Réinventez la roue !

(pour une meilleure adhérence sur les autoroutes de l'information)

Cyrille Bagard

@laughing_bit / www.chrysalide.re / @chrysalide_ref

BeeRumP #3 - 31 mai 2018

Bref aperçu

- Spécifications : PDF de 2734 pages [0]
- Taille des instructions :
  - 32 bits (ARM et Thumb32)
  - 16 bits (Thumb16)
- Encodage sur des bits non consécutifs

Nombre d'instructions

- La base (partie A8) :
  - 279 instructions classiques
  - 147 instructions SIMD
- Mode ThumbEE (partie A9) : 11 instructions
- Section système (partie B9) : 22 instructions

Le sportif intelligent évite l'effort inutile.

- M. Mégot

Pourquoi ne pas s'appuyer sur l'OpenSource ?

- Amoco : très joli code, mais 100% Python
- Capstone : ARMv7 mais pas de Dalvik...
- Radare2 : trop... généraliste (RAw DAta REcovery)

Pourquoi ne pas s'appuyer sur l'OpenSource ?

- Amoco : très joli code, mais 100% Python
- Capstone : ARMv7 mais pas de Dalvik...
- Radare2 : trop... généraliste (RAw DAta REcovery)

Principe de base

- On ne sous-traite pas son coeur de métier !

An overcomplicated way to disassemble bytes (2015)

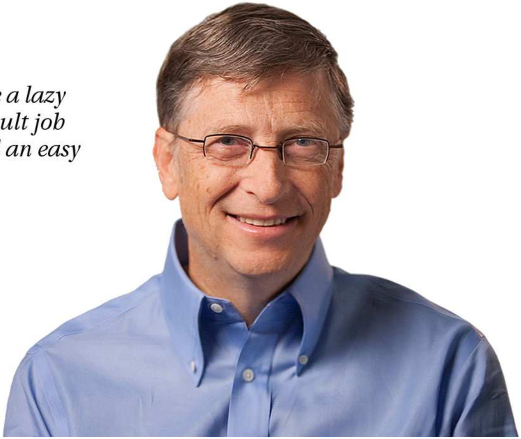
- Avast à Ekoparty : [1] et [2]
- Traitement automatisé via pdf2text
- Intervention manuelle *a posteriori* dans 20% des cas
- Sélection par masque (tri par fréquence)

Generating a Thumb2 disassembler from the specification (2016)

- Blog de Binary Ninja : [3]
- Conversion manuelle du PDF
- Génération de la table de sélection et des instructions

I will always choose a lazy person to do a difficult job because he will find an easy way to do it.

- Bill Gates



Traitement idéal des spécifications

- Entièrement automatisé
- Traduction des instructions en code C à l'aide de Python
- Langage de définition intermédiaire (réusinage...)
- Glaner un maximum d'informations :
  - masques de bits fixes
  - descriptions
  - encodages (Thumb / ARM)
- Attribuer un identifiant unique
  - méthode du Small Primes Product pour le *binary diffing* [4]

A8.8.1 ADC (immediate)

Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
ADC{S}<c> <Rd>, <Rn>, #<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	1	0	S	Rn				0	imm3			Rd				imm8							

