

Challenge SSTIC 2013 : solution

Julien Perrot
@nieluj

6 juin 2013

Résumé

Ce document présente une démarche possible pour résoudre le challenge SSTIC 2013. Comme pour les années précédentes, la validation du challenge nécessite de pouvoir extraire, depuis un fichier téléchargé sur le site de la conférence, une adresse email de la forme @sstic.org.

Quatre étapes doivent être résolues de façon séquentielle pour terminer le challenge. L'étude d'une capture réseau constitue la première étape. Cette capture contient le transfert par FTP d'une archive chiffrée en AES. Le déchiffrement de cette archive est possible en identifiant une clé secrète dissimulée dans des canaux cachés.

L'archive alors obtenue contient plusieurs fichiers, dont un script Python permettant de piloter un périphérique. Les opérations possibles sont l'envoi de données, le démarrage d'un traitement et la réception de données. Ce périphérique est en réalité un circuit programmable dont le schéma logique est également présent dans l'archive. L'identification du jeu d'instructions supporté par le circuit, rendue possible par l'analyse du schéma logique, permet de désassembler le programme chargé dans le circuit. Ce dernier est une routine de déchiffrement appliquée sur les données envoyées au circuit. La détermination de la clé secrète est réalisée à l'aide d'une attaque par force brute, en exploitant les faiblesses de conception présentes dans l'algorithme de chiffrement. Les données envoyées au circuit, une fois déchiffrées, correspondent à un fichier PostScript.

La troisième étape réside dans l'analyse de ce fichier. Une routine de déchiffrement est identifiée à l'issue d'une démarche de rétro ingénierie. Cette routine contient des conditions d'arrêt qu'il est possible d'utiliser pour dérouler une attaque par force brute sur la valeur de la clé secrète. Une fois la clé obtenue, le déchiffrement des données contenues dans le fichier PostScript permet d'obtenir un fichier contenant des contacts au format vCard.

Enfin, la dernière étape du challenge correspond au décodage d'une adresse email présente au sein du fichier vCard précédemment déchiffré.

Table des matières

1	Étude de la capture réseau	7
1.1	Récupération du fichier et premiers pas	7
1.2	Identification des canaux cachés	10
1.3	Détermination de la clé et déchiffrement	14
2	Analyse du FPGA	17
2.1	Découverte	17
2.2	Étude du fichier netlist	19
2.2.1	Description des portes logiques utilisées	21
2.2.2	Bloc x_smp	22
2.2.3	Bloc x_smd	24
2.2.4	Bloc x_r	26
2.2.5	Bloc finished<7>_imp	29
2.2.6	Bloc x_ip	30
2.2.7	Bloc x_accu	32
2.2.8	Bloc accu_w_imp	38
2.2.9	Bloc x_u	39
2.3	Synthèse de l'analyse des blocs du composant	49
2.3.1	Gestion de la mémoire	49
2.3.2	Gestion de l'exécution	49
2.3.3	Jeu d'instructions	50
2.4	Rétro-ingénierie du programme	50
2.4.1	Développement d'un désassembleur	50
2.4.2	Initialisation du programme	55
2.4.3	Bloc loc_71h	55
2.4.4	Bloc loc_0e3h	60

2.4.5	Bloc <code>loc_0eh</code>	60
2.4.6	Bloc <code>loc_16h</code>	60
2.4.7	Bloc <code>loc_24h</code>	61
2.4.8	Bloc <code>loc_40h</code>	63
2.4.9	Résultat de la rétro-ingénierie	63
2.5	Détermination de la clé	65
3	Rétro-ingénierie du fichier <code>script.ps</code>	68
3.1	Identification du fichier <code>atad</code>	68
3.2	Développement d'une boîte à outils Postscript	69
3.3	Gestion des arguments de <code>script.ps</code>	69
3.4	Étude du traitement principal de la fonction <code>main</code>	72
3.4.1	Utilisation d'un débogueur Postscript	72
3.4.2	Traitement des données de I1	74
3.4.3	Déchiffrement des données de I2	76
3.5	Analyse du programme I2	81
3.6	Analyse du programme I4	82
3.6.1	Déchiffrement des données de I4	82
3.6.2	Boucle principale	84
3.6.3	Boucle secondaire	85
3.7	Attaque par force brute	91
4	Décodage du contenu de la vCard	94
5	Conclusion	96
5.1	Synthèse	96
5.2	Remerciements	97
A	Annexes	98
A.1	Annexe : fichier <code>dump.bin</code>	98
A.2	Annexe : FPGA	99
A.2.1	Étude du schéma logique	99
A.2.2	Rétro-ingénierie	99
A.2.3	Attaque par force brute	100
A.3	Annexe : <code>script.ps</code>	100

A.3.1	Découverte	100
A.3.2	Développement d'une boîte à outils Postscript	100
A.3.3	Analyse de I2	100
A.3.4	Analyse de I4	100
A.3.5	Attaque par force brute	100
A.4	Annexe : fichier vCard	100
A.5	Annexe : conclusion	101

Table des figures

1.1	Hierarchie des protocoles dans <code>dump.bin</code>	7
1.2	Conversations TCP	8
1.3	Première conversation TCP	8
1.4	Seconde conversation TCP	9
1.5	Troisième conversation TCP	9
1.6	Représentation des ID ICMP	12
1.7	Représentation du champ <code>tos</code>	12
1.8	Représentation du champ <code>ttl</code>	13
1.9	Représentation du délai entre deux paquets	14
2.1	Environnement de développement Xilinx sur DVD	19
2.2	Ouverture du fichier <code>s.ngf</code> dans ISE	20
2.3	Bloc S développé dans ISE	20
2.4	Exemple de multiplexeur	22
2.5	Exemple de bascule D flip-flop	22
2.6	Vision macroscopique du bloc <code>x_smp</code>	22
2.7	Écriture d'un octet dans <code>x_smp</code>	23
2.8	Lecture d'un octet dans <code>x_smp</code>	24
2.9	Vision macroscopique du bloc <code>x_smd</code>	24
2.10	Entrées de <code>smd_w<7>1</code>	25
2.11	Vision macroscopique du bloc <code>x_r</code>	26
2.12	Écriture d'un octet dans <code>x_r</code>	27
2.13	Activation du signal <code>w</code>	27
2.14	Entrées de <code>r_w<7>1</code>	28
2.15	Lecture d'un octet dans <code>x_r</code>	29
2.16	Vision développée du bloc <code>finished<7>_imp</code>	29
2.17	Vision macroscopique du bloc <code>x_ip</code>	30

2.18	Vision développée du bloc x_ip	30
2.19	Entrées du bloc ip_w1	31
2.20	Entrées de la primitive PWR_3_o_smp_data_r[7]_equal_25_o<7>1	32
2.21	Vision développée du bloc x_accu	32
2.22	Utilisation en série de multiplexeurs	33
2.23	Entrées de Mmux_accu_data_w1	34
2.24	Entrées au niveau bit de Mmux_accu_data_w1	35
2.25	Entrées de Mmux_u_data_s[7]_r_data_r[7]_mux_14_OUT1	36
2.26	Entrées de la primitive PWR_3_o_smp_data_r[7]_equal_8_o<7>1	36
2.27	Entrées de Mmux_u_data_s[7]_smd_data_r[7]_mux_13_OUT1	37
2.28	Entrées du bloc PWR_3_o_smp_data_r[7]_equal_11_o<7>	37
2.29	Vision développée du bloc accu_w_imp	38
2.30	Vision macroscopique du bloc x_u	40
2.31	Vision développée du bloc x_u	40
2.32	Entrées du multiplexeur Mmux_nn00311	41
2.33	Entrées de _n0030<7>1	41
2.34	Entrées du multiplexeur Mmux_nn00361	42
2.35	Entrées de _n0035<7>1	43
2.36	Entrées du multiplexeur Mmux_nn00411	44
2.37	Entrées de _n0040<7>1	44
2.38	Entrées du multiplexeur Mmux_nn00451	45
2.39	Visualisation niveau bit de Mmux_nn00451	46
2.40	Entrées de _n0044<7>1	46
2.41	Entrées du multiplexeur Mmux_data_s1	47
2.42	Visualisation niveau bit de Mmux_data_s1	48
2.43	Entrées de _n0047<7>1	48
2.44	Affichage sous Metasm	54
2.45	Structure du programme	54
2.46	Structure du programme à l'adresse 0x71	56
A.1	Bloc S développé	99

Liste des tableaux

2.1	Entrées du bloc <code>x_smp</code>	23
2.2	Sortie du bloc <code>x_smp</code>	23
2.3	Entrées du bloc <code>x_smd</code>	24
2.4	Sorties du bloc <code>x_smd</code>	25
2.5	Entrées du bloc <code>x_r</code>	26
2.6	Sortie du bloc <code>x_r</code>	26
2.7	Entrée du bloc <code>finished<7>_imp</code>	29
2.8	Sortie du bloc <code>finished<7>_imp</code>	29
2.9	Entrées du bloc <code>x_ip</code>	30
2.10	Sortie du bloc <code>x_ip</code>	31
2.11	Entrées du bloc <code>x_accu</code>	33
2.12	Sortie du bloc <code>x_accu</code>	33
2.13	Entrées du bloc <code>accu_w_imp</code>	39
2.14	Sortie du bloc <code>accu_w_imp</code>	39
2.15	Conditions d'activation de <code>accu_w</code>	39
2.16	Entrées du bloc <code>x_u</code>	40
2.17	Sortie du bloc <code>x_u</code>	40
2.18	Jeu d'instructions supporté par le composant	50
2.19	Simplifications réalisées sur le code désassemblé	52
2.20	Table de vérité de la fonction <code>sub_71h</code>	60
3.1	Conditions d'arrêt	91

Chapitre 1

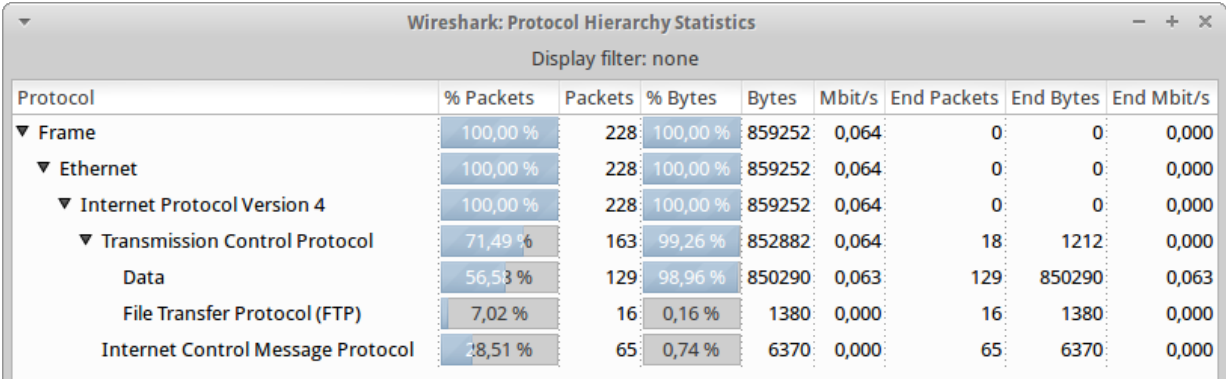
Étude de la capture réseau

1.1 Récupération du fichier et premiers pas

Le fichier `dump.bin` peut être téléchargé à la page <http://communaute.sstic.org/ChallengeSSTIC2013> :

```
$ wget --quiet http://static.sstic.org/challenge2013/dump.bin
$ ls -l dump.bin
-rw-rw-r-- 1 jpe jpe 862924 mars 25 13:42 dump.bin
$ md5sum dump.bin
968531851beed222851dbfd59140a395 dump.bin
$ file dump.bin
dump.bin: tcpdump capture file (little-endian) - version 2.4 (Ethernet, capture length 65535)
```

Le fichier obtenu est donc une capture réseau de 862 924 octets. Wireshark permet d'afficher des statistiques sur le contenu de la capture, comme présenté sur la figure 1.1.



Protocol	% Packets	Packets	% Bytes	Bytes	Mbit/s	End Packets	End Bytes	End Mbit/s
▼ Frame	100,00 %	228	100,00 %	859252	0,064	0	0	0,000
▼ Ethernet	100,00 %	228	100,00 %	859252	0,064	0	0	0,000
▼ Internet Protocol Version 4	100,00 %	228	100,00 %	859252	0,064	0	0	0,000
▼ Transmission Control Protocol	71,49 %	163	99,26 %	852882	0,064	18	1212	0,000
Data	56,58 %	129	98,96 %	850290	0,063	129	850290	0,063
File Transfer Protocol (FTP)	7,02 %	16	0,16 %	1380	0,000	16	1380	0,000
Internet Control Message Protocol	28,51 %	65	0,74 %	6370	0,000	65	6370	0,000

FIGURE 1.1 – Hiérarchie des protocoles dans `dump.bin`

D'après ces statistiques, la capture contient 228 paquets IP dont 163 datagrammes TCP et 65 messages ICMP. Les sessions TCP sont représentées à la figure 1.2.

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A→B	Bytes A→B	Packets A←B	Bytes A←B
192.168.1.13	57648	192.168.1.12	1234	5	884	5	884	0	0
192.168.1.13	36008	192.168.1.12	21	27	2 114	27	2 114	0	0
192.168.1.12	60733	192.168.1.13	44676	131	849 884	0	0	131	849 884

FIGURE 1.2 – Conversations TCP

On constate la présence de trois conversations TCP entre les adresses 192.168.1.13 et 192.168.1.12. Le contenu de la première conversation est affiché dans la figure 1.3.

Stream Content

```
Bonjour,
J'ai egare la cle pour dechiffrer mon carnet d'adresses.
Tu pourrais m'aider a la retrouver ? J'ai besoin de recuperer une adresse email
a l'interieur.
Pour t'aider, je t'envoie :
- une archive chiffee en AES par FTP
- la cle AES par canaux caches
voici l'iv utilise pour AES : 76C128D46A6C4B15B43016904BE176AC
voici le checksum de l'archive pour verifier le dechiffrement :
61c9392f617290642f9a12499de6b688
merci

PS :
Indication pour les canaux caches : 1 bit de canal cache temporel
concatene a 3 bits de canal cache non temporel.
```

Entire conversation (546 bytes)

Rechercher Enregistrer sous Imprimer ASCII EBCDIC Hex Dump C Arrays Raw

Aide Filter Out This Stream Fermer

FIGURE 1.3 – Première conversation TCP

Cette conversation contient des informations particulièrement intéressantes pour le reste du challenge. On apprend qu'une archive chiffrée est transférée par FTP et que la clé de déchiffrement est transmise par canaux cachés. Des indications sont données pour identifier ces canaux cachés :

- 1 bit est contenu dans un canal caché de type temporel ;
- 3 bits sont contenus dans un canal caché non temporel (donc dans les données).

Le contenu de la seconde conversation TCP est affiché dans la figure 1.4.

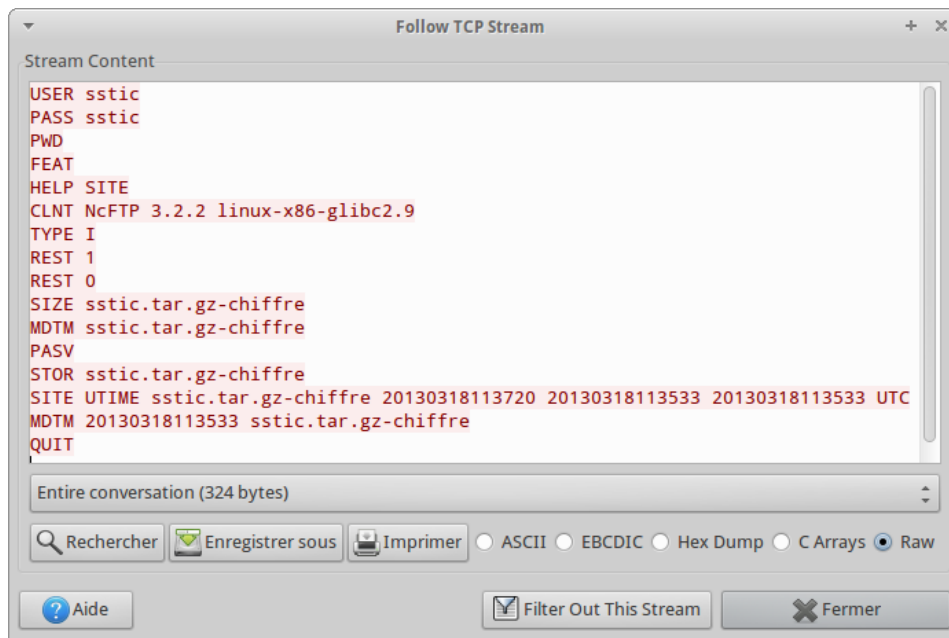


FIGURE 1.4 – Seconde conversation TCP

Cette conversation contient les commandes envoyées au serveur FTP. Enfin, le contenu de la dernière conversation est présenté dans la figure 1.5.

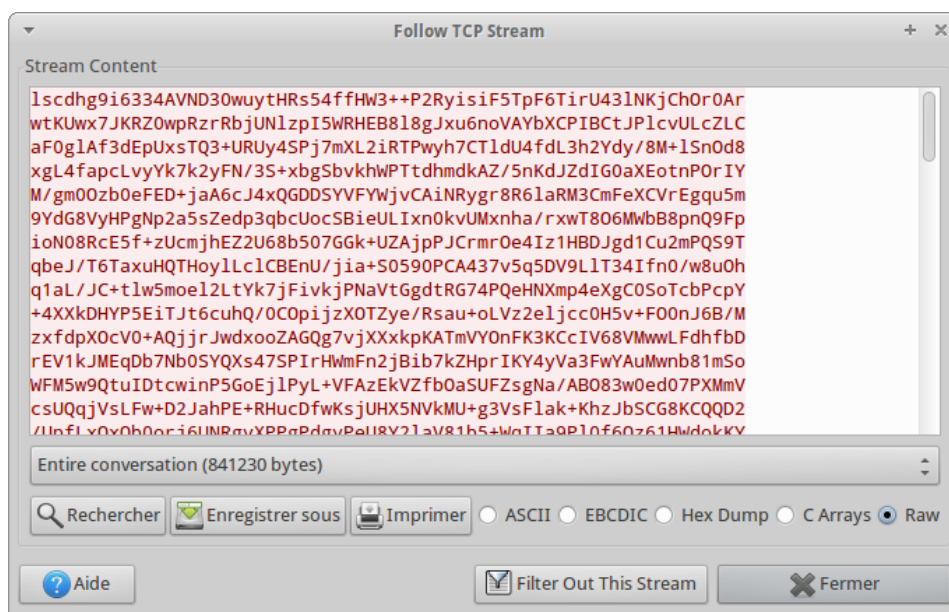


FIGURE 1.5 – Troisième conversation TCP

Cette conversation semble contenir des données codées au format Base64. Wireshark permet d'enregistrer celles-ci dans un fichier pour tenter ensuite de les décoder comme présenté ci-dessous :

```
$ base64 -d tcp-stream-3.bin > tcp-stream-3.bin.decoded
$ file tcp-stream-3.bin.decoded
tcp-stream-3.bin.decoded: data
$ hexdump -C tcp-stream-3.bin.decoded | head -n 10
00000000  96 c7 1d 86 0f 62 eb 7d f8 01 53 43 df 4c 2e ca |....b.}..SC.L..|
00000010  d1 d1 b3 9e 1f 7c 75 b7 fb e3 f6 47 28 ac 88 5e |....|u...G(..^|
00000020  53 a4 5e 93 8a b5 38 de 53 4a 8c 28 4e af 40 2b |S.^...8.SJ.(N. @+|
00000030  c2 d2 94 c3 1e c9 29 16 74 c2 94 73 ad 16 e3 50 |.....).t..s...P|
```

```

00000040  d9 73 a4 8e 56 44 71 01 f2 5f 20 27 1b ba 9e 85 |.s..VDq.._ '....|
00000050  40 61 b5 c2 3c 80 42 b4 93 e5 72 f5 0b 71 92 c2 |@a..<.B...r..q..|
00000060  68 5d 20 94 07 f7 74 4a 54 c6 c4 d0 df e5 11 53 |h] ...tJT.....S|
00000070  2e 12 3e 3e e6 5c bd a2 45 33 f0 ca 1e c2 4e 57 |..>>.\..E3....NW|
00000080  54 e1 f7 4b de 1d 98 77 2f fc 33 e9 52 9c e7 7c |T..K...w/.3.R..||
00000090  c6 02 f8 7d aa 5c 2e fc 98 93 b9 36 c8 53 7f dd |...}. \.....6.S..|

```

Le contenu décodé ressemble à des données aléatoires. On peut alors supposer qu'il s'agit de l'archive chiffrée transférée par FTP. Il reste alors à identifier la clé de déchiffrement dissimulée dans les canaux cachés évoqués dans la première conversation TCP.

1.2 Identification des canaux cachés

L'utilisation de Scapy va permettre d'inspecter les paquets contenus dans la capture.

```

Welcome to Scapy (2.2.0)
>>> capture = rdpcap("../input/dump.bin")
>>> capture
<dump.bin: TCP:163 UDP:0 ICMP:65 Other:0>

```

On retrouve bien les 228 paquets détectés par Wireshark. Le rôle des données transférées par TCP a été identifié lors du chapitre précédent. Par contre, la capture contient également 65 messages ICMP qui n'ont pas d'utilité apparente. Il semble alors naturel de s'intéresser à ces messages pour découvrir de potentiels canaux cachés.

La session Scapy présentée ci-dessous permet d'afficher le contenu du premier paquet ICMP.

```

>>> icmp_packets = capture.filter(lambda p: ICMP in p)
>>> p0 = icmp_packets[0]
>>> p0[IP].show()
###[ IP ]###
  version= 4L
  ihl= 5L
  tos= 0x2
  len= 84
  id= 0
  flags= DF
  frag= 0L
  ttl= 30
  proto= icmp
  checksum= 0xd93d
  src= 192.168.1.13
  dst= 192.168.1.12
  \options\
###[ ICMP ]###
  type= echo-request
  code= 0
  checksum= 0x71ce
  id= 0xf132
  seq= 0x1

```

L'enregistrement dans un fichier de la sortie de la méthode show permet de comparer le contenu de deux paquets, comme présenté ci-dessous :

```

$ diff -u p0.txt p33.txt
--- p0.txt      2013-05-15 11:18:14.502591653 +0200
+++ p33.txt     2013-05-15 11:19:24.390595075 +0200
@@ -1,22 +1,22 @@
###[ IP ]###

```

```

    version= 4L
    ihl= 5L
-   tos= 0x2
+   tos= 0x4
    len= 84
    id= 0
    flags= DF
    frag= 0L
-   ttl= 30
+   ttl= 20
    proto= icmp
-   checksum= 0xd93d
+   checksum= 0xe33b
    src= 192.168.1.13
    dst= 192.168.1.12
    \options\
####[ ICMP ]####
    type= echo-request
    code= 0
-   checksum= 0x71ce
-   id= 0xf132
+   checksum= 0x2d8c
+   id= 0xa35
    seq= 0x1

```

On constate alors les différences suivantes :

- au niveau IP :
 - le champ `ttl`,
 - le champ `tos`,
 - la somme de contrôle `checksum`;
- au niveau ICMP :
 - la somme de contrôle `checksum`,
 - le champ `id`.

Les deux sommes de contrôle doivent être exclues car elles sont calculées à partir des valeurs des autres champs utiles du paquet. Il reste alors à considérer les champs `ttl` et `tos` de la couche IP et le champ `id` de la couche ICMP.

Scapy permet de représenter graphiquement les différentes valeurs de ces champs. Par exemple, la commande ci-dessous génère un graphique représentant les valeurs du champ `id` de la couche ICMP.

```

>>> icmp_packets.plot(lambda p: p[ICMP].id)
<Gnuplot._Gnuplot.Gnuplot instance at 0x2e02e60>

```

Le résultat obtenu est présenté à la figure 1.6.

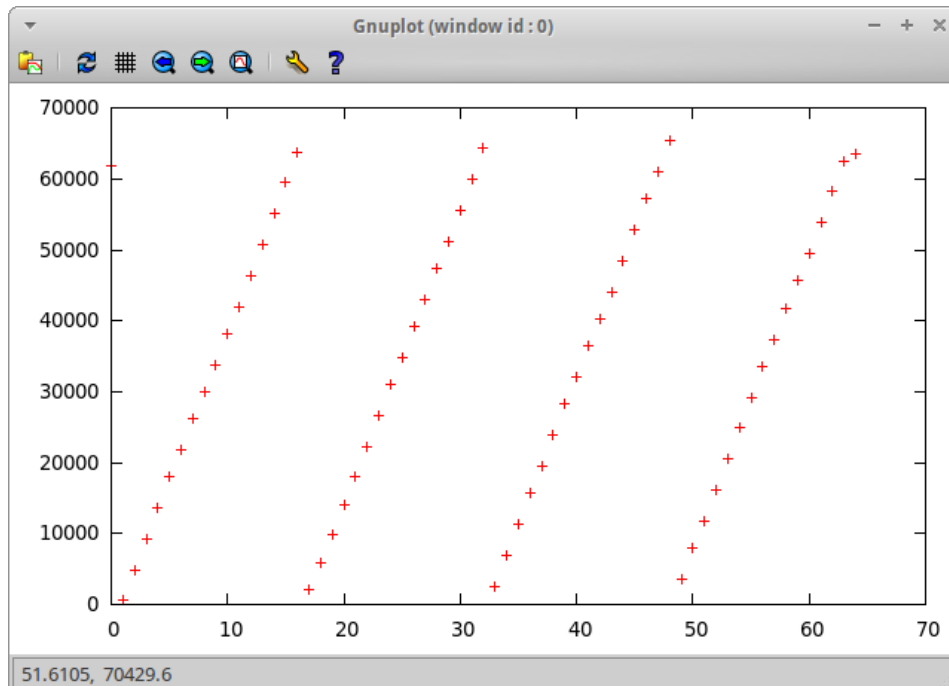


FIGURE 1.6 – Représentation des ID ICMP

L'axe des ordonnées et des abscisses correspondent respectivement aux différentes valeurs présentes pour le champ `id` et au numéro du paquet. On constate que les valeurs du champ `id` sont dispersées entre 0 et 70000. 3 bits de canaux cachés ne peuvent représenter que 8 valeurs, ce champ ne peut pas constituer un canal caché.

La même démarche est alors appliquée pour le champ `tos` de la couche IP.

```
>>> icmp_packets.plot(lambda p: p[IP].tos)
<Gnuplot._Gnuplot.Gnuplot instance at 0x2e02e60>
```

Le résultat obtenu est présenté à la figure 1.7.

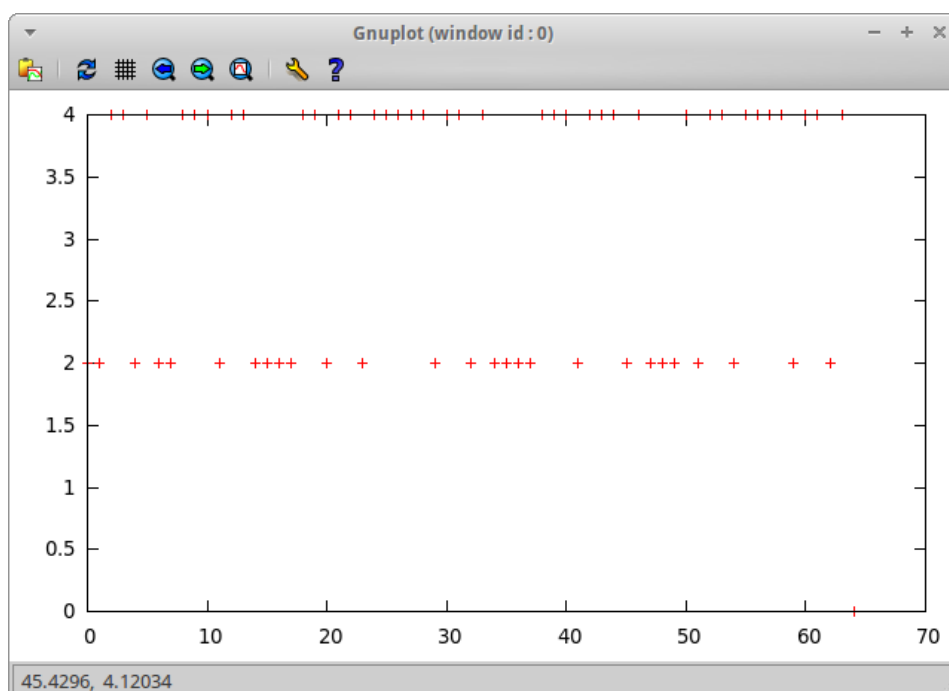


FIGURE 1.7 – Représentation du champ `tos`

Le champ `tos` ne prend que deux valeurs (2 et 4) qu'il est possible de coder sur un seul bit.

Enfin, la commande `Scapy icmp_packets.plot(lambda p: p[IP].ttl)` permet de représenter la valeur du champ `ttl` de la couche IP. Le résultat est présenté à la figure 1.8.

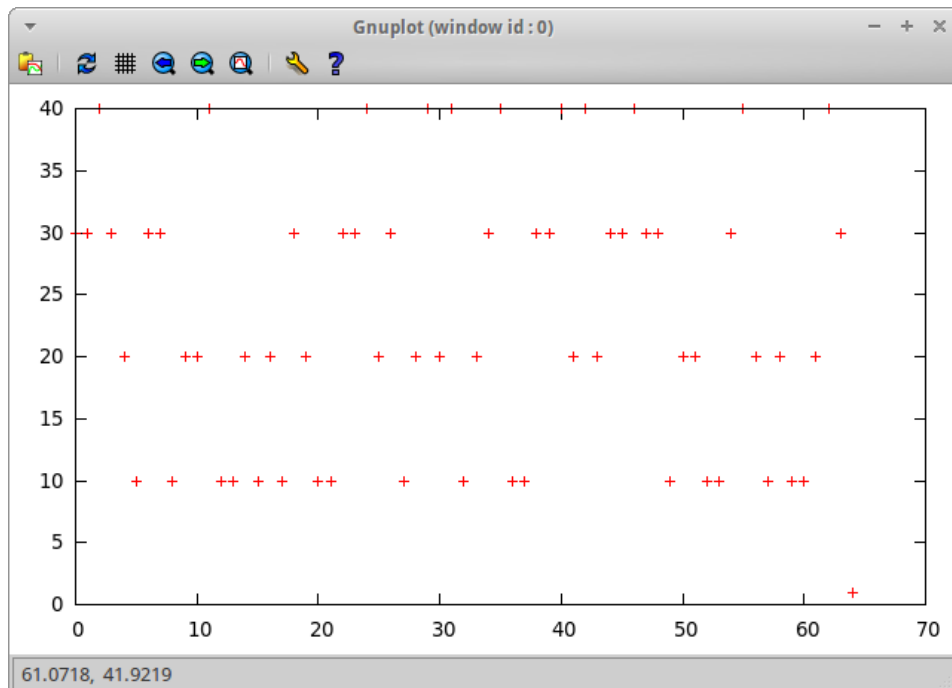


FIGURE 1.8 – Représentation du champ `ttl`

Le champ `ttl` ne prend que quatre valeurs (10, 20, 30 et 40) qui peuvent être codées sur deux bits.

Les champs `tos` et `ttl` de la couche IP semblent alors constituer de bons candidats comme canaux cachés. Il reste alors à identifier le canal caché temporel.

La première intuition est de s'intéresser au délai constaté entre l'émission de deux paquets. Ces délais peuvent être représentés graphiquement avec la session Scapy ci-dessous :

```
>>> delays = []
>>> ptime = None
>>> for p in icmp_packets:
...     if ptime:
...         delays.append(p.time - ptime)
...         ptime = p.time
...
>>> g1 = Gnuplot.Gnuplot()
>>> g1.plot(delays)
```

Le résultat obtenu est présenté à la figure 1.9.

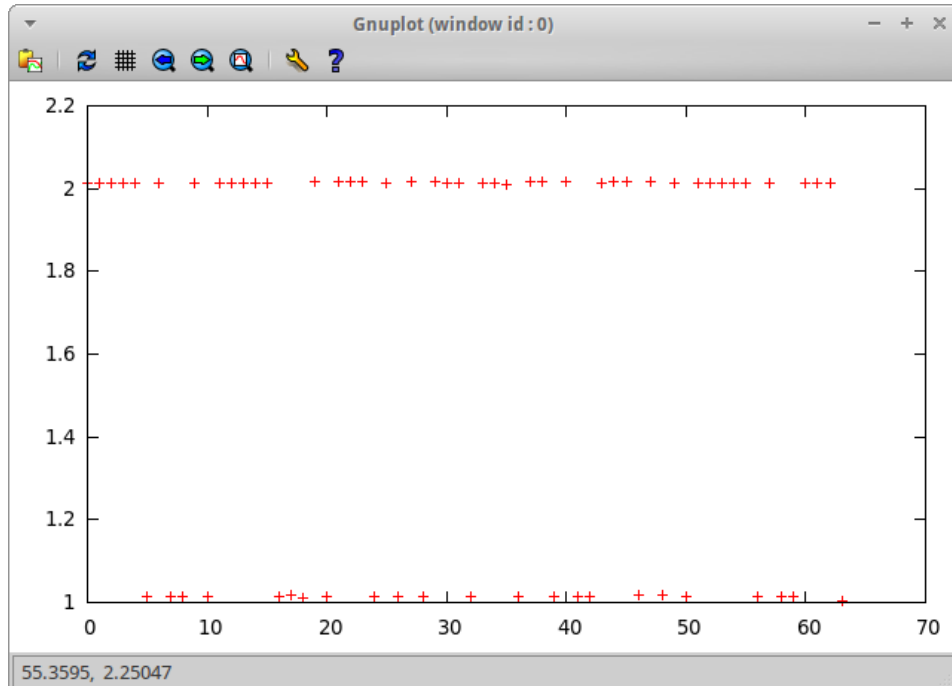


FIGURE 1.9 – Représentation du délai entre deux paquets

Les délais constatés sont centrés autour de deux valeurs (1 et 2) qui peuvent être représentées par un seul bit. Le délai entre deux paquets constitue donc un bon candidat pour stocker un bit de canal caché.

A ce stade, les hypothèses suivantes sont alors retenues :

- le bit de canal caché temporel correspond au délai d'émission entre deux messages ICMP ;
- le champ `ttl` constitue un canal caché de deux bits ;
- le champ `tos` constitue un canal caché d'un bit.

1.3 Détermination de la clé et déchiffrement

Maintenant qu'on suppose les canaux cachés identifiés, il doit être possible d'extraire la clé de déchiffrement de la capture. Cependant, le mode de codage des bits contenus dans les canaux cachés reste inconnu.

