentaires afin de pouvoir concevoir des processeurs à pile ou encore des processeurs CISC. En effet, il existe plusieurs choix possibles d'implémentations de pile matériel qui peuvent être compatible avec la conception d'un processeur en utilisant SpecHLS. Il restera par la suite à comparer ces piles et à automatiser l'utilisation de ces piles à partir d'un simulateur de jeu d'instructions de haut-niveau, n'ayant pas nécessairement conscience des choix qui seront fait. Ce choix pourra être laissé à l'utilisateur ou bien automatiquement choisit afin de maximiser l'efficacité du processeur tout en minimisant l'utilisation des ressources matérielles. D'autre part, pour pouvoir ajouter des instructions CISC aux processeurs conçu avec SpecHLS, il est nécessaire d'être capable de donner à SpecHLS un code dont toutes les itérations de la boucle décrivant le comportement du processeur effectuent le même ordre de grandeur de travail. Un tel code peut être obtenu en décrivant les instructions complexes naturellement

à l'aide de boucles, puis en transformant ces nids de boucles pour obtenir un seul niveau d'imbrication. Nous avons appelé la transformation utilisée ici le *unpipeline* et nous l'avons automatisée. Par la suite il restera à mesurer l'impact du *unpipeline* sur des circuits conçus avec la HLS, que ce soit en utilisant SpecHLS ou même seul. Obtenir une unique boucle permettrait également de pipeliner le résultat, alors qu'il n'est pas toujours possible de pipeliner efficacement un nid de boucle. D'autres version du *unpipeline* pourront également être proposées par la suite, car l'utilisation d'une FSM semblable à celle proposé dans la Section 5.3 permet d'ajouter une sémantique temporelle au code et ainsi d'obtenir un contrôle plus fin du pipeline de la boucle.

## Références


[1] Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–6, 2022.

[2] J. Michael O'Connor, Marc Tremblay. PicoJava-I : The Java Virtual Machine in Hardware. Technical report, Sun Microelectronics, 1997.

[3] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. Bringing Webassembly to Resource-Constrained Iot Devices for Seamless Device-Cloud Integration. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 261–272, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54 :265–286, 1 2008.

[5] C. J. Tsai, H. W. Kuo, Z. Lin, Z. J. Guo, and J. F. Wang. A Java processor IP design for embedded SoC. *ACM Transactions on Embedded Computing Systems*, 14 :35, 2 2015.

[6] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Kaufmann/Elsevier, 5th edition, 1994.

[7] Xilinx. Introduction to High-Level Synthesis. XACC school, 2021.

[8] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. What You Simulate Is What You Synthesize : Designing a Processor Core from C++ Specifications. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[9] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39 :4229–4239, 11 2020.

[10] Jean Michel Gorius, Simon Rokicki, and Steven Derrien. SpecHLS : Speculative Accelerator Design using High-Level Synthesis. *IEEE Micro*, 2022.

[11] Peng Tu and David Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th International Conference on Supercomputing*, pages 414–423, 1995.

[12] Ghuloum, Anwar M. and Fisher, Allan L. Flattening and Parallelizing Irregular, Recurrent Loop Nests. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 58–67, New York, NY, USA, 1995. Association for Computing Machinery.