Réalisation d'une machine algorithmique
Polytech' Grenoble - UFR-IM2AG : INFO3
Travaux pratiques ALM Hard

1 Introduction

Nous nous intéressons à une technique de cryptographie appelé *RSA* basée sur une paire de clés publique et secrète. Inventée en 1977 par trois américains (Ronald Rivest, Adi Shamir et Leonard Adleman).

Elle est largement utilisée aujourd'hui :
- chiffrement de document circulant sur Internet,
- authentification pour des logiciels (comme `ssh`),
- authentification de carte bleue,
- ...

L'algorithme que nous allons étudier consiste à calculer le chiffrement et le déchiffrement d'un entier. Cet entier peut par exemple représenter un caractère dans un texte. On peut imaginer l'intérêt qu'il soit implémenté dans un circuit particulier à côté du processeur (co-processeur) dans un ordinateur vu la complexité des calculs effectués.

Nous allons dans ce TP nous intéresser à la réalisation d'un circuit permettant l'exécution de cet algorithme. Nous nous arrêterons à l'étape de simulation d'une description en Langage Lustre du circuit. Nous ne réaliserons pas "pour de vrai" le circuit, mais il serait aisé de le faire à partir de cette description à l'aide d'outils de CAO appropriés.

Bien sûr, cette description doit être réalisée à l'aide des opérateurs logiques et de variables booléennes (portes logiques de base) ; les $if then else$ et variables entières sont interdits (sauf pour la visualisation des résultats).

2 Utilisation

Deux personnes qui veulent s'échanger des données chiffrées peuvent le faire à l'aide d'un algorithme de chiffrement et d'une même clé secrète (voir la figure 1). On parle alors de chiffrement à clé symétrique. L'échange de cette clé symétrique pose un problème puisque lui aussi doit être fait de façon sûre.

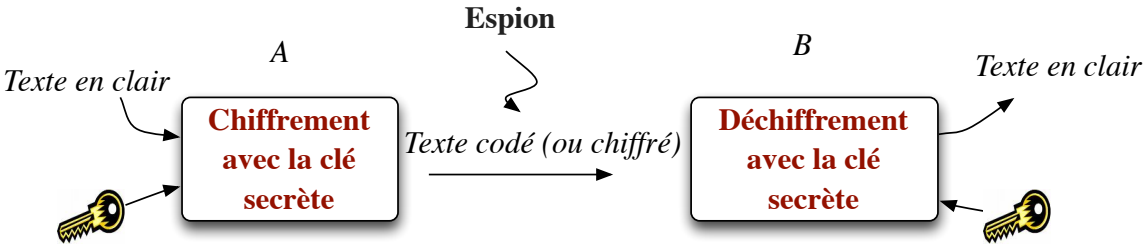

FIGURE 1 – Principe de chiffrement à clé symétrique

Une autre méthode a été inventé pour éviter ce problème, il est basé sur une paire de clés secrètes appelées clé publique et clé privée. L'intérêt de cette technique est qu'il n'ait pas la peine de partager la même clé secrète pour chiffrer/déchiffrer des données que l'on veut s'échanger.

- Imaginons que A veut envoyer un message m à B,
- A demande à B de lui envoyer une clé publique KPu (que B a calculée). Cette clé n'est pas secrète et peut être envoyé en clair.
- A chiffre m avec cette clé publique : $Kpu(m)$,
- B est le seul à pouvoir déchiffrer le message à l'aide de sa clé secrète KPr : $Kpr(Kpu(m))=m$.

La figure 2 résume le principe de son utilisation pour le chiffrement de message.