Dans le cas du champ `ttl`, quatre valeurs ont pu être observées : 10, 20, 30 et 40. Ces valeurs peuvent être codées sur 2 bits selon un codage à déterminer. Par exemple, la valeur 10 pourrait être codée par 00, 01, 10 ou 11. Il en est de même pour les valeurs 20, 30 ou 40. Le nombre de modes de codage possible correspond au nombre de permutations dans l'ensemble de valeurs à considérer, par exemple $4! = 8$ dans le cas du champ `ttl`.

Tous les modes de codage doivent alors être testés pour chaque canal caché. Cela correspond alors à $2! \times 2! \times 4! = 2 \times 2 \times 24 = 96$ combinaisons.

La première session TCP contient l'indication suivante : « 1 bit de canal cache temporel concatene a 3 bits de canal cache non temporel ». On peut donc supposer que pour chaque paquet, la clé doit être construite avec d'abord le bit du canal caché temporel puis les trois bits contenus dans le canal caché de données. Cependant, l'ordre dans lequel doivent être concaténés le bit du champ `tos` et les deux bits du champ `ttl` est inconnu. Il faut donc considérer les deux possibilités, ce qui porte le nombre de combinaisons à tester à $2 \times 96 = 192$.

Pour déterminer si une clé candidate est valide, l'empreinte MD5 des données déchiffrées par la clé est comparée à l'empreinte MD5 indiquée dans la première session TCP, à savoir `61c9392f617290642f9a12499de6b688`. Les 65¹ paquets ICMP permettent d'obtenir $64 \times 4 = 256$ bits de clé. On suppose alors que l'archive a été chiffrée à l'aide de l'algorithme AES-256.

Le script Ruby `solve-part1.rb` disponible à l'annexe A.1 permet de tester l'ensemble de ces combinaisons et s'arrête dès

1. 65 paquets sont nécessaires pour obtenir 64 délais

que les empreintes MD5 correspondent.

Le script utilise l'extension `ruby-pcap` pour obtenir les paquets ICMP depuis le fichier `dump.bin` puis appelle la fonction `process_icmp_packets` ci-dessous pour extraire les valeurs contenues dans les canaux cachés :

```
def process_icmp_packets(packets)
  pkt_info = []
  delta, tos, ttl = [], [], []
  previous_pkt_time = nil

  packets.each do |pkt|
    pkt_time = pkt.time

    if previous_pkt_time then
      delta << (pkt_time - previous_pkt_time).round
    end
    previous_pkt_time = pkt_time

    next if pkt.ip_tos == 0
    tos << pkt.ip_tos
    ttl << pkt.ip_ttl
  end

  (packets.size - 1).times do |i|
    pkt_info << { delta: delta[i], tos: tos[i], ttl: ttl[i] }
  end

  pkt_info
end
```

Le dernier paquet, pour lequel le champ `tos` vaut 0, n'est utilisé que pour calculer le dernier délai. Ensuite, le script va parcourir les différentes combinaisons possibles des modes de codage des canaux cachés afin de générer une clé candidate pour chaque combinaison. Le code ci-dessous implémente cet algorithme.

```
delta_to_idx = { 1 => 0, 2 => 1 }
tos_to_idx = { 2 => 0, 4 => 1 }
ttl_to_idx = { 10 => 0, 20 => 1, 30 => 2, 40 => 3 }

%w{ 0 1 }.permutation.each do |m_delta|
  %w{ 0 1 }.permutation.each do |m_tos|
    %w{ 00 10 01 11 }.permutation.each do |m_ttl|

      key1, key2 = "", ""
      pkt_info.each do |info|
        delta = m_delta[ delta_to_idx[ info[:delta] ] ]
        tos = m_tos[ tos_to_idx[ info[:tos] ] ]
        ttl = m_ttl[ ttl_to_idx[ info[:ttl] ] ]
        key1 << delta << tos << ttl
        key2 << delta << ttl << tos
      end

      [ key1, key2 ].each do |k|
        begin
          binkey = k.scan(/.{8,8}/).map {|x| x.to_i(2) }.pack('C*')
          r = decrypt_file(iv, sstic_tar_gz_chiffre, binkey)

          if Digest::MD5.hexdigest(r) == target_md5sum then
            $stdout.write r
            exit(0)
          end
        end
      end
    end
  end
end
```

```

    rescue OpenSSL::Cipher::CipherError => e
      $stderr.puts "[-] error: #{e}"
    end
  end
end
end
end
end

```

La fonction `decrypt_file` permet de déchiffrer les données passées en argument avec l'algorithme AES-256 en mode CBC. Cette fonction est présentée ci-dessous :

```

def decrypt_file(iv, data, key)
  cipher = OpenSSL::Cipher.new("aes-256-cbc")
  cipher.decrypt
  cipher.iv = iv
  cipher.key = key

  result = cipher.update(data)
  result << cipher.final
  result
end

```

Le vecteur d'initialisation passé en argument doit correspondre à celui indiqué dans la première session TCP du fichier `dump.bin`, à savoir `76C128D46A6C4B15B43016904BE176AC`.

L'exécution de ce script sur le fichier `dump.bin` permet d'obtenir la clé correcte et de déchiffrer le contenu de l'archive :

```

$ ./solve-part1.rb dump.bin > out.tar.gz
[*] solving part 1
[+] iv = 76C128D46A6C4B15B43016904BE176AC
[+] target md5sum = 61c9392f617290642f9a12499de6b688
[+] md5sum(r) = 61c9392f617290642f9a12499de6b688
[!] key = dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94
$ file out.tar.gz
out.tar.gz: gzip compressed data, was "archive.tar", from Unix, last modified: Mon Mar 18 12:24:37 2013

```

L'étude du contenu de l'archive obtenue fait l'objet du chapitre suivant.

Chapitre 2

Analyse du FPGA

2.1 Découverte

Le fichier obtenu à l'étape précédente est une archive tar dont le contenu est présenté ci-dessous :

```
$ tar ztvf out.tar.gz
drwxrwxr-x ealata/ealata    0 2013-03-18 12:21 archive/
-rw-rw-r-- ealata/ealata  1394 2013-03-18 12:21 archive/smp.py
-rw-rw-r-- ealata/ealata 44053 2013-03-18 12:21 archive/data
-rw-r--r-- ealata/ealata 3226052 2013-03-18 12:21 archive/s.ng
-rw-rw-r-- ealata/ealata   1035 2013-03-18 12:21 archive/decrypt.py
```

Quatre fichiers sont inclus dans l'archive, dont deux scripts écrits en Python.

Le fichier `smp.py` est une simple déclaration d'un tableau de 231 octets :

```
$ cat smp.py
smp = [0x00,
0xb0,
0x10,
0xd0,
0xb7,
[...],
0x00,
0xbd,]
$ wc -l smp.py
231 smp.py
```

Le contenu du fichier `decrypt.py` est présenté ci-dessous :

```
#!/usr/bin/python

import sys
import base64
import md5

import dev
import smp

if len(sys.argv) != 2:
    print("usage: %s key" % sys.argv[0])
    sys.exit(1)
```

```

key = int(sys.argv[1], 16)
key = [(key >> (i * 8)) & 0xff for i in range(16)]

result = []
d = open("data", "rb").read()
dev.init("sp.ngr")
for i in range(0, len(d), 224):
    smd = d[i : (i + 224)]
    smd = (key, len(smd), smd)
    dev.send_smd(smd)
    dev.send_smp(smp.smp)
    dev.start()
    dev.wait_finished()
    result = result + dev.get_data()

print "".join(result)
result_md5 = md5.new()
result_md5.update("".join(result))
result_md5 = result_md5.digest()
result_md5 = [ord(x) for x in result_md5]
target_md5 = "6c0708b3cf6e32cbae4236bdea062979"
target_md5 = [int(target_md5[x:(x + 2)], 16) for x in range(0, len(target_md5), 2)]
print(["%02x" % x for x in target_md5])
print(["%02x" % x for x in result_md5])
if result_md5 != target_md5:
    print("Bad key...")
    sys.exit(1)

result = base64.b64decode("".join(result))
d = open("atad", "wb")
d.write(result)
d.close()
sys.exit(0)

```

Ce script calcule une clé constituée de 16 octets à partir d'une chaîne hexadécimale passée en argument, initialise un périphérique `dev` avec le contenu du fichier `sp.ngr` puis démarre une boucle qui réalise les traitements suivants :

- lecture, au maximum de 224 octets, de données depuis le fichier `data` ;
- envoi au périphérique, à l'aide de la méthode `send_smd`, d'un tuple constitué des données suivantes :
 - la clé issue de l'argument passé en ligne de commande,
 - la longueur des données lues depuis le fichier `data`,
 - les données lues depuis le fichier `data` ;
- envoi au périphérique, à l'aide de la méthode `send_smp`, du tableau `smp` initialisé par le fichier `smp.py` ;
- déclenchement d'un traitement sur le périphérique en appelant la méthode `start` ;
- récupération des données sur le périphérique avec la méthode `get_data` puis ajout au tableau `result`.

La boucle termine quand toutes les données du fichier `data` ont été envoyées au périphérique. Ensuite, une chaîne est générée en concaténant les données du tableau `result`. L'empreinte MD5 de cette chaîne est comparée à la valeur `6c0708b3cf6e32cbae4236bdea062979`. Si les deux empreintes diffèrent, le script s'arrête en affichant le message `Bad key...`. Sinon, le contenu de la chaîne est décodé avec la méthode `b64decode` et le résultat est stocké dans le fichier `atad`.

En l'état, il n'est pas possible d'exécuter le script `decrypt.py`, le module `dev` étant introuvable :

```

$ ./decrypt.py
Traceback (most recent call last):
  File "./decrypt.py", line 7, in <module>
    import dev
ImportError: No module named dev

```

Il va donc être nécessaire de réaliser une analyse statique sur les fichiers disponibles. Une brève analyse du fichier `data` n'apporte pas d'informations supplémentaires :

```
$ file data
data: data
$ hexdump -C data | head -n 5
00000000 d4 8f 0a a2 84 0d 2a 4a a3 da 71 e5 58 1b eb 93 |.....*J..q.X...|
00000010 60 69 c2 8a 84 cd da 20 b7 4e 51 c5 44 db cb b3 |`i.....NQ.D...|
00000020 7c 69 8e 8a c4 25 da a8 cf 60 49 45 54 db 97 eb ||i...%...`IET...|
00000030 bc 29 96 8a 8c a3 c4 10 3f 66 61 65 98 87 bf 29 |.).....?fae...)|
00000040 bc 99 4e 5a 24 e3 3a ec ef a0 15 81 4e a7 43 4f |..NZ$.:.....N.CO|
```

Le contenu du fichier semble aléatoire, ce qui est souvent caractéristique de données chiffrées ou compressées.

Un examen similaire du fichier `s.ngr` se révèle plus fructueux :

```
$ file s.ngr
s.ngr: data
$ hexdump -C s.ngr | head -n 5
00000000 58 49 4c 49 4e 58 2d 58 44 42 20 30 2e 31 20 53 |XILINX-XDB 0.1 S|
00000010 54 55 42 20 30 2e 31 20 41 53 43 49 49 0a 58 49 |TUB 0.1 ASCII.XI|
00000020 4c 49 4e 58 2d 58 44 4d 20 56 31 2e 36 65 0a 24 |LINX-XDM V1.6e.$|
00000030 37 35 34 3d 7e 32 3f 33 26 62 64 61 68 21 6a 61 |754=~?3&bdah!ja|
00000040 6d 63 77 65 2a 54 62 6b 61 6f 79 6f 65 6b 20 53 |mcwe*Tbkaoyoek S|
```

En effet, la chaîne XILINX-XDB constitue une information intéressante sur la nature du fichier. Une recherche Google sur les mots clés « ngr Xilinx » aboutit à la page http://www.xilinx.com/support/documentation/sw_manuals/xilinx13/rtv_p_open_rtl.htm intitulée « Opening a Netlist File ». Selon Wikipedia, le terme « netlist » peut être utilisé pour décrire la connectivité d'un composant électronique. Xilinx étant une société connue pour la production de circuits logiques programmes (ou FPGA pour « field programmable gate array »), il est vraisemblable que le fichier `s.ngr` constitue la description d'un tel circuit.

2.2 Étude du fichier netlist

La société Xilinx fournit un environnement de développement nommé « ISE Design Suite ». Celui-ci est téléchargeable à l'adresse <http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm>.

En cas de problèmes de téléchargement, un kit de développement sur DVD (voir figure 2.1) peut également être demandé à l'adresse http://www.xilinx.com/onlinestore/dvd_fulfillment_request.htm.



FIGURE 2.1 – Environnement de développement Xilinx sur DVD

Une fois l'environnement de développement téléchargé et installé, il est possible d'ouvrir le fichier `s.ngr`, comme présenté sur la capture d'écran 2.2.

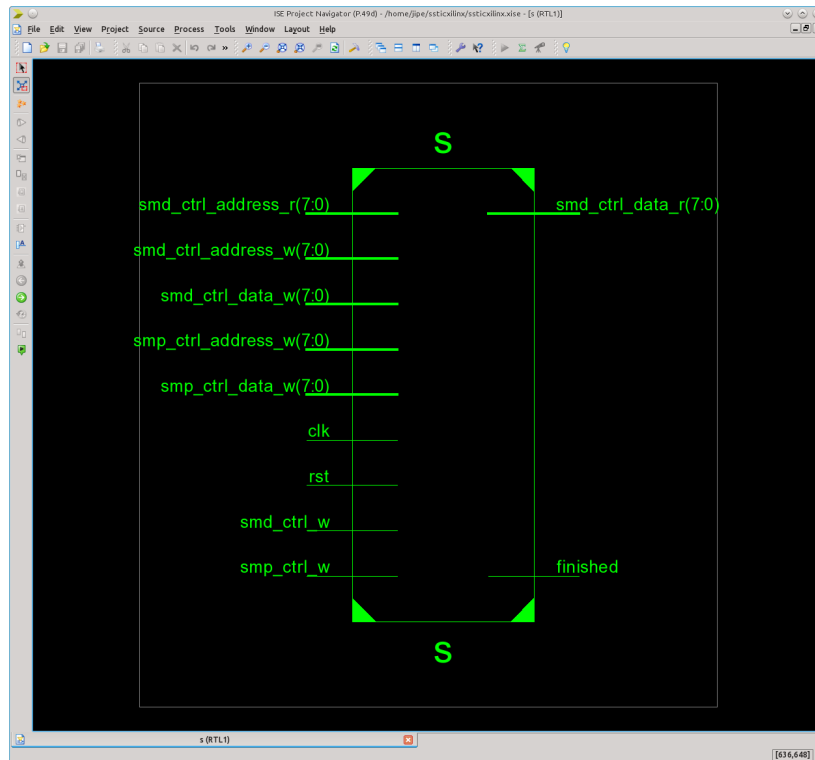


FIGURE 2.2 – Ouverture du fichier s.ngd dans ISE

Le développement du bloc S est présenté à la figure 2.3.

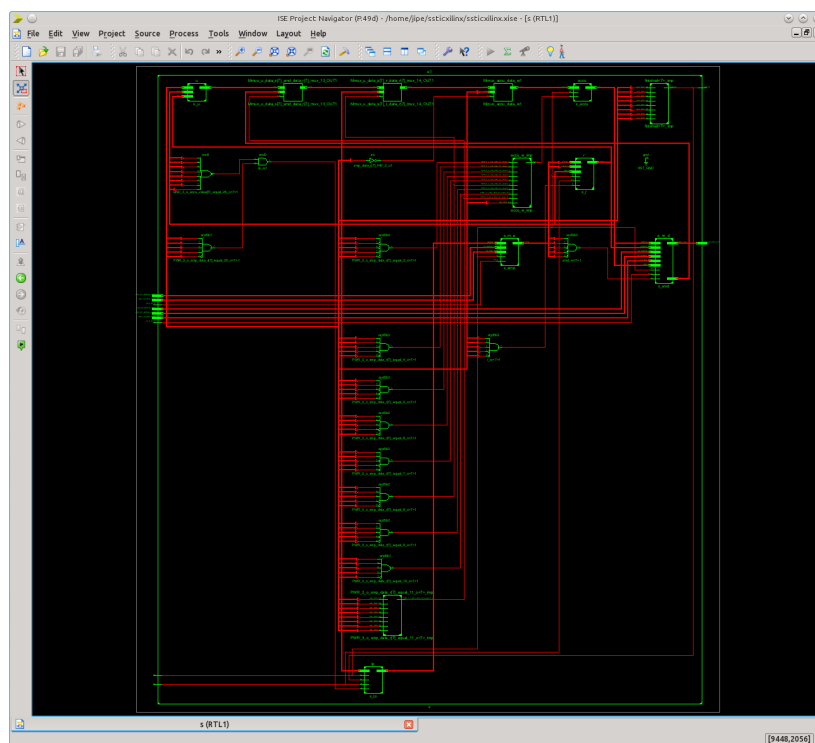


FIGURE 2.3 – Bloc S développé dans ISE

Ce bloc est rattaché à six bus de 8 bits qui sont :

- smd_ctrl_address_r;
- smd_ctrl_address_w;

- `smd_ctrl_data_r`;
- `smd_ctrl_data_w`;
- `smp_ctrl_address_w`;
- `smp_ctrl_data_w`.

On retrouve les termes de `smd` et `smp`, également utilisés dans le script `decrypt.py`. A l'aide de l'environnement ISE, il est possible de retrouver la polarité de chaque bus, c'est-à-dire si celui-ci est utilisé comme entrée ou sortie par rapport au bloc S. Tous les bus sont utilisés comme entrées, sauf `smd_ctrl_data_r` qui constitue une sortie.

Le bloc est également rattaché à cinq signaux d'un bit : `clk`, `rst`, `smd_ctrl_w`, `smp_ctrl_w` et `finished`. Ceux-ci sont utilisés comme entrées, sauf le port `finished` qui est une sortie.

Le développement du bloc S, particulièrement volumineux, est accessible en annexe [A.2.1](#).

La conception du composant fait intervenir un certain nombre de blocs principaux ainsi que de nombreuses portes logiques. L'analyse de chacun de ces blocs va se révéler nécessaire pour comprendre le fonctionnement du composant.

Ces blocs sont :

- `x_smp` (section [2.2.2](#));
- `x_smd` (section [2.2.3](#));
- `x_r` (section [2.2.4](#));
- `finished<7>_imp` (section [2.2.5](#));
- `x_ip` (section [2.2.6](#));
- `x_accu` (section [2.2.7](#));
- `accu_w_imp` (section [2.2.8](#));
- `x_u` (section [2.2.9](#)).

L'analyse détaillée de ces 8 blocs peut se révéler particulièrement fastidieuse pour le lecteur pressé. Celui-ci est invité à se rendre directement au chapitre [2.3](#) pour prendre connaissance de la synthèse de ces analyses.

2.2.1 Description des portes logiques utilisées

Le document « Xilinx 7 Series Libraries Guide for Schematic Designs », disponible en téléchargement sur le site de Xilinx ¹, peut être utilisé comme référence pour comprendre le rôle des éléments qui constituent le composant. Outre les portes logiques classiques de type `and`, `or` ou les inverseurs, deux primitives sont particulièrement importantes à connaître :

- les multiplexeurs ;
- les bascules D flip-flop.

Multiplexeur

Un multiplexeur est une primitive qui permet de réaliser une sélection sur un ensemble de données en entrée. Si le multiplexeur possède 2^n entrées, alors il doit également avoir n lignes de sélection qui permettent de sélectionner les entrées à retourner parmi les 2^n . Par exemple, la figure [2.4](#) décrit un multiplexeur qui possède deux bus de 8 bits en entrée, un bus de 8 bits en sortie et une ligne de sélection. Si cette dernière présente la valeur 0, alors les données du bus `Result` correspondra aux données du bus `Data0`. Dans le cas contraire (la ligne de sélection présente la valeur 1), les données du bus `Result` correspondra aux données du bus `Data1`.

1. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/7series_scm.pdf

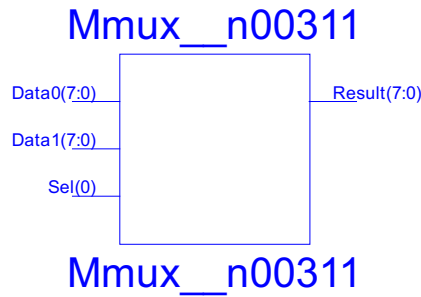


FIGURE 2.4 – Exemple de multiplexeur

Dans l'exemple précédent, la ligne de sélection n'était représentée que par un bit. On peut imaginer un multiplexeur plus complexe pour lequel la ligne de sélection serait elle même un bus de 8 bits, ce qui permet de sélectionner la sortie bit par bit entre les deux bus en entrée.

Bascule D flip-flop

La bascule D flip-flop est une primitive électronique qui permet de « figer » la valeur présentée en entrée. Elle est donc utilisée pour stocker l'information sur un état. Ce type de bascule est synchronisé sur une horloge : le changement d'état ne sera pris en compte qu'à chaque nouveau cycle d'horloge.

La figure 2.5 présente une bascule de ce type avec une entrée supplémentaire CE (pour « Clock Enable ») qui permet de désactiver la prise en compte d'un cycle horloge. Si l'entrée CE vaut 1, alors la sortie Q prendra la valeur de l'entrée D. Si CE vaut 0, la sortie Q conservera sa valeur actuelle.

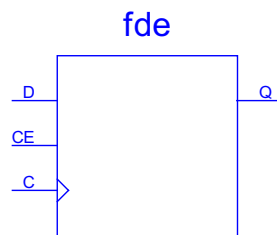


FIGURE 2.5 – Exemple de bascule D flip-flop

L'utilisation de bascules D flip-flop permet de définir des blocs plus complexes qui sont utilisés comme mémoires.

2.2.2 Bloc x_smp

La vision macroscopique du bloc x_smp est représentée sur la figure 2.6.

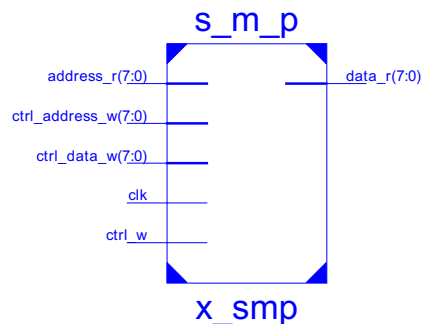


FIGURE 2.6 – Vision macroscopique du bloc x_smp

Les entrées du bloc `x_smp` sont décrites dans le tableau 2.1.

Nom	Type	Commentaire
<code>address_r</code>	bus de 8 bits	correspond à la sortie <code>ip</code> du bloc <code>x_ip</code>
<code>ctrl_address_w</code>	bus de 8 bits	correspond à l'entrée <code>smp_ctrl_address_w</code> du bloc <code>S</code>
<code>ctrl_data_w</code>	bus de 8 bits	correspond à l'entrée <code>smp_ctrl_data_w</code> du bloc <code>S</code>
<code>clk</code>	signal de 1 bit	correspond à l'entrée <code>clk</code> du bloc <code>S</code>
<code>ctrl_w</code>	signal de 1 bit	correspond à l'entrée <code>smp_ctrl_w</code> du bloc <code>S</code>

TABLE 2.1 – Entrées du bloc `x_smp`

La sortie du bloc `x_smp` est décrite dans le tableau 2.2

Nom	Type	Commentaire
<code>data_r</code>	bus de 8 bits	utilisé comme entrée par de nombreux blocs

TABLE 2.2 – Sortie du bloc `x_smp`

Le bloc `x_smp` est composé de nombreuses bascules D flip-flop qui constituent un bloc mémoire. Depuis l'extérieur du composant `S`, ce bloc peut être accédé en écriture avec les bus `smp_ctrl_address_w` et `smp_ctrl_data_w`, notamment lors de l'initialisation du périphérique réalisée par le script `decrypt.py` (méthode `send_smp`) décrit à la section 2.1.

Les autres blocs à l'intérieur du composant `S` utilisent les bus `address_r` et `data_r` pour accéder en lecture aux données de `x_smp`.

Écriture dans `x_smp`

La figure 2.7 illustre le mécanisme d'écriture dans la mémoire du bloc `x_smp`. Un premier bloc compare l'adresse présente sur le bus `ctrl_address_w` à une valeur déterminée². Si les deux valeurs sont identiques, ce bloc retourne 1 sur sa sortie. Cette sortie correspond alors à l'entrée `I0` de la primitive `and2`. La seconde entrée `I1` prend la valeur du bit `ctrl_w` qui active ou désactive l'écriture. Si ces deux entrées sont égales à 1, alors l'entrée `CE` de la bascule D flip-flop est également activée. Dans ce cas, au prochain cycle d'horloge, la bascule stockera la valeur présentée sur son entrée `D` (qui est un bus de 8 bits).

Ce schéma est itéré 256 fois dans le bloc `x_smp` pour pouvoir stocker 256 octets.

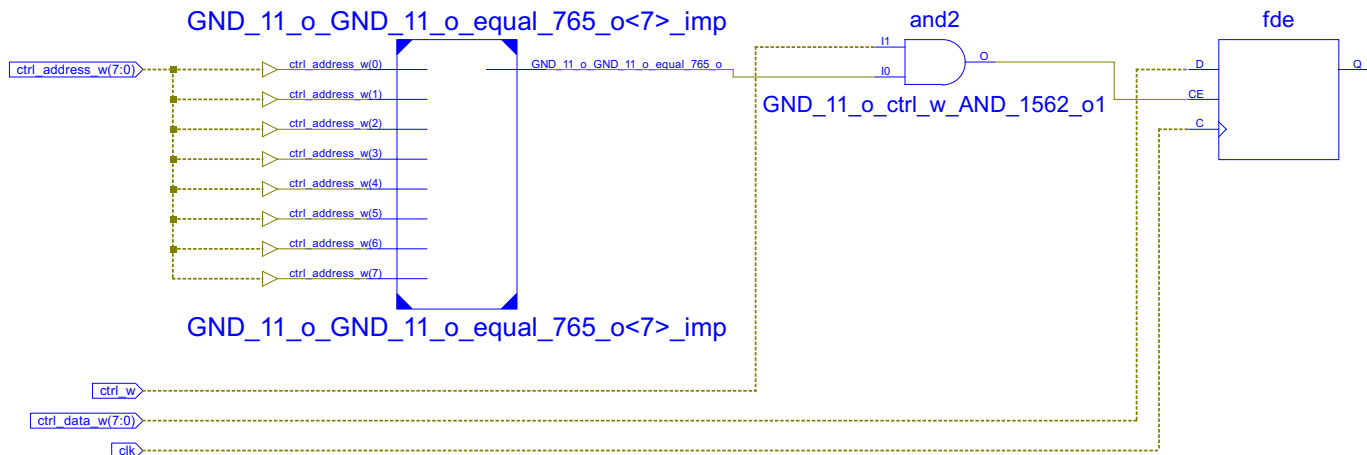


FIGURE 2.7 – Écriture d'un octet dans `x_smp`

Lecture dans `x_smp`

Les 256 bascules D flip-flop du bloc `x_smp` sont reliées aux entrées du multiplexeur `Mmux_data_r1`, comme présenté sur la figure 2.8. La sélection de l'entrée du multiplexeur est réalisée à l'aide du bus `address_r`, la sortie du multiplexeur étant alors

2. le développement du bloc permet d'identifier cette valeur qui est 16 dans cet exemple

envoyée sur le bus `data_r`.

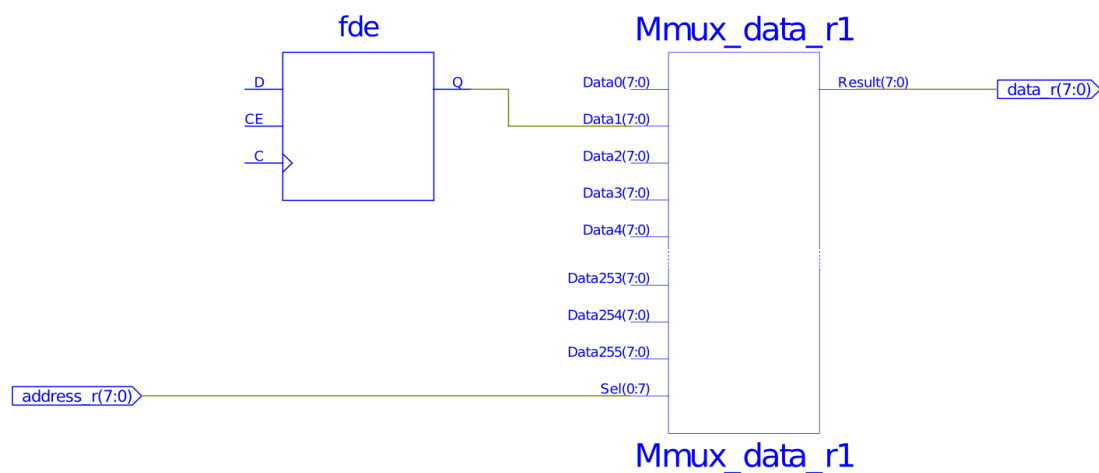


FIGURE 2.8 – Lecture d'un octet dans `x_smp`

2.2.3 Bloc `x_smd`

La vision macroscopique du bloc `x_smd` est représentée sur la figure 2.9.

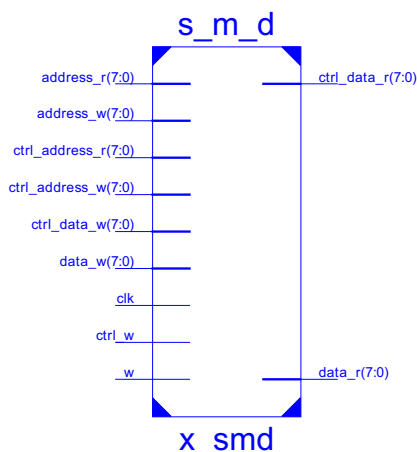


FIGURE 2.9 – Vision macroscopique du bloc `x_smd`

Les entrées du bloc `x_smd` sont décrites dans le tableau 2.3.

Nom	Type	Commentaire
<code>address_r</code>	bus de 8 bits	correspond à la sortie <code>value</code> du bloc <code>x_accu</code>
<code>address_w</code>	bus de 8 bits	correspond à la sortie <code>data_r</code> du block <code>x_r</code>
<code>ctrl_address_r</code>	bus de 8 bits	correspond à l'entrée <code>smd_ctrl_address_r</code> du bloc <code>S</code>
<code>ctrl_address_w</code>	bus de 8 bits	correspond à l'entrée <code>smd_ctrl_address_w</code> du bloc <code>S</code>
<code>ctrl_data_w</code>	bus de 8 bits	correspond à l'entrée <code>smd_ctrl_data_w</code> du bloc <code>S</code>
<code>data_w</code>	bus de 8 bits	correspond à la sortie <code>value</code> du bloc <code>x_accu</code>
<code>clk</code>	signal de 1 bit	correspond à l'entrée <code>clk</code> du bloc <code>S</code>
<code>ctrl_w</code>	signal de 1 bit	correspond à l'entrée <code>smd_ctrl_w</code> du bloc <code>S</code>
<code>w</code>	signal de 1 bit	correspond à la sortie <code>o</code> de la primitive <code>smd_w<7>1</code>

TABLE 2.3 – Entrées du bloc `x_smd`

Les sorties du bloc `x_smd` sont décrites dans le tableau 2.4.

Nom	Type	Commentaire
ctrl_data_r	bus de 8 bits	correspond à la sortie <code>smd_ctrl_data_r</code> du bloc S
data_r	bus de 8 bits	correspond à l'entrée Data1 du multiplexeur <code>Mmux[...]_13_OUT1</code>

TABLE 2.4 – Sorties du bloc `x_smd`

Ce bloc constitue une mémoire de 256 octets utilisable en lecture et écriture. Les bus `ctrl_address_w` et `ctrl_data_w` sont utilisés pour initialiser le contenu de la mémoire depuis l'extérieur du composant S (méthode `send_smd` dans le script `decrypt.py`). Les bus `ctrl_address_r` et `ctrl_data_r` servent à lire la mémoire depuis l'extérieur du composant S (méthode `get_data` de `decrypt.py`).

Le bloc peut également être accédé en lecture et écriture depuis l'intérieur du composant S à l'aide des bus `address_w`, `data_w`, `address_r` et `data_r`.

Écriture dans `x_smd`

L'écriture dans le bloc `x_smd` s'effectue de façon analogue à celle réalisée dans le bloc `x_smp`. La sélection de la bascule D flip-flop à mettre à jour est rendue plus complexe par la nécessité de prendre en compte à la fois le bus `ctrl_address_w` et le bus `address_w`.

L'écriture est possible si l'un des deux signaux `ctrl_w` ou `w` vaut 1. Dans le cas du signal `w`, la valeur correspond à la sortie de la primitive `smd_w<7>1` de type `and5b3`. Les entrées de cette primitive sont représentées à la figure 2.10.

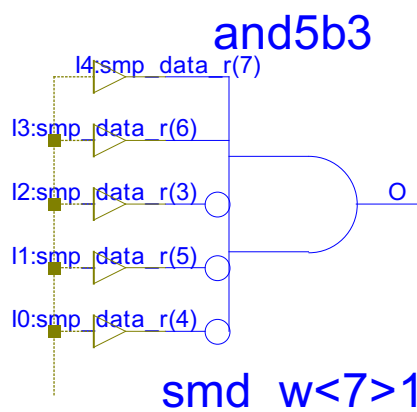


FIGURE 2.10 – Entrées de `smd_w<7>1`

La correspondance des entrées de `smd_w<7>1` avec les 5 premiers bits du bus `smp_data_r` du bloc `x_smp` est la suivante :

- `I4 = smp_data_r(7);`
- `I3 = smp_data_r(6);`
- `I2 = smp_data_r(3);`
- `I1 = smp_data_r(5);`
- `I0 = smp_data_r(4);`

La sortie d'une porte logique `and5b3` vaut 1 si les trois premières entrées sont égales à 0 et si les deux dernières égales sont à 1, ce qui se traduit par les conditions ci-dessous sur la valeur du bus `data_r` du bloc `x_smp` :

- `smp_data_r(0) = indéfini;`
- `smp_data_r(1) = indéfini;`
- `smp_data_r(2) = indéfini;`
- `smp_data_r(3) = 0;`
- `smp_data_r(4) = 0;`
- `smp_data_r(5) = 0;`
- `smp_data_r(6) = 1;`
- `smp_data_r(7) = 1.`

Si le bloc `x_smp` présente sur sa sortie une valeur de la forme `11000xxx` (où `x` peut valoir 0 ou 1), alors la valeur présentée par le bloc `x_accu` sera écrite dans `x_smd` à l'adresse en sortie du bloc `x_r`.

Lecture dans `x_smd`

Le mécanisme de lecture est similaire à celui implémenté dans le bloc `x_smp`. La différence majeure réside dans l'utilisation de deux multiplexeurs, `Mmux_ctrl_data_r1` et `Mmux_data_r1`, reliés tous les deux aux 256 bascules D flip-flop. Le premier multiplexeur va effectuer sa sélection à partir du bus `ctrl_address_r` et envoyer sa sortie sur le bus `ctrl_data_r`. Le second va utiliser le bus `address_r` pour sa sélection et envoyer sa sortie sur le bus `data_r`.