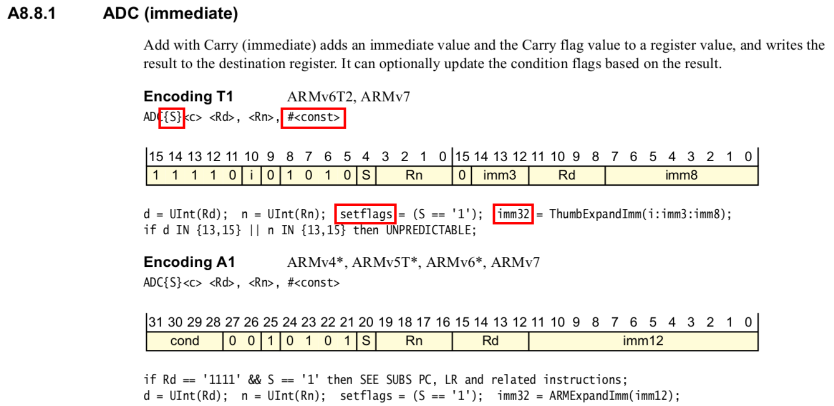
```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADC{S}<c> <Rd>, <Rn>, #<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	1	0	1	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```



A8.8.1 ADC (immediate)

Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
ADC{S}<c> <Rd>, <Rn>, #<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	1	0	S	Rn				0	imm3			Rd				imm8							

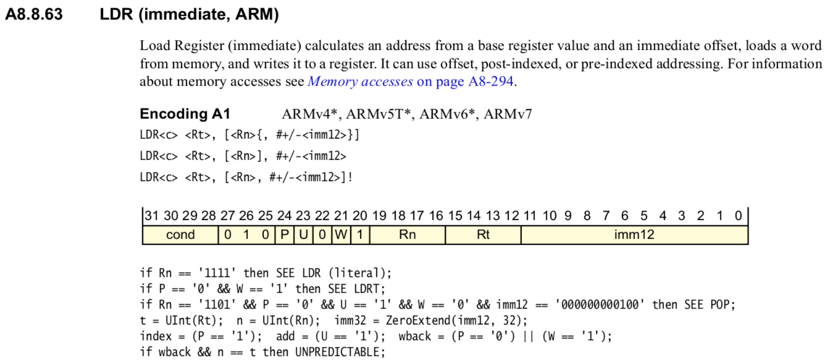
```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADC{S}<c> <Rd>, <Rn>, #<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	1	0	1	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```



A8.8.63 LDR (immediate, ARM)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-294.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>, [<Rn>{, #+/-<imm12>}]
LDR<c> <Rt>, [<Rn>], #+/-<imm12>
LDR<c> <Rt>, [<Rn>, #+/-<imm12>]!
```

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0	1	0	P	U	0	W	1	Rn	Rt	imm12

```
if Rn == '1111' then SEE LDR (literal);
if P == '0' && W == '1' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '0' && imm12 == '000000000100' then SEE POP;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```

A8.8.63 LDR (immediate, ARM)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-294.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>, [<Rn>{, #+/-<imm12>}]
LDR<c> <Rt>, [<Rn>], #+/-<imm12>
LDR<c> <Rt>, [<Rn>, #+/-<imm12>]!
```

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0	1	0	P	U	0	W	1	Rn	Rt	imm12

```
if Rn == '1111' then SEE LDR (literal);
if P == '0' && W == '1' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '0' && imm12 == '000000000100' then SEE POP;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```

Assembler syntax

`LDR{<c>}{<q>} <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE, wback==FALSE`
`LDR{<c>}{<q>} <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`LDR{<c>}{<q>} <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE, wback==TRUE`

A8.8.63 LDR (immediate, ARM)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-294.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>, [<Rn>{, #+/-<imm12>}]
LDR<c> <Rt>, [<Rn>], #+/-<imm12>
LDR<c> <Rt>, [<Rn>, #+/-<imm12>]!
```

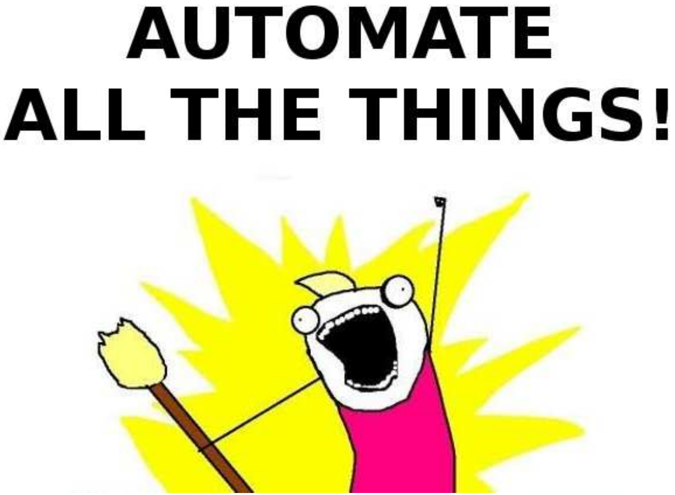
31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0	1	0	P	U	0	W	1	Rn	Rt	imm12

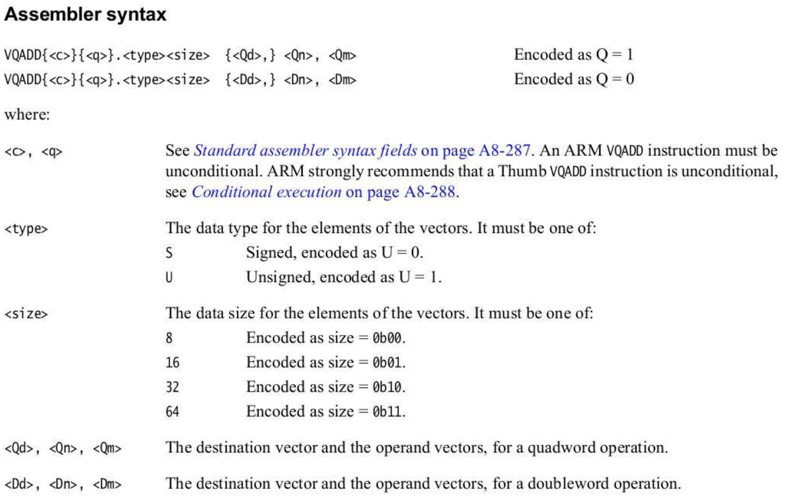
```
if Rn == '1111' then SEE LDR (literal);
if P == '0' && W == '1' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '0' && imm12 == '000000000100' then SEE POP;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```

