Compilation et assembleur

5d54626cc43ba7ce625903ffa39c049c

Groupe de sécurité de l'information, École normale supérieure

2 octobre 2019

Définition

Definition (Compilateur)

Programme qui transforme un programme écrit dans un langage **source** vers un programme en langage **objet**.

Example

`gcc` : du C vers le langage machine `ghc` : du Haskell vers le langage machine

En particulier, il est possible de compiler vers un langage intermédiaire très proche de la machine, mais néanmoins compréhensible par des humains : l'**assembleur** (e.g. *x86*, *mips*, *risc-v*, *armv7*).
Le programme qui traduit un programme assembleur en langage machine s'appelle un **assembleur**.

Cas particuliers

Il est possible de :

- Compiler vers un autre langage de haut-niveau (Haskell vers C)
- Compiler vers un pseudo-langage machine indépendant de la machine (bytecode,) $\rightarrow$ ces programmes nécessitent pour leur exécution un autre programme appelé interpréteur (`JVM`, `ocamlrun`, `LLVM`, ...)
- Compiler un programme pendant son exécution $\rightarrow$ *Just-in-time* ou *compilation à la volée* (souvent effectué par les interpréteurs sus-cités)
- Utiliser le compilateur d'un langage hôte pour compiler un code dans un autre langage $\rightarrow$ *embedded domain-specific language*

Structure d'un compilateur

Deux phases successives:

- Analyse : traduction du langage source en représentation abstraite mathématique
  - Analyse lexicale : traduction du source vers des **lexèmes** (tokens) → `lex`, `bison`
  - Analyse syntaxique : traduction des lexèmes vers un **arbre de syntaxe abstraite** (AST) → `yacc`, `menhir`
  - Typage, ...
- Synthèse : traduction de la représentation mathématique en langage objet
  - Optimisations
  - Traduction dans des langages intermédiaires plus proches de la machine (*RTL*, *Cminor*, *LLVM bytecode*, ...)
  - Production de code assembleur
  - Assemblage
  - Édition des liens (linking)

Analyse lexicale

Definition (Lexer)

Programme qui transforme une suite de caractères (écrit dans un langage source) en une suite de lexèmes (ou tokens).

Exemple :

```
int b =
    /* Here is my comment */
    (int) c + 2 * d;
```

Sera transformé par le lexer en:
int b = (int) c + 2 * d ;

Analyse lexicale

On reconnait les lexèmes grâce à des expressions régulières (reconnaissables par automate fini, théorème de Kleene, 1956).

$$\begin{aligned}
\langle digit \rangle &::= 0 - 9 \\
\langle alpha \rangle &::= a - z | A - Z \\
\langle ident \rangle &::= \langle alpha \rangle (\langle alpha \rangle | \langle digit \rangle | _)^* \\
\langle integer \rangle &::= \langle digit \rangle^+
\end{aligned}$$

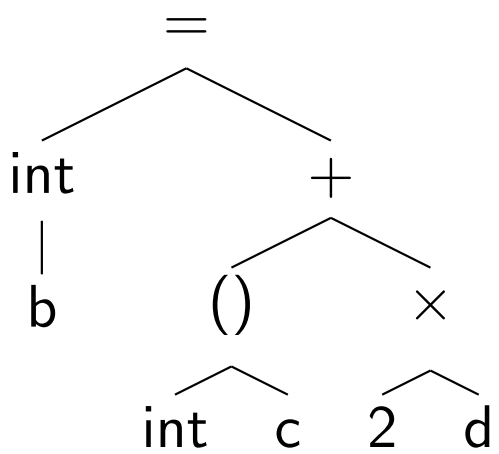
Analyse syntaxique

Definition (Parser)

Programme qui transforme une suite de lexèmes en un arbre de syntaxe abstraite.

Exemple :

int b = (int) c + 2 * d ;



Analyse syntaxique (instructions)

On construit l'arbre de syntaxe abstraite grâce à une grammaire (*Backus–Naur form*) :

path	::=	*ident*	variable
	|	*path* . *ident*	record field
	|	*path* . `all`	pointer dereference
expr	::=	*path*	l-value
	|	*integer* | `True` | `False` | ...	scalar value
	|	*expr* (+ | - | < | = | ...) *expr*	binary operator
	|	*path*`'Access`	address of an l-value
	|	`null`	null pointer
stmt	::=	*path* `:=` *expr*	assignment
	|	*path* `:= new` *type*	allocation
	|	`if` *expr* `then` *stmt** `else` *stmt** `end`	conditional
	|	`while` *expr* `loop` *stmt** `end`	"while" loop
	|	*ident* `(` *expr** `)`	procedure call

Analyse syntaxique (déclarations)

$$
\begin{array}{rcl}
\textit{procedure} & ::= & \texttt{procedure}\ \textit{ident}\ \texttt{(}\ \textit{param}^{\star}\ \texttt{)}\ \texttt{is} \\
 & & \quad \textit{local}^{\star}\ \texttt{begin}\ \textit{stmt}^{\star}\ \texttt{end} \\
\textit{param} & ::= & \textit{ident}\ \texttt{:}\ (\ \texttt{in} \mid \texttt{in out} \mid \texttt{out}\ )\ \textit{type} \\
\textit{local} & ::= & \textit{ident}\ \texttt{:}\ \textit{type}
\end{array}
$$

type	::=	`Integer` | `Real` | `Boolean`	scalar type
	|	`access` *type*	access type (pointer)
	|	*ident*	record type

