ARC: Computer Architecture

tanguy.risset@insa-lyon.fr
Lab CITI, INSA de Lyon
Version du March 16, 2023

Tanguy Risset

March 16, 2023

Table of Contents

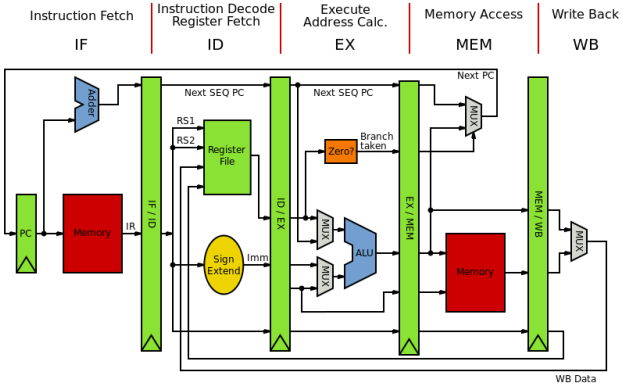
1. MIPS ISA
2. Function, procÃ©dure et Pile d'execution
3. Coming back to MIPS
4. Some additionnal useful information
   - Example of MIPS code

Study a real ISA: MIPS

- We study in more detail a particular assembly code
- Course inspired from
  - Architecture course of Peter Niebert and SÃ©verine Fratani (U. Marseille) `http://pageperso.lif.univ-mrs.fr/~peter.niebert/archi2014.php`
  - MIPS web site `https://www.mips.com/`
  - And of Course F. de Dinechin IF Architecture course (with bits of Christian Wolf)

MIPS Processor

- MIPS stands for *Microprocessor without Interlocked Pipeline Stages*
- MIPS designed by MIPS Computer Systems in 1985.
- Many version up to today (MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V and MIPS32, MIPS64 as well)
- Used in PC, and servers (DEC, NEC, Silicon Graphics) and for video games (Nintendo 64, Sony PlayStation, PlayStation 2)
- Gave birth to RISC-V, an open-source architecture.



MIPS Processor organisation

- a *register-to-register* (or *load/store*) architecture
- MIPS use 3-adress instructions (destination is the first operand)
- 32 registers
- A program counter ($PC) of 32 bits
- an Instruction register ($IR) of 32 bits
- Addressable memory of 2^{32} bytes
  - $\Leftrightarrow 2^{30}$ words of 4 bytes

understanding MIPS assembly

- From C to assembly:

```
mipsel-linux-gcc prog.c -S -o prog.s
```

prog.c

```
...
i = N*N + 3*N;
...
```

prog.s

```
...
  lw      $t0, 4($gp)        # fetch N
  mult    $t0, $t0, $t0      # N*N
  lw      $t1, 4($gp)        # fetch N
  ori     $t2, $zero, 3      # 3
  mult    $t1, $t1, $t2      # 3*N
  add     $t2, $t0, $t1      # N*N + 3*N
  sw      $t2, 0($gp)        # i = ...
...
```

MIPS assembly: compiler optimization (academic)

- From C to optimized assembly:

```
mipsel-linux-gcc prog.c -S -O3 -o prog.s
```

prog.c

```
...
i = N*N + 3*N;
...
```

prog.s

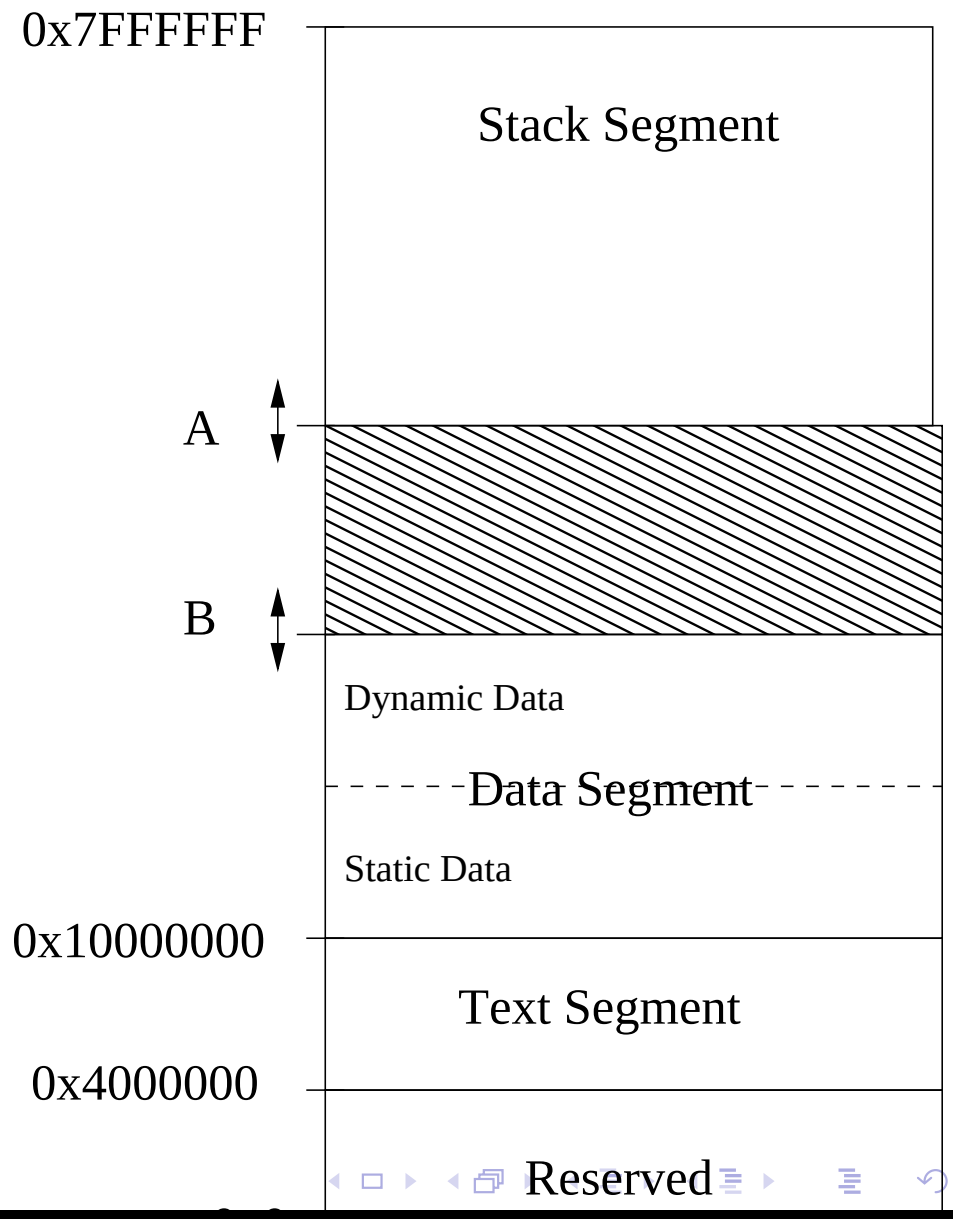
```
...
lw      $t0, 4($gp)        # fetch N
add     $t1, $t0, $zero    # cp N to $t1
addi    $t1, $t1, 3        # N+3
mult    $t1, $t1, $t0      # N*(N+3)
sw      $t1, 0($gp)        # i = ...
...
```

