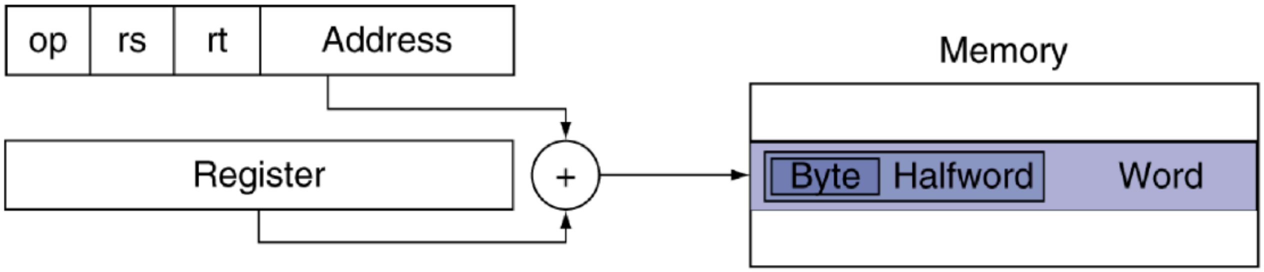


CPU Procedure Calls
Memory Addressing Modes

CS 154: Computer Architecture

Lecture #6

Winter 2020

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- Lab 03 – how is that going?

Lecture Outline

- CPU Procedure Calls
  - The MIPS Calling Convention
- Memory Addressing Modes
- Character Representations
- Parallelism and Synchronization

The MIPS Calling Convention In Its Essence

- Remember: **Preserved** vs **Unpreserved** Regs
- **Preserved**: **$s0 - $s7**, and **$ra**, and **$sp (by default)**
- **Unpreserved**: **$t0 - $t9**, **$a0 - $a3**, and **$v0 - $v1**

- Values held in **Preserved Regs** immediately before a function call MUST be the same immediately after the function returns.
  - Use the **stack memory** to save these

- Values held in **Unpreserved Regs** must always be assumed to change after a function call is performed.
  - $a0 - $a3 are for passing arguments into a function
  - $v0 - $v1 are for passing values from a function

Example

- C/C++ code:

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

<u>Remember</u>:

- Argument **n** in **`$a0`**
- Result in **`$v0`**

Example continued…

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

```
fact:
   addi $sp, $sp, -8     # adjust stack for 2 items
   sw $ra, 4($sp)        # push (save) return address
   sw $s0, 0($sp)        # push (save) argument
   move $s0, $a0
   li $t0, 1
   blt $s0, $t0, else
   mult $v0, $s0
   mflo $v0
   addi $a0, $a0, -1
   jal fact
```

```
else:
   lw $s0, 0($sp)     # restore original n
   lw $ra, 4($sp)     # restore return address
   addi $sp, $sp, 8  # pop 2 items from stack
   jr $ra

main:
   li $v0, 1
   li $a0, 5
   jal fact    # Expect to see returned value in $v0
```

Variable Storage Classes

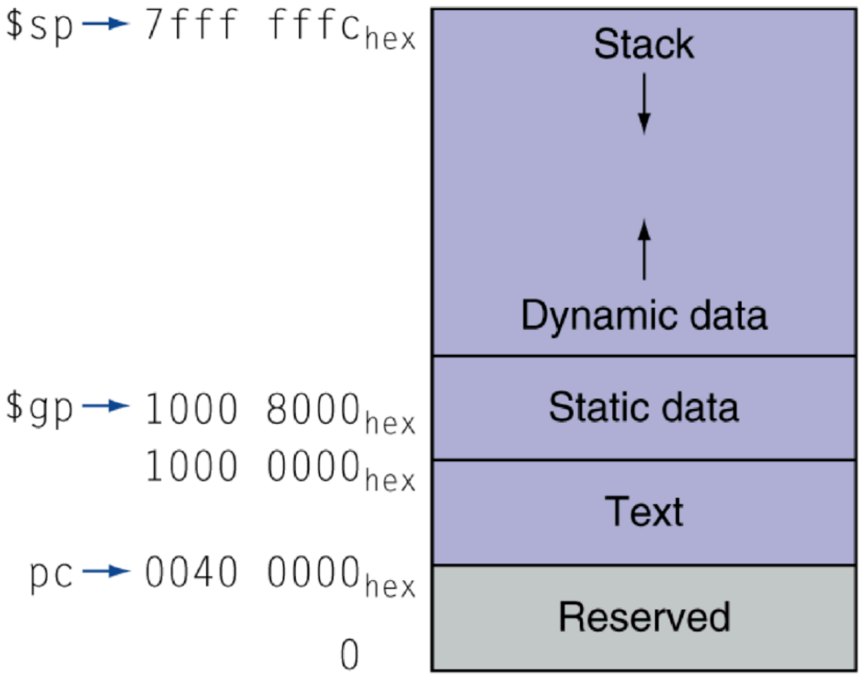
RECALL:

