Rapport du Projet de Système digital n^{o} 3

Belghiti Ismael, Desfontaines Damien,
Geoffroy Guillaume, Maillard Kenji

25 janvier 2012

Table des matières

1 Introduction

Le projet du cours de Système Digital consiste à programmer une horloge en assembleur, ce programme devant être exécuté par un processeur "fait maison". On demandait en particulier d'écrire le processeur dans un langage de description de circuits digitaux synchrones.
Dans notre groupe, nous avons encodé les circuits dans un langage nommé *Rock*, puis nous l'avons compilé avec un compilateur nommé *Obsidian*. Ces circuits forment un microprocesseur qui suit la spécification de l'architecture MIPS en Rock, ce qui nous permet de coder l'horloge en assembleur MIPS, d'assembler ce code MIPS avec un assembleur fabriqué par nos soins et enfin de charger l'exécutable obtenu dans le microprocesseur.
On présentera d'abord le langage Rock, puis nous nous intéresserons à l'implémentation du compilateur, du microprocesseur, de l'assembleur, puis de l'horloge. La dernière partie contient des informations importantes pour l'utilisation pratique de notre travail.

2 Le langage rock

Le langage de description des circuits est un langage déclaratif basé sur la notion de bloc. Nous l'avons appelé *rock*. Il y a trois types de blocs différents :

- ▷ Les blocs de base : xor, and, or, mux, not, ground, vdd et reg (déclarés dans Analysis/baseBlocks.ml)
- ▷ Les blocs déclarés par l'utilisateur, qui peuvent être paramétrés par des entiers. Les entrées et les sorties de ces blocs sont des fils, et le corps des blocs est déterminé à partir de blocs plus petits précédemment déclarés.
- ▷ Les périphériques ou blocs externes.

Il est compilé à l'aide du logiciel *obsidian* dont l'organisation est schématisée dans la figure 1. Les sections suivantes présentent les aspects spécifiques du langage et de sa compilation.

2.1 Traitement lexical

Le langage contient deux mots clés **start** et **device** ainsi que les caractères $<$, $>$, (,), $\rightarrow$, **;**, **:**, **,**, **[**, **]**, **{**, **}**, **.**, **..**, $+$, **-**, *****, /, %, ^, **@**.
Les identifiants sont de l'une des deux formes suivantes :

NomBloc : un identifiant commençant par une majuscule, il sert à identifier les blocs et les variables de blocs.

NomFil : identifiant commençant par une minuscule, il sert à identifier les noms de fils et de variables entières.

L'opération d'analyse lexicale se charge aussi d'effacer les commentaires qui sont soit de la forme « (∗ » « ∗) » pour les commentaires multi-lignes, soit de la forme « # » pour les commentaires sur une seule ligne.

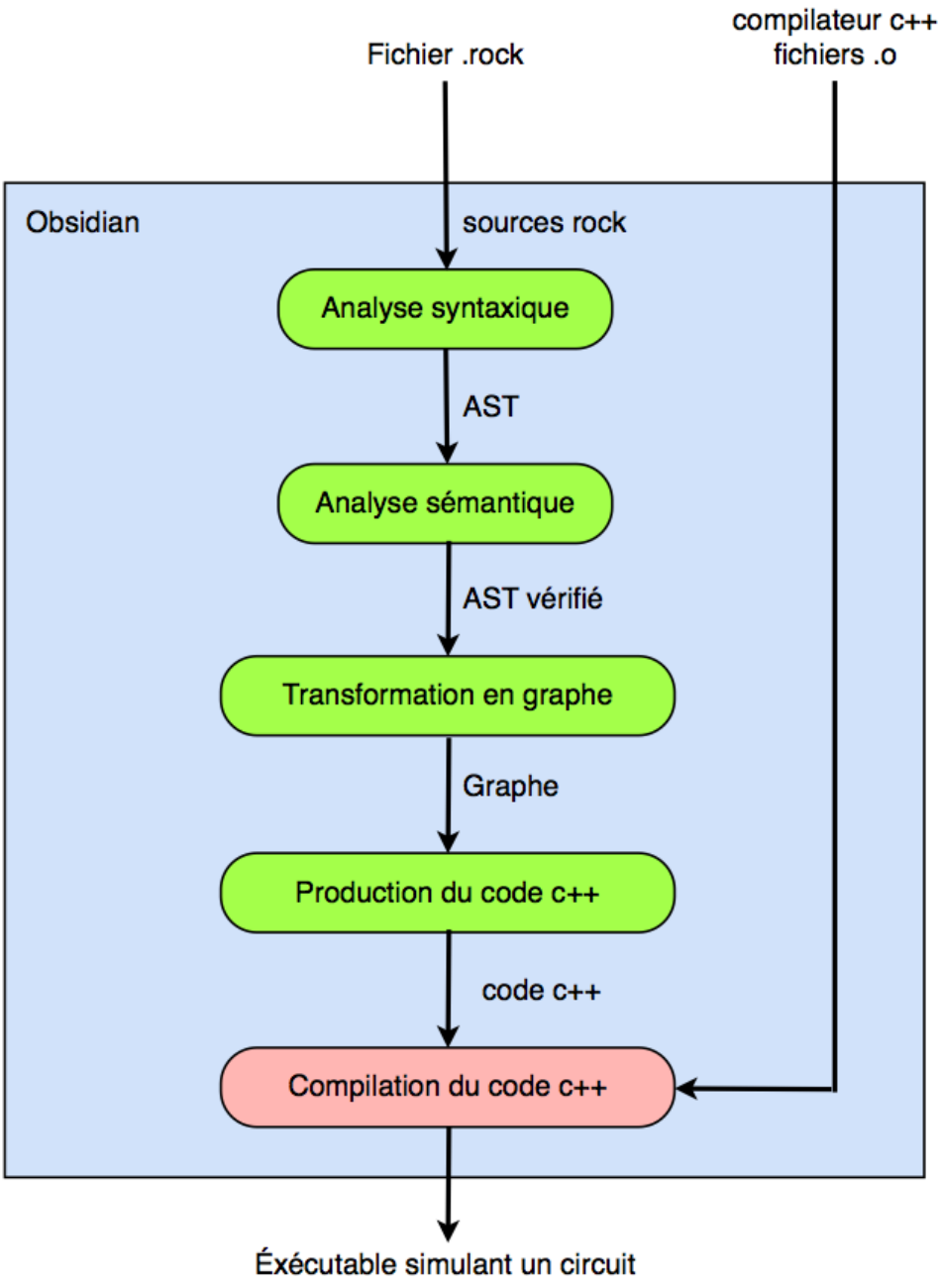

FIGURE 1 – Architecture d'Obsidian

2.2 Syntaxe du langage

Le langage suit la grammaire suivante :

fichier	::=	instruction* BlocEntrée instruction*
BlocEntrée	::=	**start** NomBloc < ConstanteEntière, ... , ConstanteEntière >
instruction	::=	BlocDéfinition | PériphériqueDéfinition
BlocDéfinition	::=	NomBloc Paramètres ? Arguments ? Instance* → Sorties ;
PériphériqueDéfinition	::=	**device** NomBloc Paramètres
Paramètres	::=	< Motif, ... , Motif >
Motif	::=	$a * n + k + b$ | $a * n + b$ | b

(avec a, b des constantes entières, $a \neq 1$ et n, k des identifiants de variables)

Arguments	::=	(DeclarationFil, ... , DeclarationFil)
DeclarationFil	::=	NomFil | NomFil [ConstanteEntière]
Instance	::=	NomBloc NomVariableBloc (Fil, ... , Fil)
Fil	::=	NomFil | NomVariableBloc **.** NomFil | NomFil [ConstanteEntière] | NomFil [ConstanteEntière **..** ConstanteEntière] | **$** (0|1)+ | { Fil, ... , Fil }
Sorties	::=	DeclarationFil : Fil, ... , DeclarationFil : Fil

2.3 Principe des blocs

Les blocs sont les unités structurantes du langage. Ils sont séparés en 3 catégories :

- ▷ Les blocs de base qui sont fournis par le compilateur (`Xor`, `And`, `Reg`, `Mux`, `Vdd`, `Gnd`, etc).
- ▷ Les blocs classiques qui définissent la majeure partie du circuit.
- ▷ Les périphériques qui permettent d'introduire des éléments externe dans le langage.

Les blocs classiques se construisent par composition. Ils ont en général la forme suivante :

```
BlocDéfini < params > (arg1, ..., argn)
   Bloc1  Instance1(arg1', ... , argp')
   Bloc2  Instance2(arg1'', ... , argq'' )
   -> sortie1 : fil1, ... , sortiek : filk ;
```

2.4 Les entiers

Les constantes entières sont présentes dans les paramètres - pour permettre de coder des blocs récursivement - et dans les fils - pour pouvoir préciser leur taille.

On peut les écrire tout aussi bien en hexadécimal (préfixe « $0x$ » ou « $0X$ »), en octal (préfixe « $0o$ » ou « $0O$ »), en binaire (préfixe « $0b$ « ou « $0B$ ») ou en décimal (sans préfixe). Ils supportent les opérations suivantes :

- ▷ $+$ l'addition
- ▷ $-$ la soustraction
- ▷ $*$ le produit
- ▷ / le quotient (division euclidienne)
- ▷ % le reste (division euclidienne)
- ▷ ^ l'exponentiation

2.5 Les fils

Les fils présents dans rock sont des fils épais. Ils sont les seuls moyens de communication entre blocs. Ils sont passés en arguments lors de la construction d'une instance et retournés à l'aide de la syntaxe `nom du bloc . nom du fil en sortie`. Les fils supportent deux opérations :

- ▷ l'extraction d'un sous-fil se fait via la syntaxe $a[\ p\ ..\ q\ ]$ et crée un nouveau fil de largeur $q-p+1$ qui est branché sur les fils sous-jacents de a indicés de p à q.
- ▷ la fusion de fils se fait via la syntaxe $\{a_1, a_2, ..., a_n\}$ et produit un fil dont la largeur est la somme des largeurs des a_i.

En plus de ces opérations, du sucre syntaxique a été rajouté pour améliorer l'expérience utilisateur :

- ▷ La syntaxe $a[n]$ est équivalente à $a[n..n]$ et produit donc un fil de taille 1.
- ▷ La syntaxe du type $01101 est équivalente à :

  ```
  Gnd G
  Vdd V
  ... { G.o, V.o, V.o, G.o, V.o } ...
  ```

  c'est à dire à un fil où chaque 1 est remplacé par un fil portant toujours la valeur 1 (instance de `Vdd`) et chaque 0 par un fil portant toujours la valeur 0 (instance de `Gnd`).

3 Concepts clés du langage

Le langage rock est muni de plusieurs concepts qui en font un outil maniable, utile, et efficace : citons notamment la récursion, les périphériques et les redéfinitions locales d'horloges.

3.1 Motifs et récursion

Le langage ne dispose pas des boucles classiques (`for`, `while`) ni de structures conditionnelles (`if`) pour manipuler le flot d'exécution. Il est par contre doté d'un système puissant de récursion et de reconnaissance de motifs. Il y a deux types de motifs :

- ▷ les motifs constants qui n'acceptent que leur valeur.
- ▷ les motifs de la forme $a * n + b$, avec a, b constants et n variable, qui acceptent les entiers p tels que $p \equiv b \bmod a$ et on a alors $n = \frac{p-b}{a}$.
- ▷ les motifs de la forme $a*n+k+b$, avec $a \neq 1$, b constants et k, n variables, qui acceptent tous les entiers p, et posent $k \equiv p - b \bmod a$, $0 \leq k < a$ et $n = \frac{p-b-k}{a}$.

Étant donnés un entier p et une liste ordonnée de motifs $(m_i)_i$, les motifs fonctionnent selon le principe suivant :

- ▷ l'entier est testé sur le i^emotif (en prenant au départ $i = 0$),
- ▷ s'il est accepté, les variables correspondant au motif sont assignées si nécessaire,
- ▷ et s'il est rejeté, on passe au motif suivant et on recommence.

Si on ne trouve aucun motif convenable, le compilateur lève une erreur.

Un tel système de motifs permet différents types de récursions. Ainsi, on peut :

- ▷ itérer sur tous les entiers avec les motifs $< n + 1 >$ et $< 0 >$,
- ▷ itérer seulement sur les puissances de deux avec les motifs $< 2 * n >$ et $< 1 >$,
- ▷ ou encore travailler sur des congruences modulo 3 avec les motifs $< 3*n >$, $< 3 * n + 1 >$ et $< 3 * n + 2 >$.

Ce système est néanmoins sujet à des limitations théoriques : il faut s'assurer que les calculs et la compilation se terminent ; et notamment qu'il n'y a pas de chaîne infinie de blocs. Pour cela, une contrainte supplémentaire a été ajoutée : à l'intérieur d'un bloc ayant comme paramètres les entiers $(n_i)_i$, seuls des blocs avec des paramètres plus petits (au sens de l'ordre sur $\mathfrak{N}$) peuvent être instanciés.

3.2 Les périphériques

Les périphériques sont un moyen de définir de nouvelles portes en plus des portes de base. On peut les utiliser pour simuler des mémoires, ou des périphériques d'entrée/sortie.

Tous les périphériques ont la même interface (les mêmes fils d'entrée et les mêmes fils de sortie) :

▷ Entrée :

Address	32 bits
Data	32 bits
Write mode	1 bit
Byte enables	4 bits
Enable interrupt	1 bit
Interrupt processed	1 bit

▷ Sortie :

Data	32 bits
Interrupt request	1 bit

Les périphériques peuvent contenir du code arbitraire, et peuvent donc a priori gérer leurs entrées et sorties de n'importe quelle manière. Cependant, ils sont faits pour simuler des périphériques mappés en mémoire : Les entrées `Address`, `Data` et `Write mode` et la sortie `Data` fonctionnent comme leur nom l'indique (l'entrée `Data` contient les données à écrire si `Write mode` = 1, la sortie `Data` contient les données lues). L'entrée `Byte enables` indique quels octets il faut prendre en compte :

- ▷ Si à l'adresse 0 se trouve la suite d'octets 11 22 33 44, et si l'on essaye de lire à l'adresse 0 avec `Byte enables` = \$1010, la valeur lue correspondra à la suite d'octets 11 00 33 00.
- ▷ Si à l'adresse 0 se trouve la suite d'octets 11 22 33 44, et si l'on essaye d'écrire 55 66 77 88 à l'adresse 0 avec `Byte enables` = \$1010, on trouvera à l'adresse 0 la suite d'octets 55 22 77 44.

Les périphériques sont pensés comme des registres, c'est-à-dire que si l'on écrit l'adresse où l'on veut lire au cycle t, on pourra lire le résultat au cycle t+1. A posteriori, il aurait été plus pratique de donner le choix entre les deux modes de fonctionnement (lecture instantanée et lecture retardée d'un cycle), car lorsqu'un périphérique n'est pas utilisé pour couper une boucle combinatoire, on perd simplement un cycle à attendre.

On déclare les types de périphériques de la façon suivante :

```
device Memory<size>
```

Puis, le type de périphérique peut être instancié comme n'importe quel autre type de bloc, par exemple, le circuit suivant :

```
Start
    Memory<5> Ram($0000...0000, $0000...0000, $1111, $0, $0)
    -> out[32] : Ram.dtata;

start Start
```

est constitué d'une mémoire sur $2^5 = 32$ octets, et sa sortie correspond au mot de 32 bits contenu à l'adresse 0 dans la mémoire. Si la mémoire est prévue pour initialiser tous ses octets à 0, la sortie de ce circuit sera :

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
...
```

Et le circuit suivant :

```
Start
    Memory<5> Ram($0000...0000, $1111...1111, $1100, $1, $0)
    -> out[32] : Ram.dtata;

start Start
```

aura pour sortie :

```
00000000000000000000000000000000
11111111111111110000000000000000
11111111111111110000000000000000
11111111111111110000000000000000
...
```

On définit les types périphériques de la façon suivante. Chaque type de périphérique est défini par une classe C++, qui doit hériter de la classe device :

```
class device
{
public :
  virtual ~device();
  virtual unsigned int cycle (unsigned int address, unsigned int data,
                              char byte_enables, bool write_enable,
                              bool interrupt_enable, bool iack, char *
                              irq)=0;
};
```

Chaque périphérique doit également fournir une méthode statique `make`, qui construit un objet de cette classe et renvoie un pointeur vers cet objet, upcasté en `device*`. Cette méthode prend un argument (de type `int`) par paramètre du périphérique. Par exemple, pour le périphérique `Memory`, la méthode `make` a le prototype suivant :

```
device * make (int size)
```

Pour chaque périphérique du circuit, sa méthode `cycle` est appelée à chaque cycle, avec comme arguments les valeurs lues dans les fils d'entrée.

3.3 Les redéfinitions d'horloges (ou enables)

Les redéfinitions d'horloges, mot savant pour "enables", permettent de redéfinir localement l'horloge du circuit : le circuit suivant :

```
Count_mod_2
    Not N(R.o)
    Reg R(N.o)
    -> o : R.o ;

Start
    Count_mod_2 C1
    Count_mod_2 @ C1.o C2
    -> o : C2.o ;

start Start
```

contient deux compteurs modulo 2 : C1 et C2, mais à l'intérieur du second, l'horloge est localement redéfinie, de sorte que le second compteur n'est actif qu'au cycles où la sortie du premier vaut 1, c'est-à-dire un cycle sur deux. La sortie du circuit est donc :

```
0
0
1
1
0
0
1
1
...
```

Lorsqu'une porte n'est pas active, le code C++ correspondant n'est simplement pas exécuté : la sortie de la porte garde sa valeur du cycle précédent, et s'il s'agit d'un registre, son contenu n'est pas modifié. Dans le cas d'un périphérique, sa méthode `cycle` n'est pas appelée.

Si on redéfinit l'horloge d'un bloc, on redéfinit l'horloge de toutes les portes et de tous les blocs qu'il contient. Si ce bloc lui-même redéfinit l'horloge de l'un de ses composants, celui-ci ne sera actif que lorsque son bloc parent l'est et que son bloc parent l'active. Par exemple, on peut utiliser les redéfinitions d'horloges pour programmer un compteur modulo 2^n. Ainsi, le circuit suivant :

```
Count<1>
    Reg M(N.o)
    Not N(M.o)
    -> out : M.o,
       carry : M.o;
```

```
Count<n>
    Count<1> Low
    Count<n-1> @ Low.carry High
    And A(Low.carry, High.carry)
    -> out[n] : {Low.out, High.out},
       carry : A.o;

Start<n>
    Count<n> C
    -> out[n] : C.out;

start Start<3>
```

a pour sortie :

```
000
100
010
110
001
101
011
111
000
100
010
...
```

L'intérêt majeur du système des redéfinitions d'horloges est qu'il permettde gagner du temps en n'exécutant que les parties du circuit qui sont utiles au cycle en cours (en plus de rendre certains codes plus simples, comme celui du compteur modulo 2^n). Par exemple, grâce aux enables, on peut définir un bloc `Ram` de manière (à peu près) efficace :

```
Mux_scalar <1> (selector, input0[1], input1[1])
    Mux M(selector, input0, input1)
    -> o : M.o;

Mux_scalar <n> (selector, input0[n], input1[n])
    Mux_scalar<n-1> Low(selector, input0[0..n-2], input1[0..n-2])
    Mux High(selector, input0[n-1], input1[n-1])
    -> o[n] : {Low.o, High.o};

Ram <0,1> (data[1], write)
    Mux M(write, R.o, data[0])
    Reg R(M.o)
    -> data[1] : R.o;
```

```
Ram <0,d> (data[d], write)
    Ram<0,d-1> Low(data[0..d-2], write)
    Ram<0,1> High(data[d-1], write)
    -> data[d] : {Low.data, High.data};

Ram <1,d> (address[1], data[d], write)
    Not N(address[0])
    Ram<0,d> @ address[0] Low(data, write)
    Ram<0,d> @ N.o        High(data, write)
    Mux_scalar<d> M(address, Low.data, High.data)
    -> data[d] : M.o;

Ram <a,d> (address[a], data[d], write)
    Not N(address[a-1])
    Ram<a-1,d> @ N.o          Low(address[0..a-2], data, write)
    Ram<a-1,d> @ address[a-1] High(address[0..a-2], data, write)
    Mux_scalar<d> M(address[a-1], Low.data, High.data)
    -> data[d] : M.o;
```

Ainsi, pour une mémoire de 2^a mots de d bits, le temps de calcul pour un cycle est en $O(a * d)$, au lieu du $O(2^a * d)$ que l'on aurait eu sans cette possibilité.

3.4 Exemples de codes et de blocs classiques

Voici quelques exemples de blocs que l'on peut définir avec ce langage.

```
### Manipulations de fils épais ###
```

On peut effectuer diverses opérations sur les fils épais, par exemple le renversement :

```
Rev<1> (in[1])
    -> out[1] : in[0];

Rev<n> (in[n])
    Rev<n-1> High(in[1..n-1])
    -> out[n] : {High.out, in[0]};
```

On peut aussi étendre les opérations à des fils épais, bit-à-bit...

```
And_vect<1> (a[1],b[1])
    And A(a[0],b[0])
    -> o : A.o;

And_vect<n> (a[n],b[n])
    And Low(a[0],b[0])
    And_vect<n-1> High(a[1..n-1],b[1..n-1])
    -> o[n] : {Low.o, High.o};
```

ou à la manière d'un produit par un scalaire...

```
And_scalar<1> (a,b[1])
    And A(a,b[0])
    -> o : A.o;

And_scalar<n> (a,b[n])
    And Low(a,b[0])
    And_scalar<n-1> High(a,b[1..n-1])
    -> o[n] : {Low.o, High.o};
```

ou encore à la manière d'un fold :

```
And_fold <1> (a[1])
    -> o : a;

And_fold <n> (a[n])
    And_fold<n-1> Tail(a[0..n-2])
    And Head(a[n-1], Tail.o)
    -> o : Head.o;
```

```
### Constantes ###
```

On peut associer automatiquement à un entier n un fil épais représentant n sur p bits :

```
Bin<0,1>
    -> o : $0;
Bin<1,1>
    -> o : $1;

Bin<2*n,p>
    Bin<n,p-1> High
    -> o[p] : {$0, High.o};
Bin<2*n+1,p>
    Bin<n,p-1> High
    -> o[p] : {$1, High.o};
```

```
### Compteurs modulo n ###
```

On peut tout d'abord définir un compteur modulo 2^n :

```
Count<1> (enable)
    Reg R(X.o)
    Xor X(R.o,enable)
    And A(R.o,enable)
    -> out : R.o,
       carry : A.o;
```

```
Count<n> (enable)
    Count<n-1> Low(enable)
    Count<1> High(Low.carry)
    -> out[n] : {Low.out, High.out},
       carry : High.carry;
```

On peut aussi ajouter un argument "reset" au compteur :

```
Count_reset<1> (e, r)
    Reg Mem(New_value.o)

    Not Not_reset (r)
    And New_value (Result.o, Not_reset.o)

    Xor Result(Mem.o,e)
    And Carry(Mem.o,e)
    -> o : Mem.o, c : Carry.o;

Count_reset<n> (e, r)
    Count_reset<n-1> L(e, r)
    Count_reset<1> H(L.c, r)
    -> o[n] : {L.o, H.o}, c : H.c;
```

Et à partir de là, on peut définir un compteur modulo n sur p bits :
Il faut d'abord définir une opération "égalité" sur des fils épais :

```
Xor_vect<1> (a[1],b[1])
    Xor X(a[0],b[0])
    -> o : X.o;
Xor_vect<n> (a[n],b[n])
    Xor Low(a[0],b[0])
    Xor_vect<n-1> High(a[1..n-1],b[1..n-1])
    -> o[n] : {Low.o, High.o};

Not_vect<1> (a[1])
    Not N(a[0])
    -> o : N.o;
Not_vect<n> (a[n])
    Not Low(a[0])
    Not_vect<n-1> High(a[1..n-1])
    -> o[n] : {Low.o, High.o};

Is_equal<n> (a[n],b[n])
    Xor_vect<n> Not_equal(a,b)
    Not_vect<n> Equal(Not_equal.o)
    And_fold<n> All_equal(Equal.o)
    -> o : All_equal.o;
```

Et le compteur :

```
Counter<n,p> (e)
    Count_reset<p> C(e, Reset.o)
    Bin<n-1,p> Mod
    Is_equal<p> Reset(C.o,Mod.o)
    -> o[p] : C.o;
```

4 Spécificités techniques d'Obsidian

Le compilateur Obsidian s'occupe de transformer un fichier rock en exécutable. Il est codé en OCaml, langage choisi en raison de ses capacités puissantes de manipulation d'expressions symboliques. Il génère du code C++ qui est ensuite compilé par un compilateur C++ classique, ce qui permet d'avoir de bien meilleures performances que si l'on avait produit nous-mêmes du code assembleur. Dans la suite de cette section, on détaillera les particularités techniques et/ou algorithmiques de chaque étape.

Les principales étapes du compilateur sont :

1. l'analyse lexicale et syntaxique (lexer, parser)
2. l'analyse sémantique
3. la transformation de l'AST en graphe
4. et la génération du code C++.

4.1 Analyse lexicale et syntaxique

On a utilisé les librairies ocamllex et menhir pour écrire le lexer et le parser. Les symboles reconnus par le lexer ainsi que la grammaire employée dans le parser ont déja été vus dans la présentation du langage.

L'arbre de syntaxe abstraite (ou AST) est formé de la manière suivante.

- ▷ À la racine, se trouve un tableau associatif qui à chaque nom de bloc associe sa définition, une liste des périphériques déclarés avec leur nombre de paramètres ainsi que du nom et des paramètres du bloc de départ (celui qui englobe tout le circuit simulé).
- ▷ Les définitions de blocs, sont des enregistrements dont les champs sont les paramètres du bloc (motifs), les entrées (fils), les instances (création de sous-bloc) et les sorties (fils). Ceux-ci sont eux même des types enregistrement jusqu'à arriver au atomes (nom de fil, nom de bloc, entier, etc).

L'AST est paramétré par un module définissant les entiers à utiliser dans l'AST (il s'agit donc d'un foncteur). L'intérêt de paramétrer la définition des entiers à l'intérieur de l'AST et de pouvoir produire à la sortie du parser un AST dans lequel la plupart des « entiers » sont en fait des *expressions entières* pas encore évaluées et qui seront transformées à l'étape suivante.

4.2 Analyse sémantique

L'analyse sémantique est un des morceaux les plus lourds d'Obsidian, mais aussi ce qui fait du langage un outil utilisable car sûr. En effet l'analyse statique pratiquée au niveau de l'analyse sémantique permet d'éliminer presque toutes les erreurs de sémantique au niveau du circuit. Concrètement, l'analyse sémantique vérifie :

- ▷ que tous les blocs employés ont été déclarés et que dans chaque bloc toutes les variables (que se soit de bloc ou de fil) ont été définies,
- ▷ que tous les fils sont corrects, c'est à dire qu'il n'y a jamais de fil vide ou d'épaisseur négative
- ▷ que les branchements opérés entre deux fils sont bien réalisés avec des fils de même largeur,
- ▷ que chaque création de sous-fils (avec la syntaxe [..]) est bien fait avec un intervalles valide (c'est à dire inclus dans l'intervalle $[|0; n-1|]$ où n est la taille du fil original),
- ▷ que chaque application de paramètres lors d'une instance est bien acceptée par un certain motif,
- ▷ et que la récursion est bien fondée : pas de suite infinie de blocs dans le circuit (vérification cruciale pour l'étape de transformation en graphe).

En particulier, l'analyse sémantique s'occupe d'une opération que l'on appelle réification par analogie avec l'opération que font certains compilateurs pour gérer le paramétrage polymorphique (par exemple avec les templates en C++). Il s'agit en fait de transformer l'AST ne contenant que des entiers « formels » (c'est à dire des *expressions entières*) en "vrais" entiers. Par exemple, le code suivant :

```
And_vect<1> (a[1],b[1])
     And A(a[0],b[0])
     -> o : A.o;

 And_vect<n> (a[n],b[n])
     And Low(a[0],b[0])
     And_vect<n-1> High(a[1..n-1],b[1..n-1])
     -> o[n] : {Low.o, High.o};

 start And_vect<4>
```

génère d'abord l'AST contenant les définitions de `And_vect<1>` et `And_vect<n>` ; mais après réification, il contiendra les définitions de `And_vect<1>`, `And_vect<2>`, `And_vect<2>`, `And_vect<4>`.

Cette multiplication de symboles peut paraître très lourde, mais elle permet une analyse en détail de chaque bloc. De plus, l'étape suivante du compilateur fait disparaître les redondances.

Il faut aussi noter que l'analyse sémantique ne vérifie pas l'absence totale de cycle combinatoire. Pour cela, il faut attendre que l'AST soit transformé en

graphe. Cette vérification est la seule qui doivent attendre la phrase de génération de code pour être opérée.

4.3 La transformation en graphe

À la sortie de l'analyse sémantique, un AST réifié est obtenu. La transformation en graphe se fait en deux étapes :

- ▷ Un premier parcours de l'AST détermine la liste des portes que contiendra le graphe. Cette liste est ensuite transformée en tableau (cette transformation permet notamment d'obtenir un accès aléatoire à chaque élément).
- ▷ Un deuxième parcours de l'AST se charge de *brancher* les fils entre les portes. Pour cela on considère l'ensemble des *bouts de fil* identifiables (c'est à dire l'ensemble des variables de fils en distinguant selon la portée) et on construit la relation d'équivalence « est branché avec » avec un Union-find. Les classes d'équivalence résultantes à la fin du parcours sont exactement l'ensemble des fils.

Au cours de cette opération, les blocs qui ne sont pas des blocs de base ou des périphériques sont éliminés, ou plutôt oubliés. Le résultat est un graphe implémenté sous la forme d'un tableau où à chaque indice est associé :

- ▷ le type de la porte
- ▷ et un tableau contenant pour chaque sortie de la porte, les portes avec lesquelles elle est branchée.

Un certain nombre d'informations, comme le nombre total de registres, ou de périphériques est aussi collecté durant cette phase - ceci afin de faciliter la génération du code.

4.4 La génération du code c++

Une fois que l'on a construit le graphe représentant le circuit, on effectue un tri topologique en "coupant" au niveau des registres et des périphériques (afin d'autoriser les boucles qui passent par ces portes). Ensuite, on écrit un code C++ qui déclare un tableau contenant les sorties des portes et qui, à chaque cycle, calcule la sortie de chaque porte à partir des sorties des portes en amont (ce renseignement est donné par le tri topologique). Les registres (et les périphériques) sont traités de manière un peu différente : leur nouvelle valeur n'est pas stockée directement dans le tableau des sorties, mais dans un autre tableau. Au début du cycle suivant, les nouvelles valeurs des registres sont copiées depuis cet autre tableau dans le tableau des sorties.

5 Le microprocesseur

L'architecture du microprocesseur a été modifiée depuis le dernier rapport. Nous utilisons maintenant des périphériques pour gérer la mémoire, les registres et le programme. Le nombre d'instructions supportées a été considérablement augmenté. D'une architecture à trois cycles par instruction, nous sommes passés

à une architecture à 4 cycles par instruction permettant une grande sécurité des échanges du contrôleur, les périphériques assurant parallèlement une protection contre les boucles combinatoires.

5.1 L'Unité Arithmétique et Logique (ALU)

L'ALU que nous avons réalisé prend en entrée deux entiers sur 32 bits ainsi qu'un code sur 4 bits décrivant l'opération à effectuer. Il renvoie alors un résultat sur 32 bits. Il s'agit d'un bloc purement combinatoire.

Notre ALU supporte 16 opérations :

1. ET, OU, XOR, NAND, NOR pour les opérations logiques binaires,
2. $+,-,x,=,\neq,<,>,\leq,\geq$ pour les opérations arithmétiques,
3. une opération renvoyant le premier opérande (utile pour les opérations de shifts),
4. et une opération de concaténation 16 bit-16 bit (utile pour le *lui*).

D'autres opérations ont été implémentées, notamment les différentes formes de décalage ("shift") dans les deux sens, en version classique ou arithmétique. Néanmoins, seule l'opération de décalage classique vers la droite est utilisé par le reste du microprocesseur (il est utilisé avec le shift_amount du R-format).

Fonctionnement : Chaque opération de l'ALU est codée séparément. Puis, à la lecture de deux entiers, toutes les opérations possibles sont exécutées simultanément, et un filtre sélectionne le résultat voulu. Tout fonctionne en un seul cycle, et les circuits correspondant aux diverses opérations ont tous été codés récursivement.

5.2 Registres, Instruction Memory et Data Memory

Le gestionnaire de registres contient 32 registres de 32 bits chacun. Il s'agit d'un périphérique de type mémoire. Il est possible de faire plusieurs lectures simultanées (un nombre arbitrairement grand) en utilisant des périphériques lecteurs auxiliaires. Dans notre cadre, on accède à au plus à deux registres à la fois, un seul périphérique auxiliaire de lecture suffit donc.

Le bloc d'instructions est également un périphérique. Chaque instruction est stockée sur 32 bits. Nous suivons les différents format MIPS (R-format, I-format, J-format) mais nous traitons à part certaines opérations de ces formats pour faciliter et mieux diviser le travail du contrôleur, comme nous le verrons dans la section Instructions.

Enfin, la Data Memory peut prendre deux formes, selon que l'on se trouve dans le mode “classique” ou en mode “horloge” avec un affichage 7-segments.

5.3 Contrôleur

Le bloc de contrôle, présent dans notre projet sous le nom de "Contrôleur" a dans notre projet non seulement le rôle du bloc "Control" de l'architecture standard MIPS mais il s'occupe aussi de la synchronisation entre les différents cycles parcourus pour une instruction donnée : tous les autres blocs sont donc positionnés en étoile par rapport à lui. Cette synchronisation est obtenue et sécurisée par l'utilisation de filtres temporels gérant les trafics d'informations entre les différents blocs. Toutes les instructions sont exécutées sur quatre cycles.

Ce contrôleur agit différemment selon le format de l'instruction à effectuer, il contient donc plusieurs décodeurs. Nous avons 3 décodeurs correspondant aux formats MIPS habituels : un pour le R-format, un pour le I-format, un pour le J-format. Mais nous avons préféré traiter certains types d'instructions à part, car elles correspondent à des échanges spécifiques entre les différents blocs du microprocesseur. Ainsi, nous avons 4 décodeurs plus spécifiques : un pour les écritures en mémoire (`sb` et `sw`), un pour les lectures mémoire (`lb` et `lw`), un pour les `branch` et enfin un pour `jr`.

Ces décodeurs sont exécutés en parallèle mais le contrôleur agit comme un filtre et laisse transparaître uniquement les actions du décodeur correspondant à l'instruction en cours. Ce filtrage s'effectue selon la valeur de l'opcode de l'opération.

5.4 Instructions

Les instructions suivent un codage relativement classique. Décrivons les intervalles de bits de chacun des types d'instructions :

5.4.1 R-format

Ce format intervient par exemple dans les opérations comme `add`, `sub`, `mul`, `and`, ...

1 (*codage du R-format*)

- ▷ *0..5 : les 6 bits de l'opcode*
- ▷ *6..10 : adresse* `rs` *(opérande 1)*
- ▷ *11..15 : adresse* `rt` *(opérande 2)*
- ▷ *16..20 : adresse* `rd` *(résultat)*
- ▷ *21..25 : shift_amount*
- ▷ *26..31 : funct code*

Les 6 bits de l'opcode sont toujours nuls ainsi que les deux derniers chiffres du funct code. Les quatre premiers bits du funct code correspondent, eux, au code de l'opération concernée (c'est ce code qui est transmis à l'ALU). Enfin le shift_amount correspond au décalage d'un shift vers la droite (c'est-à-dire à une division par $2^{\text{shift\_amount}}$).

5.4.2 I-format

Le I-format correspond aux instructions du type `addi`, `subi`, `andi`...

2 (*codage du I-format*)

- ▷ *0..5 : les 6 bits de l'opcode*
- ▷ *6..10 : adresse* `rs` *(opérande 1)*
- ▷ *11..15 : adresse* `rt` *(résultat)*
- ▷ *16..31 : valeur immédiate (opérande 2)*

Les deux derniers bits de l'opcode sont toujours 11. Les quatre premiers bits de l'opcode correspondent, eux, au code de l'opération de l'ALU. La valeur immédiate est étendue à droite sur 32 bits en répétant son dernier bit, pour une gestion correcte des nombres négatifs.

5.4.3 J-format

On définit ici un format pour la seule instruction `j` (jump).

3 (*codage du J-format*)

- ▷ *0..5 : les 6 bits de l'opcode*
- ▷ *6..31 : adresse de la destination sur 26 bits*

L'opcode est toujours 111101.

5.4.4 Branch-format

Le Branch-format regroupe les différentes instructions de type branch comme `beq`, `bne`, `bgt`, `ble`...

4 (*codage du Branch-format*)

- ▷ *0..5 : les 6 bits de l'opcode*
- ▷ *6..10 : adresse* `rs` *(opérande 1)*
- ▷ *11..15 : adresse* `rt` *(opérande 2)*
- ▷ *16..31 : adresse relative en cas de branch*

Les deux derniers bits de l'opcode sont toujours 10 tandis que les quatre premiers bits correspondent au code de l'opération de l'ALU associée (par exemple le code de l'égalité pour un `beq`). L'adresse relative est définie par rapport à l'adresse qui suit l'instruction branch.

5.4.5 JR-format

L'instruction `jr` (jump register) bénéficie d'un format à part.

5 (*codage du JR-format*)

- ▷ *0..5 : les 6 bits de l'opcode*
- ▷ *6..10 : adresse du registre*
- ▷ *11..31 : bits inutilisés*

L'opcode est toujours 000001.

5.4.6 Format de lecture mémoire

Ce format correspond aux instructions `lw` et `lb`.

6 (*codage du format de lecture*)

- ▷ *0..5 : les 6 bits de l'opcode*
- ▷ *6..10 : adresse* `rs` *(registre contenant l'adresse de base)*
- ▷ *11..15 : adresse* `rt` *(résultat)*
- ▷ *16..31 : valeur immédiate de décalage*

L'opcode est 110001 pour `lw` et 100001 pour `lb`. En pratique, le deuxième bit de l'opcode est repassé en argument au byte_enable de la mémoire (pour spécifier la plage de lecture).

5.4.7 Format de l'écriture mémoire

Ce format correspond aux instructions `sw` et `sb`.

7 (*codage du format d'écriture*)

- ▷ *0..5 : les 6 bits de l'opcode*
- ▷ *6..10 : adresse* `rs` *(registre contenant l'adresse de base)*
- ▷ *11..15 : adresse* `rt` *(registre à stocker)*
- ▷ *16..31 : valeur immédiate de décalage*

L'opcode est soit 110101 pour `sw`, soit 100101 pour `sb`. Ici aussi, le deuxième bit de l'opcode est repassé en argument au byte_enable de la mémoire (pour spécifier la plage d'écriture).

6 L'assembleur

Le microprocesseur lit en entrée le code exécutable qui se trouve dans un fichier `ram` chargé en mémoire. La génération du fichier `ram` à partir des sources MIPS incombe donc à l'assembleur. L'assembleur a été codé en OCaml, sur la même architecture générale qu'Obsidian (analyse lexicale, syntaxique, sémantique, et génération du code).
La syntaxe du MIPS est assez simple à parser et à analyser, cependant quelques limitations ont été introduites :

- ▷ Il doit y avoir au plus une instruction ou étiquette par ligne.
- ▷ Le code doit se trouver obligatoirement dans le segment de texte (introduit par l'instruction `.text`), et les données dans le segment de données (introduit par `.data`).
- ▷ Par conséquent, les instructions `.word`, `.byte`, `.half`, `.ascii`, `.asciiz` et `.space` ne sont disponibles que dans le segment de données et le autres instructions commençant par un « . » ne sont pas présentes.
- ▷ Les instructions et pseudo-instructions ne sont pas toutes présentes : il n'y a pas de `syscall` par exemple (puisqu'il n'y a pas d'OS sur le micropro-

cesseur). Certaines pseudo-instructions comme `bgt` ou `ble` ont aussi été implémentées comme des vraies instructions.
- ▷ Les entrées/sorties sont réalisées à l'aide de deux étiquettes `clock_display` et `timestamp` qui sont des adresses mappées en mémoire dans le processeur.

Les conventions de nommage employées pour les registres sont celles appelées *O*32. Les registres disponibles sont donc : $zero, $at, $v0 - $v1, $a0 - $a3, $t0 - $t9, $s0 - $s7, $k0 - $k1, $gp, $sp, $fp et $ra.

La liste des pseudo-instructions implémentées est :
- ▷ `move` copie la valeur d'un registre dans un autre registre.
- ▷ `clear` met un registre à 0.
- ▷ `la` charge une étiquette dans un registre.
- ▷ `li` charge une constante de 32 bits dans un registre.
- ▷ `b` branche sans condition.
- ▷ `beqz` branche si le registre contient 0.
- ▷ `bnez` branche si le registre ne contient pas 0.
- ▷ `beqd` branche si le registre contient 42.

7 L'horloge

Le fichier horloge.s est le fichier assembleur qui code l'horloge (d'où son nom, incroyable, n'est-il pas ?). Son fonctionnement est le suivant :

1. On va chercher le timestamp courant dans l'adresse mémoire ayant comme label *timestamp*. Le format est celui du timestamp UNIX standard, accessible (par exemple avec la commande `date +%s`. Pour mémoire, il s'agit du nombre de secondes écoulées depuis le 1er janvier 1970 à 00 : 00 : 00.

2. On transforme ce nombre de secondes en nombre de jours en faisant une simple division (codée plus loin, car indisponible dans notre microprocesseur, à partir d'une multiplication, elle aussi recodée avec des additions - de façon logarithmique bien sûr).

3. En prenant le reste de la division précédente, et par divisions successives, il est très facile d'obtenir l'heure de la journée. Ce calcul est donc effectué puis affiché dans l'afficheur 7 segments (la méthode pour cela est décrite à part).

4. Puis, on doit calculer la date. Et là, c'est nettement plus technique. Il faut commencer par corriger les décalages dûs aux années bissextiles. Deux approches sont essentiellement possibles : considérer qu'une année "normale" est une année de 365 jours, et prévoir un cas précis pour certains timestamps correspondant aux quelques 29 février ; ou bien considérer qu'une

année "normale" dure 366 jours et se contenter de modifier en dur le timestamp en ajoutant l'équivalent d'un jour chaque 28 février à 23 : 59 : 59 d'une année non bissextile. C'est cette deuxième option qui a été choisie.

5. On calcule donc le nombre de jours à ajouter, par paquets de 4 ans (on rajoute 72 heures à chaque fois) et en corrigeant en bout de course. On obtient finalement un nombre de jours "artificiels" mais qui permet de traiter tous les cas d'un seul coup.
6. On calcule le mois et le jour en faisant 12 cas, faute de méthode plus simple pour s'adapter aux durées variables des différents mois. Pour le mois de février, un des avantages est que l'on peut toujours considérer qu'il dure 29 jours - la modification précédente du timestamp garantit que l'on ne peut pas tomber dans le cas "29 février" si on n'est pas une année bissextile.
7. On affiche enfin la date, et on revient au début pour récupérer le nouveau timestamp et recommencer. "Un programme n'est pas fait pour être arrêté."

Pour afficher la date, il faut d'abord transformer un nombre sur un octet en format "afficheur 7 segments" (où chaque bit représente l'état, allumé ou éteint, d'un des 7 segments). La traduction est codée en dur, dans deux tables (selon si les nombres considérés ont deux (cas le plus courant) ou quatre chiffres (pour les années par exemple)) nommées `two_digits_to_segments` et `year_to_segments`, et il suffit de chercher au bon endroit de ces tables (au début pour 00, dans les deux cases suivantes pour 01, etc.) pour avoir le renseignement voulu. Puis, on stocke le résultat en dur dans des cases mémoires reliées directement à l'afficheur 7 segments.

8 Utilisation des outils et organisation du code

L'ensemble du projet peut se compiler à partir de la racine à l'aide de la commande `make`. Avant toute compilation, il faut éventuellement modifier le compilateur C++ employé (g++ par défaut). Les différentes cibles présentes sont :

▷ **run :** lance l'horloge avec une configuration par défaut.

▷ **run-auto :** pour lancer l'horloge en mode "autonome" (c'est-à-dire en ajoutant une seconde à chaque fois que l'on a affiché l'heure, sans se soucier du défilement réel du temps).

▷ **run-fast :** pour voir défiler les jours (une seconde par cycle).

▷ **run-flash :** pour voir défiler les années.

▷ **obsidian :** compile le compilateur.

▷ **assembleur :** compile l'assembleur.

▷ **micro :** compile le microprocesseur en mode « horloge ».

▷ **horloge :** assemble l'horloge.

▷ **compiled-tests :** compile les tests de circuits présents dans Rock/Tests et les place dans compiled-tests.

▷ **run-facto :** calcule 7 ! en utilisant notre processeur.

▷ **clean :** efface tous les programmes.

L'organisation des dossiers est la suivante :

▷ **Asm :** contient les exemples en assembleur.

▷ **AssembleurDir :** contient le code pour l'assembleur.

▷ **ObsidianCompiler :** contient le code pour Obsidian.

▷ **Resources :** contient des fichiers nécessaires pour le fonctionnement d'Obsidian.

▷ **Rock :** contient les exemples en rock ainsi que le microprocesseur (fichier Processor/Micro.rock).

Tous les programmes sont accompagnés d'une aide accessible à l'aide de l'option -help permettant de voir les options disponibles. Pour charger un programme assembleur dans le microprocesseur, que ce soit avec `Micro` ou `MicroClassique`, il faut le compiler avec l'instruction suivante :
`.assembleur -o ram nom_du_fichier_assembleur.s`.

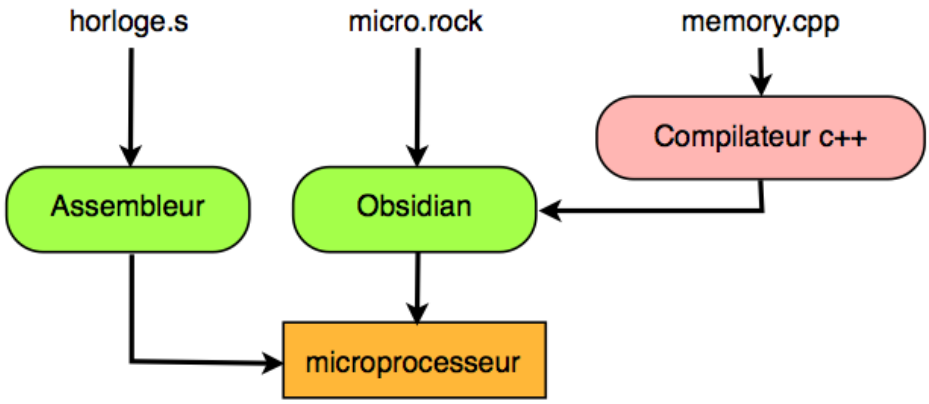

FIGURE 2 – Architecture d'Obsidian