Operating System And Computer Architecture

International Master Computer Networks & IoT Systems

Sami Taktak

sami.taktak@cnam.fr

CEDRIC — Centre d'Étude et De Recherche en Informatique et Communications
CNAM — Conservatoire National des Arts et Métiers

October 2023

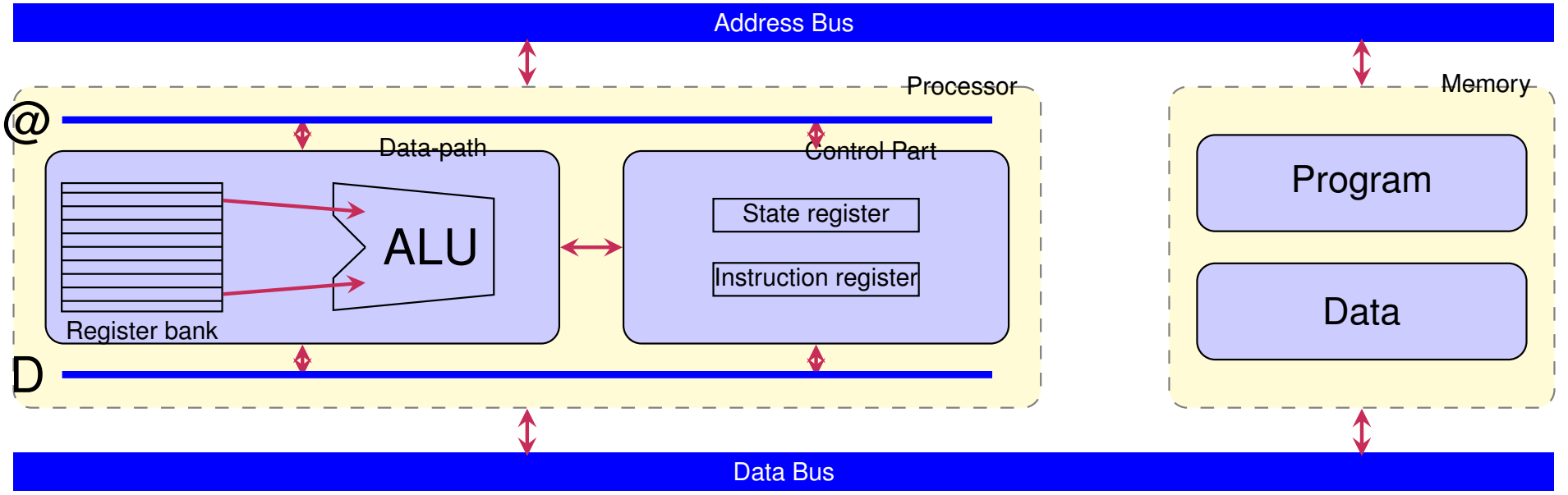
Part I

Information Storage

Information Storage

Storage Units and localization

- Data and instructions of a running program are stored in memory
- A part of these information needs to be stored inside the processor: instruction being executed + program's data
- Inside the processor, any (temporary) data is stored in registers
- Information transfer between the memory and the processor is done through buses

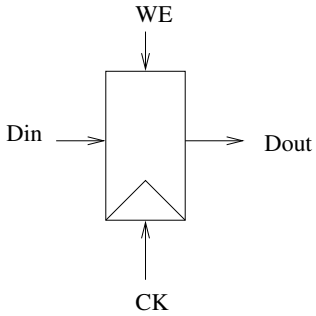


Register

n Bits Register

n bits register allow to store a binary word of *n* bits. It is composed of *n* registers of 1 bit. The 1 bit register is the elementary memory unit.

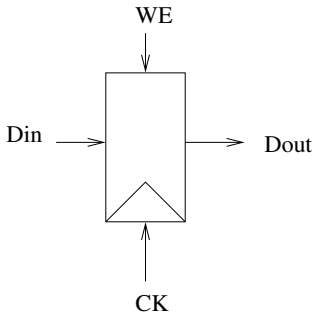
Register 1 bit



- `CK` is the clock signal
- `WE` is Write Enable signal
- `Din` is Data Input, Dout is Data Output

Register

1 Bit Registre



When `CK` switch from 0 to 1 (raising edge):

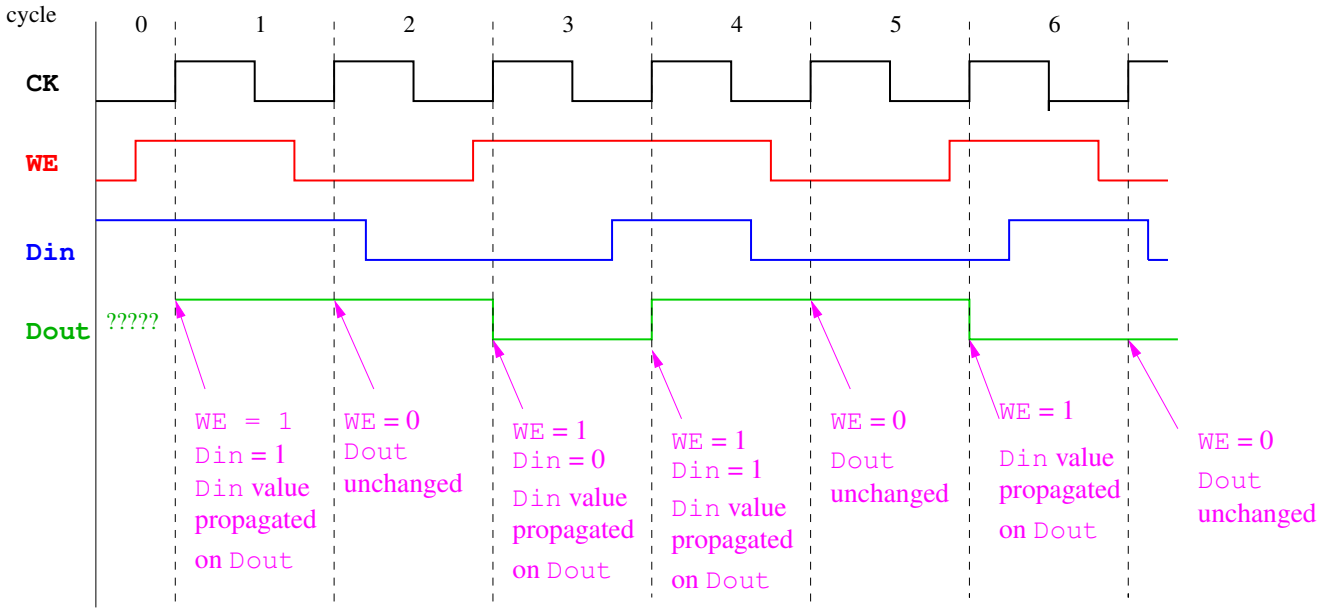
- if `WE` = 1 then `Din`'s value is stored in the register and displayed on `Dout`
- else (`WE` = 0), the register's value remain unchanged and `Dout` display the last stored value (memory effect). At any other time (i.e. falling edge, no value change), whatever `WE` value is, `Dout` is unchanged.

Digital Timing Diagram

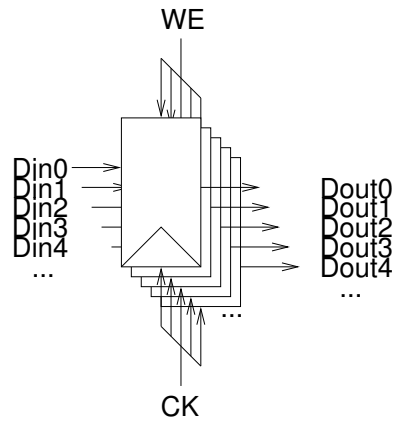
A *Digital timing diagram* represents signals values during a period of time.

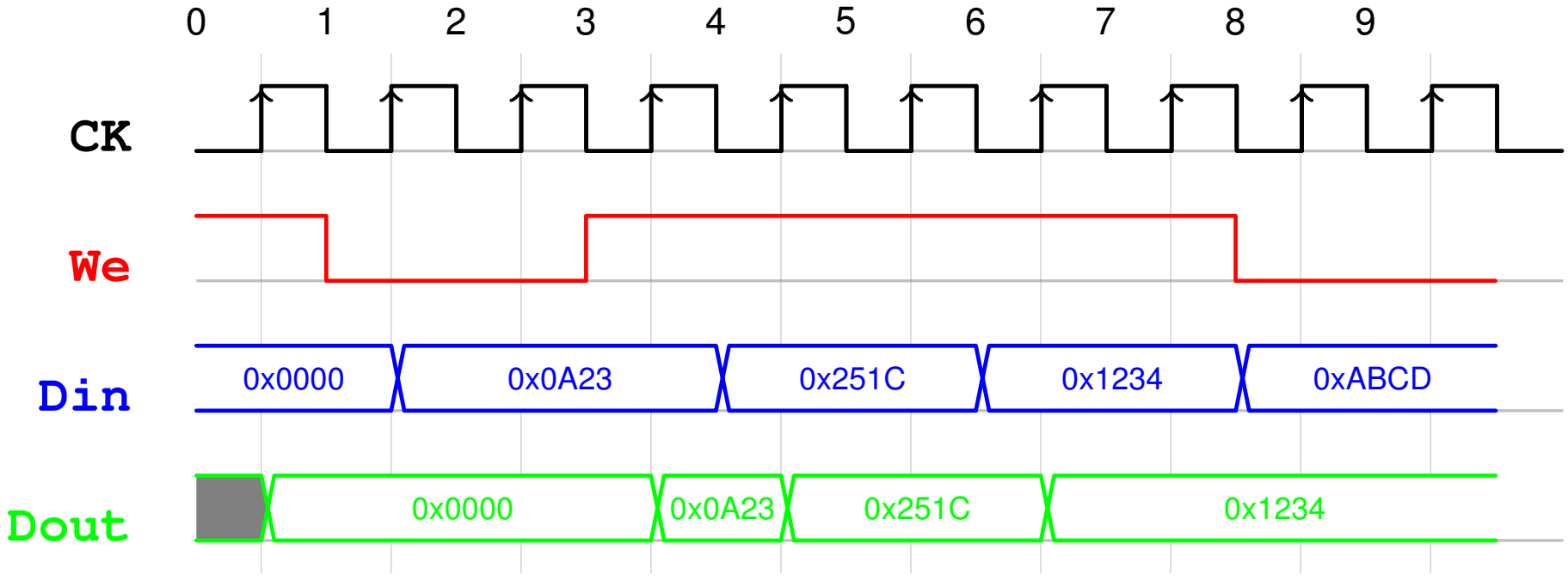
Digital Timing Diagram Example

Digital Timing Diagram example of a 1 bit register input and output signal during a 6 period time