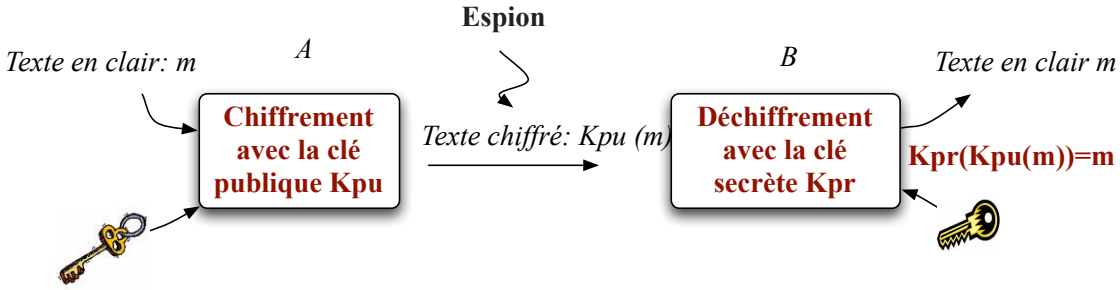

FIGURE 2 – Principe de chiffrement RSA

Si vous êtes intéressé par le sujet vous pouvez trouver sur wikipedia de plus amples informations sur la cryptographie RSA (`http://fr.wikipedia.org/wiki/Rivest_Shamir_Adleman`).

3 La théorie

Soient trois entiers naturels e, d et n respectant les propriétés suivantes :

- **Soient** p **et** q **deux nombres premiers,**
- **Soit l'entier** n **tel que** $n = p * q$ **;**
- **L'entier** e **est choisi tel que** $PGCD(e, (p-1)*(q-1)) = 1$ **;**
  avec $p, q < e < (p-1)*(q-1)$
- **L'entier** d **est choisi tel que** $e * d \ modulo \ ((p-1)*(q-1)) = 1$ **;**
  avec $p, q < d < (p-1)*(q-1)$

Alors on peut montrer que **pour tout entier naturel** $m < n$ on a :

si $cc = m^e \ modulo(n)$ **alors** $m = cc^d \ modulo(n)$

Exemple Soient $p = 7, q = 11, n = 77, e = 13, d = 37$ vérifiant ces propriétés.
Prenons $m = 18$, nous calculons $cc = 18^{13} \ modulo(77) = 46$
Nous avons $46^{37} \ modulo(77) = 18$

Autrement dit on peut chiffrer tout entier m en calculant $cc = m^e \ modulo(n)$ on peut déchiffrer en calculant $cc^d \ modulo(n)$.

(e, n) est la *clé publique* et (d, n) la *clé secrète*.

Ce principe de chiffrement est réputé sûr si les nombres premiers p et q sont "grands", en effet pour *casser* le chiffrement et trouver la clé secrète (d, n), il faut trouver p et q en connaissant $p * q$. Aujourd'hui il est recommandé d'utiliser des clés de taille supérieure à 1024 bits ($\doteq$ 300 chiffres décimaux !).

4 L'algorithme

On va s'intéresser au calcul permettant ce chiffrement, soit l'***exponentiation modulaire*** :

$$code = m^{exp} \ modulo(n)$$

Voici une première version de l'algorithme qui consiste à calculer m^{exp} par des multiplications successives par m puis à calculer le *modulo* :

```
Les données :
    m, exp, n : des entiers naturels ;
Le résultat :
    code : un entier naturel ;
Algorithme :
    Lire(m) ; Lire(exp) ; Lire(n) ;
    code ← 1 ;
    tant que exp > 0
        debut
            code ← code * m ;
            exp ← exp - 1 ;
        fin
    code = code modulo(n) ;
    Ecrire(code) ;
```

On constate que le nombre de bits nécessaire pour stocker les calculs intermédiaires de l'exponentiation peut devenir très grand bien que le résultat puisse être représenté sur le même nombre de bits que n.

On peut remarquer que $(a*b)\ modulo(t) = (a\ modulo(t))*(b\ modulo(t))$ et du coup réduire la taille des résultats intermédiaires.

