Herbert Wertheim
College of Engineering
UNIVERSITY of FLORIDA

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE

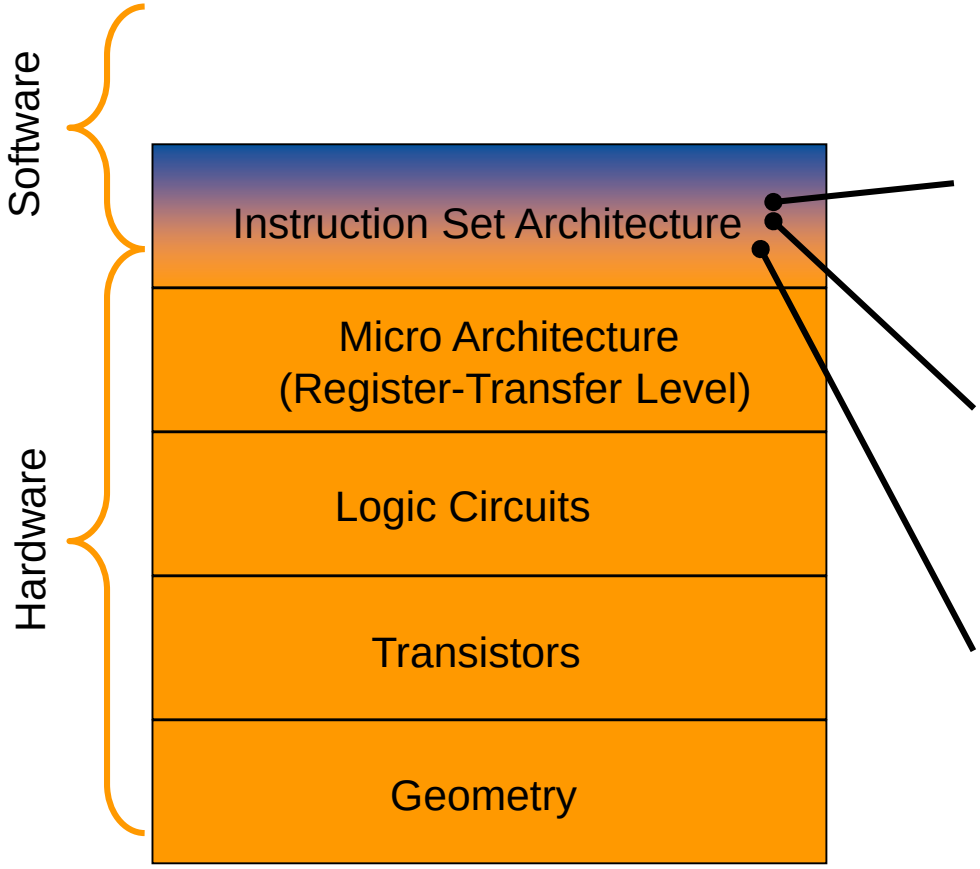
Department of Electrical & Computer Engineering

Digital Logic And Computing Systems

Chapter 08 – Instruction Set Architecture Assembler

Dr. Christophe Bobda

Instruction Set Architecture (ISA)



HW/SW interface:

Define a set of instructions, needed to control the hardware (CPU)

→ instruction set architecture

The number and type of instructions depend on the application field

In general, instructions are designed to:

1. optimize the construction and operation of the underlying hardware
2. make the program readable and easy to understand

Agenda

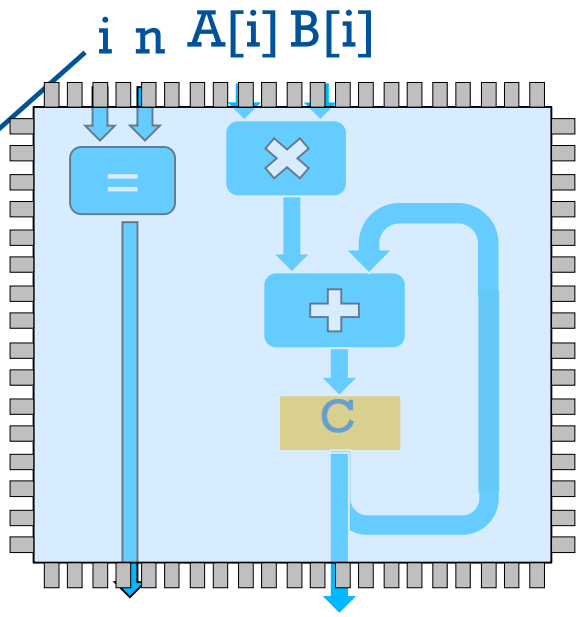
- Arithmetic
- Memory Address and Data Transfer
- Arithmetic Operations with a Constant
- Logic Operations
- Control Flow
- Instruction Coding
- Subprograms
- Addressing Modes

Computer Paradigms

- Domain Specific Computers
  - Application Specific Processor (ASIP)

```
Array A, B: [1:n]
Real c = A*B;


C = 0;
for i=1 to n do
    C = C + A[i]*B[i];
End for;
```