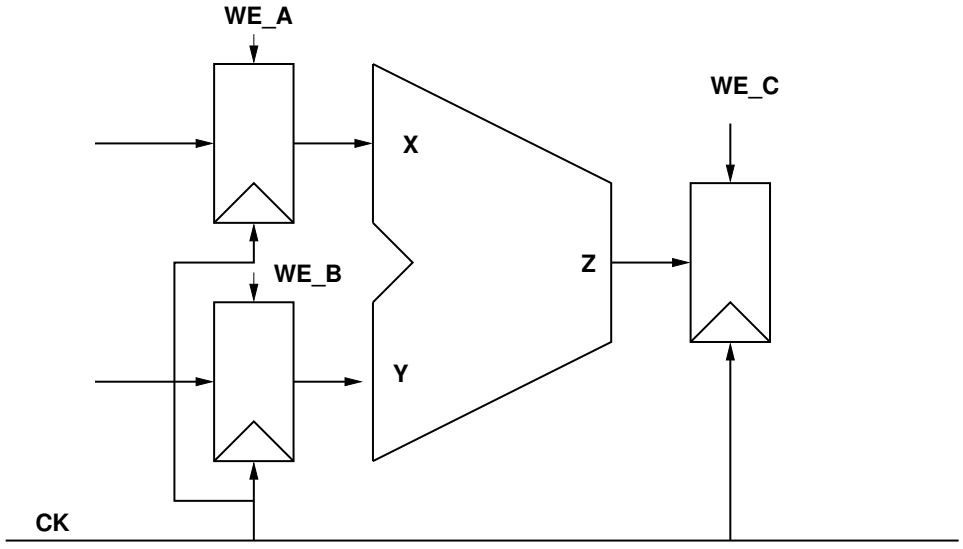


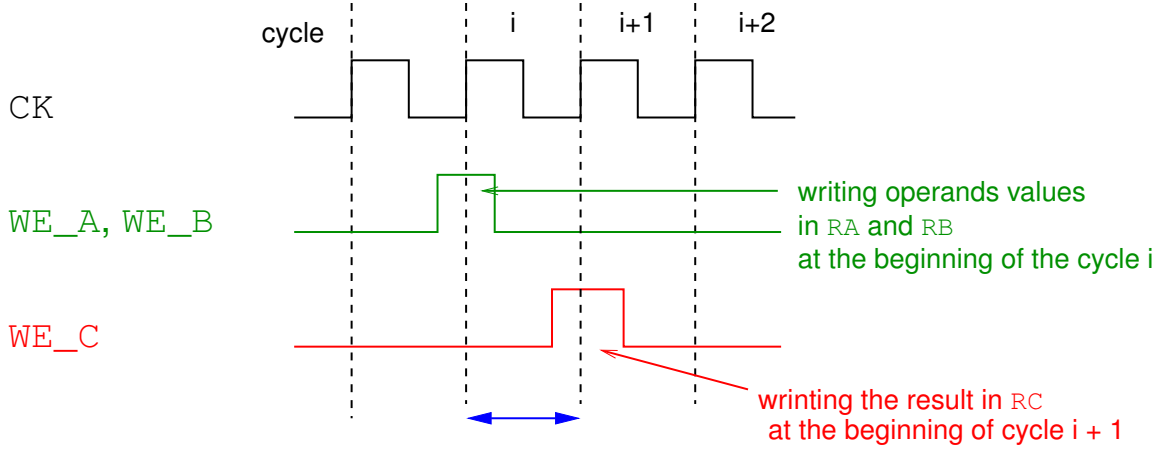
16 Bits Word Storage





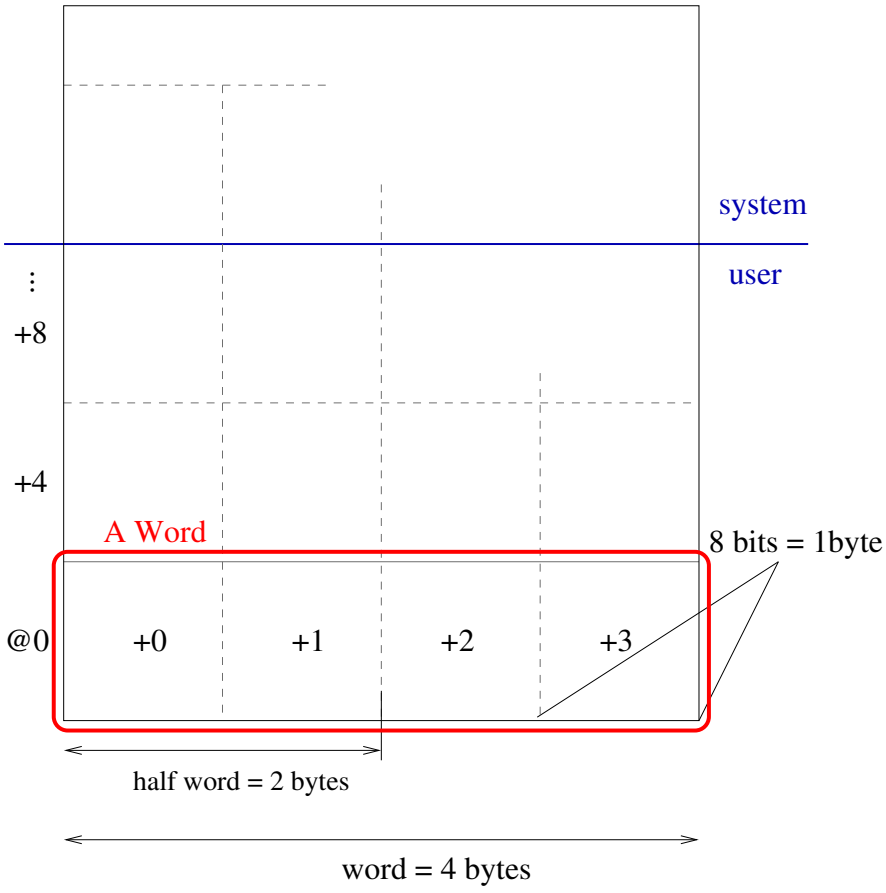
Information transfer in the ALU





transfering operands in the ALU in 1 cycle

Memory Architecture



Memory

- The memory is a table of n rows of p bits words
- byte address = position in the row + row's number
- Processor can only access memory a byte (or a word) at a time
- Addresses are represented on 32 bits
- We can have access to a word, half-word or 1 byte

Memory Capacity

Definition

Memory capacity correspond to the number of bytes it can store.

Storing Capacity

- 1 kilo-byte or 1 kB = 2^{10} bytes
- 1 mega-byte or 1 MB = 2^{20} bytes
- 1 giga-byte or 1 GB = 2^{30} bytes
- 1 tera-byte or 1 TB = 2^{40} bytes

Storing Multiple Bytes Words

Storing Multiple Bytes Words

Data of size greater of 1 byte is stored in contiguous bytes.

Little/Big-endian

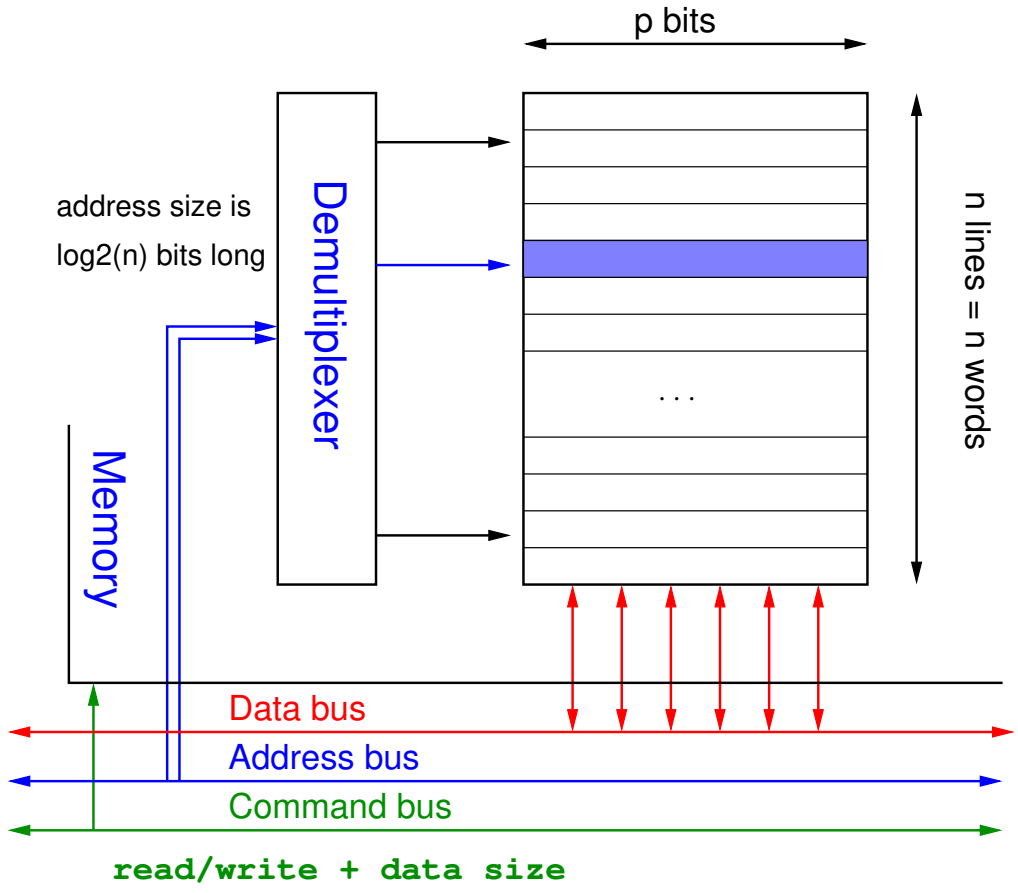
- Let $M = 0xb_3b_2b_1b_0$ be a word of 4 bytes stored at address A
- 2 storing possibilities:
  - Little-endian : least significant byte stored at the smallest address (stored first)
  - Big-endian: most significant byte stored at the smallest address (stored first)

Address	Little-endian	Big-endian
A	b_0	b_3
A + 1	b_1	b_2
A + 2	b_2	b_1
A + 3	b_3	b_0

Memory Access

Transferring Data

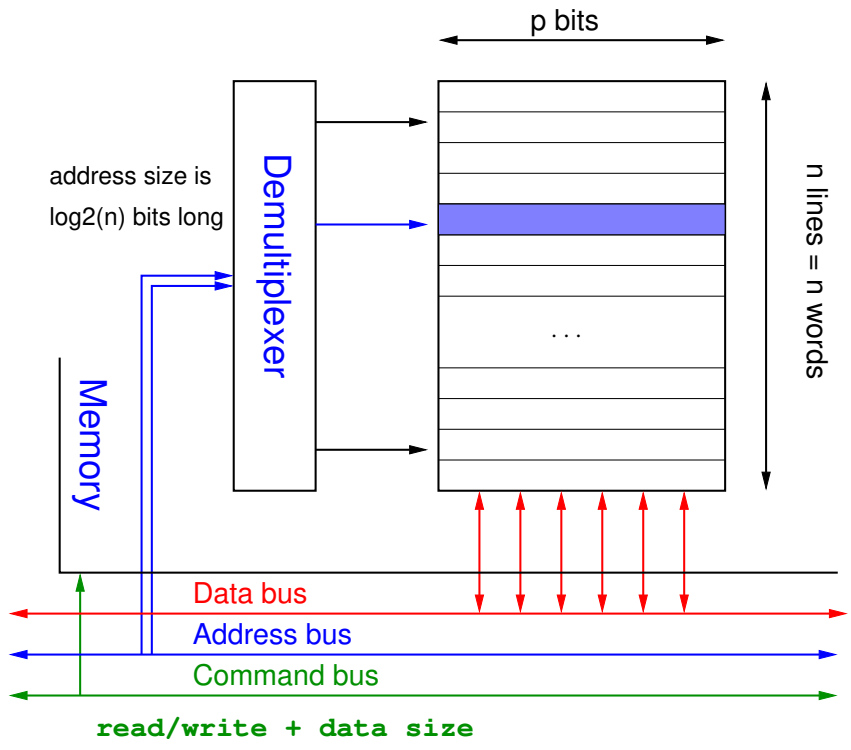
- A demultiplexer select the row corresponding to the address present on the address bus
- the command select the operation : `read` or `write`
- `read` : the read data are transferred from the memory on the data bus
- `write` : the data to write are read on the data bus and written in the memory



Data Transfer between memory and processor

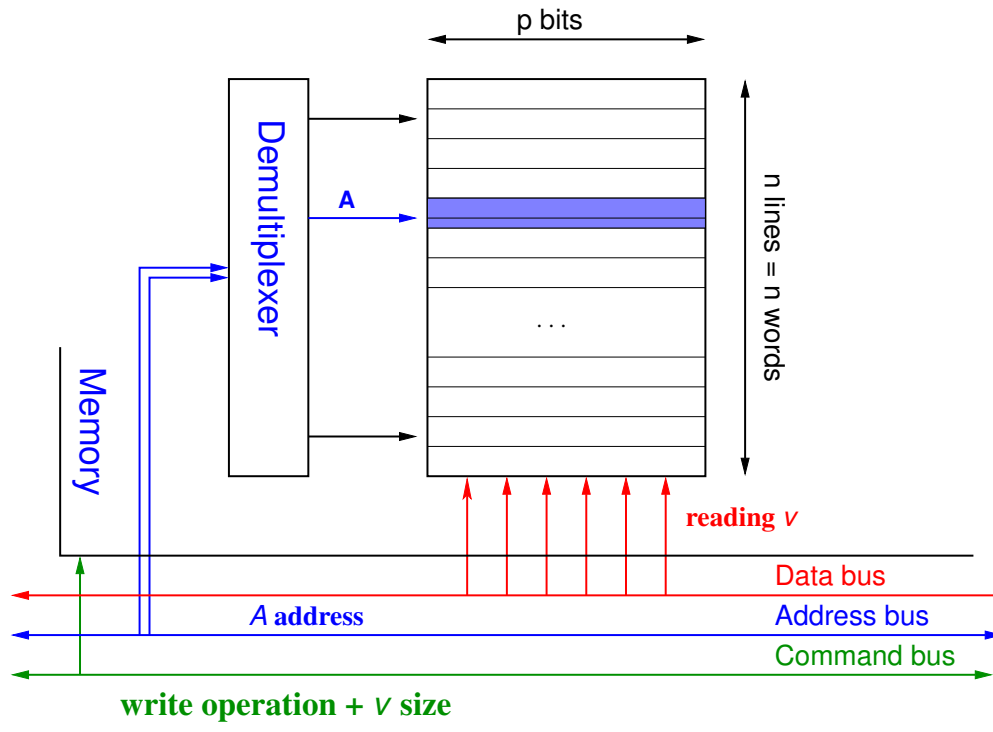
The processor initiate data transfers and provides to the memory:

- The word address to be transferred
- The word size to be transferred
- The transfer direction:
  - if processor $\rightarrow$ memory then it is a write or *store* operation
  - if memory $\rightarrow$ processor then it is a read or *load* operation
- The data to write in case of store operation



