Rétro-ingénierie d’un micrologiciel pour architecture Pi32v2

Damien Cauquil

4 Juin 2026

SSTIC, Rennes (France)

Introduction

Quarkslab

/whoami

Damien Cauquil

- Ingénieur Sécurité à Quarkslab
- Team Cryptobedded
- Reverse hardware/software

Tout a commencé en 2025 ...

Xilokar & Virtu

present

FUN with Watches

LeHACK 2025

Une montre connectée *de génie*

Opération 2 +1 offert sur les articles signalés ►Voir conditions◄

MENU GiFi Qu'est-ce qui vous ferait plaisir aujourd'hui ? GIFI Mon compte Panier

Plein air | Jardin & Extérieur | Offres du moment | Nos produits | Nos petits prix | Nos nouveautés | Nos catal

Indiquez votre magasin préféré pour voir la disponibilité des produits

Code postal ou ville

Accueil > Loisirs > Multimédia et divertissement > Informatique > Montre connectée Homday Xpert bluetooth étanche écran tactile 1,83"

Vendu par GiFi

Montre connectée Homday Xpert bluetooth étanche écran tactile 1,83"

11€90

Dont 0,1€ d'éco-part

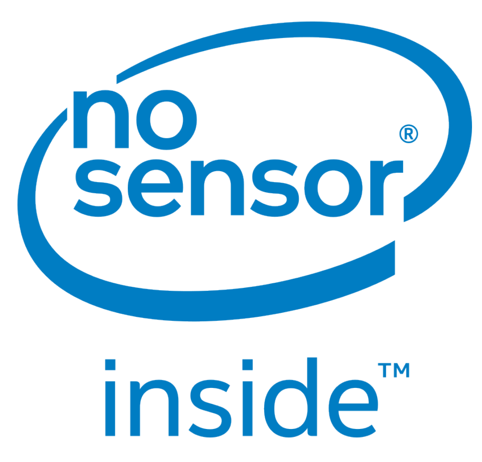
Caractéristiques du produit

- Compatibilité : **android/IOS**
- Dimensions cadran : **3,5 x 4 cm**
- Écran : **1,83 pouces**
- Entrée : **5V [DC] 0,2A**
- Batterie : **3,7V [DC] 200mAh**
- Matière(s) : **plastique ABS, alliage de zinc, silicone**

Description du produit

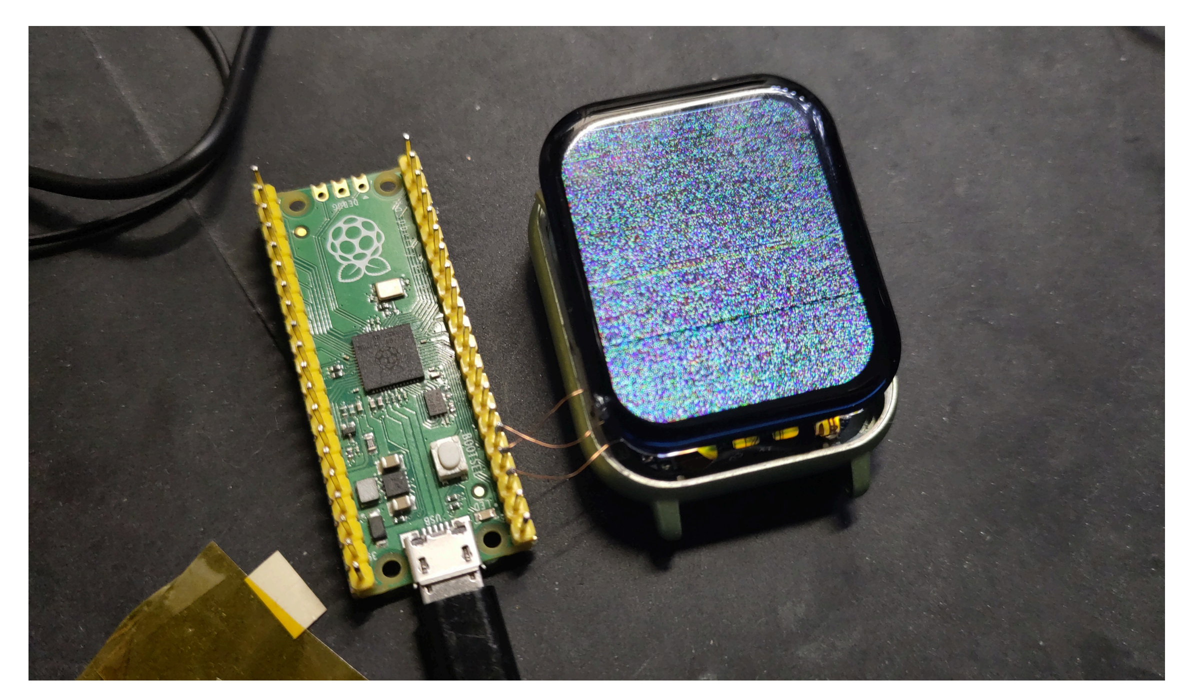
Découvrez la montre connectée Homday Xpert, un compagnon idéal pour un mode de vie actif et connecté !
Avec son écran tactile de 1,83 pouces, cette montre vous offre une interface intuitive et facile à naviguer.
Sa conception étanche vous permet de la porter sans souci lors de vos activités quotidiennes, qu'il pleuve ou que vous soyez en pleine séance d'entraînement. Grâce à

