Rétro-ingénierie d'un micrologiciel pour architecture *Pi32v2*

Damien Cauquil
dcauquil@quarkslab.com

Quarkslab

Résumé. La rétro-ingénierie de micrologiciel consiste à comprendre le fonctionnement d'un code compilé s'exécutant sur une architecture matérielle telle qu'un système-sur-puce ou un microcontrôleur, via le désassemblage et la décompilation du code qui le compose. Cette analyse requiert de posséder des outils capables d'interpréter les successions d'octets présents dans le code exécutable en fonction de l'architecture du processeur utilisé et des jeux d'instructions supportés.

Lors de l'analyse d'un micrologiciel de montre connectée, nous avons été confronté à une architecture de processeur que nous ne connaissions pas, dénommée *Pi32v2*. Cette architecture est développée par un fondeur chinois, *JieLi*, et est présente sur de nombreux systèmes-sur-puce dédiés à des applications reposant sur l'utilisation du protocole sans-fil Bluetooth. *JieLi* met par ailleurs à disposition un SDK ainsi que quelques codes exemples, sans autre ressource spécifique.

Cet article détaille la méthodologie qui nous a permis d'identifier et de documenter une partie non-négligeable du jeu d'instructions du processeur *Pi32v2*, d'ajouter le support de ces instructions dans *Ghidra*, et introduit certaines subtilités du langage *SLEIGH* et leur utilisation avancée. Une implémentation sous licence libre pour *Ghidra* de ce processeur est proposée, améliorant une implémentation existante mais incomplète réalisée par Andrey Grigoryev entre 2022 et 2024. Nous montrons enfin comment cette implémentation nous a permis d'analyser le micrologiciel et d'identifier les éléments que nous recherchions.

1 Introduction

La rétro-ingénierie de micrologiciels s'exécutant sur des systèmes embarqués (*« bare metal »*) est une phase critique pour la recherche de vulnérabilités, indispensable à la bonne compréhension du fonctionnement desdits systèmes. Elle repose essentiellement sur la capacité d'outils automatisés à transformer du code machine sous forme de suite d'octets en une série d'instructions élémentaires intelligibles et de leurs opérandes (langage assembleur). Ils sont très efficaces pour peu qu'ils soient en mesure de transposer chaque instruction en son équivalent intelligible et de connaître

son effet sur l'état du processeur et des régions mémoires accédées par le code.

Car là est leur limite : ils peuvent seulement analyser les architectures de processeur (et donc les jeux d'instructions) qui leur sont connues. Toute architecture en dehors de leur base de connaissance rend ces outils totalement impuissants, au grand désespoir de l'analyste. Certains de ces outils offrent toutefois à l'analyste la possibilité de définir le format et la manière dont ces instructions sont encodées sous forme d'octets, ce qui permet le désassemblage puis la décompilation d'un code exécutable auparavant inconnu. Quand la documentation d'une architecture de processeur est connue, créer une description de l'ensemble des instructions supportées par un processeur donné est une formalité. Cela prend du temps, certes, mais aboutit fatalement à un support correct du jeu d'instructions en question (pour peu qu'aucune erreur ne se soit glissée, ou que les définitions créées par des *grands modèles de langage* aient été vérifiées). Mais que fait-on si aucune documentation n'existe ? Comment peut-on déterminer l'encodage des instructions, de leurs opérandes et leurs *significations* ?

Cet article présente un cas concret d'une architecture de processeur partiellement connue (*Pi32v2*), rencontrée lors de l'analyse d'une montre connectée prétendant effectuer des mesures de constantes de santé, que nous soupçonnions être une arnaque.

Il détaille ainsi la méthodologie suivie afin de déterminer le jeu d'instructions et le format de ces dernières à partir des ressources à notre disposition, ainsi que l'intégration dans le logiciel de désassemblage *Ghidra* de cette architecture exotique via le langage de description employé par ce logiciel. Nous montrons ensuite comment ce travail de rétro-ingénierie nous a permis d'analyser le micrologiciel étudié et de trouver les éléments de réponse que nous recherchions. Enfin, nous présentons une implémentation sous licence libre de ce processeur pour *Ghidra* basée sur une implémentation incomplète initiée par Andrey Grigoryev en 2022, ayant permis un désassemblage correct de notre micrologiciel.

L'analyse de la montre connectée a fait l'objet en 2025 d'une présentation donnée par l'auteur et Thomas Cougnard lors de la conférence *leHACK* [3] qui n'a explicité ni la méthodologie suivie ni les difficultés rencontrées qui ont permis d'obtenir le code désassemblé du micrologiciel.

1.1 Présentation de la montre connectée et des travaux précédemment réalisés

La montre connectée que nous avons étudiée est vendue depuis fin 2024 dans les magasins de l'enseigne française GiFi à un tarif défiant

toute concurrence (environ 12 euros lors de notre achat), commercialisée sous la marque "Homday Xpert". On retrouve parmi les fonctionnalités annoncées le suivi de l'activité physique via la mesure de la fréquence cardiaque (Fig. 1), de la pression artérielle et du taux d'oxygène dans le sang. Cependant, une rapide analyse de son électronique a montré l'absence de capteurs permettant ces mesures, et c'est tout naturellement que nous nous sommes interrogés sur l'origine des données affichées par la montre.

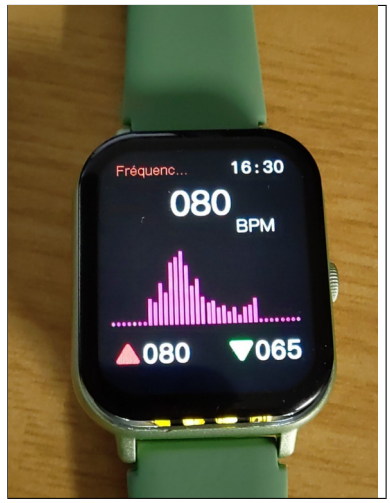


Fig. 1. La montre connectée affichant la fréquence cardiaque de son porteur.

La montre utilise un système-sur-puce peu répandu en Europe qui possède une interface de débogage propriétaire requérant un équipement adapté dont nous ne disposions pas. Il nous a donc fallu ruser afin d'accéder au micrologiciel, en exploitant une vulnérabilité de lecture arbitraire dans le code traitant le téléchargement de thèmes d'écrans personnalisés combinée à une capture des données transmises par le système-sur-puce à l'écran (Fig. 2). Nous avons ainsi extrait portion par portion les données de la mémoire flash intégrée à la montre et reconstitué son micrologiciel. Un billet publié sur le blog de *Quarkslab* [1] détaille la mise au point de cette attaque.

L'analyse matérielle de la montre a permis d'identifier un système-sur-puce `AC6958` du fondeur chinois *JieLi*, qui repose sur un processeur `Pi32v2` d'après la documentation trouvée sur Internet [4]. Ce processeur

Fig. 2. Installation matérielle de capture des données envoyées à l'écran de la montre.

est spécifique au fondeur et diffère significativement des architectures populaires comme ARM ou RISC-V.

Une analyse plus approfondie du micrologiciel indique par ailleurs la présence de plusieurs sections de données, dont la plupart montrent une entropie élevée pouvant indiquer une possible compression ou chiffrement. Seule une section dénommée *app.bin* semble avoir été fort heureusement épargnée : elle contient le code du système d'exploitation de la montre. La compréhension de ce code permettrait d'en savoir plus sur le format et le possible chiffrement des autres sections ; il nous fallait donc *absolument* obtenir au mieux une version désassemblée intelligible du code machine, voire une version décompilée des différentes fonctions d'intérêt.

1.2 Découverte du processeur `Pi32v2`

Le dépôt *Github* identifié lors de notre phase de découverte détaille précisément l'architecture du processeur *Pi32v2* ainsi qu'une partie du jeu d'instructions [6] identifié par son auteur, Andrey Grigoryev, ainsi que les formats de certaines opérandes. Le site officiel de *JieLi* mettant à disposition de la documentation [9] sur l'environnement de développement ainsi que des instructions pour l'installation de leur kit de développement (ou *« SDK »*), bien qu'en langue chinoise, est une véritable mine d'informations. Ce kit est d'ailleurs en libre accès sur leur dépôt *Github* [10].

1.3 État de l'art

Un processeur spécifique à *Ghidra* a déjà été implémenté par Andrey Grigoryev en 2022 [5], mais un rapide test montre qu'un nombre conséquent d'instructions ne sont pas correctement reconnues, interrompant le processus de désassemblage. L'analyse du micrologiciel est quasi-impossible, une

bonne partie des instructions employées par ce dernier ne pouvant être désassemblées.

Plusieurs chercheurs en sécurité se sont attaqués à la rétro-ingénierie de jeu d'instructions d'architectures processeur inconnues et l'ajout de leur support dans des outils de désassemblage ou de décompilation, documentant au passage leur méthodologie. Robert Xiao l'a fait dans le cadre de sa participation au *DragonSector CTF* [13] de 2019, Guillaume Valadon lors de son analyse d'une carte SD WiFi *FlashAir* de *Toshiba* [11] présentée à la conférence *BlackHat* en 2018, et plus récemment Willem a présenté lors du 39ème *Chaos Communication Congress* ses travaux sur la rétro-ingénierie du micrologiciel de la puce *Titan M2* [12] de *Google*. La méthode d'analyse que nous avons suivie est relativement proche de celles présentées par ces chercheurs, sachant qu'une partie des instructions ainsi que leur format général avaient déjà été déterminés et en partie documentés par Andrey Grigoryev.

2 Analyse du jeu d'instructions de l'architecture Pi32v2

Le processeur `Pi32v2` est une évolution du processeur `Pi32` conçu par *JieLi* à partir du modèle de cœur *Blackfin* d'*Analog Devices*. C'est un processeur 32 bits qui possède 16 registres généraux (`r0` à `r15`) qui peuvent être combinés un à un pour former 8 registres de 64 bits (`r1_r0`, `r3_r2`, ...). Il supporte certains mécanismes hérités de l'architecture *Blackfin*, comme l'exécution parallélisée d'instructions ou encore certaines opérations mathématiques complexes. D'après la documentation disponible en source ouverte, il supporte des instructions encodées sur 16, 32 ou 48 bits.

2.1 Identification d'instructions et de leur encodage

L'analyse du jeu d'instructions a été simplifiée par l'existence d'outils de compilation et d'une application exemple fournie par *JieLi*, ce qui nous a permis d'une part d'obtenir un fichier au format *ELF* analysable dans *Ghidra* et d'autre part de profiter de l'outil de « désassemblage » mis à disposition, `objdump`. Les outils de compilation sont basés sur *LLVM* et se présentent sous la forme d'une implémentation de `GCC` spécifique à l'architecture des processeurs en question.

La sortie produite par `objdump` et présentée dans le Listing 1 est assez déroutante pour quelqu'un habitué à désassembler du code exécutable ARM, MIPS ou même X86/64. Il n'y est pas question de mnémonique ou d'opérandes, mais d'une interprétation *algébrique* des instructions héritée de l'architecture *Blackfin*.