Voici une deuxième version de l'algorithme :

```
    code ← 1 ;
    tant que exp > 0
        debut
            code ← (code * m)modulo(n) ;
            exp ← exp - 1 ;
        fin
```

On peut encore améliorer cet algorithme en optimisant la calcul de l'exponentiation.
Ecrivons exp à l'aide de puissance de 2 (écriture binaire) : $exp = e_0*2^0+e_1*2^1+e_2*2^2....+e_k*2^k$ avec $e_i = 0$ ou 1.
On peut donc écrire : $m^{exp} = m^{e_0*2^0} * m^{e_1*2^1} * m^{e_2*2^2} * .... * m^{e_k*2^k}$
On diminue fortement le nombre de multiplications en supprimant celles pour lesquelles $e_i = 0$.
A chaque étape du calcul, il faut multiplier le résultat par m si $e_i = 1$ puis élever m au carré.

Voici une troisième version de l'algorithme utilisant cette décomposition (en gardant à l'esprit que $(a*b)\ modulo(t) = (a\ modulo(t))*(b\ modulo(t))$) :

code $\leftarrow$ **1 ;**
tant que exp $>$ **0**
debut
si (exp & 1)= 1 alors
code $\leftarrow$ **(code * m)***modulo*(**n**) **;**
m $\leftarrow$ **(m * m)***modulo*(**n**) **;**
exp $\leftarrow$ **exp** *decal*(1) **;**
fin

où & est l'opération *ET* booléen bit à bit et *decal*(1) est le décalage d'un bit à droite.

- Se persuader que l'algorithme est juste.
- Calculez le nombre de multiplication que l'on effectue dans cet algorithme et le précédent.

Voici une traduction en langage C :

```
main(int argc, char * argv[])
{
    unsigned long int m,exp,n;
    unsigned long int code;

    if (argc != 4) exit(0);
    sscanf(argv[1], "%ld", &m);
    sscanf(argv[2], "%ld", &exp);
    sscanf(argv[3], "%ld", &n);

code=1;
while (exp>0)
{
if ((exp&1)>0)
{
code= (code * m)%n;
}
exp=exp>>1;
m= (m*m)%n;
 }
printf("Resultat chiffrement %ld\n",  code);
}
}
```

Voici des exemples de paire de clés public / privée :

- avec p= 3 et q= 11 : (13, 33), (17, 33)
- avec p= 7 et q= 11 : (13, 77), (37, 77)
- avec p = 29, q = 37 : (71,1073) , (1079,1073)
- avec p= 373 et q= 757 : (761, 282361) , (138953, 282361)

- p = 1108610869535798843 et q = 1054626217282232789 :
  (65537, 00 :e1 :2c :89 :2e :49 :a5 :d0 :81 :ca :7b :28 :b8 :5a :3a :17),
  (77 :07 :e7 :ec :84 :1a :17 :32 :e6 :7c :db :a8 :0b :62 :a,
  00 :e1 :2c :89 :2e :49 :a5 :d0 :81 :ca :7b :28 :b8 :5a :3a :17) en hexadécimal

Remarque : Vous pouvez générer des paires de clé à l'aide de la commande (sous Unix) :

ssh-keygen -t rsa -b <longueur> -f <fichier> ou *openssl genrsa -out <fichier> <longueur>*.

Et pour visualiser les clés : *openssl rsa -in <fichier> -text*

Faites tourner le programme C avec quelques valeurs d'entiers à chiffrer (voir le fichier `chiffrement.c` sur le Moodle). Vous pouvez aussi essayer de faire les calculs si vous avez une calculette sous la main. Vérifier que le chiffrement/déchiffrement est juste pour ces paires de clés.
Attention il faut que le nombre à coder soit inférieur à n (avec clés = (x,n),(y,n)).

5 Le circuit

5.1 L'interface d'entrée/sortie

Nous allons concevoir une interface d'entrées/sorties simple, elle serait différente si le circuit devait être utilisé en tant que co-processeur (voir cours d'ALM du semestre 2). La figure 3 donne l'ensemble des entrées/sorties du circuit à réaliser.

