Computer Architecture (ENE1004)

▼ Lec 7

Lec 7: Instructions: Language of the Computer 6

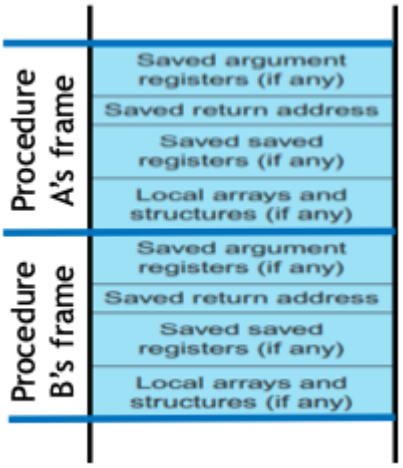
Managing Stack over Procedure Calls : 프로시저 호출을 통한 스택 관리하기

- Procedures may use local arrays or structures, which do not fit into registers
    - Such variables can be stored in the stack (in addition to the registers)
- Stack data can be segmented into procedure frames (or activation records)
    - Procedure frame (activation record) is a segment containing a procedure’s registers and variables
- Assumption: procedure A invokes procedure B
- All the registers and local variables of a procedure are kept within its procedure frame
    - Argument registers ($a0-$a3)
    - Return address register ($ra)
    - Saved registers ($s0-$s7)
    - Local variables
- Whenever a procedure is invoked or returned, its procedure frame should be created or deleted, correspondingly
- 프로시저는 레지스터에 맞지 않는 로컬 배열이나 구조를 사용할 수 있다.
    - 이러한 변수는 레지스터 이외에도 스택에 저장할 수 있다.
- 스택 데이터를 프로시저 프레임(또는 활성화 레코드)으로 세분화 할 수 있다. → 프로시저가 호출될 때마다 스택에는 해당 프로시저에 필요한 메모리 공간이 할당되어

야한다. 이것을 프로시저 프레임 또는 활성 레코드라고 한다.

  - 프로시저 프레임(활성화 레코드)은 프로시저의 레지스터와 변수를 포함하는 세그먼트이다.

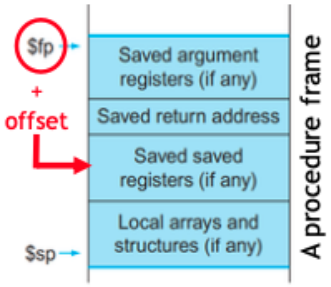
- 가정: 프로시저 A가 프로시저 B를 호출한다.
- 프로시저의 모든 레지스터와 로컬 변수는 프로시저 프레임 내에 유지된다.
  - Argument registers : 인수 레지스터 ($a0-$a3)
  - Return address register : 반환 주소 레지스터 ($ra)
  - Saved registers :저장된 레지스터 ($s0-$s7)
  - Local variables : 지역 변수
- 프로시저가 호출되거나 반환될 때마다 해당 프로시저 프레임이 생성되거나 삭제되어야 한다.

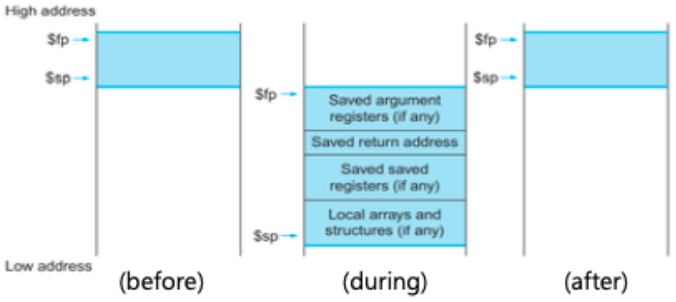


Managing Stack over Procedure Calls : $fp

- It may be hard to use $sp to locate a desired data within a procedure frame
  - A data within a procedure frame can be located by “$sp + offset” - e.g., 4($sp)
  - However, $sp may be changed during the procedure
- MIPS offers a frame pointer ($fp) that is a stable base register within a procedure
  - $fp, which points to the first word of the frame, does not change during the procedure

    - When a procedure is called or returned, $fp should be adjusted like $sp
- 프로시저 프레임 내에서 $sp를 사용하여 원하는 데이터를 찾기가 어려울 수 있다.
    - 프로시저 프레임 내의 데이터는 $sp + offset" (예" 4($sp))으로 찾을 수 있다.
    - 그러나 $sp는 프로시저 중에 변경될 수 있다.
- MIPS는 프로시저 내에서 안정적인 기본 레지스터인 프레임 포인터($fp)를 제공한다.
    - 프레임의 첫 번째 단어를 가리키는 $fp는 프로시저 중에 변경되지 않는다.
    - 프로시저가 호출되거나 반환될 때, $fp는 $sp 처럼 조정되어야 한다.

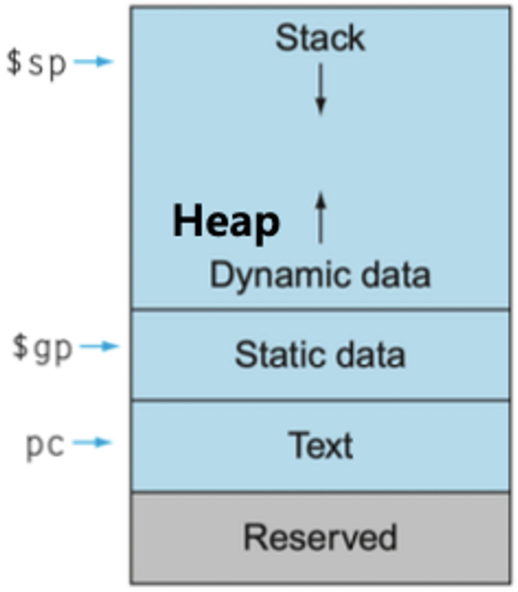




Allocating Space for New Data on the Heap: Heap에 새 데이터를 위한 공간 할당하기

- Memory space can be divided into regions, each of which has a specific purpose
    - Stack + Heap + Static data segment + Text segment
- "Text segment" for MIPS machine code
    - When your program is executed, the instructions are loaded here
    - PC indicates the currently-executed instruction
- "Static data segment" for constants & static variables
    - In C, static variable can be declared outside all procedures or with the keyword *static*
    - MIPS offers $gp (global pointer) to access static data
- "Heap" for dynamic data structures

    - In C, malloc() allocates and free() deallocates heap space
    - Heap and stack grow toward each other
- “Stack” for local variables and procedures
    - $sp indicates the most recently stored data (allocated space)
- 메모리 공간은 각각 특정 목적을 가진 영역으로 나눌 수 있다.
    - 스택 + 힙 + 정적 데이터 세그먼트 + 텍스트 세그먼트
- MIPS machine code의 “텍스트 세그먼트”
    - 프로그램이 실행되면 명령어가 여기에 로드된다.
    - PC는 현재 실행 중인 명령어를 나타낸다.
- 상수 및 정적 변수를 위한 “정적 데이터 세그먼트”
    - C에서 상수 및 정적 변수는 모든 프로시저 외부 또는 “static” 키워드를 사용하여 선언할 수 있다.
    - MIPS는 정적 데이터에 엑세스하기 위해 $gp(전역 포인터)를 제공한다.
- 동적 데이터 구조를 위한 “Heap”
    - C에서 malloc()은 힙 공간을 할당하고 free()는 힙 공간을 할당 해제한다.
    - 힙과 스택은 서로를 향해 성장한다.
- 지역 변수와 프로시저를 위한 “스택”
    - $sp는 가장 최근에 저장된 데이터(할당된 공간)를 나타낸다.



Summary of MIPS Registers

	Name	Register number	Usage
	$zero	0	The constant value 0
for procedures return	$v0–$v1	2–3	Values for results and expression evaluation
for procedures call	$a0–$a3	4–7	Arguments
for temporary data	$t0–$t7	8–15	Temporaries
for saved data	$s0–$s7	16–23	Saved
for temporary data	$t8–$t9	24–25	More temporaries
for static data segment	$gp	28	Global pointer
for procedures	$sp	29	Stack pointer
for offset within procedure	$fp	30	Frame pointer
for procedure call	$ra	31	Return address

- Register 1 ($at) is reserved for assembler
- Registers 26-27 ($k0-$k1) are reserved for operating system

MIPS Addressing for 32-bit Immediate Values : 32비트 immediate 값에 대한 MIPS 주소 지정

- What is the MIPS assembly code to load this 32-bit constant into register $t0? (이 32비트 상수를 레지스터 $t0에 로드하는 MIPS 어셈블리 코드는 무엇인가?)

  0000 0000 0011 1101 0000 1001 0000 0000

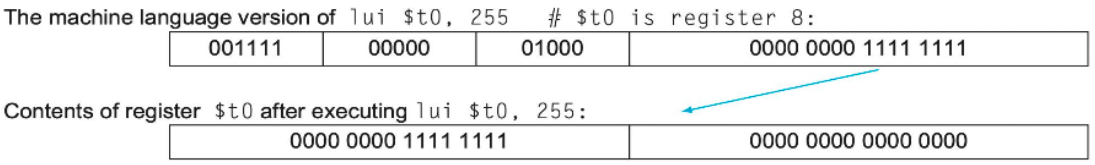
- I-type instruction can express only 16-bit constants (I-타입 명령어는 16비트 상수만 표현할 수 있다.)
  - ~~addi $t0, $zero, 4000000~~

  - The value of 4,000,000 does not fit into the 16-bit field (4,000,000의 값은 16비트 필드에 맞지 않는다.)

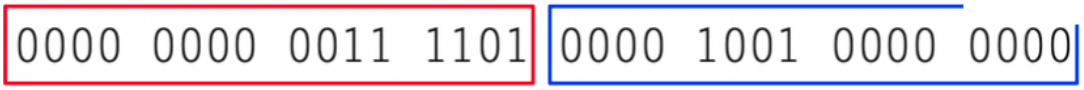
- Load upper immediate (lui)

```
lui $t0, 255 # 255 decimal(십진수) = 0000 0000 1111 11
11 binary(이진수)
```

- lui transfers the 16-bit constant field value into the leftmost 16 bits of the register (lui는 16비트 상수 필드 값을 레지스터의 가장 왼쪽 16비트로 전송한다.)
- The lower 16 bits are filled with 0s (아래 16비트는 0으로 채워진다.)



- What is the MIPS assembly code to load this 32-bit constant into register $t0? (이 32비트 상수를 레지스터 $t0에 로드하는 MIPS 어셈블리 코드는 무엇인가?)



- Load upper immediate (lui):

```
lui $t0, 61 # 61 decimal(십진수) = 0000 0000 0011 1101 binary(이진수)
```

  - This loads the value of 61 onto the upper 16 bits of $t0 (이렇게 하면 $t0의 상위 16비트에 61의 값이 로드된다.)
  - $t0 = 0000 0000 0011 1101 0000 0000 0000 0000
- The next step is to insert the lower 16 bits with a binary valur of 0000 1001 0000 0000 (다음 단계는 0000 1001 0000 0000의 이진 값으로 하위 16비트를 삽입하는 것이다.)

```
ori $t0, $t0, 2304 # 2304 decimal(십진수) = 0000 1001 0000 0000 binary(이진수)
```

  - $t0 = 0000 0000 0011 1101 0000 0000 0000 0000
  - 2304 = 0000 0000 0000 0000 0000 1001 0000 0000
  - The final value in $t0 is 0000 0000 0011 1101 0000 1001 0000 0000
- Two instructions, lui and ori, can collectively load a 32-bit constant into a register (2개의 명령어, lui와 ori는 32비트 상수를 레지스터에 집합적으로 로드

할 수 있다.)

- lui target_register upper_16_bit_value
- ori target_register target_register lower_16_bit_value

MIPS Addressing for 32-bit Addresses (32비트 주소를 위한 MIPS 주소 지정)

- How do MIPS instructions express a memory address? (MIPS 명령어는 메모리 주소를 어떻게 표현할까?)
- Jump instruction (J-type)

```
j 10000 # go to location 10000
```

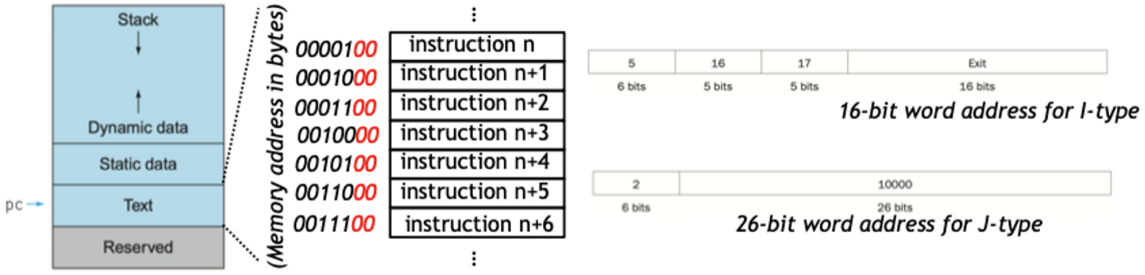
  - You have “26 bits” to express a memory address (메모리 주소를 표현하는 26비트가 있다.)
  - What if a program is bigger than 2^26 bytes (an address larger than 2^26)? (프로그램이 2^26 바이트보다 큰 경우 (2^26보다 큰 주소)에는 어떻게 해야 할까?)
- Conditional branch instructions (I-type)

```
bne $s0, $s1, Exit # go to Exit if $s0 is not equal t
o $s1
```

  - Here, you have only “16 bits” to express an address, which is much smaller than j-type (여기서는 주소를 표현할 수 있는 “16비트”만 있으며, 이는 J 타입보다 훨씬 작다.)
- Then, is there a way to express larger (e.g., 32-bit) memory addresses? (그렇다면, 더 큰 (ex: 32bit) 메모리 주소를 표현할 수 있는 방법이 있을까?)

MIPS Addressing for 32-bit Addresses: in Word

- The address specifies a location where an instruction is stored in the text segment (주소는 텍스트 세그먼트에서 명령어가 저장되는 위치를 지정한다.)
  - j 10000 # go to location 10000
  - bne $s0, $s1, Exit # go to Exit if $s0 is not equal to $s1
  - 10000 and Exit indicate target instructions
- Due to the size (word) of instructions, we do not have to consider byte offset (명령어의 크기(word)로 인해, 바이트 오프셋을 고려할 필요가 없다.)
  - The last two bits are always 00 (e.g., xxxx xxxx xxxx xxxx xx00), which wastes bits in the field (마지막 두 비트는 항상 00(예: xxxx xxxx xxxx xxxx xx00)이므로 필드에서 비트가 낭비된다.)
- If target address is specified in word-address, we express 4X larger address space (대상 주소가 워드 주소로 지정되면 4배 더 큰 주소 공간을 표현한다.)



💡 **ChatGPT 요약**

MIPS 메모리 주소 표현 방법

1. J 타입(Jump 명령어) - 점프 명령어

예시: `j 10000` 이라는 명령어는 프로그램의 `10000` 위치로 점프하라는 의미이다. 이 명령어는 메모리 주소를 표현하기 위해 26비트를 사용한다. 그러나 이 방식으로는 2^26 바이트보다 큰 메모리 주소를 표현할 수 없다. 프로그램이 2^26 바이트보다 크다면, 이 점프 명령어만으로는 모든 메모리 위치에 접근할 수 없다.

2. I 타입(조건부 분기 명령어) - 예: bne

예시: `bne $s0, $s1, Exit` 명령어는 `$s0` 와 `$s1` 이 같지 않으면 `Exit` 라벨로 분기하라는 의미이다. 이 명령어는 메모리 주소를 표현하기 위해 16비트만 사용한다. 따라서 J 타입 명령어보다 표현할 수 있는 주소 범위가 훨씬 작다.

32비트 메모리 주소 표현 방법

MIPS 명령어는 32비트 주소 공간을 다룰 수 있어야 한다. 그런데 J 타입과 I 타입 명령어는 주소를 직접적으로 전체 32비트로 표현하지 않는다.

해결책: 명령어가 저장되는 메모리 위치를 '워드(word)' 단위로 간주한다. MIPS에서 한 워드는 4바이트이다. 따라서, 주소를 워드 주소로 지정할 때, 실제 메모리 주소 공간은 4배 더 커진다. 즉, 26비트로 표현할 수 있는 메모리 주소 범위가 실제로는 4배 더 넓어지는 것이다.

이유: MIPS 아키텍처에서 모든 명령어는 4바이트(32비트)의 고정된 크기를 가진다. 그래서 명령어 주소의 마지막 두 비트는 항상 00이다(4바이트 단위로 정렬되기 때문). 이는 주소를 워드 단위로 표현할 때, 실제로 사용되는 비트 수를 줄여 더 큰 범위의 메모리 주소를 표현할 수 있게 한다.