Write Protocol of v at Address A

1. The processor puts:
   - A on address bus
   - v on data bus
   - ask for a write on command bus
   - specify v size on command bus
2. The memory reads address A on address bus, the write command on the command bus, then reads the data v on the data bus and store it at address A

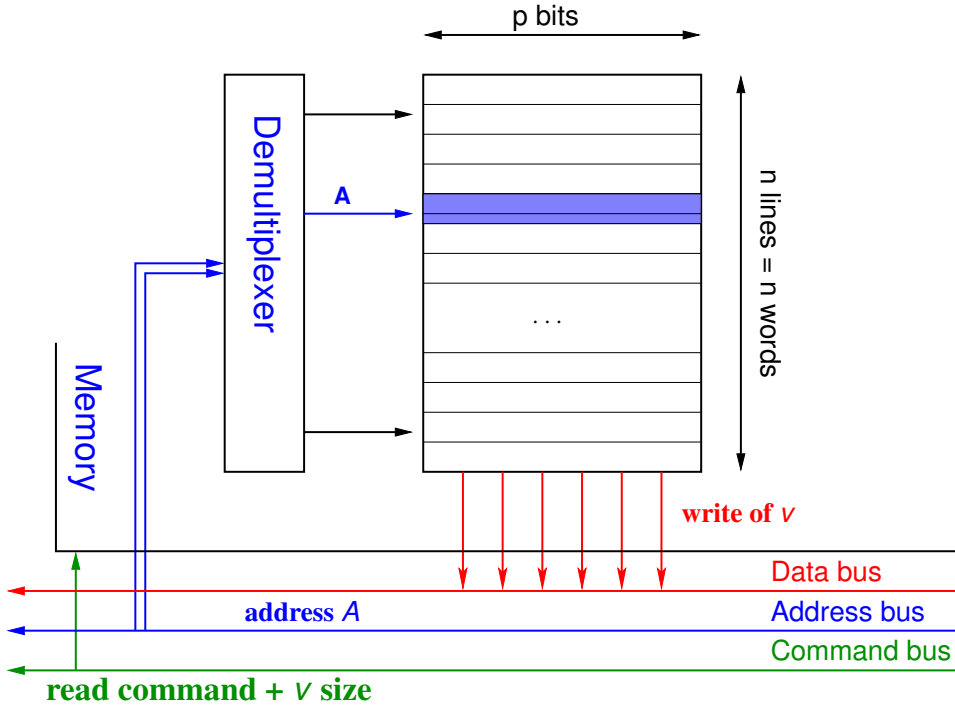


Write Effect

$Mem[A] = v$ until next write to address A or to one address used to store v

Read Protocol of *v* at Address *A*

1. The processor puts:
   - *A* on address bus
   - ask for a read on command bus
   - specify *v* size on command bus
2. The memory reads address *A* on address bus and the read order on command bus, then write *Mem*[*A*] on data bus and acknowledge the data transfer
3. The processor reads *v* on data bus and store in the destination register



Read Effect

v value (value of *Mem*[*A*]) is the value of the last write at address *A*.
A read does not modify the memory.

Part II

The Assembly language

The language machine

Assembly language

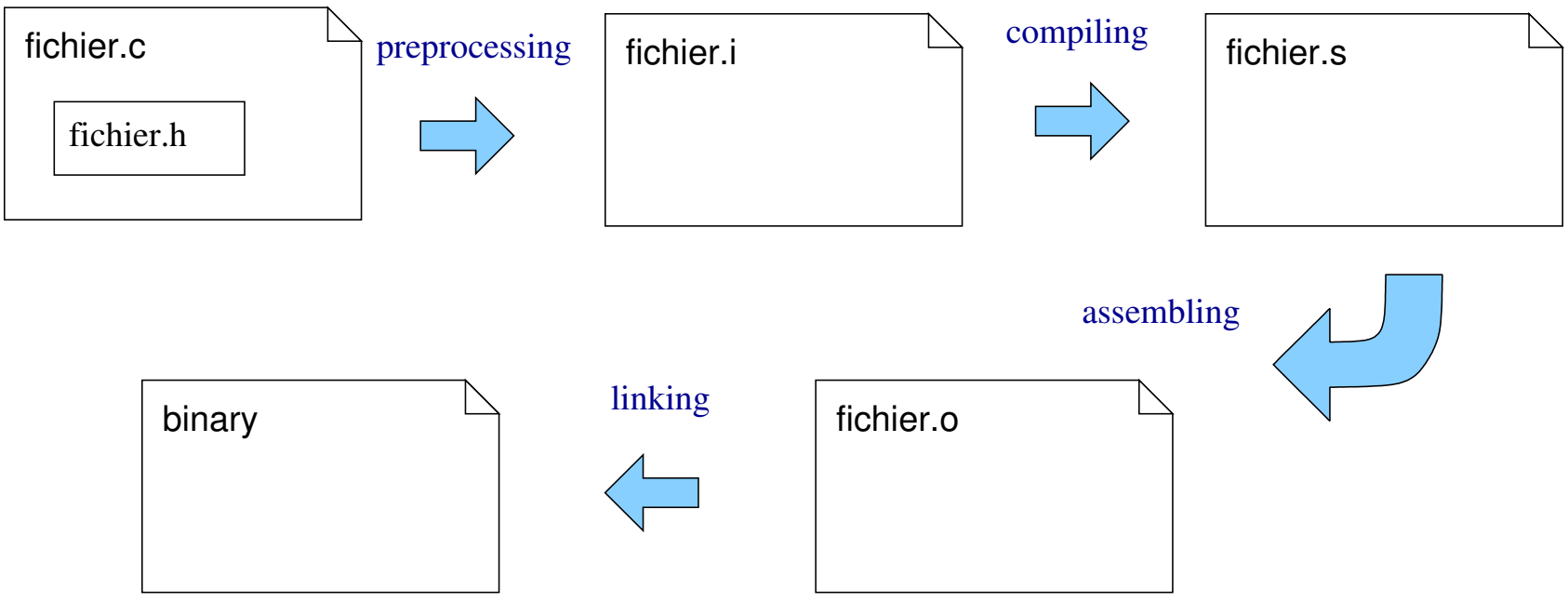
Its words == *instructions* / its vocabulary == *instruction set*

- If you know one, it's really easy to learn others.
  - Helps to design the hardware architecture
  - Helps to design the compiler
  - Maximizes the performance
  - Minimizes the cost

Our study : The MIPS32, a RISC architecture

- 80s processor: The first was born 1981 (Hennessy, Stanford University)
- Really well-suited for courses and exercises
- It is/was used : PlayStation, NEC, Siemens, Nintendo ...

Compiler



- What a compiler can do ?
- What are its choices ?

Arithmetic operations

Operations in MIPS

```
1 add  a, c, d     # The sum of c and d is placed in a
2 add  a, a, e     # The sum of c, d and e is placed in a
3 sub  a, a, f
```

Simplicity

- One operation at a time
- One instruction per line
- Exactly three operands

Simplicity favors regularity

First examples

Question 1

What is the assembly code of the following C instruction ?

```
1 a = (f + c) - (d + e);
```

Answers

a.	**add** a, f, c **sub** a, a, d **sub** a, a, e	b.	**add** t, f, c **sub** t, d, e **add** a, a, t
c.	**add** t, f, c **add** p, e, d **sub** a, t, p	d.	**add** f, f, c **add** d, d, e **sub** a, f, d

First examples

Question 2

What is the assembly code of the following C instruction ?

```
1 a = (f + c) - (d + e);
```

Answers

a.	**add** a, f, c **sub** a, a, d **sub** a, a, e	b.	**add** t, f, c **sub** t, d, e **add** a, a, t
c.	**add** t, f, c **add** p, e, d **sub** a, t, p	d.	**add** f, f, c **add** d, d, e **sub** a, f, d

Logic Operations

Shift operation **sll**, **srl**

```
1 sll a, c, 4 # a = c << 4 bits
2 a : 0000 0000 0000 0000 0000 0000 0101 0000
3 c : 0000 0000 0000 0000 0000 0000 0000 0101
```

Bit-by-bit operations **and**, **andi**, **or**, **ori**, **nor**

```
1 andi a,  c, 4 # a = c and 4
2 c : 0000 0000 0000 0000 0000 0000 1111 0100
3 4 : 0000 0000 0000 0000 0000 0000 0000 0100
4 a : 0000 0000 0000 0000 0000 0000 0000 0100
```

```
1 nor a, c, $0 # a = ! ( c || $0 )
2 c   : 0000 1111 1111 0000 1111 1011 1110 1010
3 $0  : 0000 0000 0000 0000 0000 0000 0000 0000
4 a   : 1111 0000 0000 1111 0000 0100 0001 0101
```

Operation operands

Registers

- Limited resource built directly in hardware.
- Visible for the programmer

We have 32 registers of 32 bits (a word)

Memory

Complex data structures, arrays ... do not fit into a 32-bits registers.
These structures are located in the memory

Constant

A constant operand is called an immediate and it is included inside arithmetic instructions. `addi a, b, 3`
A constant (immediate) is encoded on 16 bits.

Visible registers

Non protected registers

The width of all register that can be read or write by the instructions is *32 bits*.

Non protected registers – User

- R_i $(0 \leq i \leq 31)$ 32 general purpose registers:
  use for temporary results
- *PC* Program counter:
  address of the current instruction
- *HI*, *LO* Multiplication and division register:
  use for result on 64 bits

Visible registers

Protected registers

Protected registers – Supervisor

- *SR* (Status register):
  defines the mode user or system
- *CR* (Cause register):
  defines the cause of an interruption or exception
- *EPC* (Exception program counter):
  address of an instruction
- *BAR* (Bad address register) :
  contains the value of a bad address

General Purpose Registers (GPR)

Naming Convention

Register	Name	Description
0	$zero	Value 0, *not modifiable*
1	$at	Reserved for macro instruction
2-3	$v0,$v1	Return value and function result
4-7	$a0-$a3	Used for function parameters but not protected
8-15 24-25	$t0-$t9	Used as temporary variables valued not protected
16-23	$s0-$s8	Persistent register protected by the function call
26-27	$k0-$k1	Reserved for the kernel
28	$gp	Data segment pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Function return address

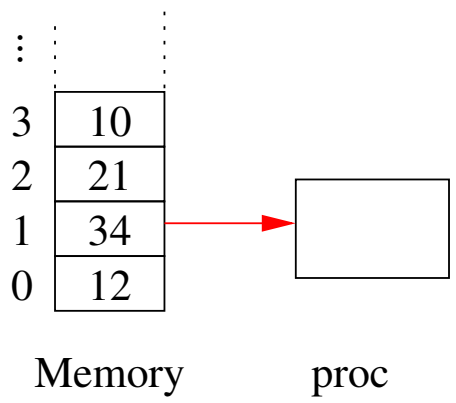
Memory Operands

Data transfers instructions

Arithmetic instructions only works with registers

- Transfer data from the memory to registers
- Need the memory address of the data

Load word



A load instruction format:

- destination register
- a constant: the *offset*
- a register containing the base address

The sum of the constant and the base address gives the memory address

Data transfer, one way

Yet another addition

Let T a table of 5 elements starting at the address contained in `$s3`, and the compiler choose `$s1` and `$s2` for variables *a* and *b*.

```
a = b + T[2];
```

Solution

```
1        lb   $t0,2($s3)   # t = T[2]
2        add  $s1,$s2,$t0  # a = b + t
```

Data transfer, one way

Yet another addition

Let T a table of 5 elements starting at the address contained in `$s3`, and the compiler choose `$s1` and `$s2` for variables *a* and *b*.

```
a = b + T[2];
```

Solution

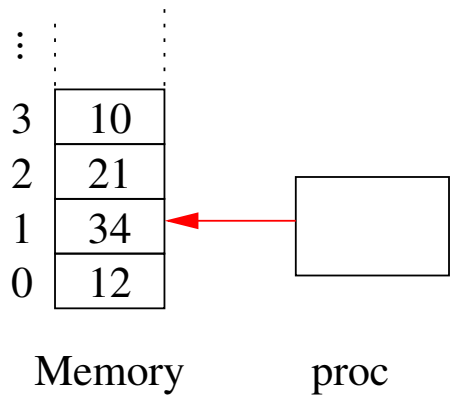
```
1        lb   $t0,2($s3)   # t = T[2]
2        add  $s1,$s2,$t0  # a = b + t
```

Storing result

Store word

- Transfer data from a register to the memory
- Need the memory address of the data

Load word



A store instruction format:

- source register
- a constant: the *offset*
- a register containing the base address.

The sum of the constant and the base address gives the memory address.