Assembler syntax

`LDR{<c>}{<q>} <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE, wback==FALSE`
`LDR{<c>}{<q>} <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`LDR{<c>}{<q>} <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE, wback==TRUE`

```
@encoding (A1) {

    @word cond(4) 0 1 0 P(1) U(1) 0 W(1) 1 Rn(4) Rt(4) imm12(12)

    @syntax {

        @subid 174

        @assert {
            P == 1
            P == 1 && W == 0
        }

        @conv {
            reg_T = Register(Rt)
            reg_N = Register(Rn)
            imm32 = ZeroExtend(imm12, 32)
            maccess = MemAccessOffset(reg_N, imm32)
        }

        @asm ldr reg_T maccess

        @rules {
            check g_arm_instruction_set_cond(cond)
        }

    }

    [...]

}
```


AUTOMATE
ALL THE THINGS!

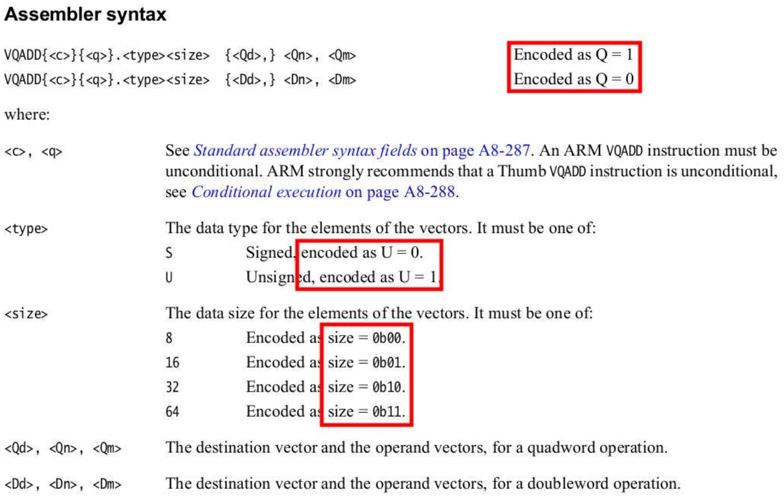


Assembler syntax

`VQADD{<c>}{<q>}.<type><size> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VQADD{<c>}{<q>}.<type><size> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c>, <q>`	See *Standard assembler syntax fields* on page A8-287. An ARM `VQADD` instruction must be unconditional. ARM strongly recommends that a Thumb `VQADD` instruction is unconditional, see *Conditional execution* on page A8-288.	
`<type>`	The data type for the elements of the vectors. It must be one of:	
	`S`	Signed, encoded as U = 0.
	`U`	Unsigned, encoded as U = 1.
`<size>`	The data size for the elements of the vectors. It must be one of:	
	`8`	Encoded as size = `0b00`.
	`16`	Encoded as size = `0b01`.
	`32`	Encoded as size = `0b10`.
	`64`	Encoded as size = `0b11`.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.	
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.	