Analyse syntaxique (exemple)

https://cs.wmich.edu/ gupta/teaching/cs4850/sumII06/The syntax of C in Backus-Naur form.htm

The syntax of C in Backus-Naur Form

```
<translation-unit> ::= {<external-declaration>}*

<external-declaration> ::= <function-definition>
                         | <declaration>

<function-definition> ::= {<declaration-specifier>}* <declarator> {<declaration>}* <compound-statement>

<declaration-specifier> ::= <storage-class-specifier>
                          | <type-specifier>
                          | <type-qualifier>

<storage-class-specifier> ::= auto
                            | register
                            | static
                            | extern
                            | typedef

<type-specifier> ::= void
                   | char
                   | short
                   | int
                   | long
                   | float
                   | double
                   | signed
                   | unsigned
                   | <struct-or-union-specifier>
                   | <enum-specifier>
                   | <typedef-name>

<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>}+ }
                              | <struct-or-union> { {<struct-declaration>}+ }
                              | <struct-or-union> <identifier>

<struct-or-union> ::= struct
                    | union

<struct-declaration> ::= {<specifier-qualifier>}* <struct-declarator-list>
```

Analyse syntaxique

En pratique :

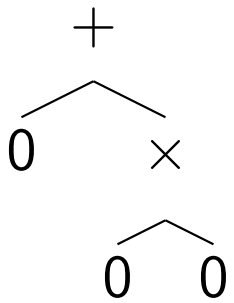
- La grammaire est **non-contextuelle** (ou algébrique) $\rightarrow$ reconnaissable par automate à pile non déterministes, **algorithme CYK**, en temps $\mathcal{O}\left(n^3 \cdot |G|\right)$.
- En pratique, on se restreint aux langages reconnaissables par automates à pile **déterministes**, c'est-à-dire que la grammaire ne contient pas d'ambiguïtés. Ce sont les langages **LR(1)**, reconnaissables en temps $\mathcal{O}\left(n \cdot |G|\right)$.

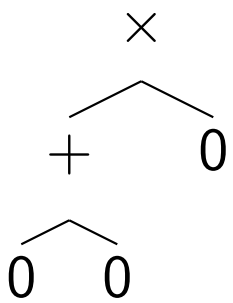
Analyse syntaxique

Exemple d'ambiguïté (reduce/reduce) :

expr	::=	*integer*	scalar value
	|	*expr* + *expr*	binary operator
	|	*expr* ∗ *expr*	binary operator

Parsons l'expression 0+0∗0 :



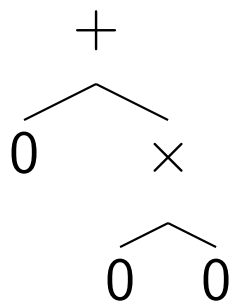


Analyse syntaxique

Solution:

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{expr} \ + \ \mathit{texpr} \\
 & | & \mathit{texpr} \\
\mathit{texpr} & ::= & \mathit{texpr} \ * \ \mathit{fexpr} \\
 & | & \mathit{fexpr} \\
\mathit{fexpr} & ::= & \mathit{integer}
\end{array}
$$

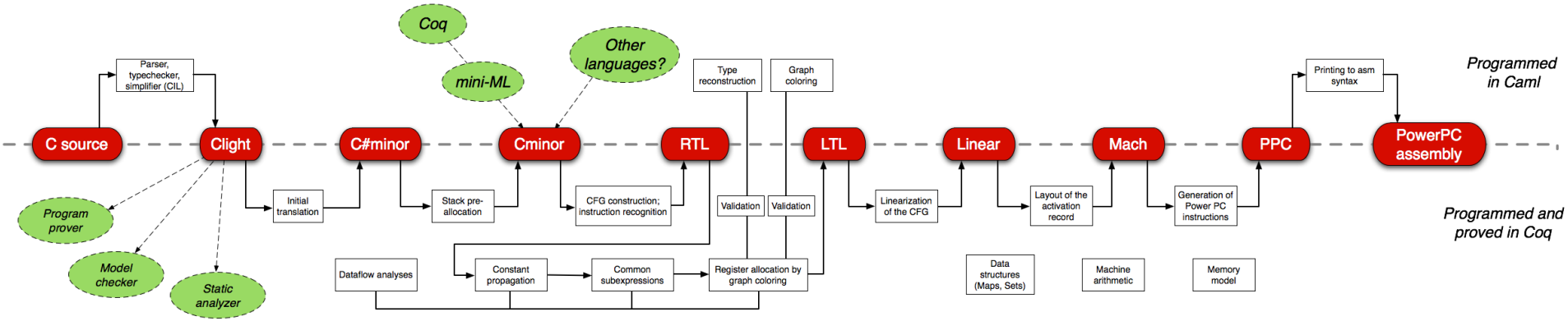
Il n'y a plus d'ambiguïté :



