

ARC: Computer Architecture

tanguy.risset@insa-lyon.fr

Lab CITI, INSA de Lyon

Version du March 30, 2026

Tanguy Risset

March 30, 2026

Table of Contents

- 1 Instruction Set Architecture (ISA, *assembleur* in French)
- 2 The RISC-V ISA example
- 3 Function, procédure et Pile d'exécution
- 4 Coming back to RISC-V
- 5 Pipelining RISC instructions: the "Von Neumann" cycle

Set Architecture Statement (ISA)

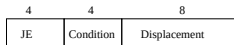
- The *instruction set* (Set Architecture statement: ISA) is of paramount importance
 - It determines the basic instructions executed by the CPU.
 - It's a balance between the hardware complexity of the CPU and the ability to express the required actions
 - It is represented in a symbolic way: the assembly code/language (ex: ADD R1,R2)
 - The tool that translates symbolic assembly code in binary code (i.e. machine code) is also called the **assembler**
- Two types of ISA:
 - CISC: Complex Instruction Set Computer
 - RISC: Reduce Instruction Set Computer

CISC: Complex Instruction Set Computer

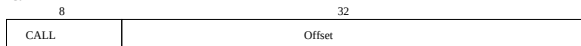
- An instruction can code several elementary operations
 - Ex: a load, an add and a store (in memory operations)
 - Ex: compute a linear interpolation of several values in memory
- Need a more complex hardware (specifically hardware accelerators)
- High variability in size and execution time for different instructions
- Produce a more compact code but more complex to generate
- Vax, Motorola 68000, Intel x86/Pentium

Example: instructions ISA of Pentium

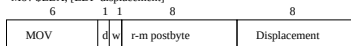
JE EIP + displacement



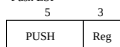
Call



Mov \$EBX, [EDI+displacement]



Push ESI



Add \$EAX, Immediate



Test \$EDX, Immediate



RISC: Reduced Instruction Set Computer

- Small simple instructions, all having the same size, and (almost) the same execution time.
- no complex instruction
- Clock speed increase with pipelining (between 3 and 7 pipeline stages)
- Code simpler to generate but less compact
- Every modern processor use this paradigm: SPARC, MIPS, ARM, PowerPC, etc.

Example: instructions of MSP430 ISA

1 operand instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	opcode			B/W	Ad		Dest reg.			

relative Jumps

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	condition			PC offset (10 bits)									

2 operands instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode			Dest reg.				Ad	B/W	As			Dest reg.			

Examples:

- PUSH.B R4
- JNE -56
- ADD.W R4,R4

Exemple of Pentium ISA

- Write a simple C program `toto.c`
- Type `gcc -S toto.c` and get the `toto.s` file
- you can also use the *compiler explorer*:
<https://gcc.godbolt.org/>
-

```
main() {
    int i=17;
    i=i+42;
    printf("%d\n", i);
}
```

⇒

```
(... instructions ...)
movl$17, -4(%rbp)
addl$42, -4(%rbp)
(... printf params... )
callprintf
(... instructions ...)
```


Disassembly

- compile the assembly code: `gcc toto.s -o toto`
- disassemble with `objdump`:
`objdump -d toto`

Adresses	Instructions binaires	Assembleur
(...)		
40052c:55		push %rbp
40052d:48 89 e5		mov %rsp,%rbp
400530:48 83 ec 10		sub \$0x10,%rsp
400534:c7 45 fc 11 00 00 00		movl \$0x11,-0x4(%rbp)
40053b:83 45 fc 2a		addl \$0x2a,-0x4(%rbp)
40053f:8b 45 fc		mov -0x4(%rbp),%eax
400542:89 c6		mov %eax,%esi
% (...)		

common properties of ISA

- An ISA first defines the **types of data** on which the processors can compute (32 bit memory addresses, integer of various sizes, etc.)
- Then it contains various **types of instructions**:
 - **Computation** instructions (add, sub, or, and, ...), with various **number of operands**
 - **Memory addressing** instructions (load, store)
 - **stack management** instructions (push, pop)
 - **Flow control** instructions (jumps)
 - **subroutine calls**

Table of Contents

- 1 Instruction Set Architecture (ISA, *assembleur* in French)
- 2 The RISC-V ISA example
- 3 Function, procédure et Pile d'exécution
- 4 Coming back to RISC-V
- 5 Pipelining RISC instructions: the "Von Neumann" cycle

RISC-V history

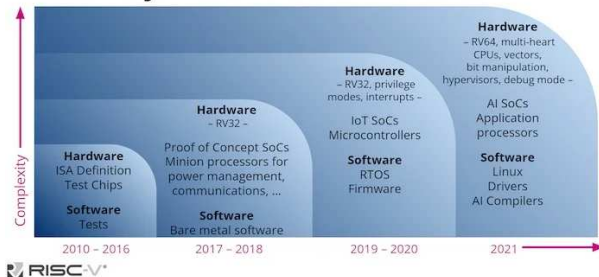
- The RISC paradigm was invented 1980 [David Patterson](#) (UC Berkeley) and [John Hennessy](#) (Stanford U.).
- They described the [MIPS](#) architecture in their books “Computer Organization and Design” and “Computer Architecture: A Quantitative Approach.”
- The MIPS was built by a commercial company (MIPS was in Nintendo 64, Sony PlayStation, PlayStation 2) and used in many architecture courses (including 3TC-ARC!).
- Hennessy and Patterson received the ACM A.M. Turing Award in 2017 (<https://amturing.acm.org/byyear.cfm>).

From 1980 to 2010, the development of the fifth generation of the RISC research project started and led to the [RISC-V](#) (pronounced “risk-five”).

RISC-V project

- RISC-V is an open instruction set architecture (ISA), this is fairly new!
- RISC-V International is a global nonprofit organization that owns and maintains the RISC-V ISA intellectual property.
- Its members range from individuals to organizations like Google, Intel, and Nvidia.

Industry innovation on RISC-V



(from <https://www.allaboutcircuits.com/>)

RISC-V Processor

- RISC-V is specified by its ISA (RV32I, for integer 32 bits for instance).
- Many extension of the ISA are specified (32 bits, 64 bits, 128 bits, Atomic Instructions, Compressed Instructions, etc.)
- The architecture can be pipelined or not, it can target small embedded systems or large powerfull machines.
- Each processor compagny can build it own RISC-V implementation as long as it respect the ISA specification.
- We will study the RV32I base integer ISA that implements the necessary operations to achieve basic functionality with 32-bit integers.

RISC-V ISA basics (RV32)

- a *register-to-register* (or *load/store*) architecture
- RISC-V use 3-address instructions (destination is the first operand)
- 32 32-bits registers (x0-x31) plus a A program counter (pc) of
- register can be name x0, x1, etc. or with their more explicit ABI names: x0 is “zero”, x1 is ra (return address), etc (<https://en.wikichip.org/wiki/risc-v/registers>)
- x0 is hardwired to value 0
- x1 is the return address (ra)
- x2 is the stack pointer (sp)
- 32-bit address space: Addressable memory of 2^{32} bytes
 - $\Leftrightarrow 2^{30}$ words of 4 bytes

understanding RISC assembly

- From C to assembly:

```
riscv64-linux-gnu-gcc -S NtimesN.c -o NtimesN.S
```

NtimesN.c

```
[...]
scanf("%d",&N);
i = N*N + 3*N;
printf("i=%d\n",i);
[...]
```

NtimesN.s

```
[...]
call    __isoc99_scanf@plt #call to scanf
lw      a5,4(sp)           #N is now in a5
mulw    a2,a5,a5           #a2 <- N*N
slliw   a4,a5,1            #a4 <- 2*N (shift left by 1)
addw    a5,a4,a5           #a4 <- 2*N+N
addw    a2,a2,a5           #a5 <- N*N + 3*N
lla     a1,.LC1            #printf args (To be explained later)
li      a0,1               #.. explained later
call    __printf_chk@plt   #call to printf
[...]
```


RISC-V register

- 32 registers in the *register file*
- Named
 - by their number: x0 x1 ...x31
 - or by their name zero ra sp fp a0 a1 ...a7 s1 s2 ...
- x0 (zero) contains value 0
- a0 ...a7 are used to pass **arguments** of a function call
- a0 a1 are used to transmit functions **result**
- t0 ...t6 and s0 ...s11 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
- tp is the thread pointer

Risc-V assembly addressing mode

- The addressing mode defines how the operands of each instruction are interpreted.
- RISC-V has four addressing modes:
 - **Immediate addressing**: the operand is a constant within the instruction
 - **Register addressing**: where the operand represents a register.
 - **Base addressing**: the operand is an address which is the sum of a register and a constant (sometimes called indirect addressing)
 - **PC-relative addressing**: the operand is an address which is the sum of PC and a constant

