Computer Architecture
(ENE1004)

Lec – 8: Instructions: Language of the Computer (Chapter 2) - 7

Nested Procedures

- All procedures are not leaf procedures
  - main() calls func_A(), which calls func_B(); here, func_A() is a nested procedure
  - Recursive procedures are also nested
- A problematic example in nested procedures
  - main() calls procedure A with an argument of 3 (①**addi $a0, $zero, 3;** ②**jal A**)
  - Procedure A calls procedure B with an argument of 7 (③**addi $a0, $zero, 7;** ④**jal B**)
  - You may find two conflicts;
    - At ③, procedure B updates **$a0** with 7; what if procedure A continues to expect that **$a0** holds 3?
    - At ④, procedure B updates **$ra** with its return address; procedure A loses its return address
- One solution is to push all the registers that must be preserved onto the stack
  - Caller pushes arg registers (**$a0-$a3**) or temp registers (**$t0-$t9**) that are needed after the call
  - Callee pushes return address register (**$ra**) and saved registers (**$s0-$s7**) used by the callee
  - Note that stack pointer (**$sp**) should be adjusted correspondingly

Nested Procedures: Example

```
int fact (int n)
{
    if (n < 1) return (1);
        else return (n * fact(n - 1));
}
```

is translated into

- **$a0** and **$ra** can be used in the subsequent call, which is kept onto the stack
- **slti** & **beq** for if-then-else statement
- If n < 1, this leaf procedure returns to the caller; here, **$a0** and **$ra** still hold the original values; so, you don't have to get those values from the stack

```
fact:
addi  $sp, $sp, -8        # adjust stack for 2 items
sw  $ra, 4($sp)           # save the return address
sw  $a0, 0($sp)           # save the argument n

slti  $t0, $a0, 1         # test for n < 1
beq  $t0, $zero, L1       # if n >= 1, go to L1

addi  $v0, $zero, 1       # return 1
addi  $sp, $sp, 8         # pop 2 items off stack
jr $ra                    # return to caller
```

```
L1:
addi  $a0, $a0, -1        # n >= 1: arg gets (n-1)
jal  fact                 # call fact with (n-1)

lw  $a0, 0($sp)           # retrun from jal; restore arg n
lw  $ra, 4($sp)           # restore return address
addi  $sp, $sp, 8         # adjust $sp to pop 2 items

mul   $v0, $a0, $v0       # return n * fact (n-1)
jr $ra                    # return to caller
```

Nested Procedures: Example

```
int fact (int n)
{
    if (n < 1) return (1);
        else return (n * fact(n - 1));
}
```

- If n >= 1, fact(n-1) is called
- The return address of fact() is here
- **$a0** and **$ra** are restored, and **$sp** is readjusted
- The current routine returns to the caller with an argument of n * fact (n-1)

is translated into

```
fact:

addi  $sp,  $sp,  -8        # adjust stack for 2 items
sw  $ra,  4($sp)            # save the return address
sw  $a0,  0($sp)            # save the argument n

slti  $t0,  $a0,  1         # test for n < 1
beq  $t0,  $zero, L1        # if n >= 1, go to L1

addi  $v0,  $zero,  1       # return 1
addi  $sp,  $sp,  8         # pop 2 items off stack
jr $ra                      # return to caller
```

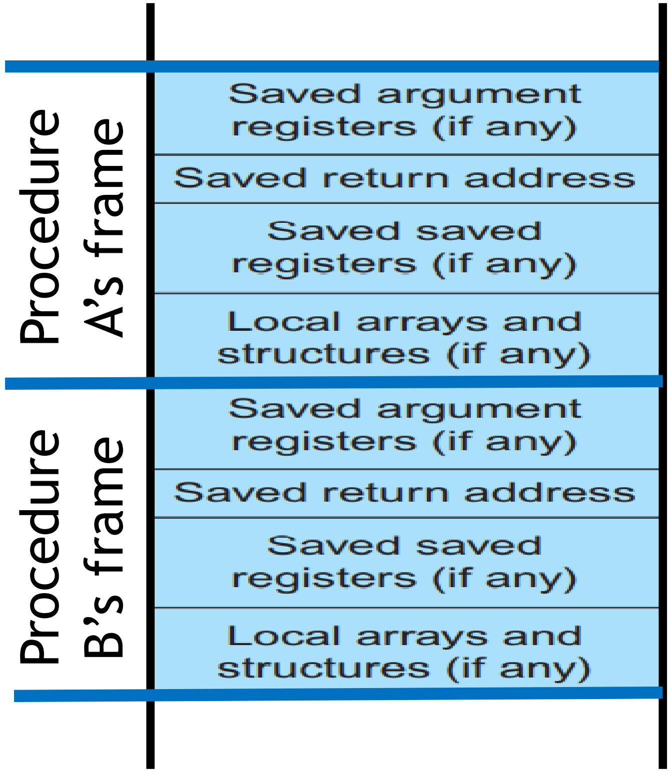
```
L1:
addi  $a0,  $a0,  -1        # n >= 1: arg gets (n-1)
jal  fact                   # call fact with (n-1)

lw  $a0,  0($sp)            # return from jal; restore arg n
lw  $ra,  4($sp)            # restore return address
addi  $sp,  $sp,  8         # adjust $sp to pop 2 items

mul   $v0,  $a0,  $v0       # return n * fact (n-1)
jr $ra                      # return to caller
```