2.2.4 Bloc `x_r`

La vision macroscopique du bloc `x_r` est représentée sur la figure 2.11.

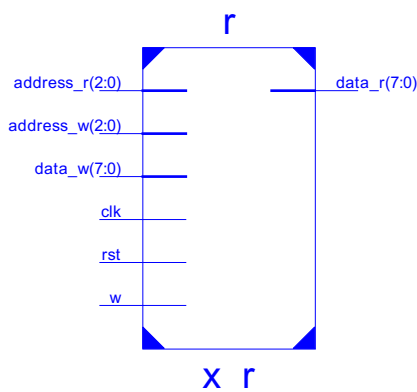


FIGURE 2.11 – Vision macroscopique du bloc `x_r`

Les entrées du bloc `x_r` sont décrites dans le tableau 2.5.

Nom	Type	Commentaire
<code>address_r</code>	bus de 3 bits	correspond à <code>smp_data_r(2:0)</code> (3 derniers bits)
<code>address_w</code>	bus de 3 bits	correspond à <code>smp_data_r(2:0)</code> (3 derniers bits)
<code>data_w</code>	bus de 8 bits	correspond à la sortie <code>value</code> du bloc <code>accu</code>
<code>clk</code>	signal de 1 bit	correspond à l'entrée <code>clk</code> du bloc <code>S</code>
<code>rst</code>	signal de 1 bit	correspond à l'entrée <code>rst</code> du bloc <code>S</code>
<code>w</code>	signal de 1 bit	correspond à la sortie <code>o</code> du bloc <code>r_w<7>1</code>

TABLE 2.5 – Entrées du bloc `x_r`

La sortie du bloc `x_r` est décrite dans le tableau 2.6.

Nom	Type	Commentaire
<code>data_r</code>	bus de 8 bits	utilisé en entrée par plusieurs blocs

TABLE 2.6 – Sortie du bloc `x_r`

Le développement du bloc `x_r` fait apparaître la présence de 8 bascules D flip-flop pouvant contenir chacune un octet. Chaque bascule peut être sélectionnée en lecture ou écriture selon la valeur présente sur les bus `address_r` et `address_w`. Ces bus correspondent aux trois derniers bits présents sur le bus `smp_data_r`.

Le bloc `x_r` semble donc correspondre à un banc de 8 registres.

La sortie de ce bloc est utilisée en entrée par les blocs `x_u`, `x_smd`, `x_ip` et par un multiplexeur.

Écriture dans le bloc x_r

Une première primitive `and` à trois entrées permet de tester la valeur présente sur le bus `address_w` et donc de sélectionner le registre. La sortie de cette primitive est envoyée sur une autre primitive `and` dont la seconde entrée correspond à la valeur présente sur `w`, ce qui permet d'activer ou désactiver l'écriture sur le registre. Ces primitives sont représentées sur la figure 2.12.

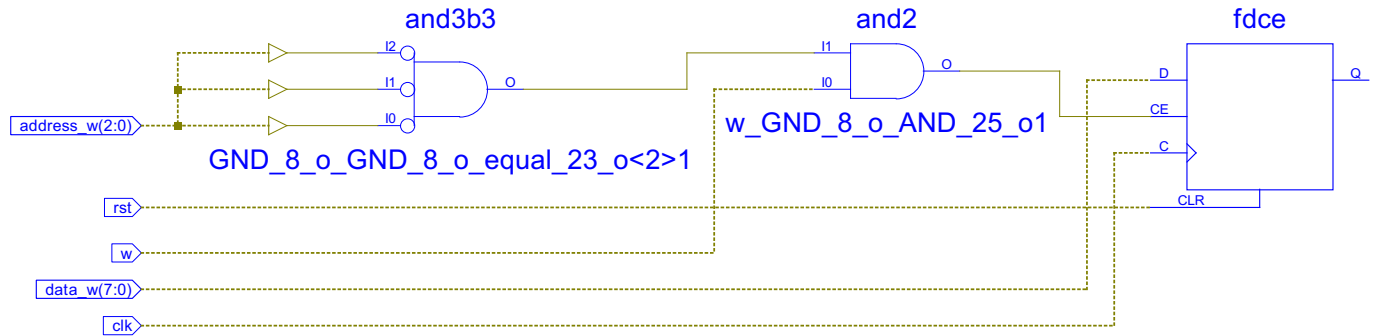


FIGURE 2.12 – Écriture d'un octet dans x_r

Si les deux conditions sont réunies alors l'entrée CE de la bascule sera activée et la valeur présente sur le bus `data_w` sera stockée au prochain cycle d'horloge.

Ce schéma est répété 8 fois pour chacune des bascules D flip-flop.

On peut également remarquer que le bit `rst` permet de réinitialiser l'état des 8 bascules.

Il reste maintenant à déterminer les conditions d'activation du signal `w`. Ce signal correspond à la valeur de sortie de la primitive `r_w<7>1` représentée à la figure 2.13.

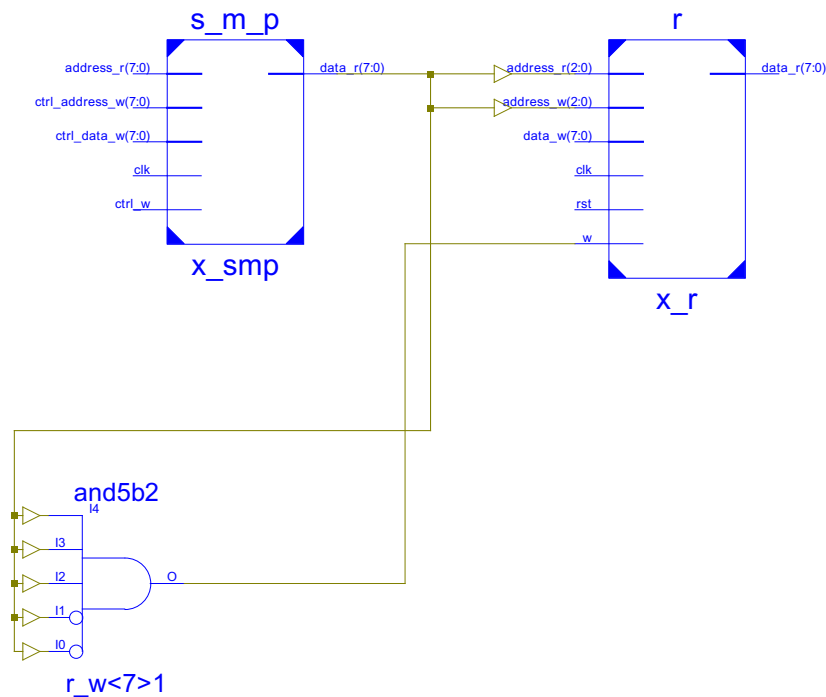


FIGURE 2.13 – Activation du signal w

Les entrées de la primitive `r_w<7>1` sont représentées à la figure 2.14.

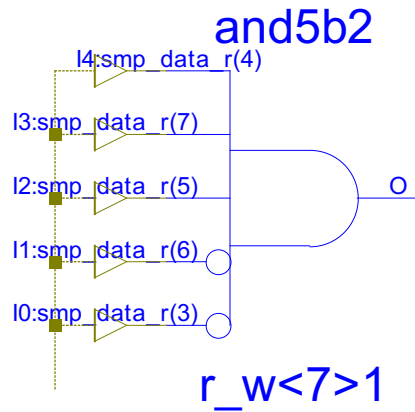


FIGURE 2.14 – Entrées de $r_w<7>1$

La correspondance de cette primitive avec les 5 premiers bits de la sortie du bloc `x_smp` est la suivante :

- `I4 = smp_data_r(4);`
- `I3 = smp_data_r(7);`
- `I2 = smp_data_r(5);`
- `I1 = smp_data_r(6);`
- `I0 = smp_data_r(3);`

Pour que la sortie de la primitive `and5b2` soit égale à 1, il faut que ses deux premières entrées soient égales à 0 et les trois dernières à 1. Le bus `smp_data_r` doit donc vérifier les conditions suivantes :

- `smp_data_r(0) = indéfini;`
- `smp_data_r(1) = indéfini;`
- `smp_data_r(2) = indéfini;`
- `smp_data_r(3) = 0;`
- `smp_data_r(4) = 1;`
- `smp_data_r(5) = 1;`
- `smp_data_r(6) = 0;`
- `smp_data_r(7) = 1.`

L'écriture dans le bloc `x_r` est donc possible si la valeur présentée par le bloc `x_smp` est de la forme `10110xxx` où `x` vaut 0 ou 1.

Lecture du bloc `x_r`

Les 8 bascules D flip-flop sont reliées au multiplexeur `Mmux_data_r1` (sur la figure 2.15, seule la première bascule est représentée). La valeur présente sur le bus `address_r` permet de sélectionner la bascule dont la valeur sera envoyée sur le bus `data_r`.

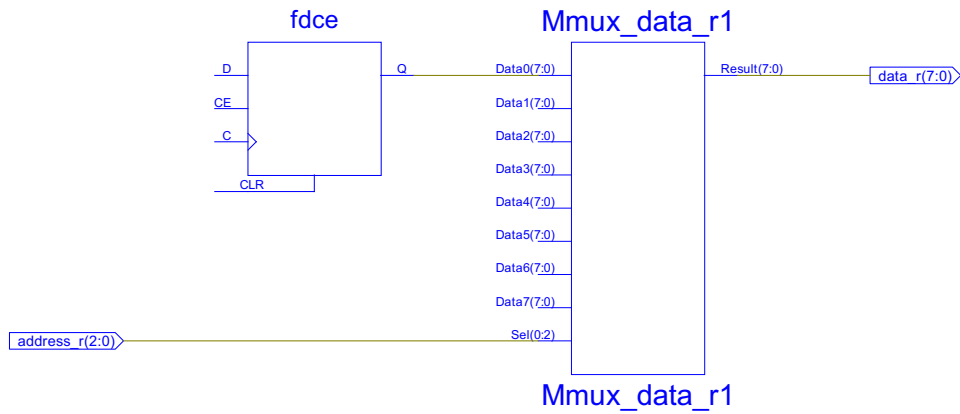


FIGURE 2.15 – Lecture d'un octet dans x_r

2.2.5 Bloc finished<7>_imp

La vision développée du bloc finished<7>_imp est représentée sur la figure 2.16.

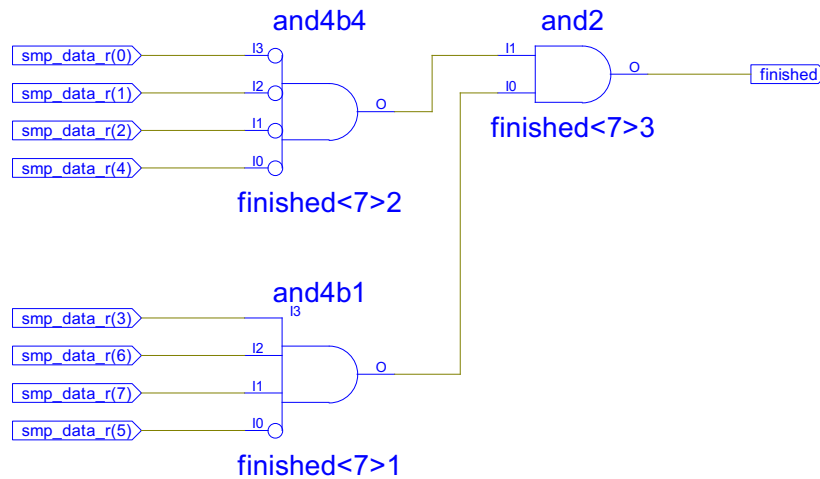


FIGURE 2.16 – Vision développée du bloc finished<7>_imp

L'entrée du bloc finished<7>_imp est décrite dans le tableau 2.7.

Nom	Type	Commentaire
smp_data_r	bus de 8 bits	correspond à la sortie data_r du bloc x_smp

TABLE 2.7 – Entrée du bloc finished<7>_imp

La sortie du bloc finished<7>_imp est décrite dans le tableau 2.8.

Nom	Type	Commentaire
finished	signal de 1 bit	correspond à la sortie finished du bloc S

TABLE 2.8 – Sortie du bloc finished<7>_imp

D'après le schéma du bloc, pour que la sortie finished soit égale à 1, les conditions ci-dessous doivent être vérifiées sur smp_data_r :

- smp_data_r(0) = 0 ;
- smp_data_r(1) = 0 ;

- smp_data_r(2) = 0;
- smp_data_r(3) = 1;
- smp_data_r(4) = 0;
- smp_data_r(5) = 0;
- smp_data_r(6) = 1;
- smp_data_r(7) = 1.

Ces conditions sont équivalentes à $\text{smp_data_r} = 11001000 = 0xc8$. Autrement dit, si la valeur $0xc8$ est présente en sortie du bloc x_smp , la sortie $finished$ du bloc S est égale à 1, ce qui termine l'exécution du composant.

2.2.6 Bloc x_ip

La vision macroscopique du bloc x_ip est représentée sur la figure 2.17.

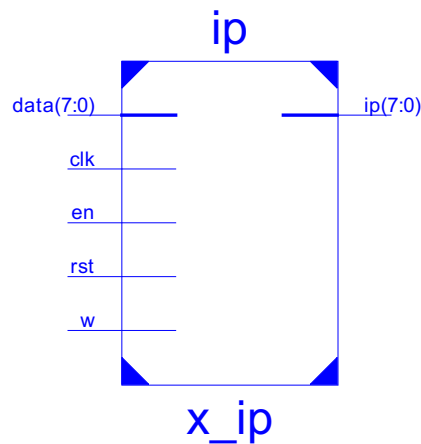


FIGURE 2.17 – Vision macroscopique du bloc x_ip

La vision développée du bloc x_ip est représentée sur la figure 2.18.

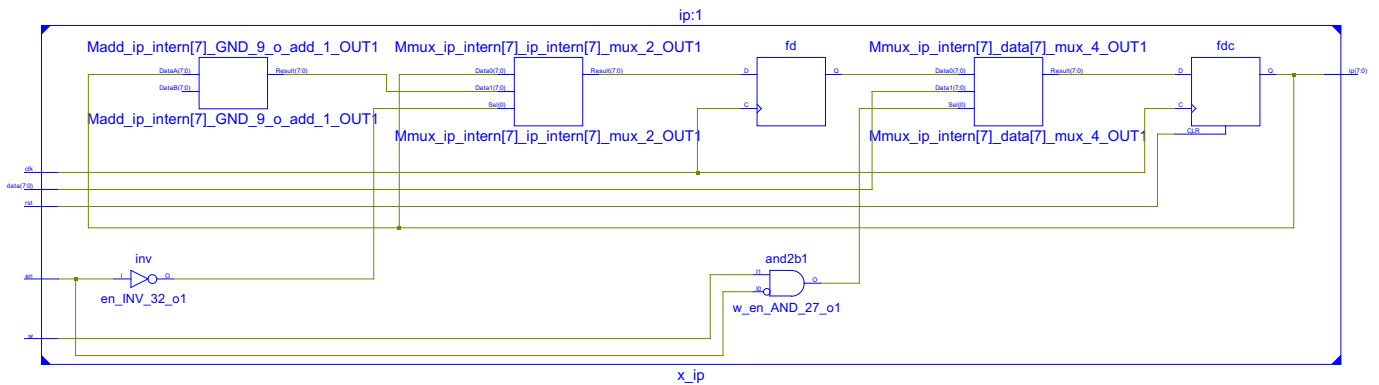


FIGURE 2.18 – Vision développée du bloc x_ip

Les entrées du bloc x_ip sont décrites dans le tableau 2.9.

Nom	Type	Commentaire
data	bus de 8 bits	correspond à la sortie r_data_r du bloc x_r
clk	signal de 1 bit	correspond à l'entrée clk du bloc S
en	signal de 1 bit	correspond à la sortie $finished$ du bloc $finished<7>_imp$
rst	signal de 1 bit	correspond à l'entrée rst du bloc S
w	signal de 1 bit	correspond à la sortie o de primitive ip_w1

TABLE 2.9 – Entrées du bloc x_ip

La sortie du bloc `x_ip` est décrite dans le tableau 2.10.

Nom	Type	Commentaire
ip	bus de 8 bits	correspond à l'entrée <code>address_r</code> du bloc <code>x_smp</code>

TABLE 2.10 – Sortie du bloc `x_ip`

Sur la vision développée du bloc `x_ip`, on peut constater que la sortie `ip` correspond à la valeur stockée dans la bascule `fdc`. La valeur de cette bascule peut être réinitialisée à zéro si le signal `rst` est activé. Ensuite, la bascule est mise à jour selon le processus suivant :

- à chaque cycle d'horloge, la valeur présentée par la bascule `fdc` sert d'entrée au premier bloc à gauche qui joue le rôle d'additionneur ;
- le premier multiplexeur possède deux entrées, la valeur courante de la bascule et la valeur incrémentée par le bloc additionneur. La ligne de sélection de ce multiplexeur correspond à la négation du signal `finished`. Si cette ligne de sélection vaut 0, la première entrée sera retenue, c'est-à-dire la valeur courante. Sinon, la valeur incrémentée sera propagée. ;
- une bascule `fd` joue le rôle de mémoire « tampon » et capture la valeur présentée par le multiplexeur précédent ;
- un second multiplexeur permet de sélectionner une valeur parmi celle stockée dans la bascule tampon et celle présentée par le bus `data`. La ligne de sélection vaut 1 si les signaux `en` et `w` sont activés, 0 sinon ;
- finalement la bascule `fdc` est mise à jour avec la valeur présentée par le second multiplexeur.

Pour résumer, le bloc `x_ip` correspond à une mémoire d'un octet mise à jour à chaque cycle d'horloge selon les conditions suivantes :

- si le signal `rst` est activé, la mémoire est mise à 0 ;
- si le signal `en` est désactivé, la mémoire conserve sa valeur actuelle ;
- si le signal `en` est activé, alors :
 - si le signal `w` vaut 0, la mémoire est incrémentée,
 - sinon, la mémoire prend la valeur présentée sur le bus `data` en entrée.

La sortie du bloc `x_ip` est alors utilisée pour adresser le bloc `x_smp`.

Pour finir l'analyse du bloc `x_ip`, il reste à déterminer comment est calculée la valeur du signal `w` qui correspond à la sortie de la primitive `ip_w1`. Pour cela, il faut donc s'intéresser aux entrées de cette dernière, présentées à la figure 2.19.

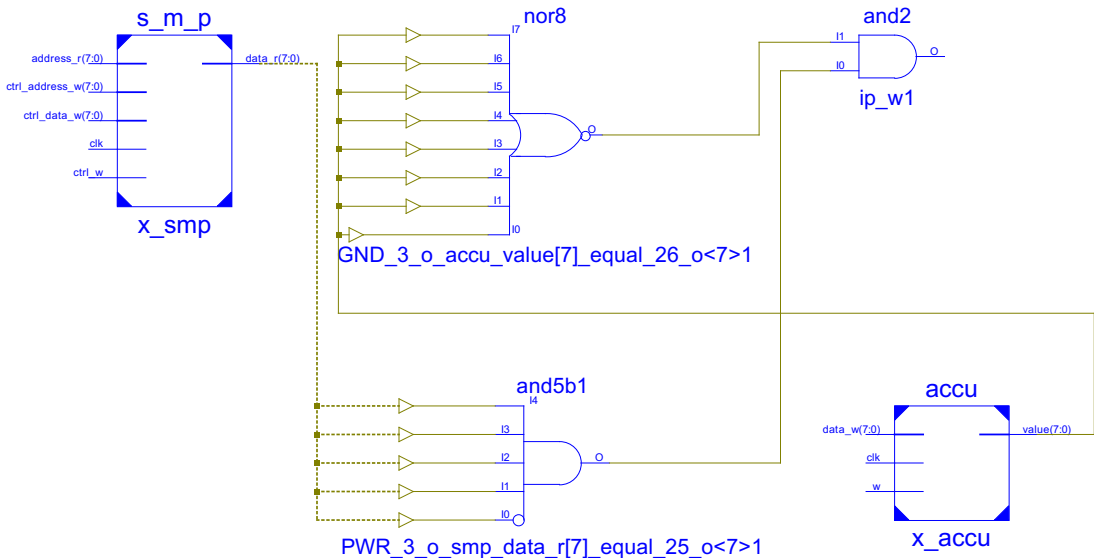


FIGURE 2.19 – Entrées du bloc `ip_w1`

Pour que la sortie de la primitive `ip_w1` soit égale à 1, il faut que ses deux entrées soient également égales à 1.

La première entrée correspond à la sortie d'une primitive `nor8`. Cette dernière prend en entrée la valeur présentée sur le bus `value` par le bloc `accu`. Pour que la sortie d'une primitive `nor8` soit égale à 1, il faut que ses 8 bits en entrée soient égaux à

0, ce qui revient à dire que le bloc **accu** présente l'octet 0 en sortie.

La seconde entrée du bloc **ip_w1** correspond à la sortie d'une primitive **and5b1**. Les entrées de cette dernière sont représentées à la figure 2.20.

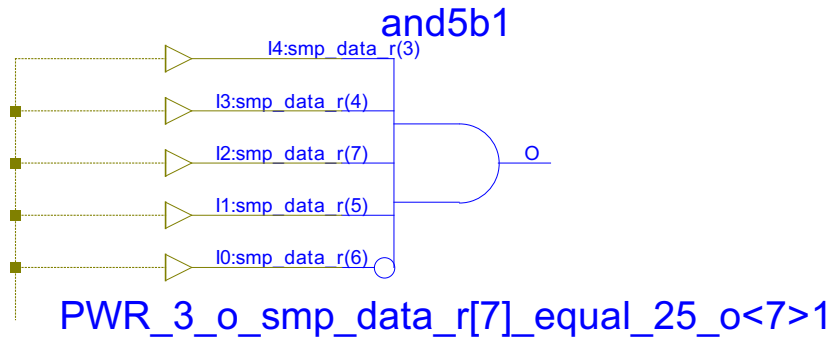


FIGURE 2.20 – Entrées de la primitive **PWR_3_o_smp_data_r[7]_equal_25_o<7>1**

La correspondance des entrées de cette primitive avec les 5 premiers bits du bus **smp_data_r** du bloc **x_smp** est la suivante :

- I4 = **smp_data_r(3)** ;
- I3 = **smp_data_r(4)** ;
- I2 = **smp_data_r(7)** ;
- I1 = **smp_data_r(5)** ;
- I0 = **smp_data_r(6)**.

Pour que la sortie de la primitive **and5b1** soit égale à 1, il faut que toutes ses entrées soient égales à 1, sauf la première qui doit être égale à 0. Le bus **data_r** doit donc vérifier les conditions suivantes :

- **smp_data_r(0)** = indéfini ;
- **smp_data_r(1)** = indéfini ;
- **smp_data_r(2)** = indéfini ;
- **smp_data_r(3)** = 1 ;
- **smp_data_r(4)** = 1 ;
- **smp_data_r(5)** = 1 ;
- **smp_data_r(6)** = 0 ;
- **smp_data_r(7)** = 1.

Pour résumer, l'entrée **w** du bloc **ip** est égale à 1 si :

- la valeur présentée en sortie par le bloc **x_accu** vaut 0 ;
- la valeur présentée en sortie par le bloc **x_smp** est de la forme **10111xxx** où **x** représente 0 ou 1.

2.2.7 Bloc **x_accu**

La vision développée du bloc **x_accu** est représentée sur la figure 2.21.

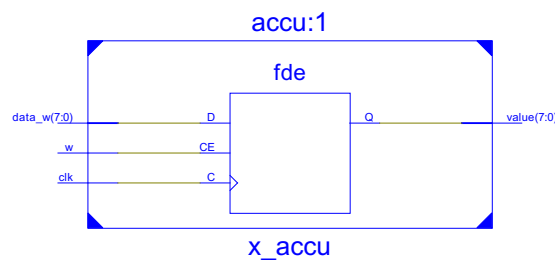


FIGURE 2.21 – Vision développée du bloc **x_accu**

Les entrées du bloc `x_accu` sont décrites dans le tableau 2.11.

Nom	Type	Commentaire
<code>data_w</code>	bus de 8 bits	correspond à la sortie <code>Result</code> du multiplexeur <code>Mmux_accu_data_w1</code>
<code>w</code>	signal de 1 bit	correspond à la sortie <code>accu_w</code> du bloc <code>accu_w_imp</code>
<code>clk</code>	signal de 1 bit	correspond à l'entrée <code>clk</code> du bloc <code>S</code>

TABLE 2.11 – Entrées du bloc `x_accu`

La sortie du bloc `x_accu` est décrite dans le tableau 2.12.

Nom	Type	Commentaire
<code>value</code>	bus de 8 bits	utilisé en entrée par plusieurs blocs

TABLE 2.12 – Sortie du bloc `x_accu`

Le bloc `x_accu` est constitué d'une seule bascule D flip-flop. L'écriture dans cette bascule est activée par l'entrée `w` qui est la sortie du bloc `accu_w_imp`.

La sortie `value` du bloc `x_accu` est utilisée par :

- le bloc `r` (bus `data_w`) ;
- le bloc `x_smd` (bus `address_r` et `data_w`) ;
- le bloc `u` (bus `data_a`) ;
- la primitive `nor8` nommée `GND_3_o_accu_value[7]_equal_26_o<7>1`.

La valeur présente sur le bus `data_w` est conditionnée par le résultat de plusieurs multiplexeurs reliés en série, comme représentés sur la figure 2.22.

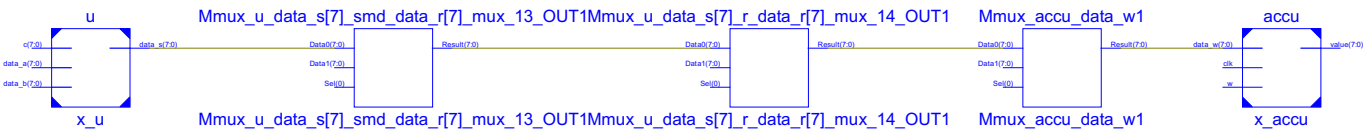


FIGURE 2.22 – Utilisation en série de multiplexeurs

Il est alors nécessaire de comprendre le fonctionnement de chacun de ces multiplexeurs, en commençant par le dernier pour remonter jusqu'au bloc `x_u`.

Multiplexeur `Mmux_accu_data_w1`

Les entrées de ce multiplexeur sont représentées sur la figure 2.23.

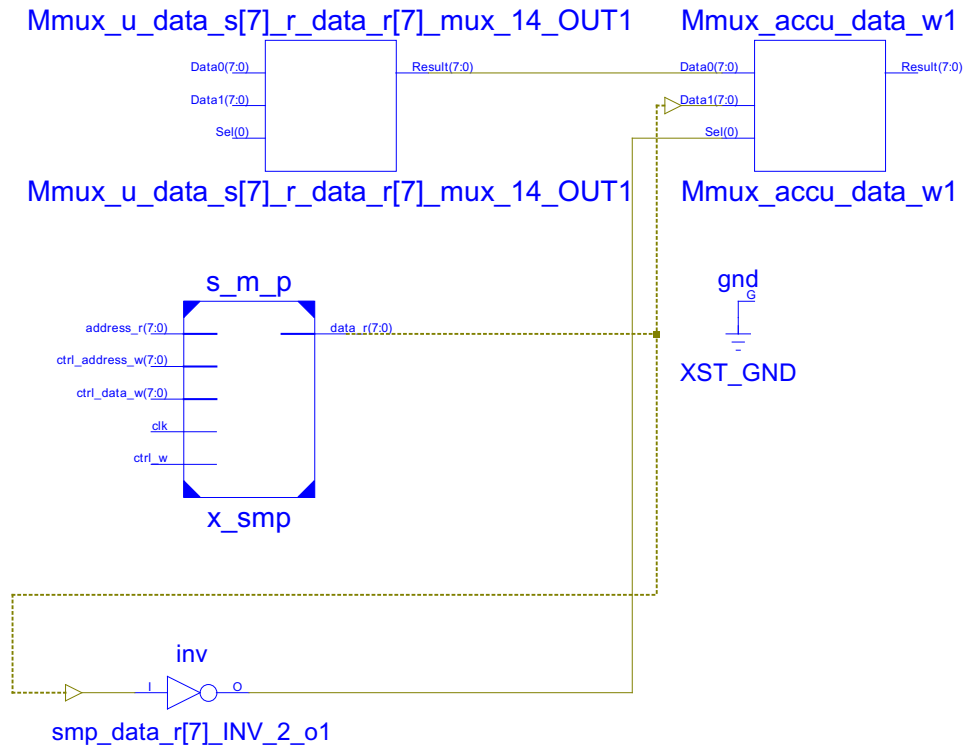


FIGURE 2.23 – Entrées de `Mmux_accu_data_w1`

Pour mieux comprendre la nature de la seconde entrée du multiplexeur, il est nécessaire d'examiner la représentation au niveau bit de cette entrée telle que représentée sur la figure 2.24.

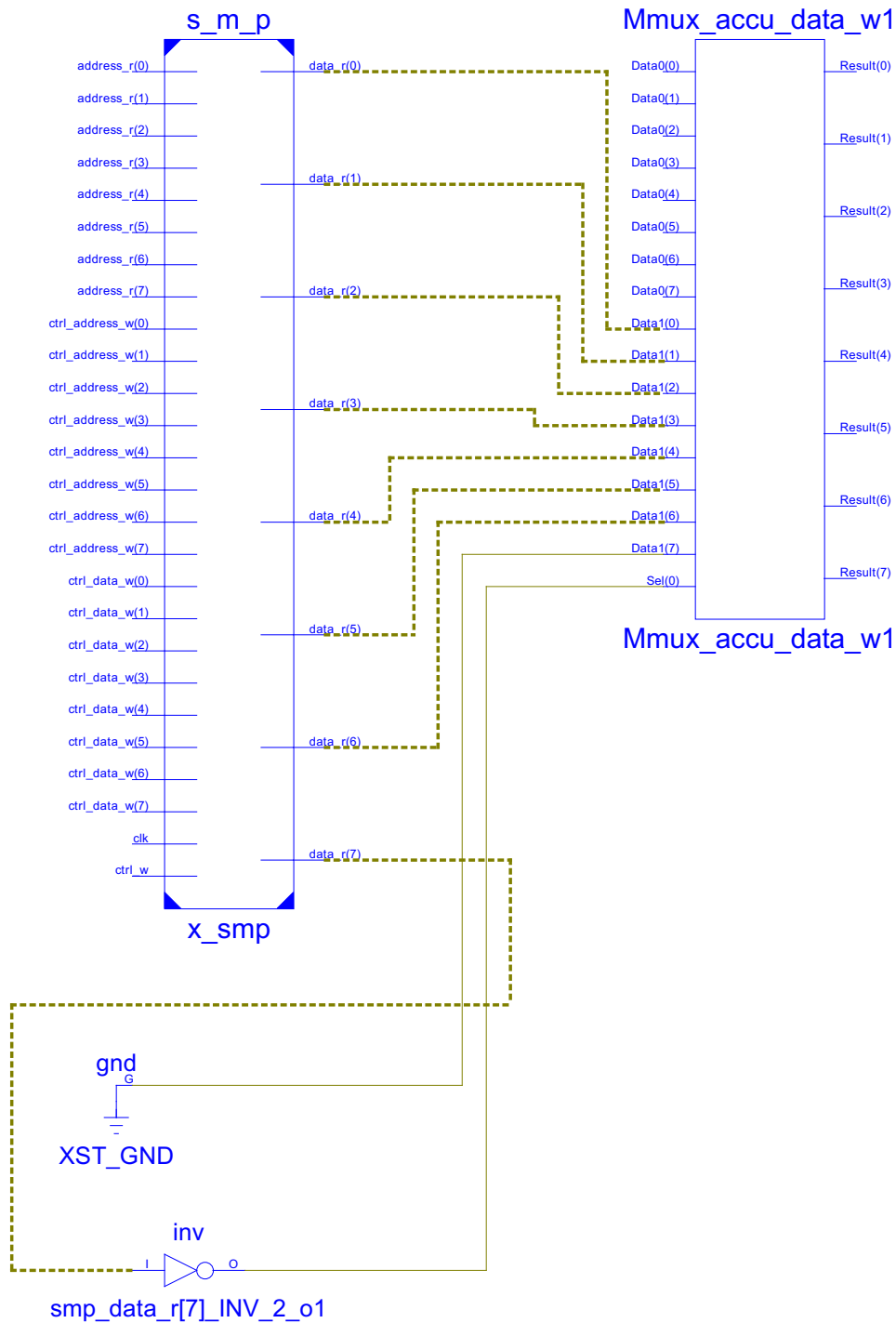


FIGURE 2.24 – Entrées au niveau bit de Mmux_accu_data_w1

La ligne de sélection du multiplexeur correspond au résultat d'un inverseur sur le bit `data_r(7)`. Autrement dit, le multiplexeur sélectionnera la seconde entrée si `smp_data_r(7) = 0`. Le premier bit de cette seconde entrée est relié à la masse et a donc pour valeur 0, les 7 autres bits correspondent aux 7 derniers bits sur le bus `data_r` du bloc `x_smp`.

Multiplexeur Mmux_u_data_s[7]_r_data_r[7]_mux_14_OUT1

Les entrées de ce multiplexeur sont représentées sur la figure 2.25.