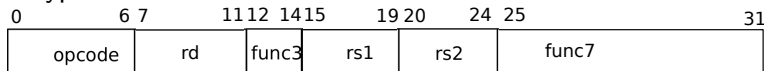
Example of RISC-V addressing mode

- Register addressing
 - `add x1, x2, x3`
 - puts in x1 the value of x2 plus the value of x3.
 - $x1 = x2 + x3$
- Immediate addressing
 - `addi x1, x2, 0x0f`
 - `addi x1, x2, 15`
 - puts in x1 the value of x2 plus 15.
 - $x1 = x2 + 15$
- Base addressing
 - `lw x1, 12(x3)`
 - puts in x1 the value situated in memory at the address obtained by adding 12 to the content of x3.
 - $x1 = \text{Memory}[x3 + 12]$
- `bne a1, a2, label`
 - branch to address of label if values in a1 and a2 are different.
 - if $(a1 \neq a2)$ then $\$PC = \text{label}$

Format of Risc-V instructions

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

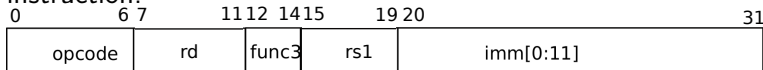
- R-types:



- Used for 3-registers instructions
- opcode is the operation code that specifies the operation
- rs1 and rs2 are the first and second source register
- rd is the destination register
- func3 and func7 are used with op to select arithmetic operation (additional opcode fields)

I-Types instruction

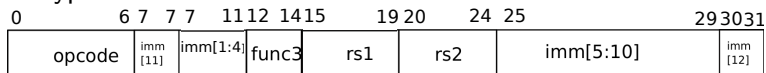
- I-Types instruction are used for load, store, branch and immediate instruction.



- rs1 is a source register
- rd is a destination register
- func3 is additionnal opcode field
- The imm 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)

B-Types instruction

- B-Types instruction are used for Branch instructions



- The `imm` split-field is a 13 bit's integer containing an address (always even hence bit 0 is implicitly 0).
- can jump from address 0 to $2^{13}=1\text{MB}$ from `$PC`.
- For longer jump, one can use other instructions (PC absolute), barely used.

Basic arithmetic and logic instruction

- R-Types instructions: add, sub and, or, xor
 - `add rd, rs1, rs2` `// rd = rs1 + rs2`
 - `xor rd, rs1, rs2` `// rd = rs1 ^ rs2`
- I-types for immediate operand operation:
 - `addi rd, rs1, 4` `// rd = rs1 + 4`
 - `li rd, 4` `// rd = 4, pseudo (addi rd, zero, 4)`

Load and store

- 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 rd, 20(rs1)` // `rd=Memory[rs1+20][0:7]`
 - `lw rd, 20(rs1)` // `rd=Memory[rs1+20](i.e.[0:31])`
 - `sb rd, 20(rs1)` // `Memory[rd+20][0:7]=rs1`
 - `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 rs1, rs2, Label`
 - if `rs1` and `rs2` have different values, the next instruction to execute is at address `Label` (i.e. `pc = Label`)
 - `beq rs1, rs2, Label` // same thing if `rs1=rs2`
- Unconditionnal branch
 - `j offset` // next instruction executed is at address `PC+offset`
 - `jr rs1` // next instruction executed is at address contained in `rs1`
- These are the only way of implementing loops in assembly:

```
[...]
        li s2, 1
while:  beq s1, zero, done
        sub s1, s1, s2
        j while
done:
```

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

Function control flow in RISC-V

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


```
jal ra, fact
```

 - saves the return address (i.e. the address of the following instruction) in the `ra` register and jumps to the label `fact` (code of `fact` function)
 - At the end of the execution of `fact`, the instruction `ret` jumps back to the address stored in `ra` (pseudo: `jalr x0, ra, 0`)
 - Arguments transmitted to `Fact` are stored in registers `a0 ... a7`
 - Return values of `Fact` are stored in registers `a0, a1`

Who save the register during Function call?

- When a function call occurs: `jal ra, 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 convention* or *ABI application binary interface*.
- For MIPS:
 - `ra, t0 ... t6, a0 ... a7`, are caller saved
 - `fp, s1 ... s11` are callee saved

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      ...      beginning 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 ra, C        call to C function
      lw a0,4(sp)      restoring return adresse of B from stack
      lw t1,8(sp)      restoring s1 from stack
      sw t0,12(sp)     restoring  s0
      add sp,sp,12     adjusst  stack pointeur value
      ...
      ret              end of B
      ...

```

Sketching code of C function

C:

```

addi    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 set
    ....
    ....
lw      ra,32(sp)      # ra <- return address (in B)
lw      fp,28(sp)      # fp <- frame pointeur of B
lw      s0,24(sp)      # restore s0
addi    sp,sp,40       # sp <- sp+40, restore B stack pointer
ret     # return to ra (B function)

```

Table of Contents

- 1 Instruction Set Architecture (ISA, *assembleur* in French)
- 2 The RISC-V ISA example
- 3 Function, procédure et Pile d'exécution
- 4 Coming back to RISC-V
- 5 Pipelining RISC instructions: the "Von Neumann" cycle

Procedure abstraction

- Let's pause a while to come back to high level language
- What is a function (or a procedure)?
- How its isolation mechanism (local variable) is implemented?
- This is implemented with a very fundamental mechanism: [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 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.

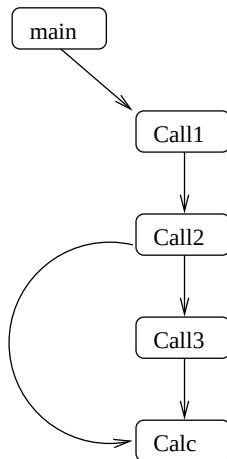
Call Graph

```

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:



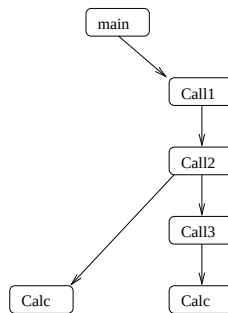
Call Tree

```

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;
end;

```

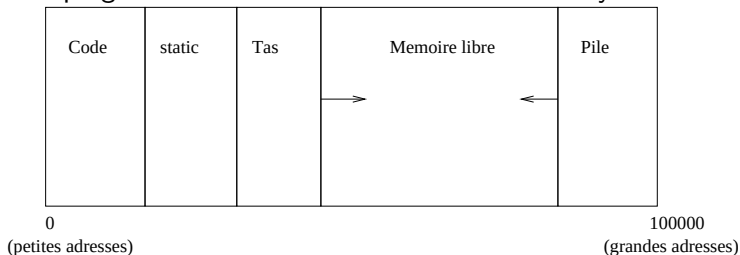
Call tree for one particular execution:



main calls *call₁*
call₁ calls *call₂*
call₂ calls *calc*
calc returns to *call₂*
call₂ calls *call₃*
call₃ calls *calc*
calc returns to *call₃*
call₃ returns to *call₂*
call₂ returns to *call₁*
call₁ returns to *main*

Execution Stack

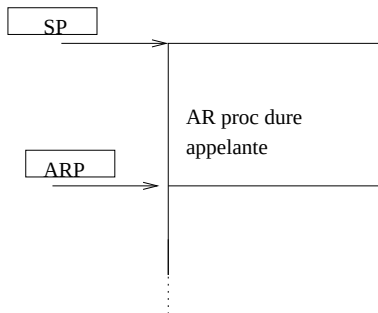
- The transfer of control mechanism between procedures is implemented thanks to the *execution stack*.
- The programmer has this vision of virtual memory:



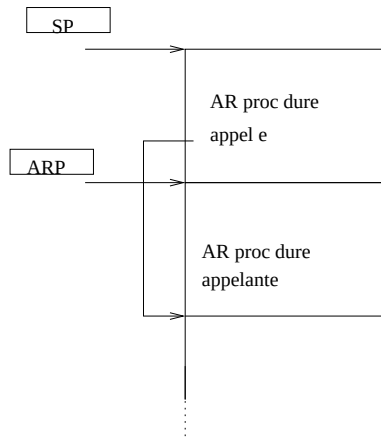
- The *heap* is used for dynamic allocation.
- The *stack* is used for the management of contexts of procedures (local variable, etc.)

Function call: status of the stack

Before the call
(AR=Activation Record)



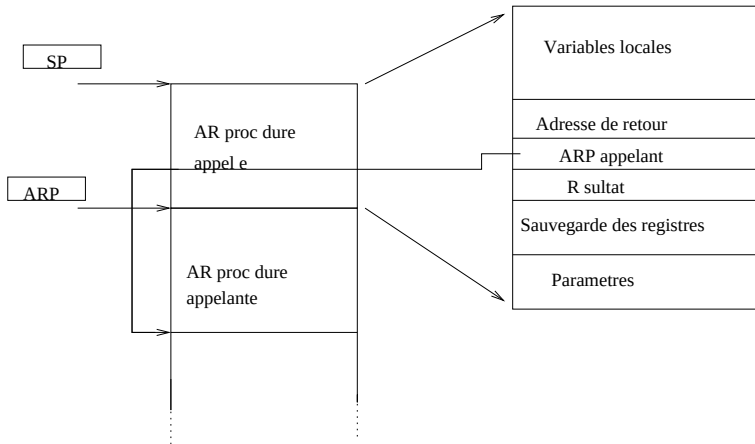
after the call



Activation record

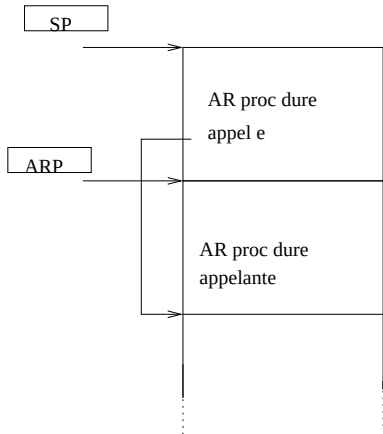
- Calling a procedure: Stacking the *activation record* (or *frame*).
- Need of a dedicated pointer for that: the *activation record pointer* (ARP) or *frame pointer* (fp))
- The frame allows to set up the *context* of the procedure.
- This frame contains
 - The space for local variables declared in the procedure
 - Information for restoring the context of the calling procedure:
 - Pointer to the frame of the calling procedure (ARP or FP for the 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



⇒

après le retour

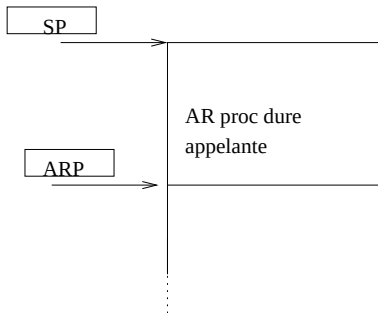


Table of Contents

- 1 Instruction Set Architecture (ISA, *assembleur* in French)
- 2 The RISC-V ISA example
- 3 Function, procédure et Pile d'exécution
- 4 Coming back to RISC-V
- 5 Pipelining RISC instructions: the "Von Neumann" cycle

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 used 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
 - (sometimes) Return address (i.e. the instruction following the `jal ra, 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 function instance.
- the **frame pointer** points to the frame of the current function
- For RISC-V, the frame pointer is `fp`

Function B calls C

```

B      ...      beginning 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 ra, C        call to C function
      lw a0,4(sp)      restoring return address of B from stack
      lw t1,8(sp)      restoring s1 from stack
      sw t0,12(sp)     restoring  s0
      add sp,sp,12     adjust stack pointer value
      ...
      ret              end of B
      ...

```

Sketching code of C function

C:

```

addi    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 set
    ....
    ....
lw      ra,32(sp)      # ra <- return address (in B)
lw      fp,28(sp)      # fp <- frame pointeur of B
lw      s0,24(sp)      # restore s0
addi    sp,sp,40       # sp <- sp+40, restore B stack pointer
ret      # return to ra (B function)

```

RISC-V Assembly for programme fib

Fibonacci 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 RISC-V pour programme fib

```

fib:
    addi    sp,sp,-48    # SP <- SP-48 :AR de 48 octet (12 mots)
    sd      ra,40(sp)    # stocke adresse retour (64 bits) a SP+40
    sd      s0,32(sp)    # sauvegarde registre s0
    sd      s1,24(sp)    # sauvegarde registre s1
    addi    s0,sp,48     # s0=ARP/FP <- SP
    mv      a5,a0        # a5 <- arg1 (N)
    sw      a5,-36(s0)    # stock arg1 (N) dans la pile (SP-12)
    lw      a5,-36(s0)    # instruction inutile (supprimée si optimisation)
    sext.w  a4,a5        # a4 <- sign extension a5(32)
    li      a5,1         # a5 <- 1
    bgt     a4,a5,.L2    # if (a4 > 1) sauter a .L2
    li      a5,1         # ici on a arg1=N<=1 donc a5 <- res=1
    j       .L3          # sauter à .L3
.L2:
    lw      a5,-36(s0)    # ici N>1, a5 <- N
    addiw   a5,a5,-1      # a5 <- a5 - 1
    sext.w  a5,a5        # sign extension
    mv      a0,a5        # a0 <- a5 (set arg in a0 for recursive call)
    call    fib          # recursive call
    mv      a5,a0        # a5 <- result from recursive call
    mv      s1,a5        # s1 <- a5
    lw      a5,-36(s0)    # a5 <- N
    addiw   a5,a5,-2      # a5 <- a5 -2
    sext.w  a5,a5        # sign extension
    mv      a0,a5        # a0 <- a5 (set arg in a0 for recursive call)
    call    fib          # recursive call
    mv      a5,a0        # a5 <- result from recursive call
    addw    a5,s1,a5      # a5 <- fib(N-1)+fib(N-1)
    sext.w  a5,a5        # sign extension

```

Assembleur RISC-V pour programme fib

```
.L3:
    mv      a0,a5      #a0 <- a5 (set result in a0)
    ld      ra,40(sp)   # restaure ra
    ld      s0,32(sp)   # restaure s0
    ld      s1,24(sp)   # restaure s1
    addi    sp,sp,48    # restaure sp
    jr      ra          # return
    .size   fib, .-fib
    .section .rodata
    .align  3
.LC0:
    .string "le resultat est %d "
    .text
    .align  2
    .globl  main
    .type   main, @function
main:
    addi    sp,sp,-32   # set AR for main
    sd      ra,24(sp)
    sd      s0,16(sp)
    addi    s0,sp,32
    mv      a5,a0       #store arg og main
    sd      a1,-32(s0)
    sw      a5,-20(s0)
    li      a0,2        # we call fig(2)
    call    fib
    mv      a5,a0       # get fib result
    mv      a1,a5       #set args for printf
    lla     a0,.LC0
    call    printf@plt
    li      a5,0
```

example 1, if-then

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

Test:

example 2, if-then-else

```
    beq s4, s5, Lab1
```

```
    add s6, s4, s5
```

```
    j Lab2
```

```
Lab1: sub s6, s4, s5
```

```
Lab2:
```

example 3, looping

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

example 4, static variable

```

.globl main
.type main, @function
    .data
var1:    .word    23           # declare storage for var1; initial
                                # value is 23

    .text
main:
    lw t0, var1               # load contents of RAM location into
                                # register $t0:  t0 = [var1] (= 23)
    li t1, 5                  # $t1 = 5    ("load immediate")
    la t2, var1               # load address of var1
    sw t1, (t2)               # store contents of register t1
                                # into RAM:  [var1] = t1 = 5

done:
    ir    ra

```

example 5, array accesses

```

        .globl main
        .type main, @function
        .data

array1: .space 12          # declare 12 bytes of storage to
                           # hold array of 3 integers

        .text
main:   la t0, array1      # load base address of array into
                           # 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 13

        li t1, -7         #t1 = -7
        sw t1, 8(t0)       #third array element set to -7

        ir  ra

```

Documentation on RISC-V assembly

- The RISC-V Instruction Set Manual Volume I: User-Level ISA

<https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>

- Risc-V assembly manual on github

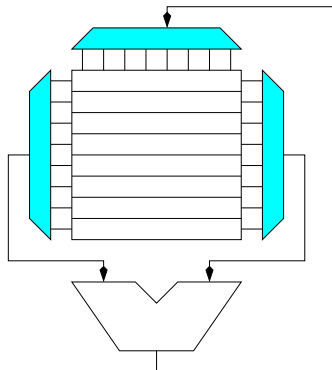
<https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md>

- CheatSheet on ARC Moodle site.

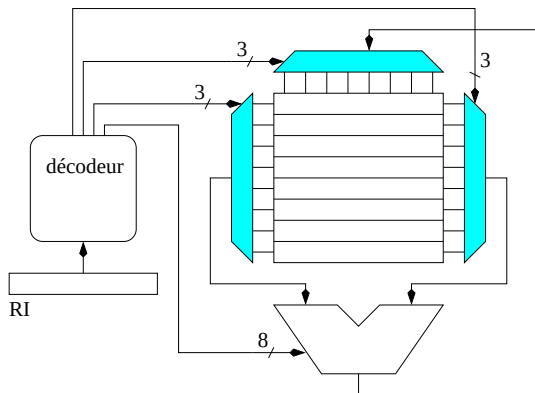
Table of Contents

- 1 Instruction Set Architecture (ISA, *assembleur* in French)
- 2 The RISC-V ISA example
- 3 Function, procédure et Pile d'exécution
- 4 Coming back to RISC-V
- 5 **Pipelining RISC instructions: the “Von Neumann” cycle**

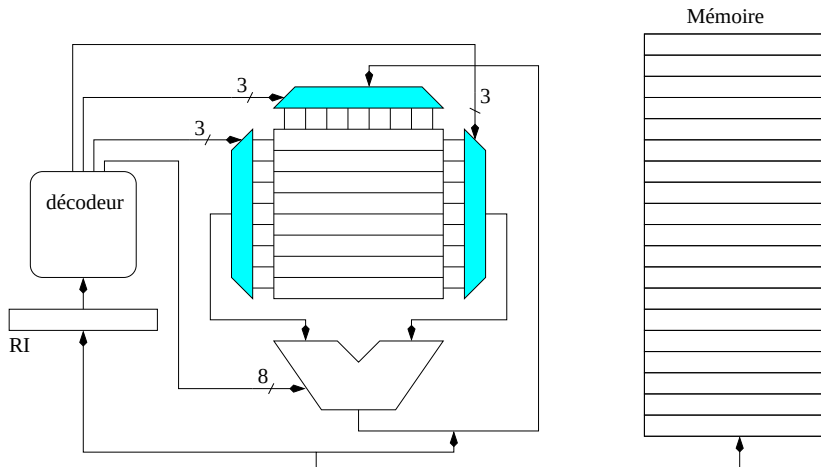
Program execution on a Processor (8 general purpose registers)