Listing 1: Code désassemblé généré par `objdump`

```
sdk.elf:        file format ELF32-pi32v2

Disassembly of section .text:
text_code_begin:
 1e00100:    81 ea 29 bd               call 227922
 1e00104:    ee ff 10 a0 00 00         sp = 40976
 1e0010a:    ed ff 10 a0 00 00         ssp = 40976
 1e00110:    d8 e8 07 00               [--sp] = {r2-r0}
 1e0011a:    c1 ff 00 00 80 00         r1 = 8388608

[...]
```

Toutefois, cette notation algébrique permet de comprendre les opérations effectuées par une instruction mais aussi de déterminer la taille d'une instruction en octets grâce au découpage réalisé par `objdump`. De même, il est aisé de chercher dans le code désassemblé des instructions similaires afin de collecter celles représentant la même opération mais avec des formats d'opérandes différentes. Une analyse différentielle de ces instructions permet de déterminer un *radical* commun et ensuite de déduire la façon dont les opérandes sont encodées, à partir de leurs représentations binaires.

Prenons par exemple l'instruction à l'adresse `1e00114 : r0 = 32230384`. D'après la documentation, le processeur *fetch* les instructions par mot de 16 bits ; il est donc logique d'interpréter les octets correspondant à cette instruction par groupe de 16 bits en orientation *little-endian* (l'octet de poids faible étant stocké avant celui de poids fort) :

```
<--- 0xFFC0 ---> <--- 0xCBF0 ---> <--- 0x01EB --->
1111111111000000 1100110111110000 0000000111101011
```

En faisant de même pour l'ensemble des variantes de cette opération, il est possible de déterminer les bits invariants correspondant au *radical* de l'instruction, c'est-à-dire l'ensemble des bits nécessaire et suffisant pour déterminer qu'il s'agit de cette instruction précise. Il faut noter que l'on parle ici non pas d'une opération en particulier mais bien d'une instruction encodée, qui correspond à une opération combinée à des formats d'opérandes précis. En effet, la grande majorité des jeux d'instructions de processeurs ont été conçus pour permettre une optimisation de la mémoire requise pour encoder chaque opération, via notamment différents formats (ou encodages) de ces opérations en fonction des opérandes. Ainsi, une opération telle qu'une addition ou l'affectation d'une valeur à un

registre peut être déclinée en plusieurs instructions permettant d'encoder les opérandes en fonction de leurs types, tout en limitant l'espace occupé par ces dernières. Ainsi, en appliquant un simple *ET* logique bit-à-bit sur les différentes variantes de l'instruction analysée nous obtenons une valeur dont les bits à 1 correspondent très probablement aux bits invariants. Plus nous disposons de variations d'une même instruction et plus ce résultat sera fiable.

Il est relativement simple d'extraire les variantes d'une instruction à partir du code désassemblé fourni par `objdump`, comme le montre le Listing 2. L'écriture d'un petit script Python (Listing 3) permet d'automatiser la recherche des bits invariants à partir des 2806 instructions identifiées, et confirme notre intuition : seuls les 12 bits de poids fort du premier mot de 16 bits codent le type d'instruction. Une fois ces bits exclus, l'encodage des opérandes apparaît assez clairement :

— l'index du registre général qui sera affecté est stocké dans les 4 bits de poids faible du premier mot de 16 bits ;
— la valeur de 32 bits affectée au registre est stockée en orientation *little-endian* dans les deux derniers mots de 16 bits.

Listing 2: Recherche des variantes d'une instruction

```
1  \$ grep -e
   ↪  '\s[0-9a-f]\+:\(\s\+[0-9a-f]\{2\}\)\{6\}\s\+r[0-9]\{1,2\} =
   ↪  [0-9]\+' disass.txt | cut -d ' ' -f 6-11 | sort -u >
   ↪  variants.txt
2  \$ head variants.txt
3  c0 ff 00 00 00 00
4  c0 ff 00 00 20 41
5  c0 ff 00 00 34 c2
6  c0 ff 00 00 40 c0
7  c0 ff 00 00 48 55
8  c0 ff 00 00 70 41
9  c0 ff 00 00 80 ff
10 c0 ff 00 00 87 93
11 c0 ff 00 00 aa c2
12 c0 ff 00 00 f0 7f
13 \$ wc -l variants.txt
14 2806 variants.txt
```

Listing 3: Code Python permettant la détermination des bits invariants

```python
""" Détection de bits invariants dans les instructions """
from struct import unpack

# On lit notre fichier et on interprète chaque instruction
# comme un assemblage de 3 mots de 16 bits
lines = open("variants.txt" , "r", encoding="utf-8").readlines()
insts = [unpack("<HHH", bytes.fromhex(l.replace(' ','').strip()))
↪  for l in lines]

# On calcule ensuite notre masque mot par mot
mask = [0xffff, 0xffff, 0xffff]
for w0,w1,w2 in insts:
    mask[0] = mask[0] & w0
    mask[1] = mask[1] & w1
    mask[2] = mask[2] & w2

print(f"Invariants: {mask[0]:04x}{mask[1]:04x}{mask[2]:04x}")
```

Le format du premier mot de 16 bits que nous avons identifié dans le cadre de l'analyse différentielle de cette instruction est lui aussi constitué de différents champs ayant des usages spécifiques. Nous l'avons analysé en comparant les premiers mots de différentes instructions identifiées dans le code désassemblé.

Longue et fastidieuse, cette phase de rétro-ingénierie permet néanmoins de discriminer les différentes instructions et de déterminer la manière dont chaque opérande est encodée. La compréhension de l'instruction, de son rôle et de son fonctionnement donne les clés pour décrire son action sur l'état du processeur et de sa mémoire, ouvrant la voie à l'émulation de l'instruction et à la décompilation grâce aux outils comme *Ghidra*. Dans le cas présent, cela nous a pris environ 4 semaines pour analyser la quasi-totalité des instructions et définir leur principe de fonctionnement ainsi que leur encodage. Certaines instructions n'ont pas été complètement déterminées du fait de l'imprécision de la notation algébrique dans certains cas, et des incohérences dans le code compilé dans d'autres.

2.2 Formats des instructions

Les instructions de ce processeur peuvent être encodées sur 16, 32 ou 48 bits en fonction des opérandes requises, toutefois les 16 premiers bits permettent au processeur de déterminer l'instruction en question ainsi que le format des opérandes. Ainsi, le décodage d'une instruction consiste

à lire un mot de 16 bits puis à déterminer s'il faut considérer les 16 ou 32 bits suivant comme faisant partie de l'instruction.

Le premier mot de 16 bits constituant une instruction *Pi32v2* est composé d'un préfixe de 3 bits définissant le groupe auquel est associée l'instruction, suivi de 13 bits dont le format varie en fonction du groupe, comme le montre la Fig. 3. Les instructions appartenant aux groupes 0 à 6 sont encodées sur 16 bits, celles du groupe 7 peuvent l'être sur 16, 32 ou 48 bits en fonction de leur sous-groupe.

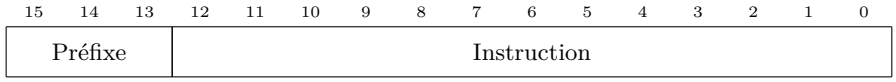


Fig. 3. Structure du premier mot de 16 bits d'une instruction

Prenons l'exemple du groupe 1, dont les instructions sont encodées sur 16 bits. Ce groupe définit un format d'instruction qui lui est propre, avec des variations en fonction des types d'opérations. La Fig. 4 donne le format général des instructions de ce groupe. Ce format a été retrouvé en analysant les différents encodages présents dans le code désassemblé obtenu précédemment, en appliquant une méthode de recherche d'invariants par type d'opération combinée à une analyse de l'encodage des valeurs immédiates et des index de registres, si utilisés dans l'opération. Ainsi dans ce groupe, une valeur immédiate peut être encodée sur 5 bits (du bit 8 au bit 12) ou 6 bits (du bit 8 au bit 12, auxquels s'ajoute le bit de poids fort stocké en bit 3), et les registres référencés sur 3 bits (du bit 0 au bit 2) ou 4 bits (du bit 0 au bit 3) en fonction des instructions. Les index de registres codés sur 3 bits ne permettent donc de manipuler que les 8 premiers registres généraux de 32 bits r0 à r7, tandis qu'un codage sur 4 bits permet de les utiliser tous (de r0 à r15).

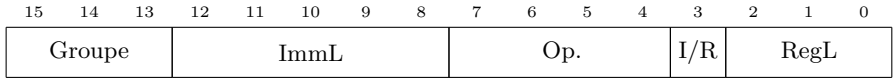


Fig. 4. Structure d'une instruction du groupe 1

Les différentes opérations de ce groupe sont définies par une valeur encodée sur 4 bits, limitant de fait leur nombre à 16. Ces opérations manipulent principalement les registres, permettant de charger une valeur immédiate (`0x4`) ou encore d'effectuer des opérations booléennes sur les

valeurs qu'ils contiennent (`0xC`). Certaines opérations permettent même de charger dans un registre le contenu pointé par le registre de pile auquel un décalage est appliqué (`0x0` et `0x8`).

Ainsi, l'instruction `r7 = 131` trouvée dans le listing de code désassemblé et constituée des octets `0x67 0x23` se décompose comme indiqué en Fig. 5. Les deux octets forment un mot de 16 bits stocké en orientation *little-endian*, et leur décodage permet de vérifier notamment que le groupe correspond bien à la valeur attendue, que l'opération associée est `0110` en binaire soit `6` en décimal, et que l'index de registre indiqué dans les trois bits de poids faible correspond bien au numéro de registre correspondant à une des opérandes de l'instruction (`r7`). Il ne reste plus qu'à comprendre comment la valeur `131` est encodée. La valeur `131` en décimal donne `100000011` en binaire, et l'on retrouve en partie cette valeur dans le champ `ImmL` de l'instruction. L'analyse de plusieurs instructions similaires montre que le champ `I/R` est toujours nul, il est alors fort probable que cette particularité fasse partie intégrante de l'instruction. En effet, la valeur immédiate que l'on peut encoder est au maximum de 5 bits, Or `131` est codée sur 8 bits. Nous faisons donc l'hypothèse que l'opération ayant pour code `6` ajoute 128 à la valeur immédiate renseignée. Cette opération peut donc placer les valeurs de 128 à 159 (128 + 31) dans un registre général dont l'index est compris entre 0 et 7.

Une fois le format général des instructions déterminé, il est plus facile de l'affiner à l'aide des autres instructions qui correspondent à ce dernier, de manière à déterminer le rôle de chaque champ dans l'interprétation de l'instruction. Il va nous falloir aussi comprendre ce que fait chaque instruction afin de pouvoir étudier le fonctionnement du micro-logiciel, d'autant plus si l'on doit faire en sorte que *Ghidra* supporte au mieux l'architecture cible. En effet, cette structure est un des éléments fondamentaux requis pour pouvoir définir le jeu d'instructions de ce processeur sous *Ghidra*, via le langage de définition *SLEIGH* propre à cet outil.

15 14 13	12 11 10 9 8	7 6 5 4	3	2 1 0
Groupe	ImmL	Op.	I/R	RegL
0x23		0x67		
001	00011	0110	0	111

Fig. 5. Décodage de l'instruction `r7 = 131` selon le format d'instruction du groupe 1.

Format des instructions du groupe 0 Les instructions du groupe 0 suivent un format global sur 16 bits (Fig. 6), mais certaines d'entre elles peuvent utiliser certains bits normalement réservés à l'encodage de registres pour définir des instructions spécialisées appartenant à un groupe identifié par un code opération unique.

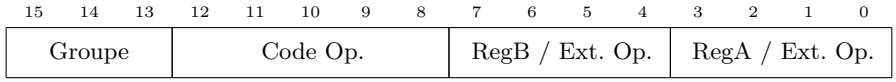


Fig. 6. Structure générale d'une instruction du groupe 0.

L'instruction *r0 = r0 + r1* sera ainsi encodée selon le format général avec un code opération de `0x18` et les registres `r0` et `r1` respectivement encodées par les index sur 4 bits `0` et `1`, ce qui donne au final l'encodage détaillé dans la Fig. 7.

15 14 13	12 11 10 9 8	7 6 5 4	3 2 1 0
Groupe	Code Op.	RegB / Ext. Op.	RegA / Ext. Op.
0x18		0x10	
000	11000	0001	0000

Fig. 7. Décodage de l'instruction `r0 = r0 + r1` selon le format d'instruction du groupe 0.

Certaines opérations utilisent des index de registres sur 3 bits au lieu de 4 afin de pouvoir faire intervenir par exemple un troisième registre dans une instruction. Il existe ainsi une variante de l'instruction précédente qui permet de sommer la valeur de deux registres et de stocker le résultat dans un troisième, avec un format quelque peu différent. Le code opération associé (`0xE`) est spécifique à cette instruction et est stocké sur les 4 bits de poids fort du code opération, le bit de poids faible étant utilisé pour encoder le bit de poids fort (*b2*) de l'index du troisième registre. Les deux bits restants (*b1* et *b2*) sont encodés respectivement dans les bits de poids fort de `RegB` et `RegA`, permettant ainsi d'encoder trois index de registres de 3 bits chacun. La Fig. 8 montre l'encodage de l'instruction *r0 = r4 + r2*.

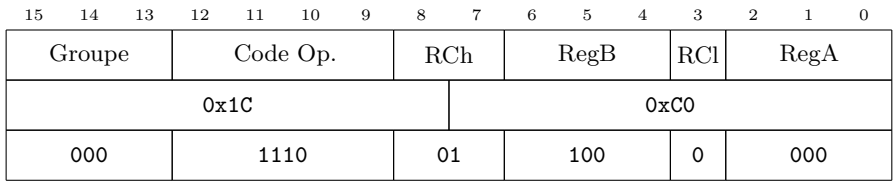


Fig. 8. Décodage de l'instruction `r0 = r4 + r2` selon le format d'instruction du groupe 0.

La place manque dans cet article pour détailler l'ensemble des cas particuliers. Toutefois, la grande majorité des instructions du groupe 0 suit peu ou prou le format général, et cela se retrouve dans notre implémentation réalisée au fur et à mesure de l'identification du jeu d'instructions.

Format des instructions des groupes 2, 3 et 5 Les instructions de ces groupes, au nombre de 4 par groupe, ont été précédemment identifiées par Andrey Grigoryev et respectent le format décrit dans la Fig. 9 ci-dessous. Les instructions sont principalement déterminées par les trois bits de poids fort du premier mot, définissant leur groupe, couplés aux bits 7 et 3. Il y a donc seulement quatre combinaisons possibles, ce qui correspond aux observations de Grigoryev. Les valeurs codées dans ces instructions sont représentées sous forme d'entiers signés avec une construction variable en fonction des instructions.

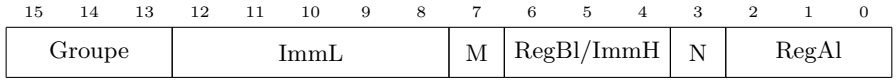


Fig. 9. Structure générale d'une instruction pour les groupes 2, 3 et 5.

Lorsque le bit N est à 1, les bits 4 à 6 sont considérés comme un index de registre *RegBl* et la valeur immédiate est stockée intégralement sur les bits 8 à 12, le bit de signe étant placé sur le bit 12. Les bits 0 à 2 codent quant à eux l'index de registre *RegAl*. Lorsque le bit N est à 0, la valeur immédiate est codée pour sa partie basse sur les bits 8 à 12, auxquels viennent s'ajouter la partie haute codée sur les bits 4 à 6, le bit de signe étant placé sur le bit 6. La valeur représentée est obligatoirement un multiple de 2, la valeur immédiate ne stockant que les 8 bits de poids fort (le bit 0 de la valeur immédiate étant à 0). Cela permet de coder des

valeurs entières signées de -255 à +255, utilisées dans le cas présent pour des sauts relatifs.

L'instruction *if (r0 != 0) goto $ + 4* (groupe 2) est encodée en utilisant ce système de valeur immédiate, avec M à 1 et N à 0, comme indiqué sur la Fig. 10. La valeur immédiate appliquée comme déplacement relatif à l'adresse du registre d'instruction est encodée sous la forme d'un entier binaire signé de 8 bits de valeur `0b00000010`, soit la valeur 2 en décimal. Cette valeur encodée doit être multipliée par 2 (soit un décalage d'un bit à gauche) pour que l'on retrouve le déplacement attendu de *+4*.

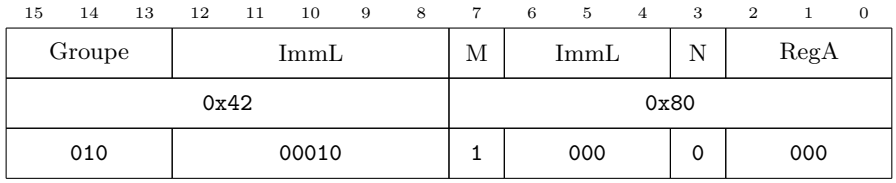


Fig. 10. Décodage de l'instruction `if (r0 != 0) goto $ + 4` selon le format d'instruction du groupe 2.

Format des instructions du groupe 4 De manière similaire, le groupe 4 différencie les instructions en fonction des bits 3 et 7, avec une légère différence par rapport aux groupes précédents : l'une des combinaisons utilise les bits 0 à 2 pour déterminer un sous-ensemble d'opérations. Le format général des instructions du groupe 4 (Fig. 11) illustre principalement les éléments permettant au processeur de déterminer l'instruction encodée. Les opérandes quant à elles sont encodées en fonction des différentes instructions et de leurs sous-ensembles, et sont détaillées par la suite. Encore une fois, le format général est très similaire à celui des groupes 2 et 3 illustrés précédemment.

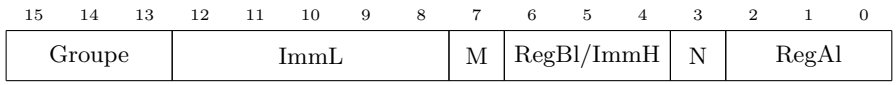


Fig. 11. Structure générale d'une instruction du groupe 4.

Lorsque le bit N est à 1, le bit M détermine l'opération encodée :

— Si M est à 1 alors il s'agit d'une opération de type `regAl = SP + imm7` ;
— Si M est à 0 alors il s'agit d'une opération de type `regAl = regBl + imm5`.

Dans le cas où le bit N est à 0, quatre opérations différentes peuvent être représentées :

— Si les bits 0 à 2 valent *0b000*, il s'agit d'une instruction de répétition de blocs d'instructions ;
— Si les bits 0 à 2 valent *0b001*, il s'agit d'un appel de sous-fonction ;
— Si les bits 0 à 2 valent *0b010*, il s'agit d'une opération de type `sp = sp + imm10` ;
— Enfin, si le bit 2 est à 1, il s'agit d'un saut avec un décalage défini par une valeur stockée sur 12 bits.

Le cas particulier de l'instruction de saut impliquant un décalage représenté par une valeur entière stockée sur 12 bits montre bien que le format de certaines instructions peut radicalement changer malgré leur appartenance à un groupe spécifique, comme l'illustre la Fig. 12. L'instruction encodée par les octets `0x05 0x80` (codant le mot `0x8005`) fait ainsi bien partie du groupe 4, et encode son adresse au travers des bits 0 à 1 (partie haute) combinés aux bits 4 à 12 (partie basse). Ainsi, cette instruction peut être interprétée comme un saut à l'adresse calculée à partir de la valeur immédiate encodée `0b01000000000`, qui vaut *512* en décimal. Les adresses des instructions étant multiples de 2, cette valeur doit être multipliée par 2 pour ainsi retrouver l'adresse de destination, en l'occurrence *1024* (décimal). Cela donne l'instruction équivalente `goto 1024` que l'on retrouve notamment dans le code désassemblé produit par la version d'*objdump* de *JieLi*.

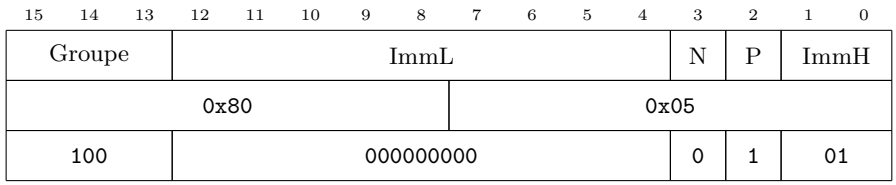


Fig. 12. Décodage de l'instruction `goto 1024` selon le format d'instruction du groupe 4.

Format des instructions du groupe 6 À ce jour, une seule instruction du groupe 6 a été identifiée et respecte le format illustré par la Fig. 13. Ce dernier repose sur un code opération encodé sur les bits 9 à 12, et trois index de registres différents dont l'encodage a déjà été explicité dans les sections précédentes. Les instructions du groupe 6, tout comme certaines instructions du groupe 7, sont exécutées en parallèle de l'instruction qui les suit. Cette particularité a des conséquences non-négligeables sur l'implémentation de ces dernières dans *Ghidra*.

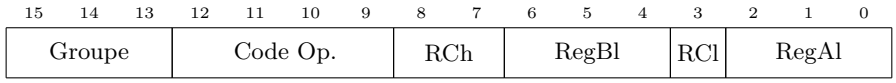


Fig. 13. Structure générale d'une instruction pour le groupe 6.

L'instruction codée par les octets `0x91 0xde` (soit `0xde91` sous sa forme 16 bits) est décodée dans la Fig. 14.

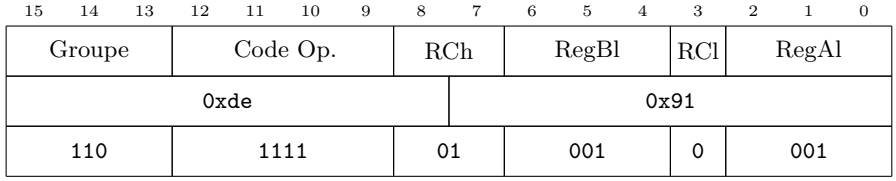


Fig. 14. Décodage de l'instruction `r1 = r1 - r2` (parallélisée) selon le format d'instruction du groupe 6.

Format des instructions du groupe 7 Ce dernier groupe contient la grande majorité des instructions supportées par l'architecture *Pi32v2* (ce qui représente exactement 216 instructions, soit environ 72% de celles identifiées), avec des encodages variables utilisant 16, 32 ou 48 bits pour représenter une instruction et ses opérandes.

Lors de notre analyse, un premier tri a été fait dans les instructions en fonction de leur taille, ce qui nous a permis de définir trois formats principaux. À partir de cette première classification, nous avons procédé par rapprochement en fonction des familles d'opérations et des types d'opérandes acceptées par ces dernières afin de définir plusieurs sous-ensembles et dans certains cas des formats spécifiques associés.

Il nous a aussi fallu déterminer comment l'exécution parallèle d'instructions était encodée, car une bonne partie des instructions de ce groupe 7 ont des variantes parallélisées, comme le montrent par exemple les deux instructions suivantes et leurs encodages différents :

Listing 4: Deux versions de la même opération, l'une parallélisée et l'autre non

```
1e3b98e:    c8 f1 30 11        r1 = r3 << r1  #
1e3b992:    0a 41                   r2 = b[r0+1] (u)
[...]
1e545e4:    c8 e1 30 11        r1 = r3 << r1
```

La différence entre ces deux versions de l'instruction *r1 = r3 « r1* est minime et ne concerne que le bit 12 du premier mot de l'instruction. Il semblerait donc que pour ce groupe en particulier, ce bit soit l'élément codant l'activation de l'exécution parallèle de l'instruction suivante. Mais ce qui semble trivial ne l'est pas forcément, et nous nous sommes rendus compte par la suite que seules les instructions ayant le bit 12 à 1 *et le bit 11 à 0* sont parallélisées, ce qui réduit le nombre de possibilités.

Nous avons ainsi pu identifier sept formats spécifiques pour les instructions du groupe 7 :

- Format I (Fig. 15) : l'instruction est encodée sur 2 mots avec le code opération encodé sur 12 bits ;
- Format II (Fig. 16) : l'instruction est encodée sur 2 mots avec le code opération encodé sur 8 bits ;
- Format III (Fig. 17) : l'instruction est encodée sur 2 mots avec le code opération sur 7 bits ;
- Format IV (Fig. 18) : l'instruction est encodée sur 2 mots et son code opération sur 6 bits ;
- Format V (Fig. 19) : l'instruction est encodée sur 2 mots, le code opération sur 9 bits (non-parallélisable) ;
- Format VI (Fig. 20) : l'instruction est encodée sur 3 mots (non-parallélisable) ;
- Format VII (Fig. 21) : l'instruction est encodée sur 3 mots et contient une valeur immédiate encodée sur 32 bits (non-parallélisable).

Considérations en vue de l'implémentation dans Ghidra L'identification des champs constituant les différentes instructions en fonction de leur groupe (code opération, index de registres, etc...) permet une meilleure compréhension de la façon dont le processeur *Pi32v2* discrimine

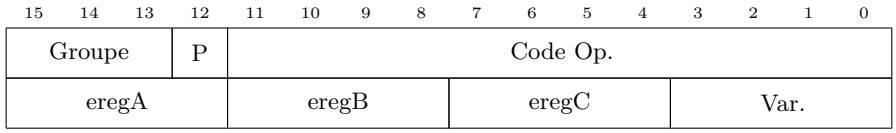


Fig. 15. Instruction du groupe 7, Format I

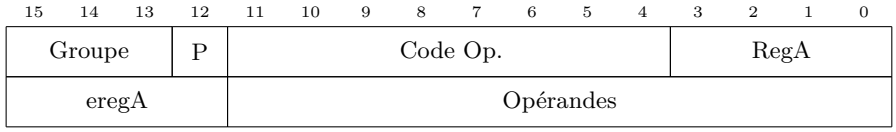


Fig. 16. Instruction du groupe 7, Format II

ces dernières au travers du premier mot de chaque instruction. C'est d'autant plus important que le langage *SLEIGH* utilisé par *Ghidra* pour la modélisation de processeurs repose exactement sur ce concept : le désassembleur consomme des tokens de taille fixe et les découpe en champs constitués de bits afin de décoder correctement les instructions présentes dans un code exécutable.

Il est important d'arriver à une vision correcte du ou des formats d'encodage des différentes instructions afin d'éviter toute collision ou mauvaise interprétation, et d'assurer une transcription fidèle de ces instructions en langage assembleur. Il en va de même pour les opérandes et leurs différents formats, et c'est exactement ce que détaille la section suivante.

2.3 Formats des opérandes

Dans la section précédente, nous avons donné pour exemple le codage d'un registre sur la base de son numéro (ou *index*) ainsi que d'une valeur entière *immédiate* contenue sous une forme particulière dans le code de l'instruction. Ces deux exemples illustrent bien les cas relativement triviaux, mais cette architecture recèle de formats quelque peu exotiques pour modéliser des adresses relatives ou encore des masques binaires.

Prenons comme exemples les instructions suivantes, extraites de notre code désassemblé :

```
15a00:      e7 e0 0c 1d        r7 = r1 + 0x2300
[...]
1e0636c:    e5 e0 80 46        r5 = r4 + 0x4000000
[...]
1e44ac4:    ea e0 40 94        r10 = r9 + 0xC0000000
```

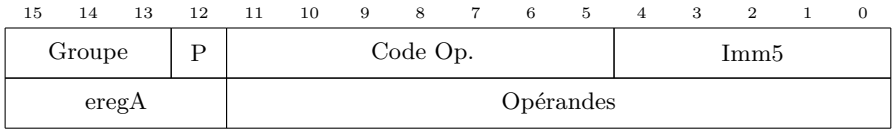


Fig. 17. Instruction du groupe 7, Format III

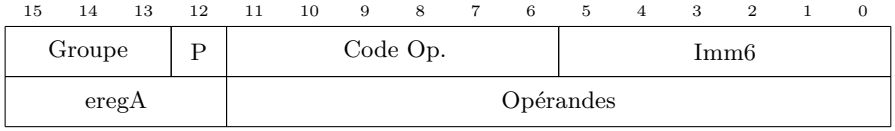


Fig. 18. Instruction du groupe 7, Format IV

Ces instructions sont issues du *groupe 7* et codées sur 4 octets, cependant elles représentent exactement le même type d'opération : une affectation d'un registre avec une valeur additionnée à la valeur d'un second registre. Nous remarquons assez rapidement que, contrairement aux opérandes de la section précédente, la valeur ajoutée au registre source n'apparaît pas de manière évidente dans les instructions codées. Ces valeurs sont donc elles-aussi encodées dans un format qu'il va nous falloir déterminer, reposant certainement sur une forme de codage optimisé.

Analysons dans un premier temps le premier mot de 16 bits de chaque instruction, car nous savons avec certitude qu'il doit correspondre au format précédemment identifié, et affichons-les en colonnes comme montré en Fig. 22. Il apparaît assez rapidement que les quatre bits de poids faible (mis en évidence ici en rouge) semblent coder le numéro du registre de destination, et qu'il n'y ait pas d'autres différences. Cela signifie donc :

— que l'index du registre de destination est codé sur les 4 bits de poids faible ;
— que les bits 4 à 12 encodent le type d'opération (hypothèse) ;
— que la valeur ajoutée au registre source est encodée dans le second mot de 16 bits.

Si l'on réalise la même mise en colonne avec les seconds mots de chaque instruction, on obtient le résultat présenté en Fig. 23. Il paraît relativement évident que les bits 28 à 31 codent l'index du registre source, on en déduit donc que la valeur ajoutée à la valeur contenue dans ce registre source est calculée à partir des bits 16 à 27. Aucune solution évidente ne nous saute aux yeux, mais l'affichage sous forme *hexadécimale* de ces valeurs dans le code désassemblé par *objdump* ne semble pas anodin : cette notation

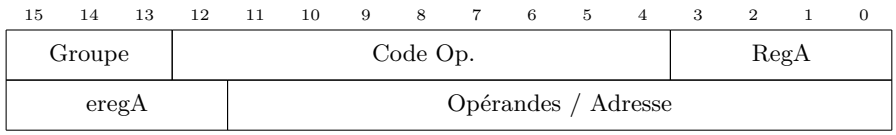


Fig. 19. Instruction du groupe 7, Format V

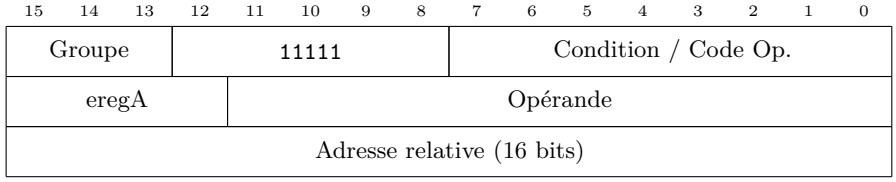


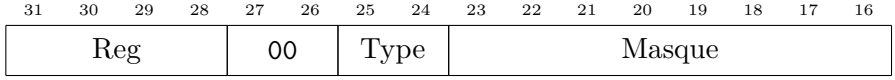
Fig. 20. Instruction du groupe 7, Format VI

pourrait indiquer une interprétation de cette valeur comme un masque binaire, représenté sous forme *hexadécimale* pour en faciliter la lecture.

Plusieurs heures auront été nécessaires pour en déceler la logique, mais nous avons finalement identifié le codage employé, qui est loin d'être trivial. Celui-ci optimise au mieux le nombre de bits nécessaires au codage d'un masque binaire afin de minimiser la taille de l'instruction encodée :

— Le simple masque binaire constitué de 8 bits et appliqué au 8 bits de poids faible d'une valeur de 32 bits (par exemple `0x000000FF`) ;
— Le masque binaire constitué de 8 bits répétés sur chaque *octet* d'une valeur de 32 bits (par exemple `0xF0F0F0F0`) ;
— Le masque binaire constitué de 8 bits répétés sur le second et troisième octet d'une valeur de 32 bits (par exemple `0xFF00FF00`) ;
— Le masque binaire constitué de 8 bits dont le bit de poids fort est actif, et décalé de 24 bits à gauche (par exemple `0xCC000000`) ;
— Le masque binaire constitué de 8 bits dont le bit de poids fort est actif, et décalé de 16 bits à gauche (par exemple `0x00CC0000`) ;
— Le masque binaire constitué de 8 bits dont le bit de poids fort est actif, et décalé de 8 bits à gauche (par exemple `0x0000CC00`).

Les masques à base de répétition de motif de 8 bits présentent les bits `26` et `27` à `0`, les bits `24` et `25` codant 4 types de masques possibles (mais nous n'en avons rencontré que deux) tandis que le masque de 8 bits est codé sur les bits `16` à `23`. Cela donne le motif suivant :



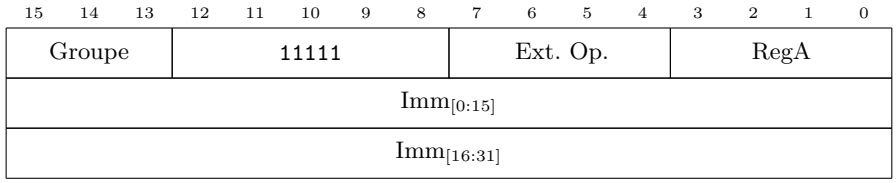
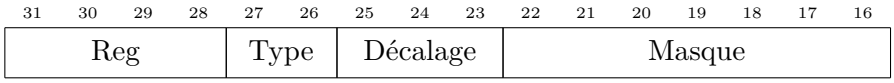


Fig. 21. Instruction du groupe 7, Format VII

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Groupe			Instruction												
1	1	1	0	0	0	0	0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1	1	1	0	0	1	0	1
1	1	1	0	0	0	0	0	1	1	1	0	1	0	1	0

Fig. 22. Décodage des instructions selon le format d'instruction général

Pour les autres, leur type est défini par la combinaison des bits 26 et 27 lorsqu'au moins un des deux est différent de 0, les 7 bits de masque définis sur les bits 16 à 22 et un décalage à droite optionnel sur les bits 23 à 25. Cela donne le format suivant :



Ainsi, le second mot de la première instruction (première ligne de la Fig. 23) se décode comme suit :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reg				Type		Décalage			Masque						
0	0	0	1	1	1	0	1	0	0	0	0	1	1	0	0

Ce qui peut se traduire en code équivalent par `((0x8C « 8) » 6)`, soit la valeur `0x2300` qui correspond bien à celle affichée dans le code désassemblé. Cette façon de coder des masques binaires est loin d'être triviale, et ce ne fut pas la seule de cet acabit que nous avons identifiée lors de l'analyse du jeu d'instructions de ce processeur. Par ailleurs, les informations décrivant ce type de format trouvées dans la documentation disponible en ligne (cf. [6]) se contentent d'énumérer toutes les possibilités rencontrées lors de l'analyse des différentes instructions, le format décrit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Second mot															
0	0	0	1	1	1	0	1	0	0	0	0	1	1	0	0
0	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0

Fig. 23. Décodage des instructions selon le format d'instruction général

ci-dessus est une interprétation plus arithmétique de cet encodage, qui simplifie notamment son implémentation.

Nous avons montré précédemment au travers de l'analyse de certaines instructions comme les sauts conditionnels (cf. 2.2) comment une valeur immédiate pouvait se trouver dans certains cas découpée en deux parties distinctes, l'une encodant les bits de poids faibles et l'autre les bits de poids forts, avec potentiellement un bit de signe si la valeur peut être un entier signé. Nous avons par ailleurs identifié 5 variantes de cet encodage permettant de représenter des offsets de 12, 16, 23 et 32 bits, en plus de ceux de 9 bits mentionnés dans la section précédente. Ces encodages ne sont pas détaillés dans cet article par manque de place, ils respectent cependant la même logique de découpage et d'intégration dans les mots de 16 bits composants les différentes instructions les employant.

3 Implémentation dans Ghidra

Une fois le format des instructions et ses variantes ainsi que les différents encodages des opérandes identifiés, nous avons tous les éléments pour ajouter le support du processeur *Pi32v2* dans *Ghidra*, ou plutôt améliorer considérablement celui déjà entamé par Andrey Grigoryev et publié sous licence Apache. Il nous faut donc principalement ajouter le support des instructions absentes faisant partie des groupes 6 et 7. Pour ce faire, nous devons modifier les définitions du processeur et de ses instructions pour modéliser leurs différents formats et décrire comment celles-ci doivent être décodées. De la même façon, nous devons aussi décrire l'encodage des opérandes afin de pouvoir les décoder et faire en sorte que *Ghidra* les interprète correctement.

Ajouter le support d'un processeur dans *Ghidra* consiste donc à définir un fichier au format `slaspec` contenant un ensemble de directives propres au langage *SLEIGH* utilisé par *Ghidra*, des fichiers annexes permettant au

désassembleur de transformer la définition du processeur en une version binaire compacte qu'il sera en mesure d'utiliser, ainsi que des fichiers de méta-données. Ces méta-données permettent à l'interface de proposer le processeur en question en fonction de certains champs liés à l'exécutable analysé, si c'est un fichier au format *ELF* par exemple. Le fichier `slaspec` peut être découpé en sous-fichiers qui seront inclus par ce dernier, permettant ainsi de catégoriser les différents types d'instructions et de faciliter la maintenance.

La définition de chaque instruction est scindée en trois parties principales :

— la représentation intelligible de l'instruction sous forme de *mnémonique* et d'*opérande(s)* ;
— les conditions à remplir pour considérer cette instruction durant la phase de décodage ;
— le pseudo-code représentant l'action de l'instruction sur la mémoire et l'état du processeur.

Ce système de définition permet ainsi à *Ghidra* d'associer à chaque instruction sa représentation intermédiaire (ou « IR »), et de s'en servir dans les phases de décompilation ou d'émulation, ce qui offre des perspectives très intéressantes en termes de recherche de vulnérabilités.

3.1 La description de l'architecture du processeur

L'architecture du processeur *Pi32v2* est caractérisée par son orientation *little-endian*, son alignement mémoire ainsi que ses registres généraux et spécifiques. Le code du Listing 5 montre la définition de ce dernier dans le fichier `pi32v2.slaspec`, où l'on peut notamment remarquer que les registres sont définis comme des éléments d'une zone mémoire (là où matériellement parlant il s'agit généralement de *buffers* n'ayant pas d'adresse en mémoire) et que certains registres comme `mult_addr` ou `cres` ne correspondent pas à de vrais registres faisant partie de la définition du processeur. Ceux-ci ont été volontairement ajoutés pour faciliter la traduction de certaines instructions en leur représentation intermédiaire, afin d'obtenir une action identique à celles des instructions d'origine.

Listing 5: Code *SLEIGH* définissant le processeur *Pi32v2*

```
define endian    = little;
define alignment = 2;
define space ram      type=ram_space      size=4 default;
define space register type=register_space size=4;

# General purpose registers
define register offset=0x0 size=4
        [ r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 ];

# Paired registers (64-bit)
define register offset=0x0 size=8
        [ r0_r1 r2_r3 r4_r5 r6_r7 r8_r9 r10_r11 r12_r13 r14_r15 ];

# Special function registers
define register offset=0x100 size=4
        [ reti rete retx rets sr4 psr cnum sr7
          sr8  sr9  sr10 icfg usp ssp sp pc
    ];

define register offset=0x0160 size=4
    [   mult_addr # Register used in IR for addresses
        cres      # Register used in conditional blocks
        rep_count # Repeat block counter (IR)
        rep_start # Repeat start address (IR)
    ];
```

3.2 La définition d'une instruction et son décodage

La définition d'une instruction et son décodage suit le format des *constructors* défini dans les spécifications du langage *SLEIGH* [8] :

```
TABLE:DISPLAY is PATTERN
[ DISASSEMBLY ]
{
    SEMANTIC
}
```

— `TABLE` définit un en-tête de table ;
— `DISPLAY` décrit l'affichage de l'instruction ;
— `PATTERN` spécifie un motif binaire utilisé pour le décodage de l'instruction ;
— `DISASSEMBLY` indique une ou plusieurs *actions de désassemblage* ;
— et enfin `SEMANTIC` fournit les *actions sémantiques* de l'instruction, en clair le pseudo-code correspondant à son action.

Les *actions sémantiques* sont exprimées sous forme de séries d'instructions atomiques décrivant le *fonctionnement* de l'instruction et son impact sur les registres et la mémoire, constituant la représentation intermédiaire de l'instruction. Les instructions atomiques sont en nombre relativement limité (63) et couvrent la majorité des usages. Il faut toutefois ruser un peu en fonction des architectures pour obtenir un comportement du pseudo-code identique à l'action (ou aux actions) propre(s) à une instruction.

La section spécifiant les *actions de désassemblage* est quelque peu particulière et sert principalement à :

— effectuer des calculs sur des valeurs issues du décodage afin d'intégrer le résultat dans l'affichage de l'instruction et de l'utiliser dans les *actions sémantiques* ;
— altérer un ou plusieurs *registres de contexte* utilisés lors du décodage des instructions.

Prenons en exemple l'instruction `r7 = 131` évoquée en 2.2, codée par le mot de 16 bits `0x2367`. Nous savons d'après le format des instructions que l'instruction est constituée d'un champ définissant le groupe auquel elle appartient (constitué des 3 bits de poids fort), ainsi que d'autres champs identifiés s'il s'agit d'une instruction du groupe 1. Ainsi, nous commençons par définir les champs utiles au décodage comme suit :

```
1  define token instr(16)
2      group = (13, 15)   # champ définissant le groupe de
       ↪  l'instruction
3      immL = (8, 12)     # champ contenant les 5 bits de poids faible
4                         #  de la valeur immédiate
5      op = (4, 7)        # champ définissant l'opération
6      immH = (3, 3)      # champ définissant le bit de poids fort de
7                         #  la valeur immédiate
8      reg = (0, 2)       # champ définissant un index de registre sur
9                         #  3 bits
10     ereg = (0, 3)      # champ définissant un index de registre sur
11                        #  4 bits
12 ;
13
14 attach variables [ reg ]
15     [ r0 r1 r2 r3 r4 r5 r6 r7 ];
16
17 attach variables [ ereg ]
18     [ r0 r1 r2 r3 r4 r5 r6 r7 r8
19       r9 r10 r11 r12 r13 r14 r15 ];
```

Les valeurs en parenthèses indiquent le numéro du bit de début et celui du bit de fin. Ainsi, `(13,15)` représente un champ de 3 bits allant du bit 13 au bit 15, tandis que `3,3` représente un champ constitué du seul bit 3. Nous définissons donc des tokens destinés à être utilisés dans la section `PATTERN` de la définition de notre instruction. La directive `attach` permet d'associer une valeur décodée à un registre selon la liste passée en paramètre. Dans notre cas la valeur contenue dans `reg` spécifie le numéro du registre de destination, nous associons donc les registres à leurs index respectifs.

La définition minimale donnée dans le Listing 6 permet la lecture d'un token de 16 bits et le décodage des valeurs de `group`, `op`, `ir` et `reg`. Le désassembleur cherche ensuite une définition d'instruction dont le motif binaire correspond dans la liste des *constructors*, afin de générer la version intelligible de l'instruction et d'y associer le pseudo-code correspondant. Dans ce contexte, la variable `reg` est associée au registre qui lui est attaché, et cette instruction pourra donc affecter une valeur aux registres r0 à r7. La définition complète de cette instruction est donnée dans le Listing 6. Le calcul de la valeur immédiate est effectué par le code situé dans les *actions de désassemblage*, définissant une nouvelle variable `imm` utilisée dans l'affichage et le pseudo-code.

Listing 6: Définition minimale permettant de décoder l'instruction `r7 = 131`

```
    # Définition d'un token de 16 bits (groupe 1)
define token instr(16)
    group = (13, 15)
    immL = (8, 12)
    op = (4, 7)
    immH = (3, 3)
    reg = (0, 2)
    ereg = (0, 3);

attach variables [ reg ]
    [ r0 r1 r2 r3 r4 r5 r6 r7 ];

attach variables [ ereg ]
    [ r0 r1 r2 r3 r4 r5 r6 r7 r8
      r9 r10 r11 r12 r13 r14 r15 ];

# Instruction de groupe 1, 'rX = 128 + imm5'
:mov reg, #imm is group=1 & op=6 & ir=0 & reg & imm5
[ imm = imm5 + 128; ] { reg = imm; }
```

Ghidra est maintenant en mesure de décoder une telle instruction en générant la bonne mnémonique associée aux bonnes opérandes mais aussi d'avoir une description précise de son fonctionnement qui correspond exactement à l'action de l'instruction. Cette description du fonctionnement de l'instruction est utilisée pour reconstruire le code C correspondant à une fonction, mais peut aussi servir à émuler l'instruction.

3.3 La chaîne de décodage et le rôle des tables

Le décodage d'une instruction débute par l'analyse du symbole racine `instruction`, qui permet d'identifier le *constructor* qui correspond, sur la base de son motif binaire (son `PATTERN`) et de la table courante. La table employée par défaut est la table *racine*, associée au symbole *instruction*, et de fait seules les *constructors* associés à cette table racine seront évalués. Si l'un de ceux-ci fait référence à une autre table dans son motif binaire, alors tous les *constructors* de cette dernière seront évalués selon le contexte de décodage actuel. Le code du Listing 7 définit un *constructor* associé à la table racine et un second associé à une table spécifique, permettant de définir une seule fois la manière dont un type d'opérande doit être décodé. La table `jaddr9` est ainsi définie, contenant un seul *constructor* dont le motif binaire repose sur les champs `imm0812` et `imm0406s` appartenant au premier token de 16 bits d'une instruction. Le calcul de la variable *res* est défini dans les actions de désassemblage, basé sur les valeurs extraites de l'instruction en cours de décodage et la variable spéciale `inst_next`, afin de calculer l'adresse de destination à partir d'un décalage correspondant à la valeur immédiate signée contenue dans les 16 bits de l'instruction. Cette variable *res* est *exportée* via le pseudo-code défini dans ce *constructor*, permettant ainsi à un *constructor* faisant référence à `jaddr9` de pouvoir l'utiliser, cette dernière étant associée au « résultat » de l'expression évaluée.

Il s'agit ici d'une utilisation bien spécifique d'un *constructor* visant non pas à produire un code assembleur lisible par un humain mais bien à définir la façon dont une ou plusieurs portions d'un token doivent être assemblées pour décoder correctement un type d'opérande. Le *constructor* `jaddr9` est employé au sein du motif binaire d'un second *constructor* associé à la table racine. Ainsi, lorsqu'un token correspondra au motif de l'instruction `call`, la valeur de l'adresse de destination sera automatiquement déduite de ce dernier et utilisée dans le code assembleur généré. L'avantage de définir le traitement d'un type d'opérande avec un *constructor* qui lui est propre réside dans la factorisation : tout *constructor* y faisant référence

verra le même traitement appliqué, ce qui facilite la maintenance de la définition des opérandes et des motifs binaires des différentes instructions.

Listing 7: Exemple d'utilisation d'en-tête de table *SLEIGH*

```
# 9-bit jump immediate (pc-relative)
jaddr9: res is imm0812 & imm0406s
  [ res = ((imm0406s << 6) | (imm0812 << 1)) + inst_next; ]
  { export *:4 res; }

#
# call reladdr9
#
:call jaddr9 is group=4 & ins0707=0 & ins0003=0x1 & jaddr9
{
    # Set return address register
    rets = inst_next;
    call jaddr9;
}
```

Ce système de tables est très puissant et permet d'une part de structurer la définition des différentes instructions, notamment en les regroupant à l'aide d'en-têtes spécifiques, et d'autre part d'automatiser le décodage d'opérandes et le calcul des valeurs associées (adresses, valeurs immédiates, etc...). Il est même possible d'utiliser le symbole `instruction` pour forcer l'analyseur à procéder à une nouvelle évaluation d'une instruction en considérant un contexte modifié par un *constructor* précédemment évalué, permettant un décodage récursif d'une instruction. Ce mécanisme de récursivité permet de mettre en œuvre des algorithmes de décodage avancés, notamment basés sur des boucles de traitement, dans la limite du nombre de récursivités autorisées.

3.4 Le registre de contexte

La définition d'une instruction relativement simple comme celle réalisée dans la section précédente ne pose pas trop de problème, malgré la syntaxe quelque peu austère du langage *SLEIGH*. Ce n'est pas le cas de toutes les instructions, comme par exemple celles reposant sur une exécution conditionnelle ou encore celles employant des *bitmaps* pour spécifier un ensemble de registres employés en opérande. Ces deux exemples sont des cas concrets rencontrés durant la rétro-ingénierie des instructions de ce processeur ayant clairement donné du fil à retordre. Cependant, le langage *SLEIGH* offre plusieurs mécanismes permettant de prendre en compte ce genre d'instructions exotiques.

Le premier outil mis à disposition est appelé *registre de contexte*. Ce registre spécifique est constitué de champs identifiés par une portion de bits du registre en question, mais ne fait pas partie des registres de l'architecture. Il est seulement accessible *durant la phase de décodage des instructions* et réinitialisé entre chaque opération de décodage. Il peut être manipulé au travers des *actions de désassemblage*, et permet de conditionner le décodage d'un token ou de futures instructions en autorisant l'utilisation de ses champs dans le motif binaire des *constructors*.

Illustrons avec un exemple concret propre au processeur *Pi32v2*, l'exécution parallélisée d'instructions. Nous ne détaillerons pas ici le fonctionnement complet de ces instructions, mais plutôt la manière dont on peut les gérer via le langage *SLEIGH*. Considérons les deux instructions parallélisées suivantes :

```
1e0036c:    01 d6              r1 = r0  #
1e0036e:    42 60                   r2 = [r4+0]
```

La première instruction est du groupe 6 et correspond à une instruction du groupe 0 parallélisée avec l'instruction suivante. Nous définissons donc un registre de contexte qui contiendra un champ permettant de mémoriser le groupe réel de l'instruction en cours ainsi que deux champs booléens définis chacun sur un seul bit :

```
define context contextreg
    instgroup = (0, 2)
    phase = (3,3)
    parallel = (4,4)
;
```

Pour exécuter ces deux instructions en parallèle, il faut que leurs pseudo-codes soient combinés et donc qu'une des instructions soit exécutée avant l'autre, d'un point de vue *sémantique*. Cela passe par la définition d'un *constructor* récursif utilisant l'opérateur `^`, basé sur le champ `phase` du registre de contexte, qui ne sera analysé que si le groupe auquel l'instruction appartient est inférieur à 6 :

```
:^instruction is phase=0 & group<6 & instruction
[ phase = 1; instgroup=group; ]
{
    build instruction;
}
```

Les *actions de désassemblage* sont critiques dans ce dernier : si `phase` reste à 0, le décodage continue via le même *constructor*, et ce jusqu'à

saturation du niveau de récursivité. C'est pourquoi nous passons sa valeur à 1 via le code `phase = 1;` situé dans la section réservée aux *actions de désassemblage*, ce qui va éviter la récursivité. Le champ `instgroup` est ensuite défini, et peut être utilisé dans d'autres *constructors* qui seront ensuite analysés, jusqu'à ce que l'instruction soit correctement traitée. La syntaxe `^instruction` utilisée dans la partie `DISPLAY` du *constructor* impose un nouveau décodage de l'instruction après l'application des actions de désassemblage.

Dans le cas présent, le champ de contexte `instgroup` se voit attribuer la valeur du groupe auquel l'instruction appartient et le décodage est de nouveau effectué en considérant la nouvelle valeur de ce champ. Ainsi, tous les *constructors* qui font référence à ce champ `instgroup` seront évalués, les motifs binaires de ces derniers requérant que `phase` ait la valeur 1 et utilisant `instgroup` au lieu de `group`. De cette manière, le décodage se déroule en plusieurs phases, chaque phase correspondant à un ensemble de *constructors* spécifiques.

Cette construction permet d'implémenter des *wrappers* qui vont effectuer un pré-traitement de certaines instructions, puis laisser des *constructors* secondaires se charger du décodage et de la génération des mnémoniques et de leurs opérandes.

3.5 Parallélisation d'instructions

Nous pouvons nous occuper maintenant du cas particulier du groupe 6. Vu que ce *constructor* va lui aussi être récursif, nous devons prévoir un mécanisme similaire au précédent pour ne pas boucler sans fin :

```
1 :^instruction is phase=0 & parallel=0 & group=6 & instruction
2 [
3     parallel = 1;
4     instgroup = 0; globalset(inst_next, instgroup);
5 ]
6 {
7     delay_slot(1);
8     build instruction;
9 }
```

Si une instruction du groupe 6 est rencontrée, ce *constructor* va activer le bit `parallel` du registre de contexte, paramétrer `instgroup` à 0, puis évaluer à nouveau l'instruction. Les conditions de notre premier *constructor* sont alors satisfaites, et l'instruction est décodée comme si c'était une instruction du groupe 0, ce qui produit le bon affichage pour celle-ci. Cependant, les *actions sémantiques* associées à cette première instruction

font appel à `delay_slot()`, une opération qui demande le décodage de l'instruction suivante et l'insertion de son pseudo-code en lieu et place de celle-ci. L'instruction suivante est donc décodée (le registre de contexte étant réinitialisé mais avec le champ `instgroup` conservé), ce qui permet de désassembler la seconde instruction en court-circuitant la détection d'exécution parallélisée. Son pseudo-code est inséré avant celui de la première instruction, et l'évaluation de la première instruction se termine. Au final, deux instructions ont été décodées et désassemblées, mais la représentation intermédiaire de la seconde a été intégrée à celle de la première. Le registre de contexte a quant à lui fait office de registre d'état lors du décodage des deux instructions, permettant de forcer le traitement d'une instruction en plusieurs étapes successives.

3.6 Implémenter des boucles à l'aide de la récursivité

Le mécanisme de récursion offert par l'emploi de l'opérateur ^ couplé à un registre de contexte permet d'implémenter de véritables automates déterministes à états finis, très utiles dans certains cas particuliers. Ces automates sont toutefois limités par le nombre maximum de récursions autorisées par le moteur de décodage, mais cela ne pose pas de problème dans la grande majorité des cas.