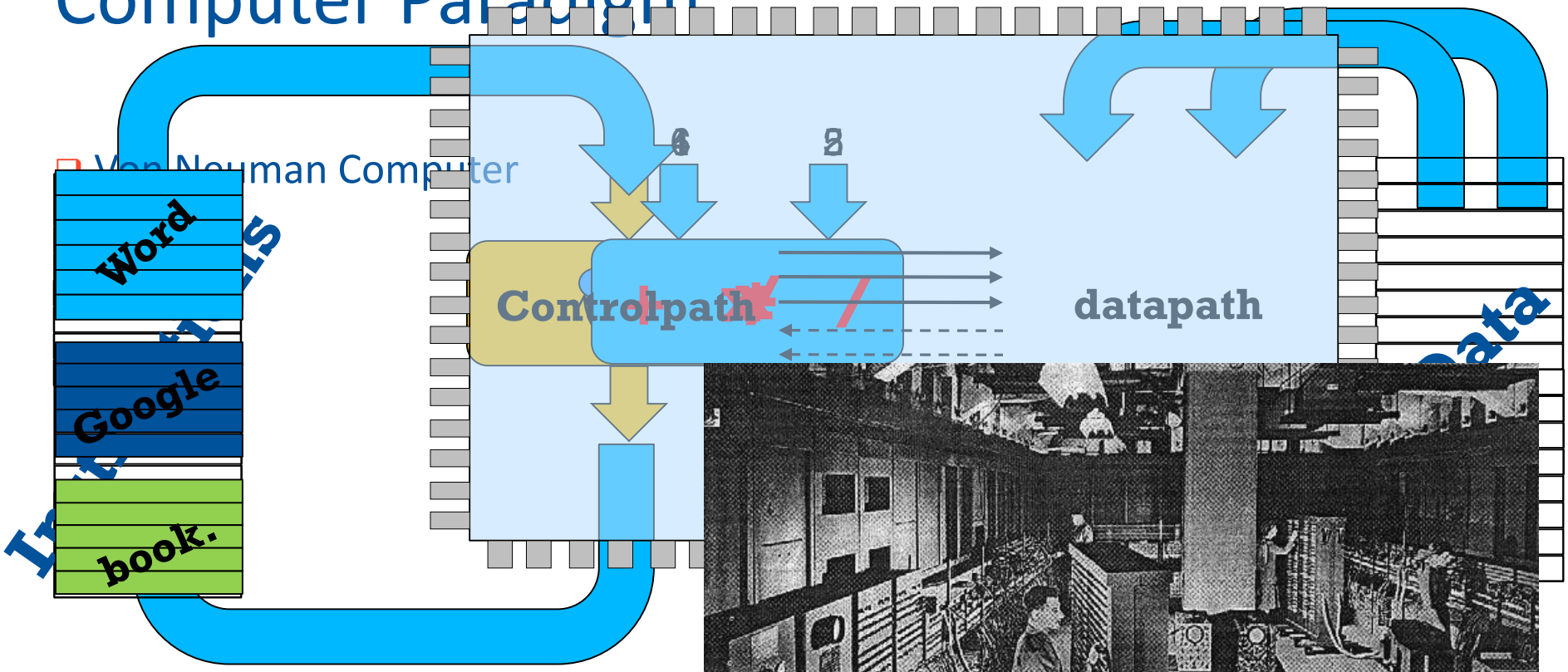


Summary

- Max. Performance
- Min. Flexibility

Computer Paradigm

- Von Neuman Computer



- Summary
  - Max Flexibility
  - Min Performance

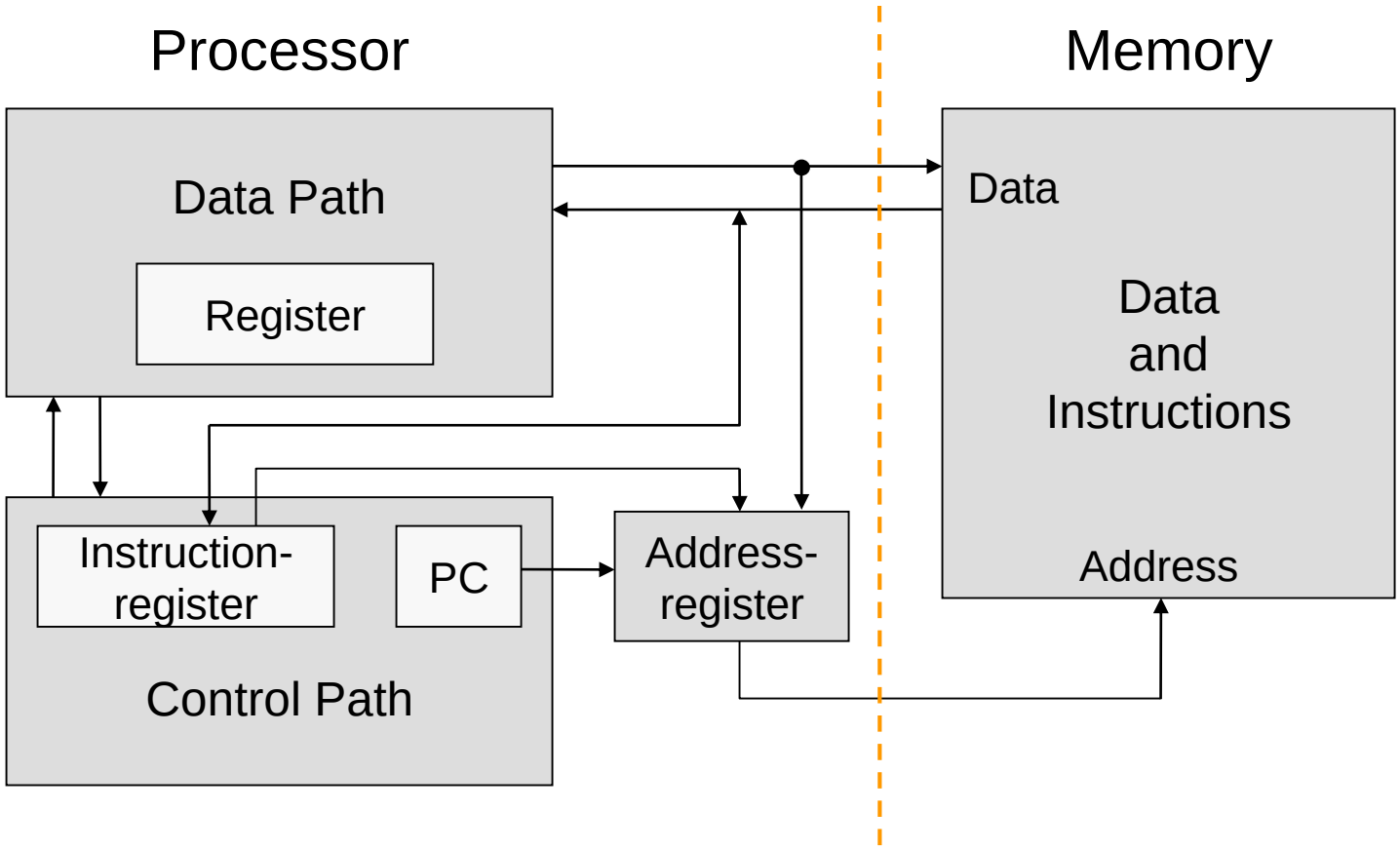
General Purpose Computing

- Examples
  - Microcontrollers
    - Motorolla 68000 familly
    - Intel MCS-51 (8056), MCS 96 (8xC196)
  - Embedded Processor (System on Chip)
    - Apple A series
    - Nvidia Tegra series
    - Qualcom Snapdragon series
    - IBM PowerPC series
  - Soft Cores
    - ARM Cortex
    - Altera Nios, Xilinx Microblazes,

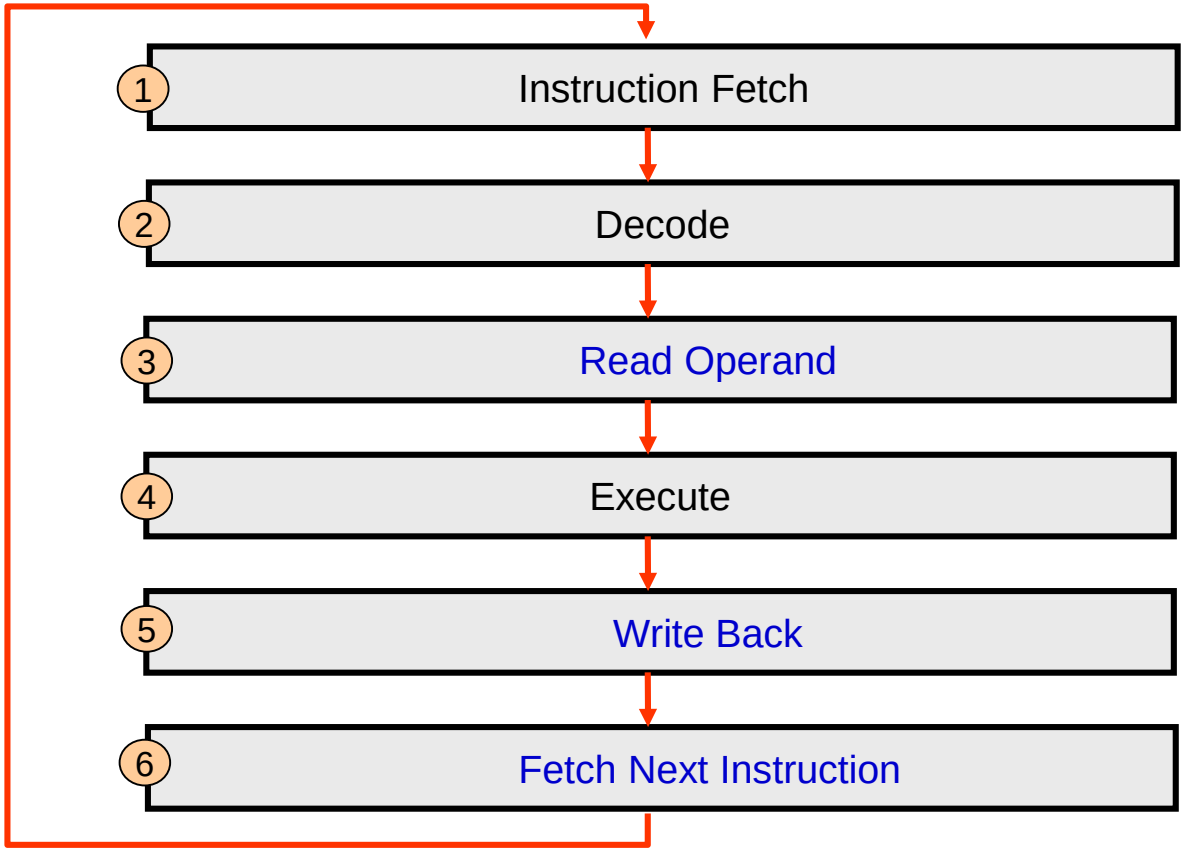
Von Neumann Computer Basic Structure

- A computer consists of
  - Processor + Memory + In/Output
- Memory consists of fixed-length words
  - Data and Instructions
- Processor consists of data path and control path
  - Data path + Control path = Central Processing Unit (CPU)
  - Program Counter (PC), store memory address of the next instruction
  - More Registers in data path
    - Operation on register much faster

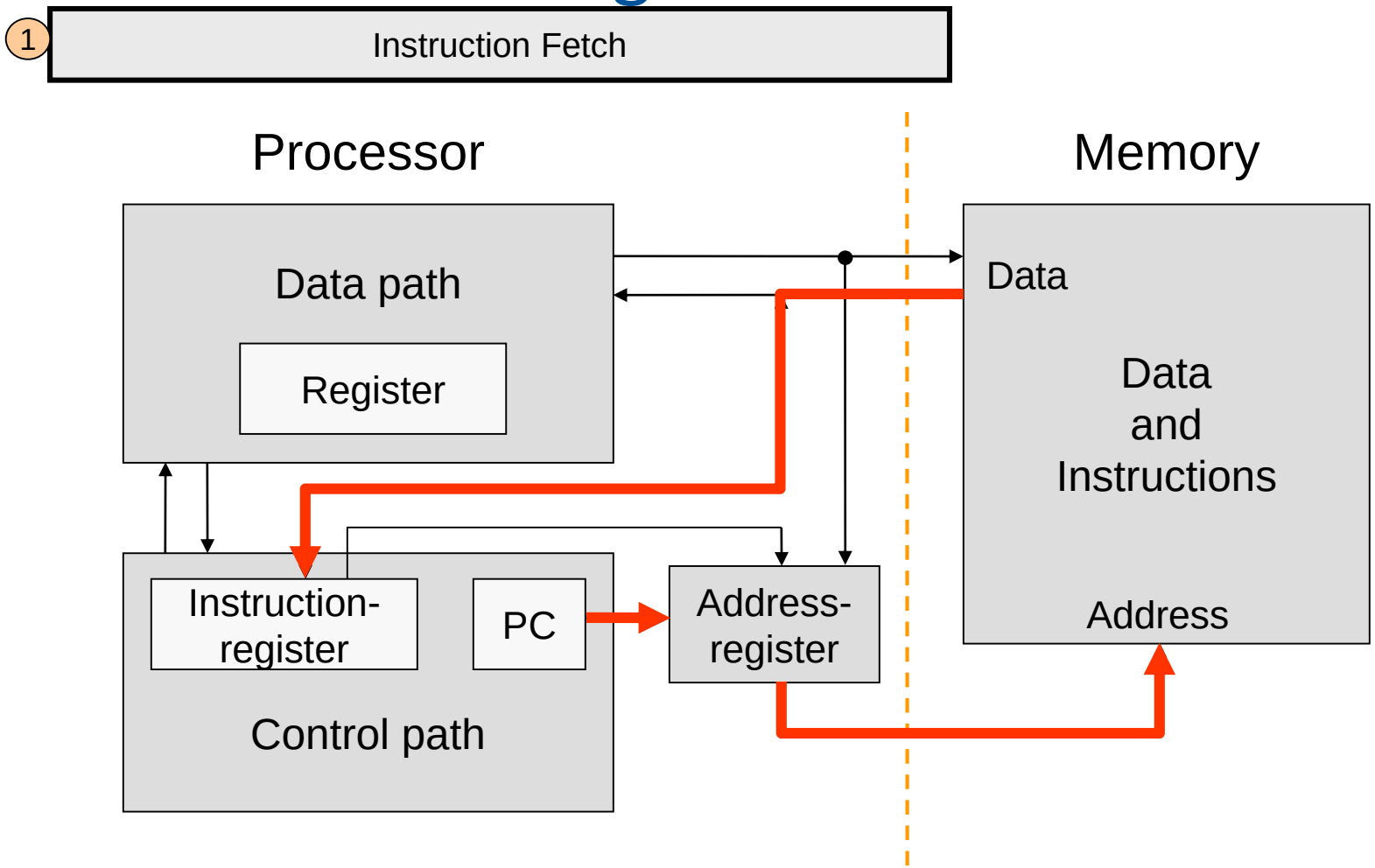
Processor/Central Processing Unit (CPU)