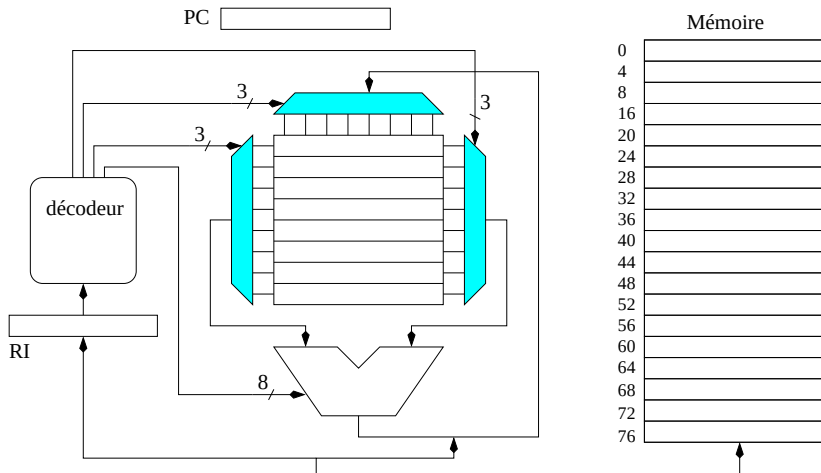
Program execution on a Processor (8 general purpose registers)



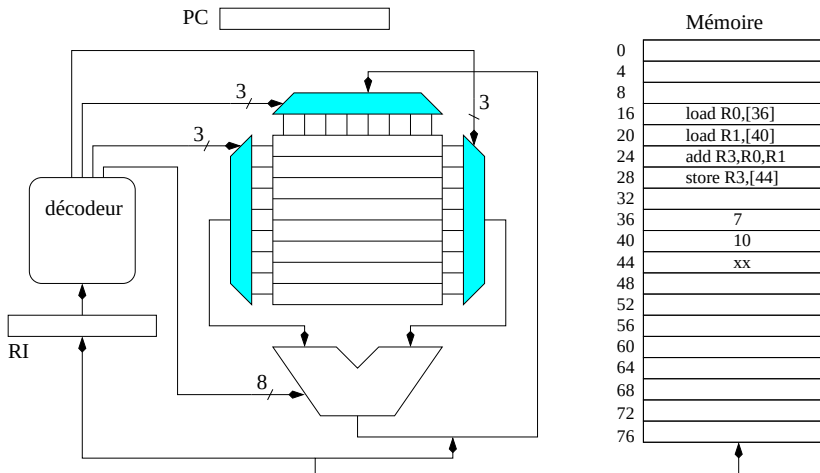
Program execution on a Processor (8 general purpose registers)



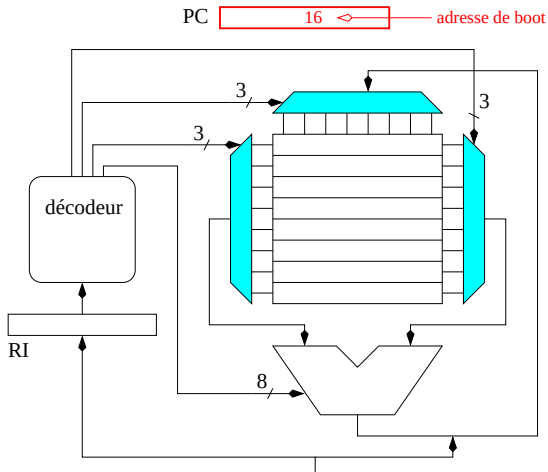
Program execution on a Processor (8 general purpose registers)



Program execution on a Processor (8 general purpose registers)



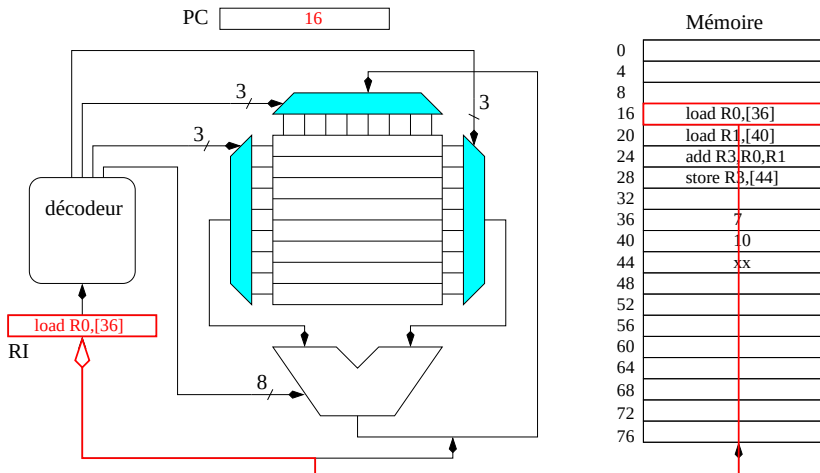
Program execution on a Processor (8 general purpose registers)



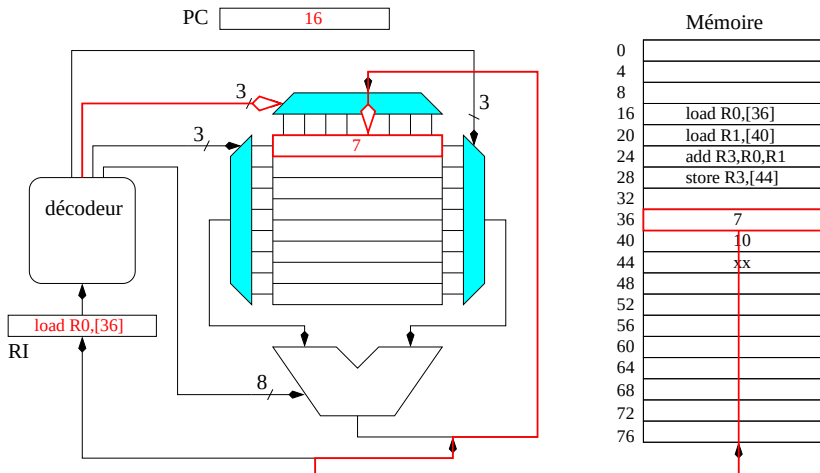
Mémoire

0	
4	
8	
16	load R0,[36]
20	load R1,[40]
24	add R3,R0,R1
28	store R3,[44]
32	
36	7
40	10
44	xx
48	
52	
56	
60	
64	
68	
72	
76	

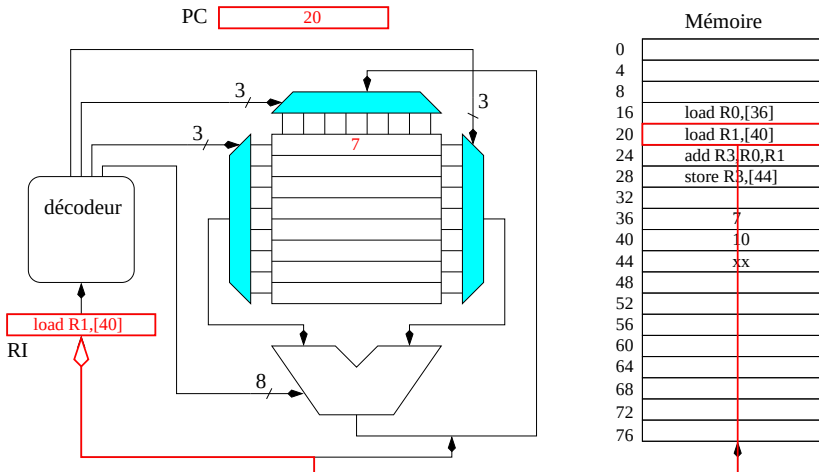
Program execution on a Processor (8 general purpose registers)



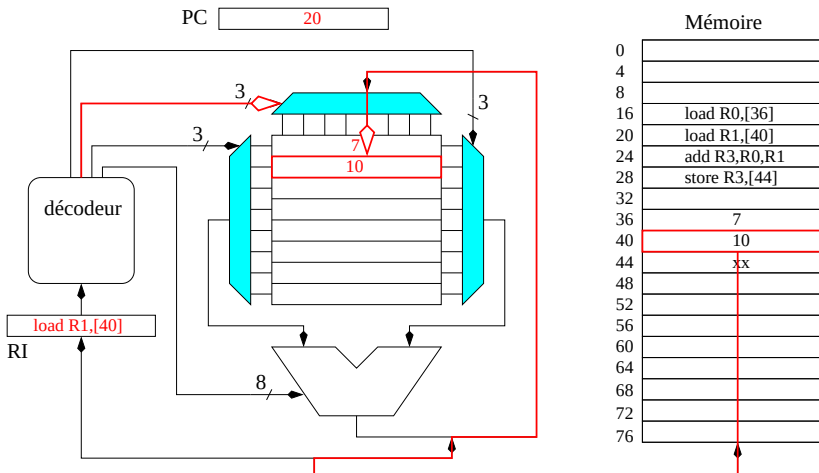
Program execution on a Processor (8 general purpose registers)



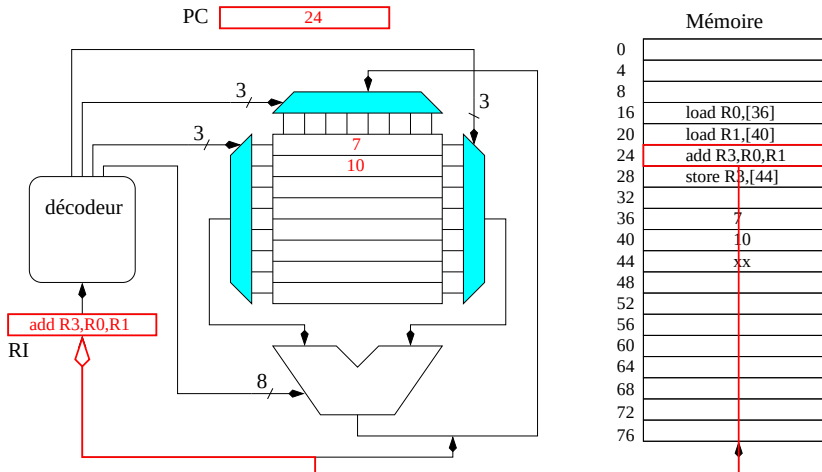
Program execution on a Processor (8 general purpose registers)



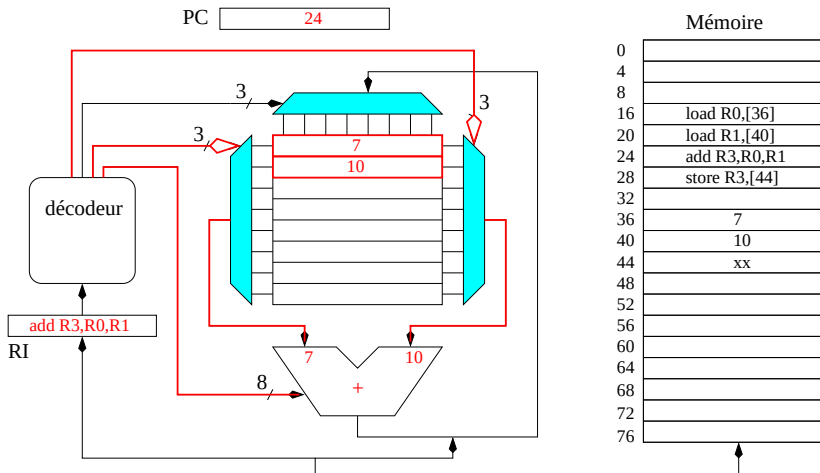
Program execution on a Processor (8 general purpose registers)



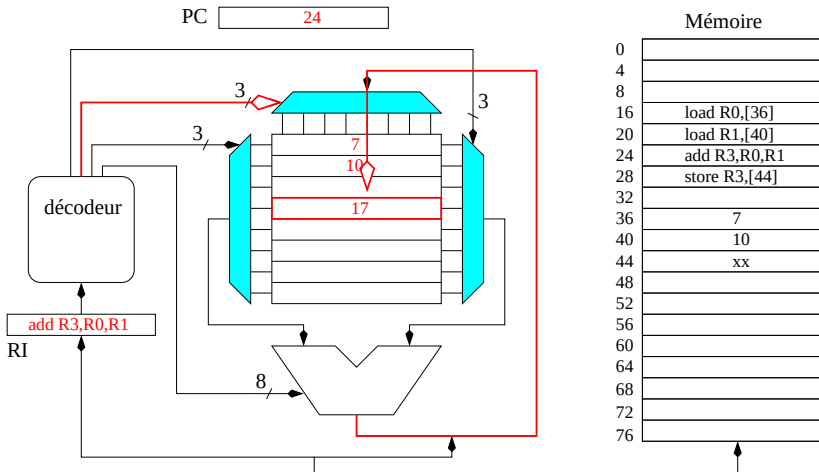
Program execution on a Processor (8 general purpose registers)



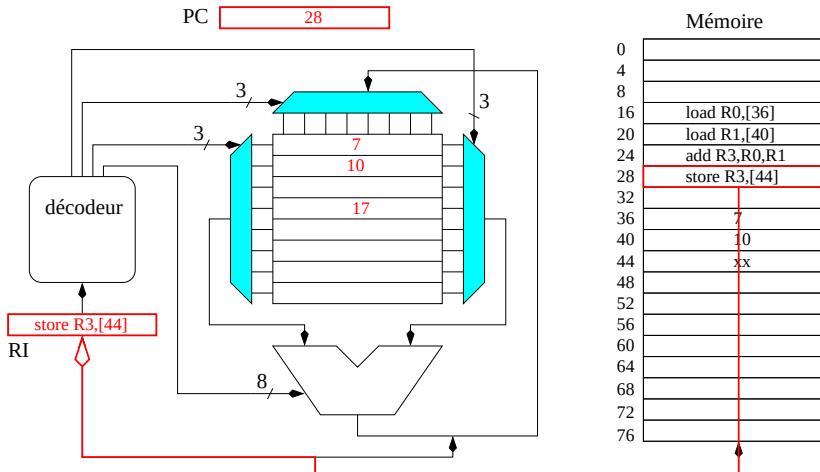
Program execution on a Processor (8 general purpose registers)



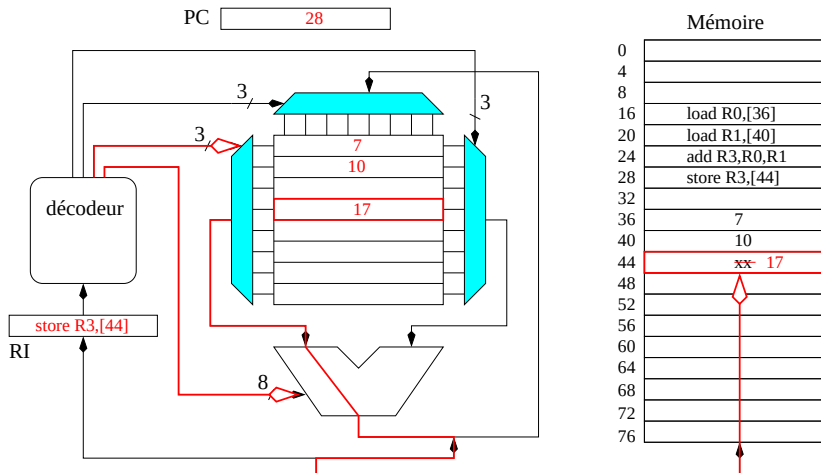
Program execution on a Processor (8 general purpose registers)



Program execution on a Processor (8 general purpose registers)



Program execution on a Processor (8 general purpose registers)

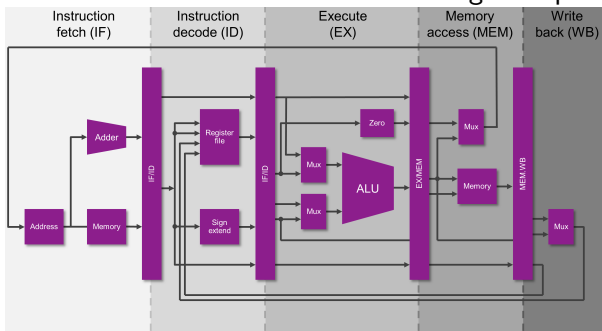


The “Von Neumann cycle”

- The so-called Von Neumann cycle is simply the **decomposition** of the execution of an instruction in **several independent stages**.
- The number of stages depend on the processor, usually 5 stages are commonly used as example:
 - **Instruction Fetch (IF)**
 - Reads the instruction from memory (at address \$PC) and write it in \$IR.
 - **Instruction Decode (ID)**
 - computes what needs to be computed before execution: jump address destination, access to register, etc.
 - **Execute (EX)**
 - executes the instruction: ALU computation if needed
 - **Memory Access (MEM)**
 - Loads (or stores) data from memory if needed
 - **Write Back (WB)**
 - Writes the result into the register file if needed

The MIPS example

- The RISC paradigm was invented by Berkeley and popularized by Hennessy and Patterson in the book on MIPS
- MIPS stands for *Microprocessor without Interlocked Pipeline Stages* and propose an architecture to execute each stage independently



from MIPS website <https://www.mips.com/>

Christian Wolf's slides

- Use Christian Wolf slides for explaining MIPS instruction pipeline
- Here

example of MIPS pipeline CPU architecture

- Taken from Hennessy/patterson book

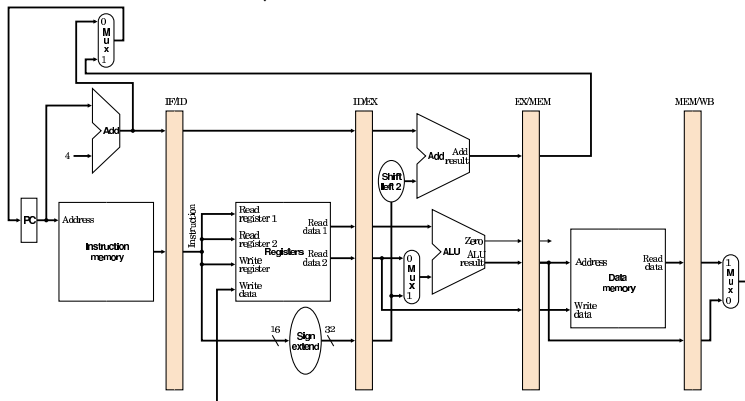
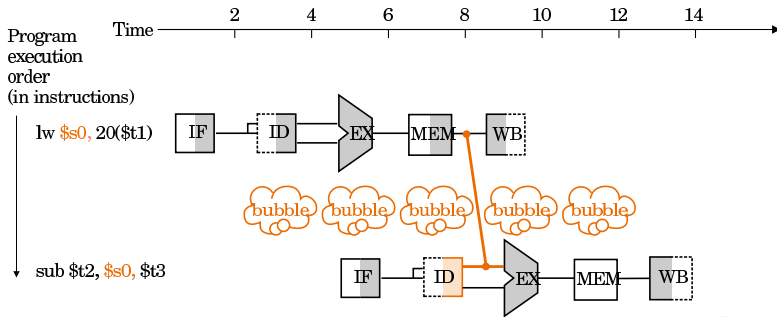


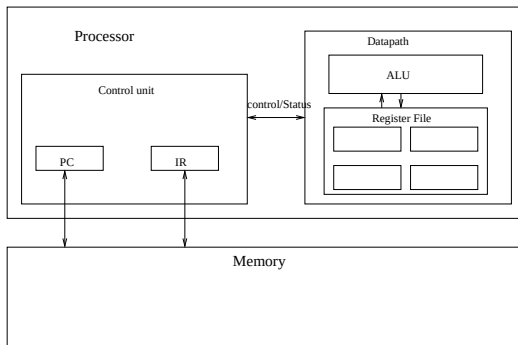
Illustration of bubble on MIPS

- When next instruction cannot be fetched directly (because it need the result of previous instruction for instance) it creates a “bubble”
- For instance: an addition using a register that was just loaded
- The value of the register will be available after the MEM stage of first instruction, hence we can delay on only on cycle, provided there is a *shortcut*.



Another illustration of instruction pipeline

- Go back to our previous representation of the processor and memory:
 - Von Neumann computer= Memory + CPU
 - CPU= = control Unit + Datapath
 - Datapath= ALU + Register file



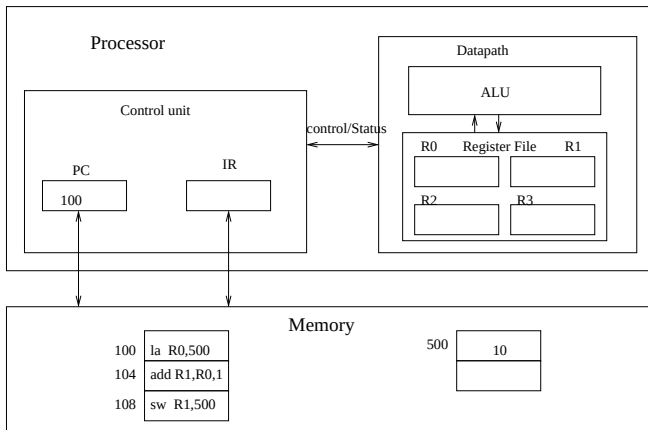
A pipeline example from MIPS

- Execute the sequence of assembly instruction:
 - load value at address 500 in register R0
 - Add 1 to R0 and put result in R1
 - store value of Register R1 at address 500
- (Think of $i=i+1$)
- Code:

```
la R0,500
add R1, R0, 1
sw R1,500
```

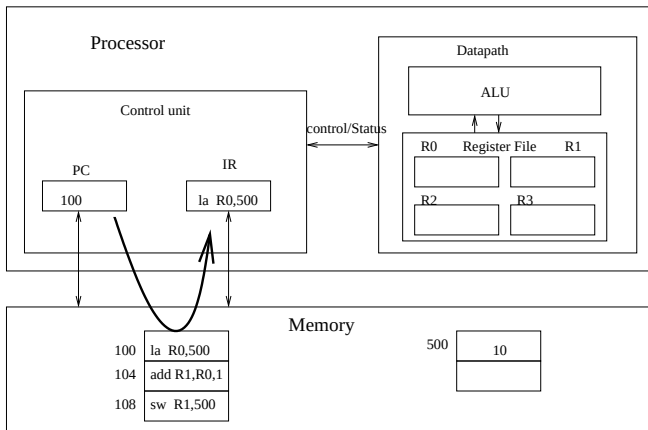
First possible execution: without pipeline

- Before execution starts, \$PC contains the address of the first instruction: 100



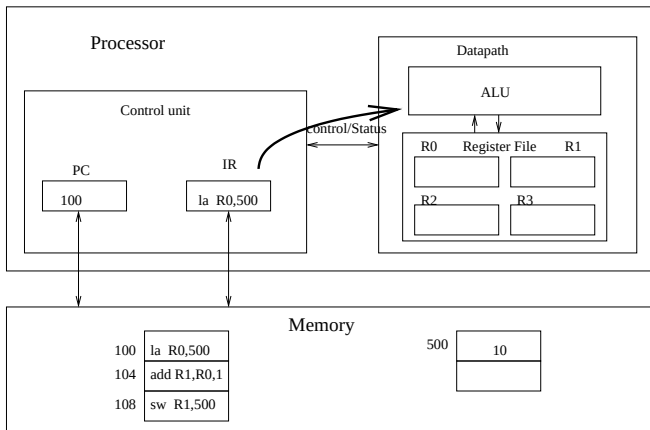
cycle 1

● Instruction Fetch



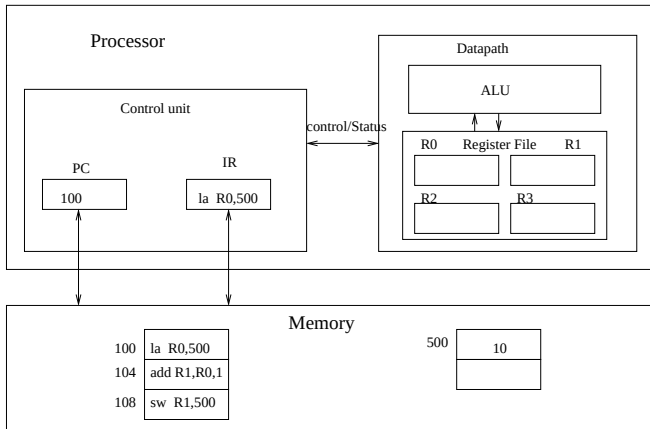
cycle 2

• Instruction Decode



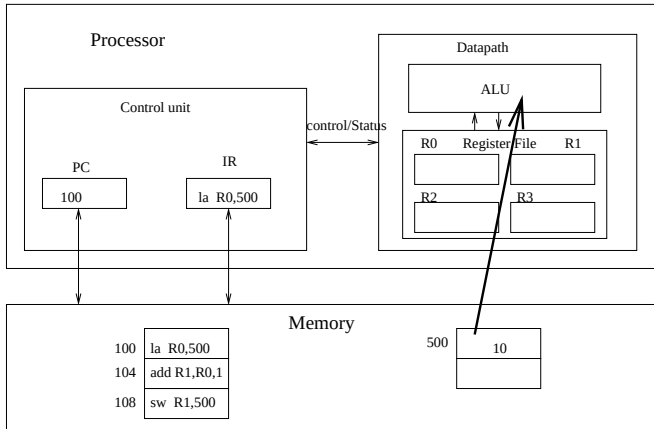
cycle 3

- Execute (nothing for load)



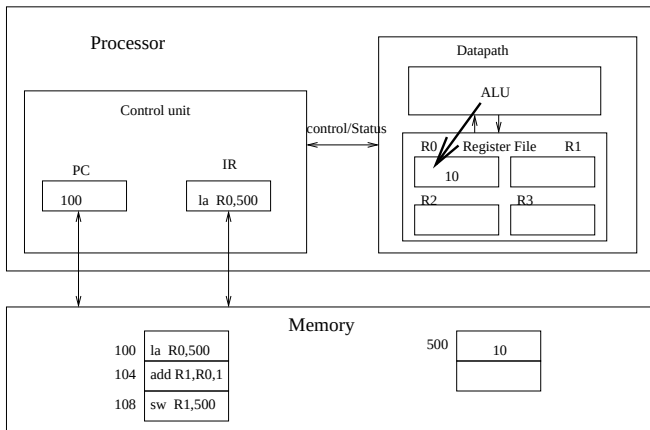
cycle 4

• Memory access



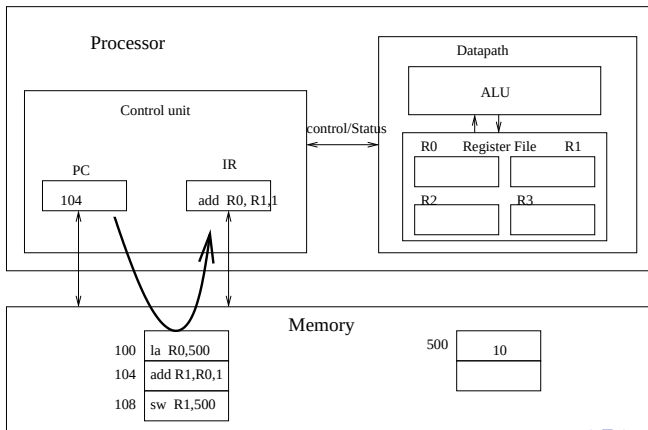
cycle 5

• Write Back



cycle 6

- increment \$PC
- Fetch next instruction
- etc. etc.

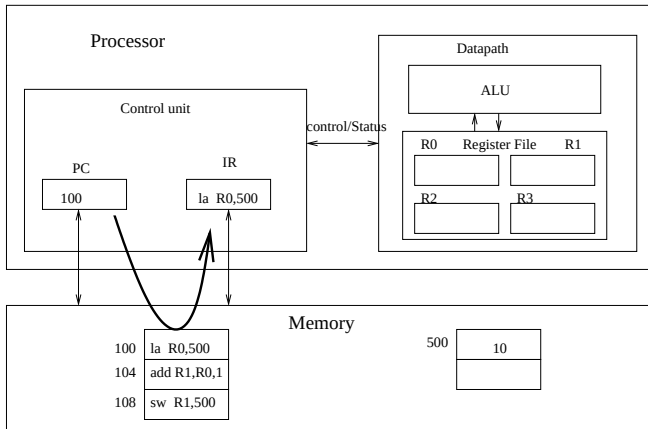


Counting CPI for non-pipelined architecture

- CPI = Cycle per instruction
- 5 cycles for executing on instruction
- $\Rightarrow$ 15 cycles for 3 instructions.

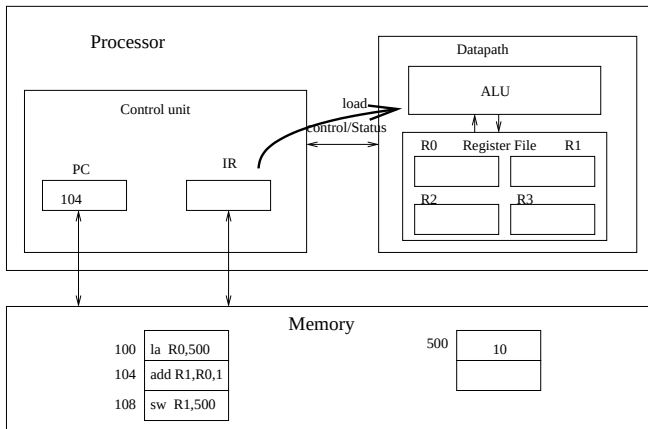
Example of pipelined execution

• Instruction Fetch (for 'load' instruction)



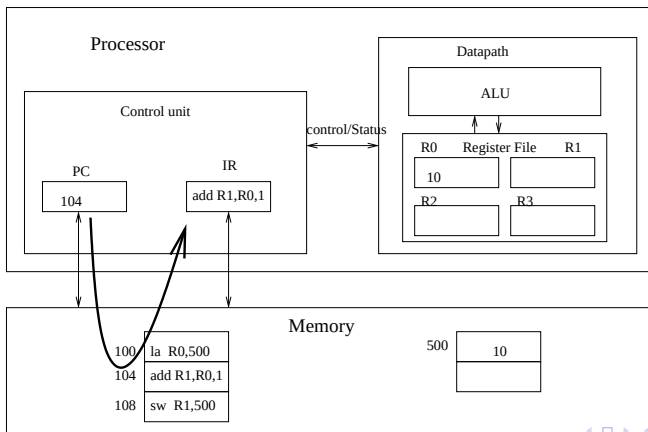
cycle 2

- Instruction Decode (for load)
- Instruction Fetch (for 'nothing' because of a bubble: instruction 'add' delayed)



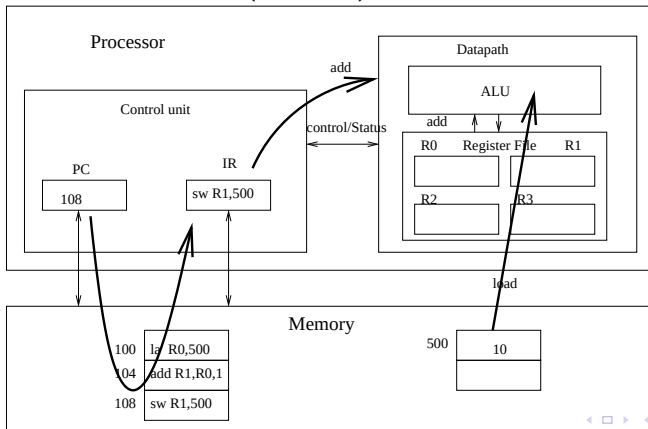
cycle 3

- Execute (for load: nothing to do)
- Instruction Decode (for 'nothing')
- Instruction fetch (for 'add')



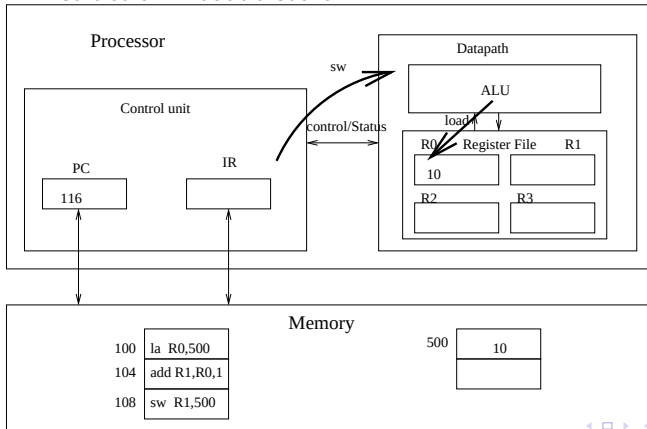
cycle 4

- Memory access (for load)
- Execute (for 'nothing')
- Instruction Decode (for add)
- Instruction fetch (for store)



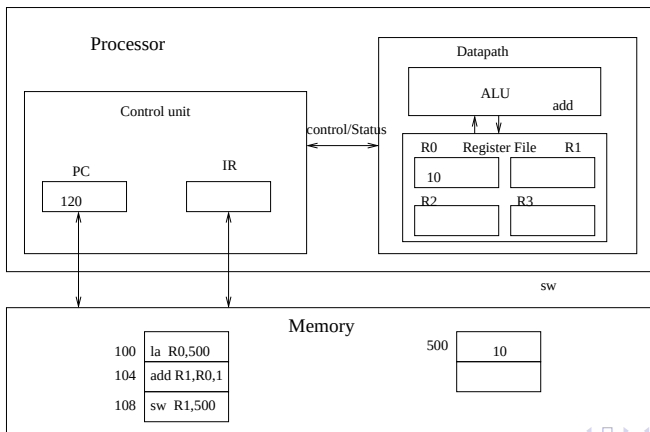
cycle 5

- Write Back (instruction load)
- Memory access (for 'nothing')
- Execute (instruction add: bypass)
- Instruction Decode store



cycle 6

- Write Back (for 'nothing')
- Memory access (instruction add, nothing to do)
- Execute (instruction store: nothing to do)



-
- The diagram illustrates the internal components of a processor and its interaction with memory. The **Processor** is divided into a **Control unit** and a **Datapath**.
- Control unit:** Contains the **PC** (Program Counter) with the value 124 and the **IR** (Instruction Register). It is connected to the **Datapath** via a **control/Status** bus.
 - Datapath:** Contains the **ALU** (Arithmetic Logic Unit) and the **Register File**.
 - The **ALU** is performing an **add** operation, indicated by the label "add" and arrows pointing to it from the Register File.
 - The **Register File** contains four registers: **R0** (value 10), **R1** (value 11), **R2**, and **R3**.
 - Memory:**
 - A table of instructions is shown on the left:

100	la R0,500
104	add R1,R0,1
108	sw R1,500
 - A **sw** (store word) instruction is being executed, moving data from **R1** to memory address **500**. This is indicated by a thick arrow labeled "sw" pointing from **R1** to a memory location containing the value 11.

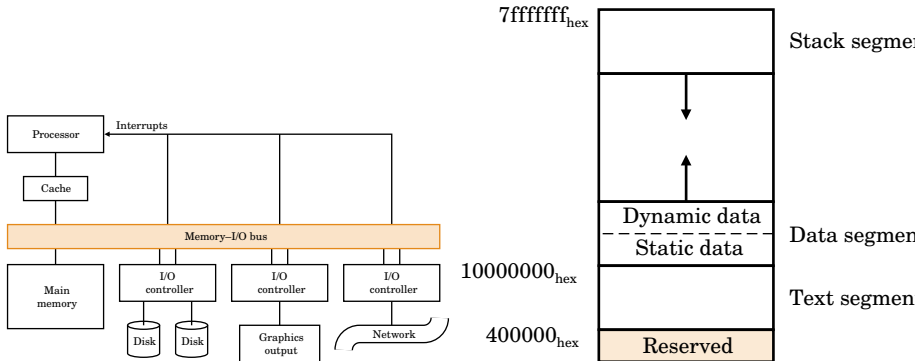
Counting CPI for both architectures

- Non-pipelined architecture:
 - 5 cycles for one instruction
 - $\Rightarrow$ 15 cycles for 3 instructions.
- Pipelined architecture:
 - 5 cycles for one instruction
 - 8 cycles for 3 instructions.
 - $\Rightarrow$ without bubbles, one instruction per cycle
 - A 'jump' instruction interrupt the pipeline (need to wait for the address decoding to fetch next instruction) $\Rightarrow$ *pipeline stall*
 - Some ISA allow to use these *delay slots*: one or two instruction *after* the jump are executed before the jump occurs.

Du langage à l'exécution



Rappels d'architecture

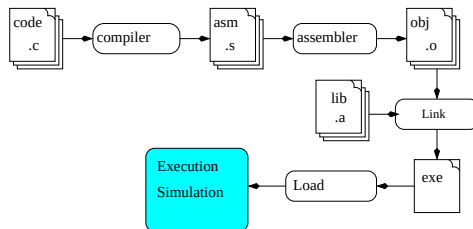


Architecture view from the programmer

- Modern systems allow
 - To run multiple independent programs in parallel (process)
 - To access memory space larger than physical memory available (virtual memory)
- For the programmer: all this is transparent
 - Only one program runs with very large memory available
- The processor view memory contains:
 - The code to execute
 - Static data (size known at compile time)
 - Dynamic data (size known at runtime: the heap, and the space needed for the execution itself: the battery)
- The programmer sees only the data (static and dynamic)

compilation process

- the complete process will translate a C program into code executable (loading and execution will take place later).



- We often call *compilation* the set compiler + assembler
- The gcc compiler also includes an assembler and linking process (accessible by options)

Your compilation process

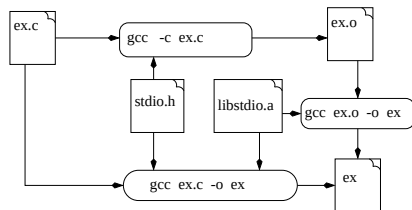
- The programmer:
 - Write a program (say a C program: `ex.c`)
 - Compiles it to an object program `ex.o`
 - links it to obtain an executable `ex`

content of `ex.c`

```
#include <stdio.h>

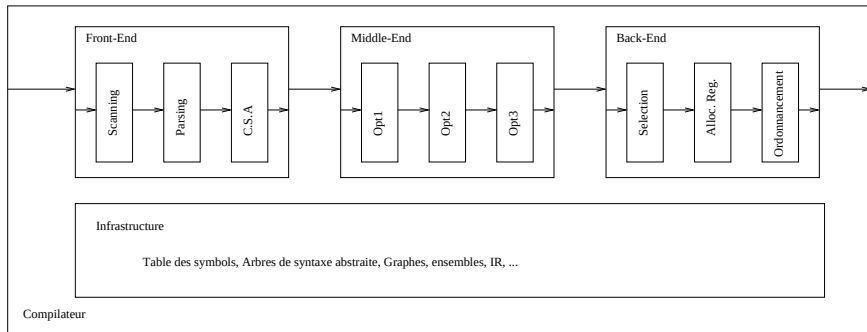
int main()
{
    printf("hello World\n");

    return(0);
}
```



Zooming on “compilation”

- The compilation process is divided in 3 phases:



Compilation: the front-end

- The front end of an embedded code compiler uses the same techniques as traditional compilers (we can want to include assembler parts directly)
- Parsing LR(1): the parser is usually generated with dedicated *metacompilation* tools such as Flex et bison for GNU

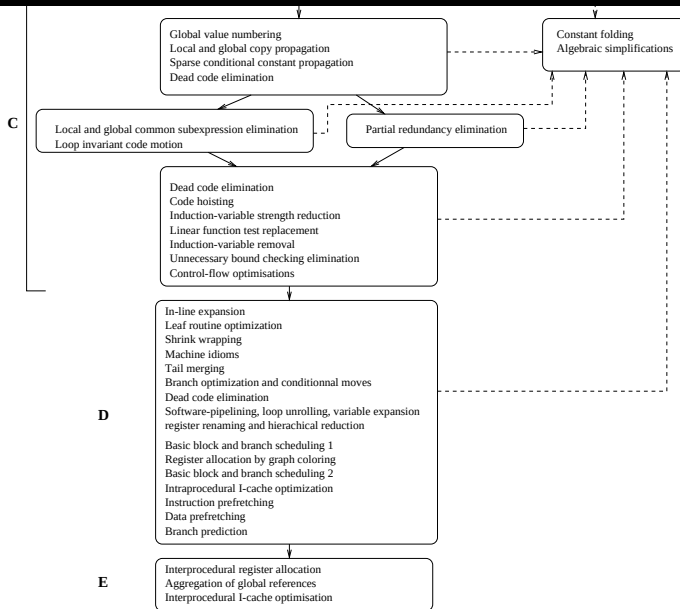
Compilation: The middle-end

- Some phases of optimizations are added, they can be very calculative
- Some example of optimisation independent of the target machine architecture
 - Elimination of redundant expressions
 - dead code elimination
 - constant propagation
- Warning: optimization can hinder the understanding of the assembler (use the -O0 options with `gcc`)

oooooooooooo

oooooooooooooooooooooooooooooooooooo

oooooooooooooooooooo



Compilation: The back-end

- The code generation phase is dedicated to architecture target. Retargetable compilation techniques are used for architectural families.
- The most important steps important are:
 - Code selection
 - Register allocation
 - instruction scheduling

GCC

- The gcc command runs several program depending on the options
 - The pre-processor cpp
 - The compiler cc1
 - The assembleur gas
 - The Linker ld
- gcc -v allow to visualize the different programs called by gcc

The pre-processor cpp or gcc -E

- the task of the pre-processor are :
 - elimination of comments,
 - inclusion of source files
 - macro substitution (`#define`)
 - conditionnal compilation.
- Example:

ex1.c

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))  
...  
f=MAX(3,b);
```

ex1.i

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))  
...  
f=((3) > (b) ? (3) : (b));
```

The compiler cc1 or gcc -S

- generate assembly code
- `gcc -S main.c -o main.S`
- Exemple :

```
void main()
{ int i;
  i=0;

  while (1)
  {
    i++;
    nop();
  }
}
```

Assembly code generated (for MSP430)

```
mov    #2558, SP        ; stack initialization de la pile
mov    r1, r4           ; r4 <- SP
mov    #0, 0(r4)        ; i initialization
inc    0(r4)            ; i++
nop                     ; nop();
jmp    $-6              ; unconditionnal jump (PC-6):
incd    SP              ;
br      #0x1158          ;
```

Assembly code produce by mspgcc -S

```

.text
.p2align 1,0
.global      main
.type        main,@function

main:
/* prologue: frame size = 2 */
.L__FrameSize_main=0x2
.L__FrameOffset_main=0x6
    mov      #(__stack-2), r1
    mov      r1,r4
/* prologue end (size=3) */
    mov      #llo(0), @r4
.L2:
    add      #llo(1), @r4
    nop
    jmp      .L2
/* epilogue: frame size=2 */
    add      #2, r1
    br       #__stop_progExec__
/* epilogue end (size=3) */
/* function main size 14 (8) */

```

Assembler as ou gas

- transform an assembly code into object code (binaire representation of symbolic assembly code)
- Option `-c` of `gcc` allow to combine compilation et assembly
`gcc -c main.c -o main.o`

Linking: ld

- Produce the executable (a.out by default) from object code of programs and library used
- There are two ways to use libraries in a program
 - Dynamic or shared libraries (default option): the code of the library is not included in the executable, the system dynamically loads the code of the library in memory when calling the program. We need then only *one* version of the library in memory even if several programs use the same library. The library must be installed on the machine, before running the code.
 - Static libraries: the code of the library is included in the executable. The executable file is bigger but you can run it on a machine on which the library is not installed.

Binary file manipulation

Some usefull command:

- `nm`

Allow to know symboles (i.e. label: function names) used in an object file or executable

```
trisset@hom\ $ nm fib.elf | grep main
000040c8 T main
```

- `objdump` allow to analyze a binary file. For instance it can get correspondance between binary representation and assembly code

```
trisset@hom$ objdump -f fib
fib:          file format elf32-msp430
architecture: msp:43, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00001100
```