Managing Stack over Procedure Calls

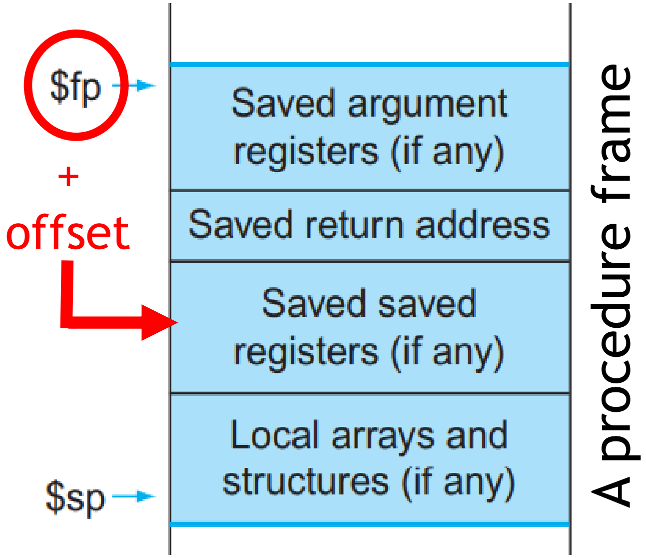
- Procedures may use local arrays or structures, which do not fit into registers
  - Such variables can be stored in the stack (in addition to the registers)
- Stack data can be segmented into procedure frames (or activation records)
  - Procedure frame (activation record) is a segment containing a procedure's registers and variables