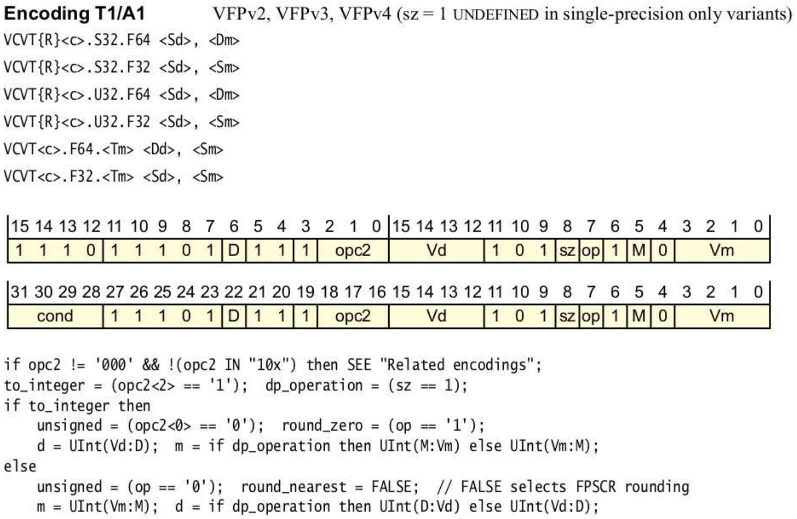
Assembler syntax

```
VQADD{<c>}{<q>}.<type><size>  {<Qd>,} <Qn>, <Qm>
VQADD{<c>}{<q>}.<type><size>  {<Dd>,} <Dn>, <Dm>
```

Encoded as Q = 1
Encoded as Q = 0

where:

`<c>`, `<q>`	See *Standard assembler syntax fields* on page A8-287. An ARM `VQADD` instruction must be unconditional. ARM strongly recommends that a Thumb `VQADD` instruction is unconditional, see *Conditional execution* on page A8-288.	
`<type>`	The data type for the elements of the vectors. It must be one of:	
	`S`	Signed, encoded as U = 0.
	`U`	Unsigned, encoded as U = 1.
`<size>`	The data size for the elements of the vectors. It must be one of:	
	`8`	Encoded as size = `0b00`.
	`16`	Encoded as size = `0b01`.
	`32`	Encoded as size = `0b10`.
	`64`	Encoded as size = `0b11`.
`<Qd>`, `<Qn>`, `<Qm>`	The destination vector and the operand vectors, for a quadword operation.	
`<Dd>`, `<Dn>`, `<Dm>`	The destination vector and the operand vectors, for a doubleword operation.	



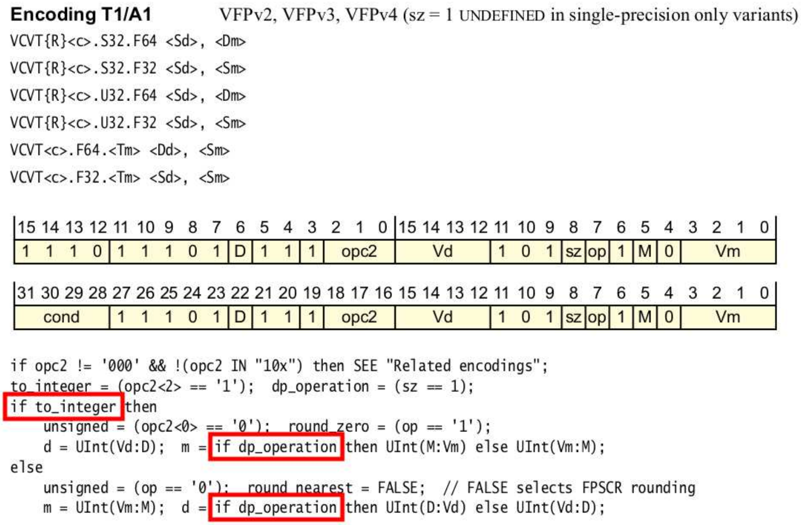
Encoding T1/A1 VFPv2, VFPv3, VFPv4 (sz = 1 UNDEFINED in single-precision only variants)

```
VCVT{R}<c>.S32.F64 <Sd>, <Dm>
VCVT{R}<c>.S32.F32 <Sd>, <Sm>
VCVT{R}<c>.U32.F64 <Sd>, <Dm>
VCVT{R}<c>.U32.F32 <Sd>, <Sm>
VCVT<c>.F64.<Tm> <Dd>, <Sm>
VCVT<c>.F32.<Tm> <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	1	opc2			Vd				1	0	1	sz	op	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	1	opc2			Vd				1	0	1	sz	op	1	M	0	Vm			