Nous avons dû développer un tel automate afin de gérer proprement un type d'opérande utilisé par certaines instructions relatives à la gestion de la pile, reposant sur une *bitmap* indiquant les registres devant être empilés ou dépilés, selon l'instruction considérée. Autrement dit, nous devons générer à partir d'une valeur représentant une telle *bitmap* une liste constituée des noms des registres sélectionnés séparés par des virgules, qui sera intégrée dans la représentation textuelle du *constructor* d'une instruction. Par exemple, nous devons être capable de générer l'instruction *push {r0, r1, r4}* à partir de sa version encodée.

On commence ainsi par définir le *constructor* de notre instruction, dont le motif binaire définit les champs du premier token correspondant à cette dernière :

```
1 :push {pshmap} is group=7 & ins0011=0x8D8 ; pshmap
2 {
3     mult_addr = sp;
4     build pshmap
5     sp = mult_addr;
6 }
```

Le caractère `;` présent dans la définition du motif binaire indique au moteur de désassemblage qu'il doit consommer un token supplémentaire, dont le décodage sera intégré dans l'opérande de l'instruction `push`. Les accolades présentes dans la section `DISPLAY` du *constructor* sont traitées comme de simples caractères et seront intégrées telles quelles dans le code assembleur généré.

Le pseudo-code quant à lui utilise un champ du registre de contexte pour mémoriser la valeur du registre de pile, et force l'évaluation du pseudo-code du *constructor pshmap* avant de restaurer la valeur du registre de pile. L'idée derrière cette série d'opérations consiste à amener Ghidra à considérer d'une part que la valeur du registre de pile a été modifiée, et d'autre part de laisser toute liberté au *constructor pshmap* de modifier le champ `mult_addr`, dont la valeur *après évaluation de ce dernier* sera transférée dans le registre de pile `sp`.

C'est donc la structure `pshmap` qui est en charge de récupérer chaque bit constituant la valeur passée en paramètre à l'instruction, l'évaluer, inclure le registre correspondant dans la liste, générer le pseudo-code et faire de même avec les bits restants. En somme, il s'agit d'implémenter un mécanisme de boucle traitant l'ensemble des bits et construisant la représentation textuelle et sémantique propre à l'instruction à décoder.

Comme pour tout algorithme reposant sur une boucle, nous avons besoin d'une variable `counter` utilisée comme compteur. Cette variable sera aussi utilisée pour générer un masque binaire afin de tester la valeur d'un bit précis. Deux variables supplémentaires seront nécessaires : une pour stocker le résultat de l'évaluation du bit courant (`bitset`) et une seconde pour éviter l'ajout d'une virgule au début de la liste des registres (`sep`). Ces variables sont définies au sein du registre de contexte :

```
1  define context contextreg
2
3      ...
4
5      # Used to generate registers list from bitmap
6      counter = (7,10)
7      sep = (11,11)
8      bitset = (12,12)
9
10     ...
```

Notre boucle doit effectuer les opérations suivantes :

1. évaluer si le bit à la position `counter` est à 1 ;

2. s'il est à 0 on ne fait rien, sinon on ajoute le nom du registre à la liste (le pseudo-code est aussi généré à cette étape) ;
3. on incrémente le compteur si ce dernier n'a pas atteint 15.

Il nous faut donc définir ces différentes opérations atomiques à l'aide de *constructors*. Le Listing 8 détaille l'implémentation des opérations `mregread`, `msep`, `next`, `pshmapreg` et `pshmreg` au travers de leurs tables dédiées. En fonction de la valeur de `counter`, ces dernières vont permettre de déterminer si le bit à la position défini par ce compteur est à 1 et de générer le pseudo-code associé. Par ailleurs, dès qu'un registre a son bit défini, la variable `sep` passe à 1 afin qu'une virgule soit automatiquement ajoutée avant chaque nom de registre.

Il ne reste plus qu'à définir une instruction récursive qui va réaliser les différentes opérations dans l'ordre, en exploitant une propriété essentielle du moteur de désassemblage : l'ordre d'évaluation des conditions du motif binaire des *constructors*. En effet, l'évaluation du motif binaire d'un *constructor* se fait de gauche à droite, et les actions de désassemblage correspondant à d'autres *constructors* sont aussi évaluées successivement. La Fig. 24 illustre cette évaluation successive et la manière dont la récursivité permet d'effectuer des boucles d'évaluation.

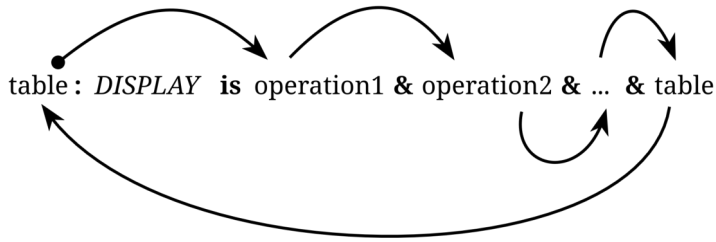


Fig. 24. Exemple de *constructor* récursif implémentant plusieurs opérations.

Cette particularité nous permet donc de définir notre *constructor* `pshmapregs` :

```
1 pshmapregs: pshmapregs^pshmreg^msep is counter<15 & mregread & msep
  ↪  & pshmreg & next & pshmapregs
2     { build pshmapregs; build pshmreg; }
3 pshmapregs: pshmreg is counter=15 & mregread & pshmreg {}
4 pshmap: pshmapregs is pshmapregs [bitset=0; counter=0; sep=0;] {}
```

C'est ici que toutes les briques sont assemblées. L'évaluation du seul *constructor* de la table `pshmap` met les variables `bitset`, `counter` et `sep` du registre de contexte à zéro, puis lance l'évaluation des *constructors* de la table `pshmapregs`, au nombre de deux. La variable `counter` valant zéro, l'évaluation des *constructors* des tables `mregread`, `msep`, `pshmreg` et `next` est effectuée dans cet ordre. De fait, le bit 0 de la bitmap est lu puis la variable `sep` mise à jour si besoin est, le pseudo-code correspondant à l'empilement du registre (si le bit correspondant est à 1) est pris en compte et le nom du registre est inséré dans la représentation littérale. Enfin, la variable `counter` est incrémentée et la référence à `pshmapregs` dans le motif binaire provoque une nouvelle évaluation, cette fois avec la variable `counter` à 1. Il en va de même jusqu'à ce que la variable `counter` atteigne 15, amenant l'évaluation du second *constructor* de la table `pshmapregs` et l'arrêt de la récursion. Le moteur dépile ensuite les éléments littéraux produits par chaque évaluation récursive, construisant la liste des registres en fonction de la bitmap.

Ce genre de construction est assez complexe à concevoir et déboguer, mais elle montre qu'il est possible d'implémenter des algorithmes de décodage complexes malgré les limitations du langage *SLEIGH*.

Listing 8: Fonctions de base de traitement de bitmap

```
# Macro définissant la façon dont les registres sont placés
# en pile
macro pushreg(reg) {
    mult_addr = mult_addr - 4; *mult_addr = reg;
}

# La table `pshmapreg` définit tout un ensemble de constructeurs
# en fonction de la valeur de `counter` (index de registre).
pshmapreg: r0 is counter=0 & r0 {pushreg(r0);}
pshmapreg: r1 is counter=1 & r1 {pushreg(r1);}
pshmapreg: r2 is counter=2 & r2 {pushreg(r2);}
pshmapreg: r3 is counter=3 & r3 {pushreg(r3);}
pshmapreg: r4 is counter=4 & r4 {pushreg(r4);}
pshmapreg: r5 is counter=5 & r5 {pushreg(r5);}
pshmapreg: r6 is counter=6 & r6 {pushreg(r6);}
pshmapreg: r7 is counter=7 & r7 {pushreg(r7);}
pshmapreg: r8 is counter=8 & r8 {pushreg(r8);}
pshmapreg: r9 is counter=9 & r9 {pushreg(r9);}
pshmapreg: r10 is counter=10 & r10 {pushreg(r10);}
pshmapreg: r11 is counter=11 & r11 {pushreg(r11);}
pshmapreg: r12 is counter=12 & r12 {pushreg(r12);}
pshmapreg: r13 is counter=13 & r13 {pushreg(r13);}
pshmapreg: r14 is counter=14 & r14 {pushreg(r14);}
pshmapreg: r15 is counter=15 & r15 {pushreg(r15);}

# Incrémente `counter` (epsilon matche n'importe quel pattern)
next: is epsilon [counter=counter+1;]{}

# Met `bitset` à 1 si le counter-ième bit est à 1, sinon à 0
# (évaluation du bit selon la position courante)
mregread: is imm1631 [bitset=(imm1631 & (1 <<
↪  counter))>>counter;]{}

# Génère le caractère de séparation en fonction de
# `bitset` et `sep`
msep: "," is  sep=1 & bitset=1 {}
msep: "" is sep=0 | bitset=0 {}

# Si `bitset` est à 1, traite le registre associé
pshmreg: pshmapreg is pshmapreg & bitset=1 [sep=1;]{}
```

3.7 Désassemblage et décompilation avec *Ghidra*

Une fois le jeu d'instructions du processeur défini à l'aide d'un fichier `slaspec`, *Ghidra* est en mesure de le traiter lors de son démarrage et de produire une transcription dans un format binaire et plus compact de ce

dernier, stocké dans un fichier `sla`. C'est cette version qui est utilisée par le moteur de *Ghidra* pour le désassemblage des instructions d'un exécutable.

Une fois le processeur sélectionné et le processus d'analyse de l'exécutable terminé, nous pouvons observer la bonne traduction du code exécutable dans un langage assembleur, mais aussi que le décompilateur arrive à *comprendre* ce que font ces instructions et à générer un code C équivalent pour l'ensemble des fonctions identifiées (cf. Fig. 25).