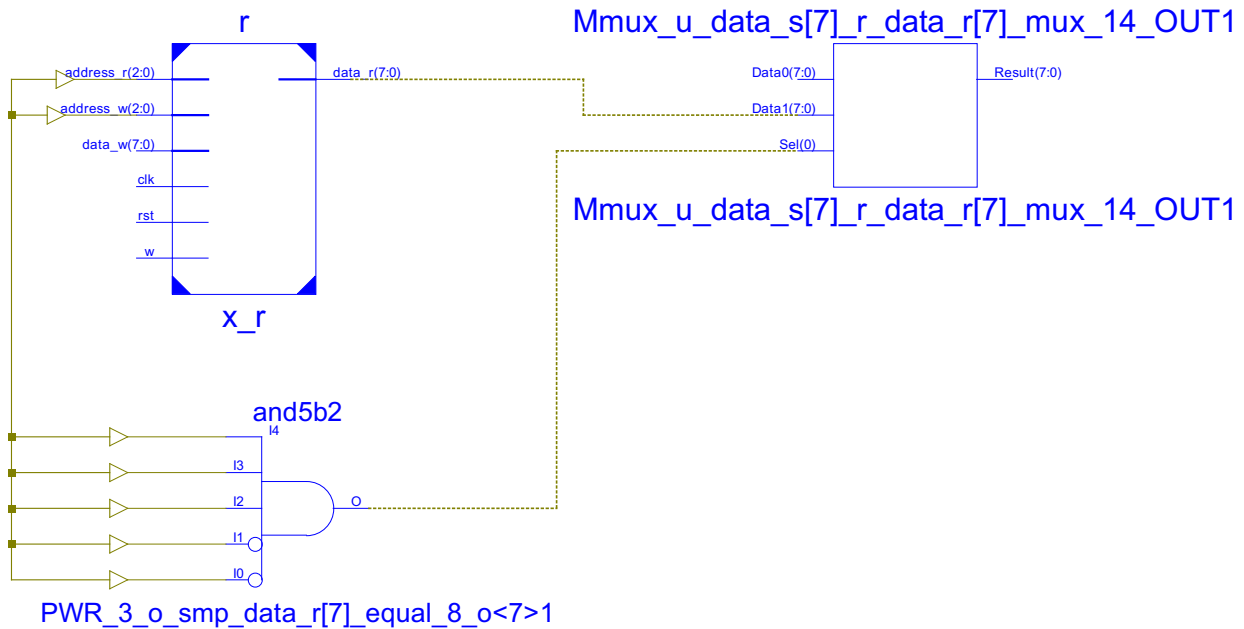


FIGURE 2.25 – Entrées de Mmux_u_data_s[7]_r_data_r[7]_mux_14_OUT1

La sélection du multiplexeur est réalisée à l'aide de la sortie d'une primitive nommée PWR_3_o_smp_data_r[7]_equal_8_o<7>1 qui correspond à une porte and5b2. Les entrées de cette primitive sont représentées à la figure 2.26.

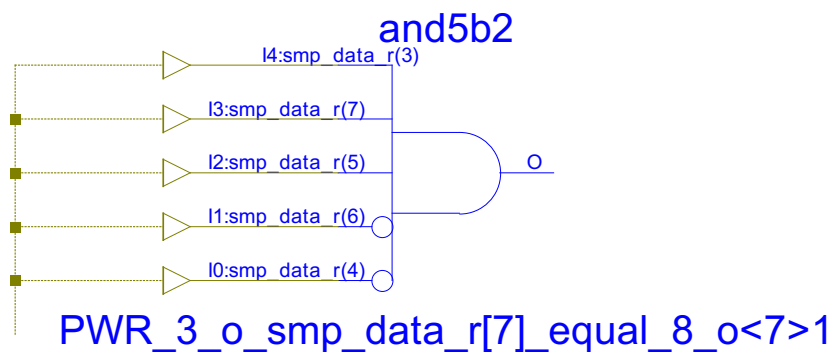


FIGURE 2.26 – Entrées de la primitive PWR_3_o_smp_data_r[7]_equal_8_o<7>1

La correspondance des entrées de la primitive avec les 5 premiers bits du bus smp_data_r du bloc x_smp est la suivante :

- I4 = smp_data_r(3);
- I3 = smp_data_r(7);
- I2 = smp_data_r(5);
- I1 = smp_data_r(6);
- I0 = smp_data_r(4).

Pour que la sortie de la primitive soit égale à 1, les conditions ci-dessous doivent alors être vérifiées :

- smp_data_r(0) = indéfini;
- smp_data_r(1) = indéfini;
- smp_data_r(2) = indéfini;
- smp_data_r(3) = 1;
- smp_data_r(4) = 0;
- smp_data_r(5) = 1;
- smp_data_r(6) = 0;
- smp_data_r(7) = 1.

Autrement dit, le multiplexeur va sélectionner la sortie du bloc x_r si la valeur présentée en sortie par le bloc x_smp est de la forme 10101xxx où x représente 0 ou 1. Les 3 derniers bits lus de x_smp servent à sélectionner le registre à lire dans le bloc x_r .

Multiplexeur $Mmux_u_data_s[7]_{smd_data_r[7]_{mux_13_OUT1}}$

Les entrées de ce multiplexeur sont représentées sur la figure 2.27.

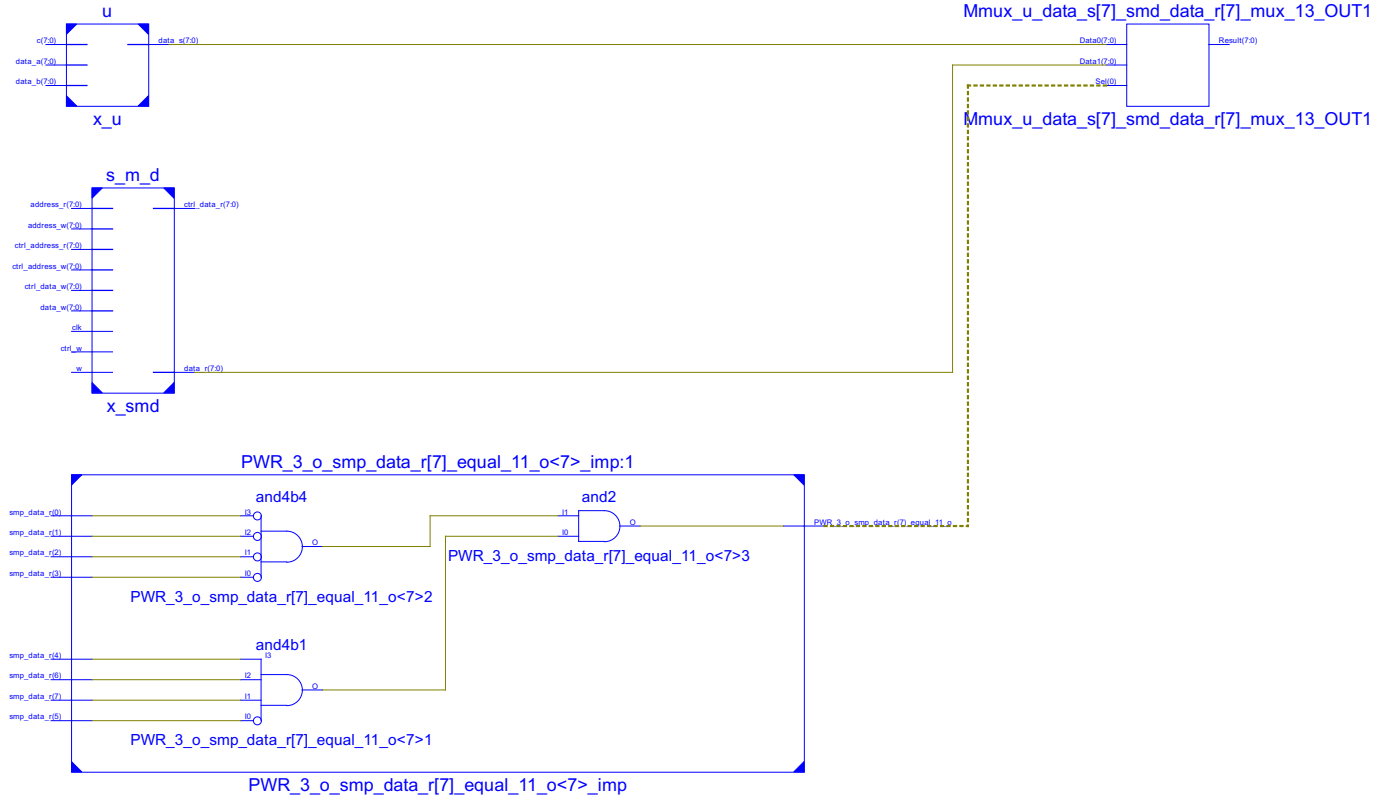


FIGURE 2.27 – Entrées de $Mmux_u_data_s[7]_{smd_data_r[7]_{mux_13_OUT1}}$

La sélection du multiplexeur est réalisée à l'aide de la sortie du bloc $PWR_3_o_smp_data_r[7]_{equal_11_o<7>}$ composé de primitives $and4b4$, $and4b1$ et $and2$. Ce bloc a pour entrée la valeur présente sur le bus $data_r$ du bloc x_smp . Les entrées de ce bloc sont représentées à la figure 2.28.

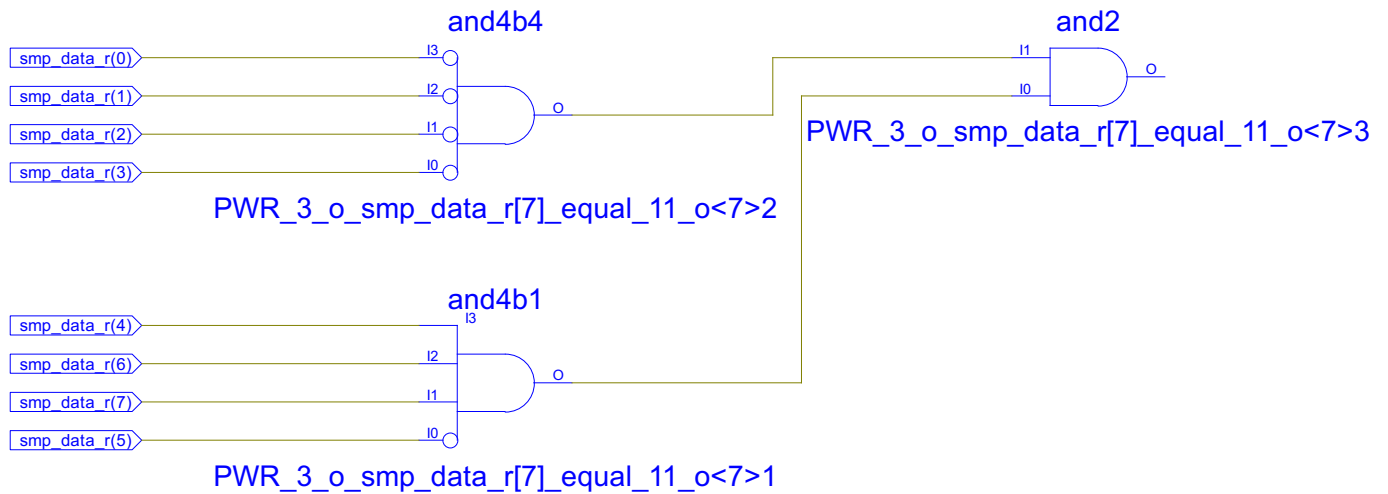


FIGURE 2.28 – Entrées du bloc $PWR_3_o_smp_data_r[7]_{equal_11_o<7>}$

La correspondance des bits de `data_r` sur les entrées de la primitive `and4b4` est la suivante :

- I3 = `smp_data_r(0)` ;
- I2 = `smp_data_r(1)` ;
- I1 = `smp_data_r(2)` ;
- I0 = `smp_data_r(3)`.

La correspondance des bits de `data_r` sur les entrées de la primitive `and4b1` est la suivante :

- I3 = `smp_data_r(4)` ;
- I2 = `smp_data_r(6)` ;
- I1 = `smp_data_r(7)` ;
- I0 = `smp_data_r(5)`.

Pour que la primitive `and2` retourne 1, il faut également que les deux primitives `and4b4` et `and4b1` retournent 1, ce qui se traduit par les conditions suivantes sur `data_r` :

- `smp_data_r(0)` = 0 ;
- `smp_data_r(1)` = 0 ;
- `smp_data_r(2)` = 0 ;
- `smp_data_r(3)` = 0 ;
- `smp_data_r(4)` = 1 ;
- `smp_data_r(5)` = 0 ;
- `smp_data_r(6)` = 1 ;
- `smp_data_r(7)` = 1.

Autrement dit, le multiplexeur va sélectionner la sortie du bloc `x_smd` si la valeur présentée en sortie par le bloc `x_smp` est égale à 11010000. Dans les autres cas, le multiplexeur va sélectionner la sortie du bloc `x_u`.

2.2.8 Bloc `accu_w_imp`

La vision développée du bloc `accu_w_imp` est représentée sur la figure 2.29.

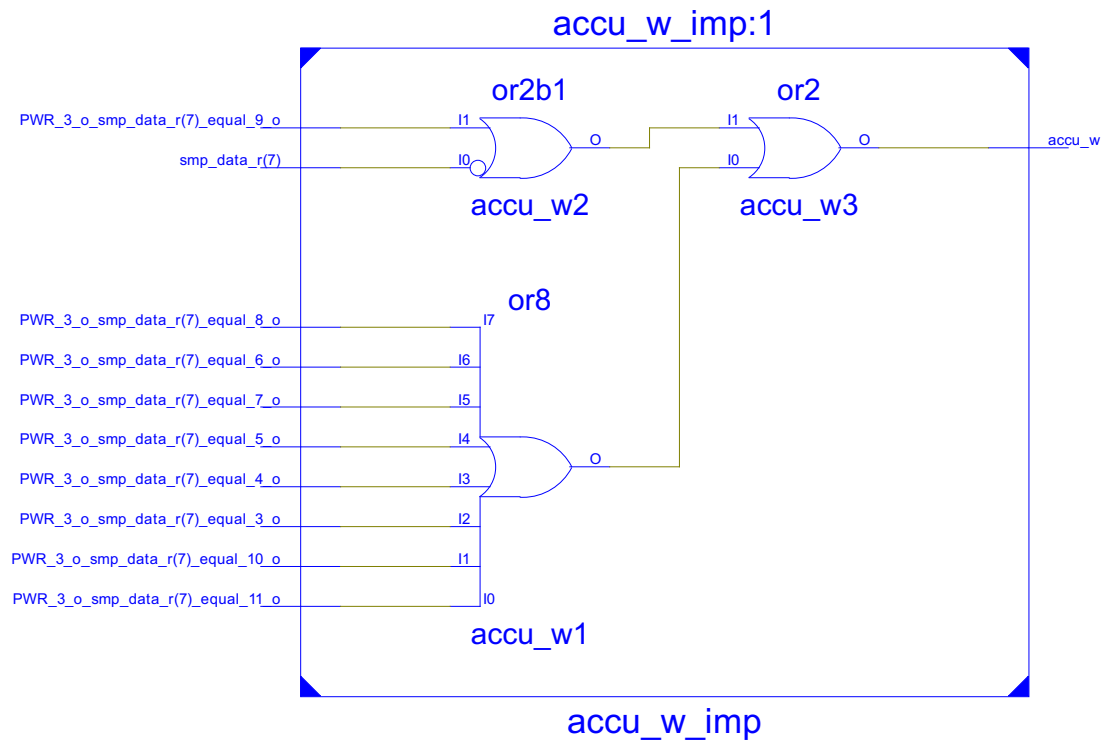


FIGURE 2.29 – Vision développée du bloc `accu_w_imp`

Les entrées du bloc `accu_w_imp` sont décrites dans le tableau 2.13.

Nom	Type	Commentaire
<code>PWR_3_o_smp_data_r(7)_equal_9_o</code>	signal de 1 bit	sortie de <code>PWR_3_o_smp_data_r[7]_equal_9_o<7>1</code>
<code>smp_data_r(7)</code>	signal de 1 bit	correspond au premier bit de la sortie <code>data_r</code> du bloc <code>x_smp</code>
<code>PWR_3_o_smp_data_r(7)_equal_8_o</code>	signal de 1 bit	sortie de <code>PWR_3_o_smp_data_r[7]_equal_8_o<7>1</code>
<code>PWR_3_o_smp_data_r(7)_equal_6_o</code>	signal de 1 bit	sortie de <code>PWR_3_o_smp_data_r[7]_equal_6_o<7>1</code>
<code>PWR_3_o_smp_data_r(7)_equal_7_o</code>	signal de 1 bit	sortie de <code>PWR_3_o_smp_data_r[7]_equal_7_o<7>1</code>
<code>PWR_3_o_smp_data_r(7)_equal_5_o</code>	signal de 1 bit	sortie de <code>PWR_3_o_smp_data_r[7]_equal_5_o<7>1</code>
<code>PWR_3_o_smp_data_r(7)_equal_4_o</code>	signal de 1 bit	sortie de <code>PWR_3_o_smp_data_r[7]_equal_4_o<7>1</code>
<code>PWR_3_o_smp_data_r(7)_equal_3_o</code>	signal de 1 bit	sortie de <code>PWR_3_o_smp_data_r[7]_equal_3_o<7>1</code>
<code>PWR_3_o_smp_data_r(7)_equal_10_o</code>	signal de 1 bit	sortie de <code>PWR_3_o_smp_data_r[7]_equal_10_o<7>1</code>
<code>PWR_3_o_smp_data_r(7)_equal_11_o</code>	signal de 1 bit	sortie de <code>PWR_3_o_smp_data_r[7]_equal_11_o<7>1</code>

TABLE 2.13 – Entrées du bloc `accu_w_imp`

La sortie du bloc `accu_w_imp` est décrite dans le tableau 2.14.

Nom	Type	Commentaire
<code>accu_w</code>	signal de 1 bit	correspond à l'entrée <code>w</code> du bloc <code>x_accu</code>

TABLE 2.14 – Sortie du bloc `accu_w_imp`

Le bloc `accu_w_imp` contrôle l'écriture dans le bloc `x_accu`. Il est constitué de trois primitives de type `or`. La sortie `accu_w` est égale à 1 si une des conditions ci-dessous est remplie :

- la valeur de `smp_data_r(7)` est 0 ;
- n'importe quelle autre entrée est égale à 1.

Les entrées de type `PWR_3_o_smp_data_r(7)_equal_X_o` correspondent à la sortie d'une primitive ou d'un bloc effectuant des tests sur la valeur présentée par le bloc `x_smp` sur sa sortie `data_r`. Certaines de ces primitives ont déjà été étudiées dans les chapitres précédents, notamment `PWR_3_o_smp_data_r[7]_equal_8_o<7>` au chapitre 2.2.7 et `PWR_3_o_smp_data_r[7]_equal_11_o<7>` au chapitre 2.2.7.

Une étude similaire des autres blocs de ce type permet de déterminer l'ensemble des valeurs de `data_r` qui activent `accu_w`. Le résultat obtenu est référencé dans le tableau 2.15.

Entrée	Condition sur <code>data_r</code>
<code>PWR_3_o_smp_data_r(7)_equal_9_o</code>	11100xxx
<code>smp_data_r(7)</code>	0xxxxxxx
<code>PWR_3_o_smp_data_r(7)_equal_8_o</code>	10101xxx
<code>PWR_3_o_smp_data_r(7)_equal_6_o</code>	10011xxx
<code>PWR_3_o_smp_data_r(7)_equal_7_o</code>	10100xxx
<code>PWR_3_o_smp_data_r(7)_equal_5_o</code>	10010xxx
<code>PWR_3_o_smp_data_r(7)_equal_4_o</code>	10001xxx
<code>PWR_3_o_smp_data_r(7)_equal_3_o</code>	10000xxx
<code>PWR_3_o_smp_data_r(7)_equal_10_o</code>	11011xxx
<code>PWR_3_o_smp_data_r(7)_equal_11_o</code>	11010000

TABLE 2.15 – Conditions d'activation de `accu_w`

2.2.9 Bloc `x_u`

La vision macroscopique du bloc `x_u` est représentée sur la figure 2.30.

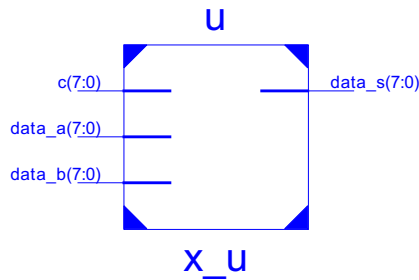


FIGURE 2.30 – Vision macroscopique du bloc x_u

La vision développée du bloc x_u est représentée sur la figure 2.31.

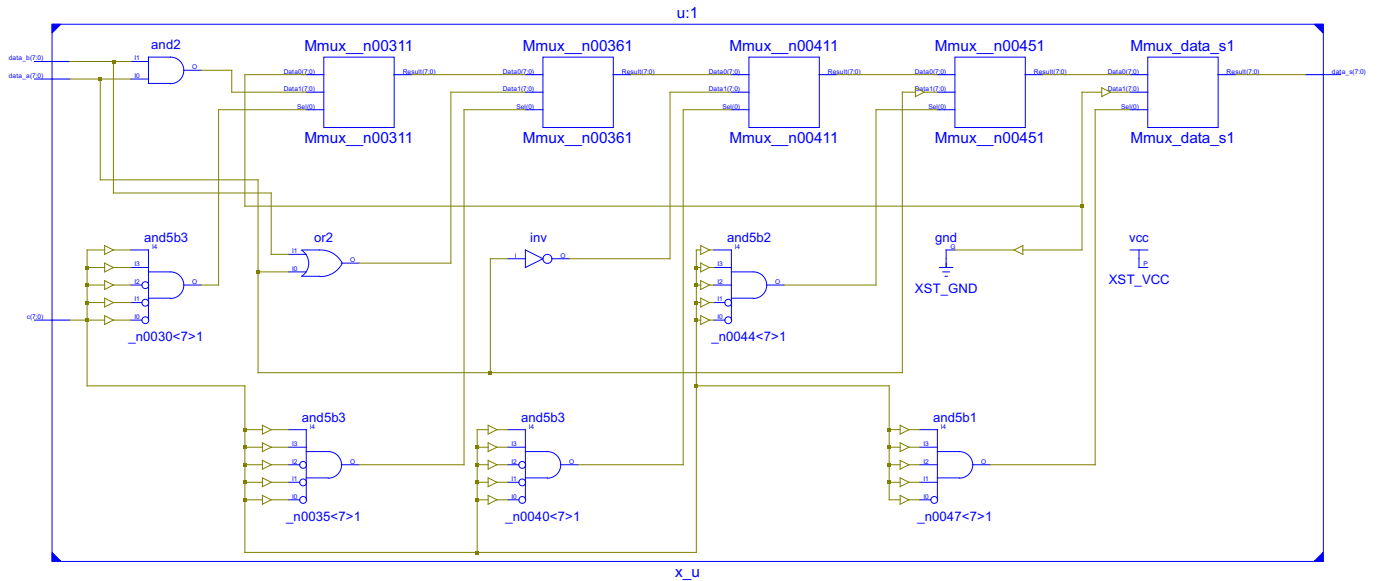


FIGURE 2.31 – Vision développée du bloc x_u

Les entrées du bloc x_u sont décrites dans le tableau 2.16.

Nom	Type	Commentaire
$data_b$	bus de 8 bits	correspond à la sortie $data_r$ du bloc x_r
$data_a$	bus de 8 bits	correspond à la sortie $value$ du bloc x_{accu}
c	bus de 8 bits	correspond à la sortie $data_r$ du bloc x_{sm}

TABLE 2.16 – Entrées du bloc x_u

La sortie du bloc x_u est décrite dans le tableau 2.17.

Nom	Type	Commentaire
$data_s$	bus de 8 bits	entrée du multiplexeur $Mmux_u_data_s[7]_{smd_data_r[7]_{mux_13_OUT1}}$

TABLE 2.17 – Sortie du bloc x_u

Le bloc x_u possède deux bus de données en entrée, qui correspondent aux sorties des blocs x_r et x_{accu} . Il possède également une entrée c dont la valeur est la sortie du bloc x_{sm} .

La sortie du bloc x_u constitue l'entrée du premier multiplexeur qui permet de mettre à jour la valeur stockée dans le bloc x_{accu} (voir section 2.2.7).

La vision développée du bloc permet de constater la présence de cinq multiplexeurs reliés en série. La sélection de ces multiplexeurs est effectué par des portes logiques de type and qui prennent en entrée des valeurs présentes sur le bus c .

Multiplexeur Mmux_nn00311

Mmux_nn00311 est le premier de la série de multiplexeurs qui constituent le bloc x_u. Ses entrées sont représentées à la figure 2.32.

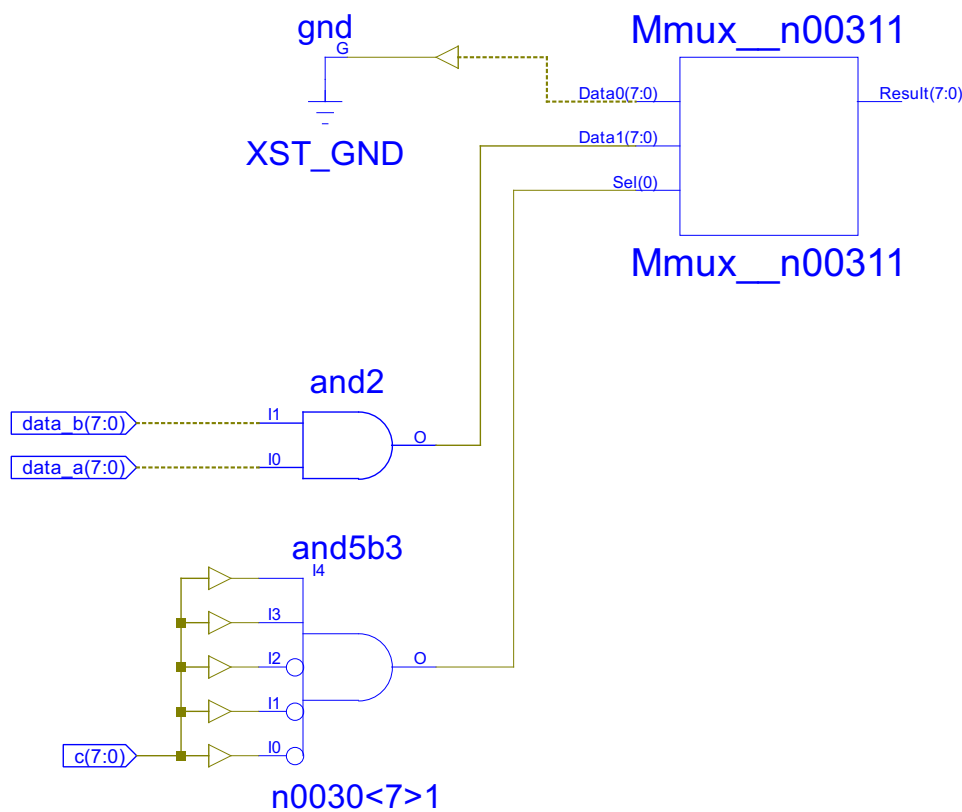


FIGURE 2.32 – Entrées du multiplexeur Mmux_nn00311

La première entrée du multiplexeur est reliée à la masse et vaut donc 0. L'autre entrée est égale au résultat de la porte logique and2 appliquée sur les valeurs présentées par x_accu et x_r.

La sélection du multiplexeur est effectuée selon la valeur de la sortie de la primitive _n0030<7>1 qui est une porte logique and5b3 dont les entrées sont représentées à la figure 2.33.

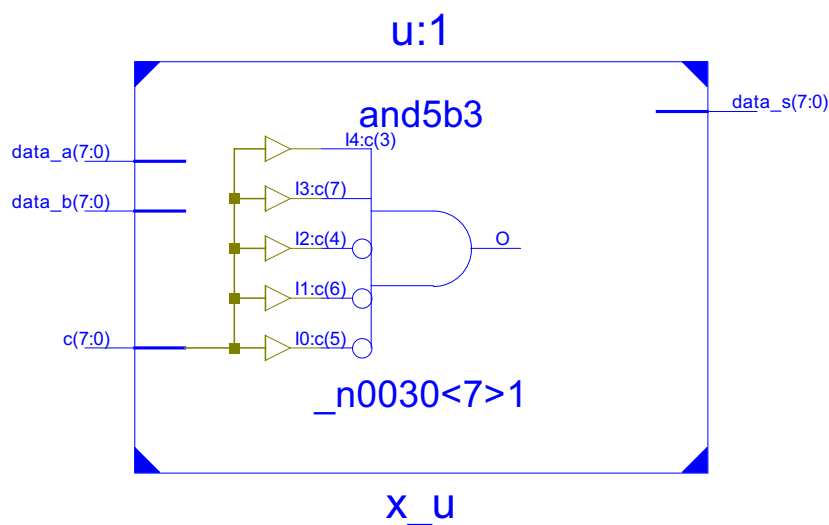


FIGURE 2.33 – Entrées de _n0030<7>1

La correspondance des entrées de la primitive avec les 5 premiers bits du bus `smp_data_r` du bloc `smp` est la suivante :

- `I4 = c(3) = smp_data_r(3);`
- `I3 = c(7) = smp_data_r(7);`
- `I2 = c(4) = smp_data_r(4);`
- `I1 = c(6) = smp_data_r(6);`
- `I0 = c(5) = smp_data_r(5).`

Pour que la sortie de la primitive `_n0030<7>1` soit égale à 1, il faut que ses trois premières entrées soient égales à 0 et les deux dernières à 1, ce qui se traduit par les conditions ci-dessous sur `smp_data_r` :

- `smp_data_r(0) = indéfini;`
- `smp_data_r(1) = indéfini;`
- `smp_data_r(2) = indéfini;`
- `smp_data_r(3) = 1;`
- `smp_data_r(4) = 0;`
- `smp_data_r(5) = 0;`
- `smp_data_r(6) = 0;`
- `smp_data_r(7) = 1.`

Si la valeur présentée par le bloc `x_smp` est de la forme `10001xxx`, alors le multiplexeur transmettra le résultat d'un « et » logique entre les valeurs présentées par les blocs `x_accu` et `x_r`. Dans le cas contraire, le multiplexeur transmettra la valeur 0 sur sa sortie.

Multiplexeur `Mmux_nn00361`

`Mmux_nn00361` est le second de la série de multiplexeurs qui constituent le bloc `x_u`. Ses entrées sont représentées à la figure 2.34.

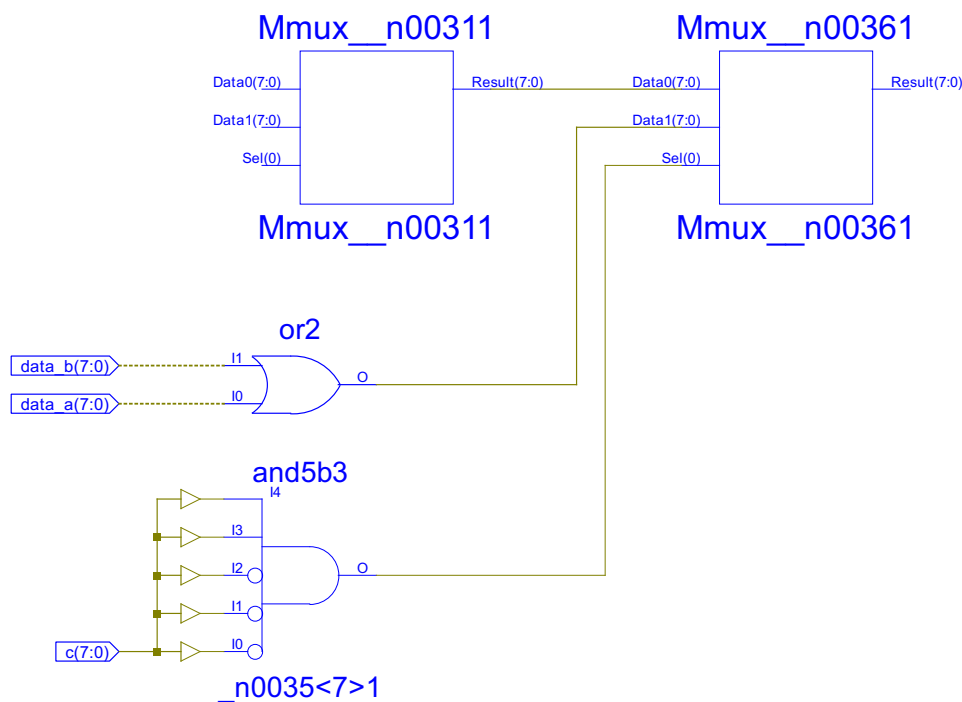


FIGURE 2.34 – Entrées du multiplexeur `Mmux_nn00361`

La première entrée du multiplexeur correspond à la sortie du multiplexeur précédent, `Mmux_nn00311`. L'autre entrée est égale au résultat de la porte logique `or2` appliquée sur les valeurs présentées par `x_accu` et `x_r`.

La sélection du multiplexeur est effectuée selon la valeur de la sortie de la primitive `_n0035<7>1` qui est une porte logique `and5b3` dont les entrées sont représentées à la figure 2.35.

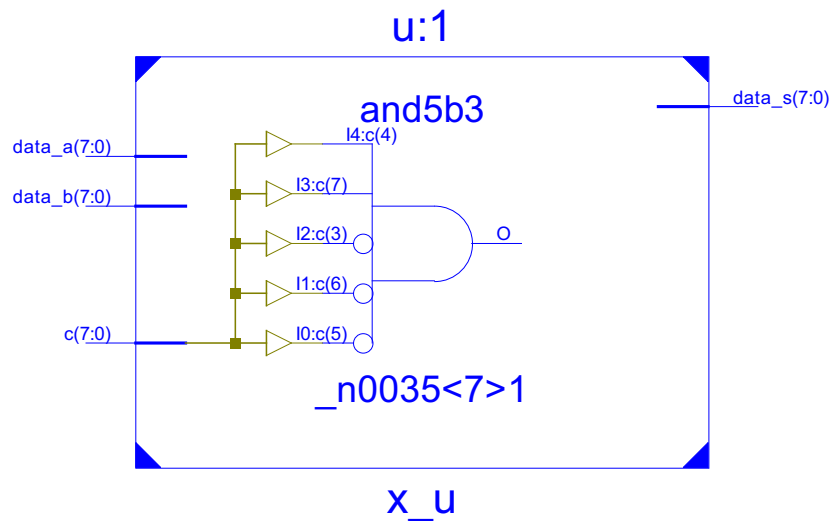


FIGURE 2.35 – Entrées de `_n0035<7>1`

La correspondance des entrées de la primitive avec les 5 premiers bits du bus `smp_data_r` du bloc `smp` est la suivante :

- `I4 = c(4) = smp_data_r(4)`;
- `I3 = c(7) = smp_data_r(7)`;
- `I2 = c(3) = smp_data_r(3)`;
- `I1 = c(6) = smp_data_r(6)`;
- `I0 = c(5) = smp_data_r(5)`.

Pour que la sortie de la primitive `_n0035<7>1` soit égale à 1, il faut que ses trois premières entrées soient égales à 0 et les deux dernières à 1, ce qui se traduit par les conditions ci-dessous sur `smp_data_r` :

- `smp_data_r(0) = indéfini`;
- `smp_data_r(1) = indéfini`;
- `smp_data_r(2) = indéfini`;
- `smp_data_r(3) = 0`;
- `smp_data_r(4) = 1`;
- `smp_data_r(5) = 0`;
- `smp_data_r(6) = 0`;
- `smp_data_r(7) = 1`.

Si la valeur présentée par le bloc `x_smp` est de la forme `10010xxx`, alors le multiplexeur transmettra le résultat d'un « ou » logique entre les valeurs présentées par les blocs `x_accu` et `x_r`. Dans le cas contraire, le multiplexeur transmettra sur sa sortie la valeur présentée par le multiplexeur précédent.

Multiplexeur `Mmux_nn00411`

`Mmux_nn00411` est le troisième de la série de multiplexeurs qui constituent le bloc `x_u`. Ses entrées sont représentées à la figure 2.36.