Data transfer, other way

Addition

Let T a table of 5 elements starting at the address contained in `$s3`, and the compiler choose `$s1` and `$s2` for variables *a* and *b*.

```
T[2] = b + a;
```

Solution

```
1        add $t0,$s1,$s2 # t = a + b
2        sb  $t0,2($s3) # T[2] = t
```

Data transfer, other way

Addition

Let T a table of 5 elements starting at the address contained in `$s3`, and the compiler choose `$s1` and `$s2` for variables *a* and *b*.

```
T[2] = b + a;
```

Solution

```
1        add $t0,$s1,$s2 # t = a + b
2        sb  $t0,2($s3) # T[2] = t
```

Data transfer

Question 3

Let T a table of 6 elements starting at the address contained in `$s3`, and the compiler choose `$s1` for variable *b*. What is the assembly code of the following C instruction?

```
T[5] = b + T[3];
```

Answers

a.
```
sb $t0, 3($s3)
add $t0, $t0, $s1
lb $t0, 5($s3)
```

b.
```
add $t0, $s1, $s3
 sb $t0, 5($s3)
```

c.
```
lb $s3, 3($t0)
add $t0, $t0, $s1
sb $s3, 5($t0)
```

d.
```
lb $t0, 3($s3)
add $t0, $t0, $s1
sb $t0, 5($s3)
```

Data transfer

Question 4

Let T a table of 6 elements starting at the address contained in `$s3`, and the compiler choose `$s1` for variable *b*. What is the assembly code of the following C instruction?

```
T[5] = b + T[3];
```

Answers

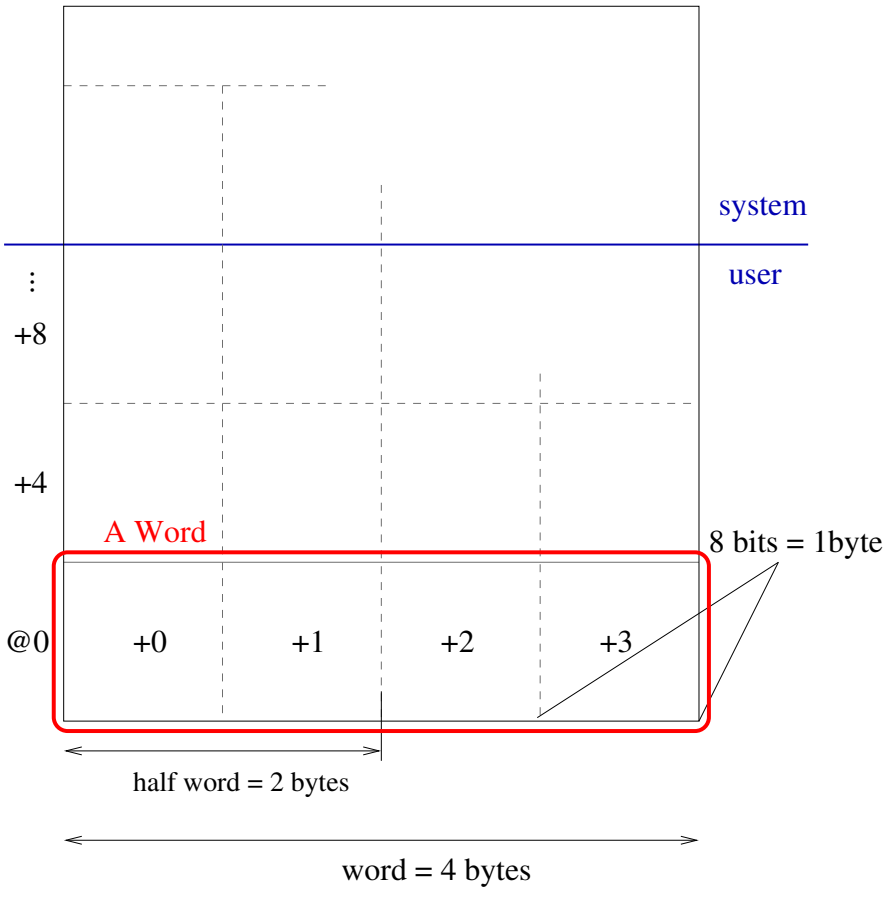
a.
```
sb $t0, 3($s3)
add $t0, $t0, $s1
lb $t0, 5($s3)
```

b.
```
add $t0, $s1, $s3
 sb $t0, 5($s3)
```

c.
```
lb $s3, 3($t0)
add $t0, $t0, $s1
sb $s3, 5($t0)
```

d.
```
lb $t0, 3($s3)
add $t0, $t0, $s1
sb $t0, 5($s3)
```

Memory addresses



Address

- The processor can only access the memory by bytes.
- Addresses are represented using 32 bits.
- Access by word, half-word or 1 byte.
- Operations
  - Load: lw, lh, lb
  - Store: sw, sh, sb

What do we obtain from a memory address ?

Aligned addresses

An object is made of n bytes, the address A is aligned if:

$$A \; mod \; n = 0$$

- @ of a word or an instruction is a multiple of 4
- @ of half-word is a multiple of 2

Offset

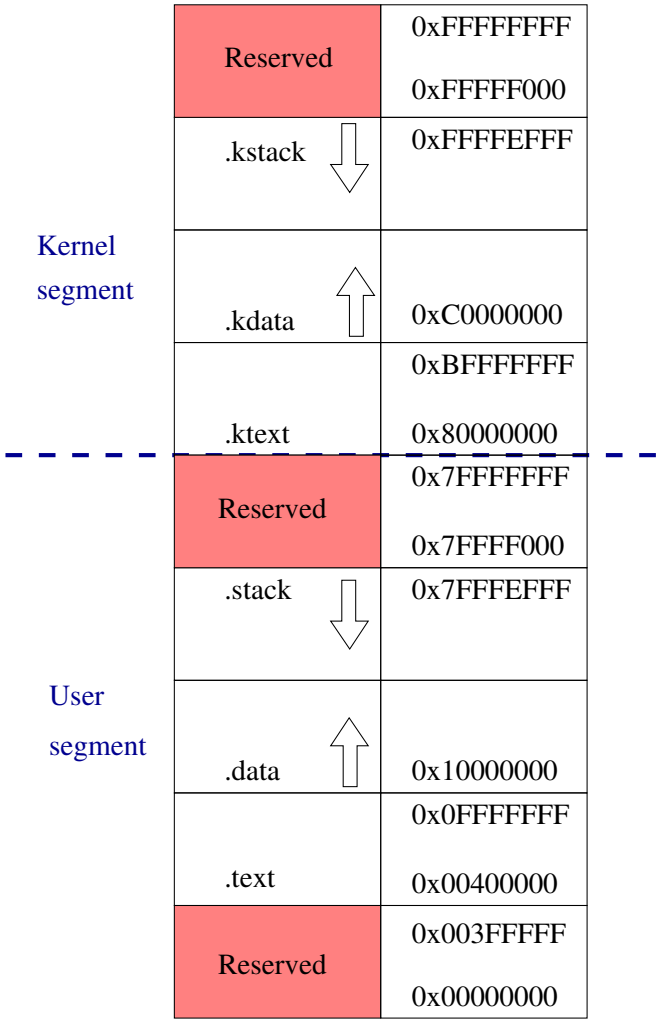
Depending of the amount of data we need, the offset is not the same :

`a = b + T[3]` The offset is 3 $\times$ 4 + the base address .

The proper offset for byte addressing in assembly code is :

```
lw $t0, 12($s3)
add $s1, $s2, $t0
```

Memory organization



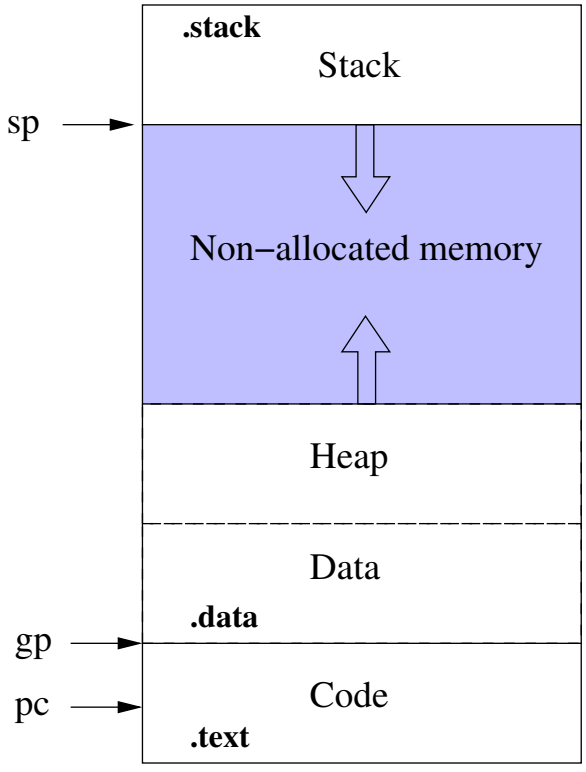
User memory segment

text: Executable code in user mode
data: Data used by the user program, can be initialized, modified
stack: Execution stack of the user program

Kernel memory segment

text: Executable code in kernel mode
data: Data used by the operating system in kernel mode
stack: Execution stack in kernel mode

Memory allocation



- Stack: Functions parameters and some local variables.
  - Size dynamically changes
  - Register `$sp` contains the top of the stack address
- Heap : Data dynamically allocated.
  - Size not known at compile time
  - Size dynamically changes
- Data : Global data statically allocated
  - Size computed by the compiler
  - Register `$gp` contains the bottom of the data address
- Code: Insctruction code
  - Size computed by the compiler
  - Register `pc` contains the address of the current instruction

Memory organization

Question 5

In which part of the memory are tables allocate with a malloc ?

Answer

a. Code
b. Data
c. Heap
d. Stack

Directives

Sections

Specify which memory segments are concerned by the following (macro)-instructions or directives.

`.text,.data,.stack ...`

Variables declaration

- `.align n` Align the address such that *n* LSB is zero.
- `.ascii string,[otherstring]` String without the null terminated character.
- `.asciiz string,[otherstring]` String with the null terminated character.
- `.byte n,[m], .half n,[m], .word n,[m]` value of *n* and *m* contains 8,16,32 bits, put in successive addresses
- `.space n` n bytes are reserved starting at the current address.

Variable declarations

```
1  int a;
2  int t[3] = {1,2,3};
3  char s[5] = {'h','e','l','l','o'
       ,'\0'};
4
5  int main()
6  {
7          exit(0);
8  }
```

```
1  .data
2  a: .space 4
3  t: .word 1,2,3
4  s: .asciiz "hello"
5  .text
6  main:
7          ori $v0, $0, 10
8          syscall
```

How to obtain 32 bits immediate ?

Instruction only have 16 bits immediate !

2 macros

- Does not belong to the instruction set
- Use $at for intermediate result

Load address	`la Rt,addr`	`lui R1,addr>>16` `ori Rt,R1,addr & 0xFFFF`
Load immediate	`li Rt,imm`	`lui R1,imm>>16` `ori Rt,R1,imm & 0xFFFF`

```
1 la $t0, 0x10010004:
2 addr:  0001 0000 0000 0001 0000 0000 0000 0100
3          lui $at, 0x1001
4 $at:   0001 0000 0000 0001 0000 0000 0000 0000
5          ori $t0,$at,0x0004
6 $t0:   0001 0000 0000 0001 0000 0000 0000 0100
```

Variable use

```c
int a;
int t[3] = {1,2,3};

int main()
{
    a = t[1];
        exit(0);
}
```

```
.data
a: .space 4
t: .word 1,2,3
.text
main:
        la $t0, t
        lw $t1, 4($t0)
        la $t2, a
        sw $t1, 0($t2)
        ori $v0, $0, 10
        syscall
```

Branch operations

Different kind of branch operations

Unconditional

`j address` : Jump to the address given by a label

`jr $ra` : Jump to the address given by a register (return from function)

`jal address` : Save the next instruction in `$31` and jump to the address given by a label

Conditional, beq,bne,bgtz,blez

`bgtz Rt, address` : if $Rt > 0$ jump to the address given by a label

`beq Rt,Rs,address` : if $Rt == Rs$ jump to the address given by a label

If ... Then ... Else

Pattern

```
1 If( condition)
2         //Instruction if true
3 //Next instructions
```

```
1 If(condition)
2         //Instruction if true
3 else
4         //Instruction if false
5 //Next instructions
```

```
1         Make the condition
2         Branch to next if false
3         Instructions if true
4 next:
5         NextInstructions
```

```
1         Make the condition
2         Branch to TRUE if true
3         Instructions if false
4         J NEXT
5 TRUE:
6         Instructions if true
7 NEXT:
8         NextInstructions
```

If, If ...

```
1           int x = -5;
2           int abs_val;
3
4   main()
5   {
6           abs_val = x;
7           if(x <=0)
8                   abs_val = -x;
9           exit(0);
10  }
```

```
1           int x = -5;
2           int abs_val;
3
4   main()
5   {
6           if(x <=0)
7                   abs_val = -x;
8           else
9                   abs_val = x;
10
11          exit(0);
12  }
```

While loop

Pattern

```
While(condition)
        //Instructions
//Next instructions
```

```
loop:
        Make the condition
        Branch to next if false
        Instructions
        J loop
next:
        NextInstructions
```

```
do
        //Instruction
while condition
//Next instructions
```

```
do:
        Instructions
        Make the condition
        Branch to do if true
next:
        NextInstructions
```

While

```
1           int s = 0;
2           int n = 0;
3
4   main()
5   {
6           do
7       {
8                       s = s + n;
9                       n = n + 1;
10          } while(n < 10);
11          exit(0);
12  }
```

Syscall

Pattern

1. Load the service number in register $v0.
2. Load argument values, if any, in $a0, $a1, $a2, as specified.
3. Issue the SYSCALL instruction.
4. Retrieve return values, if any, from result registers as specified.