```
if opc2 != '000' && !(opc2 IN "10x") then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == 1);
if to_integer then
    unsigned = (opc2<0> == '0');  round_zero = (op == '1');
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');  round_nearest = FALSE;  // FALSE selects FPSCR rounding
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```



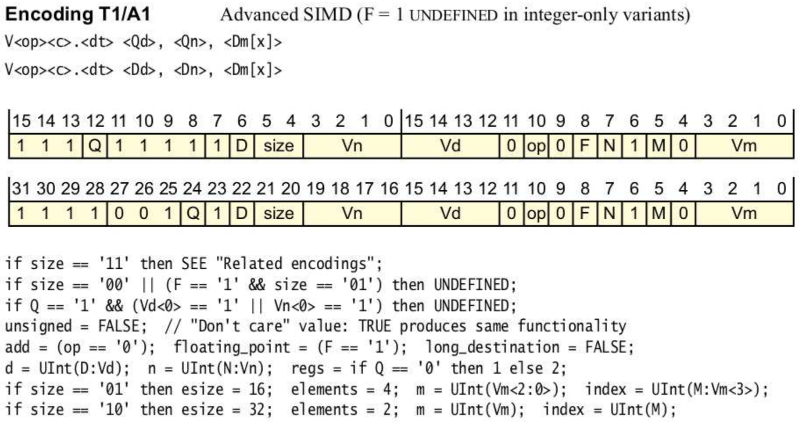
Encoding T1/A1 VFPv2, VFPv3, VFPv4 (sz = 1 UNDEFINED in single-precision only variants)

```
VCVT{R}<c>.S32.F64 <Sd>, <Dm>
VCVT{R}<c>.S32.F32 <Sd>, <Sm>
VCVT{R}<c>.U32.F64 <Sd>, <Dm>
VCVT{R}<c>.U32.F32 <Sd>, <Sm>
VCVT<c>.F64.<Tm> <Dd>, <Sm>
VCVT<c>.F32.<Tm> <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	1	opc2			Vd				1	0	1	sz	op	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	1	opc2			Vd				1	0	1	sz	op	1	M	0	Vm			

```
if opc2 != '000' && !(opc2 IN "10x") then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == 1);
if to_integer then
    unsigned = (opc2<0> == '0');  round_zero = (op == '1');
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');  round_nearest = FALSE;  // FALSE selects FPSCR rounding
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```



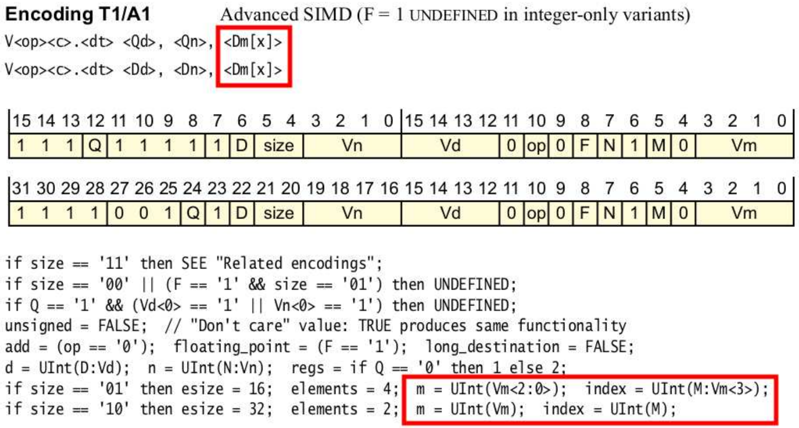
Encoding T1/A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
V<op><c>.<dt> <Qd>, <Qn>, <Dm[x]>
V<op><c>.<dt> <Dd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				0	op	0	F	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				0	op	0	F	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```



