EECS 314
Computer Architecture

Spring 2018

Chapter 2
Instructions: Language of the Computer

Instructor: Ming-Chun Huang, PhD

ming-chun.huang@case.edu

Office: Glennan 514B

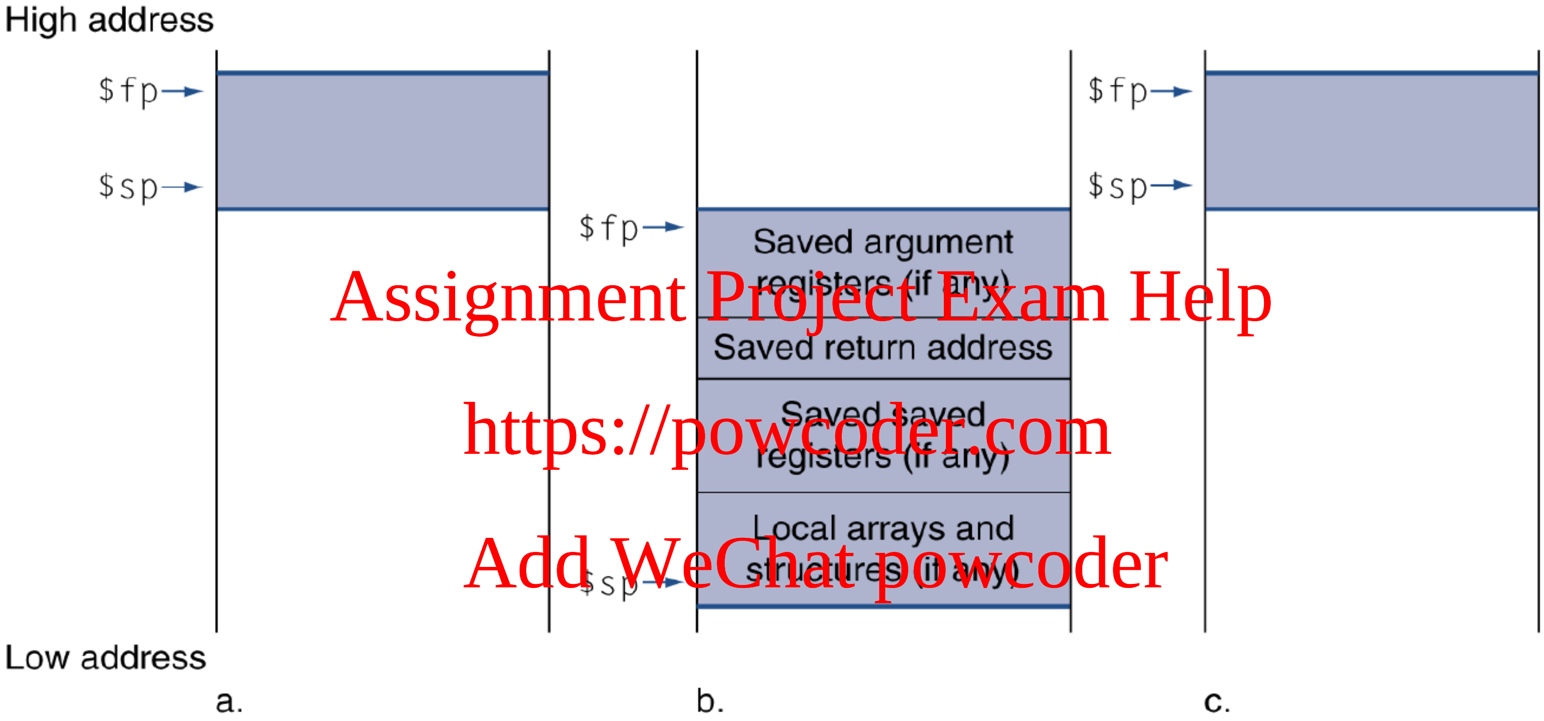
Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Local Data on the Stack


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

- ❑ Local data allocated by callee
  - e.g., C automatic variables
- ❑ Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Leaf Procedure Example

- MIPS code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, …, j in $a0, …, $a3
- f in $s0 (hence, need to save $s0 on stack)
- Result in $v0

```
leaf_example:
   addi  $sp, $sp, -4
   sw    $s0, 0($sp)
   add   $t0, $a0, $a1
   add   $t1, $a2, $a3
   sub   $s0, $t0, $t1
   add   $v0, $s0, $zero
   lw    $s0, 0($sp)
   addi  $sp, $sp, 4
   jr    $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