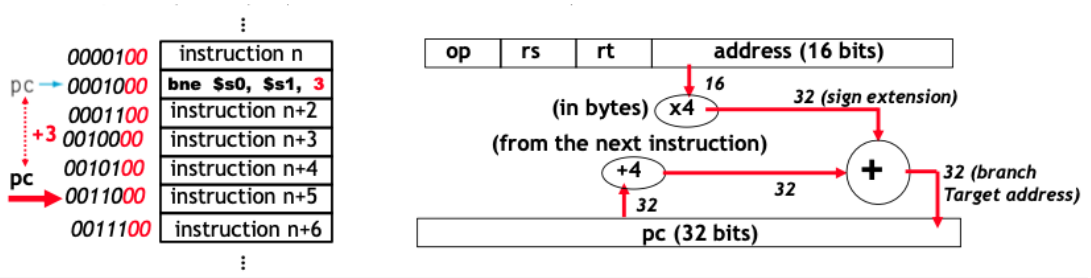
간단히 말해, MIPS에서는 메모리 주소를 더 크게 표현하기 위해 주소를 워드 단위로 계산합니다. 이 방법을 사용하면, 실제로 필요한 비트 수보다 더 넓은 범위의 메모리 주소를 표현할 수 있게 됩니다.

PC-Relative Addressing for I-Type (I-타입을 위한 PC 상대 주소 지정)

- The 16-bit field of I-type can still express only 2^16 words (and instructions)： I 유형의 16비트 필드는 여전히 2^16 단어(및 명령어)만 표현할 수

있다.

- PC-relative addressing for I-type : I 타입의 PC 상대 주소 지정
  - Target address is specified based on PC (the address of the current instruction) : 타겟 주소는 PC(현재 명령어의 주소)를 기준으로 지정된다.
  - Target address (32-bit) = PC (32-bit) + branch offset (16-bit) : 타겟 주소(32비트) = PC(32비트) + 분기 오프셋(16비트)
  - Actually, almost all loops and if statements are much smaller than 2^16 words: 실제로 거의 모든 루프와 if 문은 2^16 단어보다 훨씬 작다.
  - In MIPS, PC = (PC **+ 4**) + (branch address in word *** 4**)



💡 **ChatGPT 요약**

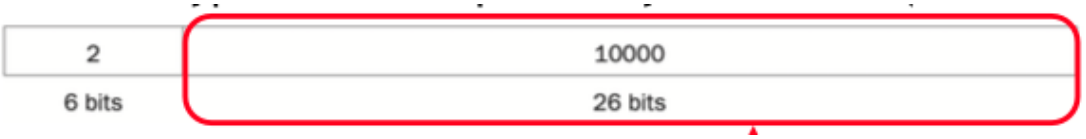
이 파트는 MIPS 아키텍처에서 I 타입 명령어에 대한 PC 상대 주소 지정에 관한 내용을 설명하고 있다.

1. **I 타입 필드의 제한:**
    - I 타입 명령어의 16비트 필드는 여전히 2^16개의 단어 또는 명령어만 표현할 수 있다.
2. **PC 상대 주소 지정:**
    - PC 상대 주소 지정은 현재 명령어의 주소(PC)를 기준으로 대상 주소를 지정하는 방식이다.
    - 대상 주소는 PC와 분기 오프셋을 더하여 계산된다.
    - MIPS 아키텍처에서는 PC 값이 (PC + 4)가 되고, 분기 주소는 워드 단위로 표현되므로 분기 주소에 4를 곱한 값을 더한다.
3. **실제 상황에서의 주소 필드 크기:**
    - 대부분의 루프와 조건문은 실제로 2^16개의 단어보다 훨씬 작다.
    - 따라서 대부분의 경우 PC 상대 주소 지정이 충분하며, 16비트의 주소 필드로도 대부분의 제어 흐름 구조를 표현할 수 있다.