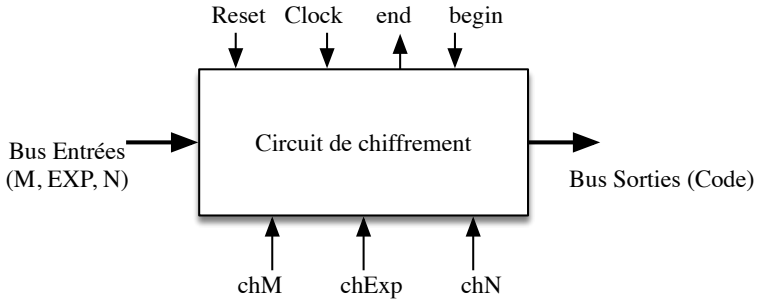

FIGURE 3 – Les entrées/sorties du circuit

1. Les entrées :
   L'utilisateur du circuit fournit via un bus d'entrée les valeurs des données `M` l'entier à chiffrer, `EXP, N` les deux éléments de la clé utilisée.
   Pour cela il doit activer l'entrée de chargement correspondante (`chM, chExp, chN`) au moment où la valeur est présente sur le Bus d'entrée. Ces chargements sont fait pendant que l'automate de la partie contrôle du circuit se trouve dans l'état initial (Voir plus loin).

Une fois que les données ont été chargées, l'entrée `begin` permet de "lancer" l'algorithme.
De plus le circuit possède une entrée `clock` permettant de cadencer ses calculs et d'être initialisé `reset`.

2. Les sorties :
Le circuit délivre sur un bus de sortie la valeur du registre contenant le résultat (voir le schéma de la partie opérative (figure 5) plus loin). L'utilisateur peut savoir si l'algorithme est fini et que le résultat voulu est sur ce bus à l'aide du signal `end`.

5.2 L'architecture du circuit

Nous allons réaliser ce circuit à l'aide d'une architecture à Partie contrôle/Partie opérative. La figure 4 donne l'architecture générale partie contrôle/partie opérative du circuit.

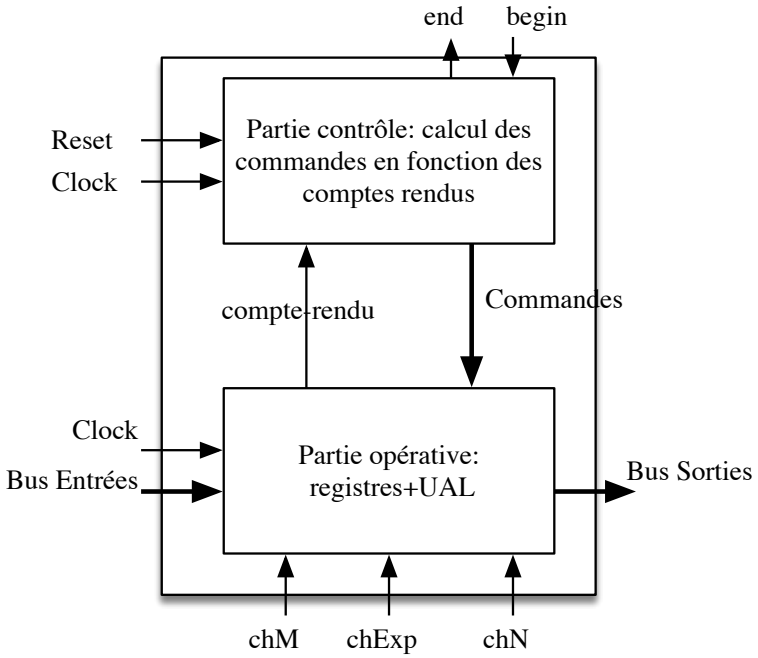

FIGURE 4 – Architecture PC/PO du circuit

5.3 Modification de l'algorithme

Nous voulons réaliser l'algorithme à partir d'une UAL effectuant seulement le décalage, la multiplication et la soustraction.