MIPS register

- 32 registers in the *register file*
- Named
  - by their number: `$0 $1 ...$31`
  - or by their name `$zero $at $v0 $v1 $a0 ...$a3 ...`
- $0 ($zero) contains value 0
- `$a0 ...$a3` are used to pass (first four) arguments of a function call
- `$v0 $v1` are used to transmit functions result
- `$s0 ...$s7` and $t0 ... $t9 are working registers, used for CPU computations
- `$sp` is the stack pointer
- `$fp` is the frame pointer (explained later)
- `$ra` contains the return address (after the end of current function)
- $gp is a pointer to global area
- `$k0, $k1` and `$at` are reserved register (for kernel and assembler)

MIPS Memory map

- The Memory Map is a convention to organize memory that must respect each code to be compatible with others.
- The MIPS memory map (very similar to all memory map) is simple
- Here we have only one physical memory chip: the RAM.



MIPS assembly addressing mode

- Addressing mode means: how the address is computed in an assembly instruction

format	address computation
$register	content of register
imm	immediate value
imm ($register)	immediate + content of register
label	addresse of label
label ± imm	addresse of label ± immediate value
label ± imm (register)	addresse of label ± (immediate value + content of register)

Example of MIPS adressing mode

- `add $s0, $s2, $s1`
  - puts in `$s0` the value of `$s1` plus the value of `$s2`.
  - `$s0=$s1+$s2`
- `addi $s0, $s1, 1`
  - puts in `$s0` the value of `$s1` plus 1.
  - `$s0=$s1+1`
- `lw $s0, 10($s3)`
  - puts in `$s0` the value situated in memory at the address obtained by adding 10 to the content of $s3.
  - `$s0=Memory[$s3+10]`
- `bne $s0, $s3, label`
  - branch to address of `label` if values in `$s0` and `$S3` are different.
  - `if ($s0 != $s3) then $PC=label`

Format of MIPS instructions

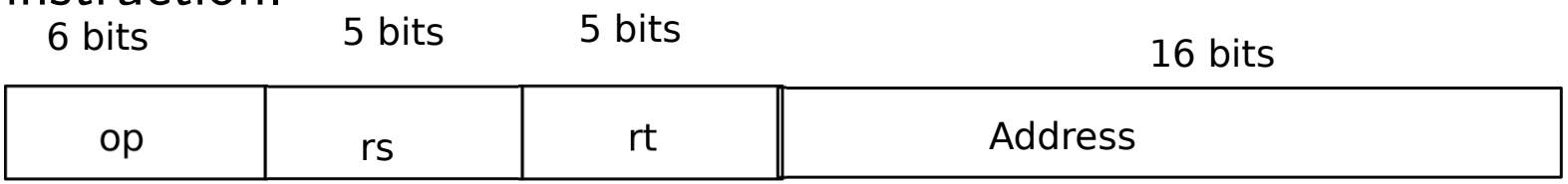
- 3 types of format: R-Type, I-Types and J-Types
- R-types:

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	func

  - Used for 3-register instructions
  - `op` is the operation code or *opcode* that specifies the operation
  - `rs` and `rt` are the first and second source register
  - `rd` is the destination register
  - `shamt` is used for shift instruction
  - `func` is used with `op` to select arithmetic operation

I-Types instruction

- I-Types instruction are used for load, store, branch and immediate instruction.