```
jal  ProcedureAddress    #jump and link
```

String Copy Example

- C code (naïve):
  - Null-terminated string

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

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

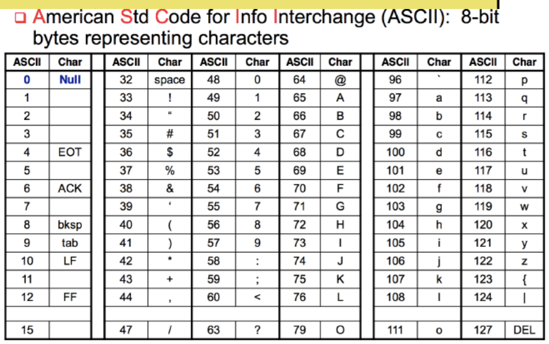
String Copy Example

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

❑ MIPS code:

e.g. x = an empty space
y = “architecture”

```
strcpy:
      addi $sp, $sp, -4         # adjust stack for 1 item
      sw   $s0, 0($sp)          # save $s0
      add  $s0, $zero, $zero    # i = 0
L1:  add  $t1, $s0, $a1        # addr of y[i] in $t1
      lbu  $t2, 0($t1)          # $t2 = y[i]
      add  $t3, $s0, $a0        # addr of x[i] in $t3
      sb   $t2, 0($t3)          # x[i] = y[i]
      beq  $t2, $zero, L2       # exit loop if y[i] == 0
      addi $s0, $s0, 1          # i = i + 1
      j    L1                   # next iteration of loop
L2:  lw   $s0, 0($sp)          # restore saved $s0
      addi $sp, $sp, 4          # pop 1 item from stack
      jr   $ra                  # and return
```



❑ American Std Code for Info Interchange (ASCII): 8-bit bytes representing characters

ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char
0	Null	32	space	48	0	64	@	96	`	112	p
1		33	!	49	1	65	A	97	a	113	q
2		34	"	50	2	66	B	98	b	114	r
3		35	#	51	3	67	C	99	c	115	s
4	EOT	36	$	52	4	68	D	100	d	116	t
5		37	%	53	5	69	E	101	e	117	u
6	ACK	38	&	54	6	70	F	102	f	118	v
7		39	'	55	7	71	G	103	g	119	w
8	bksp	40	(	56	8	72	H	104	h	120	x
9	tab	41	)	57	9	73	I	105	i	121	y
10	LF	42	*	58	:	74	J	106	j	122	z
11		43	+	59	;	75	K	107	k	123	{
12	FF	44	,	60	<	76	L	108	l	124	|
15		47	/	63	?	79	O	111	o	127	DEL

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

e.g. 5! = 5x4x3x2x1=120

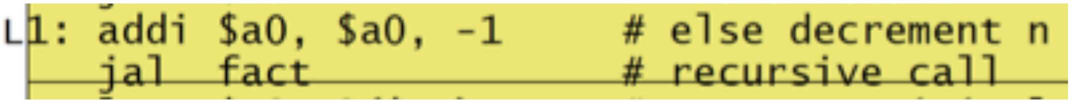
- Argument n in $a0
- Result in $v0

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

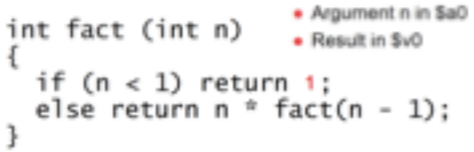
- MIPS code:

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

e.g. 5! = 5x4x3x2x1=120



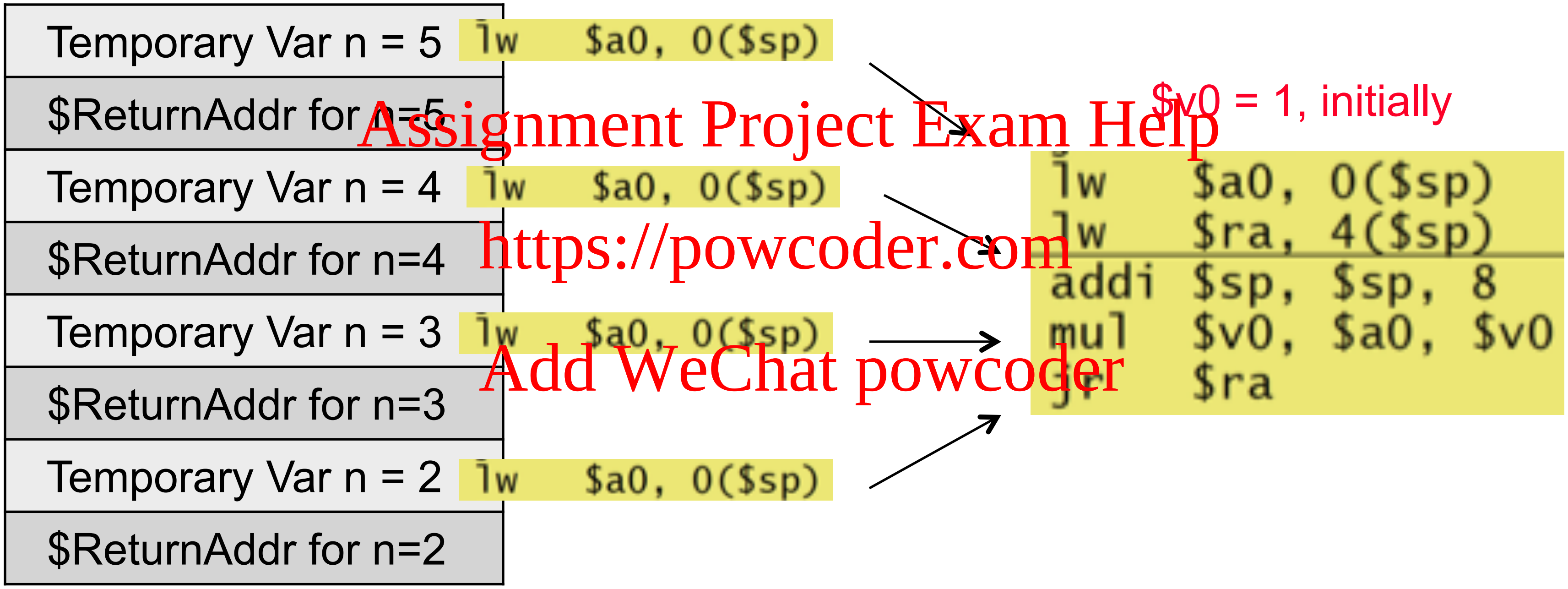
```
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
```



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

- Argument n in $a0
- Result in $v0

Stack in Memory eventually



Temporary Var n = 5

$ReturnAddr for n=5

Temporary Var n = 4

$ReturnAddr for n=4

Temporary Var n = 3

$ReturnAddr for n=3

Temporary Var n = 2

$ReturnAddr for n=2

```
lw    $a0, 0($sp)
```

```
lw    $a0, 0($sp)
```

```
lw    $a0, 0($sp)
```

```
lw    $a0, 0($sp)
```

$v0 = 1, initially

```
lw    $a0, 0($sp)
lw    $ra, 4($sp)
addi  $sp, $sp, 8
mul   $v0, $a0, $v0
jr    $ra
```

Non-Leaf Procedure Example

e.g. 5! = 5x4x3x2x1=120

- Argument n in $a0
- Result in $v0

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

❑ MIPS code:

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

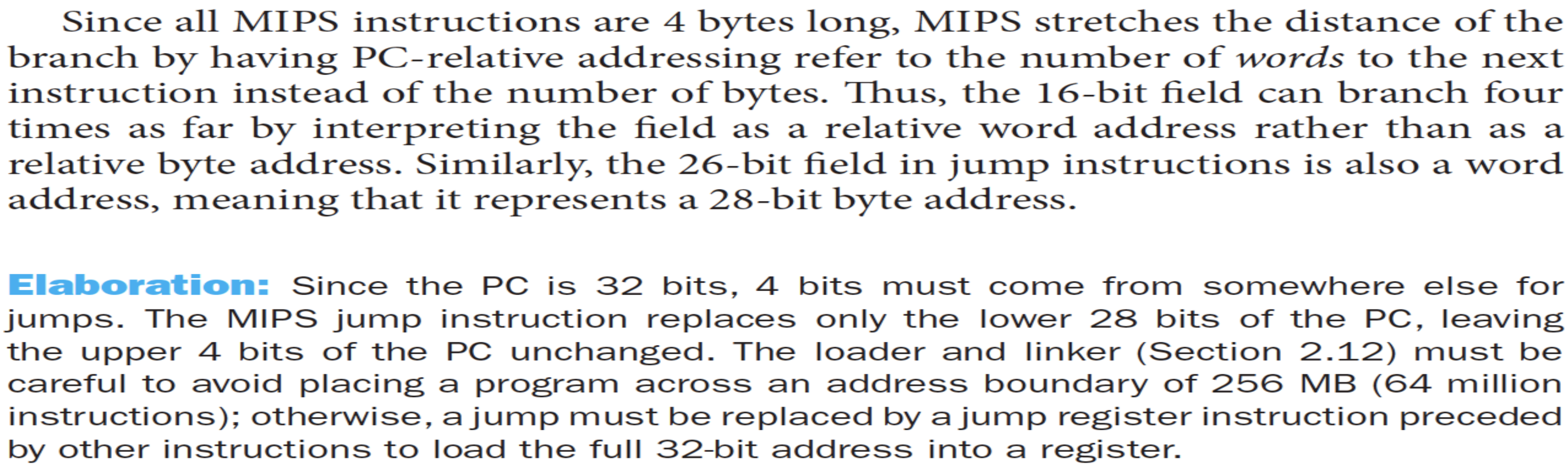
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits



Since all MIPS instructions are 4 bytes long, MIPS stretches the distance of the branch by having PC-relative addressing refer to the number of *words* to the next instruction instead of the number of bytes. Thus, the 16-bit field can branch four times as far by interpreting the field as a relative word address rather than as a relative byte address. Similarly, the 26-bit field in jump instructions is also a word address, meaning that it represents a 28-bit byte address.

Elaboration: Since the PC is 32 bits, 4 bits must come from somewhere else for jumps. The MIPS jump instruction replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged. The loader and linker (Section 2.12) must be careful to avoid placing a program across an address boundary of 256 MB (64 million instructions); otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register.

Target Addressing Example

Here is a traditional loop in C:

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

Assume that i and k correspond to registers $s3 and $s5 and the base of the array save is in $s6. What is the MIPS assembly code corresponding to this C segment?

Remember that MIPS instructions have byte addresses, so addresses of sequential words differ by 4, the number of bytes in a word. The bne instruction on the fourth line adds 2 words or 8 bytes to the address of the *following* instruction (80016), specifying the branch destination relative to that following instruction (8 + 80016) instead of relative to the branch instruction (12 + 80012) or using the full destination address (80024). The jump instruction on the last line does use the full address (20000 × 4 = 80000), corresponding to the label Loop.

Multiplied by 4

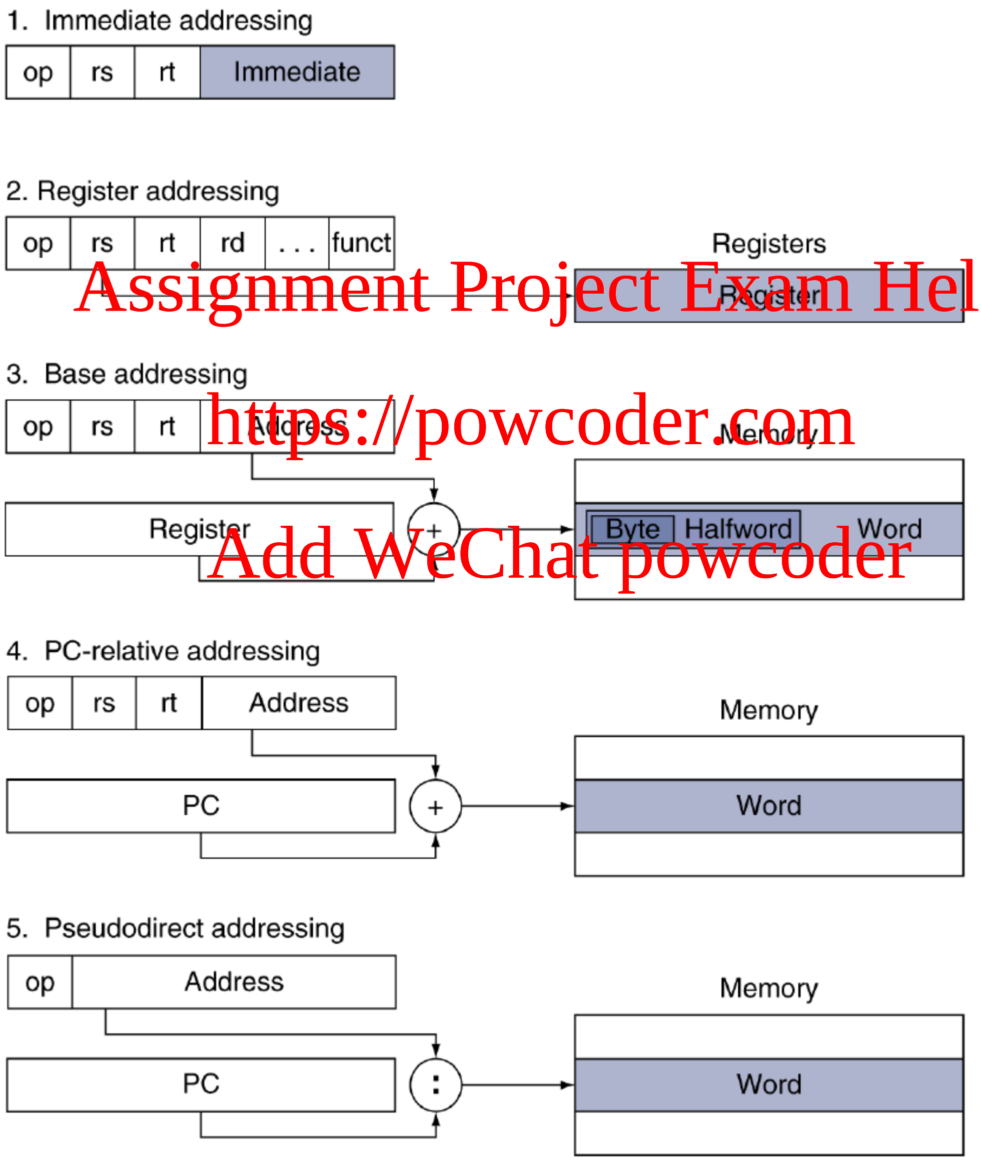
Loop:	sll $t1, $s3, 2	80000	0	0	19	9	4	0
	add $t1, $t1, $s6	80004	0	9	22	9	0	32
	lw $t0, 0($t1)	80008	35	9	8	0		
	bne $t0, $s5, Exit	80012	5	8	21	2		
	addi $s3, $s3, 1	80016	8	19	19	1		
	j Loop	80020	2	20000				
Exit:	…	80024						

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
            beq $s0,$s1, L1
                   ↓
            bne $s0,$s1, L2
            j L1
L2:         …
```

Addressing Mode Summary



C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function

- Swap procedure (leaf)

```
void swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
         move $s2, $a0           # save $a0 into $s2
         move $s3, $a1           # save $a1 into $s3
         move $s0, $zero         # i = 0
for1tst: slt  $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
         beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, –1       # j = i – 1
for2tst: slti $t0, $s1, 0        # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2        # $t1 = j * 4
         add  $t2, $s2, $t1      # $t2 = v + (j * 4)
         lw   $t3, 0($t2)        # $t3 = v[j]
         lw   $t4, 4($t2)        # $t4 = v[j + 1]
         slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
         move $a0, $s2           # 1st param of swap is v (old $a0)
         move $a1, $s1           # 2nd param of swap is j
         jal  swap               # call swap procedure
         addi $s1, $s1, –1       # j –= 1
         j    for2tst            # jump to test of inner loop
exit2:   addi $s0, $s0, 1        # i += 1
         j    for1tst            # jump to test of outer loop
```

Move params

Outer loop

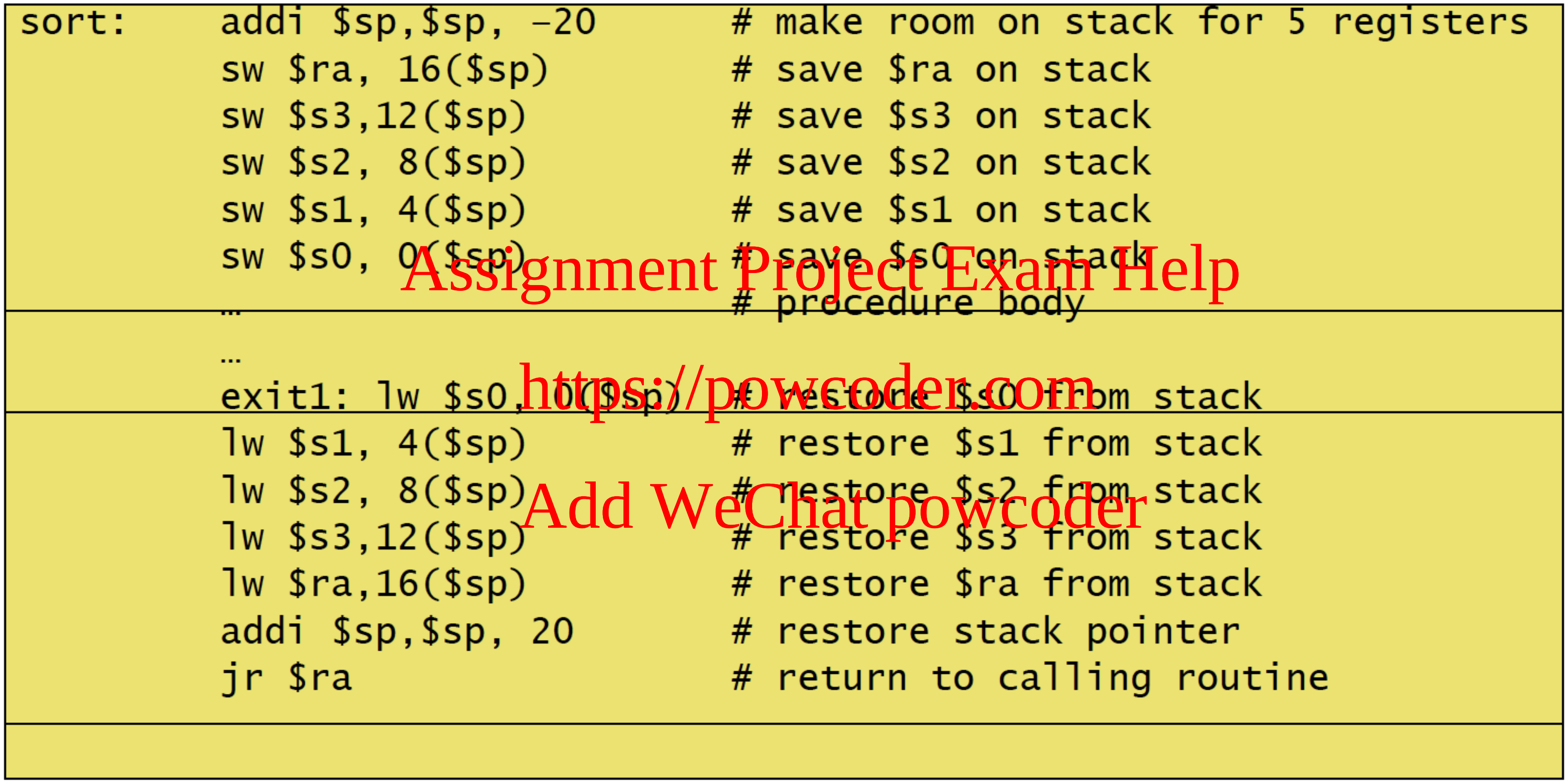
Inner loop

Pass params & call

Inner loop

Outer loop

The Full Procedure



```
sort:    addi $sp,$sp, -20         # make room on stack for 5 registers
         sw $ra, 16($sp)           # save $ra on stack
         sw $s3,12($sp)            # save $s3 on stack
         sw $s2, 8($sp)            # save $s2 on stack
         sw $s1, 4($sp)            # save $s1 on stack
         sw $s0, 0($sp)            # save $s0 on stack
         …                         # procedure body
         …
         exit1: lw $s0, 0($sp)     # restore $s0 from stack
         lw $s1, 4($sp)            # restore $s1 from stack
         lw $s2, 8($sp)            # restore $s2 from stack
         lw $s3,12($sp)            # restore $s3 from stack
         lw $ra,16($sp)            # restore $ra from stack
         addi $sp,$sp, 20          # restore stack pointer
         jr $ra                    # return to calling routine
```