Le problème avec cette montre ?


no
sensor®
inside™

Ref. : 630391
Entrée : 5V ⎓ 0.2A

Extraction du micrologiciel



https://blog.quarkslab.com/modern-tale-blinkenlights.html

Surprise, CPU custom !

```
                    ***************************************************************
                    undefined FUN_01e17186()
      undefined       r0:1          <RETURN>
                    FUN_01e17186                                   XREF[1]:     FUN_01e0ac94:01e0ac9c(c)
    01e17186 78 04         push        {0x8}
    01e17188 0c 9c         add         r4,r0,#0x1c
    01e1718a 46 e0 9c 01   movz        r6,#0x19c
    01e1718e c7 ff 20      mov         r7,#0x11320
             13 01 00
    01e17194 c8 ff a8      mov         r8,#0x10f1a8
             f1 10 00
    01e1719a d8            ??          D8h
    01e1719b ec            ??          ECh
    01e1719c 7a            ??          7Ah    z
    01e1719d 06            ??          06h
    01e1719e 41            ??          41h    A
    01e1719f 16            ??          16h
    01e171a0 80            ??          80h
    01e171a1 ea            ??          EAh
    01e171a2 6b            ??          6Bh    k
    01e171a3 cc            ??          CCh
```

Live and let die

Reverser un jeu d’instructions inconnu (ou presque)

Quarkslab

JieLi et son Pi32v2

- **Architecture CPU 32-bit** historiquement dérivée de *Blackfin* (DSP créé par *Analog Devices*)
- Instructions sur **16, 32 et 48 bits**
- *Toolchain* référencée sur Github 🤩
- *Processeur Ghidra* incomplet créé par Andrey Grigoryev (2022)
- **Beaucoup d'instructions manquantes** / non-documentées

https://kagaimiq.github.io/jielie/

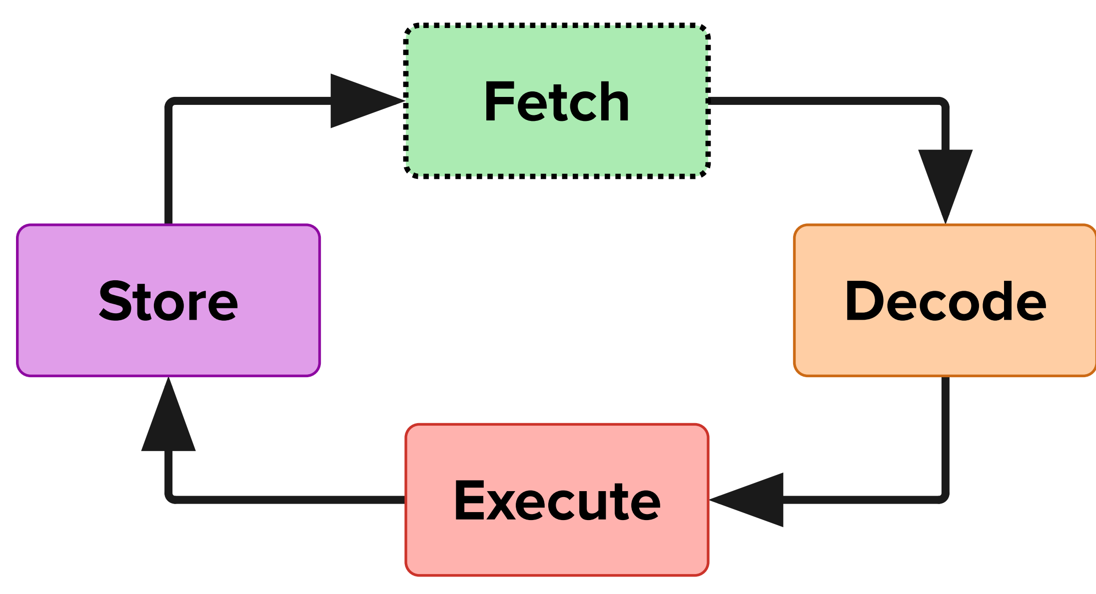
Compilation, symboles, désassemblage

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

Disassembly of section .text:
text_code_begin:
 1e00100:    81 ea 29 bd        call 227922 <boot_info_init : 1e37b56 >
 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}
 1e00114:    c0 ff f0 cb eb 01  r0 = 32230384 <psram_laddr : 1ebcbf0 >
 1e0011a:    c1 ff 00 00 80 00  r1 = 8388608 <psram_vaddr : 800000 >
 1e00120:    c2 ff 00 00 00 00  r2 = 0 <test_encode_main.c : 0 >
 1e00126:    a2 a2              r2 = r2 >> 2
 1e00128:    12 03              rep 4 r2 {
 1e0012a:    03 05                r3 = [r0++=4]
 1e0012c:    93 05                [r1++=4] = r3
```

Compilation, symboles, désassemblage

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

Disassembly of section .text:
text_code_begin:
 1e00100:    81 ea 29 bd        call 227922 <boot_info_init : 1e37b56
 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}
 1e00114:    c0 ff f0 cb eb 01  r0 = 32230384 <psram_laddr : 1ebcbf0
 1e0011a:    c1 ff 00 00 80 00  r1 = 8388608 <psram_vaddr : 800000 >
 1e00120:    c2 ff 00 00 00 00  r2 = 0 <test_encode_main.c : 0 >
 1e00126:    a2 a2              r2 = r2 >> 2
 1e00128:    12 03              rep 4 r2 {
 1e0012a:    03 05                r3 = [r0++=4]
 1e0012c:    93 05                [r1++=4] = r3
```

Cycle de l'instruction



Important

Ce schéma est **très** simplifié, les CPUs actuels sont plus complexes…

Méthodologie

1. Identifier les différentes occurrences d'une instruction (*add*, *sub*, ...)
2. Identifier les différentes formes (16, 32 ou 48 bits)
3. Pour chaque forme, identifier les bits fixes
4. Déterminer la façon dont les opérandes sont encodées

Identification des variantes

Octets	Mot(s)	Opération
`40 20`	`0x2040`	`r0 = 0`
`40 e0 5b 13`	`0xe040 0x135b`	`r0 = 4955`
`c0 ff 80 86 00 00`	`0xffc0 0x8680 0x0000`	`r0 = 34432`

Astuce

Le premier mot de 16 bits permet d'identifier une possible extension de l'instruction à 32 ou 48 bits

Identification des variantes

Octets	Mot(s)	Opération
`40 20`	`0010000001000000`	`r0 = 0`
`40 e0 5b 13`	`1110000001000000 0001001101011011`	`r0 = 4955`
`c0 ff 80 86 00 00`	`1111111111000000 1000011010000000 0000000000000000`	`r0 = 34432`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Groupe			*P*	*Code opération*											

Format des instructions du Pi32v2

16 bits		32 bits	48 bits
Groupe 0	Groupe 4	Groupe 7A	Groupe 7G
Groupe 1	Groupe 5	Groupe 7B	
Groupe 2	Groupe 6	Groupe 7C	
Groupe 3		Groupe 7D	
		Groupe 7E	

instructions parallélisables

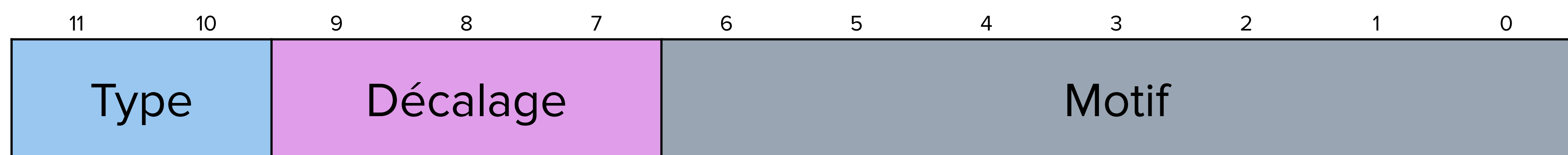
Format des opérandes

Octets	Mot(s)	Opération
`40 20`	`0010000001000000`	`r0 = 0`
`41 e0 5b 13`	`1110000001000001 0001001101011011`	`r1 = 4955`
`c0 ff 80 86 00 00`	`1111111111000010 1000011010000000 0000000000000000`	`r2 = 34432`

Format des opérandes

Octets	Mot(s)	Opération
40 20	0010000001000000	r0 = 0
41 e0 5b 13	1110000001000001 0001001101011011	r1 = 4955
c0 ff 80 86 00 00	1111111111000010 1000011010000000 0000000000000000	r2 = 34432

Encodage spécial (masque binaire)



- Le codage dépend du **type**
- Le **décalage** est optionnel
- Le **motif** décrit la valeur de base
- On peut coder `0x000000FF`, `0xFF00FF00` ou `0x00CC0000` sur 12 bits !

Amélioration du *processeur* Ghidra

Quarkslab

Désassemblage & décompilation

- Ghidra suit en partie le **cycle de l’instruction** (*fetch* et *decode*)
- Chaque instruction connue possède son *pseudo-code*
- Progression linéaire dans le *bytecode*
- Génération d’un **arbre de la syntaxe abstraite** (AST) par fonction
- Simplification et production du code C équivalent

SLEIGH: la joie

- Langage de modélisation d’architecture CPU, dérivé de *SLED*;
- La syntaxe est vraiment **austère et peu avenante** 😖;
- Techniquement, **c’est très puissant** mais difficile à maîtriser.

Et on cherche toujours le livre de recettes, qui n’existe pas… 😡

Définition du processeur et de ses registres

CPU little-endian, mot de 16 bits

```sleigh
1  define endian    = little;
2  define alignment = 2;
```

16 registres généraux de 32 bits

```sleigh
1  define register offset=0x0 size=4
2  [ r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 ];
```

Définition d’une instruction

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

- `TABLE`: nom de la *table* (optionnel)
- `DISPLAY`: mnémonique + opérandes
- `PATTERN`: motif de bits *(décodage)*
- `DISASSEMBLY`: actions de désassemblage
- `SEMANTIC`: *p-code* de l’instruction

mov reg, #imm8

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

```
:mov                     is group=1 & ins0607=1
[                                     ]
{


}
```

mov reg, #imm8

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

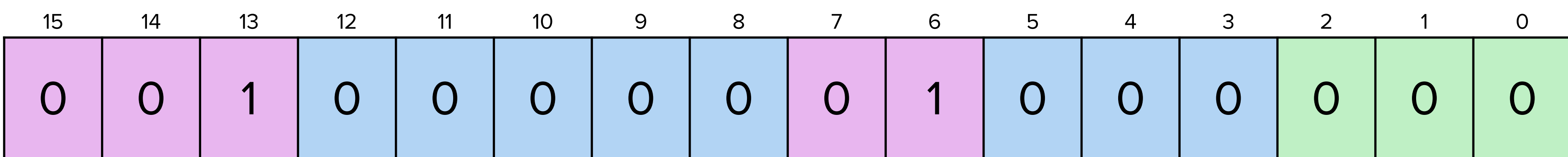
```
:mov regAl          is group=1 & ins0607=1 & regAl
[                                    ]
{

}
```

mov reg, #imm8

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

```
:mov regAl          is group=1 & ins0607=1 & regAl & imm0305 & imm0812
[ imm8 = (imm0305 << 5) | imm0812; ]
{

}
```

mov reg, #imm8




```
:mov regAl, #imm8 is group=1 & ins0607=1 & regAl & imm0305 & imm0812
[ imm8 = (imm0305 << 5) | imm0812; ]
{
   regAl = imm8
}
```

Oui mais, pour les bitmaps de registres ?

```
d9 e8 d0 00          [--sp] = {rets, r7, r6, r4}          Pi32v2
```

- `e8d9`: caractérise l'instruction **`push rets, reg0, reg1, ...`**
- **`0x00d0`**: *bitmap* qui encode les registres `r0` à `r15`

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

Oui mais, pour les bitmaps de registres ?

```
d9 e8 d0 00          [--sp] = {rets, r7, r6, r4}          Pi32v2
```

- `e8d9`: caractérise l'instruction **`push rets, reg0, reg1, ...`**
- **`0x00d0`**: *bitmap* qui encode les registres `r0` à `r15`

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

Comment générer une liste de registres en SLEIGH ?

Algorithme de base (boucle *for*)

Init

`i=0, regs=''`

i < 16 ?

oui

Bit *i* à 1 ?

non

Passe au bit suivant

`i++`

oui

Ajout registre

`regs += ',r$i'`

non

Retourne *regs*

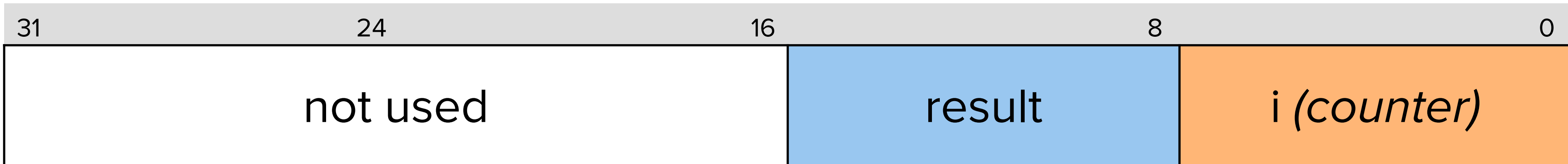
Des boucles en SLEIGH ? 🤯

- Format de **description**
- **Pas de scripting** possible
- Prévoir **tous les cas** ?

Accrochez-vous à vos sièges, ça va être un peu speed...

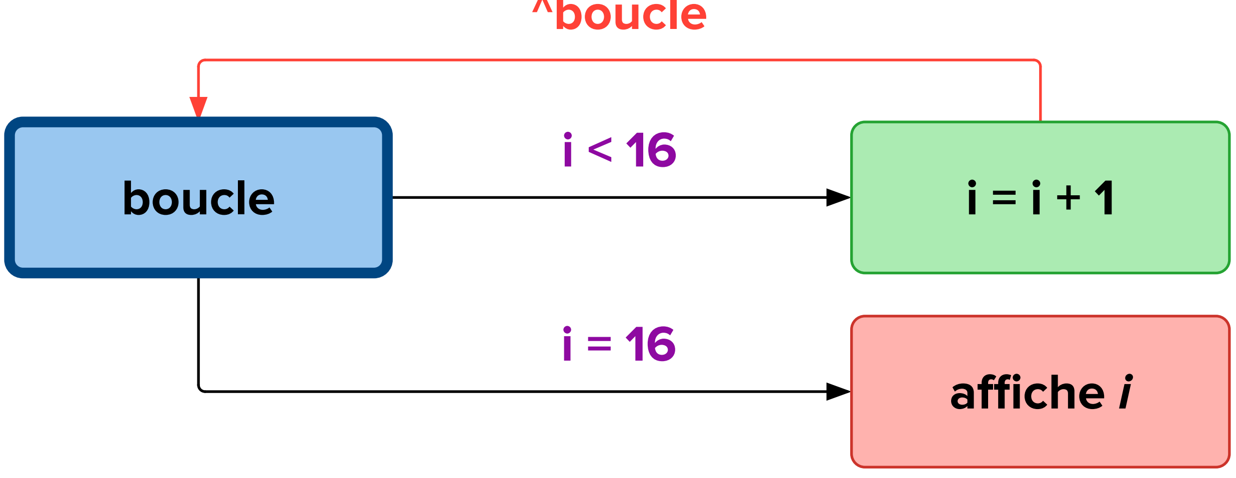
Variables de désassemblage

- SLEIGH permet de déclarer un **Context Register**
- Ce *registre spécial* contient des champs de bits (variables)
- Il est remis à zéro avant le décodage d'une instruction



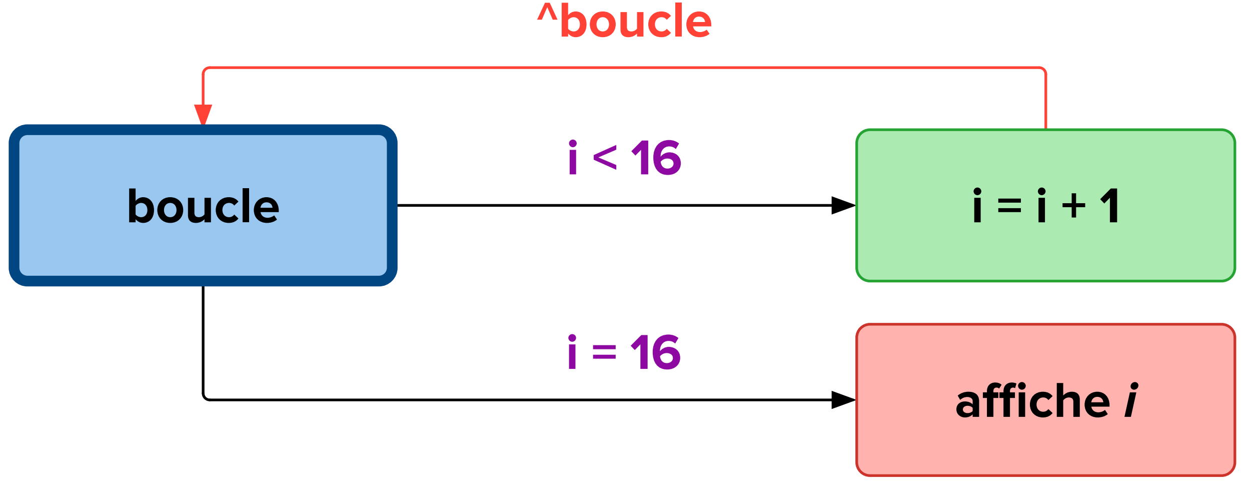
Récursivité

```
boucle:^boucle is i < 16 & boucle [ i=i+1; ] { }
boucle:i is i = 16 [ ] { }
```



Récursivité

```
boucle:^boucle is i < 16 & boucle [ i=i+1; ] { }
boucle:i is i = 16 [ ] { }
```



Ordre d’évaluation du masque

```
op1: is epsilon [ result=result+i; ]{}

op2: is epsilon [ result=result*2; ]{}

boucle:^boucle is i < 15 & op1 & op2 & ... [ i=i+1;result=0; ]{}
```

- Evaluation du **masque** de gauche à droite
- **op1** est évaluée, puis **op2**, puis …
- **epsilon** matche dans tous les cas
- On peut ainsi **chaîner des opérations** qui altèrent le *Context Register*

Nos briques de base

```
next_bit: is epsilon [ i=i+i; ] { }

test_bit: is imm1631 [ bit=(imm1631 & (1 << i))>>i; ] { }

pushreg: is bit=0 [ ] { }
pushreg:r0^" " is i=0 & bit=1 [ ] { push(r0); }
pushreg:r1^" " is i=1 & bit=1 [ ] { push(r1); }

...
```

Et l’assemblage (version simplifiée)

```
bitmap:^bitmap^pushreg is i<15 & test_bit & pushreg & next_bit
[ i=0; bitset=0; ]{}

bitmap:pushreg is i=15 & test_bit & pushreg [ ]{}
```

- On teste le *i*-ème bit **tant que i est strictement inférieur à 15** ;
- génère le nom du registre en fonction de la variable `bit` ;
- incrémente la variable `i` ;
- appel récursif tant que `i` est inférieur à 15 *(condition de fin)*.

Le résultat obtenu

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

iteration	display	context register

Le résultat obtenu

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

iteration	display	context register
`0`	`r0`	`i=0; bit=`**`1`**`;`

Le résultat obtenu

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

iteration	display	context register
`0`	`r0`	`i=0; bit=1;`
`1`	`r0`	`i=1; bit=`**`0`**`;`

Le résultat obtenu

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

iteration	display	context register
0	`r0`	`i=0; bit=1;`
1	`r0`	`i=1; bit=0;`
2	`r2 r0`	`i=2; bit=`**`1`**`;`

Le résultat obtenu

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

iteration	display	context register
0	`r0`	`i=0; bit=1;`
1	`r0`	`i=1; bit=0;`
2	`r2 r0`	`i=2; bit=1;`
…	`r2 r0`	`i=…; bit=0;`
15	`r2 r0`	`i=15; bit=0;`

Surprise: un *automate* déterministe !

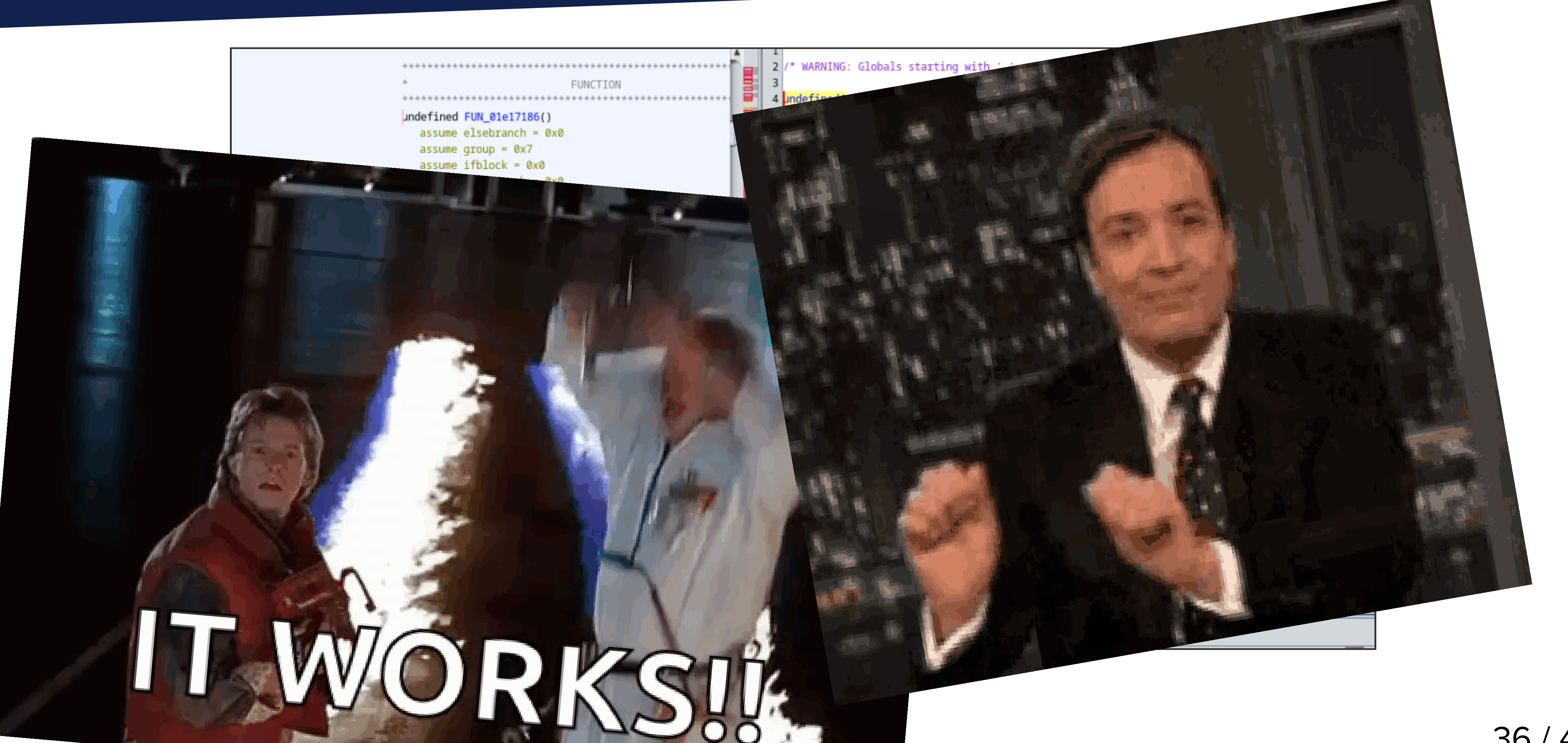
Après des semaines d’efforts…

```
*******************************************************
*                       FUNCTION
*******************************************************
undefined FUN_01e17186()
     assume elsebranch = 0x0
     assume group = 0x7
     assume ifblock = 0x0
     assume thenbranch = 0x0
undefined      r0:1        <RETURN>
               FUN_01e17186                              XREF

01e17186 78 04          push    {rets,r8,r7,r6,r5,r4}
01e17188 0c 9c          add     r4,r0,#0x1c
01e1718a 46 e0 9c 01    movz    r6,#0x19c
01e1718e c7 ff 20       mov     r7,#0x11320
         13 01 00
01e17194 c8 ff a8       mov     r8,#0x10f1a8
         f1 10 00

               LAB_01e1719a                              XREF
01e1719a d8 ec 7a 06    lw      r0,[r7+r6=>DAT_000452f0<<2]
01e1719e 41 16          mov     r1,r4
01e171a0 80 ea 6b cc    call    FUN_01e30a7a
01e171a4 05 16          mov     r5,r0
01e171a6 85 58          jnz     r5,LAB_01e171d8
01e171a8 64 e0 00 0b    mov     r0,icfg
01e171ac 60 e1 40 0f    and     r0,r0,#0x300
01e171b0 45 20          mov     r5,#0x0
01e171b2 01 ff 00       jne     r0,#0x300,LAB_01e171e6
         03 17 00
01e171b8 d0 ec 80 00    ldw     r0,r8=>DAT_0010f1a8,#0x0
01e171bc 00 f8 13 0c    je      r0,0x6,LAB_01e171e6
01e171c0 64 e0 00 0b    mov     r0,icfg
```

```
1
2  /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
3
4  undefined4 * FUN_01e17186(int param_1)
5
6  {
7    undefined4 *puVar1;
8    int iVar2;
9    undefined4 *puVar3;
10   uint in_icfg;
11
12   while( true ) {
13     puVar1 = (undefined4 *)FUN_01e30a7a(_DAT_000452f0,param_1 + 0x1c);
14     if (puVar1 != (undefined4 *)0x0) {
15       iVar2 = 7;
16       puVar3 = puVar1;
17       do {
18         *puVar3 = 0;
19         puVar3 = puVar3 + 1;
20         iVar2 = iVar2 + -1;
21       } while (iVar2 != 0);
22       puVar1[4] = puVar1 + 4;
23       puVar1[5] = puVar1 + 4;
24       return puVar1;
25     }
26     if ((in_icfg & 0x300) != 0x300) {
27       return (undefined4 *)0x0;
28     }
29     if (_DAT_0010f1a8 == 6) {
30       return (undefined4 *)0x0;
31     }
32     if ((in_icfg & 0xff) != 0) break;
33     iVar2 = FUN_01e0137e(0x12824,0);
34     if (iVar2 != 0) {
```

Decompile: FUN_01e17186 × | Bytes: app.bin × | Defined Strings ×

Après des semaines d'efforts...



Revenons à ~~nos moutons~~ notre montre

Quarkslab

Capteur ou pas capteur ?

C

```c
uVar2 = fonction_zarb(); // << mais qu'est-ce que cette fonction ?
heart_rate_current = (char)uVar2 - (char)(((int)uVar2 / 0x10 & 0xffU) << 4);
heart_rate_current += 0x41;

if (heart_rate_max <= heart_rate_current) {
  heart_rate_max = heart_rate_current;
}
/* ... */
bVar16 = heart_rate_min;
if (heart_rate_current <= heart_rate_min) {
  bVar16 = heart_rate_current;
}
```

La fonction mystère

```c
uint32_t fonction_zarb(void) {
  int A;
  longlong lVar1;
  uint uVar2;
  A = (int)*(undefined8 *)(glob_B + 0xa04);
  lVar1 = (longlong)A * 0x4c957f2d;
  uVar2 = A * 0x5851f42d + (int)((ulonglong)lVar1 >> 0x20) +
          (int)((ulonglong)*(undefined8 *)(glob_B + 0xa04) >> 0x20) *
          0x4c957f2d;
  *(ulonglong *)(glob_B + 0xa4) = CONCAT44(uVar2,(int)lVar1 + 1);
  return uVar2 & 0x7fffffff;
}
```

La fonction mystère

```c
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);
}
```

La fonction mystère

C

```c
int rand (void)
{
  struct _reent *reent

  /* This multiplier wa                                        of
     Computer Programmi                                        Third
     Edition, Addison-We                                        */
  _REENT_CHECK_RAND48(re
  _REENT_RAND_NEXT(reent
     _REENT_RAND_NEXT(re                                      05LL + 1;
  return (int)((_REENT_RA
}
```

Ça explique donc ces valeurs !

- `BPM = 65 + randint(0, 16)`
- $65 \leq \text{BPM} \leq 80$
- L'indice était sous nos yeux 🙄

Conclusion

Quarkslab

Tout ça… pour ça.

- **5 semaines** pour retrouver ∼**90% des instructions manquantes** 😅
- **218 instructions supplémentaires** ajoutées au processeur Ghidra
- Analyse par rétro-ingénierie du micrologiciel

Tout ça pour avoir la preuve irréfutable que les données sont aléatoires,
et que cette montre, c'est juste du 💩.

L’important, c’est le chemin.

- J’ai beaucoup appris sur Ghidra et le langage SLEIGH 😃
- Perdu *énormément* de temps aussi… 🤬

- **Des LLMs pour reverser des instructions ?**

  👉 pas automatique mais peut aider

Be like Alexander

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

Le mot de la fin

- Est-ce que ça valait le coup de s'acharner ? *Oh que oui* 🤣
- L'**article** dans les actes contient **beaucoup plus de détails** (RTFM)
- Un grand merci à *Thomas Cougnard* pour son aide sur l'extraction du micrologiciel
- **Merci à vous** de m'avoir écouté (ça a été ? pas trop speed ?) 😊

https://github.com/quarkslab/ghidra-jieli

Références

- 🔗 Fun with watches @ leHACK 2025 (Damien Cauquil, Thomas Cougnard)
- 🔗 JieLi Stuff, GitHub (Andrey Grigoryev)
- 🔗 Ghidra Language Specification (SLEIGH)

Thank you

Contact information

Email:	dcauquil@quarkslab.com
Phone:	+33 1 58 30 81 51
Website:	https://www.quarkslab.com