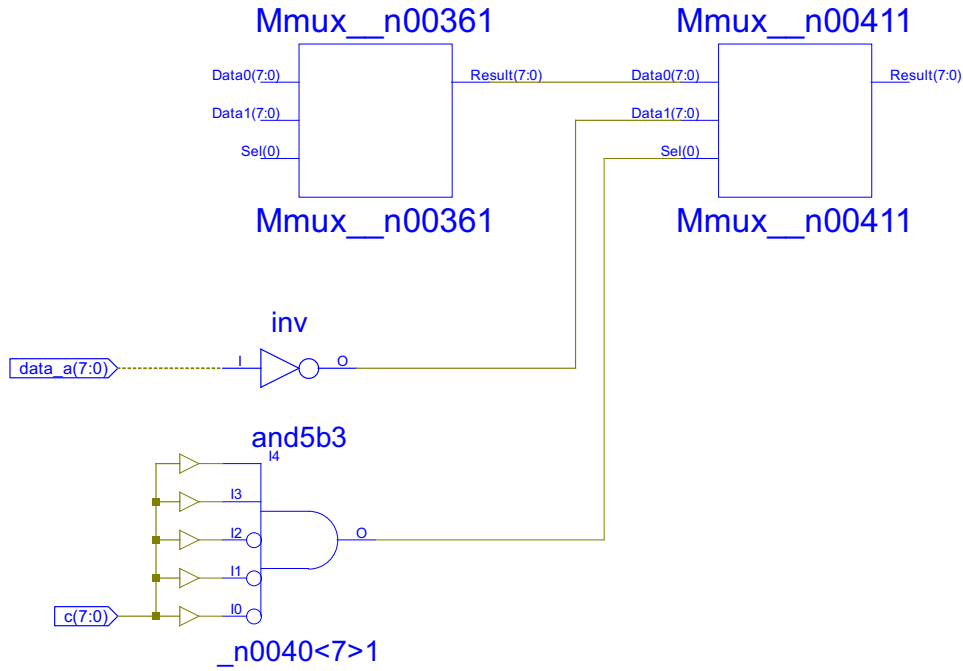


FIGURE 2.36 – Entrées du multiplexeur Mmux_nn00411

La première entrée du multiplexeur correspond à la sortie du multiplexeur précédent, Mmux_nn00361. L'autre entrée est égale au résultat de la porte logique `inv` appliquée sur la valeur présentée par `x_accu`.

La sélection du multiplexeur est effectuée selon la valeur de la sortie de la primitive `_n0040<7>1` qui est une porte logique `and5b3` dont les entrées sont représentées à la figure 2.37.

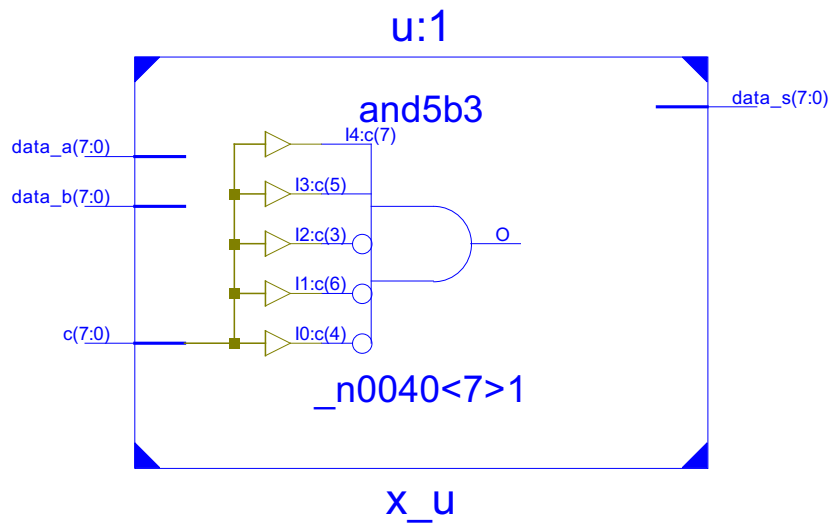


FIGURE 2.37 – Entrées de `_n0040<7>1`

La correspondance des entrées de la primitive avec les 5 premiers bits du bus `smp_data_r` du bloc `smp` est la suivante :

- `I4 = c(7) = smp_data_r(7)` ;
- `I3 = c(5) = smp_data_r(5)` ;
- `I2 = c(3) = smp_data_r(3)` ;
- `I1 = c(6) = smp_data_r(6)` ;
- `I0 = c(4) = smp_data_r(4)`.

Pour que la sortie de la primitive `_n0040<7>1` soit égale à 1, il faut que ses trois premières entrées soient égales à 0 et les deux dernières à 1, ce qui se traduit par les conditions ci-dessous sur `smp_data_r` :

- smp_data_r(0) = indéfini;
- smp_data_r(1) = indéfini;
- smp_data_r(2) = indéfini;
- smp_data_r(3) = 0;
- smp_data_r(4) = 0;
- smp_data_r(5) = 1;
- smp_data_r(6) = 0;
- smp_data_r(7) = 1.

Si la valeur présentée par le bloc x_smp est de la forme 10100xxx, alors le multiplexeur transmettra le résultat d'une inversion logique sur la valeur présentée par le bloc x_accu. Dans le cas contraire, le multiplexeur transmettra sur sa sortie la valeur présentée par le multiplexeur précédent.

Multiplexeur Mmux_nn00451

Mmux_nn00451 est le quatrième de la série de multiplexeurs qui constituent le bloc x_u. Ses entrées sont représentées à la figure 2.38.

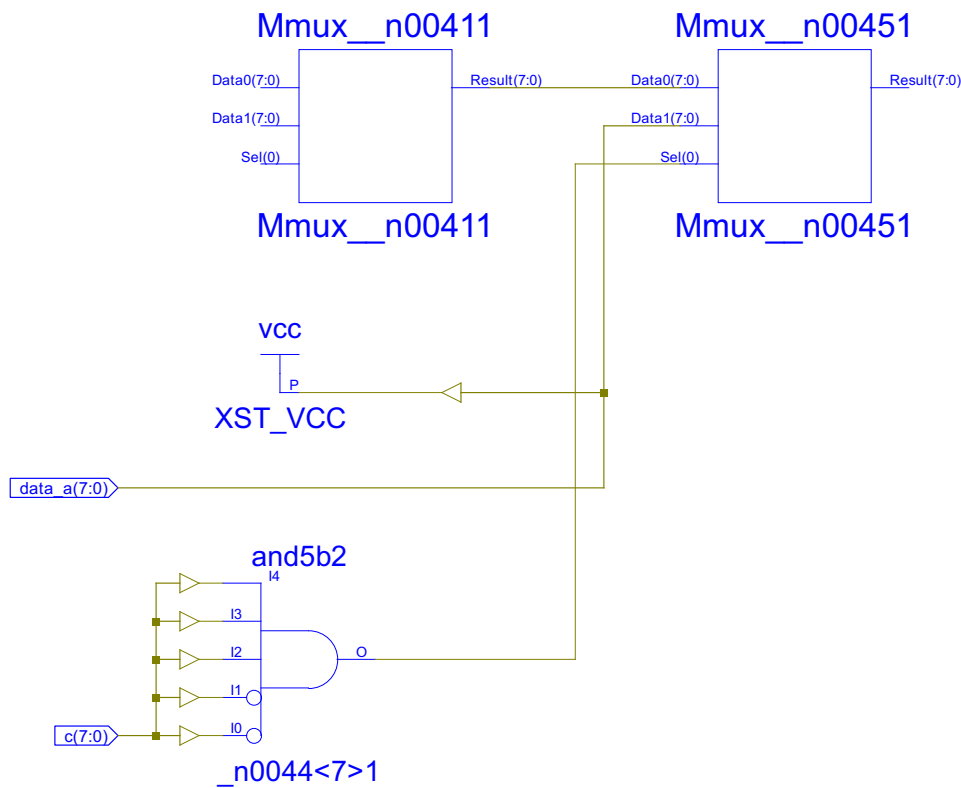


FIGURE 2.38 – Entrées du multiplexeur Mmux_nn00451

La première entrée du multiplexeur correspond à la sortie du multiplexeur précédent, Mmux_nn00411. La seconde entrée semble égale à la valeur présente sur le bus data_a, c'est-à-dire la valeur en sortie du bloc x_accu. En réalité, une visualisation en mode bit présentée à la figure 2.39 permet de comprendre la nature précise de cette seconde entrée. L'entrée Data1(7) du multiplexeur est reliée à la primitive VCC qui représente un courant continu, sa valeur est donc égale à 1. Les bits 0 à 6 de Data1 correspondent aux bits 0 à 6 de data_a. En d'autres termes, le multiplexeur prend la valeur du bloc x_accu en forçant le bit le plus significatif à 1.

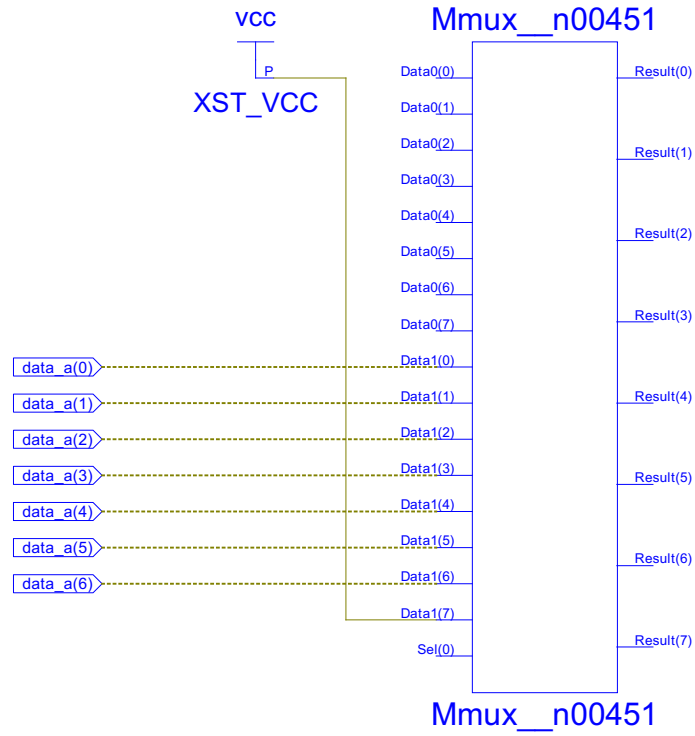


FIGURE 2.39 – Visualisation niveau bit de Mmux_nn00451

La sélection du multiplexeur est effectuée selon la valeur de la sortie de la primitive _n0044<7>1 qui est une porte logique and5b2 dont les entrées sont représentées à la figure 2.40.

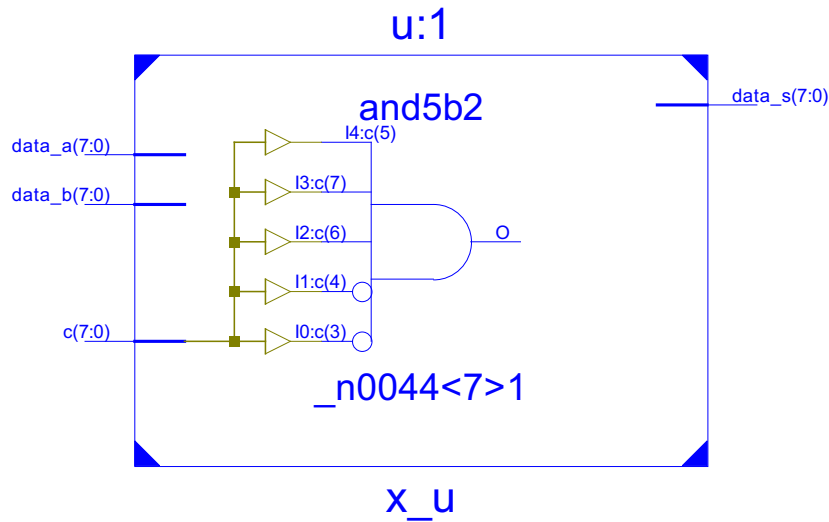


FIGURE 2.40 – Entrées de _n0044<7>1

La correspondance des entrées de la primitive avec les 5 premiers bits du bus smp_data_r du bloc smp est la suivante :

- I4 = c(5) = smp_data_r(7) ;
- I3 = c(7) = smp_data_r(7) ;
- I2 = c(6) = smp_data_r(6) ;
- I1 = c(4) = smp_data_r(4) ;
- I0 = c(3) = smp_data_r(3).

Pour que la sortie de la primitive _n0044<7>1 soit égale à 1, il faut que ses deux premières entrées soient égales à 0 et les trois dernières à 1, ce qui se traduit par les conditions ci-dessous sur smp_data_r :

- smp_data_r(0) = indéfini;
- smp_data_r(1) = indéfini;
- smp_data_r(2) = indéfini;
- smp_data_r(3) = 0;
- smp_data_r(4) = 0;
- smp_data_r(5) = 1;
- smp_data_r(6) = 1;
- smp_data_r(7) = 1.

Si la valeur présentée par le bloc `x_smp` est de la forme `11100xxx`, alors le multiplexeur transmettra la valeur présentée par le bloc `x_accu` en forçant à 1 le bit le plus significatif. Dans le cas contraire, le multiplexeur transmettra sur sa sortie la valeur présentée par le multiplexeur précédent.

Multiplexeur Mmux_data_s1

`Mmux_data_s1` est le cinquième et dernier multiplexeur du bloc `x_u`. Ses entrées sont représentées à la figure 2.41.

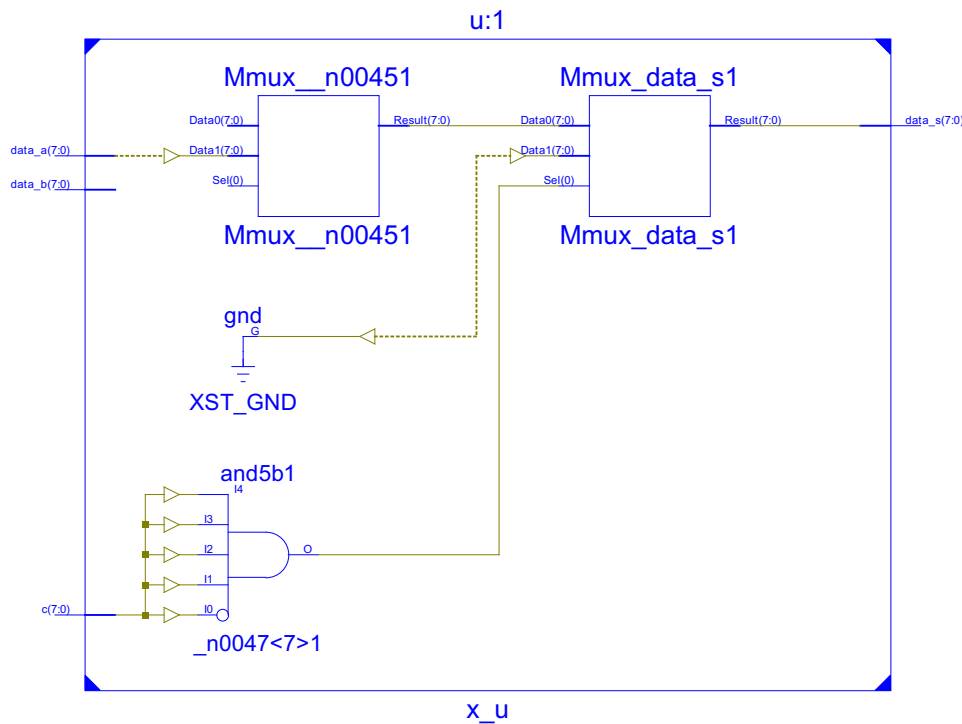


FIGURE 2.41 – Entrées du multiplexeur `Mmux_data_s1`

La première entrée du multiplexeur correspond à la sortie du multiplexeur précédent, `Mmux_nn00451`. Pour comprendre la nature de la seconde entrée du multiplexeur, une visualisation en mode bit se révèle nécessaire. Celle-ci est présentée à la figure 2.42. On remarque que le bit `Data1(0)` est relié à la masse et vaut donc toujours 0. Les bits 1 à 7 de `Data1` correspondent aux bits 0 à 6 de `data_a`, ce qui revient à effectuer un décalage à gauche d'un bit sur la valeur du bloc `x_accu`.

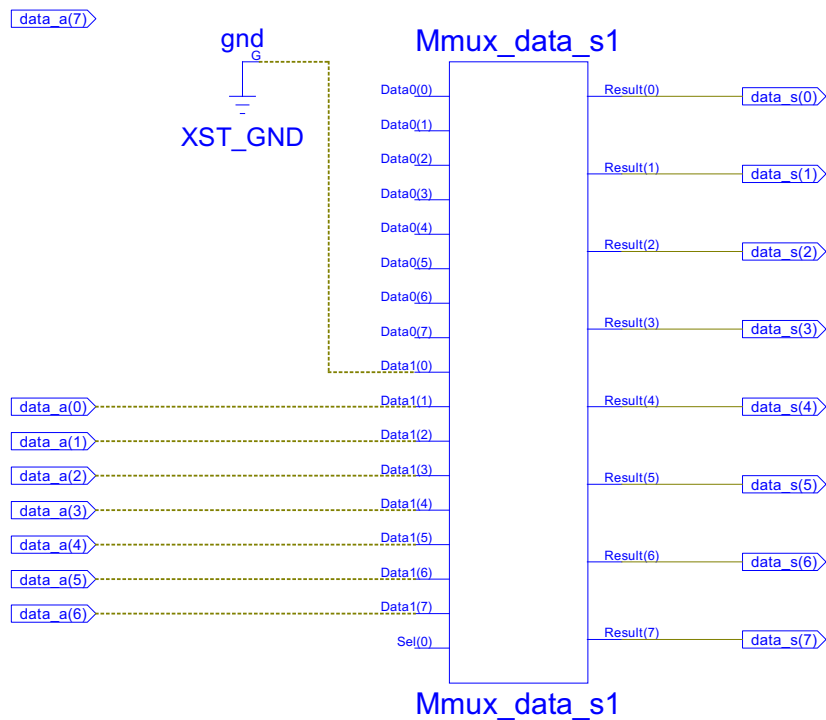


FIGURE 2.42 – Visualisation niveau bit de Mmux_data_s1

La sélection du multiplexeur est effectuée selon la valeur de la sortie de la primitive `_n0047<7>1` qui est une porte logique `and5b1` dont les entrées sont représentées à la figure 2.43.

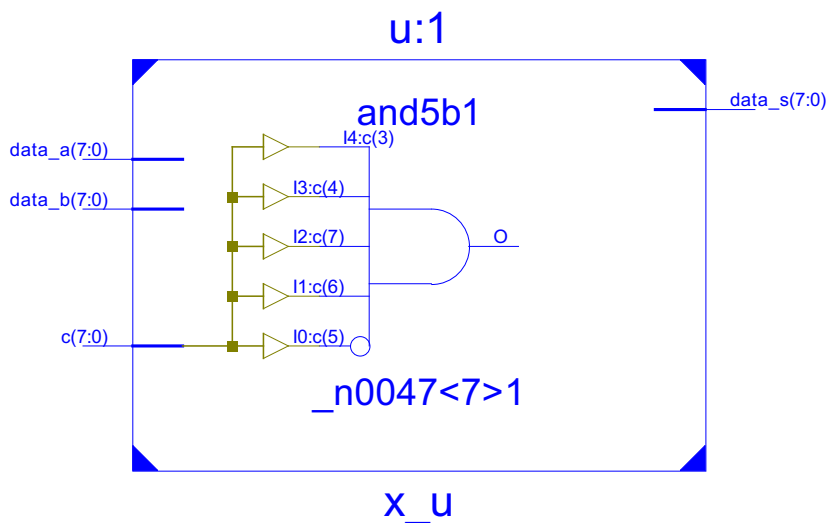


FIGURE 2.43 – Entrées de `_n0047<7>1`

La correspondance des entrées de la primitive avec les 5 premiers bits du bus `smp_data_r` du bloc `smp` est la suivante :

- $I4 = c(3) = \text{smp_data_r}(3)$;
- $I3 = c(4) = \text{smp_data_r}(4)$;
- $I2 = c(7) = \text{smp_data_r}(7)$;
- $I1 = c(6) = \text{smp_data_r}(6)$;
- $I0 = c(5) = \text{smp_data_r}(5)$.

Pour que la sortie de la primitive `_n0047<7>1` soit égale à 1, il faut que sa première entrée soit égale à 0 et les quatre autres à 1, ce qui se traduit par les conditions ci-dessous sur `smp_data_r` :

- `smp_data_r(0)` = indéfini ;
- `smp_data_r(1)` = indéfini ;
- `smp_data_r(2)` = indéfini ;
- `smp_data_r(3)` = 1 ;
- `smp_data_r(4)` = 1 ;
- `smp_data_r(5)` = 0 ;
- `smp_data_r(6)` = 1 ;
- `smp_data_r(7)` = 1.

Si la valeur présentée par le bloc `x_smp` est de la forme 11011xxx, alors le multiplexeur transmettra le résultat d'un décalage à gauche de 1 bit sur la valeur présentée par `x_accu`. Dans le cas contraire, le multiplexeur transmettra sur sa sortie la valeur présentée par le multiplexeur précédent.

2.3 Synthèse de l'analyse des blocs du composant

L'analyse des différents blocs qui constituent le composant S a permis de déterminer le fonctionnement interne de celui-ci mais également la nature des données qui lui sont transmises.

2.3.1 Gestion de la mémoire

Quatre blocs sont utilisés pour implémenter la gestion de la mémoire du composant.

Le premier de ces blocs est le bloc `r` (voir section 2.2.4) qui implémente un banc de 8 registres d'un octet. Chaque registre peut être mis à jour avec la valeur stockée dans l'accumulateur `accu`.

Le second bloc est le bloc `accu` (voir section 2.2.7) qui permet de stocker une valeur temporaire d'un octet en provenance :

- d'un registre adressé par les 3 derniers bits de la valeur présentée par `smp` ;
- du bloc mémoire `smd` adressé par la valeur courante de `accu` ;
- d'une valeur immédiate issue du décodage d'une instruction de `smp` ;
- du résultat d'une opération effectuée par `u`.

Le troisième bloc `accu_w_imp` (voir section 2.2.8) contrôle l'activation de l'écriture dans l'accumulateur en testant la valeur présentée par le bloc `smp`.

Enfin, le bloc `smd` (voir section 2.2.3) se comporte comme une mémoire de 256 octets. Elle est accessible en lecture et écriture depuis l'intérieur ou l'extérieur du composant. Dans le cas d'un accès en écriture depuis l'intérieur du composant, la valeur à écrire est lue depuis l'accumulateur `accu` et l'adressage est effectué à l'aide d'un registre.

2.3.2 Gestion de l'exécution

Le bloc `smp` (voir section 2.2.2) contient une liste d'instructions qui seront exécutées par le composant S. Ce bloc est accessible en écriture depuis l'extérieur et en lecture depuis l'intérieur.

La valeur de sortie du bloc `ip` (voir section 2.2.6) sert à adresser le bloc `smp` et donc à sélectionner la prochaine instruction à exécuter. Ce bloc implémente donc un pointeur d'instruction, comme son nom le laissait supposer. Dans le cas général, sa valeur est incrémentée de 1 à la fin d'une instruction, sauf pour une instruction spécifique qui implémente un saut conditionné par la valeur stockée dans `accu`.

Le bloc `u` (voir section 2.2.9) implémente une unité d'exécution sur les valeurs en sortie des blocs `accu` et `r`. L'opération à exécuter est déterminée par la valeur en sortie de `smp`. Les cinq opérations possibles sont :

- « et » logique entre l'accumulateur et un registre sélectionné par `smp` ;
- « ou » logique entre l'accumulateur et un registre sélectionné par `smp` ;
- une inversion logique sur la valeur de l'accumulateur ;
- l'activation du bit le plus significatif de la valeur de l'accumulateur ;

- un décalage à gauche d'un bit de la valeur de l'accumulateur.

Enfin, le bloc `finished<7>_imp` (voir section 2.2.5) termine l'exécution si l'instruction à exécuter est égale à `0xc8`.

2.3.3 Jeu d'instructions

Les opérations décrites précédemment sont activées en fonction de la valeur de sortie du bloc `smp`. L'analyse des conditions d'activation de chaque opération permet de déterminer le jeu d'instructions supporté par le composant tel que décrit au tableau 2.18.

Mnémonique	Codage	Mise à jour de <code>accu</code>	Description
<code>exit</code>	<code>11001000</code>	Non	Fin d'exécution (voir section 2.2.5)
<code>jmpz rxxx</code>	<code>10111xxx</code>	Non	Saute à l'adresse contenue dans le registre spécifié par les 3 derniers bits de l'instruction, à condition que <code>accu</code> soit égal à 0 (voir section 2.2.6)
<code>mov accu, imm</code>	<code>0xxxxxxx</code>	Oui	Stocke dans <code>accu</code> la valeur immédiate correspondante aux 7 derniers bit de <code>smp</code> (voir section 2.2.7)
<code>mov accu, [accu]</code>	<code>11010000</code>	Oui	Stocke dans <code>accu</code> la valeur de <code>smd</code> adressée par la valeur courante de <code>accu</code> (voir section 2.2.7)
<code>mov accu, rxxx</code>	<code>10101xxx</code>	Oui	Stocke dans <code>accu</code> la valeur du registre sélectionné par les 3 derniers bits de l'instruction (voir section 2.2.7)
<code>mov rxxx, accu</code>	<code>10110xxx</code>	Non	Stocke la valeur de <code>accu</code> dans le registre sélectionné par les 3 derniers bits de l'instruction (voir section 2.2.4)
<code>mov [rxxx], accu</code>	<code>11000xxx</code>	Non	Stocke la valeur de <code>accu</code> dans <code>smd</code> à l'adresse spécifiée par la valeur du registre sélectionné par les 3 derniers bits de l'instruction (voir section 2.2.3)
<code>shl accu, 1</code>	<code>11011xxx</code>	Oui	Décalage à gauche de la valeur stockée dans <code>accu</code> (voir section 2.2.9)
<code>not accu</code>	<code>10100xxx</code>	Oui	Inversion bit par bit de la valeur stockée dans <code>accu</code> (voir section 2.2.9)
<code>msb accu</code>	<code>11100xxx</code>	Oui	Force à 1 le bit le plus significatif de la valeur stockée dans <code>accu</code> (voir section 2.2.9)
<code>and accu, rxxx</code>	<code>10001xxx</code>	Oui	Stocke dans <code>accu</code> le résultat du « Et » logique entre la valeur courante de <code>accu</code> et la valeur du registre sélectionné par les 3 derniers bits de l'instruction (voir section 2.2.9)
<code>or accu, rxxx</code>	<code>10010xxx</code>	Oui	Stocke dans <code>accu</code> le résultat du « Ou » logique entre la valeur courante de <code>accu</code> et la valeur du registre sélectionné par les 3 derniers bits de l'instruction (voir section 2.2.9)

TABLE 2.18 – Jeu d'instructions supporté par le composant

2.4 Rétro-ingénierie du programme

2.4.1 Développement d'un désassembleur

Une fois le jeu d'instruction déterminé, l'ajout du support du composant dans Metasm est possible. La spécification du décodage des instructions est contenue dans le code ci-dessous.

```
require 'metasm/cpu/ssstic2013/main'

module Metasm

class Sstic2013

  def init_sstic2013
```

```

@opcode_list = []
@valid_args.update [ :r, :imm, :accu, :mem_reg,
  :mem_accu, :addr ].inject({}) { |h, v| h.update v => true }
@fields_mask.update :r => 7, :mem_reg => 7, :mem_accu => 0, :imm => 0x7f
@fields_shift.update :r => 0, :mem_reg => 0, :mem_accu => 0, :imm => 0

addop 'exit', 0b1100_1000, :stopexec
addop 'jmpz', 0b1011_1000, :r, :setip
addop 'shl', 0b1101_1000, :accu
addop 'not', 0b1010_0000, :accu
addop 'msb', 0b1110_0000, :accu
addop 'and', 0b1000_1000, :accu, :r
addop 'or', 0b1001_0000, :accu, :r
addop 'mov', 0b0000_0000, :accu, :imm
addop 'mov', 0b1101_0000, :accu, :mem_accu
addop 'mov', 0b1100_0000, :mem_reg, :accu
addop 'mov', 0b1010_1000, :accu, :r
addop 'mov', 0b1011_0000, :r, :accu
addop 'jmp', nil, :addr, :setip, :stopexec

end

end # end of class Sstic2013

end # end of module Metasm

```

L'ensemble du support du composant est présent dans le fichier `gem` disponible à l'annexe [A.2.2](#). L'installation s'effectue avec la commande ci-dessous :

```

$ gem install metasm-sstic2013-ext
Successfully installed metasm-sstic2013-ext-0.0.1
1 gem installed
Installing ri documentation for metasm-sstic2013-ext-0.0.1...
Installing RDoc documentation for metasm-sstic2013-ext-0.0.1...

```

Le paquet `gem` fournit le binaire `disas-sstic2013` qui permet d'appeler le désassembleur de Metasm en initialisant le bon processeur. Le code de ce script est présenté ci-dessous.

```

#!/usr/bin/env ruby

require 'metasm'
require 'metasm/cpu/sstic2013'

if ARGV.size < 1 then
  $stderr.puts "usage: disas-sstic2013 input [plugin]"
  exit 0
end

input, plugin = ARGV.shift, ARGV.shift

data = File.open(input, "rb").read

exe = Metasm::Shellcode.decode(data, Metasm::Sstic2013.new)
dasm = exe.disassembler
dasm.load_plugin plugin if plugin

exe.disassemble
puts dasm

```

Avant de pouvoir appeler `disas-sstic2013`, il est nécessaire d'extraire les données du fichier `smp.py` avec les commandes ci-dessous³ :

```

irb(main):001:0> smp = eval(File.read('smp.py'))
=> [0, 176, 16, 208, 183, [...], 77, 118, 180, 0, 188, 171, 183, 0, 189]
irb(main):002:0> File.open("smp.bin", "wb") {|f| f.write smp.pack('C*')}
=> 231

```

Il ne reste plus qu'à appeler `disas-sstic2013` en spécifiant le chemin de `Metasm` pour désassembler le code binaire obtenu.

```

$ export RUBYLIB=~/.git/metasm
$ disas-sstic2013 smp.bin

entrypoint_0:
    mov accu, 0                                ; @0  00

// Xrefs: 0e6h
loc_1:
    mov r0, accu                              ; @1  b0

// Xrefs: 51h 0e6h
loc_2:
    mov accu, 10h                             ; @2  10
    mov accu, [accu]                          ; @3  d0
    mov r7, accu                              ; @4  b7
    mov accu, r0                              ; @5  a8
    not accu                                  ; @6  a0
    mov r6, accu                              ; @7  b6
    mov accu, loc_0eh                         ; @8  0e
    mov r5, accu                              ; @9  b5
    mov accu, loc_71h                         ; @0ah 71
    mov r4, accu                              ; @0bh b4
    mov accu, 0                               ; @0ch 00
    jmpz r4                                   ; @0dh bc  x:loc_71h
[...]

```

Le résultat complet est disponible à l'annexe [A.2.2](#).

L'utilisation d'un accumulateur rend le code désassemblé très verbeux. Par exemple, il serait souhaitable de pouvoir simplifier `mov accu, loc_0eh ; mov r5, accu` en `mov r5, loc_0eh`.

Le désassembleur de `Metasm` possède un plugin de déobfuscation qui permet de réaliser ce genre de simplifications. Le tableau [2.19](#) présente les simplifications imaginées.

Instructions originales	Instructions simplifiées
<code>mov rx, addr ; jmp rx</code>	<code>jmp addr</code>
<code>mov accu, value ; mov rx, accu</code>	<code>mov rx, value</code>
<code>mov accu, 0 ; mov rx, accu ; mov ry, accu</code>	<code>mov rx, 0 ; mov ry, 0</code>
<code>mov accu, 0 ; jmpz rx</code>	<code>jmp rx</code>
<code>mov accu, value ; msb accu</code>	<code>mov accu, newvalue</code> où <code>newvalue</code> est égale à <code>value</code> avec le bit de poids fort à 1
<code>mov accu, rx ; shl accu ; mov rx, accu</code>	<code>shl rx</code>
<code>mov accu, rs ; mov rd, accu</code>	<code>mov rd, rs</code>
<code>mov accu, rd ; or accu, rs ; mov rd, accu</code>	<code>or rd, rs</code>
<code>mov accu, rs ; or accu, rd ; mov rd, accu</code>	<code>or rd, rs</code>

TABLE 2.19 – Simplifications réalisées sur le code désassemblé

3. fort heureusement, la syntaxe Python du fichier `smp.py` est compatible avec Ruby

Ces simplifications ont été implémentées dans un plugin dont le code est présenté ci-dessous :

Fichier deobfusc-sstic2013.rb

```
module Deobfuscate
  Patterns = {
    'mov r(\d), (\h+h?) ; jmp r\1' => 'jmp %2',
    'mov accu, 0 ; mov r(\d), accu ; mov r(\d), accu' => 'mov r%1, 0 ; mov r%2, 0',
    'mov accu, (\h+h?) ; mov r(\d), accu' => lambda { |dasm, list|
      # kludge
      list.last.address != 0x72 ? 'mov r%2, %1' : nil
    },
    'mov accu, 0 ; jmpz r(\d)' => 'jmp r%1',
    'mov accu, (\h+h?) ; msb accu' => lambda { |dasm, list|
      value = list.first.instruction.args[1]
      p = (1 << 7) | value.reduce
      "mov accu, #{p}"
    },
    'mov accu, r(\d) ; shl accu ; mov r\1, accu' => 'shl r%1',
    'mov accu, r(\d) ; mov r(\d), accu' => 'mov r%2, r%1',
    'mov accu, r(\d) ; or accu, r(\d) ; mov r\1, accu' => 'or r%1, r%2',
    'mov accu, r(\d) ; or accu, r(\d) ; mov r\2, accu' => 'or r%2, r%1',
  }
end

path = File.join(Metasm::Metasmdir, 'samples', 'dasm-plugins', 'deobfuscate.rb')
eval(File.read(path))
```

Il ne reste plus qu'à exécuter `disas-sstic2013` en spécifiant le chemin vers ce plugin pour obtenir le code déobfusqué :

```
$ disas-sstic2013 smp.bin deobfusc-sstic2013.rb
```

```
entrypoint_0:
    mov r0, 0                                ; @0 00b0

// Xrefs: 4eh
loc_2:
    mov accu, 10h                            ; @2 10
    mov accu, [accu]                          ; @3 d0
    mov r7, accu                             ; @4 b7
    mov accu, r0                             ; @5 a8
    not accu                                 ; @6 a0
    mov r6, accu                             ; @7 b6
    mov r5, loc_0eh                          ; @8 0eb5
    jmp loc_71h                             ; @0ah 71 x:loc_71h
[...]
```

Le résultat est disponible à l'annexe [A.2.2](#). On peut remarquer que les quatre instructions originales de 0xa à 0xd ont été simplifiées en une seule instruction.