  - `rs` is a source register (an address) for loads, store
  - `rs` is an operand for conditionnal branch
  - `rt` is a source register for branch
  - `rt` is a destination register for other I-Types instruction
  - The `address` field is a 16 bit's integer in two's-complement code , ranging from -32 768 to 32 767 (remind that this is a problem in many cases)

J-Types instruction

- J-Types instruction are used for Jump to absolute address

6 bits	26 bits
op	Address

  - The `address` field is a 26 bit's integer containing the address of the *word*, hence the real address is obtain by multiplying by four (shifting two bits).
  - can jump from address 0 to 2^{28}=256MB from $PC.
  - For longer jump, on can use the instruction `jr`:
    `jr $ra`
    jump to 32 bit address contained in register $ra

Basic arithmetic and logic instruction

- R-Types instructions: add, sub, mul, div, and, or, xor
  - `add $t0, $t1, $t2 // $t0 = $t1 + $t2`
  - `mul $s0, $s1, $a0 // $s0 = $s1 * $a0`, pseudo
- I-types for immediate operand operation:
  - `addi $t0, $t1, 4 // $t0 = $t1 + 4`
  - `addi $t0, $0, 4 // $t0 = 4`
  - `li $t0, 4 // $t0 = 4, pseudo`

Load and store

- MIPS load and store operation use *indexed addressing*
  - the address operand specifies a signed constant and a register
  - These values are added to generate effective address
- byte instruction: `lb` and `sb` transfer one byte
  - `lb $t0, 20($a0)  // $t0=Memory[$a0+20]`
  - `sb $t0, 20($a0)  // Memory[$a0+20]=$t0`
  - `sb` stores only the lowest byte of operand register
- Word instruction: `lw` and `sw` operates on word (i.e. 32 bits)
- Remind that address have to be aligned to 32 bit world, hence must be multiple of 4.

Branches

- Conditional branch
  - `bne $t0, $t1, Label`
  - if `$t0` and `$t1` have different values, the next instruction to execute is at address Label
  - `beq $t0, $t1, Label  // same thing if $t0=$t1`
- Unconditionnal branch
  - `j toto // next instruction executed is at address toto`
  - `jr $s2 // next instruction executed is at address contained in $s2`
- These are the only way of implementing loops in assembly:

```
        li $t2, 0
        li $t3, 1
while:  beq $t1, $0, done
        add $t2, $t1, $t2
        sub $t1, $t1, $t3
        j while
done:
```

```
t2=0
while (t1 != 0) {
   t2 = t2 + t1
   t1=t1-1
}
```

Function control flow in MIPS

- MIPS uses the *jump-and-link* (`jal`) instruction to call functions
  - Example:

    ```
    jal Fact
    ```

  - saves the return address (i.e. the address of the following instruction) in the `$ra` register and jumpt to the address of `Fact`
- At the end of the execution of `Fact`, the instruction `jr $ra` jumps back to the address stored in `$ra`
- Arguments transmited to `Fact` are stored in registers `$a0` ...`$a3`
- Return values of `Fact` are stored in registers `$v0` `$v3`

Who save the register during Function call?

- When a function call occurs: `jal Fact`, who save the register?
  - The Caller (who knows which register he will use after the call)?
  - Or the callee (who knows which register he will use during its execution)?
- This convention is part of the *calling convetion* or ABI *application binary interface*.
- For MIPS:
  - `$t0 - $t9 $a0 - $a3 $v0 $v1` are caller saved (if needed)
  - `$s0 - $s7 $ra` are callee saved (if needed)

Function call example with MIPS

- Let says: function B calls function C
- Function B wants to save `$t0`, `$t1` and `$a0` because it will need them after the return of C.
- this is done using the stack via the stack pointer `$sp`

The Stack

- The stack is use to store all *local* information (in the sense local to the current function)
- That includes (say for function C):
  - local variable
  - Callee saved register if needed
  - Return address (i.e. the instruction following the `jal C` instruction).
  - (sometimes) the parameters passed to C
  - (sometimes) the result of C
  - In many ISA, the parameters and the results are passed through dedicated registers
- All these data constitute the frame of the fonction instance.
- the frame pointeur points to the frame of the current function
- For MIPS, the frame pointer is `$fp`

Function B calls C

```
B   ...                 beguinning of  B
    ...
    sw $t0,0($sp)       saving  $t0 in stack
    sw $t1,-4($sp)      saving  $t1 in stack
    sw $a0,-8($sp)      saving  $a0 in stack
    sub $sp,$sp,12      correct stack pointer
    jal C               call to C function
    lw $a0,4($sp)       restoring return addresse of B from stac
    lw $t1,8($sp)       restoring $s1 from stack
    sw $t0,12($sp)      restoring  $s0
    add $sp,$sp,12      adjusst  stack pointeur value
    ...
    jr $ra              end of B
    ...
```

Sketching code of C function

```
C:
    subu    $sp,$sp,40      # C need 40 Bytes for its frame
    sw      $ra,32($sp)     # store return address (inst. in B)
    sw      $fp,28($sp)     # store frame pointer
    sw      $s0,24($sp)     # store $s0 (because C uses it)
    move    $fp,$sp         # $fp <- $sp: frame pointer of C se
       ....
      ....
    lw      $ra,32($sp)     # $ra <- return address (in B)
    lw      $fp,28($sp)     # $fp  <- frame pointeur of B
    lw      $s0,24($sp)     # restore $s0
    addu    $sp,$sp,40      # $sp <- $sp+40, restore B stack po
    j       $ra             # return to $ra (B function)
```

Table of Contents

1. MIPS ISA
2. Function, procÃ©dure et Pile d'execution
3. Coming back to MIPS
4. Some additionnal useful information
   - Example of MIPS code

Procedure abstraction

- Let's pause a while to come back to high level langage
- What is a function (or a procedure)?
- How its isolation mecanisme (local variable) is implemented?
- This is implemented with a very fundamental mecanism: the Stack and the Activation Record (or Frame) of each procedure.