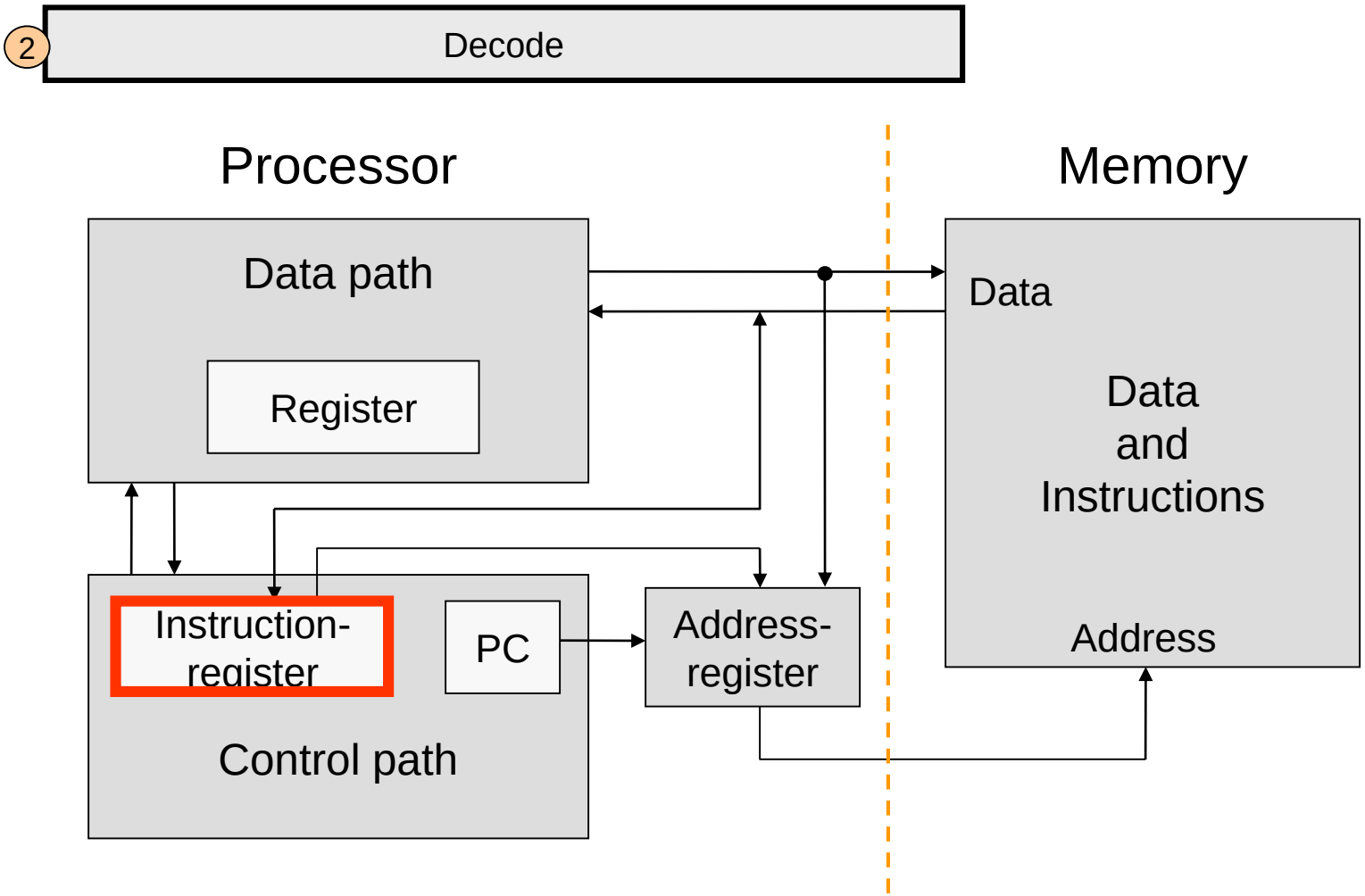
Execution Cycle



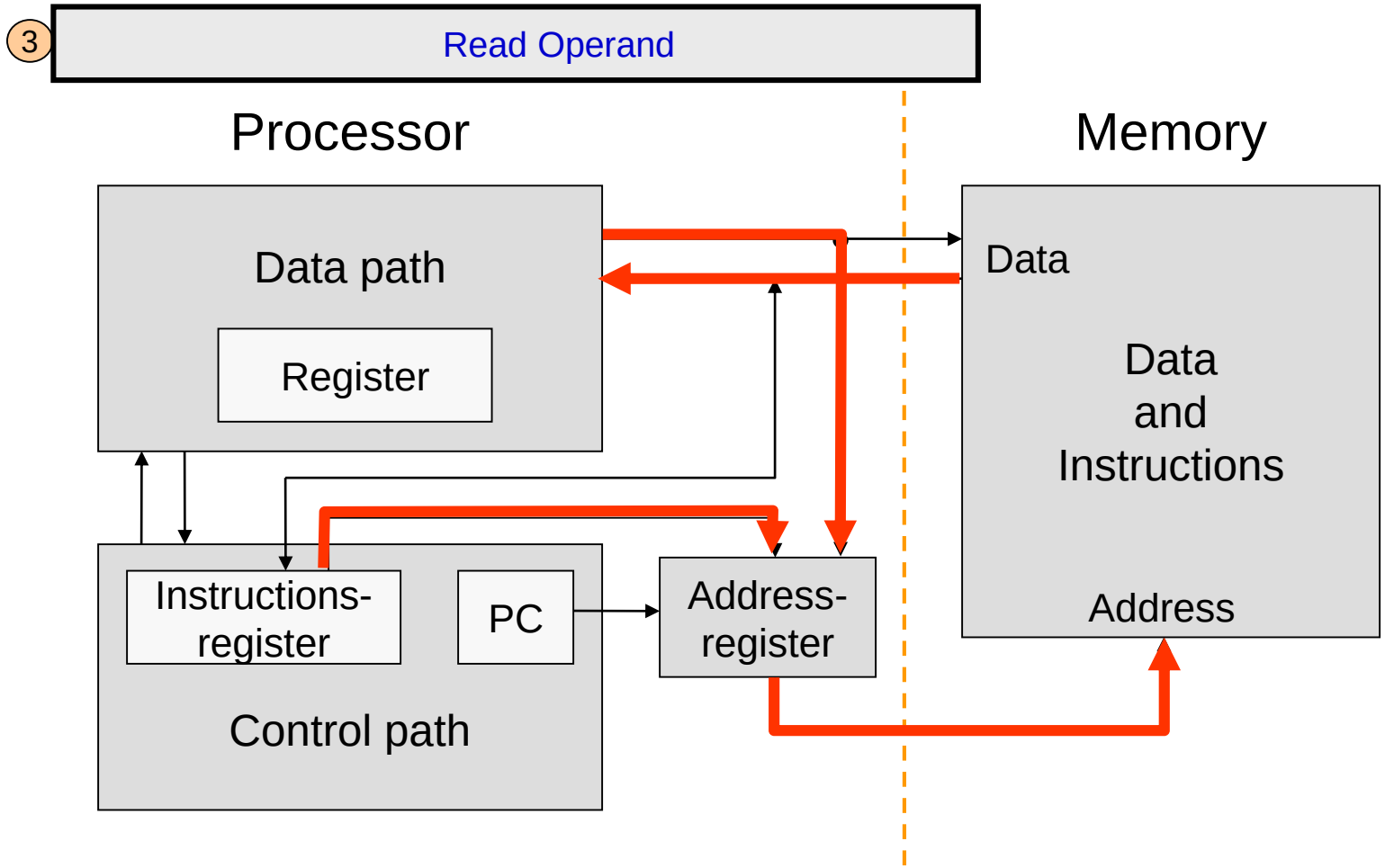
Instruction fetching



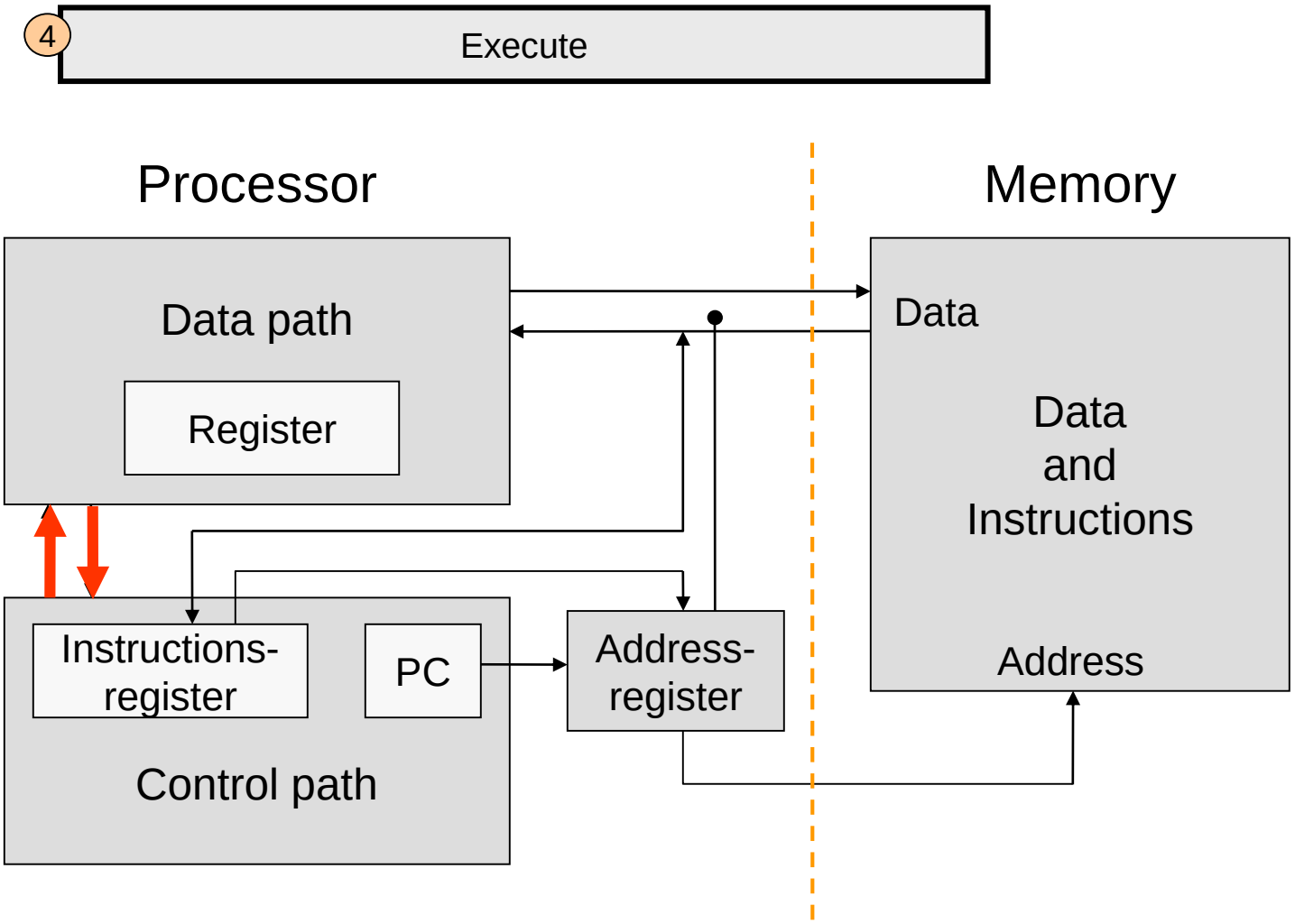
Instruction decode



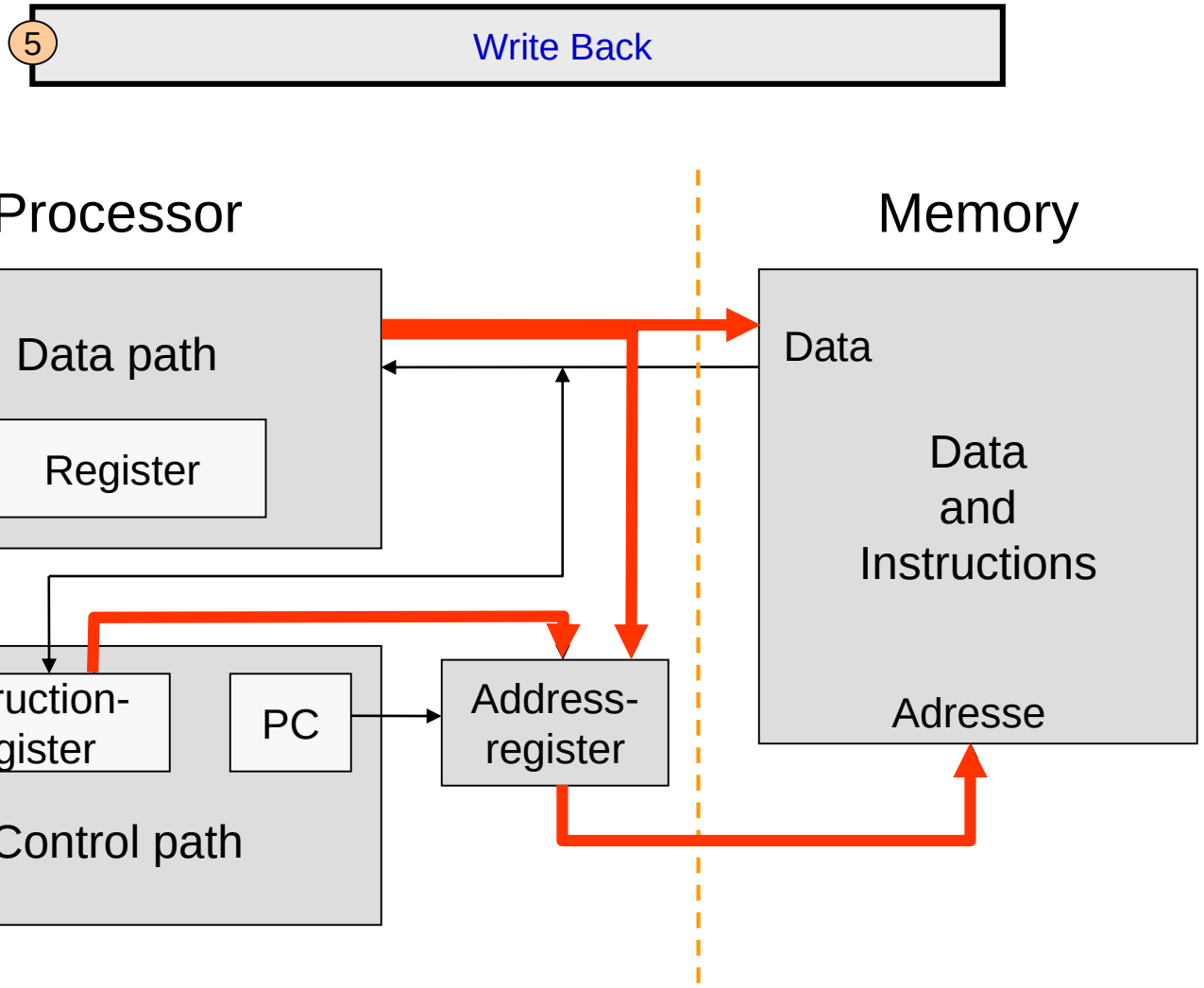