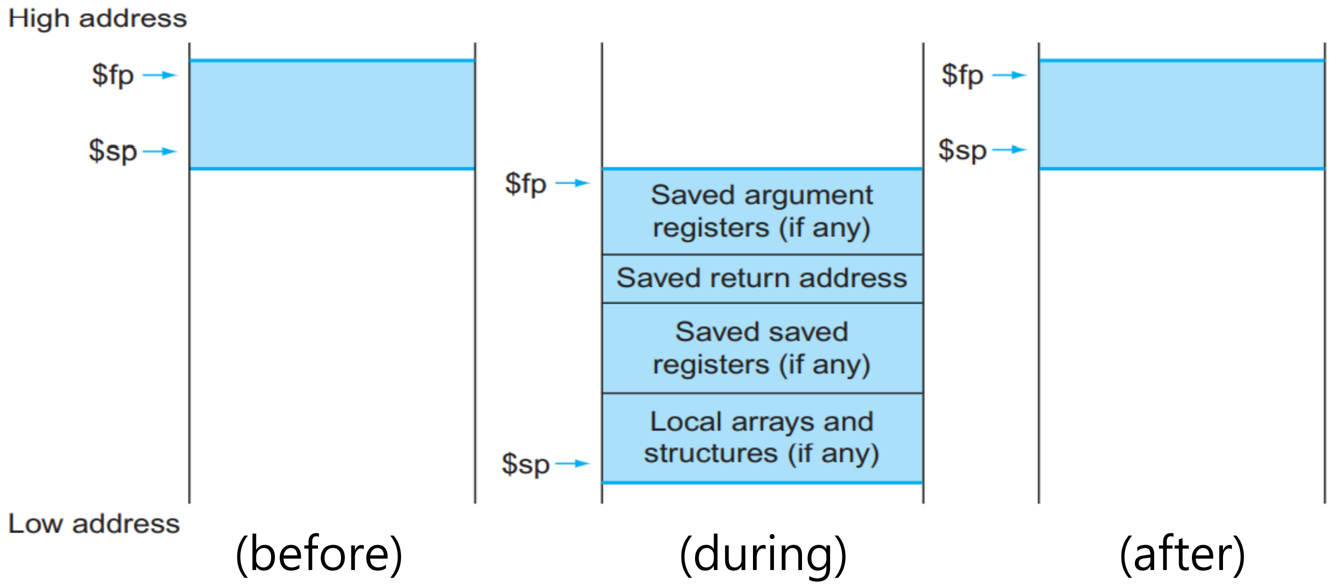


- Assumption: procedure A calls procedure B
- All the registers and local variables of a procedure are ke pt within its procedure frame
  - Argument registers (**$a0-$a3**)
  - Return address register (**$ra**)
  - Saved registers (**$s0-$s7**)
  - Local variables
- Whenever a procedure is invoked or returned, its procedu re frame should be created or deleted

Managing Stack over Procedure Calls: **$fp**

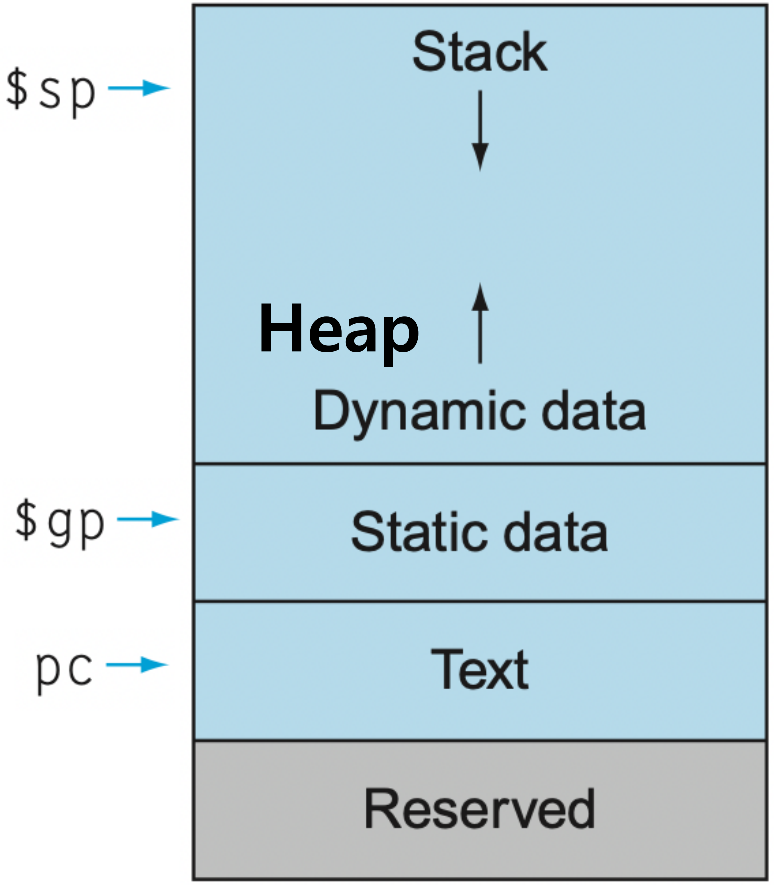
- It may be hard to use **$sp** to locate a desired data within a procedure frame
  - A data within a procedure frame can be located by "$sp + offset" - e.g., **4($sp)**
  - However, **$sp** may be changed during the procedure
- MIPS offers a frame pointer (**$fp**) that is a stable base register within a procedure
  - **$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**





Allocating Space for New Data on the Heap

- 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
    - Static variables exist across exits from and entries to procedure
    - In C, declared outside all procedures or with the keyword *static*
    - MIPS offers **$gp** (global pointer) to access static data
- "Heap" for dynamic data structures
    - In C, malloc() allocates and free() deallocates heap space
    - Heap and stack grow toward each other
- "Stack" for automatic variables (local to procedures)
    - **$sp** indicates the most recently stored data (allocated space)

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 Immediates

- What is the MIPS assembly code to load this 32-bit constant into register **$t0**?

  0000 0000 0011 1101 0000 1001 0000 0000

  - ~~**addi $t0, $zero, 4000000**~~
- I-type instruction can express only 16-bit constants

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

  - The value of 4,000,000 does not fit into the 16-bit field
- Load upper immediate (**lui**): **lui $t0, 255** # 255 decimal = 0000 0000 1111 1111 binary
  - lui transfers the 16-bit constant field value into the leftmost 16 bits of the register
  - The lower 16 bits are filled with 0s

The machine language version of `lui $t0, 255 # $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255:`

0000 0000 1111 1111	0000 0000 0000 0000

MIPS Addressing for 32-bit Immediates

- What is the MIPS assembly code to load this 32-bit constant into register **$t0**?

  0000 0000 0011 1101 | 0000 1001 0000 0000

- Load upper immediate (**lui**): **lui $t0, 61** # 61decimal=0000 0000 0011 1101 binary
  - This loads the value of 61 onto the upper 16 bits of **$t0**
  - **$t0** = 0000 0000 0011 1101 0000 0000 0000 0000
- The next step is to insert the lower 16 bits with a binary value of 0000 1001 0000 0000
  - **ori $t0, $t0, 2304** # 2304 decimal = 0000 1001 0000 0000
  - $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
  - **lui target_register upper_16_bit_value**
  - **ori target_register target_register lower_16_bit_value**

MIPS Addressing for 32-bit Addresses

- How do MIPS instructions express a memory address?
- Jump instruction (J-type)
  - **j 10000 # go to location 10000**

2	10000
6 bits	26 bits

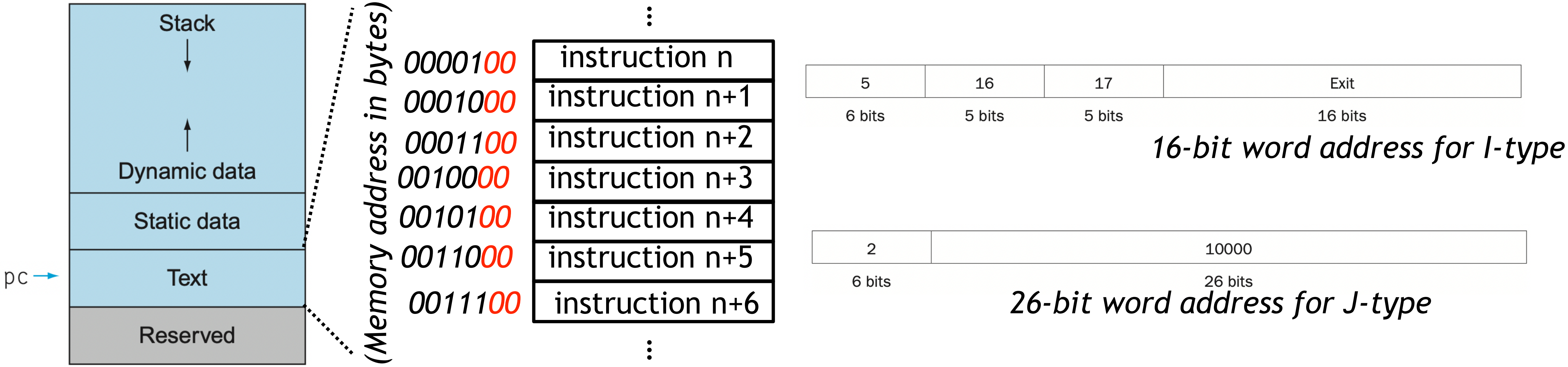
  - You have “26 bits” to express a memory address
  - What if a program is bigger than 2^26 bytes (an address larger than 2^26)?
- Conditional branch instructions (I-type)
  - **bne $s0, $s1, Exit # go to Exit if $s0 is not equal to $s1**

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

  - Here, you have only “16 bits” to express an address, which is much smaller than j-type
- Then, is there a way to express larger (e.g., 32-bit) memory addresses?

MIPS Addressing for 32-bit Addresses: in Word

- The address specifies a location where an instruction is stored in the text segment
  - **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
  - The last two bits are always 00 (e.g., xxxx xxxx xxxx xxxx xx00), which wastes bits in the field
- If target address is specified in word-address, we express 4X larger address space

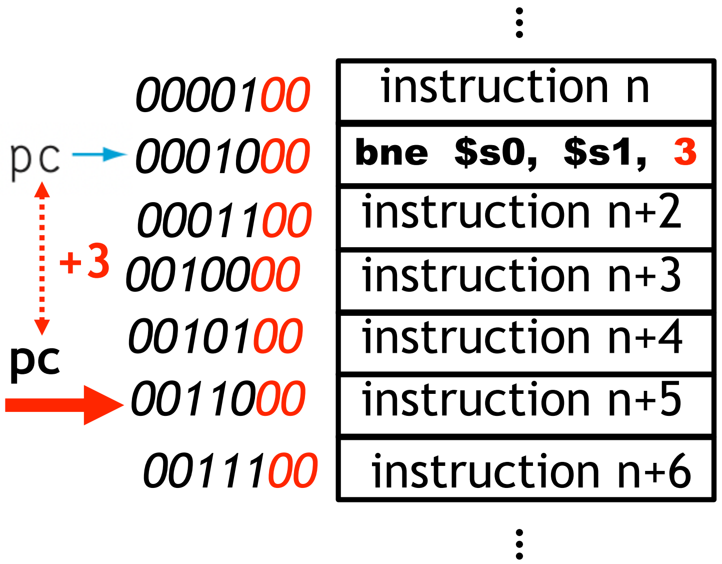


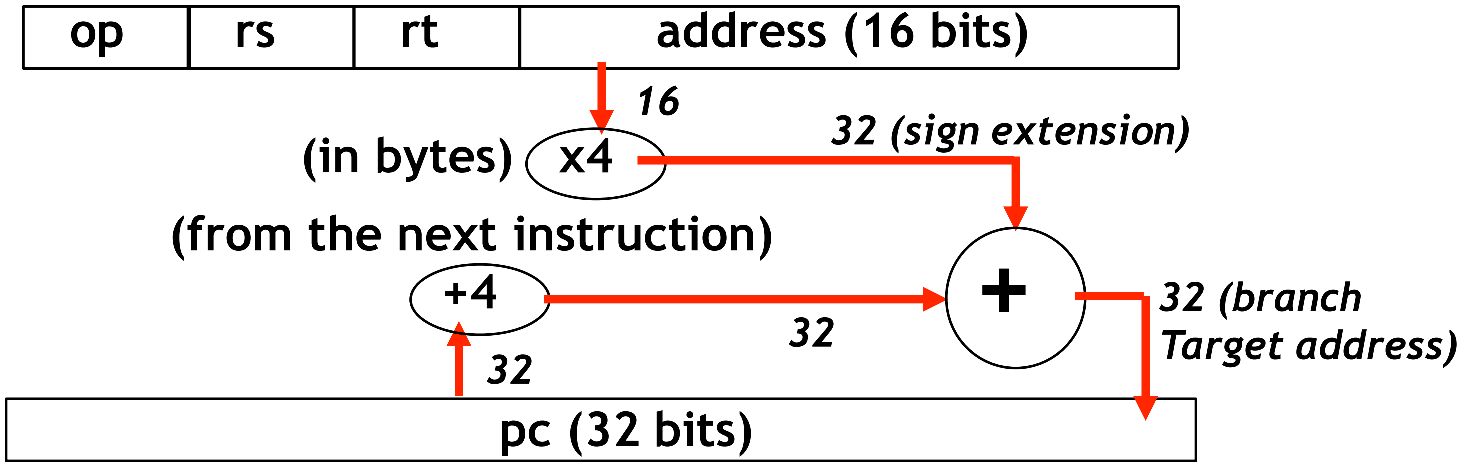
PC-Relative Addressing for I-Type

- The 16-bit field of I-type can still express only 2^16 words (and instructions)

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing for I-type
  - Target address is specified based on PC (the address of the current instruction)
  - Target address (32-bit) = PC (32-bit) + branch offset (16-bit)
  - Actually, almost all loops and if statements are much smaller than 2^16 words
  - In MIPS, PC = (PC + 4) + (branch address in word * 4)



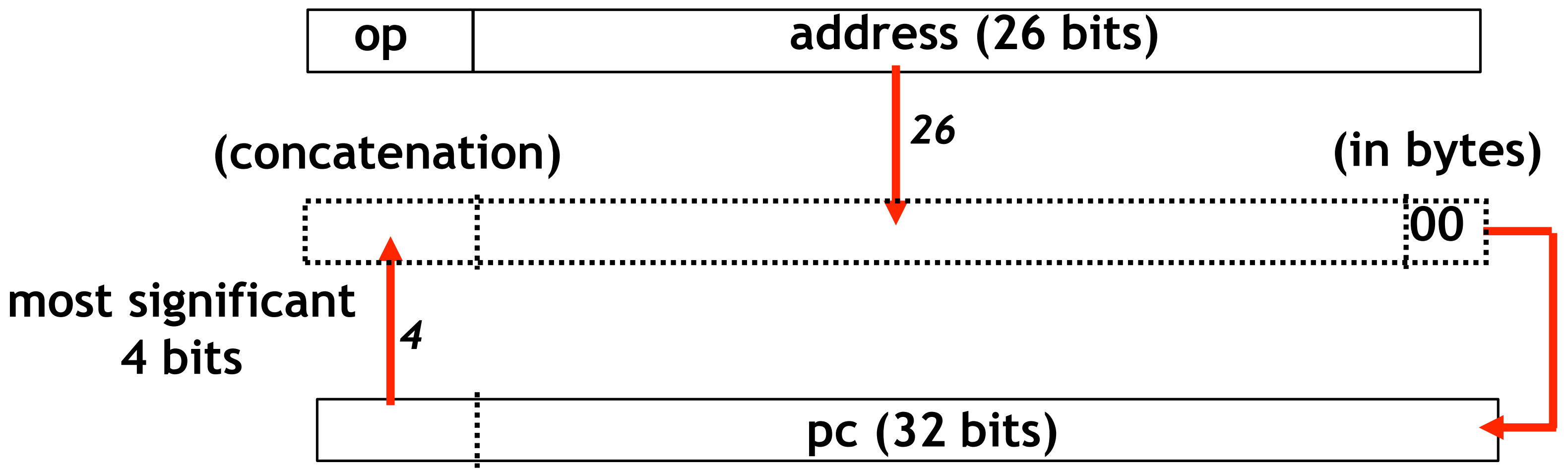


Pseudo-direct Addressing for J-Type

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

2	10000
6 bits	26 bits

- 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)



MIPS Addressing for 32-bit Addresses: Example

```
Loop:  sll   $t1, $s3, 2      # Temp reg $t1 = 4 * i
       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:
```

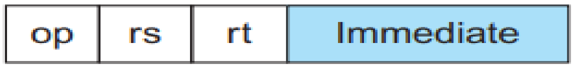
is assembled to

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	. . .					

- Assumption: the loop sta rts 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 = 80 024
- **j Loop at 80020**
  - In pseudo-direct addressi ng mode, the target addr ess (Loop) is set to 20000
  - 0000 ⊕ 20000 * 4 = 80000

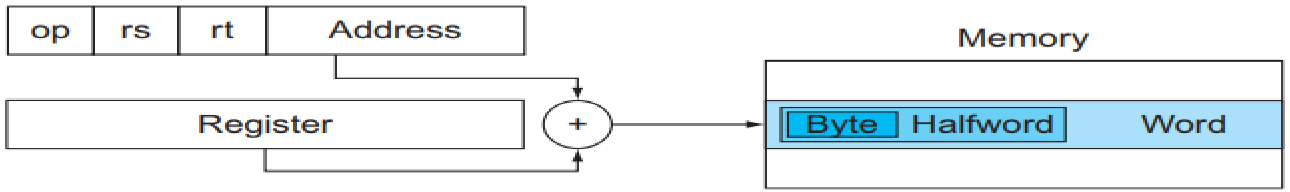
MIPS Addressing Modes

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

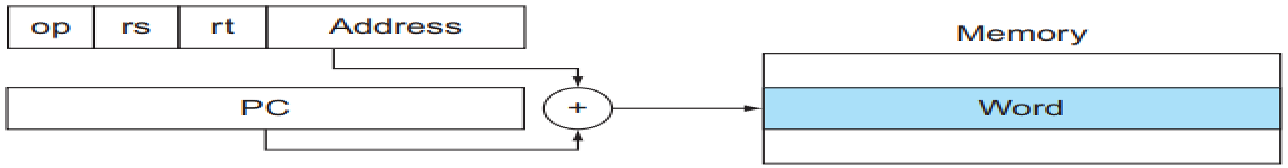


- (2) Register addressing

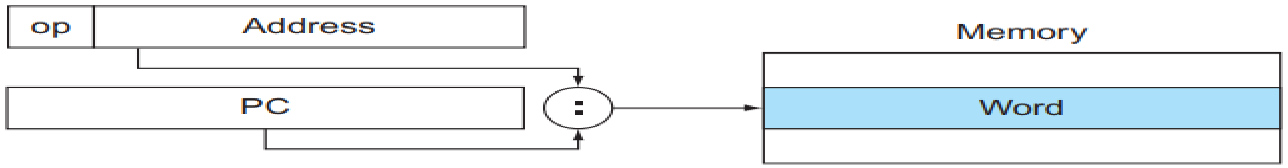
- (3) Base addressing



- (4) PC-relative addressing



- (5) Pseudo-direct addressing



MIPS Organization (Summary)