Notion of procedure

- Procedures (or functions) are the basic units for compilers
- Three important abstraction:
  - Control abstraction: parameter passing and result transmission
  - Memory abstraction: variable lifetime (local variables)
  - Interface: procedure's signature

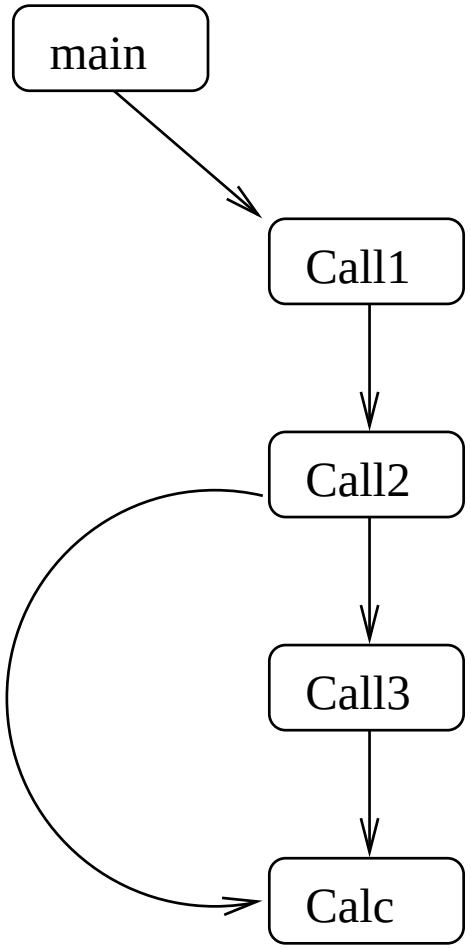
Procedure Control Transfer

- Transfer mechanism of control between procedures:
  - when calling a procedure, the control is given to the procedure called;
  - when this called procedure ends, the control is returned to the calling procedure.
  - Two calls to the same procedure create two em independent instances (or invocations).
- two useful graphic representations:
  - The call graph: represents the information written in the program.
  - The call tree: represents a particular execution.

Call Graph

```
procedure calc;
begin { calc}
    ...
end;
procedure call1;
    var y...
    procedure call2
        var z: ...
        procedure call3;
            var y....
            begin { call3}
                x:=...
                calc;
            end;
        begin { call2}
            z:=1;
            calc;
            call3;
        end;
    begin { call1}
        call2;
        ...
    end;
```

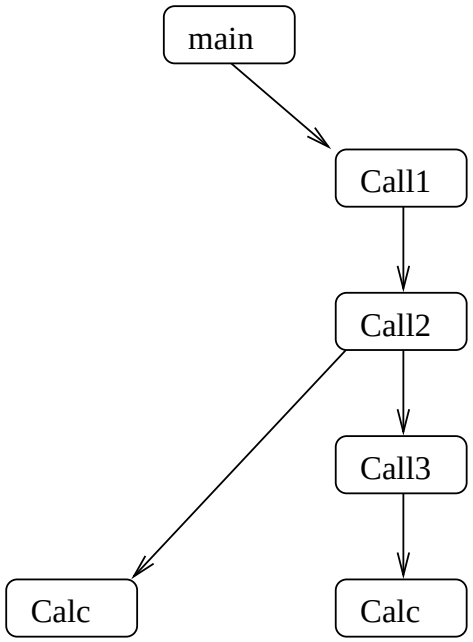
Call Graph:



Call Tree

```
procedure calc;
begin { calc}
    ...
end;
procedure call1;
    var y...
    procedure call2
        var z: ...
        procedure call3;
            var y....
            begin { call3}
                x:=...
                calc;
            end;
        begin { call2}
            z:=1;
            calc;
            call3;
        end;
    begin { call1}
        call2;
    end;
```

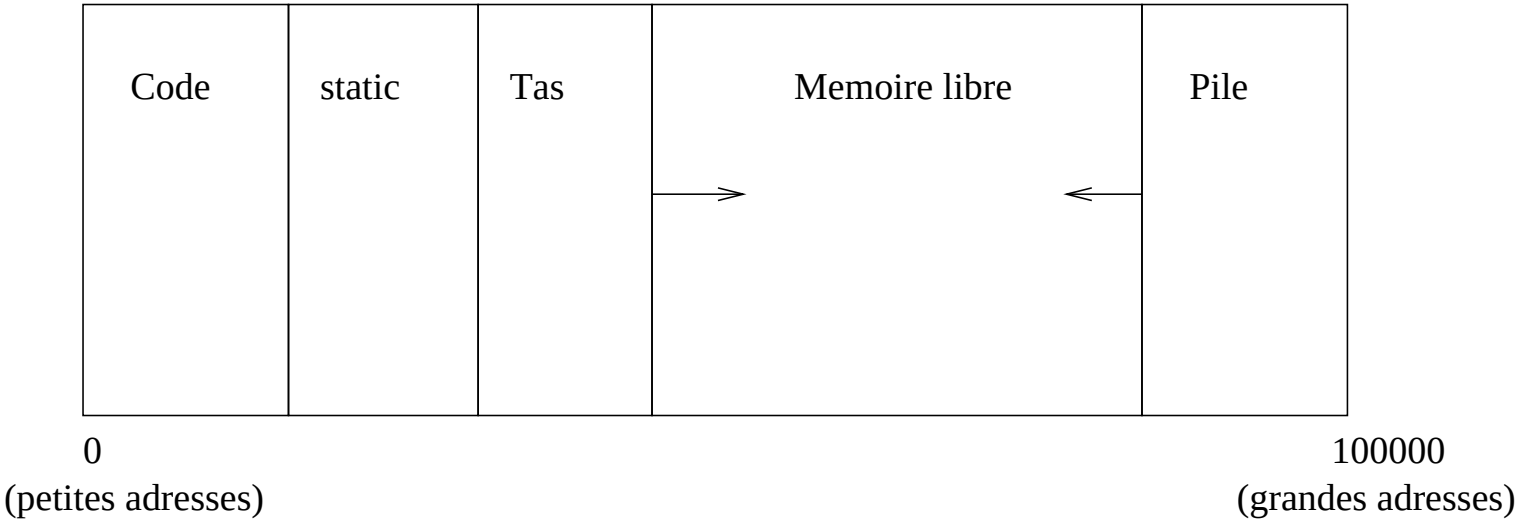
Call tree for one particular execution:



main calls $call_1$
$call_1$ calls $call_2$
$call_2$ calls *calc*
calc returns to $call_2$
$call_2$ calls $call_3$
$call_3$ calls*calc*
calc returns to $call_3$
$call_3$ returns to $call_2$
$call_2$ returns to $call_1$
$call_1$ returns to *main*

Execution Stack

- The transfer of control mechanism between procedures is implemented thanks to the *execution stack*.
- The programmer has this vision of virtual memory:

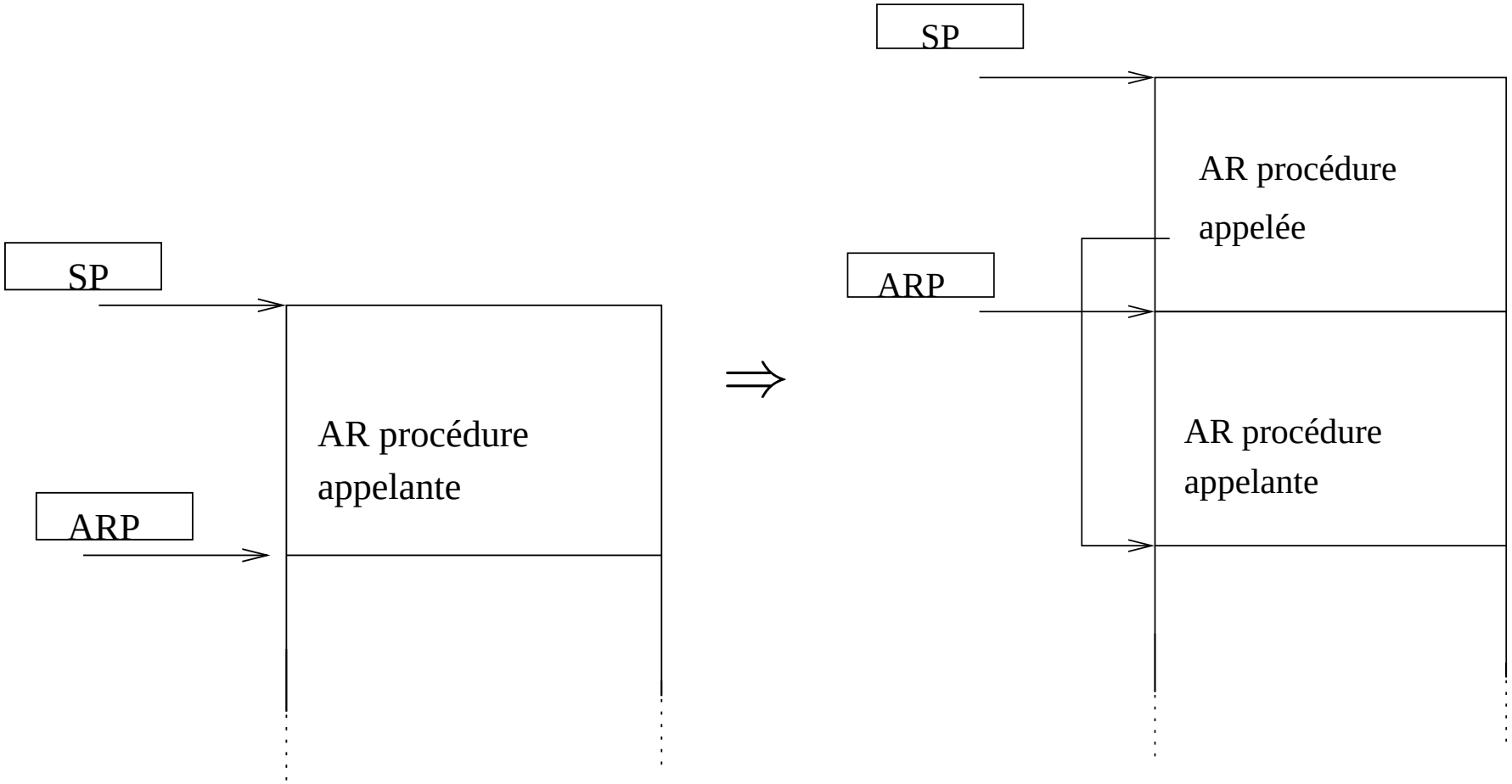


- The *heap* is used for dynamic allocation.
- The *stack* is used for the management of contexts of procedures (local variable, etc.)

Function call: status of the stack

Before the call
(AR=Activation Record)