Basic services

Service	Code	Args	result
print int	1	$a0: integer to print	
print string	4	$a0 = address of null-terminated string to print	
read int	5		$v0 = integer read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	
exit	10		

Functions

The program follows 6 Steps

1. Place parameters in a place where the function can access
2. Transfer control to the function
3. Acquire the storage resources for the function
4. Perform the task
5. Place the result where the calling program can access it
6. Return control to the point of origin

```
1  main :
2          addi $a0,$0, 5
3          jal succ
4          or  $a0, $v0, $0
5          ori $v0, $0, 1
6          syscall
7          ori $v0, $0, 10
8          syscall
9  succ :
10   addi $v0, $a0, 1
11   jr $ra
```

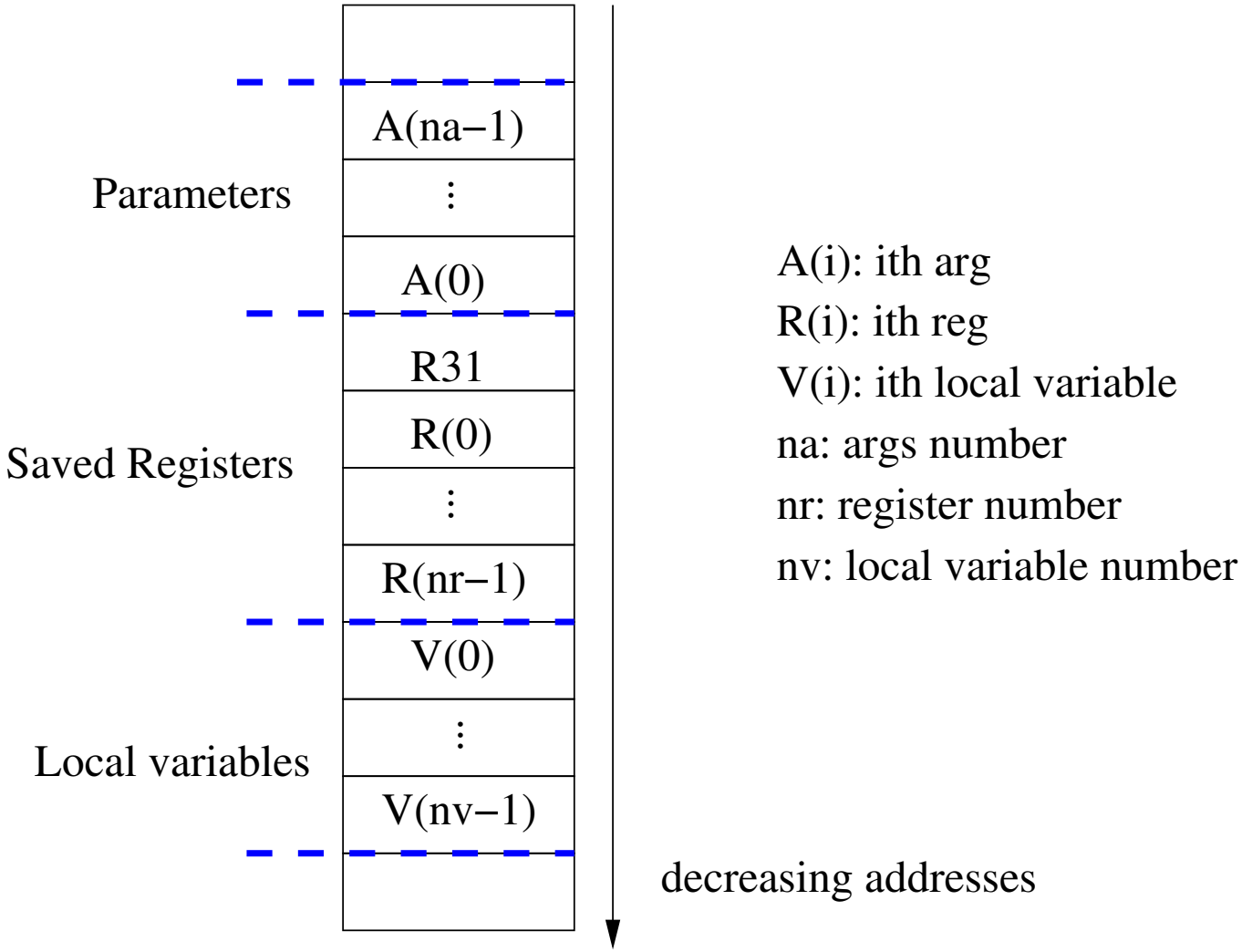
Convention call

Define by the constructor

- Arguments are in $a0..$a3 then in the stack;
- The return address is in $ra;
- The return value is in $v0;
- The previous cited registers (except $ra) and ($s0..$s9) may be modified by the callee and the caller must save them if needed (caller-save registers)
- Registers ($s0..$s8) and $ra has to be saved by the callee (callee-save registers).

Each function has got its own part of the stack: the execution context.

Stack organization



A(i): ith arg
R(i): ith reg
V(i): ith local variable
na: args number
nr: register number
nv: local variable number

Example

```
1  int abs_val(int n)
2  {
3          if (n < 0)
4                  return -n;
5          else
6                  return n;
7  }
8
9  main()
10 {
11         printf("%x",abs_val(-5));
12         exit(0);
13 }
```

The golden rules

Caller

- Decrement the stack pointers according the number of parameters (allocated stack space):
  `$29 <- $29 - 4 x (na + nr)`
- Store the arguments in the stack `$29,$29+4,...`
- Save temporary registers
- Jump to the function and
- Increment the stack pointers according the number of parameters: `$29 <- $29 + (4 x na)`
- Restores temporary registers

Callee

- Prologue: Save registers and local variables
- Instructions
- Epilogue: Restore the value of the saved registers and come back to the caller function.

Function

Prologue

- Decrement the stack pointers according to the number of registers and local variables:
  `$29 <- $29 - 4 x (nv+nr+1)`
- Store $31 and the others registers in the stack
- Load the parameters:
  `$29 + 4 x (nv+nr+1),$29 + 2 x 4 x (nv+nr+1)...`

Epilogue

- Restore the value of the saved registers
- Increment the stack pointer according the number of registers and local variables:
  `$29 <- $29 + 4 x (nv + nr + 1 )`
- Jump to the caller function: `jr $31`

directives.s

```
1  .data
2
3  var1  :   .word 0x1234567
4  var2  :   .word 0x89ABCDEF
5  table:   .byte 1,2,3,4
6            .half 0x32
7  next:    .byte 1,2,3
8           .align 2
9           .byte 12
10          .align 4
11
12 msg1:    .asciiz "Hello"
13 msg2:    .ascii "World"
14          .byte 0xF
15          .align 4
16 array   : .word 42,122,23,6,1337
```

if.s

```
1  .data
2  x :              .word -5          #x
3  abs_val :        .space 4          # abs_val
4
5  .text
6  main :
7          la $t1, x                       #load address of x
8          lw $t2, 0($t1)        # $t2 contains the value of x
9          bgez $t2, next       # if x >= 0 jump
10         sub $t2, $0, $t2    # else
11 next :
12         la $t1, abs_val       # load address of abs_val
13         sw $t2, 0($t1)       # store the result in abs_val
14         ori $v0, $0, 10        # prepare the syscall
15         syscall                      # end
```

ifelse.s

```
.data
x :                 .word -5   #x
abs_val :           .space 4   # abs_val
.text
main :
        la $t1, x               #load address of x
        lw $t2, 0($t1)          # $t2  contains the value of x
        bgez $t2, true          # if x >= 0 jump
        sub $t3, $0, $t2        # $t3 <- (-x)
        j next
true:
        add $t3, $t2, $0        #$t3 <- (x)
next :
        la $t1, abs_val         # loead address of abs_val
        sw $t3, 0($t1)          # store the result in abs_val
        ori $v0, $0, 10         # prepare the syscall
        syscall                 # end
```

while.s

```
1  .text
2  main :
3          ori $s0, $0, 0          #s
4          ori $s1, $0, 0          #n
5
6  do :
7          add $s1, $s1, $s0       #s <- s+n
8          addi $s0, $s0, 1        #n <-n+1
9          slti $t0, $s0, 10       #$t0 = 1 if n == 10
10         bne $t0, $0, do # Jump while not greater
11         ori $v0, $0, 10           # syscall preparation
12         syscall                 #exit(0)
```

function.s

```
1  .text
2  main :
3          addi $t0, $0, -5
4          addiu $sp, $sp, -4   #First args
5          sw $t0, 0($sp)          # A(0) = -5
6          jal abs_val
7          addiu $sp, $sp, 4    #put back the stack pointer
8          or $a0, $v0, $0           #take the function result
9          ori $v0, $0, 1
10         syscall                 #print int
11         ori $v0, $0, 10           #exit
12         syscall
13
14 abs_val :
15         addiu $sp, $sp, -8    #space for registers
16         sw $ra, 4($sp)           # save the return address
17         sw $s1, 0($sp)           # save the $t1
18         lw $s1, 8($sp)            # load A(0)
19         add $v0, $s1, $0         # $v0 had the result
20         bgez $s1, next
21         sub $v0, $0, $s1
22 next :
23         lw $ra, 4($sp)          # restore the previous context
24         lw $s1, 0($sp)
25         addiu $sp, $sp, 8
26         jr $ra                        # return to the main
```

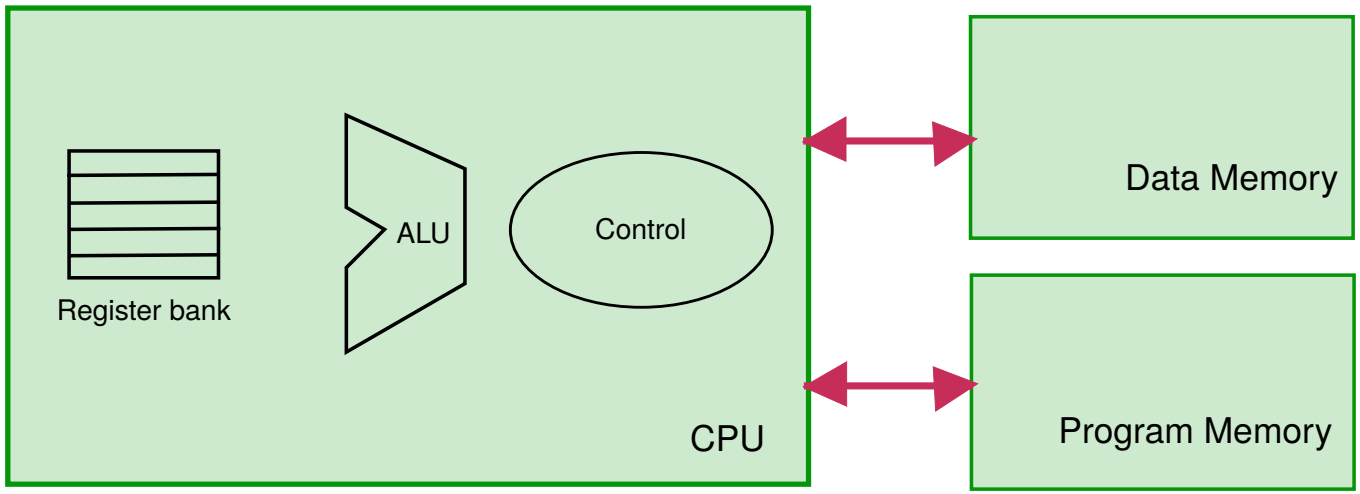
Part III

The instruction set architecture - ISA

External Architecture

Defining the external architecture

- Software visible registers
- Memory addressing
- Instruction set
- Exception and interrupt handlers



The instruction set architecture - ISA

Find a language that:

- Helps to design the hardware architecture
- Helps to design the compiler
- Maximizes the performance
- Minimizes the cost

How the computer see instructions ?

A computer only understands numbers:
Everything has got a number representation

- address
- instructions
- strings
- register's number

Instruction set

Instruction

- It is the way to give orders to the processor.
- It defines which operators apply to which operands.