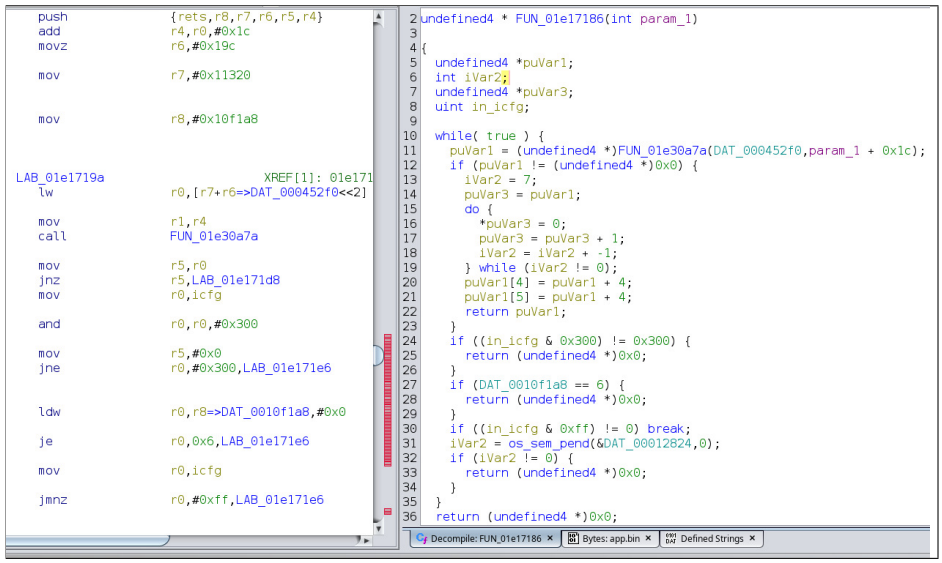


Fig. 25. Instructions *Pi32v2* désassemblées dans Ghidra et code C de la fonction correspondante

4 Analyse du micrologiciel et identification du code recherché

Après cette douloureuse étape de rétro-ingénierie du jeu d'instructions du processeur *Pi32v2* et la définition de celui-ci à l'aide du langage *SLEIGH*, il est désormais temps d'analyser le micrologiciel en question. Certaines instructions semblent ne pas être correctement désassemblées, signe qu'il reste encore une partie du jeu d'instructions à déterminer. Cela ne gêne cependant pas trop l'analyse, la grande majorité des fonctions ne faisant pas appel à ces instructions. Nous cherchions en particulier à déterminer de quelle manière la montre connectée étudiée arrivait à

mesurer des constantes de santé telles que la fréquence cardiaque ou le taux d'oxygénation du sang, car nous supposions que ces dernières avaient de grandes chances d'être basées sur de l'aléatoire à cause de l'absence de capteurs dans l'électronique de la montre.

Nous avons ainsi pu retrouver la présence de fonctions permettant de formater des chaînes de caractères, dont celle formatant les valeurs relatives à la pression sanguine : 3 chiffres pour la pression systolique, 2 pour la pression diastolique (Listing 11). À partir de là, nous sommes remontés à l'aide des références croisées jusqu'au code générant la valeur utilisée pour l'affichage, pour nous rendre compte que toutes ces valeurs reposaient en réalité sur une graine aléatoire et un savant calcul permettant de s'assurer qu'elles se situent dans une plage acceptable. Le Listing 12 montre le cas particulier du calcul de la fréquence cardiaque, qui est mathématiquement située entre `0x41` et `0x41 + 0xf`, soit 65 et 80 en décimal, ce que l'on observe aisément au regard des valeurs minimales et maximales affichées par la montre (Fig. 26).

La fonction de génération de valeur aléatoire présentée dans le Listing 9 correspond à l'implémentation de la fonction `rand()` de la bibliothèque *newlib* [7], qui est définie dans le Listing 10. L'utilisation de la constante décimale `6364136223846793005`, soit `0x5851f42d4c957f2d` sous sa forme hexadécimale, est relativement flagrante dans l'implémentation identifiée dans le micrologiciel. Ce n'est par ailleurs pas très surprenant, la bibliothèque *newlib* est très souvent employée dans les chaînes de compilation des systèmes embarqués.

Listing 9: Code décompilé de la fonction de génération de nombres aléatoires trouvée dans le micrologiciel

```
uint32_t rand(void)
{
  int rand_state;
  longlong lVar1;
  uint uVar2;

  rand_state = (int)*(undefined8 *)(_random_state + 0xa04);
  lVar1 = (longlong)rand_state * 0x4c957f2d;
  uVar2 = rand_state * 0x5851f42d + (int)((ulonglong)lVar1 >> 0x20)
  ↪  +
          (int)((ulonglong)*(undefined8 *)(_random_state + 0xa04)
  ↪  >> 0x20) * 0x4c957f2d;
  *(ulonglong *)(_random_state + 0xa4) = CONCAT44(uVar2,(int)lVar1
  ↪  + 1);
  return uVar2 & 0x7fffffff;
}
```

Listing 10: Définition de la fonction `rand()` dans *newlib*

```
int
rand (void)
{
  struct _reent *reent = _REENT;

  /* This multiplier was obtained from Knuth, D.E., "The Art of
     Computer Programming," Vol 2, Seminumerical Algorithms, Third
     Edition, Addison-Wesley, 1998, p. 106 (line 26) & p. 108 */
  _REENT_CHECK_RAND48(reent);
  _REENT_RAND_NEXT(reent) =
     _REENT_RAND_NEXT(reent) * __extension__ 6364136223846793005LL
  ↪  + 1;
  return (int)((_REENT_RAND_NEXT(reent) >> 32) & RAND_MAX);
}
```

Listing 11: Formatage des valeurs de pression artérielle

```
if ((DAT_00015a0d == '\x01') || (DAT_00015a0e == '\x01')) {
  /* Formatage des valeurs avec "%03d/%02d" */
  _sprintf(
    &s_bp_systolic,
    s_%03d/%02d_01eb8740,
    (uint)blood_pressure_systolic,
    (uint)blood_pressure_diastolic
  );

  /* Dessin des valeurs à l'écran. */
  puVar1[2] = 0xffff;
  puVar1[1] = (uint32_t)&s_bp_systolic;
  *puVar1 = 0x23;
  draw_text(param_1,0x193,0x30,0x31,*puVar1,puVar1[1],puVar1[2]);

  /* ... */
```

Listing 12: Génération aléatoire de la fréquence cardiaque et bornage de la valeur entre 65 et 80

```
uVar2 = randint();
heart_rate_current = ((char)uVar2 - (char)(((int)uVar2 / 0x10 &
↪  0xffU) << 4)) + 0x41;
if (heart_rate_max <= heart_rate_current) {
  heart_rate_max = heart_rate_current;
}
uVar9 = 0;
uVar17 = 0;
bVar16 = heart_rate_min;
if (heart_rate_current <= heart_rate_min) {
  bVar16 = heart_rate_current;
}
bVar12 = heart_rate_current;
if (heart_rate_min != 0) {
  bVar12 = bVar16;
}
```

Ces observations confirmèrent le caractère aléatoire de cette valeur, ainsi que le qualificatif « arnaque » que l'on pouvait dès lors associer à cette montre connectée. La phase d'analyse du code exécutable a été relativement courte comparée au temps requis pour identifier l'ensemble du jeu d'instructions de ce processeur, pour un résultat certes fortement supposé mais désormais prouvé.

Fig. 26. Valeurs minimales et maximales de fréquence cardiaque affichées par la montre connectée

5 Conclusion

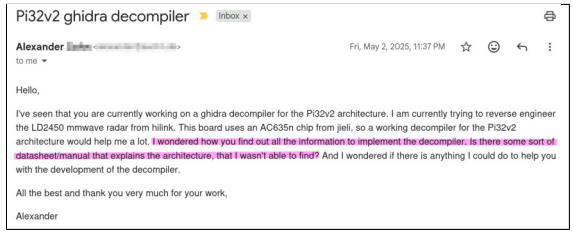
La rétro-ingénierie d'instructions de processeurs inconnus n'est pas une tâche évidente et requiert un bon sens de l'analyse ainsi que la connaissance des principales méthodes d'encodage d'instructions. Le processeur *Pi32v2* avait déjà été étudié par un autre chercheur, ce qui nous a grandement aidé lors de l'ajout du support des instructions manquantes. L'existence d'une chaîne de compilation librement téléchargeable a aussi été d'une aide considérable. Au final, ce sont 218 instructions supplémentaires qui ont été ajoutées aux 80 instructions déjà supportées par l'implémentation originale.

Il est d'ailleurs assez cocasse de noter qu'une personne ayant observé la création d'un dépôt *Github* dérivé de celui de Grigoryev, alors que nous étions en plein travail sur la rétro-ingénierie des instructions manquantes, nous a envoyé un courriel demandant comment nous avions réussi à trouver le format des instructions manquantes (Fig. 27). Nous espérons sincèrement que cet article et l'implémentation améliorée que nous proposons aideront les lecteurs et Alexander à mieux comprendre la méthode appliquée ainsi que les problématiques généralement rencontrées dans ce type d'exercice.

L'implémentation du processeur *Pi32v2* que nous avons réalisée est disponible sur notre dépôt *Github* [2] sous licence libre, et devrait être soumise pour intégration au dépôt d'Andrey Grigoryev.

Références

1. Damien Cauquil. A modern tale of blinkenlights. `https://blog.quarkslab.com/modern-tale-blinkenlights.html`, 2026.
2. Damien Cauquil. ghidra-jieli. `https://github.com/virtualabs/ghidra-jieli`, 2026.


Pi32v2 ghidra decompiler Inbox ×

Alexander [redacted] Fri, May 2, 2025, 11:37 PM

to me

Hello,

I've seen that you are currently working on a ghidra decompiler for the Pi32v2 architecture. I am currently trying to reverse engineer the LD2450 mmwave radar from hilink. This board uses an AC635n chip from jieli, so a working decompiler for the Pi32v2 architecture would help me a lot. I wondered how you find out all the information to implement the decompiler. Is there some sort of datasheet/manual that explains the architecture, that I wasn't able to find? And I wondered if there is anything I could do to help you with the development of the decompiler.

All the best and thank you very much for your work,

Alexander

Fig. 27. Courriel reçu courant mai 2025, lorsque nous travaillions sur les définitions des instructions du processeur *Pi32v2*

3. Thomas Cougnard Damien Cauquil. Fun with watches. `https://github.com/quarkslab/conf-presentations/blob/master/Confs/LeHack25/lehack25_fun-with-watches_dcauquil_xilokar.pdf`, 2025.
4. Andrey Grigoryev. JieLi STUFF. `https://kagaimiq.github.io/jielie/`, 2022.
5. Andrey Grigoryev. ghidra-jieli. `https://github.com/kagaimiq/ghidra-jieli`, 2024.
6. Andrey Grigoryev. pi32v2. `https://kagaimiq.github.io/jielie/cpu/pi32v2.html`, 2024.
7. Cygnus Support. Sourceware's newlib. `https://www.sourceware.org/newlib/`, 2024.
8. Ghidra Development Team. SLEIGH, Constructors. `https://ghidra.re/ghidra_docs/languages/html/sleigh_constructors.html`, 2026.
9. JieLi Tech. Bienvenue dans la documentation de Jerry Tools. `https://doc.zh-jieli.com/Tools/zh-cn/index.html`, 2026.
10. JieLi Tech. fw-AC63_BT_SDK. `https://github.com/Jieli-Tech/fw-AC63_BT_SDK/`, 2026.
11. Guillaume Valadon. Reversing a Japanese Wireless SD Card From Zero to Code Execution. `https://i.blackhat.com/us-18/Wed-August-8/us-18-Valadon-Reversing-a-Japanese-Wireless-SD-Card-From-Zero-to-Code-Execution.pdf`, 2018.
12. Willem. Reverse engineering the Pixel TitanM2 firmware. `https://media.ccc.de/v/39c3-reverse-engineering-the-pixel-titanm2-firmware`, 2025.
13. Robert Xiao. [DSCTF 2019] CPU Adventure – Unknown CPU Reversing. `https://www.robertxiao.ca/hacking/dsctf-2019-cpu-adventure-unknown-cpu-reversing/`, 2019.