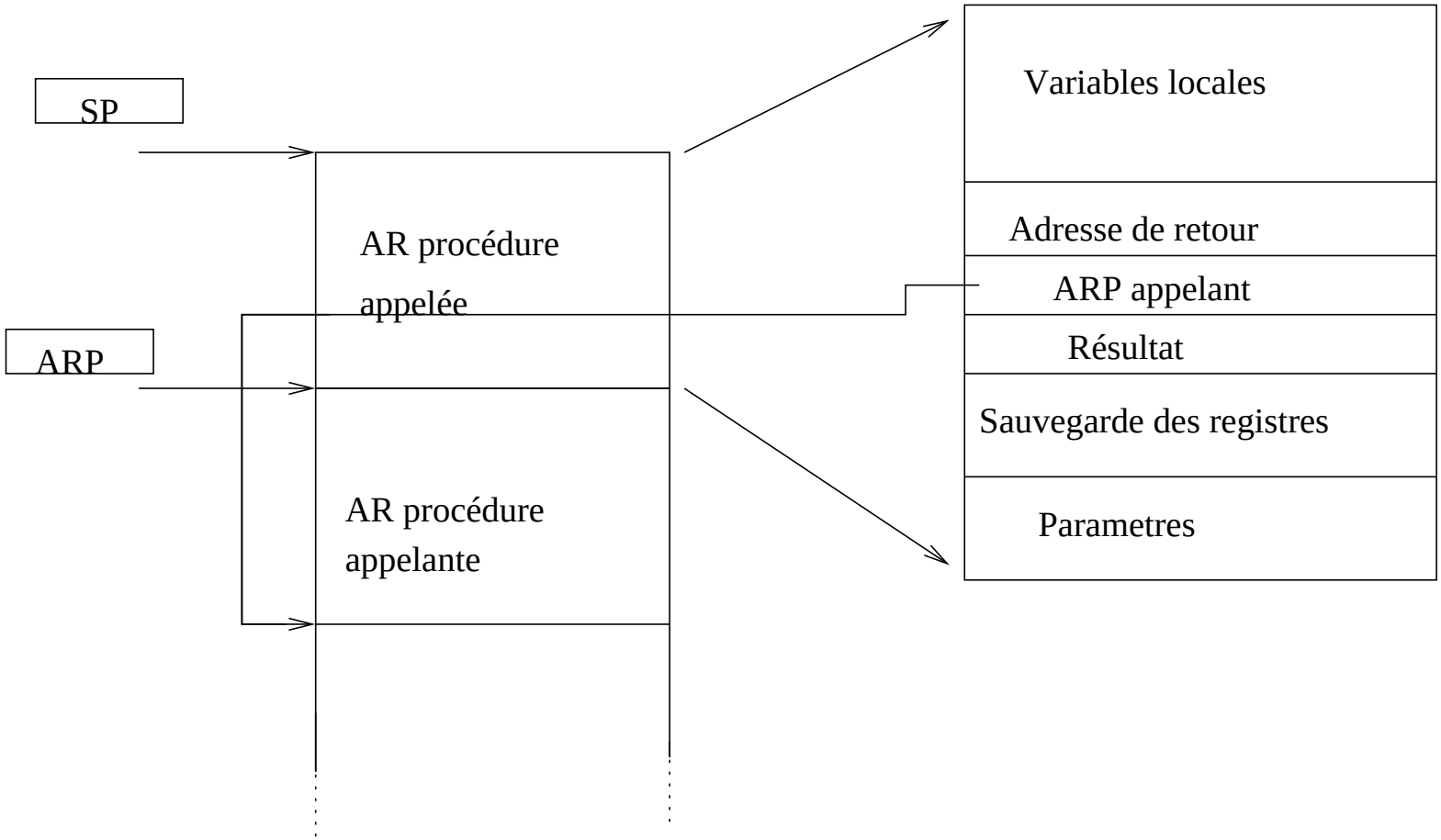
after the call



Activation record

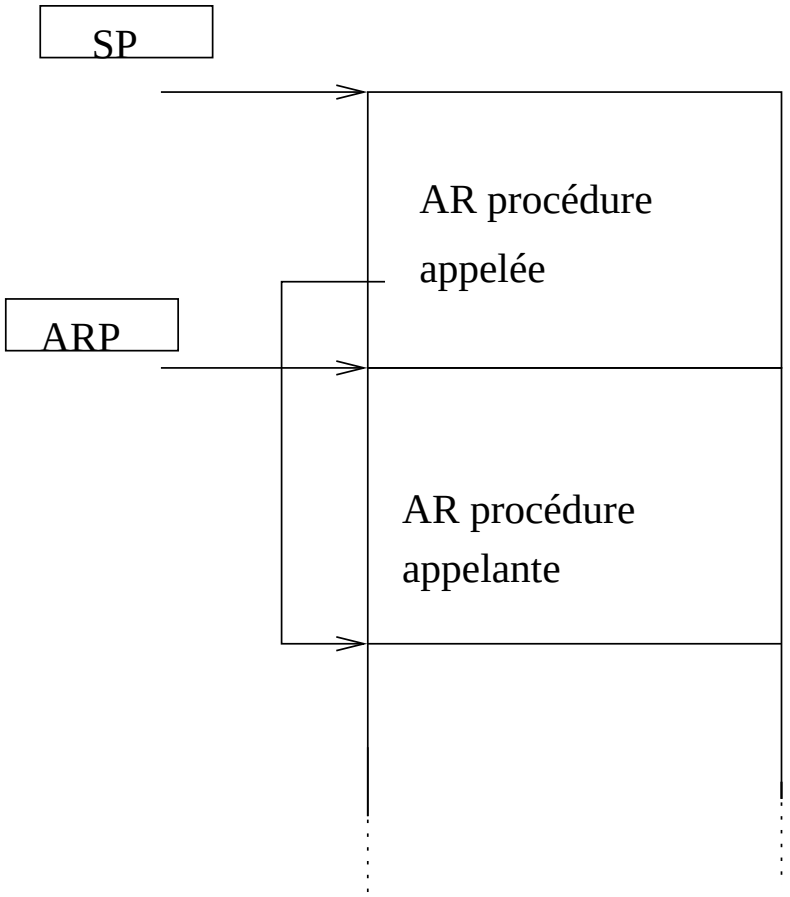
- Calling a procedure: Stacking the *activation record* (or *frame*).
- Need of a dedicated pointer for that: the *activation record pointer* (ARP) or *frame pointeur* (`$fp`))
- The frame allows to set up the *context* of the procedure.
- This frame contains
  - The space for local variables declared in the procedure
  - Information for restoring the context of the calling procedure:
    - Pointer to the frame of the calling procedure (ARP or FP for em frame pointer).
    - Address of the return instruction (statement following the call of the appellant proceedings).
    - Eventually save the state of the registers at the time of the call.