1. Trouvez comment exprimer le calcul du *modulo* à partir de soustractions.
2. Redonnez l'algorithme de telle manière que les seules opérations effectuées soient le décalage, la multiplication et la soustraction.

5.4 La partie opérative

1. La figure 5 donne une ébauche de la partie opérative du circuit. Elle est composée des registres `CODE, M, EXP` et `N` contenant les valeurs des variables de l'algorithme. Chacun de ces registres est sensible au front montant de `CK` (l'horloge), il possède une commande de chargement `chNom`.

   Le multiplexeur 2 vers 1 permet de charger au départ les données depuis le bus extérieur.
   L'UAL doit permettre d'effectuer le décalage, la multiplication et la soustraction. Il calcul les flags permettant de délivrer les conditions de l'algorithme (`compte-rendus` à destination de la partie contrôle).

2. Dessinez une partie opérative permettant d'effectuer tous les calculs et affectations apparaissant dans l'algorithme trouvé à la question précédente. Ne pas oublier les calculs des conditions.

   Les bus `opérande 1` et `opérande 2` seront réalisés à l'aide de multiplexeurs n bits. Fixer les codes des signaux de sélections des deux multiplexeurs utilisés pour les deux bus.

   Donnez un nom à tous les signaux de commande de cette partie opérative. Vérifiez que cette PO permet bien tous les calculs et affectations apparaissant dans l'algorithme.

3. Fixez un codage des commandes de calcul de l'UAL `opUAL` et décrivez en Lustre une UAL permettant de faire ces opérations en supposant que vous disposez d'un multiplieur et d'un soustracteur n bits.
   Le multiplieur n bits est donné (voir sur le Moodle le fichier `Multnbitsbase2.lus`).

4. Décrivez en lustre le node `registre à n bits` à partir de la bascule donnée (voir sur le Moodle). Dans les différentes bascules utilisées pour fabriquer les registres, l'entrée `CK` est connectée au signal d'horloge `Clock` du circuit.
   Le chargement ou le non chargement du registre est commandé par l'entrée `CHAR/Enable` des bascules. Par exemple, le chargement effectif du registre `M` a lieu si l'entrée `chM` (connectée à l'entrée `CHAR` des bascules) est à 1.

5. Décrivez en Lustre cette partie opérative.

5.5 La partie contrôle

1. Donnez le graphe de l'automate de contrôle de la PO précédemment trouvée en faisant apparaître pour chaque état le calcul et l'affectation effectués.

2. Pour chaque état, donnez la valeur des sorties de l'automate (commandes de la partie opérative) déduite des actions apparaissant sur le graphe de l'automate.

3. Après avoir fixé un codage 1 parmi N des états de l'automate, décrivez en Lustre une réalisation de cet automate. On ne donnera pas le dessin du circuit (avec des bascules et portes logiques) qui est fastidieux et n'apporte rien de plus.

4. Comment se fait l'initialisation de ce circuit ?

5.6 Conseil de réalisation

1. Décider des codes utilisés dans l'UAL et sur les multiplexeurs avant de se lancer dans le Lustre. Faire apparaître ces choix sur le dessin de la PO donné dans la figure 5
2. Pour débuguer ne pas hésiter à ajouter en sortie du circuit d'autres variables. En particulier il est intéressant de connaître l'état courant de la partie contrôle.
3. Attention au dépassement de capacité de votre circuit ; choisissez des valeurs initiales pas trop grandes en fonction du nombre de bits `n` fixé pour votre circuit.
4. Vérifiez que votre circuit fonctionne en comparant sa sortie à celle du programme C donné. Essayez avec la paire de clé (13,33) (17,33) sur des nombres inférieurs à 33. Pour faciliter la lecture du résultat, utilisez le node `entiernat` (donné dans `multnbitsbase2.lus`) qui affiche en décimal l'entier égal à la valeur en base 2 passé en paramètre.