- A C/C++ variable is generally a location in memory
- A variable has **type** (e.g. int, char)
  and **storage class** (automatic vs. static)

- **Automatic variables**: local to a part of the program, created & discarded
- **Static variables**: global vars (declared outside or using **static** in C/C++)

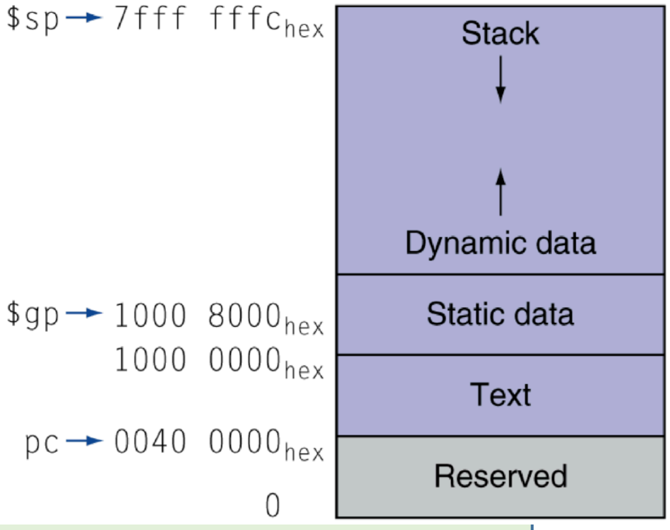
- MIPS software reserves the **global pointer register**, **$gp**, to get access to automatic variables.

Memory Layout



- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - **$gp** initialized to address allowing ±offsets into this segment
- Heap: dynamic data
  - e.g., malloc/free in C, new in C++,
    used for linked lists, dynamic arrays, etc…
- Stack: automatic storage

Stack & Heap in MIPS



- The **stack** is used for saving vars when procedures (functions) are called
  - Also used to store some local vars to the function that can't fit in registers, like local arrays or structures
- The stack starts in the **high end** of memory and grows **down**

- The **heap** is used for saving vars that are dynamic data structures
- The heap starts in the **low end** (after static data) and grows **up**
  - Allows the stack and heap to grow toward each other, allowing efficient use of memory.

Character Data in Computers

Byte-encoded character sets like:

- **ASCII** (7 bits, i.e. 128 characters)
  - No longer used, in favor of UTF-8, which is…
- **Unicode**: 8, 16, and 32-bit character set
  - Used in Java, C++ wide characters, …
  - Contains most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings (8-bits, 16-bits, respectively)

Character Data in Assembly

- Must be stored in memory (Use the **.data** directive)

- Loading them from memory to a register requires:
  lw (load word), **lh** (load half-word), or **lb** (load byte)
    - Especially if you want to do an operation on the data
      (like to change the value of the data)

  Or **la** (load address)
    - Especially if you want to do a syscall on the data
      (you need the address for that)

- When you use **lh** or **lb**, the sign is extend to 32 bits

- Equivalents with **sw** (store word), **sh** (store half-word), and **sb** (store byte)

Representation of Strings

- Characters combined = strings

- 3 choices for representing a string:
  1. 1st position of the string is reserved to give the length of a string (int)
  2. There's an accompanying var for the length of the string (usually in a structure)
  3. The last position of a string is indicated by a EOS character (null or \0)

- C/C++ uses #3
  - So, the string "UCSB" is **5** bytes because the last one is \0

Example

C code (naïve), i.e. with null-terminated string

```
void strcpy (char x[], char y[])
{
  int i;
  i = 0;
  while ((x[i]=y[i])!='\0’)
      i += 1;
}
```

- Addresses of vars **x**, **y** in **$a0**, **$a1**
- Variable **i** in **$s0**

Example in Assembly

```
strcpy:
     addi $sp,$sp,-4     # adjust stack for 1 more item
     sw $s0, 0($sp)      # save $s0, will use it for i
     add $s0, $zero, $zero  # i = 0 (why not use li?)
L1:  add $t1, $s0, $a1   # &y[i] in $t1 (no ref + ix4?)
     lbu $t2, 0($t1)     # $t2 = y[i] (i.e. dereferenced)
     add $t3, $s0, $a0   # &x[i] in $t3
     sb $t2, 0($t3)      # x[i] = y[i]
     beq $t2, $zero, L2  # if y[i] == 0 (i.e. \0), go to L2
     addi $s0, $s0, 1    # else, i = i + 1
     j L1                # Repeat loop
L2:  lw $s0, 0($sp)      # y[i] == 0: end of string.
     addi $sp, $sp, 4    # Restore old $s0; pop 1 word off stack
     jr $ra              # return
```

Branch Addressing

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

I-Type of instruction **(beq , bne)**

- Branch instructions specify:

  Opcode + 2 registers + target address
- Most branch targets are *near* the branch instruction in the *text* segment of memory
  - Either ahead or behind it
- Addressing can be done relative to the value in PC Reg. ("PC-Relative Addressing")
  - Target address = PC + offset (in words) x 4
  - **PC is already incremented by 4 by this time**

Branching Far Away

If branch target is too far to encode with 16-bit offset, then assembler will rewrite the code

- Example

```
        beq $s0, $s1, L1      # L1 is far away
```

↓

```
        bne $s0, $s1, L2      # rewritten…
        j L1
    L2: ...
```

Jump Addressing

op	address
6 bits	26 bits

J-Type of instruction **(`j , jal`)**

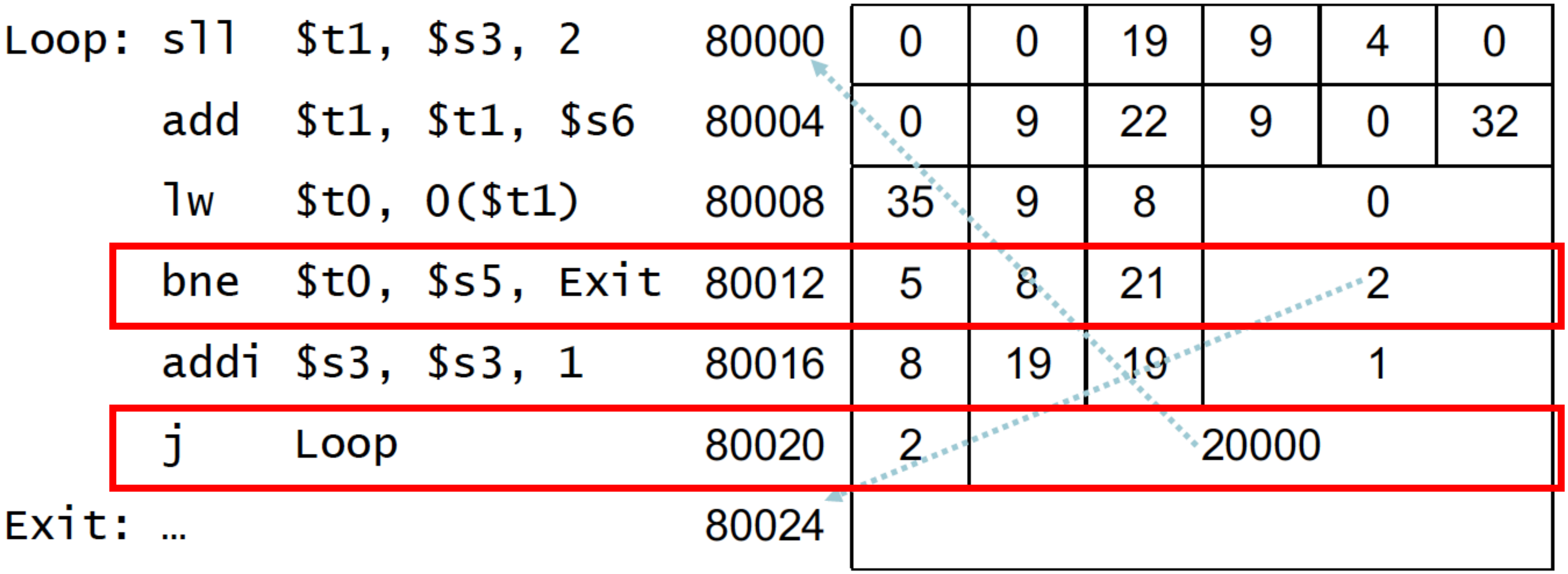
- Jump (j and jal) targets could be anywhere in *text* segment

- Encode full address in instruction

- Direct jump addressing
  - Target address = (address x 4) **OR** (PC[31: 28])
  - i.e. Take the **4** most sig. bits in PC
    and concatenate the **26** bits in “address” field
    and then concatenate another **00** (i.e x 4)

Target Addressing Example

- Assume Loop is at location 80000



`Loop: sll $t1, $s3, 2`	80000	0	0	19	9	4	0
`add $t1, $t1, $s6`	80004	0	9	22	9	0	32
`lw $t0, 0($t1)`	80008	35	9	8	0		
`bne $t0, $s5, Exit`	80012	5	8	21	2		
`addi $s3, $s3, 1`	80016	8	19	19	1		
`j Loop`	80020	2	20000				
`Exit: …`	80024						

Addressing Mode Summary

Examples:

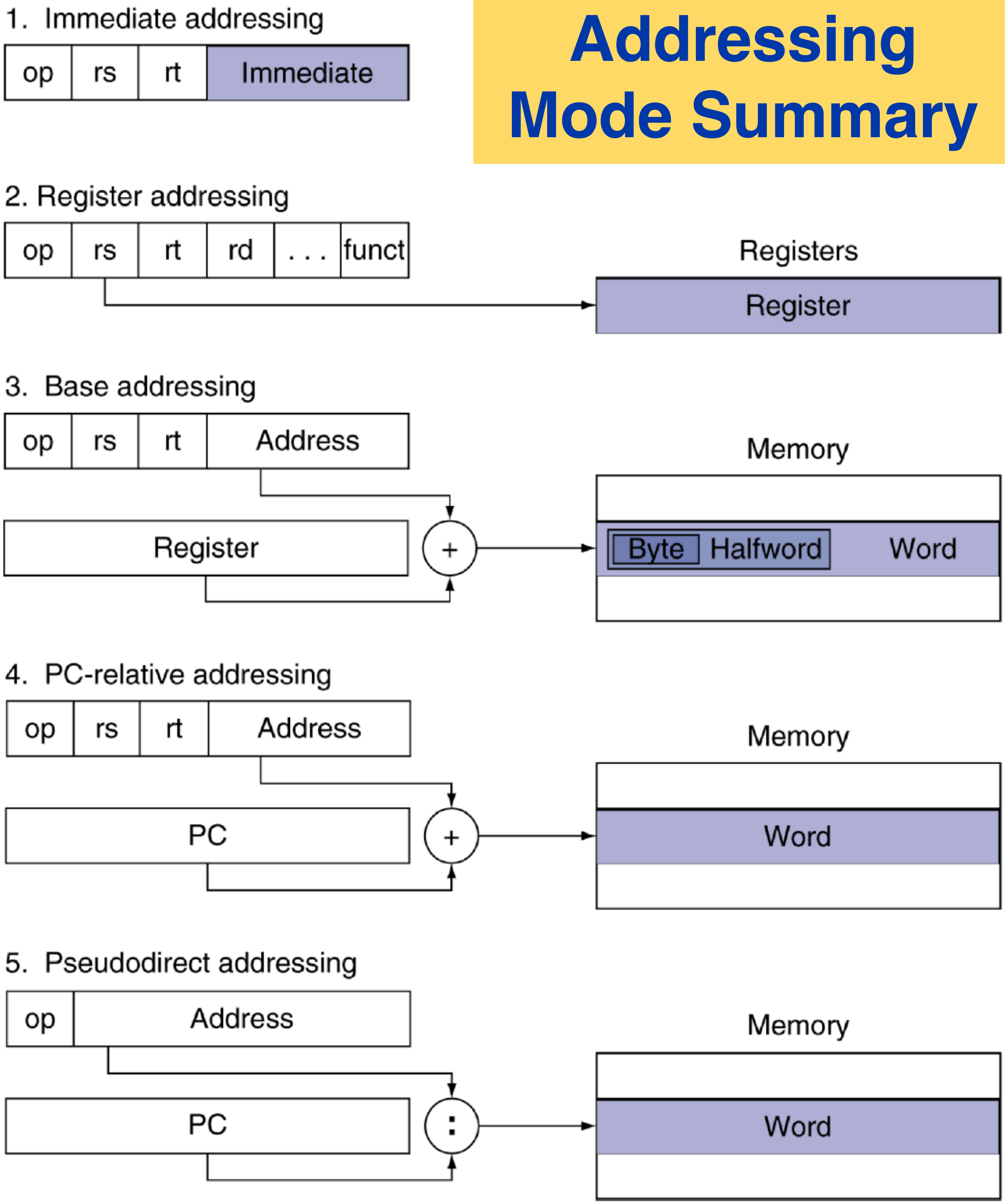
```
addi $t0, $t0, 42

add $t0, $t1, $t3

lw $t0, 4($t1)

beq $t0, $t1, L1

j L1
```



YOUR TO-DOs for the Week

- Readings!
  - Chapters 2.11 – 2.13

- Turn in Lab 3!

</LECTURE>