Read Operands



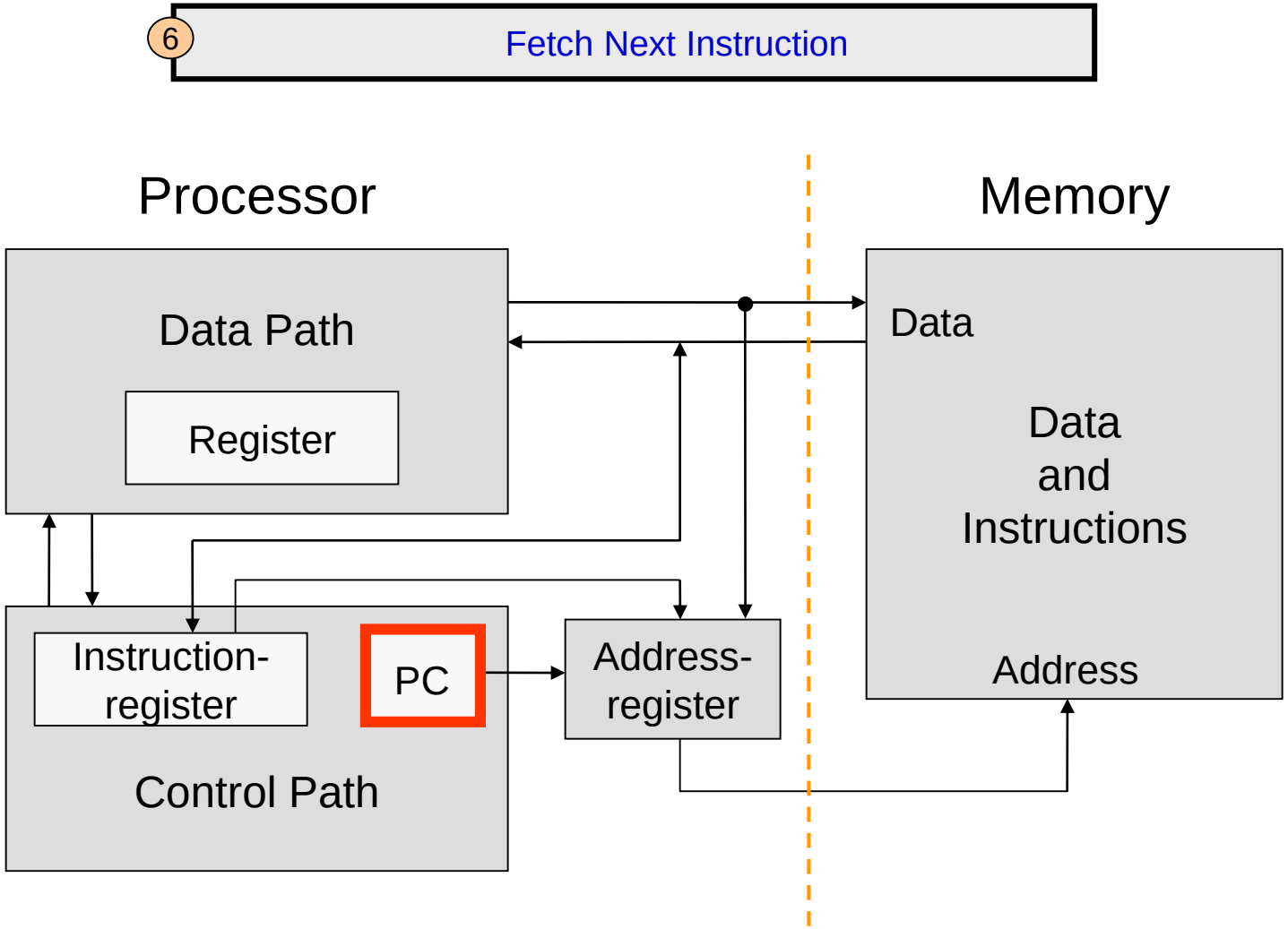
Execute


4
Execute
Processor
Memory
Data path
Register
Data
Data
and
Instructions
Address
Instructions-
register
PC
Address-
register
Control path