Instructions types

The MIPS32 has got 57 instructions:

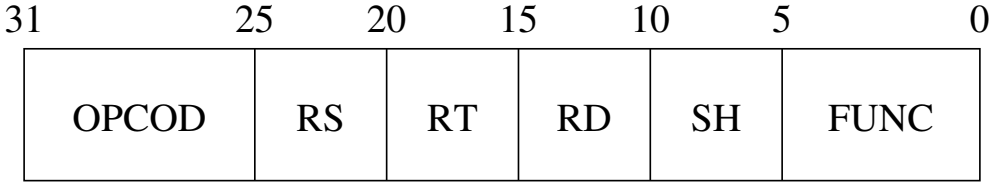
- 33 arithmetic and logic instructions between registers;
- 12 control instructions (branch);
- 7 memory transfer instructions;
- 5 system call instructions.

Defining the kind of operation to perform

An instruction is encoded using 32 bits.

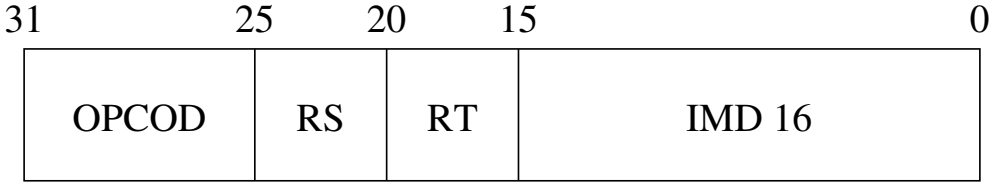
- Form **R**

  Register-register operations: arithmetic/logic operations.



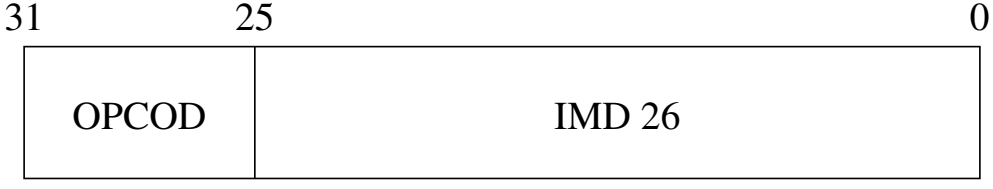
- Form **I**

  Register-immediate operations: read/write memory, conditional branch, artih/logic with a constant.



- Form **J**

  Unconditional branch only



RS: Source register

RT: Transfer register

RD: Destination register

SH: Shift value

IMD: Constant value

Operation Code

INS 31 : 29 \ INS 28 : 26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

BCOND tag

INS 16

INS 20 \	0	1
0	BLTZ	BGEZ
1	BLTZAL	BGEZAL

COPRO tag

INS 23

INS 25 \	0	1
0	MFC0	MTC0
1	RFE	

Special operation Code

INS 5:3 \ INS 2:0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Now, we can speak to a processor !

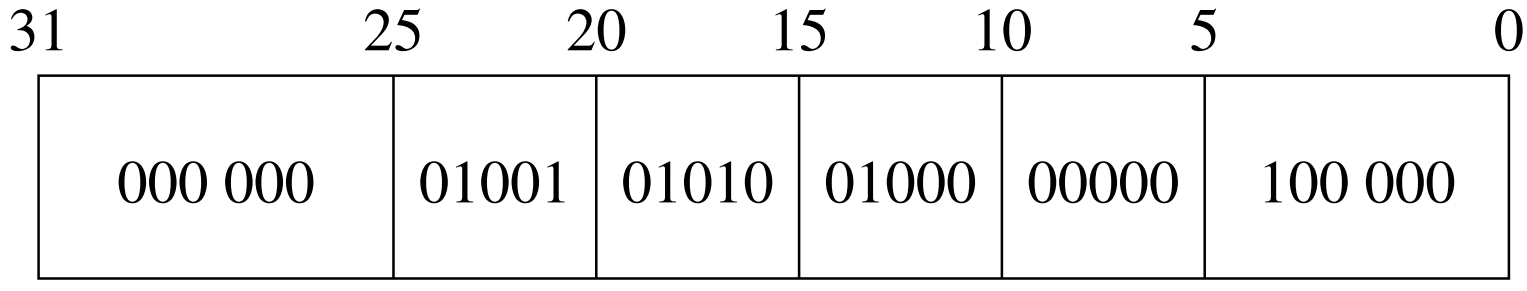
Encode

What is the representation of `add $8,$9,$10`?

Now, we can speak to a processor !

Encode

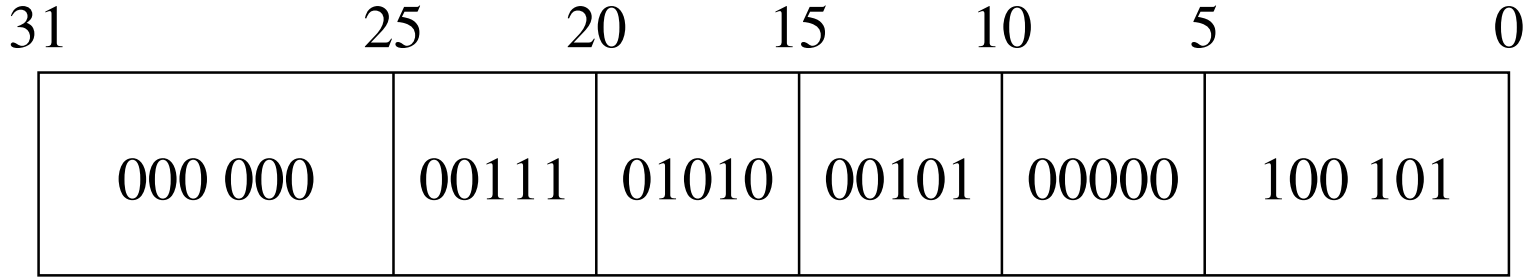
What is the representation of `add $8,$9,$10`?



Now, we can understand a processor !

Question

What does it mean ?



Answer

a. `ori $5,$7,10`

b. `or $7,$5,$10`

c. `or $5,$7,$10`

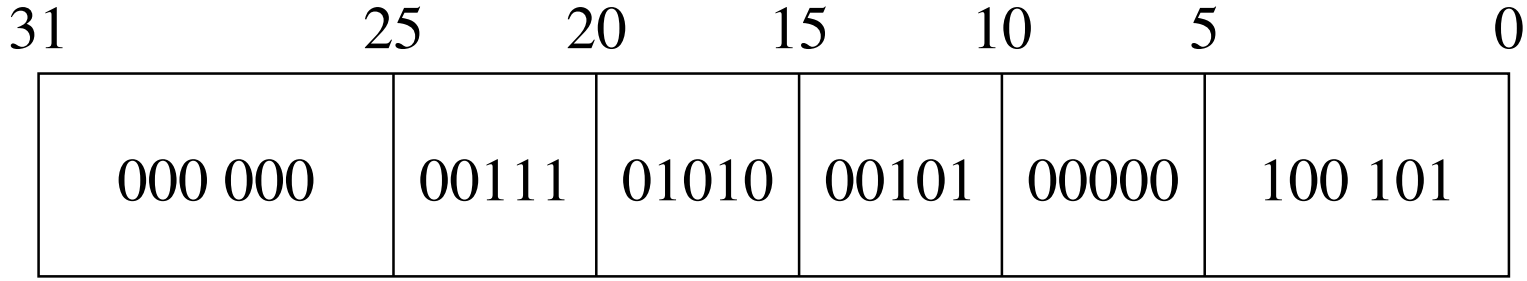
d. `and $5,$7,$14`

e. `add $7,$5,$14`

Now, we can understand a processor !

Question

What does it mean ?



Answer

a. `ori $5,$7,10`

b. `or $7,$5,$10`

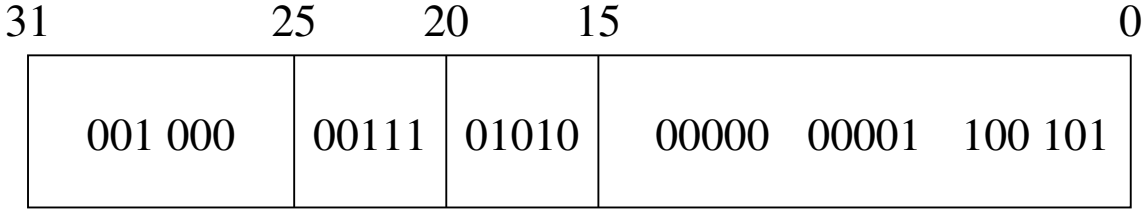
c. `or $5,$7,$10`

d. `and $5,$7,$14`

e. `add $7,$5,$14`

Number representation

Another instruction with immediate



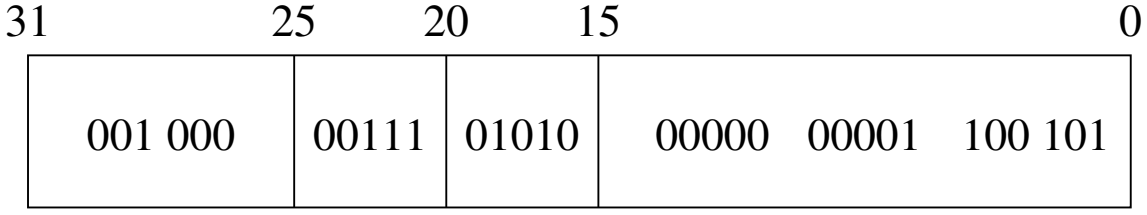
addi $10,$7,0d101

Representing integers

- Sign and magnitude
  - The most significant bit represents the sign.
  - The absolute value is represented by binary digits
- One's complement
  - Complement of the absolute value
- Two's complement
  - Complement of the absolute value
  - Add 1 to the previous result

Number representation

Another instruction with immediate



addi $10,$7,0d101

Representing integers

- Sign and magnitude
  - The most significant bit represents the sign.
  - The absolute value is represented by binary digits
- One's complement
  - Complement of the absolute value
- Two's complement
  - Complement of the absolute value
  - Add 1 to the previous result

Remarks on integer numbers

- All logic operations with immediate are unsigned
- All arithmetic operations with immediate are signed
- To keep the sign a sign extension is made:

```
R10 <- R7 + 0000000000000000 0000000001100101
```

(101_d)

```
R10 <- R7 + 1111111111111111 1111111110011011
```

(-101_d)

Question 6

What is the code for `lw $5, -4($8)`

Answers

	31–26	25–21	20–16	15–0
a.	100 011	01000	00101	11111 11111 11 100
b.	100 011	00101	01000	11111 11111 11 100
c.	101 011	01000	00101	11111 11111 11 100
d.	100 011	01000	00101	00000 00000 00100

Back to the memory

```
1  .data
2          a: .space 4
3          t: .word 1,2,3
4          s: .asciiz "hello
               "
5
6  .text
7  main:
8          la $a0, s
9          ori $v0, $0, 4
0          syscall
1          lw $t1 , -4($t0)
2          la $t2, a
3          sw $t1, ($t2)
4          ori $v0,$0,10
5          syscall
```

	Adress	Value
.data	0x10010014	0x0000006f
	0x10010010	0x6c6c6568
	0x1001000C	0x00000003
	0x10010008	0x00000002
	0x10010004	0x00000001
	0x10010000	0x00000000
.text	0x00400024	0x0000000c
	0x00400020	0x3402000a
	0x0040001C	0xad490000
	0x00400018	0x342a0000
	0x00400014	0x3c011001
	0x00400010	0x8d09fffc
	0x0040000C	0x0000000c
	0x00400008	0x34020004
	0x00400004	0x34240010
	0x00400000	0x3c011001