5.7 Pour ceux qui veulent aller plus loin

Rajoutez à votre description la réalisation d'une sortie signalant un dépassement des capacités de votre circuit, c'est à dire signalant que le résultat de l'algorithme est faux sur le nombre de bit prévu.

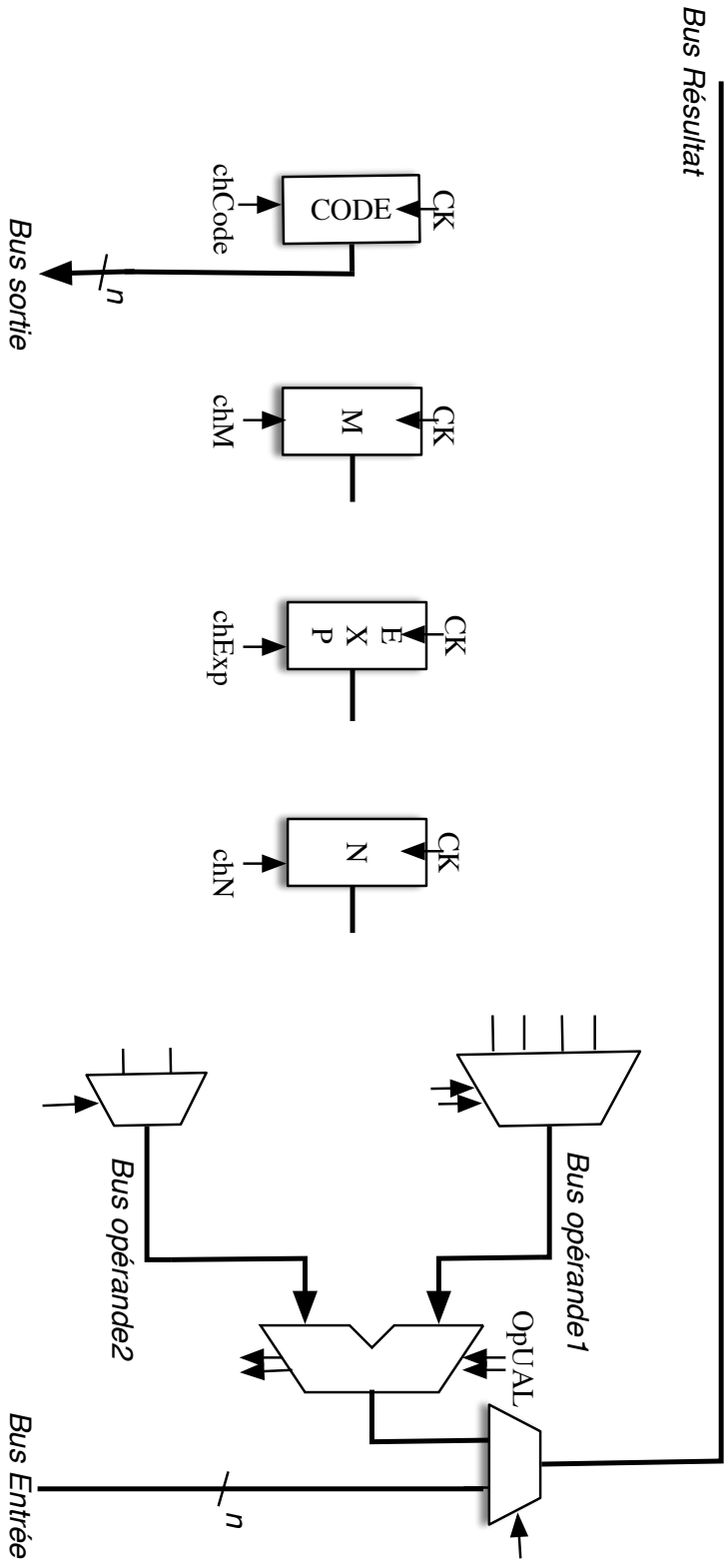

FIGURE 5 – La partie opérative du circuit