Result write back



Next Instruction fetching



MIPS Arithmetic operations

- Arithmetic Basic Instructions

Meaning	MIPS Instruction
a = b + c	`add a, b, c`
a = b – c	`sub a, b, c`

  - All MIPS arithmetic instructions have 3 operands
  - Long expressions are divided by the compiler into small instruction sequences

```
a = b + c + d + e        add  a, b, c
                         add  a, a, d
                         add  a, a, e
```

  - Insertion of temporary variables by the compiler if needed

```
f = (g + h) – (i + j)    add  t0, g, h
                         add  t1, i, j
                         sub  f, t0, t1
```

MIPS Arithmetic operations

- MIPS arithmetic operations can be executed only with registers
  - 32 general purpose registers, each with 32 bits
  - Datatype with 32-bit is called a word
  - A convention defines how registers are named and how they should be used
    - 8 registers for variables of the source code: $s0, $s1,..., $s7
    - 8 registers for temporary variables: $t0, $t1,..., $t7
    - Compiler (assembler programmer) doesn't have to know this convention
    - Convention is important for assembled programs to work together

MIPS Arithmetic operations

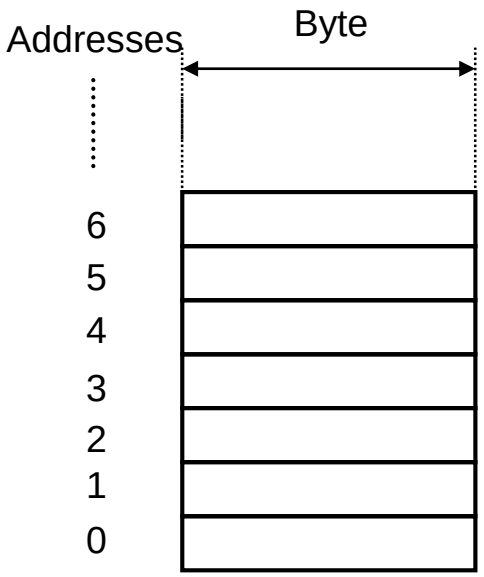
- Example:
  - Assembling the following java (C/C++)- instructions:
    `f = (g + h) – (i + j)`
  - Assumption: variables `g, h, i and j` are in registers `$s1, $s2, $s3, $s4`. Result `f` will be store in register `$s0`.

```
add   $t0, $s1, $s2   # $t0=g+h
add   $t1, $s3, $s4   # $t1=i+j
sub   $s0, $t0, $t1   # f=(g+h)-(i+j)
```

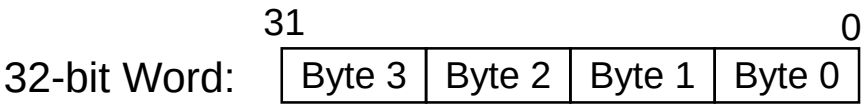
  - The symbol # is used to place comments, from # until the end of the line
  - Limited number of registers: not enought to hold all variable of a program
    → variables, program, arrays are store in memory

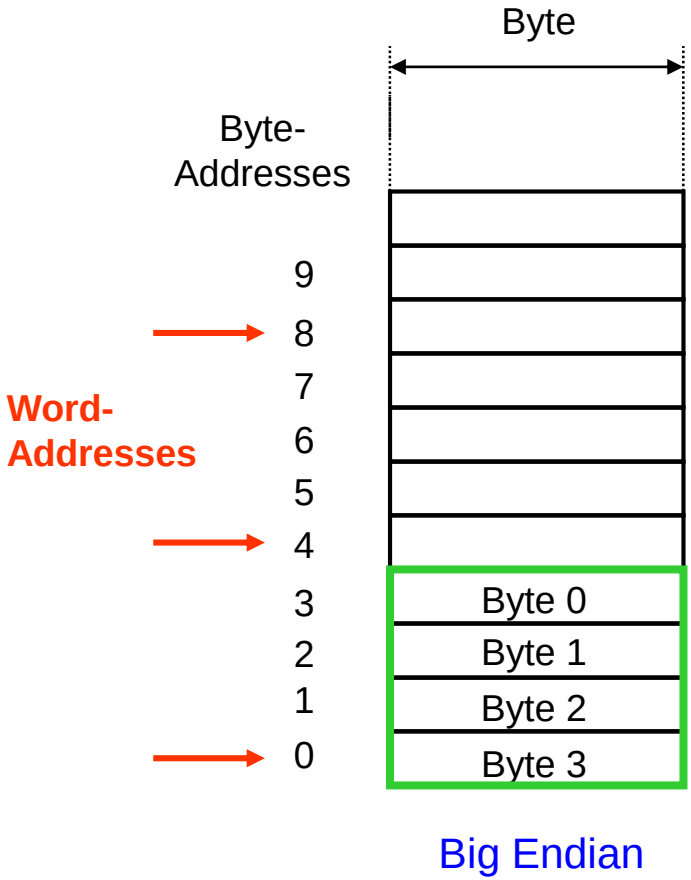
MIPS Memory Addressing

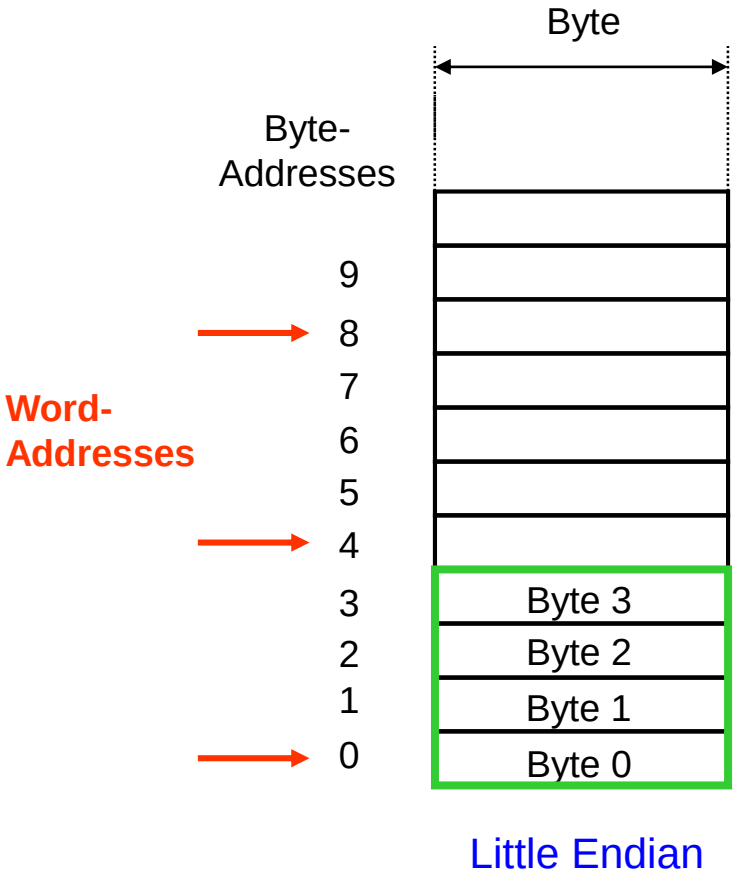
- Memory is addressed byte-wise
  - The memory ≈ large 1-dimensional array of bytes
  - Lowest address is 0
  - How is a word (4 Bytes) stored in memory?
    - MIPS uses "Big-Endian" convention:
      - MSB of the word is at the lowest by address
      - A word is accessed with the address of it's highest byte
    - Alignment restriction: Word address must be a multiple of 4



Big Endian vs. Little Endian



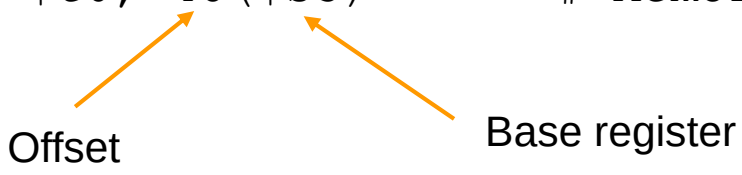




MIPS Data Transfer Operations

- Data can be copied between memory and registers only with data transfer instructions
  - `lw` (load word)
  - `sw` (store word)
- Example: java (C/C++)-Instruction: `A[12] = h + A[8]`
  - Assumption: `A` is an array of words. The variable `h` is in register `$s2`, the base address of `A` is register `$s3`

```
lw   $t0, 32($s3)      # $t0=Memory[$s3+32]
add  $t0, $s2, $t0     # $t0=h+A[8]
sw   $t0, 48($s3)      # Memory[$s3+48]=$t0
```



MIPS Arithmetic Operationen

- How do we perform the arithmetic operation `a = a + 3` ?
  - We can store the constant in memory, then use the `lw` to copy in register
  - Assumption: variable `a` is in register `$s1`. Constant `3` in memory at address store in `$t1`

```
lw   $t0, 0($t1)      # $t0=3
add  $s1, $s1, $t0    # a=a+3
```

  - Because arithmetic with constants is so common, MIPS ISA has a special instruction for that:

`addi` (add immediate)

```
addi  $s1, $s1, 3       # $s1=$s1+3
```

MIPS Logic operations

- Logic operations can only be executed with registers
- Shift operations

  `sll` (shift left logical)

  `srl` (shift right logical)

```
sll $t2, $s0, 4    # $t2=$s0 << 4 bit
```

- Logic operation

  `and` (bitwise AND)