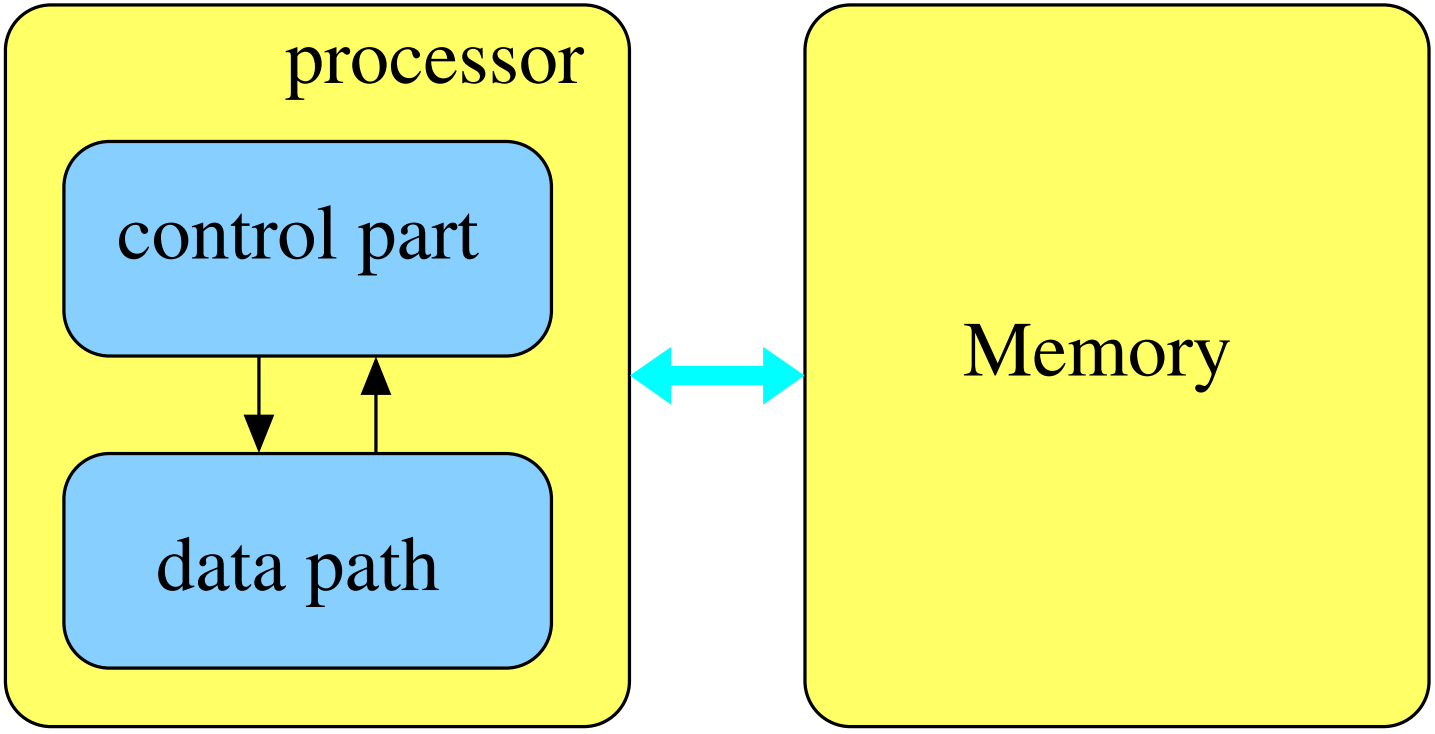
Part IV

The Micro-architecture

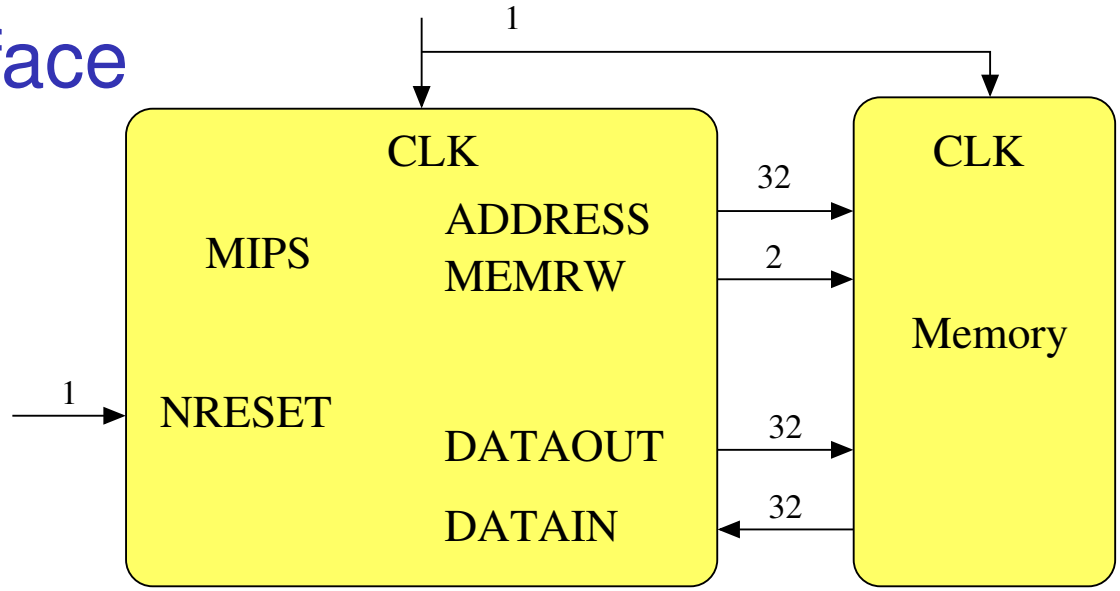
Internal architecture

Micro programmed MIPS32 architecture

- This is nor an optimal version neither the actual version of the MIPS
- Basic techniques for micro-processor
- Basis for pipe-line understanding



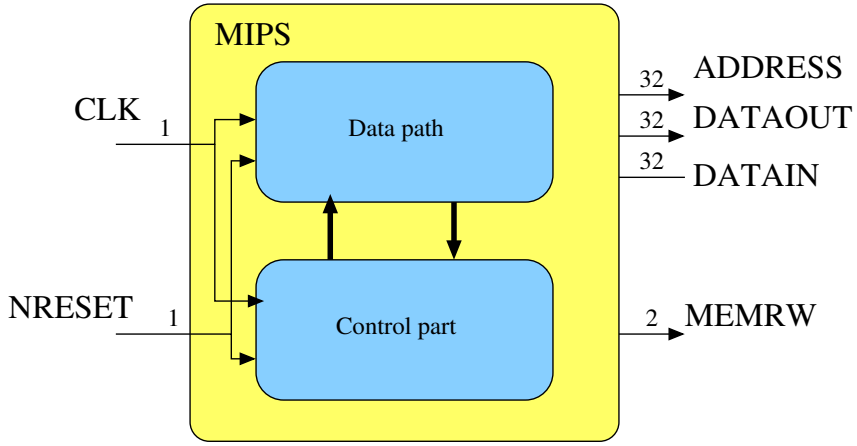
MIPS interface



Ports

- `CLK`: (1 bit) Input clock
- `NRESET`: (1 bit) Synchronous initialization signal
- `ADDRESS`: (32 bits) Address of the selected memory case. The MIPS can access to 2^{32} bytes.
- `MEMRW`: (2 bits) Define the memory access type (read/write/NOP).
- `DATAOUT`: (32 bits) The data to write to the memory.
- `DATAIN`: (32 bits) The data or instruction read from the memory.

Internal architecture



Two parts

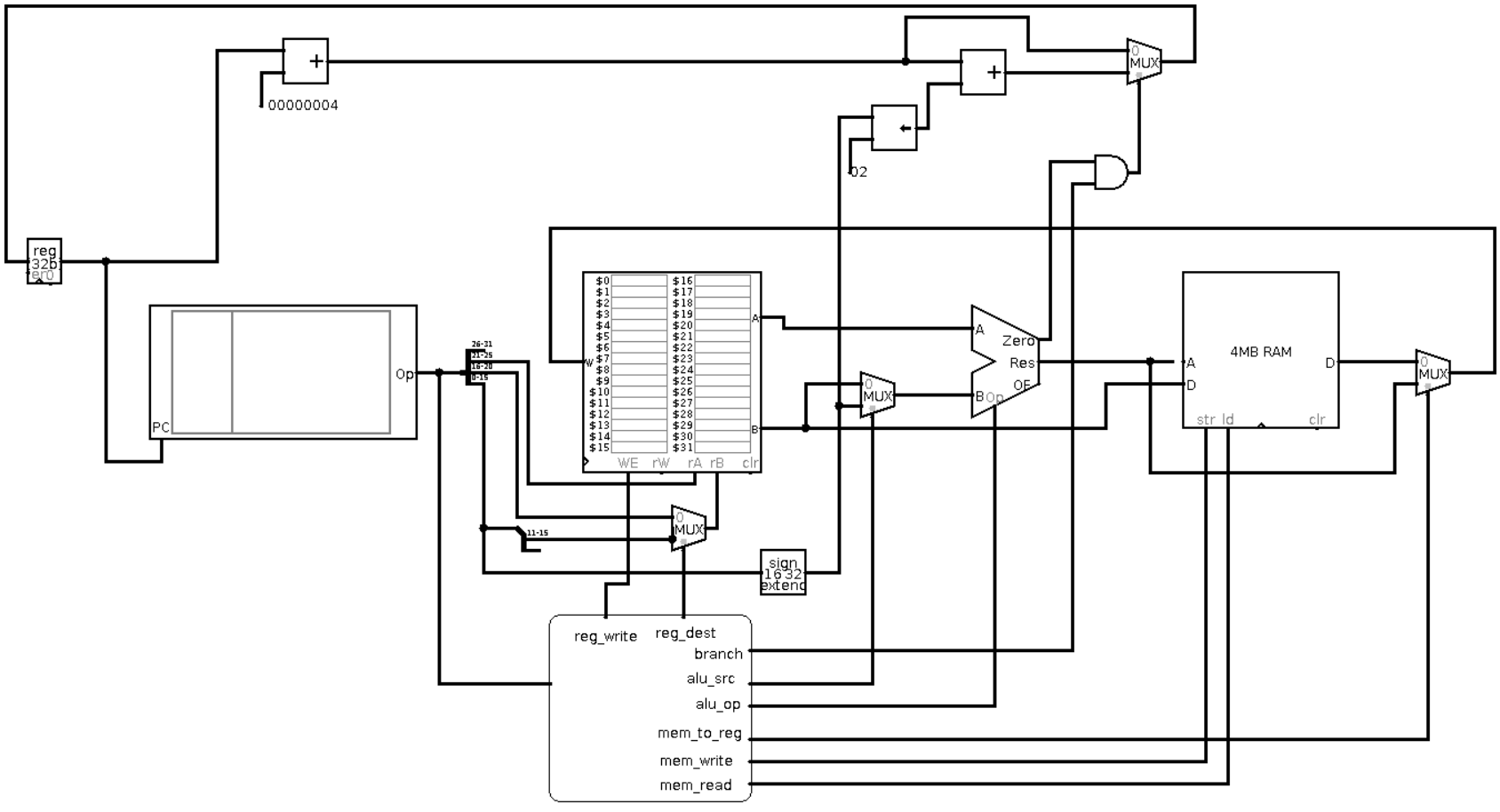
Data Path (operative part)

- Contains: Registers/ALU/Memory access ports
- Realizes: elementary transfer between registers
- One transfer = One cycle
- Controlled by the control part

Control Part (sequential part)

- Finite states automaton
- Defines at each cycle which elementary data transfer to do
- The outputs to the data-path : Micro-instructions
- The input from the data-path : indicators

Processor overview



Multicycle implementation

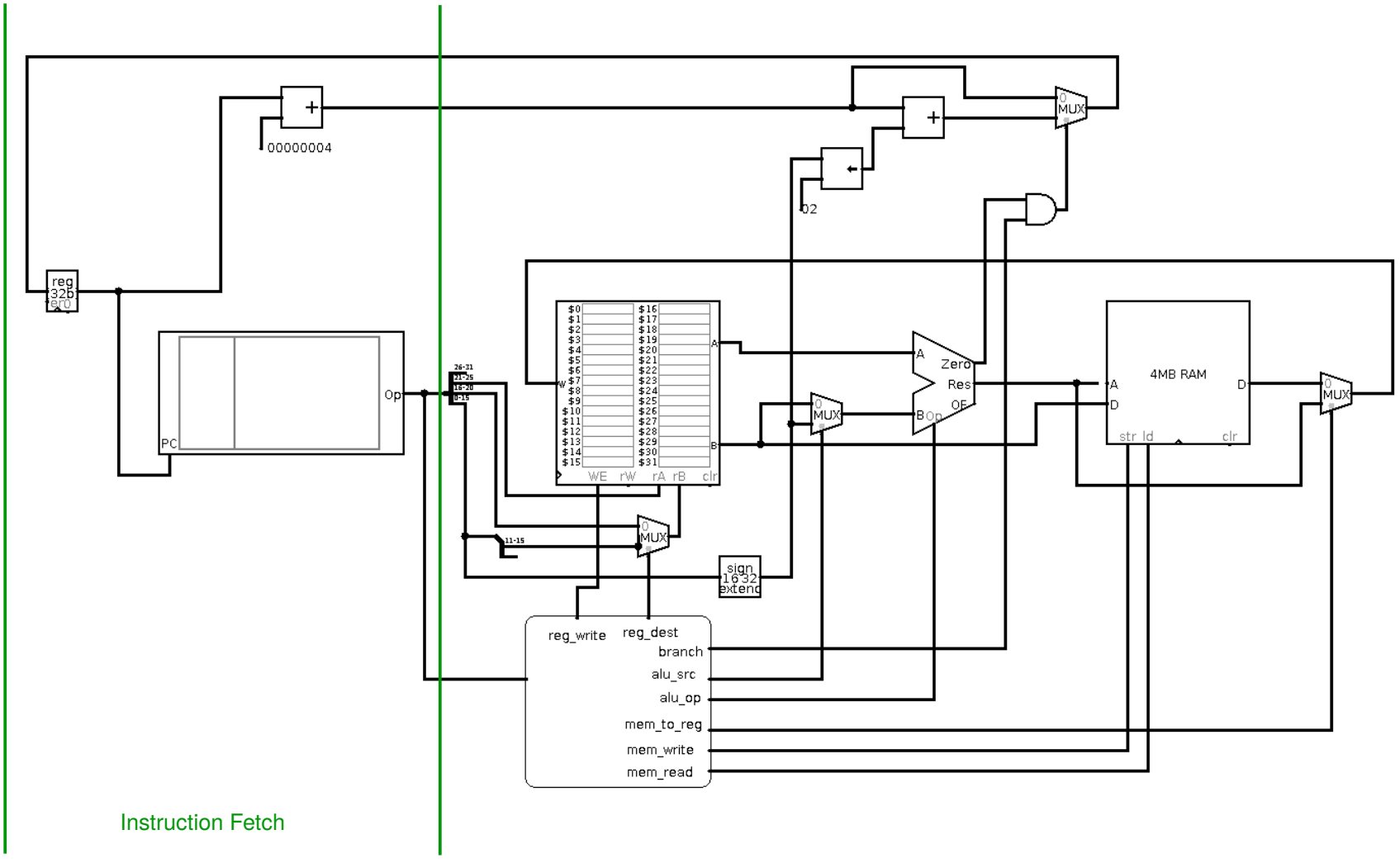
Each step = one clock cycle

- Functional unit may be used more than one time for one instruction.
- Reduce the need of required hardware.
- Add temporary registers.

