les écoles supérieures

MINF
Programmation des PIC32MX

Chapitre 4

Pile et sous-programmes

Théorie PIC32MX

Christian HUBER (CHR)
Serge CASTOLDI (SCA)
Version 1.9 novembre 2017

CONTENU

4. PILE ET SOUS-PROGRAMMES

Ce chapitre décrit le mécanisme de la pile, son usage lors des appels de sous-programmes, ainsi que les mécanismes utilisés par le compilateur C pour passer les paramètres aux sous-programmes.

Comme la gestion de la pile du PIC32MX est "software", on trouve en partie des informations sur la gestion de la pile dans la documentation du compilateur :

- Document MPLAB-XC32-Users-Guide.pdf (documentation du compilateur xc32).
  Ce document contient 2 parties intéressantes :
  - 7.4 Stack
  - Chapter 15. Main, Runtime Start-up and Reset

  Ce chapitre se base sur le compilateur xc32 v1.42.

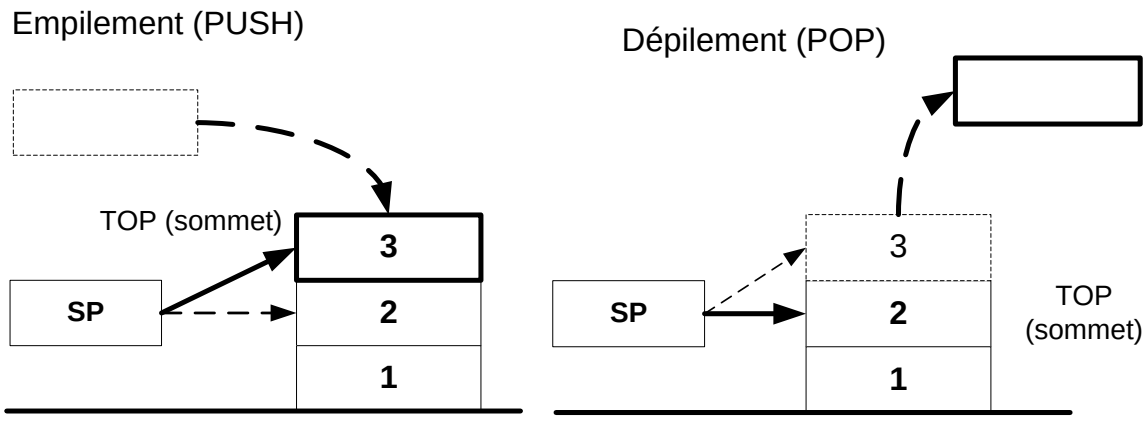
4.1. LE MÉCANISME DE LA PILE

Le mécanisme de pile (en anglais *stack*), s'appuie sur deux éléments :

- La zone de mémoire pour la pile (*stack space*),
- Le pointeur de pile (*stack pointer).*

La pile est principalement utilisée pour sauvegarder et restaurer les adresses de retour des sous-programmes.

Le principe de la pile est un stockage LIFO : Last In, First Out. C'est-à-dire comme une pile de carton, le premier élément que l'on peut reprendre est celui que l'on a empilé en dernier.



La dernière valeur placée sur la pile s'appelle le sommet (*top of stack*).

4.2. MÉCANISME DE LA PILE DU PIC32

Paragraphe 7.4 de la documentation du compilateur.

7.4 STACK

The PIC32 devices use what is referred to in this user's guide as a "software stack". This is the typical stack arrangement employed by most computers and is ordinary data memory accessed by a push-and-pop type instruction and a stack pointer register. The term "hardware stack" is used to describe the stack employed by Microchip 8-bit devices, which is only used for storing function return addresses.

The PIC32 devices use a dedicated stack pointer register `sp` (register 29) for use as a software Stack Pointer. All processor stack operations, including function calls, interrupts and exceptions, use the software stack. It points to the next free location on the stack. The stack grows downward, towards lower memory addresses.

By default, the size of the stack is 1024 bytes. The size of the stack can be changed by specifying the size on the linker command line using the `--defsym_min_stack_size` linker command line option. An example of allocating

a stack of 2048 bytes using the command line is:

```
xc32-gcc foo.c -Wl,--defsym,_min_stack_size=2048
```

The run-time stack grows downward from higher addresses to lower addresses. Two working registers are used to manage the stack:

• Register 29 (`sp`) – This is the Stack Pointer. It points to the next free location on the stack.

• Register 30 (`fp`) – This is the Frame Pointer. It points to the current function's frame.

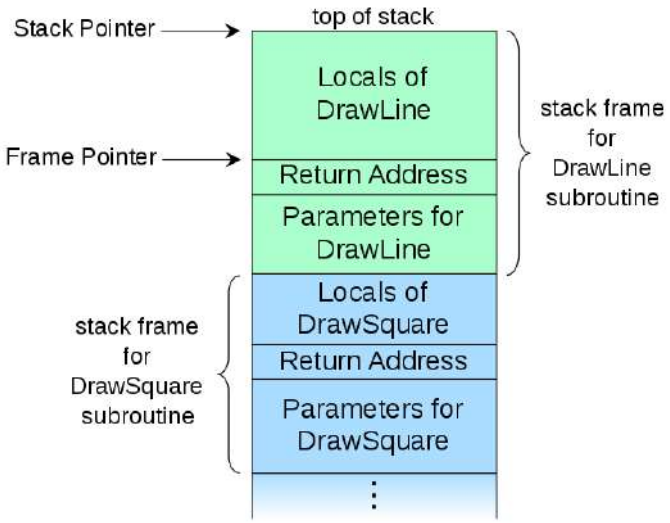
No stack overflow detection is supplied.

The C/C++ run-time start-up module initializes the stack pointer during the start-up and initialization sequence, see Section 15.3.3 "Initialize Stack Pointer and Heap".

On peut extraire de ce document les informations suivantes :

- Gestion software de la pile, contrairement au PIC18F.
- La taille par défaut du stack est de 1024 bytes, extensible par une commande.
- La stack grandit vers le bas (des adresses hautes vers les adresses basses).
- Registre 29 (SP) - Stack pointer (pointeur de pile). Il pointe sur la prochaine place libre sur le stack.
- Registre 30 (FP) - Frame pointer. Il pointe au début du frame courant. C'est une copie du stack pointer avant l'appel de la fonction courante.

La figure ci-dessous illustre les notions de stack pointer et frame pointer :



Source : Wikipédia

La fonction DrawLine est appelée par la fonction DrawSquare.
Les opérations sont, dans l'ordre :

1. La fonction DrawSquare place sur la pile les paramètres d'appel de DrawLine.
2. L'adresse de retour est placée sur la pile, puis la fonction DrawLine est appelée.
3. La fonction DrawLine place ses variables locales sur la pile.

Chaque fonction possède son stack frame (son "bloc" de pile qu'elle utilise). Les variables locales peuvent être référées par rapport au frame pointer.