  `andi` (bitwise AND immediate)

  `or` (bitwise OR)

  `ori` (bitwise OR immediate)

  `nor` (bitwise NOR

MIPS Logic Operations

- Example
  - Assumption: `$s0` store value `0x00000009`

```
sll   $s1, $s0, 2    # $s1=0x00000024=36_10= 9 x 4
or    $s1, $s1, $s0  # $s1=?
andi  $s1, $s1, 15   # $s1=?
andi  $s2, $s2, 0    # $s2=?
nor   $s2, $s2, $s1  # $s2=?
```

MIPS Branching

- Control Flow
  - Machine/Assembler programs consist of a sequence of instructions
  - Instruction are executed sequentially
  - The program counter is incremented by 4 (word address) for each instruction
  - Branch instructions are used to change program flow (if-then-else, loop)
- Conditional branch

  `beq` (branch if equal)

  `bne` (branch if not equal)
- Unconditional jump

  `j` (jump)

Translation `if-then-else`

- Example:
  - C-Program:

```
if (i==j) {
    f = g + h;
} else {
    f = g - h;
}
```

  - Assembler Program:

    Assumption: variables `f, g, h, i, j` in `$s0,$s1,$s2,$s3` and `$s4`

```
        bne   $s3, $s4, Else    # if (i!=j) goto Else
        add   $s0, $s1, $s2     # f=g+h
        j     Exit               # goto Exit
Else:   sub   $s0, $s1, $s2     # f=g-h
Exit:   ...
```

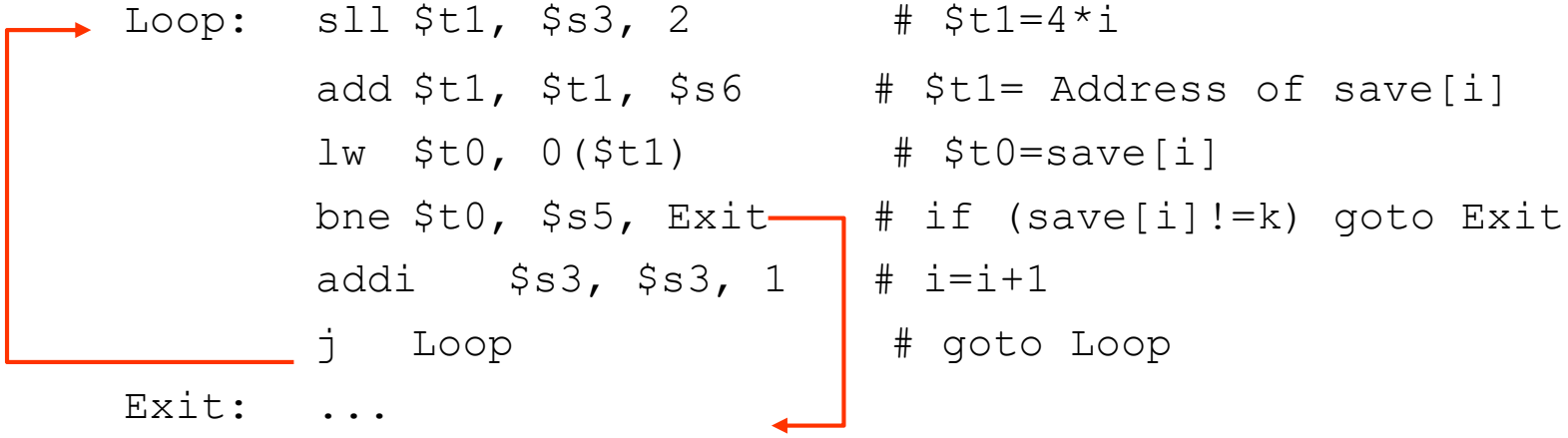
  - `Else, Exit` are Labels

Loop Translation

- Example:
  - C-Program:

```
while
   (save[i]=k){
   i = i + 1;
}
```

  - Assembler Program:
  - Assumption: variables `i, k` in `$s3, $s5`. Base address of `saved` in `$s6`



```
Loop:   sll $t1, $s3, 2      # $t1=4*i
        add $t1, $t1, $s6    # $t1= Address of save[i]
        lw  $t0, 0($t1)      # $t0=save[i]
        bne $t0, $s5, Exit   # if (save[i]!=k) goto Exit
        addi   $s3, $s3, 1   # i=i+1
        j   Loop             # goto Loop
Exit:   ...
```

MIPS Branching

- MIPS has no branching instruction that checks the conditions smaller than or bigger than. Instead, we use:

  `slt` (set less than)

  `slti` (set less than immediate)

```
slt $t0, $s3, $s4 # if ($s3<$s4) $t0=1 else $t0=0
```

- MIPS has a special register: `$zero`
  - `$zero` is permanently set to 0. Assignments to `$zero` are ignored
  - With register `$zero` and instructions `beq, bne, slt` and `slti` we can implement various conditions

Instruction Format

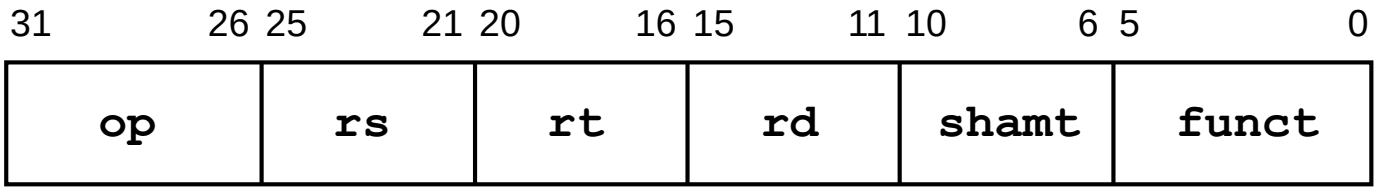
- Instructions in assembler must be translated in machine instructions to be processed by the CPU
  - Instructions must be code in binary → Instruction coding

```
add   $t0, $s1, $s2   → 0x00001000110010010000000100000
```

- All MIPS instructions have 32 bits
  - Because different instructions have different number of operand, instructions are divided in three categories
    - R-Type Instructions
    - I-Type Instructions
    - J-Type Instructions

Instruction Format - R-Type

- Instruction format R-Type (Register-Format) is used for arithmetic and logic instructions



`op`	Operation code (OP-Code)
`rs`	First operand register
`rt`	Second operand register
`rd`	Result register
`shamt`	Shift amount
`funct`	Function, options for one operation

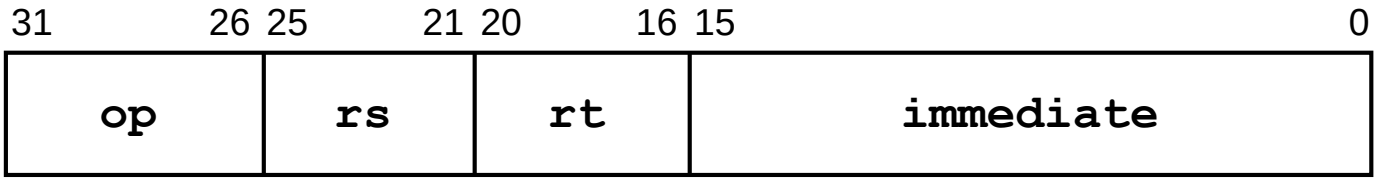
- Example:

```
add  $t0, $s1, $s2
```

000000	10001	10010	01000	00000	100000

Instruction Format - I-Type

- Instructions format I-Type (Immediate-Format) is used for:
  - Immediate version of arithmetic and logic operations,
  - Data transfer instructions and conditional branching



`immediate` Constant or Address

- Immediate value is a 16-bit signed number in two's complement
  - Values between -2^{15} and $+2^{15}-1$

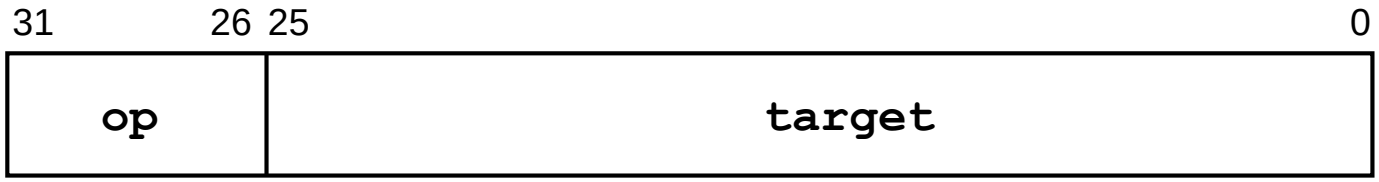
- Example:

```
sw  $s1, -4($s2)
```

101011	10010	10001	1111111111111100

Instruction Format - J-Type

- Instruction format J-Type (Jump-Format) is used for unconditional jump



`target` branch address

- The target is a 26-bit number that will be interpreted as word address

Procedure (Subprograms)

- Execution
  - The control is transferred to the procedure (callee)
  - The calling procedure executes until completion
  - The control is transfer back to the main program (caller)
- MIPS-support for procedure (subprogram) execution
  - Special register to secure the return address: `$ra`
  - Jump instruction