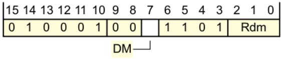
Encoding T1/A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

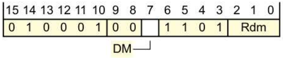
```
V<op><c>.<dt> <Qd>, <Qn>, <Dm[x]>
V<op><c>.<dt> <Dd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				0	op	0	F	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				0	op	0	F	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```


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



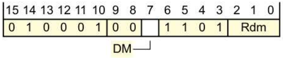
```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a == UInt(Ra);
```

```
STM{<amode>}<c> <Rn>, <registers>^
```

```
VMVN{<c>}{<q>}.dt>  <Qd>, #<imm>
VMVN{<c>}{<q>}.dt>  <Dd>, #<imm>
```

```
VBIC{<c>}{<q>}.<dt>  {<Qd>,} <Qd>, #<imm>
VBIC{<c>}{<q>}.<dt>  {<Dd>,} <Dd>, #<imm>>
```

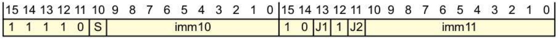
Encoded as $Q = 1$
Encoded as $Q = 0$



```
d = UInt(Rd);   n = UInt(Rn);   m = UInt(Rm);   a == UInt(Ra);
```

```
imm16 = imm4:1mm12;
```

```
STM{<amode>}<c> <Rn>, <registers>^
```

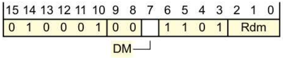


```
imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
```

```
VMVN{<c>}{<q>}.dt>  <Qd>, #<imm>
VMVN{<c>}{<q>}.dt>  <Dd>, #<imm>
```

```
VBIC{<c>}{<q>}.<dt>  {<Qd>,} <Qd>, #<imm>
VBIC{<c>}{<q>}.<dt>  {<Dd>,} <Dd>, #<imm>>
```

Encoded as Q = 1
Encoded as Q = 0

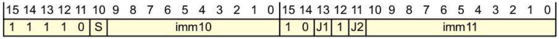


```
VFM<y><c><q>.F32 <Qd>, <Qn>, <Qm>
VFM<y><c><q>.F32 <Dd>, <Dn>, <Dm>
VFM<y><c><q>.F64 <Dd>, <Dn>, <Dm>
VFM<y><c><q>.F32 <Sd>, <Sn>, <Sm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a == UInt(Ra);
```

```
imm16 = imm4:1mm12;
```

```
STM{<amode>}<c> <Rn>, <registers>^
```



```
imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
```

```
VMVN{<c>}{<q>}.dt>  <Qd>, #<imm>
VMVN{<c>}{<q>}.dt>  <Dd>, #<imm>
```

```
VBIC{<c>}{<q>}.<dt>  {<Qd>,} <Qd>, #<imm>      Encoded as Q = 1
VBIC{<c>}{<q>}.<dt>  {<Dd>,} <Dd>, #<imm>>     Encoded as Q = 0
```

Avantages du format de définition

- simple et lisible
- souple : @assert + @hooks + @rules
- générique : ARMv7 + Dalvik (+ Java + ARMv8 + etc.)
- productif : 117k lignes de C générées

Avantages du format de définition

- simple et lisible
- souple : @assert + @hooks + @rules
- générique : ARMv7 + Dalvik (+ Java + ARMv8 + etc.)
- productif : 117k lignes de C générées

Apports de la nouvelle roue

- humilité : mise en perspective des travaux précédents
- apprentissage : survol de l'ensemble des détails d'ARMv7
- avancées : réutilisation ultérieure des développements

Très courte procédure pour découvrir :

`https://www.chrysalide.re/?title=Installation`

Merci pour votre attention !

Des remarques, questions ou commentaires ?

[0] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html

[1] https://www.ekoparty.org/archivo/2015/eko11-ARM_disassembling_with_a_twist.pdf
[2] https://drive.google.com/file/d/0B0l-Qo3D3sAoMEhkcFBFVzRiNEk/view

[3] https://binary.ninja/2016/10/27/generating-a-thumb2-disassembler-from-the-specification.html

[4]
http://actes.sstic.org/.../SSTIC05-article-Flake-Graph_based_comparison_of_Executable_Objects.pdf