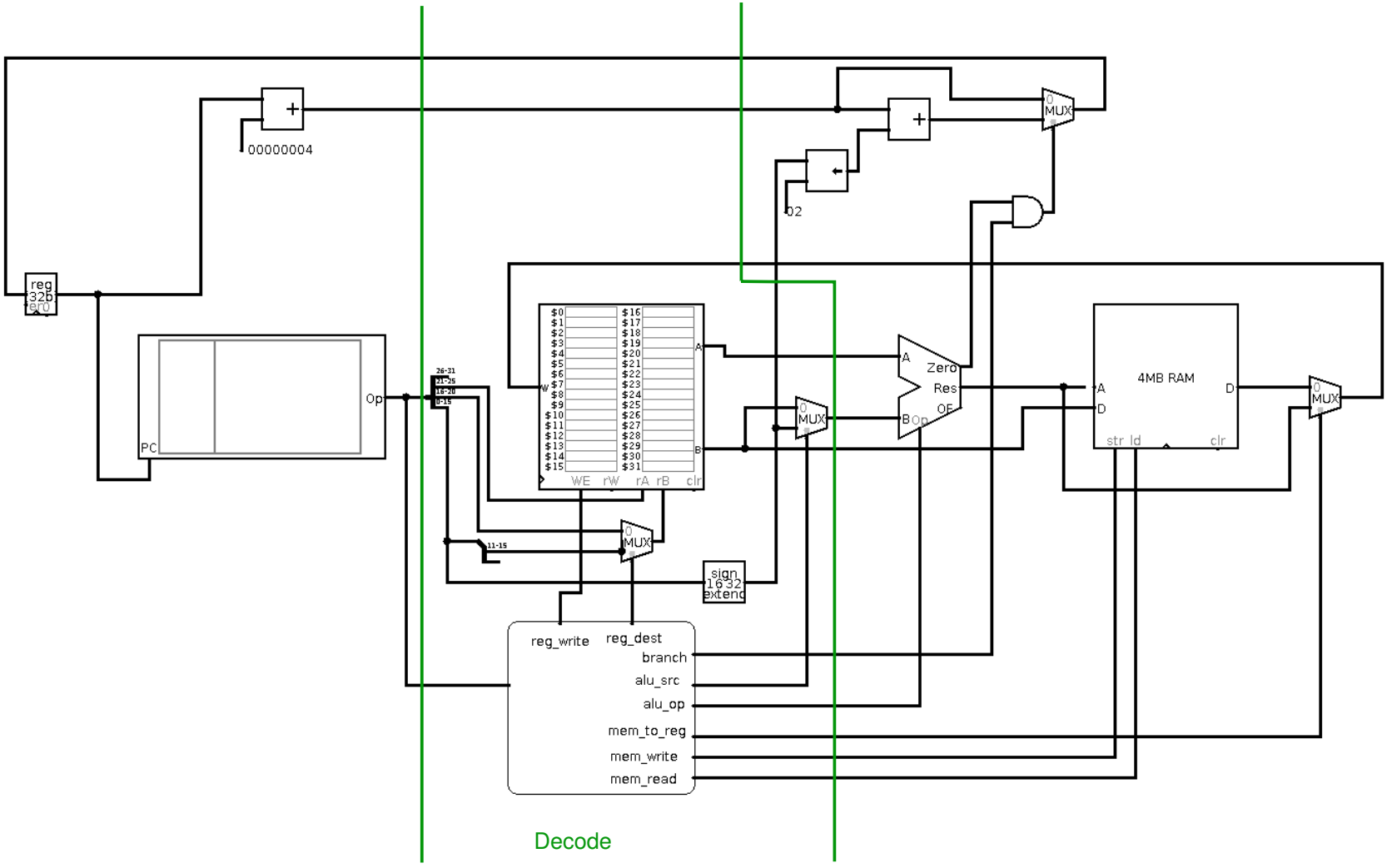
Instruction execution

1. Instruction fetch
2. Instruction decode and register fetch
3. Execution, memory address computation, branch completion
4. Memory access
5. Write back into register bank

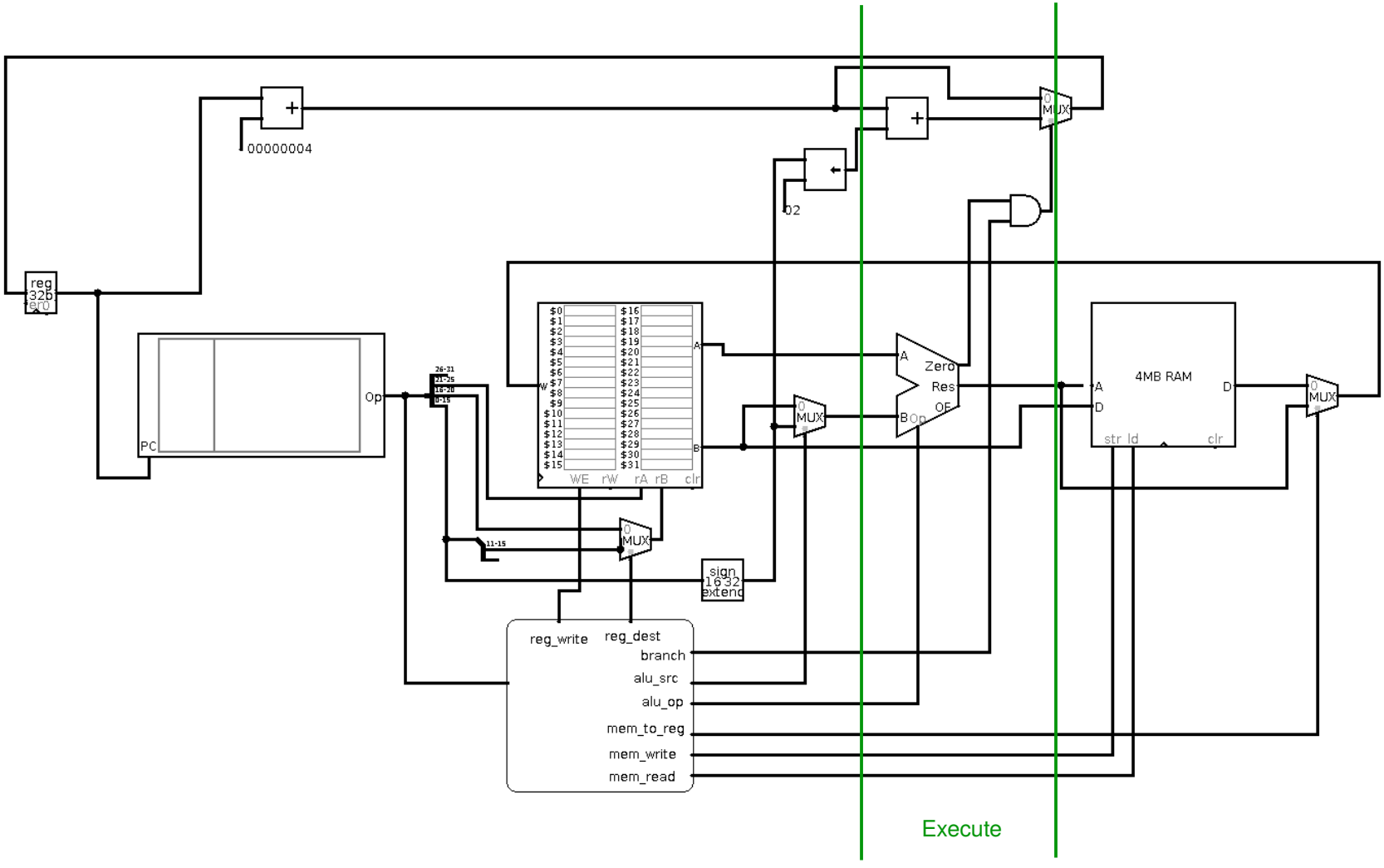
Program Execution



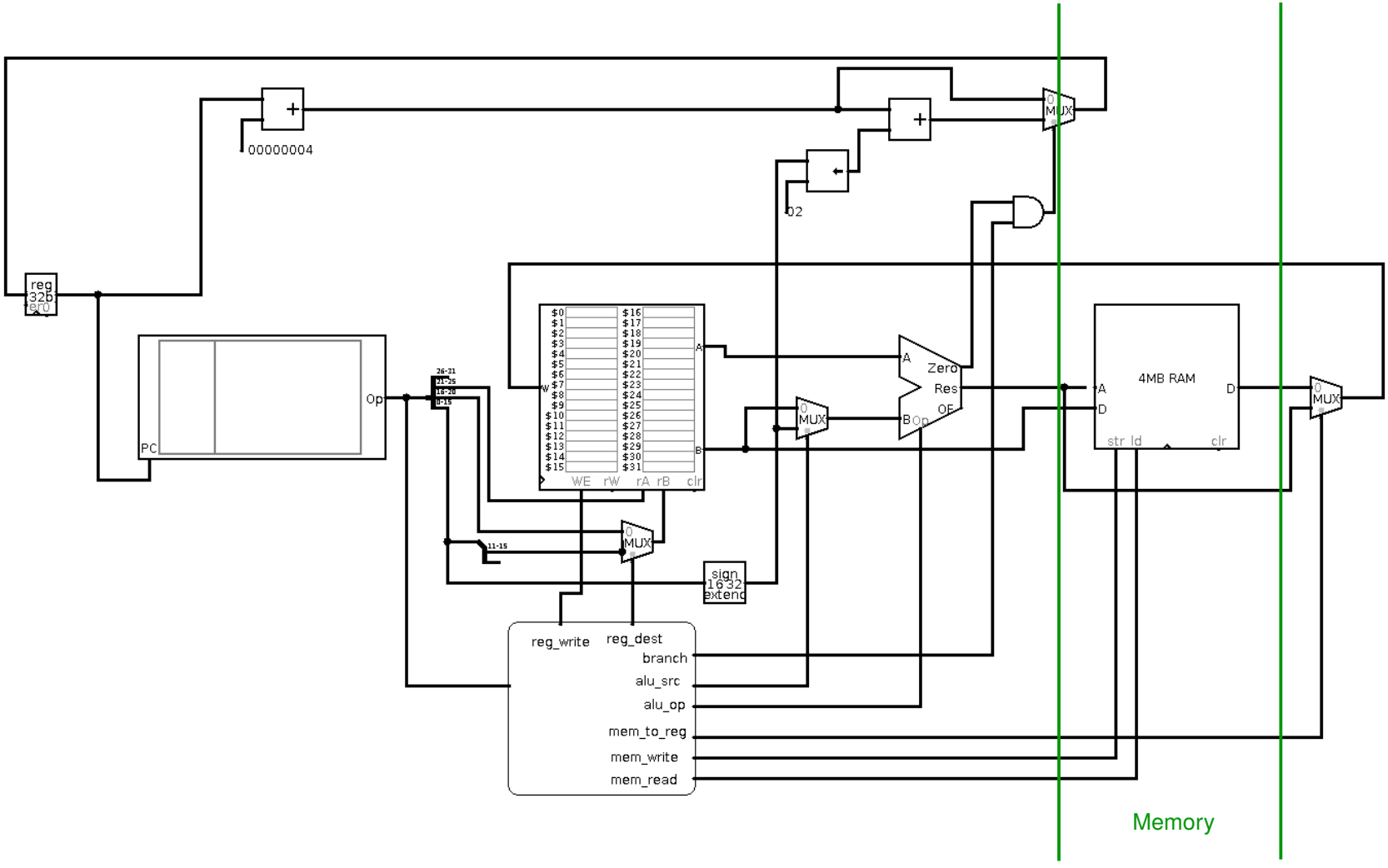
Program Execution



Program Execution



Program Execution



Program Execution