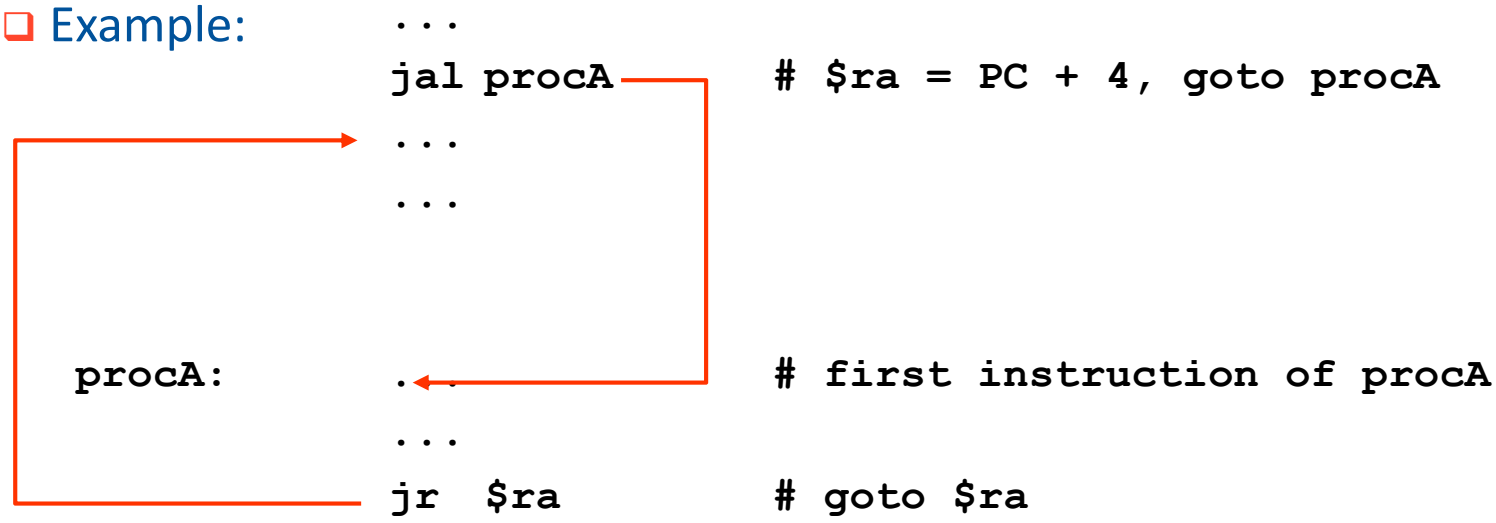
    `jal` (jump and link) save return address in `$ra` and jumps to start address of the procedure

    `jr` (jump register) jumps to the provided in the register

    `jr $ra` jump to the return address in the main program

Procedure (Subprograms)

- Example:

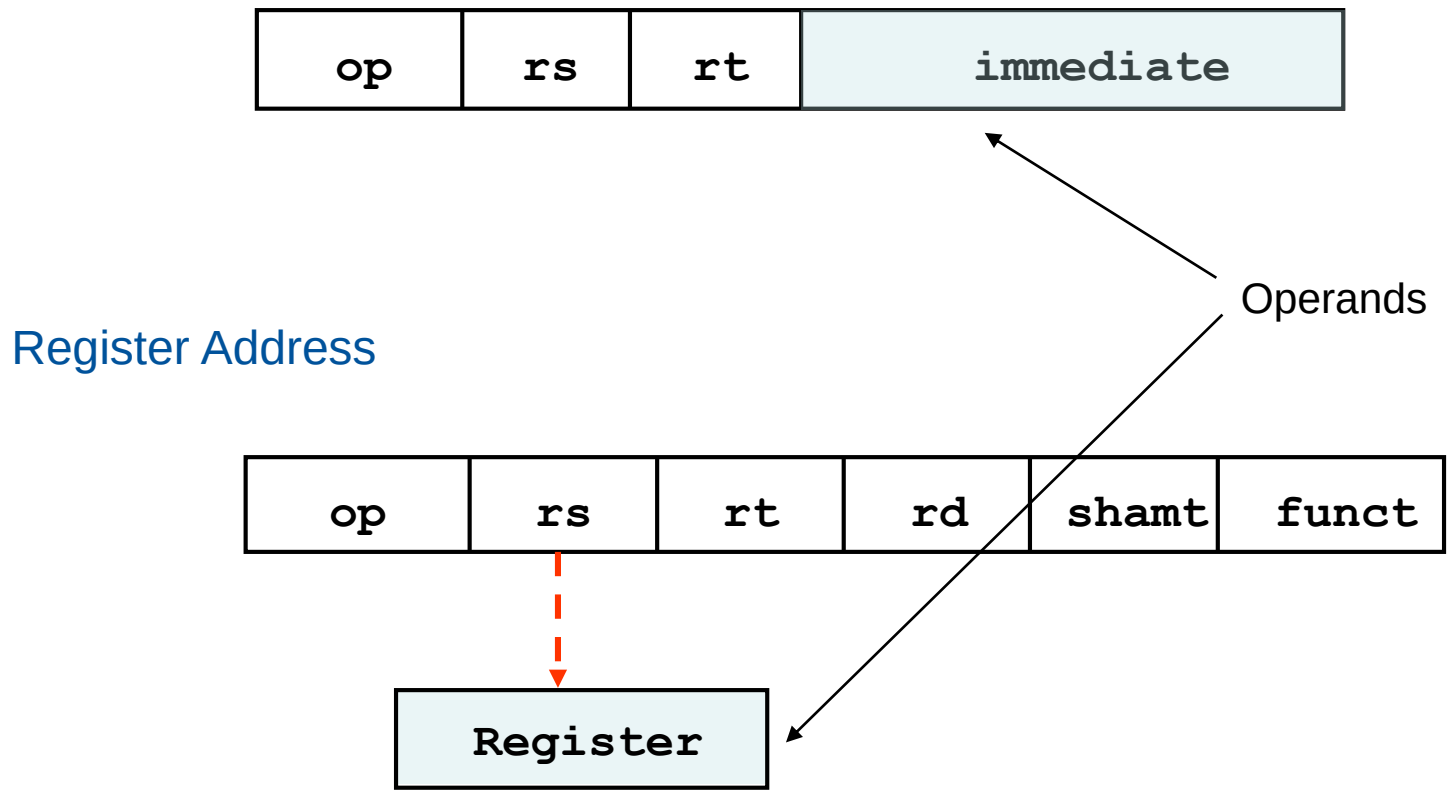


- A procedure takes arguments as input and produces results.
  MIPS uses the following convention to pass arguments and results:
  - 4 Registers for arguments: `$a0, ..., $a3`
  - 2 Registers for results: `$v0, $v1`

MIPS Addressing Modes

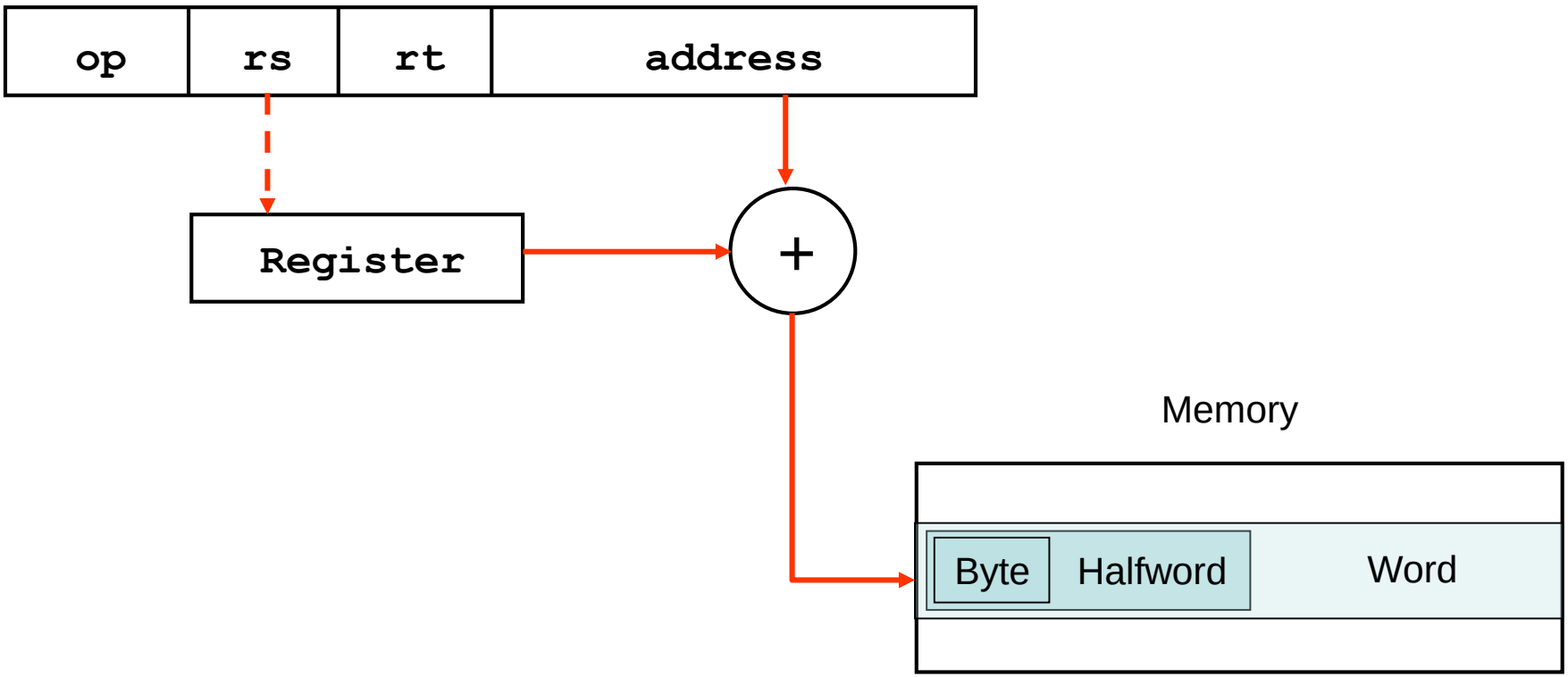
- Direct Addressing

immediate is a 16-bit signed number
→ range [-2^{15}, $+2^{15}-1$]