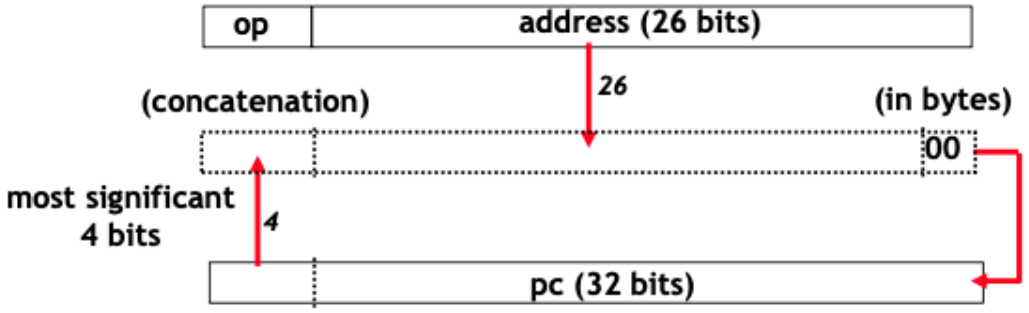
이러한 방식을 통해 MIPS 아키텍처에서 상대적으로 작은 주소 필드를 사용하여 대다수의 제어 흐름 구조를 표현할 수 있다.

Pseudo-direct Addressing for J-Type

- The 26-bit field of J-type can still express only 2^26 words (and instructions)



- Pseudo-direct addressing for J-type
    - Target address is specified patially based on PC (the address of the current instruction)
    - Target address (32-bit) = Upper 4 bits of PC (32-bit) ⊕ branch offset (26-bit)



ChatGPT 요약

이 파트는 MIPS 아키텍처에서 J 타입 명령어에 대한 유사한 직접 주소 지정에 관한 내용을 설명하고 있다.

1. **J 타입 필드의 제한:**
   - J 타입 명령어의 26비트 필드는 여전히 2^26개의 단어 또는 명령어만 표현할 수 있습니다.
2. **유사한 직접 주소 지정:**
   - 유사한 직접 주소 지정은 현재 명령어의 주소(PC)를 부분적으로 기준으로 대상 주소를 지정하는 방식이다.
   - 대상 주소는 PC의 상위 4비트와 분기 오프셋을 XOR 연산하여 계산된다.
   - 이를 통해 상대적으로 큰 주소 공간을 표현할 수 있다.

이러한 방식을 통해 MIPS 아키텍처에서 상대적으로 작은 J 타입 주소 필드로도 큰 주소 공간을 표현할 수 있다.

MIPS Addressing for 32-bit Addresses: Example

```
Loop:
    sll $t1, $s3, 2 # Temp reg $t1 = i * 4
    add $t1, $t1, $s6 # $t1 = address of save[i]
    lw $t0, 0($t1) # Temp reg $t0 = save[i]
    bne $t0, $s5, Exit # go to Exit if save[i] != k
    addi $s3, $s3, 1 # i = i + 1
```

```
    j Loop # go to Loop
Exit:
```

is assembled to

- Assumption: the loop starts at location 80000 in memory
- bne $t0, $s5, Exit at 80012
  - In PC-relative addressing mode, the target address (Exit) is set to 2
  - 80012+4 (the following in struction of PC) + 2*4 = 80024
- j Loop at 80020
  - In pseudo-direct addressing mode, the target address (Loop) is set to 20000
  - 0000 ⊕ 20000 * 4 = 80000

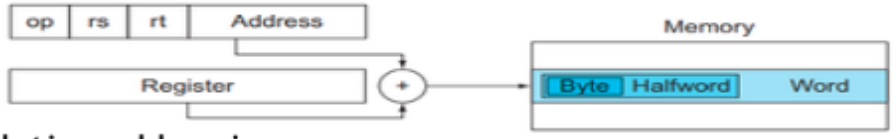
80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

MIPS Addressing Modes

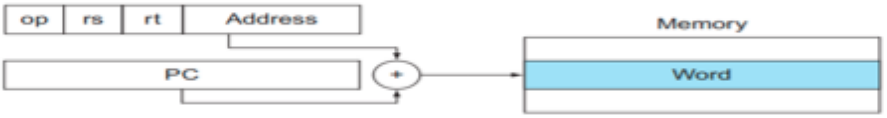
- Addressing mode: how machine instructions identify the operand(s)
- (1) Immediate addressing

- (2) Register addressing

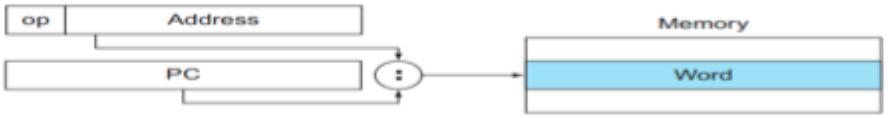
- (3) Base addressing



- (4) PC-relative addressing



- (5) Pseudo-direct addressing



MIPS Organization (Summary)