Phase d'analyse

- Par ailleurs, le compilateur génère des erreurs si des violations sont repérées (erreurs lexicales, erreurs syntaxiques, ...)
- Un lexer ne retourne **jamais** en arrière : il construit à chaque fois le plus long lexème valide à partir de la position courante.
- D'autres analyses (*sémantiques*) peuvent être effectuées : **typage statique**, **vérification des noms** (noms de variables uniques, ...), **data analysis**, **control-flow analysis**

Phase de synthèse



Le but est de transformer l'arbre de syntaxe abstraite en code objet (assembleur). Plusieurs étapes :

- Sélection d'instructions / Optimisations (propagation de constantes, elimination de code mort, factorisation de code, ...)
- Register transfer language (RTL) / Single static assignment (SSA) / Continuation passing style (CPS)
- Location transfer language (LTL)
- Assembleur (code linéarisé)

Un peu d'architecture matérielle

Computer Organization and Design: the HW/SW interface, Patterson et Hennessy

Rappel : les nombres sont représentés en binaires dans une machine.

Plusieurs représentations :

- Entier non signé :
  $1 \times 128 + 0 \times 64 + 1 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 166$
- Entier signé :
  $1 \times (-128) + 0 \times 64 + 1 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = -90$

Un peu d'architecture matérielle

Mais pour un même nombre `0x96134f6f` (en 32 bits, 4 octets) :

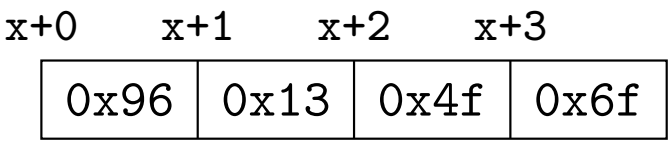


Figure : Big-endian

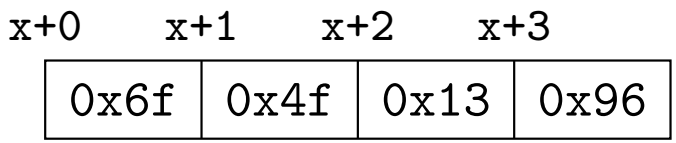
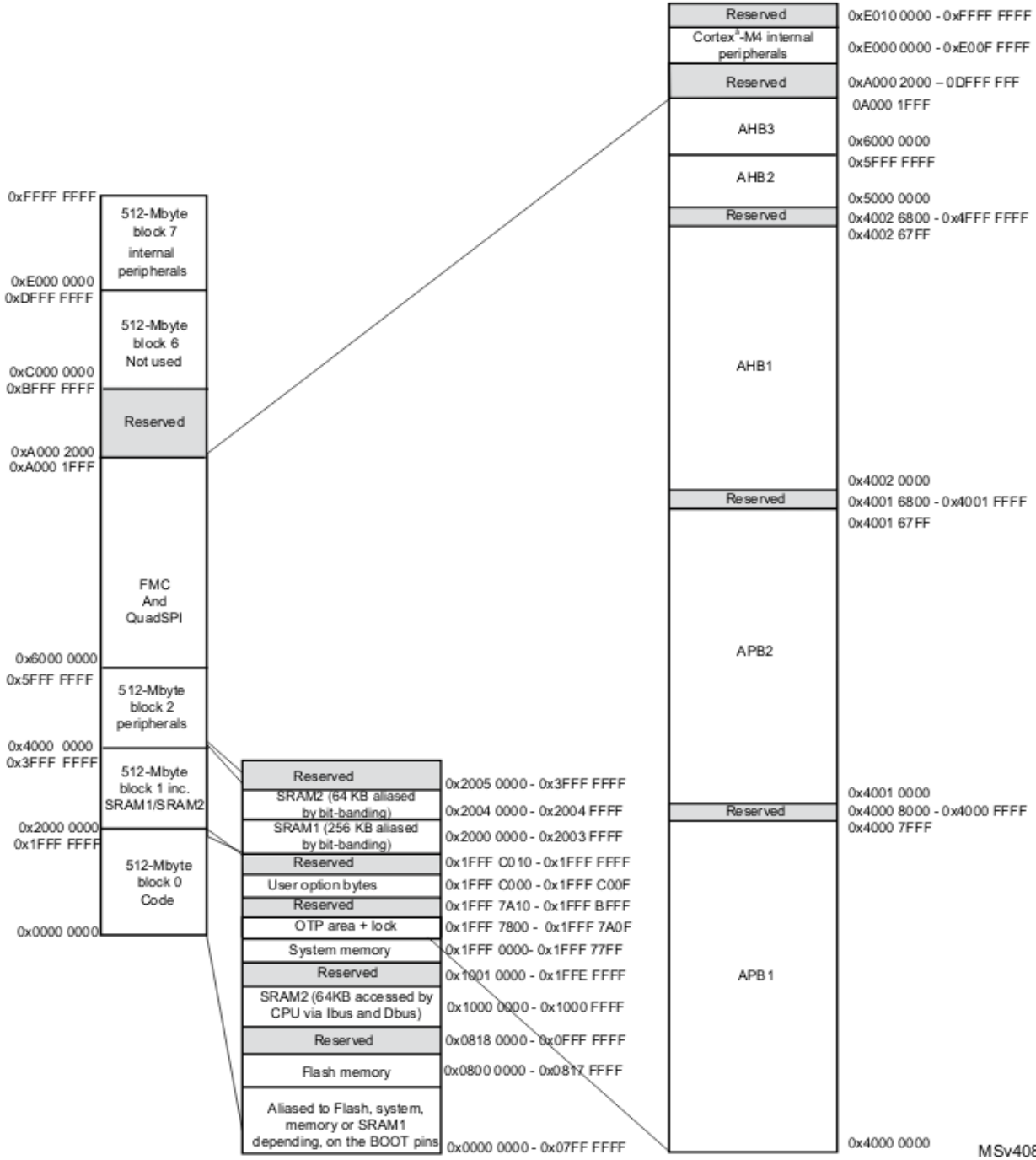


Figure : Little-endian

Les registres/la mémoire

Un ordinateur contient deux types de mémoires :

- Les registres : ils contiennent exactement un mot, et sont localisés dans le processeur. Très rapides d'accès, mais leur nombre est restreint. On les désigne par leur nom de registre.
- La mémoire volatile : elle peut contenir beaucoup de données, mais avec un temps d'accès plus long. On y accède grâce à une adresse (pointeur en C). Les adresses ont une taille de 1 mot (i.e. un registre).
- Autres : ROM, périphériques, ... Leur accès se fait aussi grâce à une adresse.


0xFFFF FFFF
512-Mbyte block 7 internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte block 6 Not used
0xC000 0000
0xBFFF FFFF
Reserved
0xA000 2000
0xA000 1FFF
FMC And QuadSPI
0x6000 0000
0x5FFF FFFF
512-Mbyte block 2 peripherals
0x4000 0000
0x3FFF FFFF
512-Mbyte block 1 inc. SRAM1/SRAM2
0x2000 0000
0x1FFF FFFF
512-Mbyte block 0 Code
0x0000 0000
Reserved
0x2005 0000 - 0x3FFF FFFF
SRAM2 (64 KB aliased by bit-banding)
0x2004 0000 - 0x2004 FFFF
SRAM1 (256 KB aliased by bit-banding)
0x2000 0000 - 0x2003 FFFF
Reserved
0x1FFF C010 - 0x1FFF FFFF
User option bytes
0x1FFF C000 - 0x1FFF C00F
Reserved
0x1FFF 7A10 - 0x1FFF BFFF
OTP area + lock
0x1FFF 7800 - 0x1FFF 7A0F
System memory
0x1FFF 0000- 0x1FFF 77FF
Reserved
0x1001 0000 - 0x1FFE FFFF
SRAM2 (64KB accessed by CPU via Ibus and Dbus)
0x1000 0000 - 0x1000 FFFF
Reserved
0x0818 0000 - 0x0FFF FFFF
Flash memory
0x0800 0000 - 0x0817 FFFF
Aliased to Flash, system, memory or SRAM1 depending, on the BOOT pins
0x0000 0000 - 0x07FF FFFF
Reserved
0xE010 0000 - 0xFFFF FFFF
Cortex®-M4 internal peripherals
0xE000 0000 - 0xE00F FFFF
Reserved
0xA000 2000 – 0DFFF FFF
0A000 1FFF
AHB3
0x6000 0000
0x5FFF FFFF
AHB2
0x5000 0000
Reserved
0x4002 6800 - 0x4FFF FFFF
0x4002 67FF
AHB1
0x4002 0000
Reserved
0x4001 6800 - 0x4001 FFFF
0x4001 67FF
APB2
0x4001 0000
Reserved
0x4000 8000 - 0x4000 FFFF
0x4000 7FFF
APB1
0x4000 0000
MSv40822V2

Revenons aux registres

Une architecture définit le nombre de registres et leur taille :

- x86:32 : 4 general purpose registers (`eax`, `ebx`, `ecx`, `edx`), 6 segment registers (`cs`, `ds`, `es`, `fs`, `gs`, `ss`)
- rv64gc : 31 general purpose registers de 64 bits (`x1`-`x31`) + 2 registres spéciaux (`x0` et `pc`) + ...
- aarch64 : 31 general purpose registres de 64 bits (`x0`-`x30`) + 4 registres spéciaux de 64 bits (`zr`, `sp`, `pc`, `elr`) + ...

Revenons aux registres

Une architecture définit le nombre de registres et leur taille :

- x86:32 : 4 general purpose registers (`eax`, `ebx`, `ecx`, `edx`), 6 segment registers (`cs`, `ds`, `es`, `fs`, `gs`, `ss`)
- rv64gc : 31 general purpose registers de 64 bits (`x1`-`x31`) + 2 registres spéciaux (`x0` et `pc`) + ...
- aarch64 : 31 general purpose registres de 64 bits (`x0`-`x30`) + 4 registres spéciaux de 64 bits (`zr`, `sp`, `pc`, `elr`) + ...

Mais aussi registres un peu spéciaux : les `flags`

- ARM CPSR : NZCV qui renseignent sur l'opération précédente
- ...

Revenons aux registres

En réalité c'est plus compliqué que cela :

- Opérations flottantes
- Extensions au jeu d'instruction (SSE, AVX, ...)
- Caches
- Mémoire virtualisée
- Niveaux de privilège

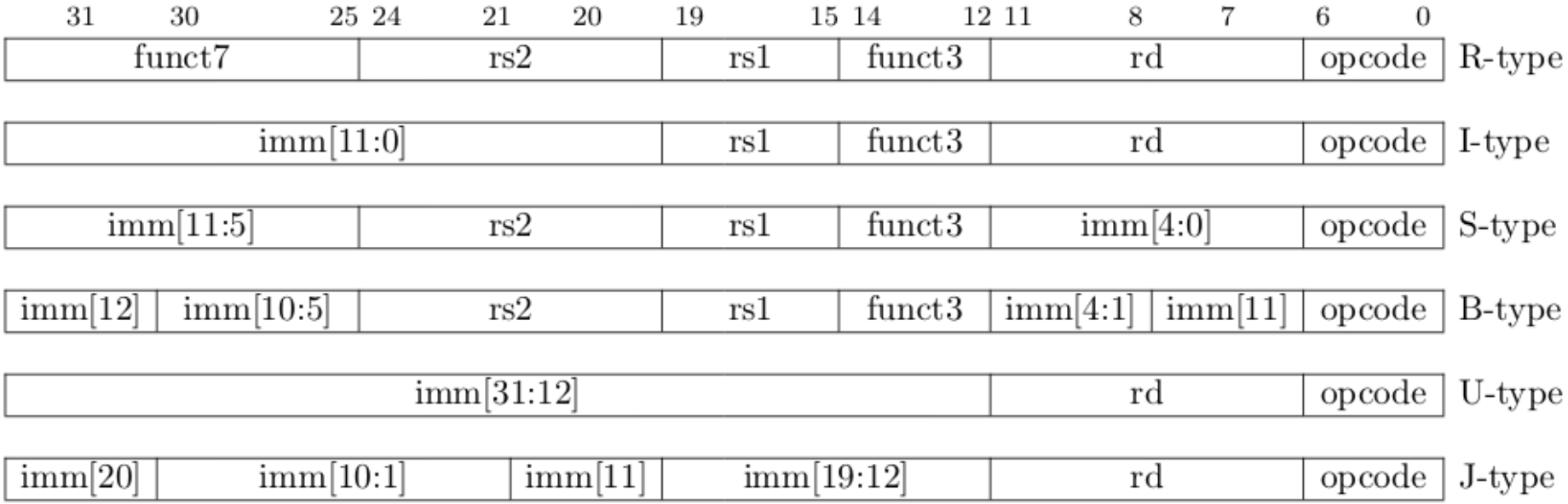
Et les instructions dans tout ça ?

Plusieurs types d'instructions en assembleur :

- Opérations arithmétiques et logiques : addition, soustraction, ...
- Opérations mémoire : load, store.
- Opérations de flot de contrôle : unconditional and conditional branches, function calls.

Les instructions sont stockées en mémoire, et leur longueur/format est spécifié dans l'architecture.

Le pointeur vers le code se nomme le **program counter** (ou **pc**).



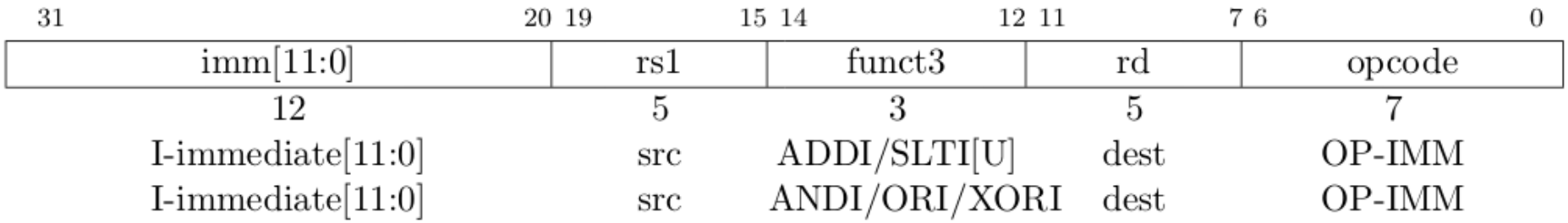
Opérations

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI

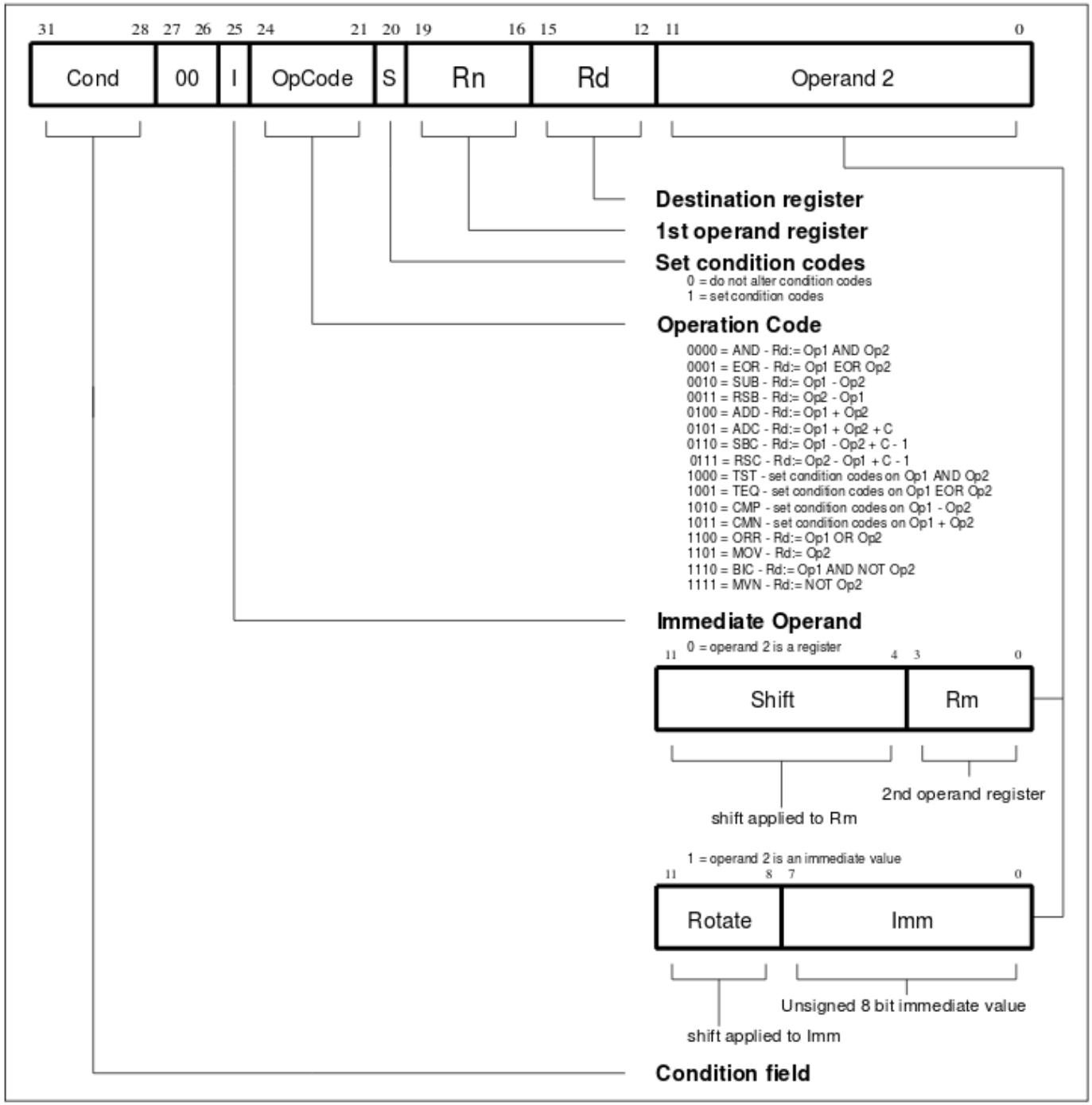
Opérations arithmétiques

Regardons par exemple `addi` en risc-v (I-type)

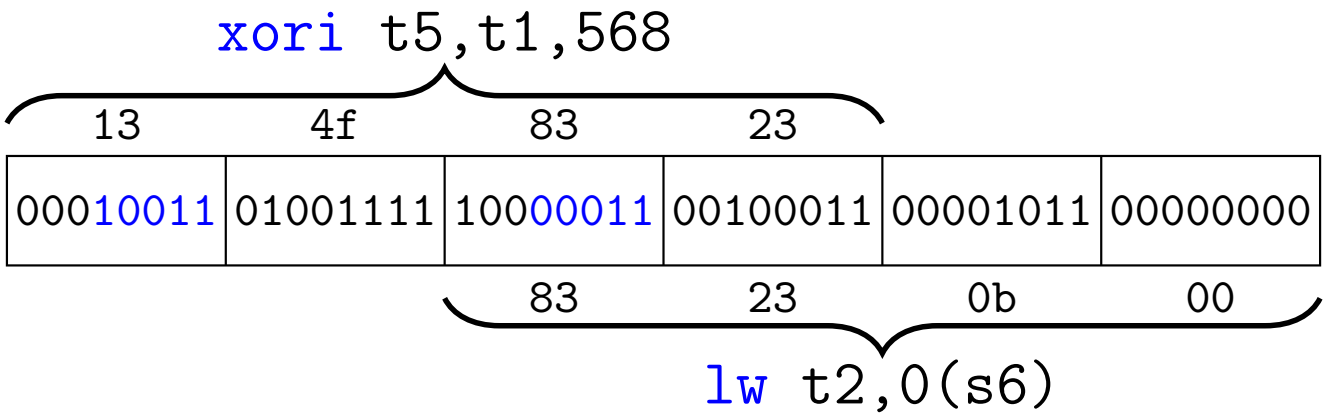


31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the MV *rd, rs1* assembler pseudo-instruction.


31 28 27 26 25 24 21 20 19 16 15 12 11 0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
Immediate Operand
0 = operand 2 is a register
11 4 3 0
Shift
Rm
2nd operand register
shift applied to Rm
1 = operand 2 is an immediate value
11 8 7 0
Rotate
Imm
Unsigned 8 bit immediate value
shift applied to Imm
Condition field

Overlapping code



Première abstraction : les fonctions

- But du jeu : réutiliser des morceaux de code.
- Besoin d'une interface qui permette d'en contrôler les effets de bord.
- Besoin de généricité : le même code doit fonctionne avec plusieurs données.
- Compilateurs différents, langages différents, ...
- Représentation mémoire des structures de données.

$\Rightarrow$ **Application binary interface** (ABI)

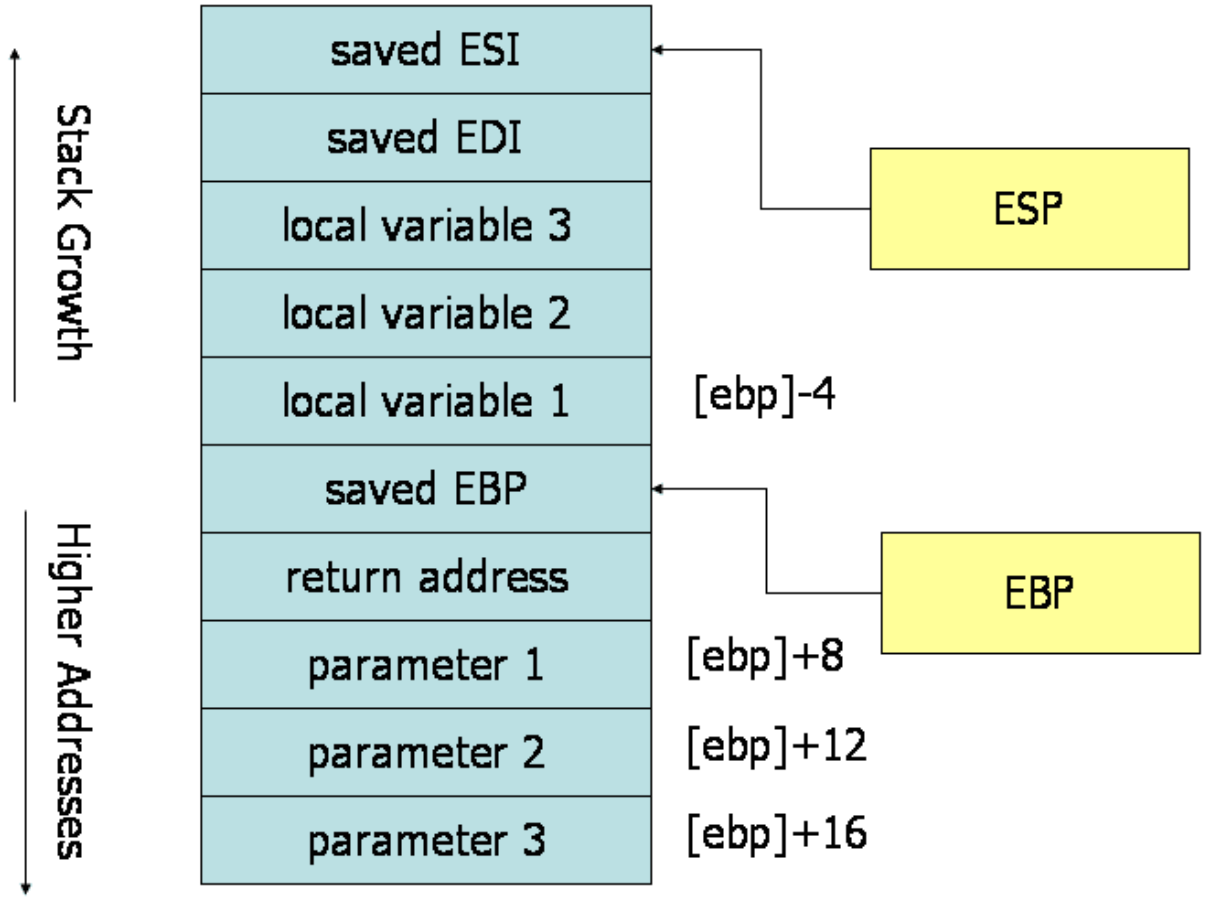
La fonction n'a pas le droit de faire n'importe quoi avec la mémoire ou les registres.

Premiers pas dans le reverse : l'ABI

Register	ABI Mnemonic	Meaning
x0	zero	Zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporary registers
x8-x9	s0-s1	Callee-saved registers
x10-x17	a0-a7	Argument registers
x18-x27	s2-s11	Callee-saved registers
x28-x31	t3-t6	Temporary registers
f0-f7	ft0-ft7	Temporary registers
f8-f9	fs0-fs1	Callee-saved registers
f10-f17	fa0-fa7	Argument registers
f18-f27	fs2-fs11	Callee-saved registers
f28-f31	ft8-ft11	Temporary registers

Figure : Naming convention for registers, per RISC-V ELF psABI.

ABI : Pile d'appel



De façon plus générale

L'ABI définit une convention d'appel :

- Où se trouvent les paramètres ? (registres, pile, ...)
- Quels sont les registres dont la valeur reste inchangée ?
- Alignements des paramètres et de la pile ?

Exemples :

- `https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md`
- `https://www.agner.org/optimize/calling_conventions.pdf`
- `web.archive.org/web/20080410091312/http://msdn2.microsoft.com/en-us/library/zxk0tw93%28vs.71%29.aspx`
- `https://www.ohse.de/uwe/articles/gcc-attributes.html#func-fastcall`

Et dans gcc

```
int __attribute__((stdcall)) function(int a, int b);

int __attribute__((fastcall)) function(int a, int b);

int __attribute__((cdecl)) function(int a, int b);

int __attribute__((sysv_abi)) function(int a, int b);

int __attribute__((no_caller_saved_registers))
                    function(int a, int b);

int __attribute__((thiscall)) function(int a, int b);
```

En pratique : exemple d'une fonction

```
addi      sp,sp,-16
sd        ra,8(sp)

jal       ra,dummy
lui       a0,0x9932
lui       a3,0x23371
lui       a2,0xa0212
mv        a1,zero
jal       ra,dummy4

ld        ra,8(sp)
mv        a0,zero
addi      sp,sp,16
ret
```

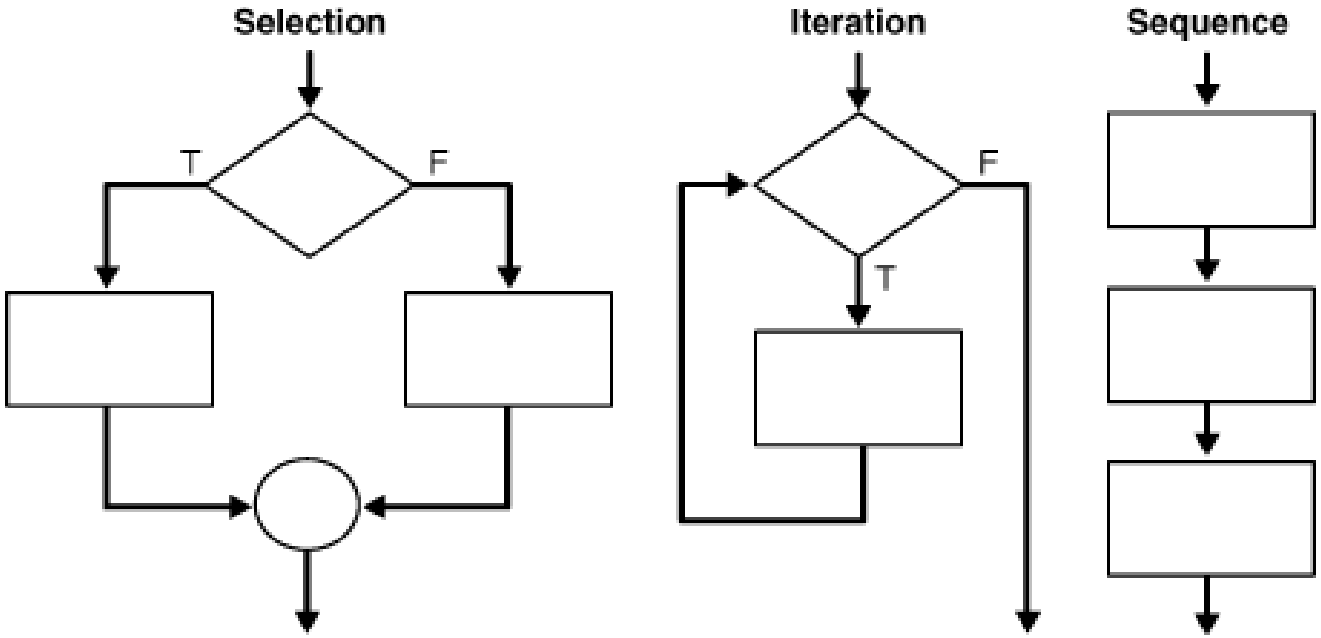
Pour en revenir à la compilation

La traduction de l'arbre de syntaxe abstraite peut se faire quasi-automatiquement vers un langage qui n'est pas tout à fait de l'assembleur :

- $expr1 + expr2 \Rightarrow$ `x` $= expr1;$ `y` $= expr2;$ `ADD z, y, x;`
- $*expr1 \Rightarrow$ `x` $= expr1;$ `LW y,0(x)`
- $fun(expr1) \Rightarrow$ `x` $= expr1;$ `PUSH x; CALL fun`

Programmation structurée (deuxième abstraction)

Il manque les instructions de flots de controle:



Ce qu'il reste à faire

- Prendre en compte l'analyse de durée de vie (defuse chain).
- Allocation des registres (coloriage d'un graphe d'interférence).
- Optimisation des instructions en trop.

Fin

Des questions ?

Programme de cet après midi : rétro-ingénierie statique, merci d'avoir sur votre machine :

- objdump/gcc/readelf/strings
- Ghidra 9.1

Pour approfondir/bibliographie :
`https://www.lri.fr/~filliatr/ens/compil/`
`https://perso.telecom-paristech.fr/guilley/ENS/program_2019_2020.html/`