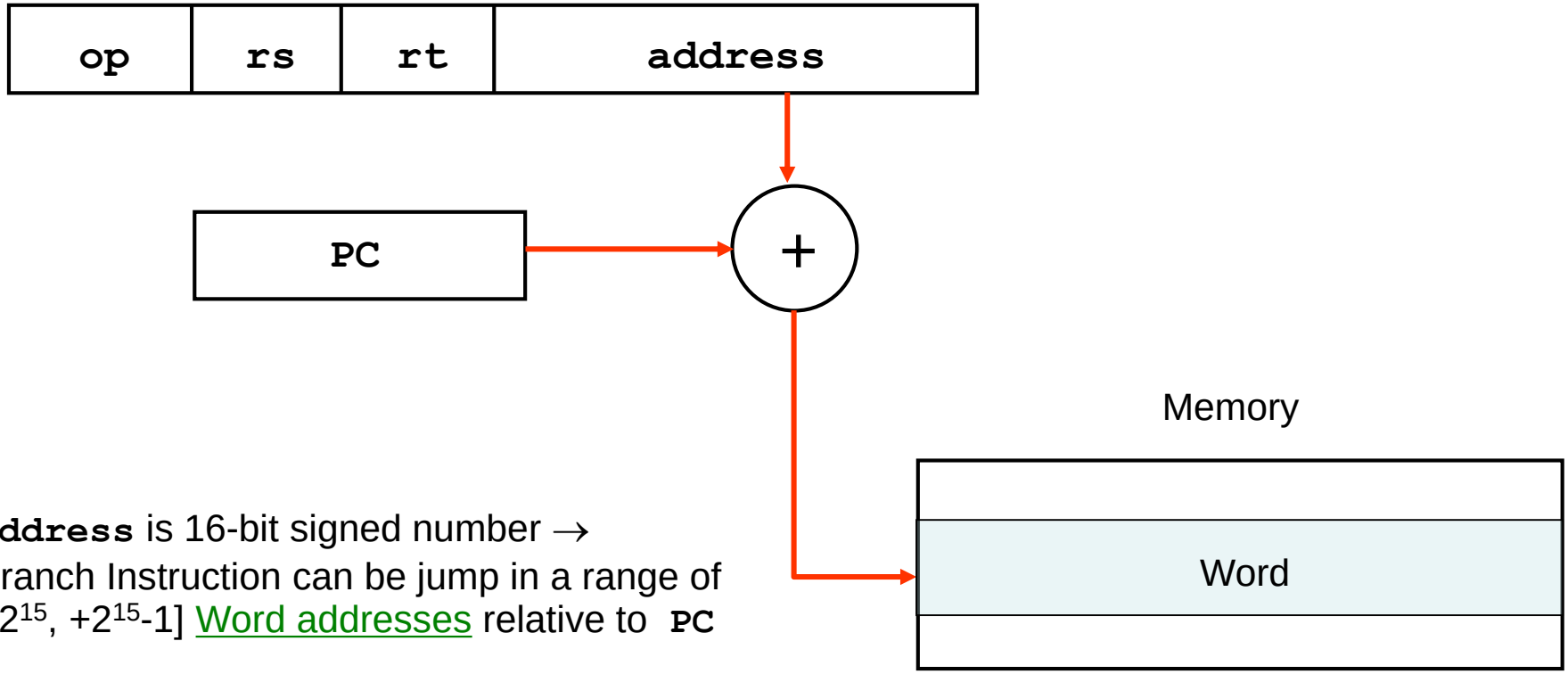
MIPS Addressing Modes

- Base Addressing



MIPS Addressing Modes

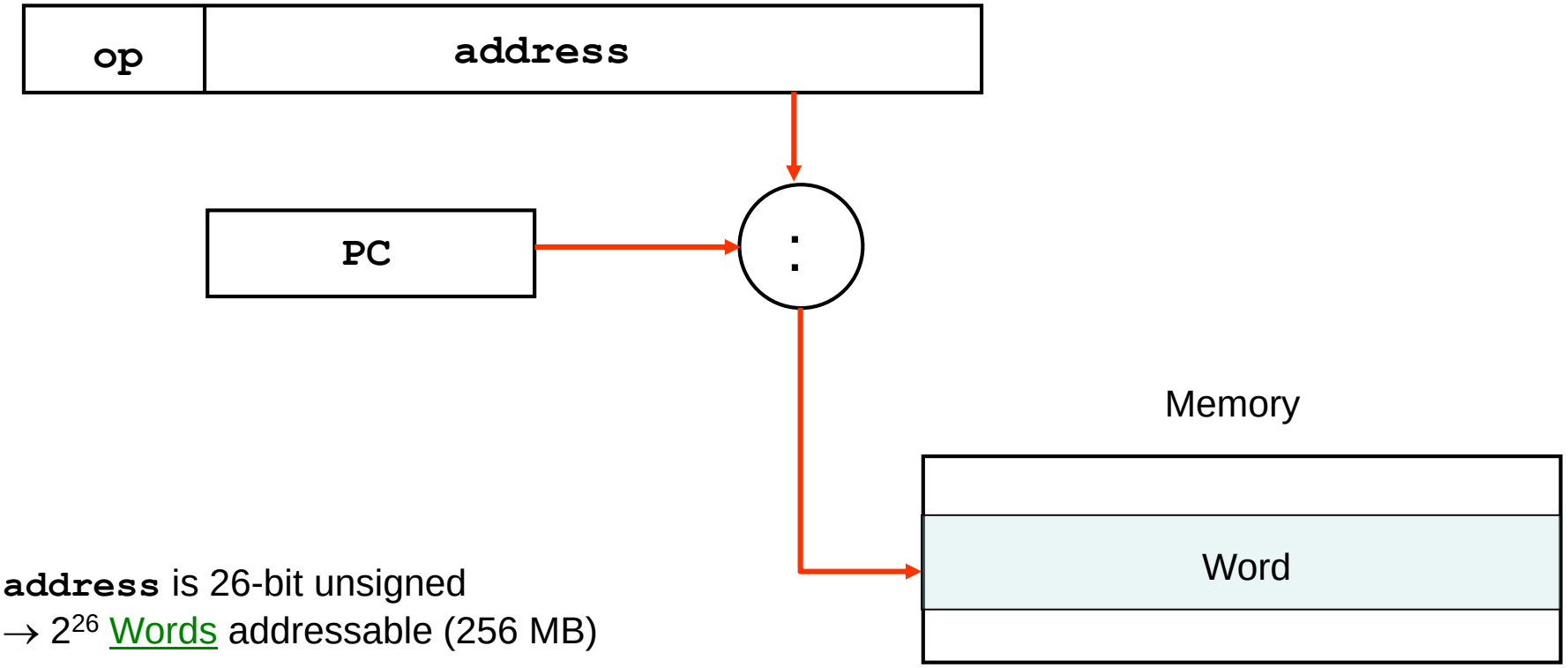
- PC-Relative Addressing



address is 16-bit signed number →
Branch Instruction can be jump in a range of
$[-2^{15}, +2^{15}-1]$ Word addresses relative to **PC**

MIPS Addressing Modes

- Pseudo direct addressing



address is 26-bit unsigned
→ 2^{26} Words addressable (256 MB)

Concatenation with the upper 4 bits of the **PC**
→ 2^{30} Words addressable

Summary: MIPS Operands

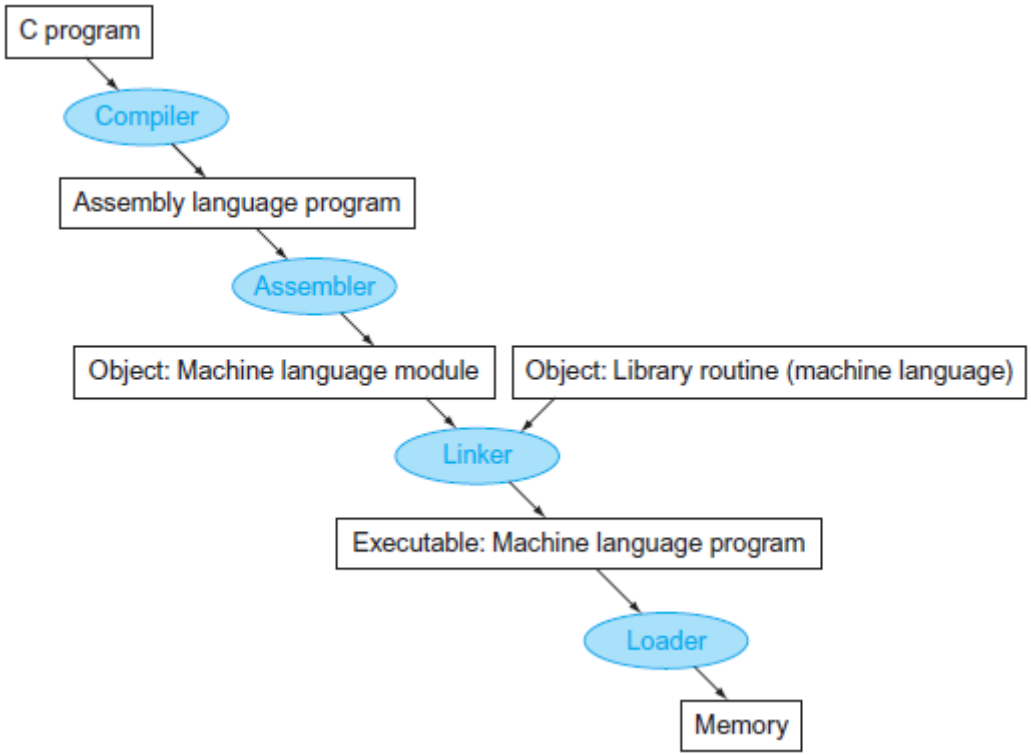
MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Sun

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Compiler, Assembler, Binder and Loader



UF Herbert Wertheim College of Engineering
UNIVERSITY of FLORIDA

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE