Projet du cours
Système Digital

I. Belghiti, D. Desfontaines,
G. Geoffroy, K. Maillard

26 janvier 2012

1. Le langage Rock
2. Le compilateur Obsidian
3. Le microprocesseur
4. Horloge et calendrier

Sommaire

1. Le langage Rock
   - Description du langage
   - Concepts particuliers
2. Le compilateur Obsidian
3. Le microprocesseur
4. Horloge et calendrier

Description du langage

Le langage de description des circuits est un langage déclaratif basé sur la notion de blocs :

- Les blocs de base : `Xor`, `And`, `Or`, `Mux`, `Not`, `Gnd`, `Vdd` et `Reg`.
- Les blocs définis par l'utilisateur.
- Les périphériques ou blocs externes.

Il est compilé à l'aide du logiciel *Obsidian*.

Exemple d'emploi

```
HalfAdder ( a, b)
    Xor X( a, b)
    And A( a, b)
    -> o : X.o, c : A.o ;

ParallelAdder <1> ( a, b, c)
    HalfAdder H1( a, b)
    HalfAdder H2( c, H1.o)
    Or O( H1.c, H2.c) (a, b)
    -> o : H2.o, c : O.o ;

ParallelAdder <n> (a[n] , b[n], c)
    ParallelAdder<n-1> A (a[0..n-2], b[0..n-2], c)
    ParallelAdder<1> F (a[n-1], b[n-1], A.c)
    -> o[n] : { A.o, F.o }, c : F.c ;

start ParallelAdder < 3 >
```

Structure de l'additionneur

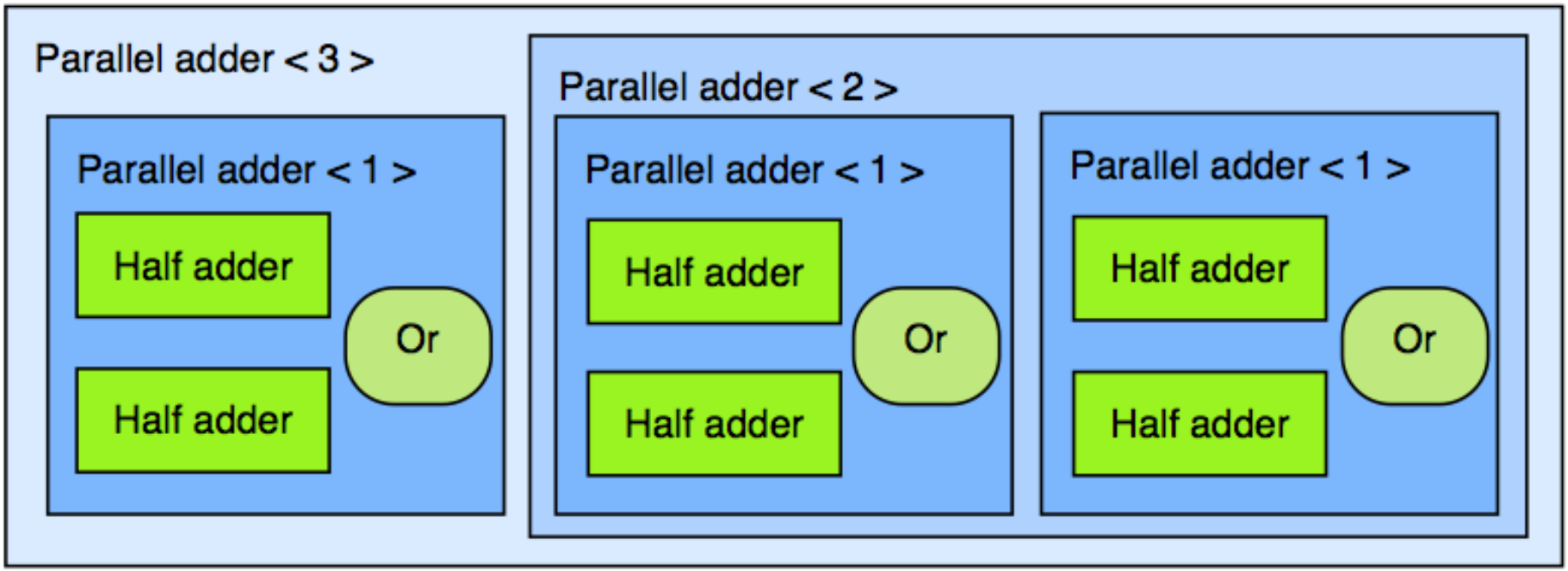

Figure: Additionneur parallèle sur 3 bits (Structure)