Le lancement de l'interface graphique de Metasm sur ce code déobfusqué est effectué avec la ligne de commande ci-dessous :

```
$ ruby -r'metasm/cpu/sstic2013' ~/git/metasm/samples/disassemble-gui.rb \
--cpu Sstic2013 --plugin deobfusc-sstic2013.rb smp.bin
```

La capture d'écran [2.44](#) illustre la représentation graphique du programme sous Metasm. La représentation complète des blocs de code est disponible au format `svg` à l'annexe [A.2.2](#).

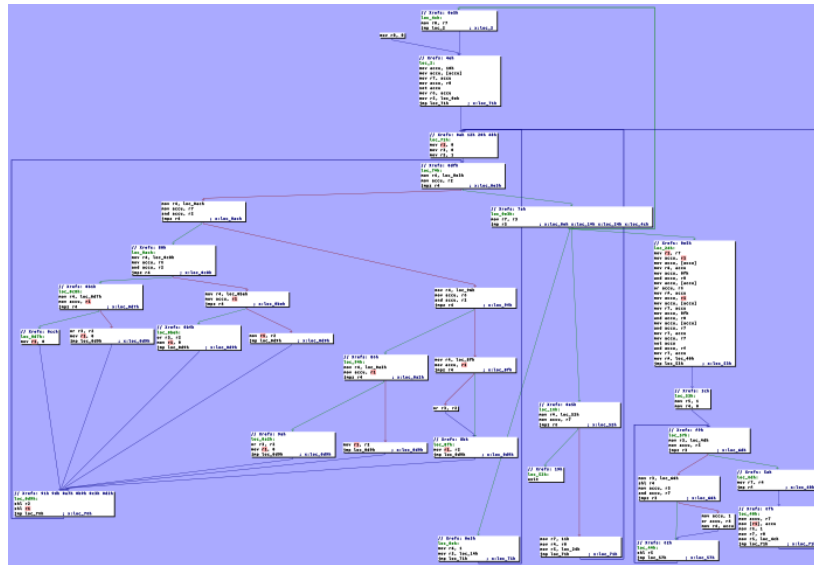


FIGURE 2.44 – Affichage sous Metasm

La figure 2.45 représente l'enchaînement des différents blocs de code qui constituent le programme.

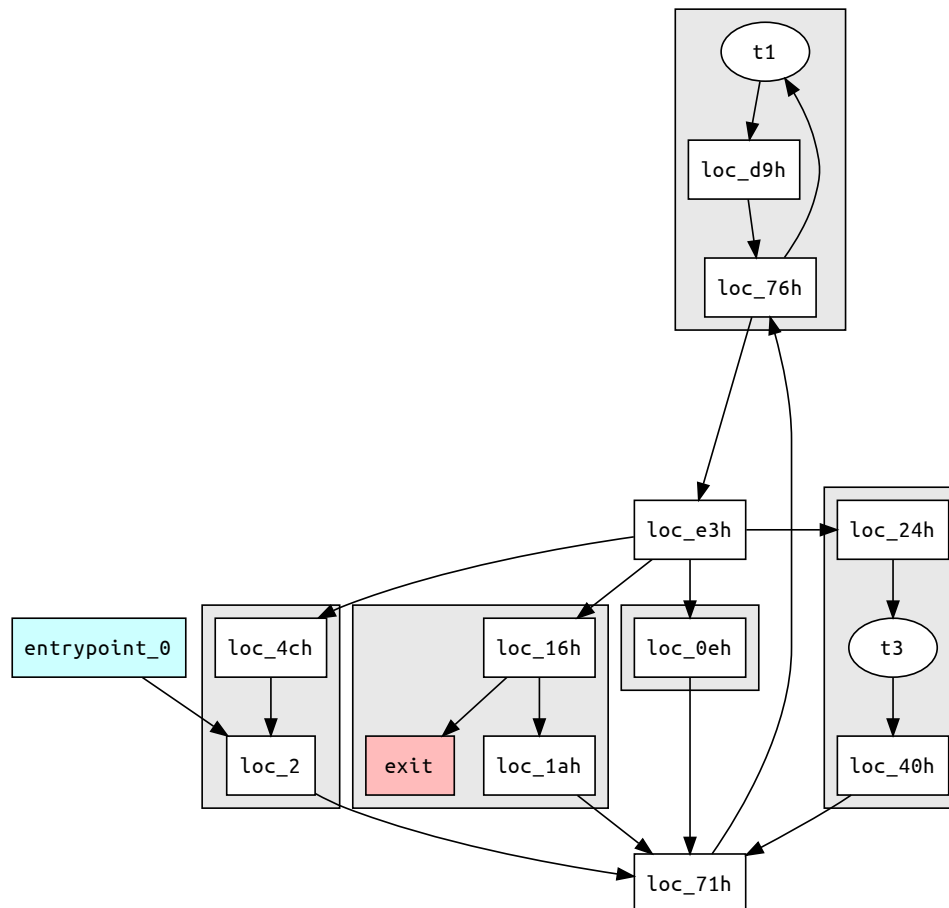


FIGURE 2.45 – Structure du programme

Le déroulement général du programme ressemble à une boucle dont le déroulement est le suivant :

- exécution d'un traitement parmi les blocs `loc_4ch`, `loc_16h`, `loc_0eh`, `loc_24h` ;
- passage dans le bloc `loc_71h` ;
- exécution d'un traitement dans le bloc à l'adresse `loc_76h` ;

- passage dans le bloc `loc_e3h` puis nouvelle itération.

2.4.2 Initialisation du programme

L'initialisation du programme est réalisée à l'adresse `0` par le code ci-dessous :

```
entrypoint_0:
    mov r0, 0                                ; @0 00b0

loc_2:
    mov accu, 10h                            ; @2 10
    mov accu, [accu]                         ; @3 d0
    mov r7, accu                             ; @4 b7
    mov accu, r0                             ; @5 a8
    not accu                                 ; @6 a0
    mov r6, accu                             ; @7 b6
    mov r5, loc_0eh                          ; @8 0eb5
    jmp loc_71h
```

Le registre `r7` est initialisé à `smd[16]`, c'est-à-dire la longueur du bloc de données à traiter. Le registre `r6` est initialisé à `~0 = 255`. Le registre `r5` prend la valeur `loc_0eh` puis le code saute à l'adresse `loc_71h`.

2.4.3 Bloc `loc_71h`

La structure du programme à l'adresse `loc_71h` est représentée à la figure 2.46.

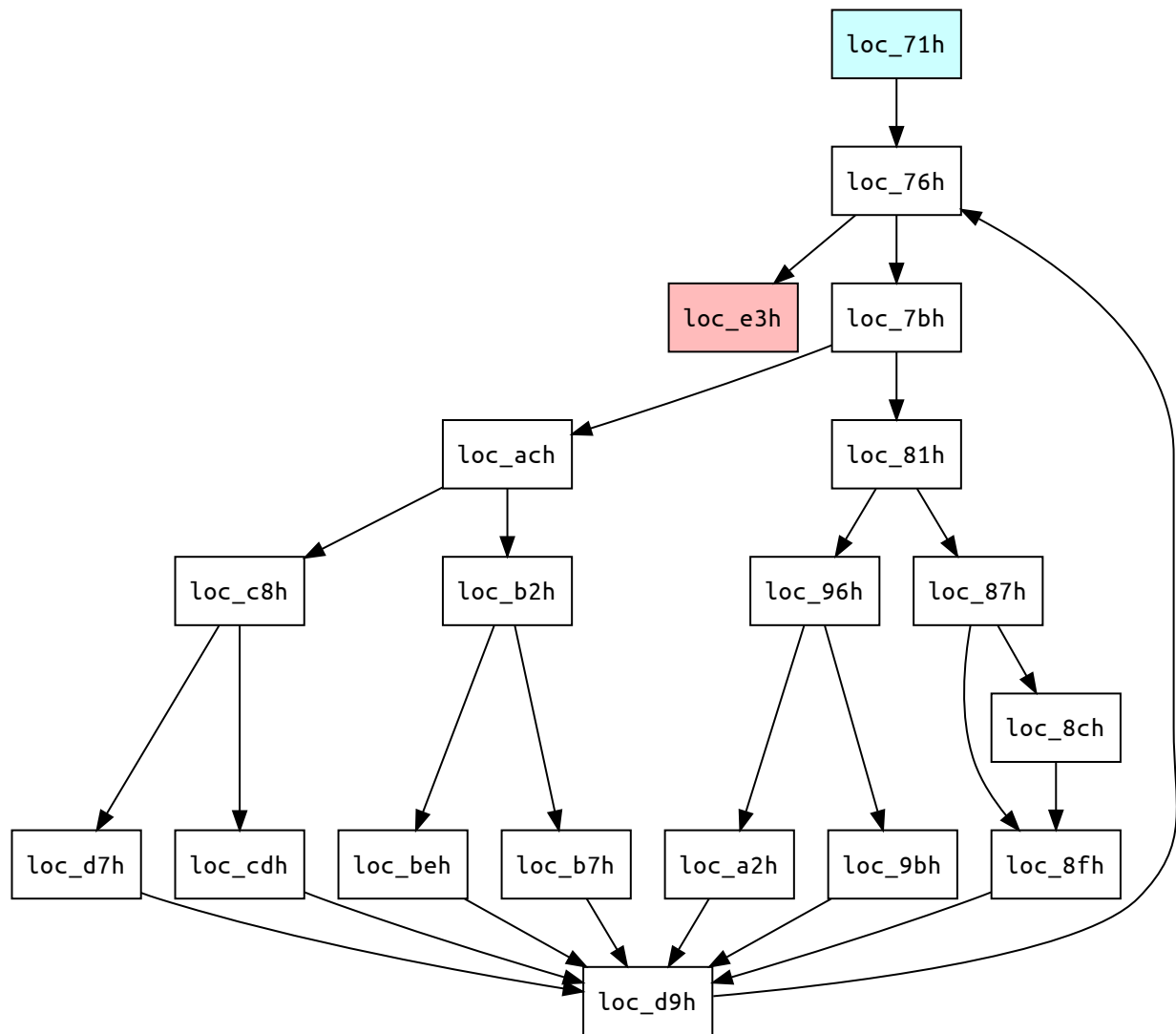


FIGURE 2.46 – Structure du programme à l'adresse 0x71

Il est possible de reconnaître une structure de boucle. Le bloc `loc_76h` correspond au début d'une nouvelle itération alors que le bloc `loc_d9h` semble terminer une itération. A l'intérieur d'une itération, un enchaînement de sauts conditionnels permet d'implémenter un arbre binaire de décision à trois niveaux.

Le code exécuté à l'adresse `loc_71h` commence par les instructions :

```

loc_71h:
    mov r1, 0                ; @71h 00
    mov r3, 0                ; @72h b1b3
    mov r2, 1                ; @74h 01b2

loc_76h:
    mov r4, loc_0e3h         ; @76h 63e0b4
    mov accu, r2              ; @79h aa
    jmpz r4                   ; @7ah bc x:loc_0e3h

    mov r4, loc_0ach         ; @7bh 2ce0b4
    mov accu, r7              ; @7eh af
    and accu, r2              ; @7fh 8a
    jmpz r4                   ; @80h bc x:loc_0ach
  
```

Les registres `r1`, `r2` et `r3` sont initialisés. Si le registre `r2` est nul alors le programme saute à l'adresse `0xe3`, sinon l'exécution continue à l'adresse `0x7b`. Ensuite, la valeur de `r7 & r2` est stockée dans l'accumulateur. Si cette valeur est différente de 0,

l'exécution continue à l'adresse 0x81 avec les instructions ci-dessous :

mov r4, loc_96h	; @81h 16e0b4
mov accu, r6	; @84h ae
and accu, r2	; @85h 8a
jmpz r4	; @86h bc x:loc_96h
mov r4, loc_8fh	; @87h 0fe0b4
mov accu, r1	; @8ah a9
jmpz r4	; @8bh bc x:loc_8fh
or r3, r2	; @8ch ab92b3
loc_8fh:	
mov r1, r2	; @8fh aab1
jmp loc_0d9h	; @91h 59 x:loc_0d9h
loc_96h:	
mov r4, loc_0a2h	; @96h 22e0b4
mov accu, r1	; @99h a9
jmpz r4	; @9ah bc x:loc_0a2h
mov r1, r2	; @9bh aab1
jmp loc_0d9h	; @9dh 59 x:loc_0d9h
loc_0a2h:	
or r3, r2	; @0a2h ab92b3
mov r1, 0	; @0a5h 00b1
jmp loc_0d9h	; @0a7h 59 x:loc_0d9h

Si r7 & r2 est égal à 0, le programme saute à l'adresse 0xac et exécute les instructions ci-dessous :

loc_0ach:	
mov r4, loc_0c8h	; @0ach 48e0b4
mov accu, r6	; @0afh ae
and accu, r2	; @0b0h 8a
jmpz r4	; @0b1h bc x:loc_0c8h
mov r4, loc_0beh	; @0b2h 3ee0b4
mov accu, r1	; @0b5h a9
jmpz r4	; @0b6h bc x:loc_0beh
mov r1, r2	; @0b7h aab1
jmp loc_0d9h	; @0b9h 59 x:loc_0d9h
loc_0beh:	
or r3, r2	; @0beh aa93b3
mov r1, 0	; @0c1h 00b1
jmp loc_0d9h	; @0c3h 59 x:loc_0d9h
loc_0c8h:	
mov r4, loc_0d7h	; @0c8h 57e0b4
mov accu, r1	; @0cbh a9
jmpz r4	; @0cch bc x:loc_0d7h
or r3, r2	; @0cdh aa93b3
mov r1, 0	; @0d0h 00b1
jmp loc_0d9h	; @0d2h 59 x:loc_0d9h
loc_0d7h:	

<code>mov r1, 0</code>	<code>; @0d7h 00b1</code>
loc_0d9h:	
<code>shl r2</code>	<code>; @0d9h aad8b2</code>
<code>shl r1</code>	<code>; @0dch a9d8b1</code>
<code>jmp loc_76h</code>	<code>; @0dfh 76 x:loc_76h</code>

Une traduction littérale des instructions en assembleur vers C donne le résultat ci-dessous :

```
uint8_t r0, r1, r2, r3, r4, r5, r6, r7;
```

```
void sub_71h(void) {
    /* loc_71h */
    r1 = 0;    r2 = 1;    r3 = 0;
```

```
loc_76h:
    if (!r2)
        goto loc_0e3h;

    /* loc_7bh */
    if (r2 & r7) {
        /* loc_81h */
        if (r2 & r6) {
            /* loc_87h */
            if (r1) {
                /* loc_8ch */
                r3 |= r2;
            }
            /* loc_8fh */
            r1 = r2;
            goto loc_0d9h;
        } else {
            /* loc_96h */
            if (r1) {
                /* loc_9bh */
                r1 = r2;
                goto loc_0d9h;
            } else {
                /* loc_0a2h */
                r3 |= r2;
                r1 = 0;
                goto loc_0d9h;
            }
        }
    } else {
        /* loc_0ah */
        if (r2 & r6) {
            /* loc_0b2h */
            if (r1) {
                /* loc_0b7h */
                r1 = r2;
                goto loc_0d9h;
            } else {
                /* loc_0beh */
                r3 |= r2;
                r1 = 0;
                goto loc_0d9h;
            }
        }
    } else {
```

```

        /* loc_0c8h */
        if (r1) {
            /* loc_0cdh */
            r3 |= r2;
            r1 = 0;
            goto loc_0d9h;
        } else {
            /* loc_0d7h */
            r1 = 0;
        }
    }
}

loc_0d9h:
    r2 <=<= 1;
    r1 <=<= 1;
    goto loc_76h;

loc_0e3h:
    return;
}

```

Certaines simplifications peuvent alors être réalisées pour obtenir le code C équivalent ci-dessous :

```

void sub_71h_1(void) {
    r1 = 0;    r2 = 1;    r3 = 0;

    while (r2) {
        if (r2 & r7) {
            if (r2 & r6) {
                if (r1)
                    r3 |= r2;
                r1 = r2;
            } else {
                if (r1)
                    r1 = r2;
                else {
                    r3 |= r2;
                    r1 = 0;
                }
            }
        }
        else {
            if (r2 & r6) {
                if (r1)
                    r1 = r2;
                else {
                    r3 |= r2;
                    r1 = 0;
                }
            }
            else {
                if (r1) {
                    r3 |= r2;
                    r1 = 0;
                }
                else
                    r1 = 0;
            }
        }
        r2 <=<= 1;
        r1 <=<= 1;
    }
}

```

```

    }
}

```

La boucle `while` effectue 8 itérations en décalant la valeur de `r2` et `r1` à chaque fois. A l'intérieur de la boucle, deux tests sont réalisés pour savoir si le bit des registres `r6` et `r7` à la position spécifiée par `r2` vaut 0 ou 1. Un dernier test est réalisé sur la valeur de `r1`. Selon le résultat de ces trois tests imbriqués, les valeurs de `r3` et `r1` sont modifiées. Ces opérations peuvent être synthétisées par la table de vérité présentée dans le tableau 2.20.

Entrées			Sorties	
r6	r7	r1	r3	r1
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

TABLE 2.20 – Table de vérité de la fonction `sub_71h`

Cette table de vérité correspond à une addition bit à bit modulo 255 des octets stockés dans `r6` et `r7`, le résultat étant stocké dans le registre `r3`. Le registre `r1` sert à propager la retenue entre deux itérations.

Une fois l'addition effectuée, le code saute alors à l'adresse `loc_0e3h`.

2.4.4 Bloc `loc_0e3h`

Le code à l'adresse `loc_0e3h` est présenté ci-dessous :

```

loc_0e3h:
    mov r7, r3                ; @0e3h  abb7
    jmp r5                    ; @0e5h  00  x:loc_0eh x:loc_16h x:loc_24h x:loc_4ch

```

`r7` prend la valeur de `r3`, c'est-à-dire le résultat de l'addition entre `r6` et la valeur précédente de `r7`. Le programme continue alors à l'adresse stockée dans le registre `r5` qui a été initialisé à la valeur `loc_0eh` par le bloc à l'adresse `loc_2`.

2.4.5 Bloc `loc_0eh`

Le code à l'adresse `loc_0eh` est présenté ci-dessous :

```

loc_0eh:
    mov r6, 1                 ; @0eh  01b6
    mov r5, loc_16h           ; @10h  16b5
    jmp loc_71h

```

Les registres `r6` et `r5` sont initialisés respectivement à 1 et `loc_16h`. Le programme saute alors à l'adresse `loc_71h` pour effectuer une addition entre `r6` et `r7`. A la fin de l'addition, le programme continue son exécution à l'adresse stockée dans `r5`, donc `loc_16h`.

2.4.6 Bloc `loc_16h`

Le code à l'adresse `loc_16h` est présenté ci-dessous :

```

loc_16h:
    mov r6, loc_52h           ; @16h 52b6
    mov accu, r7              ; @18h af
    jmpz r6                   ; @19h be x:loc_52h

    mov r7, 11h               ; @1ah 11b7
    mov r6, r0                ; @1ch a8b6
    mov r5, loc_24h           ; @1eh 24b5
    jmp loc_71h

```

Si le registre r7 contient la valeur 0 alors le programme saute à l'adresse loc_52h qui correspond à l'instruction `exit`. Sinon, r7 prend la valeur de l'addition entre `0x11 = 17` et r0 puis le programme saute à l'adresse loc_24h.

2.4.7 Bloc loc_24h

Le code à l'adresse loc_24h est présenté ci-dessous :

```

loc_24h:
    mov r1, r7                ; @24h afb1
    mov accu, r1              ; @26h a9
    mov accu, [accu]          ; @27h d0
    mov r6, accu              ; @28h b6
    mov accu, 0fh             ; @29h 0f
    and accu, r0              ; @2ah 88
    mov accu, [accu]          ; @2bh d0
    or accu, r6               ; @2ch 96
    mov r6, accu              ; @2dh b6
    mov accu, r1              ; @2eh a9
    mov accu, [accu]          ; @2fh d0
    mov r7, accu              ; @30h b7
    mov accu, 0fh             ; @31h 0f
    and accu, r0              ; @32h 88
    mov accu, [accu]          ; @33h d0
    and accu, r7              ; @34h 8f
    mov r7, accu              ; @35h b7
    mov accu, r7              ; @36h af
    not accu                  ; @37h a0
    and accu, r6              ; @38h 8e
    mov r7, accu              ; @39h b7
    mov r6, loc_40h           ; @3ah 40b6
    jmp loc_53h               ; @3ch 53 x:loc_53h

```

Ce code est équivalent à la fonction C ci-dessous :

```

void sub_24h(void) {
    r1 = r7;
    r6 = smd[r1];
    r6 |= smd[r0 & 0xf];
    r7 = smd[r1];
    r7 &= smd[r0 & 0xf];
    r7 = ~r7;
    r7 = r7 & r6;
}

```

Il est possible de la simplifier en la fonction suivante :

```

void sub_24h_1(void) {
    uint8_t b, k;

```

```

b = smd[r7];
k = smd[r0 % 16];
r7 = (b | k) & ~(b & k);
}

```

Cela revient à calculer un « ou »-exclusif entre `smd[r7]` et `smd[r0 % 16]` car

$$p \oplus q = (p \vee q) \wedge \neg(p \wedge q)$$

où \oplus représente la disjonction exclusive, \vee la disjonction, \wedge la conjonction et \neg la négation.

Une fois ce calcul effectué, le programme continue son exécution à l'adresse `loc_53h` dont le code est présenté ci-dessous :

```

loc_53h:
    mov r5, 1                ; @53h 01b5
    mov r4, 0                ; @55h 00b4

loc_57h:
    mov r3, loc_6dh          ; @57h 6db3
    mov accu, r5             ; @59h ad
    jmpz r3                  ; @5ah bb x:loc_6dh

    mov r3, loc_66h          ; @5bh 66b3
    shl r4                   ; @5dh acd8b4
    mov accu, r5             ; @60h ad
    and accu, r7             ; @61h 8f
    jmpz r3                  ; @62h bb x:loc_66h

    mov accu, 1              ; @63h 01
    or accu, r4              ; @64h 94
    mov r4, accu             ; @65h b4

loc_66h:
    shl r5                   ; @66h add8b5
    jmp loc_57h              ; @69h 57 x:loc_57h

loc_6dh:
    mov r7, r4               ; @6dh acb7
    jmp r6

```

La traduction littérale de ces instructions permet d'obtenir la fonction C ci-dessous :

```

void sub_53h(void) {
    r5 = 1;
    r4 = 0;

loc_57h:
    if (r5 == 0)
        goto loc_6dh;
    r4 <=< 1;
    if (r7 & r5)
        r4 |= 1;
    r5 <=< 1;
    goto loc_57h;

loc_6dh:
    r7 = r4;
    sub_40h();
}

```

La simplification de cette fonction aboutit à la fonction ci-dessous :

```
void sub_53h_1(void) {
    r5 = 1;
    r4 = 0;

    while (r5) {
        r4 <=< 1;
        if (r7 & r5)
            r4 |= 1;
        r5 <=< 1;
    }
    r7 = r4;
    sub_40h();
}
```

Cette fonction va inverser l'ordre des bits de l'octet contenu dans r7. Par exemple, `sub_53h(11011110) = 01111011`.

Le programme continue ensuite son exécution à l'adresse `loc_40h`.

2.4.8 Bloc `loc_40h`

Le code exécuté à l'adresse `loc_40h` est présenté ci-dessous :

```
loc_40h:
    mov  accu, r7                ; @40h  af
    mov  [r1], accu              ; @41h  c1
    mov  r6, 1                   ; @42h  01b6
    mov  r7, r0                  ; @44h  a8b7
    mov  r5, loc_4ch              ; @46h  4cb5
    jmp  loc_71h                  ; @48h  71  x:loc_71h

loc_4ch:
    mov  r0, r7                  ; @4ch  afb0
    jmp  loc_2
```

La valeur de r7 va être écrite dans `smd` à la position indiquée par r1. Ensuite, r7 prend la valeur du résultat de l'addition de r0 et 1. Cette valeur est affectée à r0 puis le programme saute à l'adresse `loc_2`.

2.4.9 Résultat de la rétro-ingénierie

Les différentes étapes étudiées précédemment peuvent être synthétisées par la fonction C ci-dessous :

```
void program(void) {
    /* entrypoint */
    r0 = 0;

loc_2:
    r7 = smd[16];
    r6 = ~r0;

    /* loc_71h, loc_0e3h */
    r7 += r6

    /* loc_0eh */
    r6 = 1;
}
```

```

/* loc_71h, loc_0e3h */
r7 += r6;

/* loc_16h */
if (r7 == 0)
    exit(EXIT_SUCCESS);

r7 = 17; r6 = r0;

/* loc_71h, loc_0e3h */
r7 += r6;

/* loc_24h */
r1 = r7;
r7 = smd[r1] ^ smd[r0 % 16];

/* loc_53h */
r5 = 1; r4 = 0;

while (r5) {
    r4 <= 1;
    if (r7 & r5)
        r4 |= 1;
    r5 <= 1;
}
r7 = r4;

/* loc_40h */
smd[r1] = r7;

r6 = 1;
r7 = r0;

/* loc_71h, loc_0e3h */
r7 += r6;

/* loc_4ch */
r0 = r7;
goto loc_2;
}

```

La condition de sortie du programme est que `smd[16] + ~r0 + 1 == 0`, ce qui équivaut à `r0 == smd[16]`. La simplification de la fonction précédente aboutit alors à la fonction ci-dessous :

```

void program_1(void) {
    uint8_t i, j, k, len, b, r;

    len = smd[16];
    for (i = 0; i < len; i++) {
        k = smd[i % 16];
        b = smd[17 + i];
        b ^= k;

        r = 0;
        for (j = 0; j < 8; j++) {
            r <= 1;
            if (b & (1 << j))
                r |= 1;
        }
        smd[17 + i] = r;
    }
}

```

```
}  
}
```

2.5 Détermination de la clé

Le programme analysé est donc une routine de déchiffrement appliquée sur chaque bloc de données envoyé au composant. La conception de cette routine comporte des faiblesses qui peuvent être exploitées pour retrouver la clé de déchiffrement. Ces faiblesses sont :

- la valeur d'un octet déchiffré ne dépend que de la valeur de l'octet chiffré et d'un seul octet de clé ;
- un même octet de clé est réutilisé tel quel pour déchiffrer plusieurs octets chiffrés ;
- à partir de la position i d'un octet dans le bloc déchiffré, il est trivial d'identifier l'octet de clé utilisé : il suffit de calculer $i \% 16$ pour retrouver l'index de celui-ci dans la clé complète de 16 octets.

De plus, d'après le script `decrypt.py`, les données déchiffrées sont codées en base64. En effet, le script effectue les traitements ci-dessous si la clé est correcte :

```
if result_md5 != target_md5:  
    print("Bad key...")  
    sys.exit(1)  
  
result = base64.b64decode("".join(result))  
d = open("atad", "wb")  
d.write(result)  
d.close()  
sys.exit(0)
```

Cette information sur les données déchiffrées va se révéler très intéressante pour retrouver la clé de déchiffrement. En effet, il est alors possible de tester la valeur des octets obtenus pour discriminer les clés incorrectes de celle recherchée. Un caractère peut correspondre à un codage base64 si celui-ci est un caractère alpha-numérique, le caractère '+', le caractère '/' ou un saut de ligne.

En exploitant les faiblesses identifiées dans la routine de déchiffrement et la contrainte sur le format du contenu déchiffré, la clé peut être retrouvée en déroulant l'attaque suivante :

- un état S est initialisé contenant toutes les valeurs possibles (0 à 255) pour chacun des 16 octets de la clé ;
- des clés candidates `key_cand` sont générées et des tests de déchiffrement sont effectués :
 - chaque octet du bloc déchiffré est examiné pour vérifier s'il correspond à un codage en base 64,
 - si ce n'est pas le cas, on peut alors exclure dans l'état S la valeur de l'octet de clé utilisé pour chiffrer l'octet examiné, c'est-à-dire `key_cand[i % 16]` où i est la position de l'octet courant,
 - sinon, l'octet suivant est examiné.
- ces tests de déchiffrement sont répétés jusqu'à l'identification d'un seul octet valide pour chacune des 16 positions de la clé.

L'état des clés candidates est initialisé par la fonction ci-dessous :

```
void init_S(void) {  
    int i, j;  
    for (i = 0; i < 16; i++) {  
        S[i] = malloc(sizeof(uint8_t) * 255);  
        for (j = 0; j < 256; j++) {  
            S[i][j] = 1;  
        }  
    }  
}
```

Le premier tableau représente les 16 positions de la clé. Pour chacune de ces 16 positions, un tableau est initialisé avec 255 valeurs à 1. Cela signifie qu'à cette position, l'octet de clé peut prendre n'importe quelle valeur entre 0 et 255. Pour exclure la valeur v à la position i , il suffit d'affecter la valeur 0 à $S[i][v]$. L'attaque se termine lorsqu'il ne reste plus qu'une seule valeur à 1 dans chacune des 16 positions.

La fonction ci-dessous implémente le test sur la condition d'arrêt de l'attaque :

```
int need_bf(void) {
    int i, j, c;
    for (i = 0; i < 16; i++) {
        c = 0;
        for (j = 0; j < 256; j++) {
            c += S[i][j];
            if (c > 1)
                return 1;
        }
    }
    return 0;
}
```

Enfin, le corps de l'attaque correspond au code ci-dessous :

```
uint8_t is_base64(uint8_t b) {
    return (b == '\r' || b == '\n' || b == '+' || b == '/' || isalnum(b));
}

void do_bf(void) {
    int i, j, k, count = 0;
    size_t remaining, current_block_size;
    uint8_t key_cand[16];
    uint8_t tmp;
    char *block_in;

    remaining = datalen;
    while (remaining > 0) {
        current_block_size = (remaining >= BLOCK_SIZE) ? BLOCK_SIZE : remaining;
        remaining -= current_block_size;

        for (i = 0; i < 256; i++) {
            for (j = 0; j < 16; j++)
                key_cand[j] = i;

            block_in = data + count * BLOCK_SIZE;
            decrypt_block(key_cand, current_block_size, (const uint8_t*) block_in, block_out);

            for (k = 0; k < current_block_size; k++) {
                tmp = block_out[k];
                j = k % 16;
                if (!is_base64(tmp)) {
                    /* la valeur key_cand[k % 16] pour l'octet à la position k % 16 n'est pas possible */
                    S[j][key_cand[j]] = 0;
                }
            }

            if (need_bf() == FALSE)
                return;
        }
        count++;
    }
}
```

Le fichier `solve-part2.c` qui implémente l'attaque est disponible à l'annexe [A.2.3](#).

L'exécution de ce programme sur le fichier `data` permet d'identifier la bonne clé et de déchiffrer les données :

```
$ gcc -pedantic -o solve-part2 solve-part2.c
$ ./solve-part2 archive_data > atad
[*] solving part 2
[!] key = e683dcbc16ef58c665ac23d31e6da125
$ md5sum atad
6c0708b3cf6e32cbae4236bdea062979  atad
```

L'empreinte MD5 du fichier obtenu correspond bien à celle testée dans le script `decrypt.py` :

```
target_md5 = "6c0708b3cf6e32cbae4236bdea062979"
```

Chapitre 3

Rétro-ingénierie du fichier `script.ps`

3.1 Identification du fichier `atad`

Une fois le fichier `atad` obtenu, le premier réflexe est de décoder son contenu avec la commande `base64` :

```
$ md5sum atad
6c0708b3cf6e32cbae4236bdea062979  atad
$ base64 -d atad > atad.dec
$ file atad.dec
atad.dec: ASCII text, with very long lines
$ wc -l atad.dec
1 atad.dec
$ ls -l atad.dec
-rw-rw-r-- 1 jpe jpe 32610 avril 27 17:10 atad.dec
```

On obtient donc un fichier de 32610 octets constitué d'une seule ligne. La commande ci-dessous permet d'afficher le début du fichier :

```
$ dd if=atad.dec bs=1 count=64 2>/dev/null
/I1 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIHexDeco%
```

Une recherche Google des mots clés `currentfile /SubFileDecode filter` retourne un certain nombre de résultats relatifs au langage de programmation Postscript, notamment une documentation de référence d'Adobe nommée « Filters and Reusable Streams », téléchargeable à l'adresse <http://partners.adobe.com/public/developer/en/ps/sdk/TN5603.Filters.pdf>.

Postscript est un langage de programmation mis au point par Adobe dont la principale utilité est de décrire des documents en vue de les imprimer. Un interpréteur Postscript fonctionne comme un calculateur en notation polonaise inverse : chaque mot du programme est évalué, le résultat étant ensuite placé sur une pile.

Pour vérifier que le fichier `atad.dec` est bien un programme Postscript, il est possible de tenter une exécution avec l'interpréteur Ghostscript, comme présenté ci-dessous :

```
$ gs atad.dec
GPL Ghostscript 9.07 (2013-02-14)
Copyright (C) 2012 Artifex Software, Inc. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
missing '--' preceding script file
usage: gs -- script.ps key
```

D'après le message indiqué, le script prend en argument une clé. Il est alors possible de tenter une seconde exécution en passant une clé quelconque.

```
$ gs -- atad.dec aaaaaaaaaa
GPL Ghostscript 9.07 (2013-02-14)
Copyright (C) 2012 Artifex Software, Inc. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
```

Cette fois ci, aucun message n'est affiché. Le script termine rapidement sans aucune indication. Il y a fort à parier que la clé passée en argument n'est pas celle attendue. Un examen détaillé du fichier `atad.dec` va sans doute se révéler nécessaire pour déterminer la clé correcte.

3.2 Développement d'une boîte à outils Postscript

Pour faciliter l'analyse du programme Postscript, un boîte à outils a été développée en Ruby. Les deux fonctionnalités de cette dernière sont :

- la possibilité de réindenter un programme Postscript existant ;
- la présence d'un débogueur Postscript (voir chapitre 3.4.1).

Pour installer cet outil, il suffit de récupérer le fichier `ruby-postscript-0.0.1.gem` disponible à l'annexe A.3.2 et de l'installer avec la commande ci-dessous :

```
$ gem install ruby-postscript-0.0.1.gem
Fetching: awesome_print-1.1.0.gem (100%)
Successfully installed awesome_print-1.1.0
Successfully installed ruby-postscript-0.0.1
2 gems installed
Installing ri documentation for awesome_print-1.1.0...
Installing ri documentation for ruby-postscript-0.0.1...
Installing RDoc documentation for awesome_print-1.1.0...
Installing RDoc documentation for ruby-postscript-0.0.1...
```

L'outil `ruby-ps` peut alors être invoqué en ligne de commande :

```
$ ruby-ps -h
Usage: ruby-ps [options] input.ps [postscript arguments]

Specific options:
  -p, --pretty-print      Parse the specified file and pretty print its content on stdout
  -b, --break             Break on start
  -c, --breakpoints x,y,z List of breakpoints (default: rdebug)
  -h, --help              Show usage
```

Des exemples d'utilisation sont présentés au chapitre suivant ainsi qu'au chapitre 3.4.1.

3.3 Gestion des arguments de `script.ps`

Dans sa version brute, la compréhension du fichier `atad.dec` est difficile car l'intégralité du programme tient sur une seule ligne. Il est alors nécessaire de réindenter le fichier, soit à la main, soit automatiquement en utilisant un outil. L'outil `ruby-ps` décrit au chapitre précédent permet d'obtenir une version correctement indentée du programme Postscript :

```
$ ruby-ps -p ../part2/archive/atad.dec > script.ps
$ wc -l script.ps
353 script.ps
```

Le résultat obtenu, `script.ps`, est disponible en annexe A.3.1 de ce document.

Le programme commence par définir quatre blocs de données dans les variables I1, I2, I3 et I4, comme présenté ci-dessous :

Extrait de script.ps

```
0 /I1 currentfile
1 0 (cafebabe) /SubFileDecode filter
2 /ASCIIHexDecode filter
3 /ReusableStreamDecode filter
4 cf760bc77db1f282e881ede9a10122b220887466b973b[...]
5 cafebabe
6 def
```

Le mode de définition est similaire pour les autres variables I2 jusqu'à I4.

Ces quatre blocs de données sont définis à l'aide de l'instruction `filter` dont le comportement diffère en fonction du premier argument précisant le type de filtre à appliquer, à savoir :

- `/SubFileDecode` qui permet de faire référence à des données déjà présentes au sein même du programme, ces données étant alors délimitées par un marqueur textuel (`cafebabe` dans le cas de `script.ps`);
- `/ASCIIHexDecode` qui va lire les données présentes dans une source données pour les convertir ensuite en binaire depuis leur représentation hexadécimale;
- `/ReusableStreamDecode` qui permet de rendre une source de données réutilisable plusieurs fois¹.

L'enchaînement de ces trois filtres permet d'obtenir une référence vers un bloc de données binaires issues d'une chaîne hexadécimale présente dans le code source du programme. L'utilisation des quatre références I1, I2, I3 et I4 sera examinée dans la suite de ce document.

Le programme continue ensuite par définir une fonction `error` qui se contente d'afficher une chaîne de caractères sur la sortie d'erreur de l'interpréteur.

Extrait de script.ps

```
33 /error
34 {
35   (%stderr) (w) file
36   exch
37   writestring
38 }
39 bind
40 def
```

Ensuite, une fonction est associée au gestionnaire d'erreur comme ci-dessous :

Extrait de script.ps

```
42 errordict
43 /handleerror
44 {
45   quit
46 }
47 put
```

Le programme définit alors la fonction principale `main`. La structure de cette fonction est présentée ci-dessous :

```
/main
{ mark
  shellarguments
  { counttomark
    1 eq
    { dup
      length
      exch
      /ReusableStreamDecode filter
      exch
      2 idiv
```

1. par défaut, une source de données Postscript se ferme automatiquement lorsque toutes les données ont été lues.


```

    string
    readhexstring
    pop
    dup
    length
    16 eq
    { <<traitement principal>> } if
    false
  }
  { (no key provided\n) error true } ifelse
}
{ (missing '--' preceding script file\n) error true } ifelse

{ (usage: gs -- script.ps key\n) error flush } if
} bind def

```

La fonction commence par effectuer un certain nombre de vérifications élémentaires, notamment pour s'assurer que des arguments ont bien été passés en ligne de commandes. Par exemple, si la procédure `shellarguments` retourne `false`, la seconde opérande de l'instruction `ifelse` est exécutée et le message d'erreur `missing '--' preceding script file` est affiché.

Un appel à la page de manuel de l'interpréteur Ghostscript nous permet de retrouver la signification du paramètre `--` :

OPTIONS

`-- filename arg1 ...`

Takes the next argument as a file name as usual, but takes all remaining arguments (even if they have the syntactic form of switches) and defines the name "ARGUMENTS" in "userdict" (not "systemdict") as an array of those strings, before running the file. When Ghostscript finishes executing the file, it exits back to the shell.

Le code de la procédure `shellarguments` est disponible dans le fichier `gs_init.ps` fourni avec Ghostscript :

```

/shellarguments      % -> shell_arguments true (or) false
{ /ARGUMENTS where
  { /ARGUMENTS get dup type /arraytype eq
    { aload pop /ARGUMENTS []/null store []/true }
    { pop []/false }
    ifelse }
  { []/false } ifelse
} bind def

```

Pour que `shellarguments` dépose `true` sur la pile (permettant ainsi de passer le premier `ifelse`), il suffit de passer un argument en ligne de commande après le nom du script à exécuter. Cette conclusion est conforme au comportement observé lors du premier test à la section 3.1.

Dans ce cas de figure, la première opérande de l'instruction `ifelse` est exécutée. L'instruction `counttomark` va compter le nombre d'opérandes présentes sur la pile jusqu'à la première occurrence d'une marque. En pratique, cet appel compte le nombre d'éléments déposés sur la pile par `shellarguments`, donc le nombre d'arguments passés en ligne de commande. Le nombre attendu est 1, sinon le programme termine prématurément en affichant `no key provided`.

Si le nombre d'arguments passés en ligne de commande est bien égal à 1, le programme continue. A ce stade, la pile contient une marque (`-mark-`) et la clé passée en argument au script. Le code alors exécuté est présenté ci-dessous avec un commentaire pour chaque instruction :

	Extrait de <code>script.ps</code>	
57	<code>dup</code>	<i>% la clé est dupliquée sur la pile</i>
58	<code>length</code>	<i>% la longueur de la copie de la clé est déposée sur la pile</i>
59	<code>exch</code>	<i>% inversion de la clé originale et de sa longueur de la clé</i>

```
60 /ReusableStreamDecode filter % un filtre réutilisable est créé à partir de la clé originale
61 exch % la longueur de la clé et le filtre sont échangées sur la pile
62 2 idiv % la longueur de la clé est divisée par 2
63 string % allocation d'une chaîne de caractères (longueur sur la pile)
64 readhexstring % conversion des données hexa du filtre, résultat dans la chaîne
65 pop % le code de retour de readhexstring est supprimé de la pile
66 dup % la chaîne de caractères est dupliquée
67 dup % la longueur de la chaîne dupliquée est déposée sur la pile
68 16 eq % cette longueur est comparée à 16
```

Pour résumer, les instructions précédentes convertissent la clé passée en paramètre d'une représentation hexadécimale vers un format binaire, stockent le résultat dans une nouvelle chaîne de caractères et testent si la longueur de celle-ci est égale à 16 octets. La clé passée en paramètre doit donc être constituée de 32 caractères hexadécimaux pour correspondre à 16 octets binaires (soit 128 bits).

Dans la suite de ce document, la variable `K` sera utilisée pour désigner la clé binaire de 16 octets passée en argument au script.

Un essai peut alors être réalisé avec une clé d'une longueur correcte :

```
$ gs -- script.ps `ruby -e 'puts "A"*32`
GPL Ghostscript 9.07 (2013-02-14)
Copyright (C) 2012 Artifex Software, Inc. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
```

L'exécution termine rapidement, sans apporter plus d'informations. Il est possible, en modifiant simplement le code de `script.ps`, de s'assurer que le test sur la longueur de la clé soit bien vérifié. Par exemple, la modification ci-dessous va afficher le message `key length ok` sans perturber la suite de l'exécution du programme.

```
16 eq
{ (key length ok) == <<traitement principal>> }
if
```

Un nouvel essai avec le code modifié affiche bien le message attendu :

```
$ gs -- script.ps `ruby -e 'puts "A"*32`
GPL Ghostscript 9.07 (2013-02-14)
Copyright (C) 2012 Artifex Software, Inc. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
(key length ok)
```

Après s'être assuré qu'il n'y a pas de sortie prématurée du programme à cause des vérifications sur les arguments, il faut maintenant s'attarder sur l'analyse du cœur de la fonction `main` pour comprendre l'utilisation de la clé passée en paramètre.

3.4 Étude du traitement principal de la fonction `main`

3.4.1 Utilisation d'un débogueur Postscript

Pour faciliter l'analyse du programme `script.ps`, il est possible d'utiliser le débogueur intégré à l'outil `ruby-ps` décrit au chapitre 3.2. Celui-ci permet de suivre une instruction pas à pas en affichant le contenu de la pile. Pour invoquer le débogueur, il suffit d'ajouter l'instruction `rdebug` dans le corps du programme. Dans notre cas, cette instruction a été ajoutée juste après le test sur la longueur de la clé et donc avant l'exécution du traitement principal.

```
16 eq
{
  rdebug
  <<traitement principal>>
}
```

if

L'exécution du fichier `script.ps` est possible avec la ligne de commande ci-dessous :

```
$ ruby-ps script.ps `ruby -e 'print "A"*32'`
[!] break for rdebug
> help
break object: add a breakpoint for the specified object
del object: remove breakpoint for the specified object
continue: continue the program execution until next breakpoint
step: break at next instruction
show (breakpoints|stack|userdict|systemdict): show the specified object
trace: enable / disable call tracing
irb: spawn irb shell
locate count: show position in the current array
exit: exit the debugger
```

Le débogueur permet de démarrer une console IRB, ce qui se révèle utile pour inspecter son état interne. Par exemple, le code ci-dessous va enregistrer dans un fichier les données binaires qui correspondent aux filtres I1, I2, I3 et I4 :

```
irb(main):001:0> @ctx.userdict.keys
=> [:I1, :I2, :I3, :I4, :error, :main]
irb(main):002:0> @ctx.userdict.select {|k, io| k =~ /I\d/}.each {|k, io|
irb(main):003:1*   File.open("data/#{k}.dat", "wb") {|f| f.write io.read}
irb(main):004:1>   io.rewind
irb(main):005:1> }
=> {:I1=>#<StringIO:0x0000000275cfe0>, :I2=>#<StringIO:0x00000002768f48>,
   :I3=>#<StringIO:0x00000002793478>, :I4=>#<StringIO:0x000000027c4ac8>}
```

On peut alors vérifier que les fichiers ont été correctement créés :

```
$ ls -al data/I*.dat
-rw-rw-r-- 1 jpe jpe 9856 avril 30 19:23 data/I1.dat
-rw-rw-r-- 1 jpe jpe 2888 avril 30 19:23 data/I2.dat
-rw-rw-r-- 1 jpe jpe  96 avril 30 19:23 data/I3.dat
-rw-rw-r-- 1 jpe jpe 2280 avril 30 19:23 data/I4.dat
```

Le débogueur peut aussi être utilisé pour afficher le contenu de la pile ou afficher les prochaines instructions qui vont être exécutées :

```
> show stack
[
  [0] : "-mark-",
  [1] "\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA"
]
> locate
rdebug
I1
32 exch
mark
1 index
resetfile
1 index
[...]
```

L'exécution jusqu'au prochain point d'arrêt peut être tracée :

```
> break index
> trace
[!] trace on
```

[illegible]

3.4.2 Traitement des données de I1

Le débogueur va être utile pour comprendre le fonctionnement de l'extrait de code Postscript ci-dessous qui constitue le début du traitement principal de la fonction `main` :

```

71 I1 % dépose I1 sur la pile
72 32 exch % dépose la valeur 32 puis échange avec I1
73 mark % dépose une marque sur la pile
74 1 index % récupère une référence à I1
75 resetfile % repositionne I1
76 1 index % récupère une référence à I1
77 {
78     counttomark % nombre d'opérandes sur la pile jusqu'à -mark-
79     1 sub % nombre d'opérandes - 1 (pointe sur I1)
80     index % dépose I1 sur la pile
81     counttomark % nombre d'opérandes sur la pile jusqu'à -mark-
82     2 add % nombre d'opérandes + 2 (pointe sur 32)
83     index % dépose 32 sur la pile
84     4 mul % 4 * 32 = 128
85     string % chaîne vide de 128 octets
86     readstring % stocke dans la chaîne vide 128 octets de I1
87     pop % enlève le résultat de readstring (true ou false)
88     dup % duplique la chaîne lue précédemment
89     () eq % teste si la chaîne est vide
90     {
91     pop
92     exit
93     }
94     if
95 }
96 loop

```

Avant le déroulement de l'instruction `loop`, `I1` et la valeur 32 sont déposées sur la pile.

L'instruction `loop` va alors dérouler le tableau présent sur la pile tant que l'instruction `exit` n'aura pas été exécutée. En posant un point d'arrêt au niveau de l'instruction `if`, il est possible d'inspecter l'état de la pile avant une potentielle sortie de boucle. La trace des deux premiers arrêts est présentée ci-dessous :

```

[!] break for rdebug
> break if
> continue
[!] break for if
> stack
[
  [0] :"-mark-",
  [1] "\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA",
  [2] 32,
  [3] #<StringIO:0x00000000cddc78>,
  [4] :"-mark-",
  [5] #<StringIO:0x00000000cddc78>,
  [6] "\xCFv\v\xC7}\xB1\xF2\x82\xE8\x81[...]\x1A\xC4\x92\xBD\r\v?/\xE3p",
  [7] false,
  [8] [
    [0] :pop,
    [1] :exit
  ]
]
> irb
1.9.3-p362 :001 > @ctx.stack[6].size
=> 128
1.9.3-p362 :002 > exit
> continue
[!] break for if
> stack
[
  [0] :"-mark-",
  [1] "\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA",
  [2] 32,
  [3] #<StringIO:0x00000000cddc78>,
  [4] :"-mark-",
  [5] #<StringIO:0x00000000cddc78>,
  [6] "\xCFv\v\xC7}\xB1\xF2\x82\xE8\x81[...]\x1A\xC4\x92\xBD\r\v?/\xE3p",
  [7] "\x9C\x9EQ\xC9\xD6\x94d\x81\x9F[...]\0"\xC8\xBC46\xCF/\x9B-\e",
  [8] false,
  [9] [
    [0] :pop,
    [1] :exit
  ]
]
]

```

On constate qu'à chaque itération, une nouvelle chaîne de 128 octets est déposée sur la pile. La comparaison avec la chaîne vide retourne `false`, l'instruction `exit` n'est pas exécutée et une nouvelle itération est réalisée.

A ce stade, on peut supposer que chaque itération va lire 128 octets de l'objet Postscript `-file-` (ou `StringIO` en Ruby) qui correspond à `I1`, la boucle `loop` s'arrêtant dès lors qu'il n'y aura plus de données à lire. Pour tester cette hypothèse, les 128 premiers octets de `I1` sont comparés à la première chaîne déposée sur la pile.

```

$ dd if=data/I1.dat count=128 bs=1 2>/dev/null | hexdump -C
00000000  cf 76 0b c7 7d b1 f2 82  e8 81 ed e9 a1 01 22 b2  |.v..}.....".|
00000010  20 88 74 66 b9 73 b8 54  21 8b 85 c2 30 d6 73 3a  |.tf.s.T!...0.s:|
00000020  b4 59 fd a9 a8 79 97 36  64 13 03 12 d5 ff 3e 1a  |.Y...y.6d....>.|
00000030  8e 2f 25 dc 6d 3a da 1d  af 6e 48 14 38 a3 a7 fe  |./%.m:...nH.8...|
00000040  8e 36 a7 7e 6b e0 ab 31  92 b6 a1 83 e0 2f 84 72  |.6.~k..1..../.r|
00000050  15 63 b9 58 6d 29 77 33  5d cd ba 80 53 09 94 c0  |.c.Xm)w3]...S...|
00000060  97 cd 05 4b af 0d 69 0a  b5 8e 65 76 c1 0a 6e 4d  |...K..i...ev..nM|
00000070  6f d2 00 05 62 1a 6e c4  92 bd 0d 0b 3f 2f e3 70  |o...b.n.....?/.p|

```

On retrouve bien le début (\xCFv) et la fin (\xE3p) de la première chaîne de la pile dans la sortie hexadécimale des 128 premiers octets de I1.

Pour continuer l'analyse, le point d'arrêt sur `if` est supprimé, un nouveau point d'arrêt est positionné après la boucle `loop` sur l'instruction `]` (ligne 104 de `script.ps`). Cette instruction va regrouper dans un tableau les opérandes présentes sur la pile jusqu'au premier objet de type `-mark-`.

```
> del if
> break ]
> continue
[!] break for ]
> step
[!] break for 4
> stack
[
  [0] : "-mark-",
  [1] "\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA",
  [2] 32,
  [3] #<StringIO:0x000000026ccdf0>,
  [4] #<StringIO:0x000000026ccdf0>,
  [5] [
    [ 0] "\xCFv\v\xC7}\xB1\xF2\x82\xE8\x81[...]\x1A\xC4\x92\xBD\r\v?/\xE3p",
    [ 1] "\x9C\x9EQ\xC9\xD6\x94d\x81\x9F[...]0"\xC8\xBC46\xCF/\x9B-\e",
    [...]
    [76] "P\xFB\x9F\xAB\xCEs\xCA\a\xD5\xD6[...] \x9B\x85\xC5\x8F"
  ]
]
```

On constate que la cinquième opérande sur la pile est un tableau de 77 chaînes, probablement de 128 octets chacune. On peut le vérifier avec un shell IRB, comme présenté ci-dessous :

```
> irb
irb(main):001:0> @ctx.stack[5].join.size
=> 9856
irb(main):002:0> @ctx.stack[5].join.size / 77
=> 128
irb(main):003:0> @ctx.stack[5].join == File.open("data/I1.dat", "rb").read
=> true
```

La concaténation des 77 chaînes et du contenu du fichier `I1.dat` sont identiques, notre précédente hypothèse est donc vérifiée.

Les instructions Postscript suivantes (`4 1 roll pop pop pop`) ne servent qu'à éliminer les opérandes déposées de façon temporaire sur la pile.

3.4.3 Déchiffrement des données de I2

Le code Postscript ci-dessous est ensuite exécuté (lignes 109 à 132) :

		Extrait de script.ps
109	I2	<i>% I2 sur la pile</i>
110	0 index	<i>% référence à I2 sur la pile</i>
111	resetfile	<i>% repositionne I2 au début des données</i>
112	61440 string	<i>% crée sur la pile une chaîne de 61440 octets</i>
113	readstring	<i>% lis le contenu de I2 dans cette chaîne</i>
114	pop	<i>% retire de la pile le résultat de readstring</i>
115	dup	<i>% duplique la chaîne contenant les données de I2</i>
116	3 index	<i>% copie une référence vers la clé K sur pile</i>
117	2 2 getinterval	<i>% dépose sur la pile une sous chaîne de la clé K avec le 3e et 4e octets</i>
118	dup	<i>% duplique la sous chaîne</i>

```

119  exch           % échange les deux sous chaînes
120  dup            % duplique la 2e sous chaîne
121  length         % dépose sur la pile la longueur de la sous chaîne (donc 2)
122  2 index        % dépose sur la pile une référence vers la première sous chaîne
123  length         % dépose sur la pile la longueur de la première sous chaîne (toujours 2)
124  add            % ajoute les longueurs des sous chaînes, 2 + 2 = 4
125  string         % dépose sur la pile une chaîne vide de 4 octets
126  dup            % duplique sur la pile la chaîne vide nouvellement créée
127  dup            % pareil
128  4 2 roll       % effectue une rotation des arguments sur la pile
129  copy           % copie les octets 3 et 4 de la clé au début de la chaîne destination
130  length         % retourne la longueur de la sous chaîne (2 octets)
131  4 -1 roll      % effectue une rotation des arguments sur la pile
132  putinterval    % copie les octets 3 et 4 de la clé à la fin de la chaîne destination

```

Après l'exécution de ces instructions, la pile contient 6 éléments qui sont :

- 1 : un objet de type `-mark-` ;
- 2 : une chaîne de caractères contenant les 16 octets de la clé K ;
- 3 : un tableau de 77 chaînes de caractères de 128 octets issues de la lecture des données de I1 ;
- 4 : les données de I2 ;
- 5 : une copie des données de I2 ;
- 6 : une chaîne de 4 octets contenant deux fois les octets 3 à 4 de la clé (c'est-à-dire K[2] K[3] K[2] K[3]).

Les instructions ci-dessous (lignes 133 à 142) sont alors exécutées :

```

133  0 0 1 1       % dépose 0 sur la pile et initialise la boucle for (2 itérations)
134  {
135      pop         % enlève le compteur de la boucle
136      2 index    % récupère la 3e opérande sur la pile
137      length     % dépose la longueur de cette opérande
138  }
139  for
140
141  exch           % inverse les deux longueurs
142  1 sub          % soustrait 1 à l'opérande sur la pile

```

Par rapport à l'étape précédente, 3 éléments ont été ajoutés sur la pile :

- 7 : la valeur 0 ;
- 8 : la longueur de l'extrait de clé, 4 ;
- 9 : la longueur des données de I2 - 1, soit 2887.

Ces trois valeurs sont utilisées comme paramètre à la prochaine boucle `for`. Il y a donc 722 itérations effectuées ($2888/4 = 722$).

Une exécution pas-à-pas à l'aide du débogueur permet de comprendre les traitements effectués par la boucle `for` entre les lignes 143 et 205. Avant la première itération, la chaîne de 4 octets correspondant à K[2] K[3] K[2] K[3] est présente sur la pile.

Le code ci-dessous extrait de I2 une sous chaîne de 4 octets à la position déterminée par le compteur de boucle.

```

144  3 copy
145  exch
146  length
147  getinterval
148  2 index
149  mark
150  3 1 roll
151  0 1 3 -1 roll

```

```
152 dup
153 length
154 1 sub
155 exch
156 4 1 roll
```

Ensuite, le script va générer un tableau de 4 octets en calculant un « ou »-exclusif octet par octet entre la première chaîne de 4 octets sur la pile et la sous-chaîne extraite de I2 précédemment.

```
157 {
158     dup
159     3 2 roll
160     dup
161     5 1 roll
162     exch
163     get
164     3 1 roll
165     exch
166     dup
167     5 1 roll
168     exch
169     get
170     xor
171     3 1 roll
172 }
173 for
174
175 pop
176 pop
177 ]
```

Le tableau précédemment obtenu est alors transformé en chaîne de caractères avec le code ci-dessous :

```
178 dup
179 length
180 string
181 0 3 -1 roll
182 {
183     3 -1 roll
184     dup
185     4 1 roll
186     exch
187     2 index
188     exch
189     put
190     1 add
191 }
192 forall
```

Enfin, la chaîne obtenue est enregistrée dans I2 à la position déterminée par le compteur de boucle. Cette chaîne est aussi déposée sur la pile pour l'itération suivante.

```
194 pop
195 4 -1 roll
196 dup
197 5 1 roll
198 3 1 roll
199 dup
200 4 1 roll
201 putinterval
```



```
202  exch
203  pop
```

Il s'agit donc d'un algorithme de déchiffrement appliqué sur les données de I2. Cet algorithme travaille sur des chaînes de 4 octets qui peuvent en réalité être converties en entiers de 32 bits.

La spécification de l'algorithme correspond au pseudo-code ci-dessous :

Algorithm 1 Boucle de déchiffrement

Require: $b0$ et $b1$ sont les deux octets de clé, $I2$ est un tableau de 722 entiers de 32 bits

```
function DECRYPT( $b0, b1, I2$ )
   $key \leftarrow b0 \mid b1 \ll 8 \mid b0 \ll 16 \mid b1 \ll 24$ 
  for  $i = 1 \rightarrow 722$  do
     $key = key \oplus I2[i]$ 
     $I2[i] = key$ 
  end for
end function
```

▷ \oplus : ou-exclusif

Pour comprendre quelle est la nature des données une fois déchiffrées, il faut s'intéresser à la suite du programme Postscript, notamment entre les lignes 207 à 215 :

```
207  0 1 1
208  {
209    pop
210    pop
211  }
212  for
213
214  cvx
215  exec
```

La boucle avec les deux instructions `pop` sert à retirer de la pile les opérandes positionnées temporairement pour la boucle de déchiffrement. L'instruction `cvx` est utilisée pour convertir une chaîne de caractères en un programme Postscript, l'instruction `exec` va ensuite exécuter le résultat de la conversion.

L'interpréteur Postscript en Ruby permet d'examiner l'état de la pile avant l'instruction `cvx` :

```
$ ruby-ps -c cvx script.ps `ruby -e 'print "A"*32`
[!] break for cvx
> stack
[
  [0] :"-mark-",
  [1] "\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA",
  [2] [
    [ 0] "\xCFv\v\xC7}\xB1\xF2\x82\xE8\x81[...]x1An\xC4\x92\xBD\r\v?/\xE3p",
    [ 1] "\x9C\x9EQ\xC9\xD6\x94d\x81\x9F[...]0"\xC8\xBC46\xCF/\x9B-\e",
    [...]
    [76] "P\xFB\x9F\xAB\xCEs\xCA\a\xD5\xD6[...] \x9B\x85\xC5\x8F"
  ],
  [3] "o2}f4a)\"?g:k3\"rV}Y}:~1o5n0[...]f}f8dW'"
]
> irb
1.9.3-p362 :001 > @ctx.stack[3].size
=> 2888
1.9.3-p362 :002 > exit
> step
[!] break for exec
> irb
```

```
1.9.3-p362 :001 > @ctx.stack[3]
=> [:o2]
```

L'instruction `cvx` a bien tenté d'analyser la chaîne présente sur la pile pour la convertir en programme. Les octets 3 et 4 de la clé n'étant pas ceux attendus, le déchiffrement de I2 n'a pas abouti à un code source Postscript correct : le programme converti ne contient qu'une instruction, `o2`.

Maintenant qu'une condition est connue sur le contenu déchiffré de I2, il est possible de mettre en place une attaque par force brute sur l'espace des clés (255 possibilités pour chacun des octets de clé en entrée).

Le script Ruby ci-dessous, disponible en annexe [A.3.3](#), implémente l'algorithme de déchiffrement et permet d'identifier les valeurs correctes de la clé à l'aide d'une attaque par force brute. La condition d'arrêt est que les données déchiffrées contiennent une instruction Postscript valide, par exemple `roll`.

```
#!/usr/bin/env ruby

encrypted = File.read(ARGV.first).unpack('L*')

a = 256.times.to_a
a.product(a).each do |b0, b1|
  key = [ b0, b1, b0, b1 ].pack('C*').unpack('L').first

  decrypted = encrypted.map {|x| key ^ x }.pack('L*')

  next unless decrypted.include?("roll")

  $stderr.puts "key = %2.2x%2.2x" % [ b0, b1 ]
  $stderr.puts decrypted

  exit(0)
end
```

L'exécution de ce script sur les données de I2 nous permet de déterminer les octets 3 à 4 de la clé :

```
$ ./bfi2.rb data/I2.dat
key = f7a8
20 dict begin /T [ 8#32732522170 8#35061733526 8#4410070333 [...]
```

Il est alors possible de relancer l'interpréteur avec les octets corrects :

```
$ ruby-ps -c exec script.ps `ruby -e 'print "A" * 4 + "f7a8" + "A" * 24`'
[!] break for exec
> stack
[
  [0] :"-mark-",
  [1] "\xAA\xAA\xF7\xA8\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA",
  [2] [
    [ 0] "\xCFv\vx7}\xB1\xF2\x82\xE8\x81[...]\x1An\xC4\x92\xBD\r\v?/\xE3p",
    [ 1] "\x9C\x9EQ\xC9\xD6\x94d\x81\x9F[...]\0"\xC8\xBC46\xCF/\x9B-\e",
    [...]
    [76] "P\xFB\x9F\xAB\xCEs\xCA\axD5xD6[...]\x9B\x85\xC5\x8F"
  ],
  [3] [
    [ 0] 20,
    [ 1] :dict,
    [ 2] :begin,
    [ 3] :"/T",
    [ 4] :"[",
    [...]
    [195] :bind,
```

```

    [196] :def
  ]
]

```

Cette fois ci, le déchiffrement s'est correctement effectué : un programme Postscript est présent sur la pile et va être exécuté par l'instruction `exec`. L'analyse de ce programme fait l'objet du chapitre suivant.

3.5 Analyse du programme I2

L'instruction `cvx`, telle qu'implémentée dans l'interpréteur Postscript en Ruby, stocke le résultat de la conversion dans un fichier comme présenté ci-dessous :

extrait d'interpreter.rb

```

746 def execute_cvx
747   arg = @stack.pop
748
749   File.open("cvx_#{@cvx_count}.ps", "w") do |f|
750     Postscript::PP.pp(arg, 4, f)
751   end
752   @cvx_count += 1
753
754   parser = Postscript::Parser.new(arg)
755
756   array = parser.parse_objects
757   @stack << array
758 end

```

Le fichier résultant `cvx_0.ps` a été renommé en `I2.ps` et est disponible en annexe [A.3.3](#).

Une recherche Google sur certaines séquences d'instructions caractéristiques contenues dans `I2.ps`, par exemple `/W 1 31 bitshift` ligne 41, retourne comme résultat le lien : http://www.cs.cmu.edu/~dst/Adobe/Gallery/pdf_sec-ps.txt. Ce document, disponible en annexe [A.3.3](#), implémente des algorithmes nécessaires au chiffrement de fichiers PDF et en particulier l'algorithme MD5. Dans ce document, la fonction MD5 est définie ainsi :

```

/md5 {
  20 dict begin

  % initialise a,b,c,d,x
  /a 16#67452301 def
  /b 16#efcdab89 def
  /c 16#98badcfe def
  /d 16#10325476 def
  /x 16 array def

  [...]

  16 string
  [ [ a b c d ] { 3 { dup -8 bitshift } repeat } forall ]

  [...]
} bind def

```

On retrouve un code similaire à la fin du fichier `I2.ps` :

```

/calc
{
  20 dict
  begin
  /a 1732584193 def

```

```

/b 4023233417 def
/c 2562383102 def
/d 271733878 def
/x 16 array
def

[...]

16 string
[
[ a b c d ]
{
  3
  {
    dup
    -8 bitshift
  }
  repeat
}
forall

[...]
}
bind
def

```

L'étude des différentes constantes présentes dans le fichier `I2.ps` permet de confirmer cette hypothèse. Le résultat du déchiffrement de `I2` commence par :

```
20 dict begin /T [ 8#32732522170 8#35061733526 8#4410070333
```

La notation `8#32732522170` est utilisée pour préciser la base d'une valeur entière, en l'occurrence 8 dans cet exemple. La commande ci-dessous permet de convertir l'entier en base 16 :

```
$ ruby -e "puts '32732522170'.to_i(8).to_s(16)"
d76aa478
```

La recherche de `0xd76aa478` dans Google retourne de nombreuses implémentations de l'algorithme MD5.

En conclusion, l'exécution du fichier `I2.ps` va donc associer une implémentation de MD5 à la fonction nommée `calc`.

3.6 Analyse du programme I4

3.6.1 Déchiffrement des données de I4

A la suite de l'exécution de `I2`, l'interpréteur retourne dans `script.ps` et va exécuter les instructions suivantes :

	Extrait de script.ps
216	<code>I3</code> <i>% I3 sur la pile</i>
217	<code>resetfile</code> <i>% repositionne I3 au début des données</i>
218	<code>I4</code> <i>% I4 sur la pile</i>
219	<code>0 index</code> <i>% référence à I4 sur la pile</i>
220	<code>resetfile</code> <i>% repositionne I4 au début des données</i>
221	<code>61440 string</code> <i>% crée sur la pile une chaîne de 61440 octets</i>
222	<code>readstring</code> <i>% lis le contenu de I4 dans cette chaîne</i>
223	<code>pop</code> <i>% retire de la pile le résultat de readstring</i>
224	<code>dup</code> <i>% duplique la chaîne contenant les données de I4</i>
225	<code>3 index</code> <i>% copie une référence vers la clé K sur pile</i>
226	<code>0 2 getinterval</code> <i>% dépose sur la pile une sous chaîne de la clé K avec le 1er et 2e octets</i>

```
227 dup           % duplique la sous chaîne
228 exch          % échange les deux sous chaînes
```

On retrouve une séquence similaire à celle qui a servi pour le déchiffrement de I2 avec les exceptions suivantes :

- I3 est déposé sur la pile pour le repositionner au début de ses données ;
- I4 est déposé sur la pile à la place de I2 ;
- les deux premiers octets de la clé sont déposés sur la pile (0 2 getinterval) à la place des octets 3 et 4.

En parcourant la suite du code, on retrouve bien entre les lignes 252 à 313 l'implémentation de la routine de déchiffrement utilisée précédemment.

Il est alors possible de réutiliser le script décrit à la section 3.4.3 pour identifier les deux premiers octets de la clé.

```
$ ./bfi2.rb data/I4.dat
bac9
0 0 0 0 2 2 16 4 sub { 6 index exch 4 getinterval
```

Les valeurs attendues pour les deux premiers octets sont donc 0xba 0xc9.

Comme dans le cas de I2, le déchiffrement de I4 est suivi des instructions cvx exec pour convertir les données déchiffrées en un programme Postscript qui sera alors exécuté.

On peut alors lancer ruby-ps avec les deux octets de clé identifiés pour inspecter le contenu de la pile avant l'appel à l'instruction exec :

```
$ ruby-ps -c exec script.ps `ruby -e 'print "bac9f7a8" + "A" * 24'`
[!] break for exec
> continue
[!] break for exec
> stack
[
  [0] : "-mark-",
  [1] "\xBA\xC9\xF7\xA8\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA",
  [2] [
    [ 0] "\xCFv\v\xC7}\xB1\xF2\x82\xE8\x81[...]\x1A\xC4\x92\xBD\r\v?/\xE3p",
    [ 1] "\x9C\x9EQ\xC9\xD6\x94d\x81\x9F[...]\x0"\xC8\xBC46\xCF/\x9B-\e",
    [...]
    [76] "P\xFB\x9F\xAB\xCEs\xCA\axD5\xD6[...]\x9B\x85\xC5\x8F"
  ],
  [3] [
    [ 0] 0,
    [ 1] 0,
    [ 2] 0,
    [ 3] 0,
    [ 4] 2,
    [ 5] 2,
    [ 6] 16,
    [ 7] 4,
    [ 8] :sub,
  ]
]
```

La conversion des données déchiffrées de I4 par l'instruction cvx a produit un fichier cvx_1.ps. Ce fichier a été renommé en I4.ps et est disponible à l'annexe A.3.4.

Les quatre premiers octets de la clé ont été identifiés. Il faut maintenant comprendre le fonctionnement de I4.ps pour trouver les 12 octets manquants. Pour simplifier l'étude de I4.ps, le fichier script.ps a été modifié de la façon suivante : les instructions cvx exec ont été remplacées par pop (data/I4.ps) run. Ainsi, il est possible de modifier le code de I4, notamment pour ajouter l'instruction rdebug afin de déclencher le débogueur.

3.6.2 Boucle principale

Le corps du script I4.ps est composé de deux boucles imbriquées dont la principale est commentée ci-dessous :

```
% dépose quatre 0 sur la pile
0 0 0 0
% initialise la boucle avec un compteur de 2 à 12, incrément de 2 (6 itérations)
2 2 16 4 sub {
  % pose sur la pile une référence sur la clé
  6 index
  % échange la référence de la clé avec le compteur de boucle
  exch
  % récupère 4 octets de la clé à la position spécifiée par le compteur
  4 getinterval
  % répète la boucle secondaire 10240 fois
  10240 { <<boucle secondaire>> } repeat
  % enlève de la pile une opérande temporaire de la boucle
  pop

  % dépose sur la pile une référence à I1 (tableau de 77 chaînes de 128 octets)
  4 index
  % calcule la longueur totale de I1
  0 1 index { length add } forall

  % initialise une chaîne vide de la longueur des données de I1
  string
  % concatène les données de I1 dans la nouvelle chaîne
  0 3 2 roll { 3 copy putinterval length add } forall
  pop

  % calcule le MD5 de la concaténation des données de I1
  calc
  % lis 16 octets depuis I3 et les stocke dans une chaîne
  I3 16 string readstring pop
  % compare les deux chaînes
  ne
  {
    % la comparaison a échoué, boucle de temporisation
    0 1 1073741823 { pop } for

    (Key is invalid. Exiting ...\n) error
    flush
    quit
  } if
} for

% supprime les opérandes temporaires de la pile
pop pop pop pop
% ouvre le fichier output.bin en écriture
(output.bin) (w) file
exch
1 index
resetfile
% écrit les données de I1 dans output.bin
{ 1 index exch writestring } forall

closefile
```

Avant l'exécution de I4, la pile de l'interpréteur contient les trois opérandes suivantes :

- un objet de type `-mark-` ;
- la clé passée en argument au script ;
- un tableau contenant 77 chaînes de 128 caractères qui correspondent aux données de I1.

La boucle principale de I4 effectue alors les traitements suivants :

- récupération de 4 octets de la clé, de la position 2 jusqu'à la position 12 (la première itération va extraire K[2] K[3] K[4] K[5], la suivante K[4] K[5] K[6] K[7], etc) ;
- effectue 10240 itérations de la boucle secondaire ;
- concatène les données de I1 dans une chaîne et calcule l'empreinte MD5 de cette chaîne ;
- compare l'empreinte MD5 obtenue avec 16 octets de I3 ;
- quitte le programme si la comparaison échoue.

On peut en conclure que I3 constitue une liste d'empreintes MD5 qu'il est possible d'afficher avec la commande ci-dessous :

```
irb(main):001:0> md5 = File.read("data/I3.dat").bytes.each_slice(16).to_a.map { |x|
irb(main):002:1*   x.pack('C*').unpack('H*').first
irb(main):003:1> }
=> ["338f25667eb4ec47763dab51c3fa41cb", "a329e18536b83159b3a690a0265ec519",
    "aae94f0e715376c4f087bccdd0be3b4", "a114f8be746142c44978faa76dae62cf",
    "197d7bce4eb38dd68c8ce5f69f326e1e", "ffceae3f72f8eaa38e019a59b1dc0997"]
```

On peut également supposer que les 10240 itérations de la boucle secondaire effectuent des modifications sur les données de I1, en fonction des 4 octets extraits de la clé principale. L'objectif à ce stade est de comprendre la nature de ces modifications afin de déterminer quelles sont les valeurs que doivent prendre les 4 octets de clé pour que l'empreinte MD5 calculée sur les données de I1 correspondent à celle lue dans I3.

3.6.3 Boucle secondaire

A l'entrée d'une itération, quatre valeurs entières sont présentes sur la pile ainsi qu'une chaîne de quatre octets. Par la suite, ces quatre entiers seront identifiés par a, b, c, d et la chaîne par S. En ajoutant l'instruction `rdebug` à l'entrée de la boucle ligne 8, il est possible d'inspecter l'état de la pile :

```
$ ruby-ps script.ps `ruby -e 'print "bac9f7a8" + "A" * 24'`
[!] break for rdebug
> stack
[
  [0] : "-mark-",
  [1] "\xBA\xC9\xF7\xA8\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA\xAA",
  [2] [
    [ 0] "\xCF\v\v\xC7}\xB1\xF2\x82\xE8\x81[...]\x1A\xC4\x92\xBD\r\v?/\xE3p",
    [ 1] "\x9C\x9EQ\xC9\xD6\x94d\x81\x9F[...]0"\xC8\xBC46\xCF/\x9B-\e",
    [...]
    [76] "P\xFB\x9F\xAB\xCEs\xCA\a\xD5\xD6[...] \x9B\x85\xC5\x8F"
  ],
  [3] 0,
  [4] 0,
  [5] 0,
  [6] 0,
  [7] "\xF7\xA8\xAA\xAA"
]
```

On retrouve les entiers a, b, c et d initialisés à 0 ainsi que l'extrait de clé K[2..5] = `\xF7\xA8\xAA\xAA` = S.

La boucle commence par convertir S en un entier de 32 bits identifié par T par la suite.

```
8 0 0 1 3      % initialise T à 0 et les paramètres de la boucle à 0 1 3
9 {
10 3 -1 roll   % S en haut de la pile
```

```

11     dup          % duplique S
12     4 1 roll    % T en haut de la pile
13     exch        % inverse T et S
14     get         % récupère un octet de S
15     exch        % échange T et l'octet courant
16     8 bitshift  % décale de 8 la valeur de T
17     add         % ajoute la valeur de l'octet courant à T
18 }
19 for
20
21 exch
22 pop

```

Ensuite, des transformations sont effectuées sur la valeur de T par les instructions ci-dessous :

Extrait de I4.ps

```

23 dup          % duplique T
24 -2 bitshift  % T >> 2
25 exch        % inverse T et T >> 2
26 dup          % duplique T
27 -3 bitshift  % T >> 3
28 1 index     % dépose une référence de T sur la pile
29 -7 bitshift  % T >> 7
30 xor         % (T >> 7) ^ (T >> 3)
31 exch        % T en haut de la pile
32 dup          % duplique T
33 4 1 roll    % rotation de 4 opérantes vers le haut de la pile
34 xor         % T ^ (T >> 7) ^ (T >> 3)
35 xor         % (T >> 2) ^ T ^ (T >> 7) ^ (T >> 3)
36 1 and       % ( (T >> 2) ^ T ^ (T >> 7) ^ (T >> 3) ) & 1
37 31 bitshift  % ( (T >> 2) ^ T ^ (T >> 7) ^ (T >> 3) ) & 1 << 31
38 exch        % T en haut de la pile
39 -1 bitshift  % T >> 1
40 or          % (T >> 1) | ( (T >> 2) ^ T ^ (T >> 7) ^ (T >> 3) ) & 1 << 31

```

On peut reconnaître un registre à décalage à rétroaction linéaire (ou LFSR en anglais pour « linear feedback shift register ». Les calculs effectués sont :

$$b = (T \oplus (T \gg 2) \oplus (T \gg 3) \oplus (T \gg 7)) \& 1$$

$$T = (T \gg 1) | (b \ll 31)$$

L'entier T est alors converti en chaîne de 4 octets avec les instructions ci-dessous :

Extrait de I4.ps

```

41 4 string     % S = nouvelle chaîne vide de 4 octets
42 exch        % inverse T et S
43 3 -1 0
44 {
45     3 copy    % copie 3 opérantes sur la pile
46     exch     % inverse les 2 opérantes
47     255 and  % b = T & 0xff
48     put      % S[i] = b
49     pop
50     -8 bitshift % T >>= 8
51 }
52 for
53
54 pop

```

Le résultat de la conversion devient la nouvelle valeur de la chaîne S.

S est alors comparée avec deux autres chaînes par les instructions ci-dessous :


```

55 dup
56 (UUUU) le
57 {
58     1
59 }
60 {
61     dup
62     <aaaaaaaa> le
63     {
64         -1
65     }
66     {
67         0
68     }
69     ifelse
70 }
71 }
72 ifelse

```

Si *S* est inférieure à 'UUUU' alors 1 est déposé sur la pile. Sinon, si *S* à \xaa\xaa\xaa\xaa, -1 est déposé sur la pile. Dans les autres cas, 0 est déposé sur la pile. Le résultat de cette comparaison est référencé par *r* par la suite

Les instructions suivantes effectuent des calculs sur la valeur de *c* :

```

74 4 -1 roll % déplace c en haut de la pile
75 add      % additionne c et r
76 5 index  % dépose une référence à I1 sur la pile
77 length   % dépose le nombre d'éléments de I1 sur la pile, soit 77
78 add      % ajoute 77 à l'addition précédente
79 5 index  % dépose une référence à I1 sur la pile
80 length   % dépose le nombre d'éléments de I1 sur la pile, soit 77
81 mod      % calcule c + r + 77 % 77
82 3 1 roll % repositionne c dans la pile

```

Les lignes 84 à 148 effectuent les mêmes traitements sur *S*, à savoir :

- conversion de la chaîne de caractères *S* en entier *T* ;
- mise à jour de *T* avec le registre à décalage à rétroaction linéaire ;
- conversion de *S* en chaîne de caractères depuis *T* ;
- comparaison de *S* avec 'UUUU' et \xaa\xaa\xaa\xaa.

Les instructions ci-dessous sont alors exécutées pour mettre à jour la valeur de *d* :

```

149 3 -1 roll % déplace d en haut de la pile
150 add      % additionne d + r (résultat de la comparaison des chaînes)
151 5 index  % dépose une référence à I1 sur la pile
152 0 get     % dépose le premier élément de I1 sur la pile
153 length   % dépose la longueur du premier élément de I1, soit 128
154 4 idiv    % 128 / 4 = 32
155 add      % d + r + 32
156 5 index  % dépose une référence à I1 sur la pile
157 0 get     % dépose le premier élément de I1 sur la pile
158 length   % dépose la longueur du premier élément de I1, soit 128
159 4 idiv    % 128 / 4 = 32
160 mod      % d + r + 32 % 32
161 exch     % repositionne d dans la pile

```

Les instructions ligne 162 à 208 mettent à jour la valeur de S avec le registre à décalage à rétroaction linéaire. La suite du programme est présentée ci-dessous :

	Extrait de I4.ps	
209	6 -2 roll	% positionne I1 et a en haut de la pile
210	2 copy	% duplique I1 et a
211	8 2 roll	% repositionne les opérandes de la pile
212	get	% I1[a]
213	4 index	% b en haut de la pile
214	4 mul	% 4 * b
215	7 index	% dépose une référence sur I1 en haut de la pile
216	5 index	% c en haut de la pile
217	get	% I1[c]
218	4 index	% d en haut de la pile
219	4 mul	% 4d
220	4 5 copy	% dépose 4 sur la pile et duplique 5 opérandes
221	dup	% duplique 4
222	4 1 roll	% rotation vers le haut de 4 opérandes
223	getinterval	% extrait 4 octets de I1[c] à la position 4d = I1[c][4d, 4]
224	4 1 roll	% rotation vers le haut de 4 opérandes
225	getinterval	% extrait 4 octets de I1[a] à la position 4b = I1[a][4b, 4]
226	exch	% échange I1[c][4d, 4] et I1[a][4b, 4]
227	dup	% duplique I1[c][4d, 4]
228	length	% longueur de I1[c][4d, 4] soit 4
229	string	% nouvelle chaîne vide de 4 octets
230	0 3 -1 roll	% dépose 0 sur la pile puis rotation vers le bas de 3 opérandes
231	{	% pour chaque octet de I1[c][4d, 4]
232	3 copy	% duplique 3 opérandes
233	put	% copie un octet dans la nouvelle chaîne
234	pop	
235	1 add	% incrémente la position de destination
236	}	
237	forall	
238		
239	pop	

Ces instructions vont donc extraire une sous-chaîne de 4 octets à la position 4b de l'élément a de I1. De plus, la sous-chaîne de 4 octets à la position 4d de l'élément c de I1 est sauvegardée dans une nouvelle chaîne.

Les instructions suivantes sont exécutées :

	Extrait de I4.ps	
209	exch	% échange I1[a][4b, 4] et la copie de I1[c][4d, 4]
210	3 -1 roll	% rotation vers le bas de 3 opérandes
211	pop	% enlève 4 de la pile
212	4 -2 roll	% rotation vers le bas de 4 opérandes
213	3 -1 roll	% rotation vers le bas de 3 opérandes
214	putinterval	% écrit dans I1[c] à la position 4d la valeur de I1[a][4b, 4]
215	putinterval	% écrit dans I1[a] à la position 4b la valeur de la copie de I1[c][4d, 4]
216	5 index	% référence à I1 sur la pile
217	5 index	% a sur la pile
218	get	% I1[a]
219	4 index	% b sur la pile
220	4 mul	% 4b
221	4 getinterval	% I1[a][4b, 4] sur la pile
222	1 index	% référence à S sur la pile
223	0 0 1 1	% dépose 0 sur la pile
224	{	% 2 itérations
225	pop	% retire le compteur de boucle
226	2 index	% référence sur la 3e opérande de la pile
227	length	% longueur de l'opérande
228	}	

```

229 for
230
231     exch           % inverse les 2 longueurs (4 chacune)
232     1 sub         % 4 - 1 = 3

```

Une permutation est donc réalisée entre `I1[a][4b, 4]` et `I1[c][4d, 4]`, ce qui nécessite l'usage d'une chaîne temporaire. La boucle `for` entre les lignes 264 et 325 effectue une seule itération. Celle-ci commence par les instructions suivantes :

```

                                Extrait de I4.ps
265 3 copy           % duplique 3 opérandes de la pile : I1[a][4b, 4], S et 0
266 exch            % échange S et 0
267 length          % longueur de S soit 4
268 getinterval     % extrait une chaîne de 4 octets de I1[a][4b, 4] à la position 0
269 2 index         % référence à S sur la pile
270 mark            % dépose une marque sur la pile
271 3 1 roll        % rotation vers le haut de 3 opérandes
272 0 1 3 -1 roll   % 0 et 1 sur la pile puis rotation vers le bas de 3 opérandes
273 dup             % duplique S sur la pile
274 length          % longueur de S sur la pile, soit 4
275 1 sub           % 4 - 1 = 3
276 exch           % échange S et 3
277 4 1 roll        % rotation vers le haut de 4 opérandes (0 1 3 en haut de la pile)
278 {               % 4 itérations (i = 0 à 3)
279     dup          % duplique le compteur de boucle i
280     3 2 roll    % rotation vers le haut de 3 opérandes
281     dup         % duplique S
282     5 1 roll    % rotation vers le haut de 5 opérandes
283     exch        % échange le compteur de boucle i et S
284     get         % S[i]
285     3 1 roll    % rotation vers le haut de 3 opérandes
286     exch        % échange le compteur de boucle i et I1[a][4b, 4]
287     dup         % duplique I1[a][4b, 4]
288     5 1 roll    % rotation vers le haut de 5 opérandes
289     exch        % échange le compteur de boucle i et I1[a][4b, 4]
290     get         % I1[a][4b, 4][i]
291     xor         % S[i] ^ I1[a][4b, 4][i]
292     3 1 roll    % rotation vers le haut de 3 opérandes
293 }
294 for
295
296 pop             % suppression d'une opérande temporaire
297 pop             % suppression d'une opérande temporaire
298 ]               % création d'un tableau avec les 4 octets calculés dans la boucle

```

Ces instructions vont donc produire un tableau de 4 octets qui correspondent au calcul d'un ou-exclusif entre `S` et `I1[a][4b, 4]`, octet par octet. Le programme continue avec les instructions suivantes :

```

                                Extrait de I4.ps
299 dup            % duplication du tableau de 4 octets
300 length         % longueur du tableau, soit 4
301 string         % nouvelle chaîne vide 4 octets
302 0 3 -1 roll    % 0 sur la pile puis rotation vers le bas de 3 opérandes
303 {              % pour chaque octet du tableau
304     3 -1 roll   % rotation vers le bas de 3 opérandes
305     dup        % duplique la nouvelle chaîne
306     4 1 roll   % rotation vers le haut de 4 opérandes
307     exch       % échange l'octet courant et la nouvelle chaîne
308     2 index    % référence vers la position sur la pile
309     exch       % échange la position et l'octet courant
310     put        % écrit dans la nouvelle chaîne l'octet courant à la position spécifiée
311     1 add     % incrémente la position
312 }

```

```

313 forall
314
315 pop           % supprime une opérande temporaire
316 4 -1 roll    % rotation vers le bas de 4 opérandes
317 dup          % duplique I1[a][4b, 4]
318 5 1 roll     % rotation vers le haut de 5 opérandes
319 3 1 roll     % rotation vers le haut de 3 opérandes
320 dup          % duplique la chaîne précédemment créée
321 4 1 roll     % rotation vers le haut de 4 opérandes
322 putinterval % écrit dans I1[a][4b, 4] la chaîne précédemment créée
323 exch         % échange S et la chaîne précédemment créée
324 pop          % supprime une référence S de la pile

```

Le contenu de la sous-chaîne I1[a][4b, 4] est donc mis à jour en effectuant un ou-exclusif octet par octet avec la chaîne S.

Enfin, l'itération secondaire se termine par les instructions ci-dessous :

Extrait de I4.ps

```

328 pop          % supprime une opérande temporaire
329 pop          % supprime une opérande temporaire
330 5 1 roll     % rotation vers le haut de 5 opérandes
331 2 copy       % duplique c et d
332 7 -3 roll    % rotation vers le bas de 7 opérandes
333 pop          % enlève a de la pile
334 pop          % enlève b de la pile

```

Ces instructions préparent la nouvelle itération, en remplaçant a par c et b par d. a et b sont donc les valeurs sauvegardées des entiers c et d.

On peut remarquer que toutes les opérations sont effectuées sur des chaînes de 4 octets qui peuvent être considérées comme des représentations big-endian d'entiers 32 bits. En effectuant cette transformation, le déroulement de la boucle secondaire peut alors être résumé par l'algorithme 2.

Algorithm 2 Boucle secondaire de I4

Require: T est un entier de 32 bits

function LFSR(T)

$b \leftarrow (T \oplus (T \gg 2) \oplus (T \gg 3) \oplus (T \gg 7)) \& 1$

$\triangleright \oplus$: ou-exclusif

$T \leftarrow (T \gg 1) | (b \ll 31)$

return T

end function

Require: T est un entier de 32 bits

function CMP(T)

if $T < 0x55555555$ **then**

return 1

else

if $T < 0xaaaaaaaa$ **then**

return -1

else

return 0

end if

end if

end function

Require: k est une chaîne de 4 octets, c est un entier compris entre 0 et 76, d est un entier compris entre 0 et 31, $I1$ est un tableau contenant 77 sous-tableaux de 32 entiers

function DECRYPT($k, c, d, I1$)

$T \leftarrow k[0] \ll 24 | k[1] \ll 16 | k[2] \ll 8 | k[3]$

for $i = 1 \rightarrow 10240$ **do**

$a \leftarrow c$

$b \leftarrow d$

$T \leftarrow \text{LFSR}(T)$

$c \leftarrow c + \text{CMP}(T) + 77 \bmod 77$

$T \leftarrow \text{LFSR}(T)$

$d \leftarrow d + \text{CMP}(T) + 32 \bmod 32$

$T \leftarrow \text{LFSR}(T)$

$\text{tmp} \leftarrow I1[c][d]$

$I1[c][d] \leftarrow I1[a][b]$

$I1[a][b] \leftarrow \text{tmp} \oplus \text{Hton}(T)$

$\triangleright \oplus$: ou-exclusif

end for

end function

L'appel à *Hton* à la fin de la boucle est nécessaire pour s'assurer que le résultat soit bien stocké sous forme big-endian dans le tableau *I1*.

3.7 Attaque par force brute

La compréhension de l'algorithme de déchiffrement permet de mettre en place une attaque par force brute. A partir de la description de la boucle principale de I4 au chapitre 3.6.2, les conditions d'arrêt ci-dessous peuvent être formulées :

Itération	Extrait de clé à tester	Empreinte MD5 à vérifier
1	K[2] K[3] K[4] K[5]	338f25667eb4ec47763dab51c3fa41cb
2	K[4] K[5] K[6] K[7]	a329e18536b83159b3a690a0265ec519
3	K[6] K[7] K[8] K[9]	aae94f0e715376c4f087bccdd0be3b4
4	K[8] K[9] K[10] K[11]	a114f8be746142c44978faa76dae62cf
5	K[10] K[11] K[12] K[13]	197d7bce4eb38dd68c8ce5f69f326e1e
6	K[12] K[13] K[14] K[15]	ffceae3f72f8eaa38e019a59b1dc0997

TABLE 3.1 – Conditions d'arrêt

A la fin des six itérations, chaque octet de la clé va donc être utilisé au moins une fois. On peut également constater un recouvrement de deux octets au niveau de l'extrait de clé à tester entre deux itérations. De fait, une itération ne permettra de déterminer que les deux derniers octets, les deux premiers ayant été validés par l'itération précédente.

Avant la première itération, les quatre premiers octets $K[0]$ $K[1]$ $K[2]$ $K[3]$ ont déjà pu être identifiés grâce au déchiffrement des données de I2 et I4 comme décrit aux chapitres 3.4.3 et 3.6.1. Le principe de l'attaque par force brute, appliqué à la première itération, est alors de tester toutes les possibilités pour $K[4]$ et $K[5]$ (de 0 à 255) jusqu'à obtenir l'empreinte MD5 à vérifier sur les données déchiffrées de I1. La détermination de $K[4]$ et $K[5]$ permet ensuite d'identifier $K[6]$ et $K[7]$ grâce à la seconde itération. Ainsi, il est possible d'obtenir les 12 octets manquants de la clé à la fin des six itérations.

Quelques précautions sont néanmoins à respecter pour que l'attaque réussisse :

- chaque test de déchiffrement réalisé sur deux octets candidats va modifier le contenu de I1. En cas d'échec, le test suivant doit réutiliser le dernier état correct de I1 et non pas les données modifiées par le test précédent ;
- les données de I1 déchiffrées par une itération sont utilisées en entrée par l'itération suivante. De façon similaire, les index c et d , utilisés par l'algorithme de déchiffrement 2, ne sont pas réinitialisés entre deux itérations et doivent donc être conservés tels quels.

Le fichier `solve-part3.c`, disponible à l'annexe A.3.5, implémente cette attaque par force brute. L'algorithme 2 est implémenté par la fonction `decrypt_I4` présentée ci-dessous :

```
void decrypt_I4(struct bf_ctx *ctx, uint8_t *key, uint8_t *md5sum) {
    int i;
    uint32_t t, tmp;
    MD5_CTX md5_ctx;
    uint32_t **data;
    uint8_t a, b, old_a, old_b;

    t = key[0] << 24 | key[1] << 16 | key[2] << 8 | key[3];

    data = ctx->data;
    a = ctx->a;
    b = ctx->b;

    for (i = 0; i < 10240; i++) {
        old_a = a;
        old_b = b;

        t = lfsr(t);
        a = (a + cmp(t) + DATA_COUNT) % DATA_COUNT;

        t = lfsr(t);
        b = (b + cmp(t) + DATA_LEN) % DATA_LEN;

        t = lfsr(t);

        /* Swapping data[c][d] and data[a][b] */
        tmp = data[a][b];
        data[a][b] = data[old_a][old_b];

        data[old_a][old_b] = tmp ^ htonl(t);
    }

    ctx->a = a; ctx->b = b;

    MD5_Init(&md5_ctx);
    for (i = 0; i < DATA_COUNT; i++) {
        MD5_Update(&md5_ctx, ctx->data[i], sizeof(uint32_t) * DATA_LEN);
    }
    MD5_Final(md5sum, &md5_ctx);
}
```

```
}
```

La compilation du programme nécessite les fichiers `md5.c` et `md5.h`, également disponibles à l'annexe [A.3.5](#). La ligne de commande ci-dessous permet de compiler le programme :

```
$ gcc -O3 -march=native -fomit-frame-pointer -fopenmp -pedantic -o solve-part3 solve-part3.c md5.c
```

Le programme utilise OpenMP pour paralléliser le déroulement de la boucle sur le premier octet de clé à identifier, ce qui permet d'accélérer la résolution. Ainsi, la clé complète est identifiée en une dizaine de secondes sur une machine récente avec quatre coeurs. Le résultat du déchiffrement final est alors affiché sur la sortie standard.

```
$ time ./solve-part3 script.ps > part4.vcard
[*] solving part 3
[+] hollywood mode engaged
[+] starting 4 threads
[!] key = bac9f7a8721fad3c9fcf271eed9abbc8
./solve-part3 script.ps > part4.vcard  41,88s user 0,08s system 381% cpu 10,986 total
$ file out
out: vCard visiting card
```

Chapitre 4

Décodage du contenu de la vCard

Le fichier obtenu au chapitre précédent contient une liste de contacts au format vCard. Une entrée en particulier attire l'attention :

```
BEGIN:VCARD
VERSION:2.1
FN:Challenge SSTIC
N:Challenge;SSTIC
ADR;WORK;PREF;QUOTED-PRINTABLE;;Campus Beaulieu;Rennes
TEL;CELL:
EMAIL;INTERNET:sys_socketpair stub_fork sys_socketpair [...] sys_getpgrp sys_setregid sys_syslog
END:VCARD
```

L'adresse email du contact a été remplacée par une liste d'appels systèmes Unix. Chaque appel système peut alors être remplacé par sa valeur numérique, qu'il est possible de retrouver dans un fichier d'entête. Par exemple, le fichier `/usr/include/x86_64-linux-gnu/asm/unistd_64.h` contient les définitions suivantes :

```
#define __NR_socketpair          53
__SYSCALL(__NR_socketpair, sys_socketpair)
#define __NR_setsockopt          54
__SYSCALL(__NR_setsockopt, sys_setsockopt)
#define __NR_getsockopt          55
__SYSCALL(__NR_getsockopt, sys_getsockopt)
```

Le script `solve-part4.rb`, disponible à l'annexe [A.4](#), permet alors d'extraire du fichier vCard la liste des appels systèmes pour générer une liste d'entiers qui correspondent à la valeur numérique de chaque appel. Cette liste d'entiers peut alors être considérée comme une série d'octets d'une chaîne de caractères.

Le corps du script est présenté ci-dessous :

```
b = {}
headers_data.each_line do |line|
  if line =~ /define __NR_(^[^s]+\s+(\d+)) / then
    b["sys_#{1}"] = $2.to_i
    b["stub_#{1}"] = $2.to_i
  end
end

a = syscalls.map {|x| b[x]}
raise if a.include? nil

email = a.pack('C*')
$stderr.puts "[!] email: #{email}"
```

Son exécution sur le fichier vCard permet finalement de retrouver l'adresse email de validation du challenge :

```
$ ./solve-part4.rb part4.vcard
[*] solving part 4
[+] found syscall definitions at /usr/include/x86_64-linux-gnu/asm/unistd_64.h
[!] email: 59575e0e71f1e3e9946bc307fc7a608d0b568458@challenge.sstic.org
```

Chapitre 5

Conclusion

5.1 Synthèse

Les scripts et programmes développés dans le cadre de ce challenge ont été regroupés dans l'archive `solve-sstic2013.tar.bz2` disponible à l'annexe [A.5](#).

L'archive contient les fichiers suivants :

```
$ tar jxvf solve-sstic2013.tar.bz2
solve-sstic2013/
solve-sstic2013/solve-sstic2013.sh
solve-sstic2013/solve-part4.rb
solve-sstic2013/solve-part2.c
solve-sstic2013/solve-part3.c
solve-sstic2013/md5.h
solve-sstic2013/Makefile
solve-sstic2013/solve-part1.rb
solve-sstic2013/md5.c
```

La résolution s'effectue alors avec les commandes suivantes :

```
$ time ./solve-sstic2013.sh dump.bin
[+] correct md5sum for dump.bin
[*] solving part 1
[+] iv = 76C128D46A6C4B15B43016904BE176AC
[+] target md5sum = 61c9392f617290642f9a12499de6b688
[+] md5sum(r) = 61c9392f617290642f9a12499de6b688
[!] key = dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94
[*] solving part 2
[!] key = e683dcbc16ef58c665ac23d31e6da125
[*] solving part 3
[+] hollywood mode engaged
[+] starting 4 threads
[!] key = bac9f7a8721fad3c9fcf271eed9abbc8
[*] solving part 4
[+] found syscall definitions at /usr/include/x86_64-linux-gnu/asm/unistd_64.h
[!] email: 59575e0e71f1e3e9946bc307fc7a608d0b568458@challenge.sstic.org
./solve-sstic2013.sh dump.bin 44,12s user 0,20s system 347% cpu 12,761 total
```

La résolution des quatre parties du challenge permet finalement de retrouver l'adresse email de validation.

L'ensemble des développements réalisés dans le cadre de ce challenge seront disponibles dès la fin de l'édition 2013 du SSTIC à l'adresse <https://github.com/nieluj/sstic2013>.

5.2 Remerciements

Je tiens à remercier les auteurs de ce challenge, ainsi que l'ensemble du comité d'organisation du SSTIC. Le crû de cette année fut d'une excellente facture et m'a permis de découvrir le monde fabuleux des circuits programmables qui m'était resté inconnu jusqu'à présent.

Annexe A

Annexes

A.1 Annexe : fichier `dump.bin`

Enregistrer `solve-part1.rb` :

A.2 Annexe : FPGA

A.2.1 Étude du schéma logique

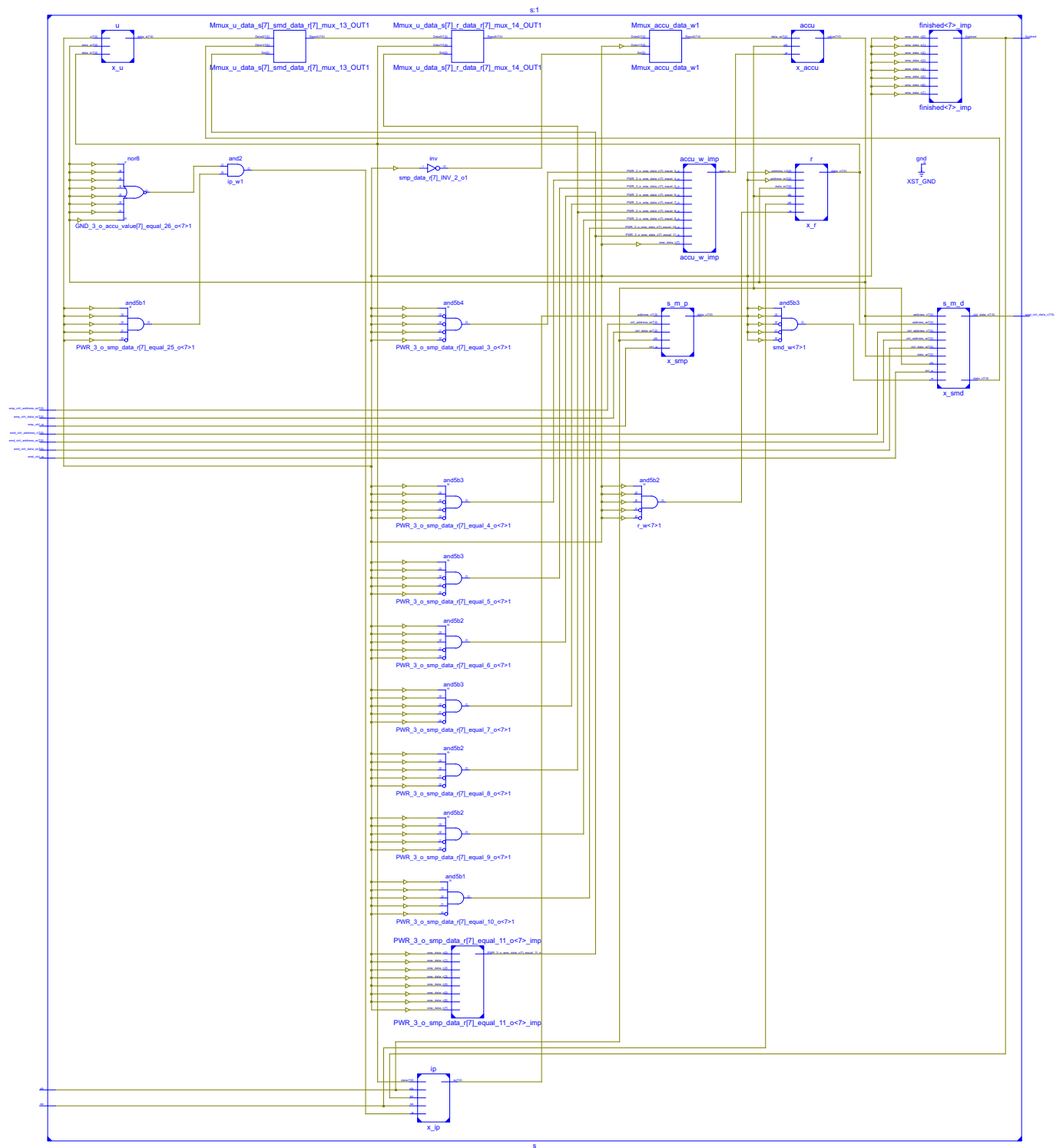


FIGURE A.1 – Bloc S développé

A.2.2 Rétro-ingénierie

Enregistrer metasm-sstic2013-ext-0.0.1.gem :

Enregistrer `smp.asm` :

Enregistrer `smp-deobfusc.asm` :

Enregistrer `dasm.svg` :

A.2.3 Attaque par force brute

Enregistrer `solve-part2.c` :

A.3 Annexe : script.ps

A.3.1 Découverte

Enregistrer `script.ps` :

A.3.2 Développement d'une boîte à outils Postscript

Enregistrer `ruby-postscript-0.0.1.gem` :

A.3.3 Analyse de I2

Enregistrer `I2.ps` :

Enregistrer `pdf_sec-ps.txt` :

Enregistrer `bfi2.rb` :

A.3.4 Analyse de I4

Enregistrer `I4.ps` :

A.3.5 Attaque par force brute

Enregistrer `solve-part3.c` :

Enregistrer `md5.c` :

Enregistrer `md5.h` :

A.4 Annexe : fichier vCard

Enregistrer `solve-part4.rb` :

A.5 Annexe : conclusion

Enregistrer `solve-sstic2013.tar.bz2` :

Bibliographie

- [1] Xilinx, *ISE Design Suite* <http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm>
- [2] Xilinx, *Xilinx 7 Series FPGA Libraries Guide for Schematic Designs* http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/7series_scm.pdf
- [3] Adobe, *PostScript Language Reference* <http://partners.adobe.com/public/developer/en/ps/PLRM.pdf>
- [4] Adobe, *PostScript Language Tutorial and Cookbook* <http://partners.adobe.com/public/developer/en/ps/sdk/sample/BlueBook.zip>
- [5] Adobe Developers Association, *Filters and Reusable Streams* <http://partners.adobe.com/public/developer/en/ps/sdk/TN5603.Filters.pdf>