Content of the Frame



Return to calling function

avant le retour



$\Rightarrow$

aprÃ¨s le retour

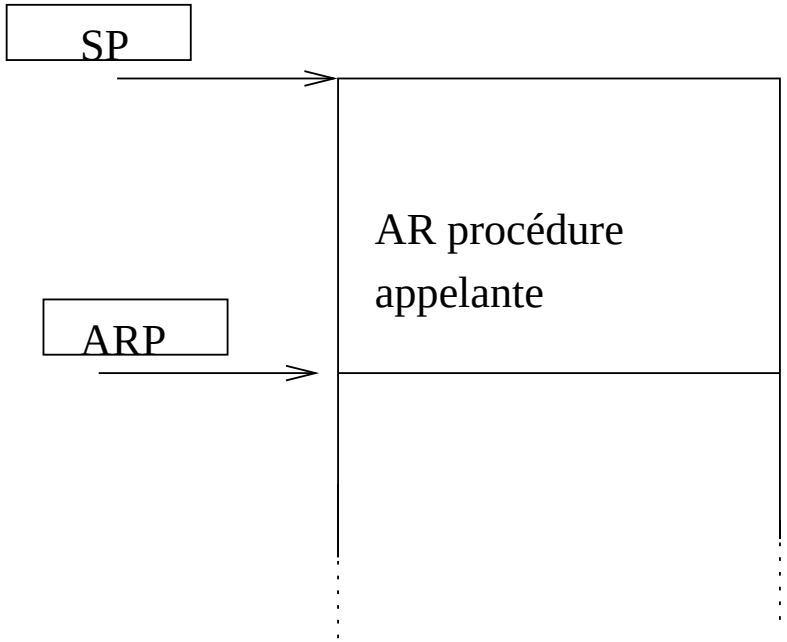


Table of Contents

1. MIPS ISA
2. Function, procÃ©dure et Pile d'execution
3. Coming back to MIPS
4. Some additionnal useful information
   - Example of MIPS code

Coming back to previous call example with B and C

- Let says: function B calls function C
- Function B wants to save `$t0`, `$t1` and `$a0` because it will need them after the return of C.
- this is done using the stack via the stack pointer `$sp`

The Stack

- The stack is use to store all *local* information (in the sense local to the current function)
- That includes (say for function C):
  - local variable
  - Callee saved register if needed
  - Return address (i.e. the instruction following the `jal C` instruction).
  - (sometimes) the parameters passed to C
  - (sometimes) the result of C
  - In many ISA, the parameters and the results are passed through dedicated registers
- All these data constitute the frame of the fonction instance.
- the frame pointeur points to the frame of the current function
- For MIPS, the frame pointer is `$fp`

Function B calls C

```
B   ...                 beguinning of  B
    ...
    sw $t0,0($sp)       saving  $t0 in stack
    sw $t1,-4($sp)      saving  $t1 in stack
    sw $a0,-8($sp)      saving  $a0 in stack
    sub $sp,$sp,12      correct stack pointer
    jal C               call to C function
    lw $a0,4($sp)       restoring return addresse of B from stac
    lw $t1,8($sp)       restoring $s1 from stack
    sw $t0,12($sp)      restoring  $s0
    add $sp,$sp,12      adjusst  stack pointeur value
    ...
    jr $ra              end of B
    ...
```

Sketching code of C function

```
C:
    subu    $sp,$sp,40      # C need 40 Bytes for its frame
    sw      $ra,32($sp)     # store return address (inst. in B)
    sw      $fp,28($sp)     # store frame pointer
    sw      $s0,24($sp)     # store $s0 (because C uses it)
    move    $fp,$sp         # $fp <- $sp: frame pointer of C se
       ....
      ....
    lw      $ra,32($sp)     # $ra <- return address (in B)
    lw      $fp,28($sp)     # $fp  <- frame pointeur of B
    lw      $s0,24($sp)     # restore $s0
    addu    $sp,$sp,40      # $sp <- $sp+40, restore B stack po
    j       $ra             # return to $ra (B function)
```

MIPS Assembly for programme fib

Fibbonacci suite program:

```
int fib (int i)
{
  if (i<=1) return(1);
  else return(fib(i-1)+fib(i-2));
}

int main (int argc, char *argv[])
{
  fib(2);
}
```

Assembleur MIPS pour programme fib

```
fib:
        .frame        $fp,40,$ra        # vars= 8, regs= 3/0, args= 16, extra= 0
        .mask        0xc0010000,-8
        .fmask        0x00000000,0
        subu        $sp,$sp,40         # SP <- SP-40 :AR de 40 octet (10 mots)
        sw        $ra,32($sp)          # stocke adresse retour SP+32
        sw        $fp,28($sp)          # stocke ARP appelant SP+28
        sw        $s0,24($sp)          # sauvegarde registre $s0
        move        $fp,$sp            # ARP <- SP
        sw        $a0,40($fp)          # stocke Arg1 dans la pile (ARP+40)
        lw        $v0,40($fp)          # charge Arg1 dans $v0
        slt        $v0,$v0,2           # $v0 <- 1 si $v0<2 0 sinon
        beq        $v0,$0,$L2          # branch L2 si $v0==0
        li        $v0,1                # $v0 <- 0x1 ($v0 sera le registre contenant le  res)
        sw        $v0,16($fp)          # stocke le resultat dans la pile
        j        $L1                   # saute Ã  L1
$L2:
        lw        $v0,40($fp)          # charge Arg1 dans $v0
        addu        $v0,$v0,-1         # retranche 1
        move        $a0,$v0            # $a0 <- $v0 ($a0 contient Arg1 pour l'appel recursif)
        jal        fib                  # jump and link fib ($ra<-next instr)
        move        $s0,$v0             # $s0 <- $v0 ($v0: res appel fib)
        lw        $v0,40($fp)           # charge Arg1 dans $v0
        addu        $v0,$v0,-2          # retranche 2
        move        $a0,$v0             # $a0 <- $v0 ($a0: contient Arg1 pour l'appel recursif)
        jal        fib                  # jump and link fib ($ra<-next instr)
        addu        $s0,$s0,$v0         # $s0 <- $s0+$v0 ($v0: res appel fib)
        sw        $s0,16($fp)           # stocke le resultat dans la pile
```