4.3. INITIALISATION DU STACK ET DU HEAP

Paragraphe 15.3.3

15.3.3 Initialize Stack Pointer and Heap

The Stack Pointer (`sp`) register must be initialized in the start-up code. To enable the start-up code to initialize the `sp` register, the linker must initialize a variable which points to the end of KSEG0/KSEG1 data memory[1].

The linker allocates the stack to KSEG0 on devices featuring an L1 data cache. It allocates the stack to KSEG1 on devices that do not have an L1 cache.

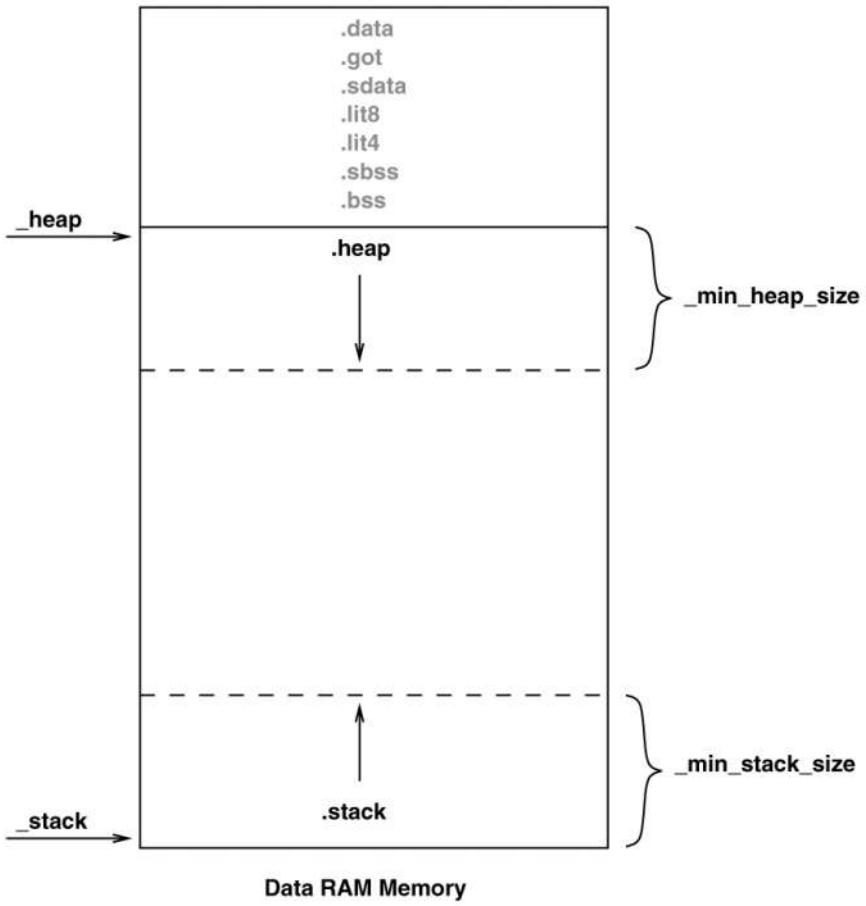
This variable is named `_stack`. The user can change the minimum amount of stack space allocated by providing the command line option `--defsym _min_stack_size=`*N* to the linker. `_min_stack_size` is provided by the linker script with a default value of `1024`. On a similar note, the user may wish to utilize a heap with their application. While the start-up code does not need to initialize the heap, the standard C libraries (`sbrk`) must be made aware of the heap location and its size. The linker creates a variable to identify the beginning of the heap. The location of the heap is the end of the utilized KSEG0/KSEG1 data memory.

The linker allocates the heap to KSEG0 on devices that have an L1 cache. It allocates the heap to KSEG1 on devices that do not have an L1 cache.

This variable is named `_heap`. A user can change the minimum amount of heap space allocated by providing the command line option `--defsym _min_heap_size=`*M* to the linker. If the heap is used when the heap size is set to zero, the behavior is the same as when the heap usage exceeds the minimum heap size. Namely, it overflows into the space allocated for the stack.

The heap and the stack use the unallocated KSEG0/KSEG1 data memory, with the heap starting from a low address in KSEG0/KSEG1 data memory, and growing upwards towards the stack while the stack starts at a higher address in KSEG1 data memory and grows downwards towards the heap. The linker attempts to allocate the heap and stack together in the largest gap of memory available in the KSEG0/KSEG1 data memory region. If enough space is not available based on the minimum amount of heap size and stack size requested, the linker issues an error.

4.3.1. ORGANISATION DU STACK ET DU HEAP



4.3.2. LES REGISTRES DU PIC32

REGISTERS		
0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

C'est le registre 29 sp stack pointer qui est utilisé pour gérer la pile. Le registre 30 fp/s8 frame pointer permet de pointer sur des blocs (frame) dans la pile.

4.4. PUSH AND POP INSTRUCTIONS

Sources :
Les explications qui suivent sont tirées du site web :
http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Mips/stack.html
Understanding the Stack - Daté du 22.06.2003, consulté le 2.11.2017

Certains processeurs disposent d'instructions **push** et **pop** spécifiques. Dans le cas de l'architecture MIPS, il n'y en a pas. Il est possible de réaliser l'équivalent de ces instructions en manipulant directement le stack pointer.

Par convention, on utilise le registre $r29 (sp) comme pointeur de pile.

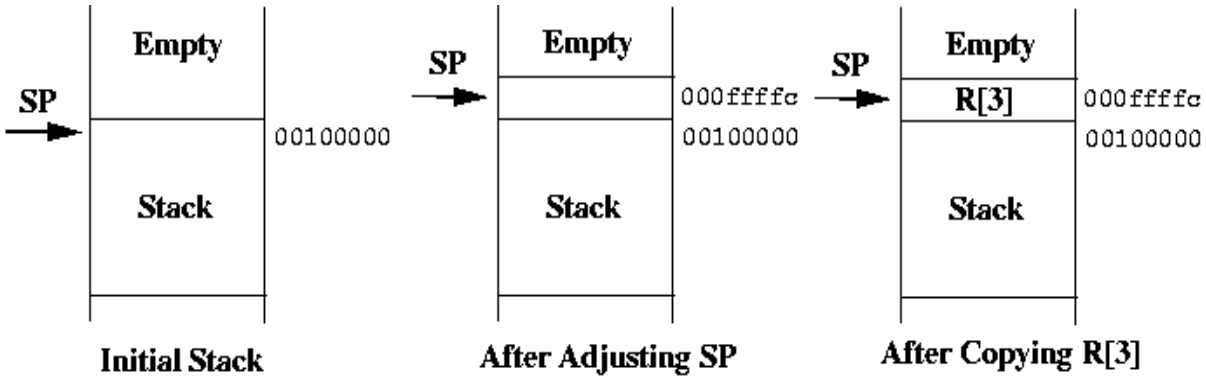
4.4.1. RÉALISATION DE L'ACTION PUSH

Voici comme exemple la réalisation du **push $r3,** ce qui correspond à déposer le registre $r3 sur la pile.

```
push: addi $sp, $sp, -4      # Decrement stack pointer by 4
      sw   $r3, 0($sp)       # Save $r3 to stack
```

Le label push n'est pas nécessaire; il sert de commentaire.

Voici un diagramme qui illustre l'action push :



Le diagramme de gauche montre la situation avant l'action.

Le diagramme du centre illustre l'effet du décrément de 4 du stack pointer (l'adresse diminue, ce qui fait graphiquement monter le pointeur). Une donnée de 32 bits (4 bytes) est mise sur la pile, d'où la valeur 4.

Le diagramme de gauche montre la situation après la copie du contenu du registre $r3. La valeur de $r3 est placée à l'adresse **0x000f'fffc**

Remarque : il est possible d'arriver au même résultat de la manière suivante :

```
push:  sw $r3, -4($sp)    # Copy $r3 to stack
```

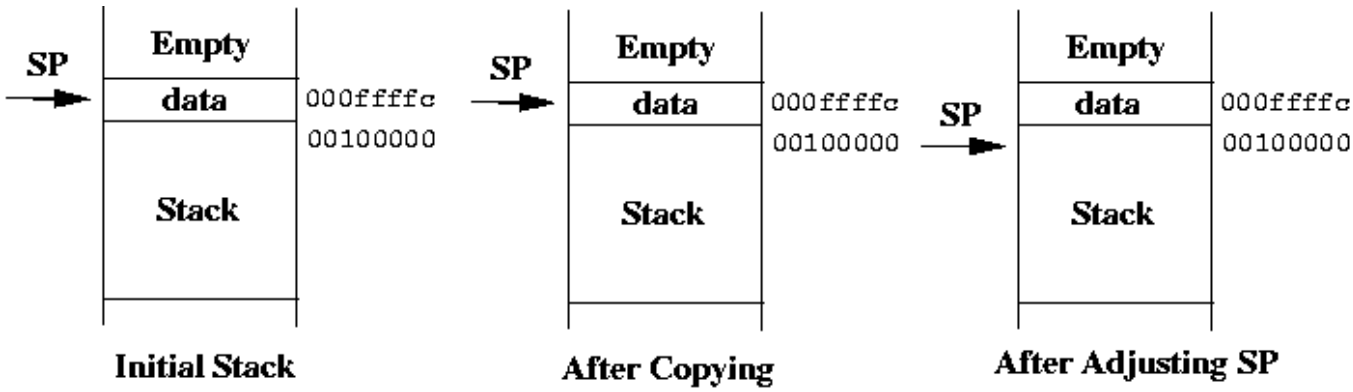
Cependant, dans la pratique on préfère la 1ère méthode pour avoir des offsets positifs par rapport au SP, surtout lorsque l'on push plusieurs éléments.

4.4.2. RÉALISATION DE L'ACTION POP

La récupération de la valeur (popping off the stack) correspond à la manœuvre inverse. D'abord on copie le data du stack dans le registre, puis on ajuste la valeur du stack pointer.

```
pop:  lw   $r3, 0($sp)    # Copy from stack to $r3
      addi $sp, $sp, 4    # Increment sp by 4
```

Voici l'illustration de l'action Pop :



Le diagramme de gauche montre la situation initiale.

Le diagramme du centre montre la situation après la copie. La valeur "data" est copiée dans le registre que l'on ne voit pas sur le diagramme.

Le diagramme de droite montre la situation après ajustement du stack. Le data reste sur le stack mais n'est plus accessible. Il sera vraisemblablement écrasé lors d'une action Push.

4.4.3. ACTION PUSH ET POP AVEC PLUSIEURS VALEURS

Lorsque l'on doit placer plusieurs valeurs sur le stack, on effectue alors un ajustement unique du stack qui correspond à la taille de l'ensemble. De même pour la récupération, on fera un ajustement unique.

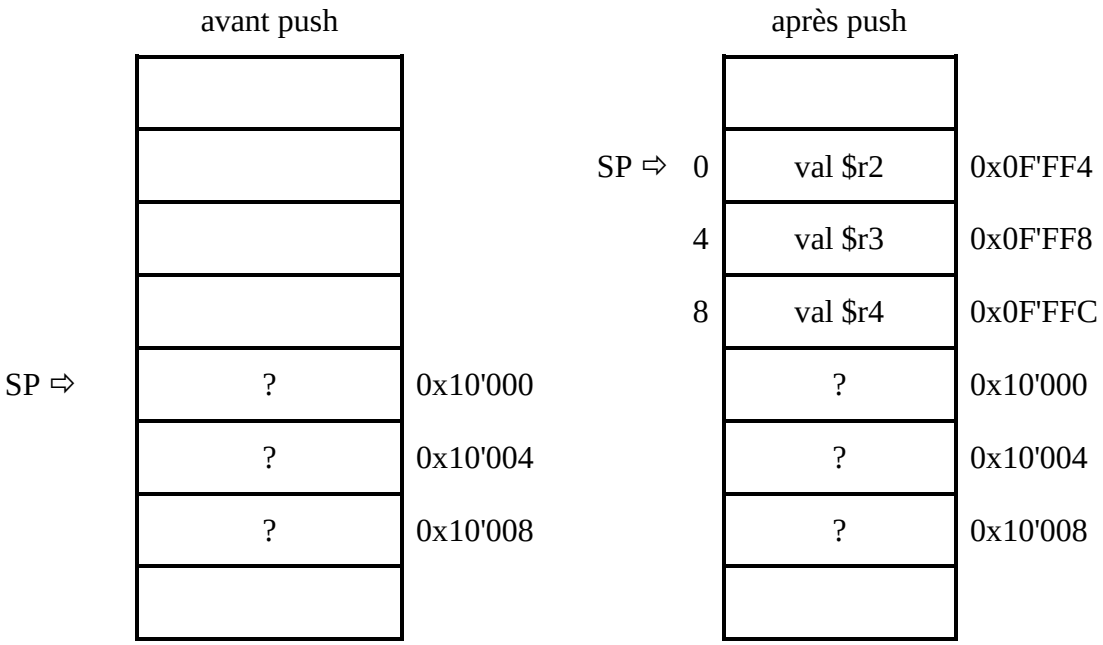
Par exemple, si l'on souhaite placer les registres $r2, $r3 et $r4 sur la pile, cela donne le code suivant :

```
push:  addi $sp, $sp, -12  # Decrement sp by 12
       sw   $r2, 0($sp)    # Save $r2 to stack
       sw   $r3, 4($sp)    # Save $r3 to stack
       sw   $r4, 8($sp)    # Save $r4 to stack
```

Comme chaque registre a une taille de 4 bytes, il est nécessaire de décrémenter la valeur du stack pointer de 12 (3 *4) pour disposer de l'espace nécessaire pour stocker les 3 registres.

L'ordre de stockage des registres est libre; le choix s'est naturellement porté sur l'ordre croissant des registres.

4.4.3.1. ILLUSTRATION DU PUSH DE 3 REGISTRES

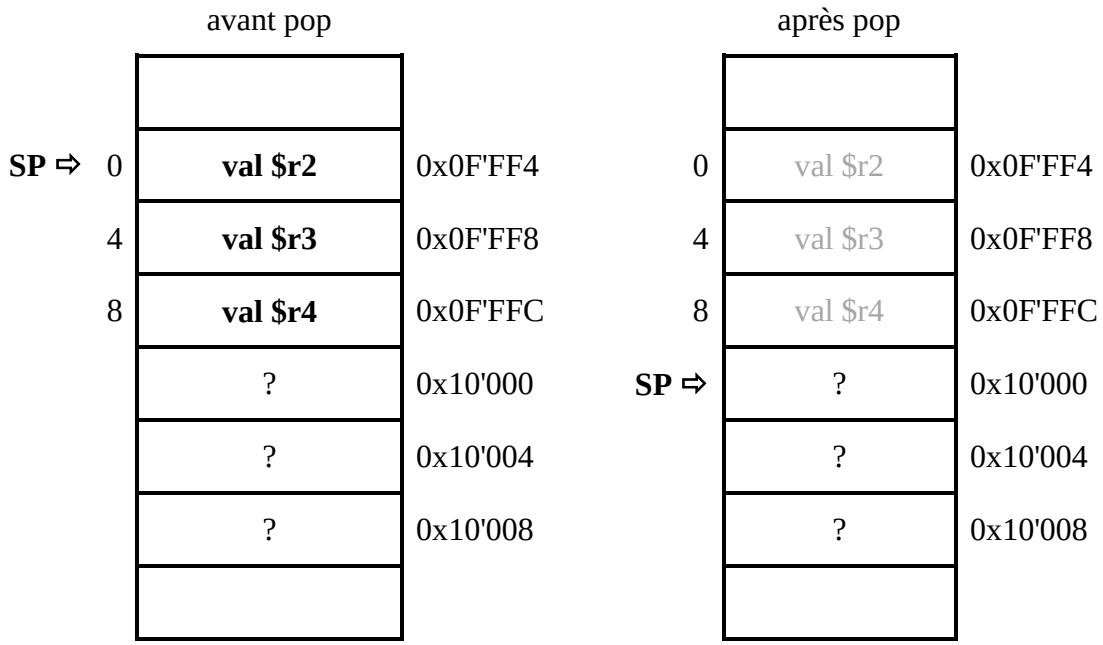


4.4.3.2. ACTION POP DES 3 REGISTRES

L'action pop correspond à la manœuvre inverse : copies puis ajustement.

```
pop:  lw   $r2, 0($sp)   # Copy from stack to $r2
      lw   $r3, 4($sp)   # Copy from stack to $r3
      lw   $r4, 8($sp)   # Copy from stack to $r4
      addi $sp, $sp, 12  # Increment sp by 12
```

4.4.3.3. ILLUSTRATION DU POP DE 3 REGISTRES



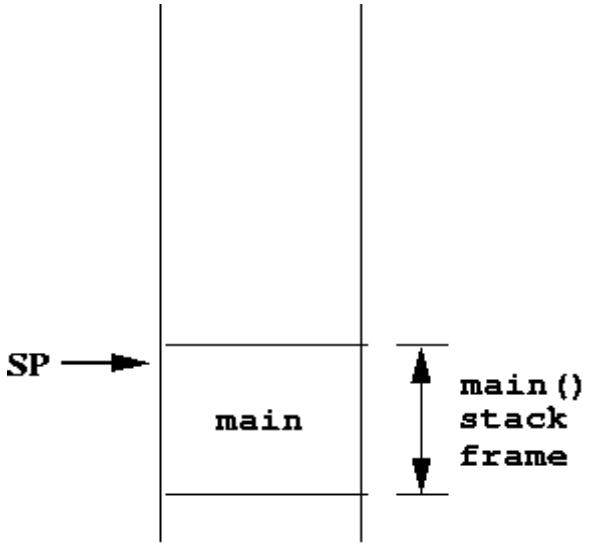
Remarque : après l'action pop, les valeurs ont été copiées, mais elles restent dans la pile. La position du SP est modifiée.

4.5. RELATION ENTRE PILE ET FONCTIONS

4.5.1. PILE ET FONCTIONS

Let's now see how the stack is used to implement functions. For each function call, there's a section of the stack reserved for the function. This is usually called a ***stack frame.***

Let's imagine we're starting in **main()** in a C program. The stack looks something like this:

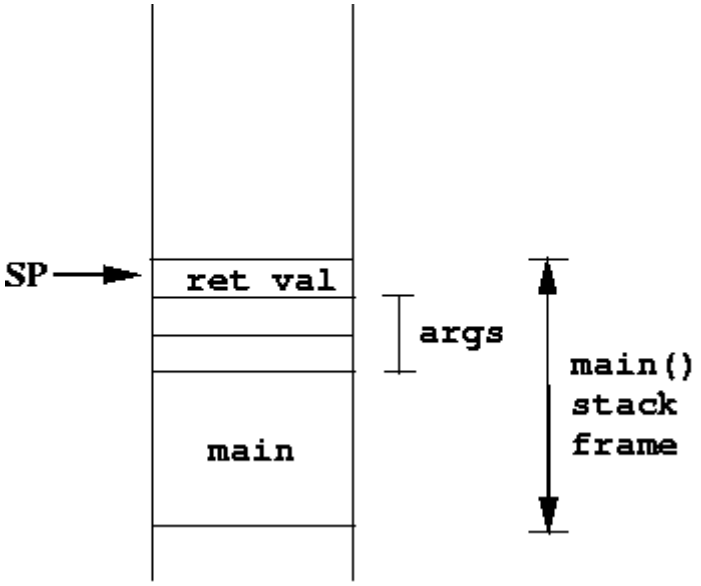


We'll call this the *stack frame* for **main()**. It is also called the *activation record*. A stack frame exists whenever a function has started, but yet to complete.

Suppose, inside of body of **main()** there's a call to **foo()**. Suppose **foo()** takes two arguments. One way to pass the arguments to **foo()** is through the stack. Thus, there needs to be assembly language code in **main()** to "push" arguments for **foo()** onto the the stack.

4.5.1.1. SITUATION STACK LORS DE LA PRÉPARATION DE L'APPEL DE FOO

Voici le contenu du stack frame du main() lors de la préparation de l'appel de la fonction foo().

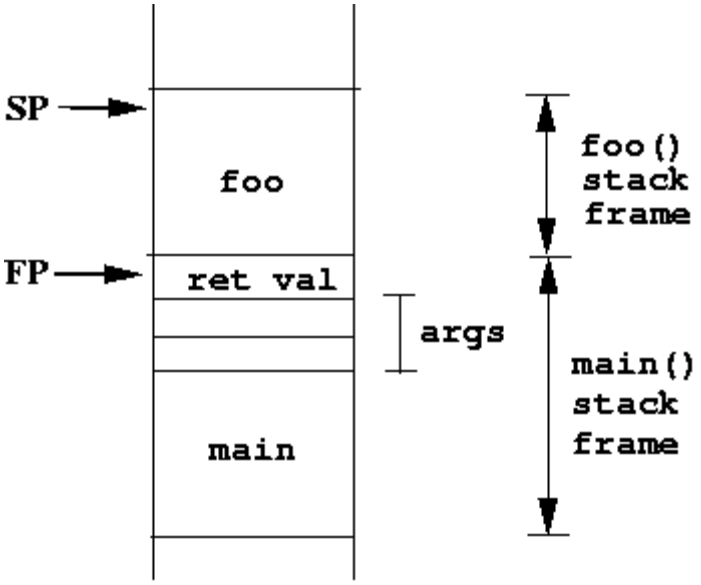


On constate une augmentation de la taille du stack frame du main. Les arguments sont placés dans ce stack frame et une place est prévue pour stocker la valeur de retour de la fonction.

☝ La valeur de retour (ret val) sera mise en place à la fin de l'exécution de la fonction.

4.5.1.2. SITUATION LORS DE L'EXÉCUTION DE FOO

La fonction **foo()** peut avoir besoin de variables locales, ce qui conduit à prendre une zone supplémentaire sur le stack (en anglais : ***foo()*** *needs to push some space on the stack*).



Cela crée la nécessité d'un pointeur supplémentaire : le FP (frame pointer), pour mémoriser l'ancienne position du SP.

foo() can access the arguments passed to it from **main()** because the code in **main()** places the arguments just as **foo()** expects it.

We've added a new pointer called **FP** which stands for ***frame pointer***. **The frame pointer points to the location where the stack pointer *was***, just before **foo()** moved the stack pointer for **foo()**'s own local variables.

Having a frame pointer is convenient when a function is likely to move the stack pointer several times throughout the course of running the function. The idea is to keep the frame pointer fixed for the duration of **foo()**'s stack frame. The stack pointer, in the meanwhile, can change values.

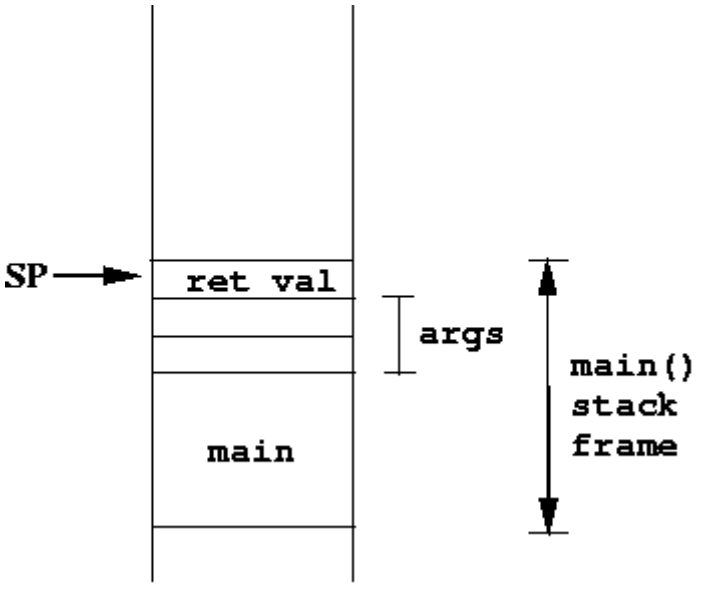
Thus, we can use the frame pointer to compute the locations in memory for both arguments as well as local variables. Since it doesn't move, the computations for those locations should be some fixed offset from the frame pointer.

And, once it's time to exit **foo()**, you just have to set the stack pointer to where the frame pointer is, which effectively pops off **foo()**'s stack frame. It's quite handy to have a frame pointer.

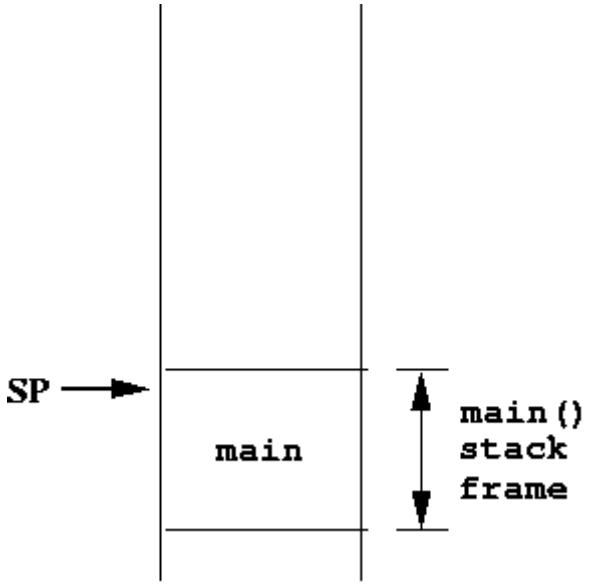
We can imagine the stack growing if **foo()** calls another function, say, **bar()**. **foo()** would push arguments on the stack just as **main()** pushed arguments on the stack for **foo()**.

4.5.1.3. SITUATION APRÈS L'EXÉCUTION DE FOO

So when we exit **foo()** the stack looks just as it did before we pushed on **foo()**'s stack frame, except this time the return value has been filled in.



Once **main()** has the return value, it can pop that and the arguments to **foo()** off the stack.



4.5.2. DÉBORDEMENT DE PILE (STACK OVERFLOW)

While stacks are generally large, they don't occupy all of memory. It is possible to run out of stack space.

For example, consider the code we had for **factorial**.

```
int fact( int n ) {
   if ( n == 0 )
      return 1 ;
   else
      return fact( n - 1 ) * n ;
 }
```

Suppose **fact(-1)** is called. Then, the base case is never reached (well, it might be reached once we decrement so far that **n** wraps around to 0 again). This causes one stack frame after another to be pushed. Once the stack limit has been reached, we enter into invalid memory addresses, and the operating system takes over and kills your programming, telling you your program has a stack overflow.

Probably the most common cause of stack overflow is a recursive function that doesn't hit the base case soon enough. For fans of recursion, this can be a problem, so just keep that in mind.

Some languages (say, ML : Metalanguage) can convert certain kinds of recursive functions (called "tail-recursive" functions) into loops, so that only a constant amount of space is used.

4.5.3. IMPLÉMENTATION DANS LE MIPS32

In the previous discussion of function calls, we said that arguments are pushed on the stack and space for the return value is also pushed.

This is how CPUs used to do it. With the RISC revolution (admittedly, nearly 20 years old now) and large numbers of registers used in typical RISC machines, the goal is to (try and) avoid using the stack.

Why? The stack is in physical memory, which is RAM. Compared to accessing registers, accessing memory is much slower, probably on the order of 100 to 500 times as slow to access RAM than to access a register.

MIPS has many registers, so it does the following:

- There are four registers used to pass arguments: $a0, $a1, $a2, $a3.
- If a function has more than four arguments, or if any of the arguments is a large structure that's passed by value, then the stack is used.
- There must be a set procedure for passing arguments that's known to everyone based on the types of the functions. That way, the caller of the function knows how to pass the arguments, and the function being called knows how to access them. Clearly, if this protocol is not established and followed, the function being called would not get its arguments properly, and would likely compute bogus values or, worse, crash.
- The return value is placed in registers $v0, and if necessary, in $v1.

In general, this makes calling functions a little easier. In particular, the calling function usually does not need to place anything on the stack for the function being called.

However, this is clearly not a panacea. In particular, imagine **main()** calls **foo()**. Arguments are passed using **$a0** and **$a1**, say.

What happens when **foo()** calls **bar()**? If **foo()** has to pass arguments too, then by convention, it's supposed to pass them using **$a0** and **$a1**, etc. What if **foo()** needs the argument values from **main()** afterwards?

To prevent its own arguments from getting overwritten, **foo()** needs to save the arguments to the stack.

☝ Thus, we don't entirely avoid using the stack.

4.5.4. LEAF PROCEDURES

In general, using registers to pass arguments and for return values doesn't prevent the use of the stack. Thus, it almost seems like we postpone the inevitable use of the stack. Why bother using registers for return values and arguments?

Eventually, we have to run code from a leaf procedure. This is a function that does not make any calls to any other functions. Clearly, if there were no leaf procedures, we wouldn't exit the program, at least, not in the usual way (If this isn't obvious to you, try to think about why there must be leaf procedures).

In a leaf procedure, there are no calls to other functions, so there's no need to save arguments on the stack. There's also no need to save return values on the stack. You just use the argument values from the registers, and place the return value in the return value registers.

4.6. MÉCANISME APPEL ET PASSAGE DES PARAMÈTRES

4.6.1. PRINCIPE CALL ET RETURN

Voici le principe de réalisation du CALL à une fonction et du RETURN.

4.6.1.1. RÉALISATION DU CALL

Le "call" utilise l'instruction **JAL** (Jump And Link) :

```
jal address      # $ra = $pc + 4; goto address;
```

Le JAL effectue 2 actions:

- le stockage de l'adresse de retour (la prochaine instruction) dans $ra : $ra = $pc + 4
- Le saut à l'adresse de la fonction

4.6.1.2. RÉALISATION DU RETURN

Le "return" utilise l'instruction JR (Jump Register) :

```
jr $ra           # goto $ra = address retour;
```

4.6.2. APPELS IMBRIQUÉS

Voici un complément obtenu de :
http://jjc.hydrus.net/cs61c/handouts/proced1.pdf

Le mécanisme du JAL et du JR ne convient que pour un appel à un seul niveau et sans paramètre. Pour supporter des appels imbriqués, il est impératif de mémoriser $ra sur la pile.

Les 4 principes à respecter :

- Les valeurs des registres $s0-$s7 doivent être préservées : Ces registres doivent avoir la même valeur en quittant la procédure qu'à l'entrée dans la procédure.
- Idem pour registre $ra : L'adresse contenue doit être disponible à la fin de la procédure et être valable.
- Idem pour les valeurs (adresses) contenues dans $sp et $fp : ces dernières doivent également être préservées.
- Les registres, $t0-$t9, $a0-$a3, et $v0-v1 peuvent être modifiés par la fonction.

4.6.2.1. PRINCIPE DE L'APPEL IMBRIQUÉ

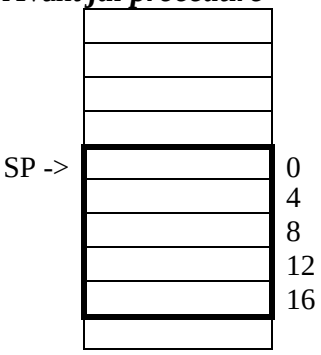
Voici une situation d'une fonction nommée *procedure* qui en appelle une 2ème (*procedure2*).
L'appel est réalisé par ***jal procedure***.

```
procedure:
Prologue {  addi $sp, $sp, -12
            sw $ra, 8($sp)
            sw $s0, 4($sp)
            sw $s1, 0($sp)
Body     {  addi $s0, $zero, 50
            addi $s0, $zero, -23
            add $a0, $s0, $s1
            jal procedure2
Epilogue {  lw $s1, 0($sp)
            lw $s0, 4($sp)
            lw $ra, 8($sp)
            addi $sp, $sp, 12
            jr $ra
```

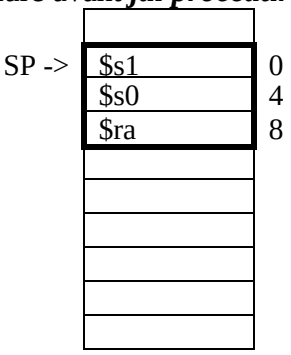
Cette fonction ne reçoit pas de paramètre et ne retourne pas de valeur.

- Dans le **prologue**, il y a création de place sur le stack, puis sauvegarde de 3 registres.
- Dans le corps de la fonction (*body*), il y a modification des valeurs des 3 registres et l'appel de procédure2.
- Dans l'**épilogue**, il y a récupération des 3 registres et réajustement du stack en prévision du retour.

Avant ***jal procedure***



Dans procedure avant ***jal procedure2.***



Remarque : Cet exemple ne permet pas de connaitre le contenu du stack frame du contexte appelant.

4.6.3. APPEL ET PASSAGE DE PARAMÈTRE

Bien souvent, lors de l'appel d'une fonction, il est nécessaire de lui fournir des paramètres et aussi de récupérer le résultat.

Au niveau du MIPS32, 4 registres sont prévus pour cela : **$a0** à **$a3**. Si davantage d'arguments sont nécessaires, ils seront placés dans la pile. Dans ce cas, ils doivent être placés dans la pile avant les éléments habituellement sauvés, car l'utilisateur de la fonction est celui qui alloue de la mémoire et la libère après l'appel de la fonction.

Voici la situation de la fonction Ftest qui elle-même appelle la fonction FDivAB.

```
// Fonction FDivAB
int FDivAB (int A, int B)
{
    return (A / B);
}

// Ftest
int Ftest (int ValA, int ValB)
{
    int Ratio;
    Ratio =  FDivAB(ValA, ValB);
    return Ratio;
}
```

Ainsi que le programme principal :

```
void main() {
    int Res1, Res2;
    int V1 = 10;
    int V2 = 20;
    int V3 = 30;
    int V4 = 40;

     // Appels de la fonction Ftest
    Res1 = Ftest(V1, V2);
    Res2 = Ftest(V3, V4);
}
```

4.6.3.1. FONCTION MAIN EN ASSEMBLEUR

Voici le programme principal en assembleur :

```
24:                     void main() {
9D000088  27BDFFD0      ADDIU SP, SP, -48
9D00008C  AFBF002C      SW RA, 44(SP)
9D000090  AFBE0028      SW S8, 40(SP)
9D000094  03A0F021      ADDU S8, SP, ZERO
25:                         int Res1, Res2;
26:                         int V1 = 10;
9D000098  2402000A      ADDIU V0, ZERO, 10
9D00009C  AFC20010      SW V0, 16(S8)    // V1 en 16(S8)
27:                         int V2 = 20;
9D0000A0  24020014      ADDIU V0, ZERO, 20
9D0000A4  AFC20014      SW V0, 20(S8)    // V2 en 20(S8)
28:                         int V3 = 30;
9D0000A8  2402001E      ADDIU V0, ZERO, 30
9D0000AC  AFC20018      SW V0, 24(S8)    // V3 en 24(S8)
29:                         int V4 = 40;
9D0000B0  24020028      ADDIU V0, ZERO, 40
9D0000B4  AFC2001C      SW V0, 28(S8)    // V4 en 28(S8)
30:
31:                          // Appel de la fonction  Ftest
32:                         Res1 = Ftest(V1, V2);
9D0000B8  8FC40010      LW A0, 16(S8)
9D0000BC  8FC50014      LW A1, 20(S8)
9D0000C0  0F400010      JAL Ftest
9D0000C4  00000000      NOP
9D0000C8  AFC20020      SW V0, 32(S8)  // Res1 en 32(S8)
33:                         Res2 = Ftest(V3, V4);
9D0000CC  8FC40018      LW A0, 24(S8)
9D0000D0  8FC5001C      LW A1, 28(S8)
9D0000D4  0F400010      JAL Ftest
9D0000D8  00000000      NOP
9D0000DC  AFC20024      SW V0, 36(S8)  // Res2 en 36(S8)
34:                     }
9D0000E0  03C0E821      ADDU SP, S8, ZERO
9D0000E4  8FBF002C      LW RA, 44(SP)
9D0000E8  8FBE0028      LW S8, 40(SP)
9D0000EC  27BD0030      ADDIU SP, SP, 48
9D0000F0  03E00008      JR RA
9D0000F4  00000000      NOP
```

prologue

épilogue

On constate :

- Même le **main** contient un prologue et un épilogue
- Le prologue se charge de préparer la place sur la pile et de sauvegarder RA (adresse de retour) et S8/FP (frame pointer).
- Ensuite, tous les stockages de variables locales sont faits à l'aide du frame pointer S8/FP.
- L'épilogue rétablit la situation du début de fonction (l'inverse des opérations du prologue), puis retourne à l'appelant (JR RA).

4.6.3.2. FONCTION FTEST EN ASSEMBLEUR

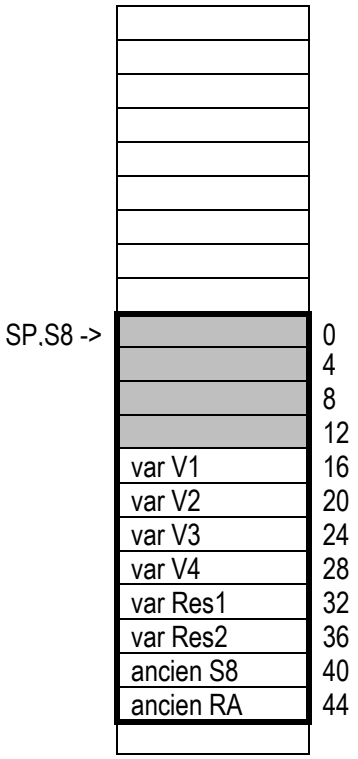
Voici l'équivalent en assembleur de la fonction **Ftest** :

```
int Ftest (int ValA, int ValB)
{
prologue
    ADDIU SP, SP, -32          // Crée place sur stack
    SW RA, 28(SP)              // push $ra (return addr)
    SW S8, 24(SP)              // push $s8 (frame pointer)
    ADDU S8, SP, ZERO          // $s8 = $sp (nouveau fp)
    SW A0, 32(S8)              // push $a0 (sauvegarde ValA)
    SW A1, 36(S8)              // push $a1 (sauvegarde ValB)
  int Ratio;
  Ratio =  FDivAB(ValA, ValB);
    LW A0, 32(S8)              // pop $a0 (passage param. ValA)
    LW A1, 36(S8)              // pop $a1 (passage param. ValB)
    JAL FDivAB
    NOP
    SW V0, 16(S8)              // push $v0
  return Ratio;
    LW V0, 16(S8)              // pop $v0
}
épilogue
    ADDU SP, S8, ZERO          // $sp = $s8
    LW RA, 28(SP)              // pop $ra
    LW S8, 24(SP)              // pop $s8
    ADDIU SP, SP, 32           // maj du SP
    JR RA                      // retour
```

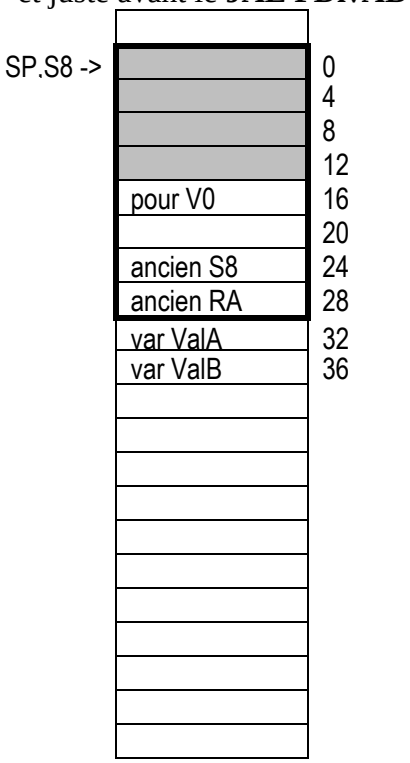
4.6.3.3. EVOLUTION PILE AVEC FONCTION FTEST

Pour bien comprendre la manœuvre, voici la situation du stack :

Avant l'appel de Ftest



Dans la fonction après
addiu $sp, $sp, -32
et juste avant le **JAL FDivAB**



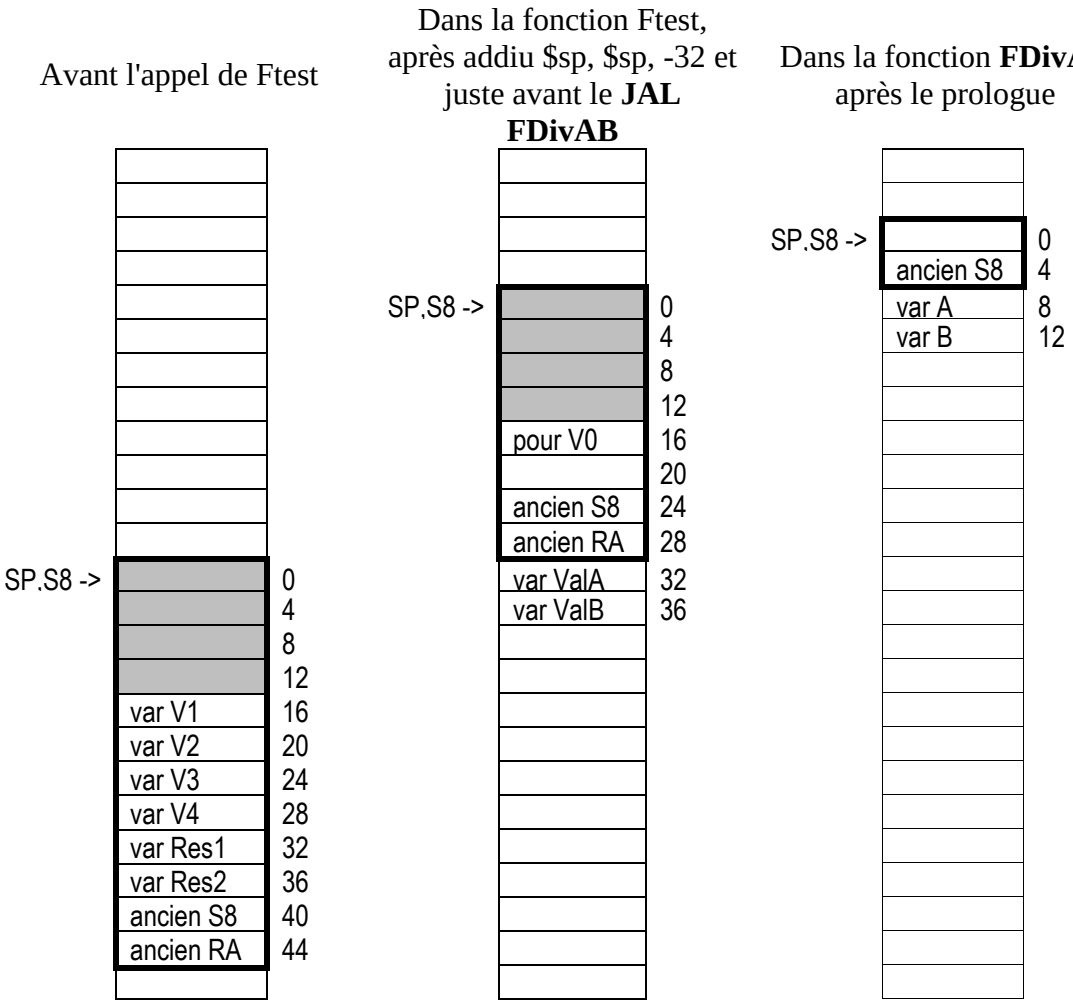
☝ Le listing en assembleur du main permet de comprendre le contenu de la pile.

On remarque que dans le stack frame du main, les 4 premiers mots sont laissés libres pour le futur stockage de $a0, $a1, $a2 et $a3.

On remarque aussi que les variables du main sont placées dans le stack frame. C'est d'ailleurs également le cas pour toute autre fonction.

4.6.3.4. EVOLUTION DE LA PILE AVEC APPEL IMBRIQUÉ

Pour mieux comprendre ce qui ce passe, il est intéressant d'obtenir la situation de la pile lorsque que l'on effectue l'appel à la fonction intérieure.



Appel de la fonction FDivAB dans Ftest :

```
Ratio =  FDivAB(ValA, ValB);
  LW A0, 32(S8)          // pop $a0 (ValA)
  LW A1, 36(S8)          // pop $a1 (ValB)
  JAL FDivAB
```

On remarque que le stack frame du contexte appelant laisse de la place pour les variables. La fonction FDivAB ne sauve pas RA car elle n'appelle pas d'autre fonction (leaf procedure).

4.6.3.5. FONCTION FDIVAB EN ASSEMBLEUR

Voici l'équivalent en assembleur de la fonction FDivAB :

```
// Fonction FDivAB
int FDivAB (int A, int B)
{
    ADDIU SP, SP, -8      // Crée place sur stack
    SW S8, 4(SP)          // push $s8 (frame pointer)
    ADDU S8, SP, ZERO     // $s8 = $sp (nouveau fp)
    SW A0, 8(S8)          // push $a0 (passage param. A)
    SW A1, 12(S8)         // push $a1 (passage param. B)
    return (A / B);
    LW V1, 8(S8)          // Effectue A / B
    LW V0, 12(S8)
    DIV V1, V0
    TEQ V0, ZERO
    MFHI V0
    MFLO V0               // Résultat dans V0
}
    ADDU SP, S8, ZERO     // $sp = $s8
    LW S8, 4(SP)          // pop $s8
    ADDIU SP, SP, 8       // maj du SP
    JR RA                 // Retour
```

(prologue : lignes 4 à 8 ; épilogue : lignes 16 à 19)

On observe l'utilisation directe de RA sans sauvegarde.

4.7. CONCLUSION

Ce chapitre a tenté de montrer le mécanisme de la pile et de son utilisation avec des frame pour assurer le passage de paramètres aux fonctions et la gestion des retours.

4.8. HISTORIQUE DES VERSIONS

4.8.1. VERSION 1.0 FÉVRIER 2014

Création du document (en grande partie en anglais) et découverte du mécanisme.

4.8.2. VERSION 1.5 NOVEMBRE 2014

Saut à la version 1.5 pour cohérence avec la nouvelle version du cours liée à Harmony. Amélioration des explications en anglais par des titres et remarques en français.

4.8.3. VERSION 1.7 NOVEMBRE 2015

Saut à la version 1.7 pour cohérence avec la nouvelle version du cours liée à Harmony. L'exemple n'a pas été adapté à Harmony. Maintien des explications en anglais.

4.8.4. VERSION 1.7BIS NOVEMBRE 2015

Reprise des exemples d'évolution de la pile sur la base d'un exemple plus simple. Suppression de l'exemple avec DispHex.

4.8.5. VERSION 1.8 NOVEMBRE 2016

Adaptation des références à la documentation. Traduction en français de la section push et pop. La section gestion pile et trame reste en anglais. Complément de l'exemple pratique.

4.8.6. VERSION 1.9 NOVEMBRE 2017

Reprise et relecture par SCA.