Graphe sous-jacent

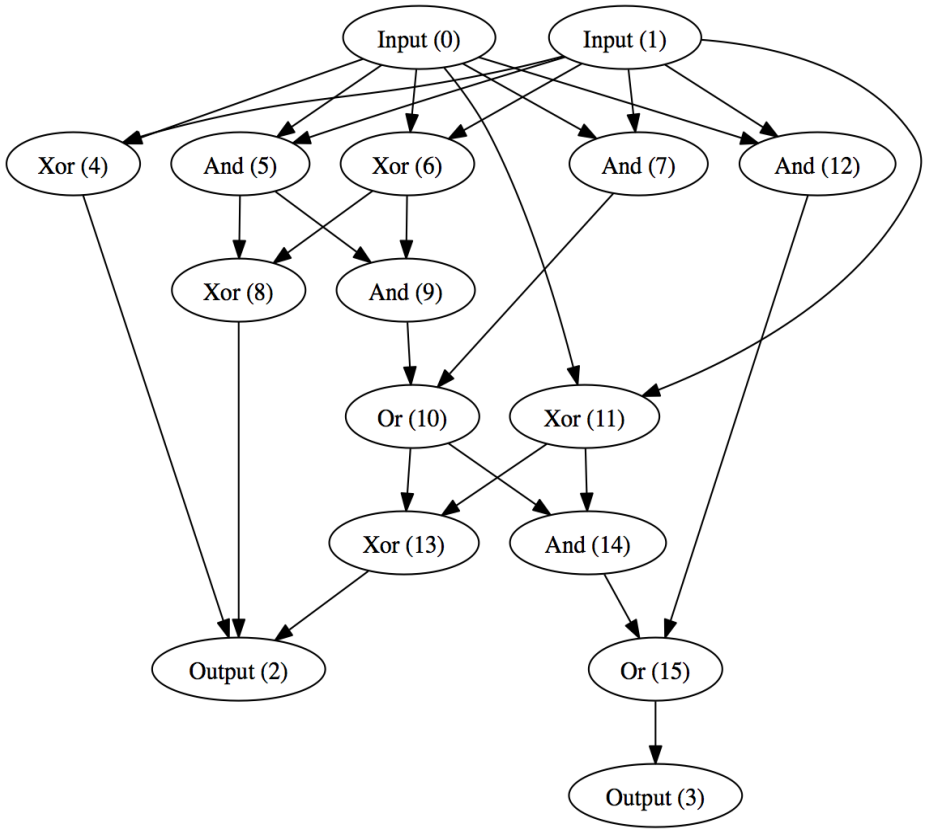

Figure: Additionneur parallèle sur 3 bits (Graphe)

Syntaxe du langage

Le langage suit la grammaire suivante :

fichier	::=	instruction* BlocEntrée instruction*
BlocEntrée	::=	**start** NomBloc $< n_1, \ldots, n_k >$
instruction	::=	BlocDéfinition
		| PériphériqueDéfinition
BlocDéfinition	::=	NomBloc Paramètres ?
		Arguments ? Instance* $\rightarrow$ Sorties ;
PériphériqueDéfinition	::=	**device** NomBloc Paramètres
Paramètres	::=	< Motif, ... , Motif >

Syntaxe du langage

Motif	::=	$a * n + k + b$
		| $a * n + b$
		| b
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

Les blocs

Les blocs 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 d'instances de blocs définis dans le reste des sources. On accède à aux sorties des blocs avec la syntaxe `nom_du_bloc.nom_du_fil_en_sortie`.

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

Les fils

Tous les fils sont épais. Ils 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 $\{a_1, a_2, \ldots, a_n\}$ 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 `{ G.o, V.o, V.o, G.o, V.o }` où `G` est une instance de `Gnd` et `V` une instance de `Vdd`.

Les motifs

Le flôt d'éxécution du programme est manipulé par de la reconnaissance de motif sur les paramètres d'un bloc.

Les motifs reconnus sont :

- 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 $n = \frac{p-b-k}{a}$.

Une légère phase de calcul symbolique permet de réduire un motif comme $7\hat{}3 + y + 3 * x - 21 + 2 * x$ en $5 * x + y + 322$ qui est bien valide.

La récursion

Un tel système de motifs permet différents types de récursion. 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 qui peut aussi ce faire avec le motif $< 3 * n + k >$.

Compteur modulo 2^n

Par exemple pour 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;
```

Les redéfinitions d'horloge

- On peut faire en sorte qu'un bloc ne reçoive le *top* de l'horloge qu'à certains cycles. Par exemple, dans le code suivant :

  ```
  Reg @ horloge R(valeur)
  ```

  le registre `R` prend en entrée le fil `valeur`, mais ne le prend en compte qu'aux cycles où le fil `horloge` est à `1`.

- Si on redéfinit l'horloge d'un bloc, on redéfinit l'horloge de tous les blocs qu'il contient.

- On peut cumuler les redéfinitions d'horloge.

Les redéfinitions d'horloge - Exemple

Ainsi, on peut réaliser un compteur modulo 2^n à l'aide de redéfinitions d'horloge :

```
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;
```

Les redéfinitions d'horloge - Avantages

- Le code est plus facile à écrire, plus clair, et souvent plus court.
- On gagne en vitesse d'exécution.

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) :

Entrées :		Sorties :	
Address	(32 bits)	Data	(32 bits)
Data	(32 bits)	Interrupt request	(1 bit)
Write mode	(1 bit)		
Byte enables	(4 bits)		
Enable interrupt	(1 bit)		
Interrupt processed	(1 bit)		

Les périphériques

- Les périphériques peuvent contenir du code arbitraire : à chaque périphérique correspond un objet, dont la méthode `cycle` est appelée à chaque cycle.
- Ils sont faits pour simuler des périphériques mappés en mémoire.
- Les périphériques fonctionnent comme les registres : si l'on écrit l'adresse au cycle t, on lit le résultat au cycle $t + 1$.
- A posteriori, il aurait été plus pratique de permettre de choisir le type de comportement.

Les périphériques

- On déclare les types de périphériques de la façon suivante :

```
device Memory<size>
```

- Puis, on peut les instancier normalement :

```
Memory<5> Ram($0000...0000, $1111...1111,
                $1011, $1, $0)
```

Ram.data vaut successivement :

```
00000000000000000000000000000000
00000000000000000000000000000000
11111111000000001111111111111111
11111111000000001111111111111111
...
```

Sommaire

1 Le langage Rock

2 Le compilateur Obsidian

3 Le microprocesseur

4 Horloge et calendrier

Organisation globale

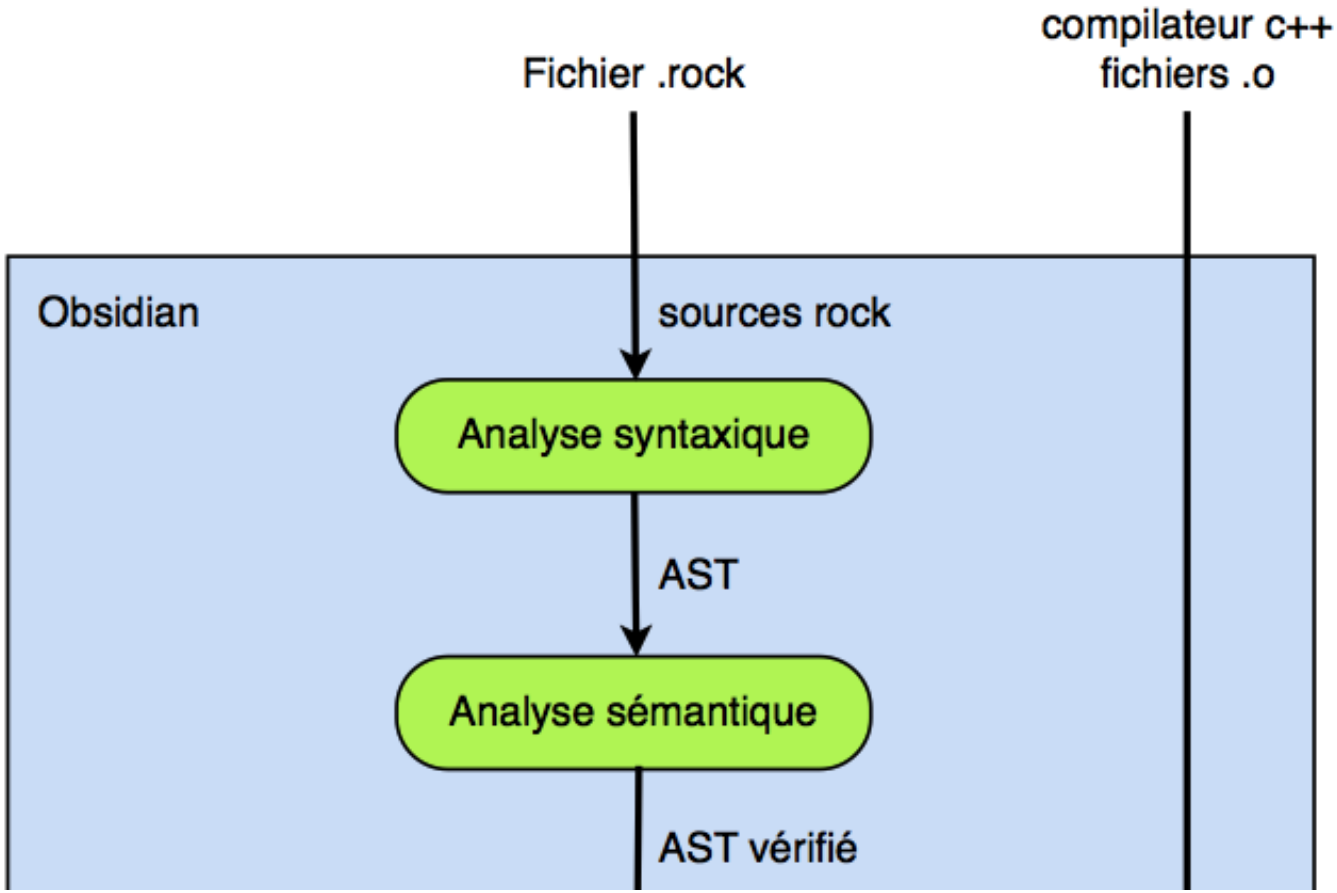

Figure: Architecture d'Obsidian

Organisation globale

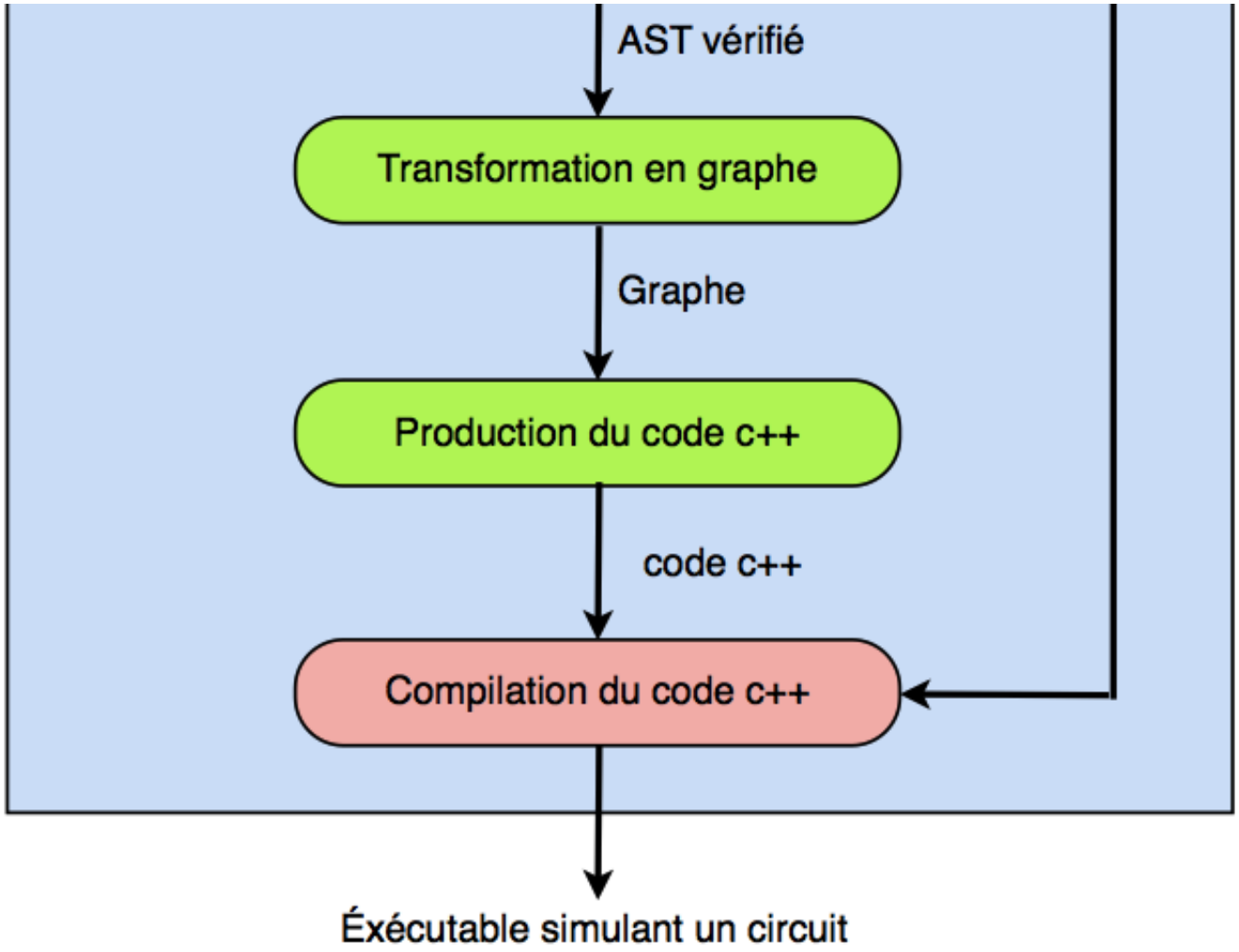

Figure: Architecture d'Obsidian

Analyse sémantique

L'analyse sémantique se charge d'analyser statiquement le circuit afin de repérer et de résoudre les problèmes qu'il pourrait contenir. En particulier, cette étape vérifie :

- que tous les noms de variable employés ont été déclarés et que leur emploi est cohérent avec leur nature (taille des fils, nom des sorties de blocs),
- que chaque application de paramètres lors d'une instance est bien acceptée par un certain motif,
- que la récursion est bien fondée.

L'AST est transformé de fond en comble : les paramètres et toutes les expressions en dépendant sont évalués lors de cette phase, en particulier les récursions sont explicitées.

Construction du 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.
- 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 » en utilisant un algorithme d'Union-find. Les classes d'équivalence résultantes à la fin du parcours sont exactement l'ensemble des fils.

Génération du code

- On effectue un tri topologique sur le graphe, en essayant de grouper les blocs où l'horloge est redéfinie de la même façon.
- On produit un code C++ qui simule le fonctionnement du circuit.
- Un tableau pour les sorties des portes, un tableau temporaire pour les registres et les périphériques.
- À chaque cycle, on calcule les sorties des blocs dans l'ordre donné par le tri topologique.
- Les sorties des registres et des périphériques sont mises à jour à la fin du cycle.

Sommaire

1 Le langage Rock

2 Le compilateur Obsidian

3 Le microprocesseur
- L'ALU
- Registres, Instruction Memory et Data Memory
- Contrôleur
- L'assembleur

4 Horloge et calendrier

Le microprocesseur

Le microprocesseur suit l'architecture MIPS classique.

Il utilise le système de périphériques pour la mémoire de données, les registres et la mémoire contenant le programme.

Chaque instruction s'effectue sur 4 cycles.

opérations supportées

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. une opération de concaténation 16 bits - 16 bits (utile pour le *lui*).

Code des opérations

Chaque opération est codée sur 4 bits :

0000 : Et
1000 : Ou
0100 : Xor
1100 : Nand
0010 : Nor
1010 : Addition
0110 : Soustraction
1110 : Multiplication

0001 : Egalité
1001 : Différence
0101 : Inférioté stricte
1101 : Supériorité
0011 : Infériorité
1011 : Supériorité
0111 : Lui
1111 : Srl

Fonctionnement

Toutes les opérations possibles sont exécutées parallèlement.

Le résultat est filtré selon le code de l'opération (sur 4 bits)

Toutes les opérations ont été codées récursivement.

Les registres

On utilise classiquement 32 registres sur 32 bits.

On peut faire plusieurs lectures simultanées grâce à des **périphériques auxiliaires de lecture**.

Instruction Memory

Chaque instruction est stockée sur 32 bits selon les formats MIPS :

- R-format : $opcode(6), rs(5), rt(5), rd(5), shift(5), funct(6)$
- I-format : $opcode(6), rs(5), rt(5), imm(16)$
- J-format : $opcode(6), add(26)$

Nous avons cependant certaines spécificités d'encodage.

Instruction Memory

Si une instruction utilise l'ALU, le code de l'opération associée se trouve soit dans l'opcode, soit dans le funct code.

Par exemple, les 4 premiers bits de l'opcode de **bgt** sont 1101 ce qui correspond à l'opération $>$.

Exemple d'instructions

Par exemple, l'instruction **add $a0, $a0, $a1** (R-format) donne :
000000 00100 10100 00100 00000 101000

Un code calculant la suite de Fibonnaci :

```
10101100000000001000000000000000
10101110000100001000000000000000
00000010000000000100000000101000
10101110000000000000000000000000
10101101000100000000000000000000
11110101000000000000000000000000
```

Data Memory

La Data Memory peut prendre deux formes :

- mode “classique”
- mode “horloge” pour avoir des afficheurs 7-segments.

Contrôleur

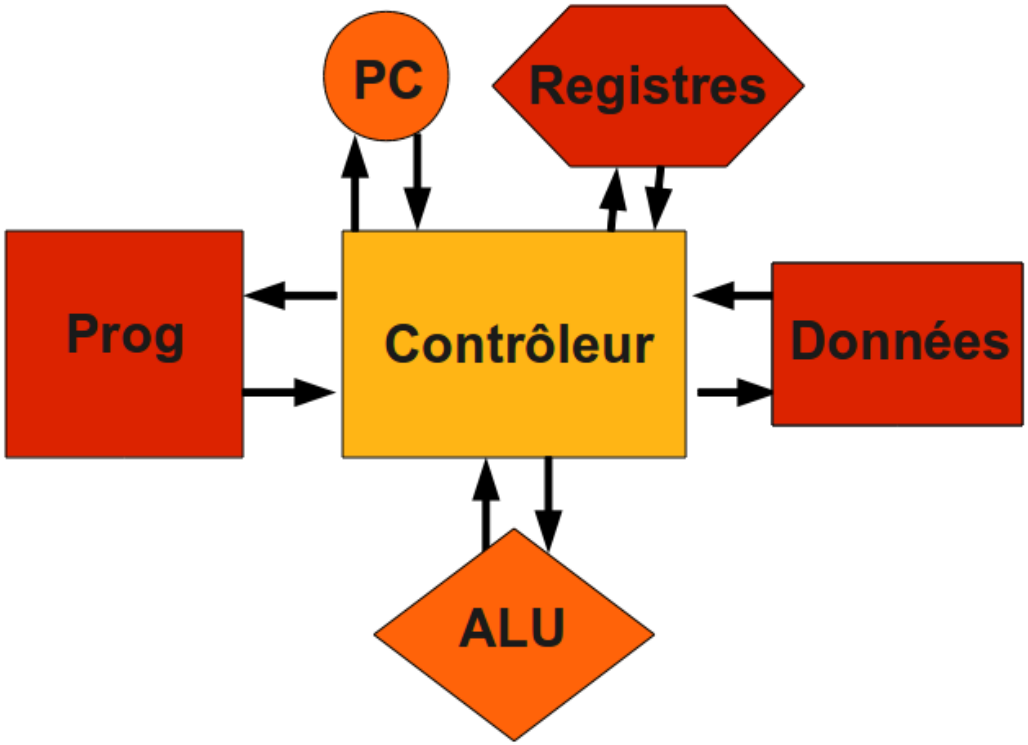

Figure: Contrôleur

Decodeurs

Le Contrôleur est divisée en 7 sous-unités.
Chacune traite un type d'instruction :

1. les R-formats classiques (add, mul, ..)
2. les I-formats classiques (andi, subi, ...)
3. Jump
4. Jump Register
5. les opérations de type *Branch*
6. l'écriture dans la mémoire de données
7. la lecture dans la mémoire de données

Traitement d'une instruction

Le traitement d'une instruction se fait sur 4 cycles.

Les sous-unités sont exécutées en parallèle mais le contrôleur **filtre les actions selon l'opcode** de l'instruction :

Seules les actions du décodeur concerné transparaissent.

Traitement d'une instruction

Chaque décodeur correspond à un type d'échange entre les différents composants. Cycle par cycle on a, pour une opération arithmétique en $R - format$:

1. Chargement de l'instruction
2. Cycle vide
3. Demande de *rs* et *rt* au gestionnaire de registres
4. Ecriture du résultat de l'ALU à l'adresse *rt* et incrémentation PC

Filtre temporel

Ces différents échanges sont effectués et sécurisés grâce à des filtres temporels :

```
Cycleur <a,b,c,d> ()
    RegIni <d> R1(R4.o)
    RegIni <c> R2(R1.o)
    RegIni <b> R3(R2.o)
    RegIni <a> R4(R3.o)
   -> o : R4.o;
```

```
Cycleur<0,0,1,0> CRS ()
Cycleur<0,0,1,0> CRT ()
Cycleur<0,0,0,1> CRD ()
Cycleur<0,0,0,1> CycleLectEcr()
FiltreSel <5> FRS(rs[0..4], CRS.o)
FiltreSel <5> FRT(rt[0..4], CRT.o)
FiltreSel <5> FRD(rd[0..4], CRD.o)
Ou <5> GR_add(FRT.o, FRD.o)

Cycleur<0,0,0,1> FALU()
FiltreSel <32> ValOp1(gr_reader[0..31], FALU.o)
FiltreSel <32> ValOp2(gestReg[0..31], FALU.o)

Cycleur<0,0,0,1> CPC()
Entier <4,b> Quatre()
FiltreSel <b> Ajout(Quatre.o, CPC.o)
Adder <b> NouvPC(pc[0..b-1], Ajout.o)

Srl<32,5> Res (alu[0..31], shift[0..4])
```

Afin de fournir un code éxécutable au microprocesseur, un assembleur a été codé. Il fournit les éléments suivant :

- l'implémentation des mnémoniques correspondant aux instructions du processeur.
- des étiquettes, notamment celles correspondant au périphériques mappés en mémoire (`clock_display` et `timestamp`).
- des pseudo-instructions comme `move` ou `beqd` augmentant la maniabilité du processeur.

Sommaire

1 Le langage Rock

2 Le compilateur Obsidian

3 Le microprocesseur

4 Horloge et calendrier
- Principe
  - Objectifs et contraintes
  - Plan d'attaque
- Mise en oeuvre
  - Multiplication, division
  - Calcul de la date et de l'heure
  - Communication avec le reste du système
- Fonctionnalité surprise
  - Aspect historique

Objectifs et contraintes

Objectif : Écrire en assembleur MIPS un programme prenant un timestamp en entrée et renvoyant la date et l'heure.

Quelques contraintes supplémentaires :

- Pas d'instructions `mul` ni `div`
- Pas de décalage à droite (= multiplication par 2^n)
- Renvoi de la date et heure sous le format "afficheur 7 segments"

Plan d'attaque

Il faut donc :

- Recoder la multiplication à partir de l'addition
- Recoder la division à partir de la multiplication
- Écrire l'algorithme de transformation d'un timestamp en format date/heure classique
- Trouver un moyen efficace de convertir un entier décimal en format "afficheur 7 segments"

Principe de la multiplication

Supposons que l'on veuille multiplier $a0 et $a1. L'algorithme est le suivant :

1. Si $a1 = 0, on renvoie 0
2. Si $a1 = 1, on renvoie $a0
3. Sinon, soit $x = 0$ si $a1 est pair, et $x =$ $a1 sinon
4. On fait $a0 = $a0/2
5. On appelle la fonction multiplier (récursivement, donc) pour calculer ($a0 $\times$ $a1)/2, que l'on multiplie par 2 et que l'on stocke dans $v0
6. On fait $v0 = $v0 $+ x$
7. Et on renvoie $v0 !

Code MIPS obtenu

```
multiplier:
        bne     $a1, $zero, multiplier_debut
        li      $a0, 0 # Si $a1 = 0, on met $a0 = 0
  multiplier_debut:
        move    $v0, $a0
        li      $t0, 1
        beq     $a1, $t0, multiplier_fin # Si $a1 = 1, on renvoie $a0
        addi    $sp, $sp, 4
        sw      $ra, 0($sp)
        addi    $sp, $sp, 4
        sw      $zero, 0($sp)
        andi    $t0, $a1, 1
        beq     $t0, $zero, multiplier_fintest
        sw      $a0, 0($sp) # On stocke x
  multiplier_fintest:
        srl     $a1, $a1, 1 # Division par 2
        jal     multiplier # On fait l'appel récursif
        add     $v0, $v0, $v0 # Multiplication par 2
        lw      $t0, 0($sp) # On ajoute x
        addi    $sp, $sp, -4
        add     $v0, $v0, $t0
        lw      $ra, 0($sp)
        addi    $sp, $sp, -4
  multiplier_fin:
        jr      $ra
```

Principe de la division

Comme à l'école primaire ! Divisons $a0 par $a1 :

1. On initialise le quotient $v0 à 0
2. Si a1 est plus grand que $a0, on s'arrête là
3. Sinon, on décale $a1 de n bits le faire arriver juste en-dessous de $a0
4. On fait $\$a0 = \$a0 - (\$a1 \times 2^n)$, et on rajoute 2^n au quotient
5. On revient à l'étape 2, et on recommence !

Code MIPS obtenu

```
diviser:
        li      $v0, 0 # Initialisation
    diviser_debut:
        bgt     $a1, $a0, diviser_fin # Test de fin
        move    $t1, $a1
        li      $t2, 1
    diviser_petiteboucle:
        bgt     $t1, $a0, diviser_finpetiteboucle
        add     $t1, $t1, $t1 # On décale $t1 (=$a1 au départ)
                              # jusqu'à dépasser $a0
        add     $t2, $t2, $t2 # On en profite pour calculer 2^n
        j       diviser_petiteboucle
    diviser_finpetiteboucle:
        srl     $t1, $t1, 1 # On est allés un cran trop loin
        srl     $t2, $t2, 1
        sub     $a0, $a0, $t1 # On enlève ce qu'il faut à $a0
        add     $v0, $v0, $t2 # On rajoute ce qu'il faut à $v0
        j       diviser_debut # Et on retourne à l'étape 2
    diviser_fin:
        jr      $ra
```

Calcul de l'heure

Le calcul de l'heure est la partie "simple" de l'algorithme de transformation du timestamp. En effet, le timestamp UNIX correspond au nombre de secondes écoulées depuis le 1^{er} janvier 1970. Comme il y a 86400 secondes dans une journée, il suffit donc de :

- Diviser le timestamp par 86400 secondes et prendre le reste : on obtient le nombre de secondes depuis le début de la journée.
- Diviser cette quantité par 3600 : on obtient le nombre d'heures écoulées depuis le début de la journée
- Prendre le reste de la division précédente, et le diviser par 60 : on obtient le nombre de minutes depuis le début de l'heure courante
- Et le reste de la précédente division donne directement les secondes écoulées depuis le début de la minute courante.

Calcul de la date - Années bissextiles

La première division par 86400 donne le nombre de jours écoulés depuis le 01/01/1970. Pour calculer la date correspondante, on ne peut pas directement opérer par divisions comme précédemment : il faut d'abord prendre en compte le problème des années bissextiles.

Ma méthode est la suivante : on considère qu'une année "normale" dure 366 jours et on se contente de modifier en dur le nombre de jours écoulés, en ajoutant un jour artificiel à la fin de chaque 28 février d'une année non bissextile.

Ainsi, lorsqu'on est par exemple le 1er janvier 1970, on considère non pas que $31 + 28 = 59$ jours se sont écoulés mais 60, pour compenser le fait que dans toute la suite, on considère que février a 29 jours.

Calcul de la date

Pour calculer le nombre d'années : on fait une division par 366 et on rajoute 1970.

Pour calculer mois et jours : on fait du cas par cas pour chaque mois, faute de formule mathématique simple.

```
addi    $t0, $t0, 29 # Février
bgt     $t0, $s4, moistrouve
addi    $s5, $s5, 1
move    $t1, $t0
addi    $t0, $t0, 31 # Mars
bgt     $t0, $s4, moistrouve
addi    $s5, $s5, 1
move    $t1, $t0
addi    $t0, $t0, 30 # Avril
bgt     $t0, $s4, moistrouve
addi    $s5, $s5, 1
move    $t1, $t0
```

Entrée / Sorties

Le programme prend un seul paramètre en entrée : le timestamp UNIX. Ce renseignement est disponible dans une adresse spéciale :

```
li      $t0, timestamp
lw      $s7, 0($t0)
```

En sortie, il écrit dans une adresse spéciale du système le résultat de la transformation d'un entier en suite d'octets en format "afficheur 7 segments" :

```
add     $t0, $s2, $s2
li      $t5, two_digits_to_segments
add     $t0, $t0, $t5
lb      $t1, 0($t0)
la      $t2, clock_display
sb      $t1, 0($t2)
add     $t0, $s2, $s2
addi    $t0, $t0, 1
li      $t5, two_digits_to_segments
add     $t0, $t0, $t5
lb      $t1, 0($t0)
sb      $t1, 1($t2)
```

Conversion

Où `two_digits_to_segments` est la table suivante :

```
two_digits_to_segments:
    .byte 0x3F, 0x3F, 0x3F, 0x06, 0x3F, 0x5B, 0x3F, 0x4F, 0x3F, 0x66, 0x3F,
    0x6D, 0x3F, 0x7D, 0x3F, 0x07, 0x3F, 0x7F, 0x3F, 0x6F, 0x06, 0x3F, 0x06,
    0x06, 0x06, 0x5B, 0x06, 0x4F, 0x06, 0x66, 0x06, 0x6D, 0x06, 0x7D, 0x06,
    0x07, 0x06, 0x7F, 0x06, 0x6F, 0x5B, 0x3F, 0x5B, 0x06, 0x5B, 0x5B, 0x5B,
    0x4F, 0x5B, 0x66, 0x5B, 0x6D, 0x5B, 0x7D, 0x5B, 0x07, 0x5B, 0x7F, 0x5B,
    0x6F, 0x4F, 0x3F, 0x4F, 0x06, 0x4F, 0x5B, 0x4F, 0x4F, 0x4F, 0x66, 0x4F,
    0x6D, 0x4F, 0x7D, 0x4F, 0x07, 0x4F, 0x7F, 0x4F, 0x6F, 0x66, 0x3F, 0x66,
    0x06, 0x66, 0x5B, 0x66, 0x4F, 0x66, 0x66, 0x66, 0x6D, 0x66, 0x7D, 0x66,
    0x07, 0x66, 0x7F, 0x66, 0x6F, 0x6D, 0x3F, 0x6D, 0x06, 0x6D, 0x5B, 0x6D,
    0x4F, 0x6D, 0x66, 0x6D, 0x6D, 0x6D, 0x7D, 0x6D, 0x07, 0x6D, 0x7F, 0x6D,
    0x6F, 0x7D, 0x3F, 0x7D, 0x06, 0x7D, 0x5B, 0x7D, 0x4F, 0x7D, 0x66, 0x7D,
    0x6D, 0x7D, 0x7D, 0x7D, 0x07, 0x7D, 0x7F, 0x7D, 0x6F, 0x07, 0x3F, 0x07,
    0x06, 0x07, 0x5B, 0x07, 0x4F, 0x07, 0x66, 0x07, 0x6D, 0x07, 0x7D, 0x07,
    0x07, 0x07, 0x7F, 0x07, 0x6F, 0x7F, 0x3F, 0x7F, 0x06, 0x7F, 0x5B, 0x7F,
    0x4F, 0x7F, 0x66, 0x7F, 0x6D, 0x7F, 0x7D, 0x7F, 0x07, 0x7F, 0x7F, 0x7F,
    0x6F, 0x6F, 0x3F, 0x6F, 0x06, 0x6F, 0x5B, 0x6F, 0x4F, 0x6F, 0x66, 0x6F,
    0x6D, 0x6F, 0x7D, 0x6F, 0x07, 0x6F, 0x7F, 0x6F, 0x6F
```

! ? !

L'École normale dite « de l'an III », est créée à Paris par la Convention qui décrète le 9 brumaire an III que :

(article 1er) « Il sera établi à Paris une École normale, où seront appelés, de toutes les parties de la République, des citoyens déjà instruits dans les sciences utiles, pour apprendre, sous les professeurs les plus habiles dans tous les genres, l'art d'enseigner. ».

(Wikipédia)

Calendrier républicain

« 9 brumaire an III » ?

De 1792 à 1806 (période incluant donc la création de l'École Normale), en France, le calendrier grégorien n'a plus été utilisé, au profit du calendrier républicain, inventé par les révolutionnaires pour oublier l'empreinte religieuse présente dans le calendrier jusqu'à lors, depuis le nom des mois (Juin - Junon) jusqu'au nom des jours de la semaine (Mardi - jour de Vénus), sans oublier bien sûr les saints de chaque jour de l'année.

Détails

Ainsi, les mois, dans l'ordre (en commançant par l'équivalent de septembre) sont : Vendémiaire, Brumaire, Frimaire, Nivôse, Pluviôse, Ventôse, Germinal, Floréal, Prairial, Messidor, Thermidor, Fructidor.

Les jours de la "décade" : Primidi, Duodi, Tridi, Quartidi, Quinditi, Sextidi, Septidi, Octidi, Nonidi, Décadi.

Fonctionnement

Bonne nouvelle : Les années bissextiles sont synchronisées avec celles du calendrier grégorien !

Bonne nouvelle 2 : Tous les mois font 30 jours ! (on rajoute 5 ou 6 jours en fin d'année pour éviter le décalage avec l'année solaire)

Bonne nouvelle 3 : Le système d'heure est plus simple ! Il y a dix heures dans une journée, chacune coupée en cent "minutes", chacune coupée en cent "secondes", et ainsi de suite, jusqu'à "la plus petite portion commensurable de la durée"

Quoique... Pour un timestamp en secondes, ça va peut-être pas être si pratique que ça...

Démo !

Sous vos yeux ébahis...