Assembleur MIPS pour programme fib

```
$L1:
        lw        $v0,16($fp)          # $v0 <- resultat
        move        $sp,$fp            # SP <- ARP
        lw        $ra,32($sp)          # $ra <- adresse retour
        lw        $fp,28($sp)          # ARP <- ARP appelant
        lw        $s0,24($sp)          # restaure $s0
        addu        $sp,$sp,40         # SP->SP+40
        j        $ra                   # jump adresse retour
        .end        fib
        .align        2
        .globl        main
        .ent        main
main:
        .frame        $fp,24,$ra       # vars= 0, regs= 2/0, args= 16, extra= 0
        .mask        0xc0000000,-4
        .fmask        0x00000000,0
# partie ajoutÃ©e pour afficher le resultat
.data
str: .asciiz  "Le resultat est "
.text
        subu        $sp,$sp,24      # SP <- SP-24 :AR de 24 octet (6 mots)
        sw        $ra,20($sp)       # stocke adresse retour SP+20
        sw        $fp,16($sp)       # stocke ARP appelant SP+16
        move        $fp,$sp         # ARP <- SP
        sw        $a0,24($fp)       # stocke Arg1 dans la pile (ARP+24)
        sw        $5,28($fp)        # stocke Arg2 dans la pile (ARP+48)
        li        $a0,2             # $a0 <- 2 ($a0: Arg1)
        jal        fib              # jump and link fib ($ra<-next instr)
# partie ajoutÃ©e pour afficher le resultat
        move $16,$2           # $16 <- resultat de l'appel a fib
        li $v0, 4          # $v0 <- code pour afficher une chaine (4)
```

Table of Contents

1 MIPS ISA

2 Function, procÃ©dure et Pile d'execution

3 Coming back to MIPS

4 Some additionnal useful information
- Example of MIPS code

Assember directives

.align n	Align the next datum on specified byte boundary (0=byte, 2=word, etc.).
.ascii str	store the string in memory, but do not null-terminate it.
.asciiz str	Store the string in memory and null-terminate it.
.byte b1,..., bn	Store the n values in successive bytes of memory.
.data <addr>	The following data items should be stored in the data segment
.double d1,..., dn	Store the n floating point double precision numbers in successive memory locations.
.extern sym size	Declare that the datum stored at sym is size bytes large and is a global symbol.
.globl sym	Declare that symbol sym is global and can be referenced from other files.
.space n	Allocate n bytes of space in the current segment.
.text <addr>	The next items are put in the user text segment.
.word w1,..., wn	Store the n 32-bit quantities in successive memory words.

example 1 (Fratini/Niebert)

```
    bne $s0, $s1, Test
    add $s2, $s0, $s1
Test:
```

example 2 (Fratini/Niebert)

```
    beq $s4, $s5, Lab1
    add $s6, $s4, $s5
    j Lab2
Lab1:sub $s6, $s4, $s5
Lab2:
```

example 3 (Fratini/Niebert)

```
    li $t2, 0
    li $t3, 1
while:beq $t1, $0, done
    add $t2, $t1, $t2
    sub $t1, $t1, $t3
    j while
done:
```

example 4 (U. Illinois)

```
        .data
var1:   .word  23          # declare storage for var1; initial
                           # value is 23

        .text
__start:
        lw $t0, var1         # load contents of RAM location in
                             # register $t0:  $t0 = var1
        li $t1, 5            #  $t1 = 5   ("load immediate")
        sw $t1, var1         # store contents of register $t1
                             #into RAM:  var1 = $t1
done
```

example 5 (U. Illinois)

```
        .data
array1: .space 12          #  declare 12 bytes of storage to
                           # hold array of 3 integers
        .text
__start: la $t0, array1          #  load base address of array
                                 #register $t0
        li $t1, 5                #$t1 = 5   ("load immediate")
        sw $t1, ($t0)            #first array element set to 5;
                                 #indirect addressing
        li $t1, 13               #$t1 = 13
        sw $t1, 4($t0)           #second array element set to 1
        li $t1, -7               #$t1 = -7
        sw $t1, 8($t0)           #third array element set to -7
done
```

Documentation on MIPS assembly

More precise documentation on MIPS assembly code can be obtained at:

- http://igm.univ-mlv.fr/ens/IR/IR1/2007-2008/Archi/ManuelSPIM.php (brief documentation from U. Marne la vallÃ©e)
- http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm (brief documentation from U. of illinois at Chicago).
- https://en.wikibooks.org/wiki/MIPS_Assembly, wikibook
- https://www.cs.unibo.it/~solmi/teaching/arch_2002-2003/AssemblyLanguageProgDoc.pdf, MIPS Assembly langage programmer's Guide.