UCCD1133

Introduction to Computer Organisation and Architecture

Chapter 4

Computer Architecture and Organisation Fundamental

Slides adapted from http://booksite.elsevier.com

Disclaimer

- This slide may contain copyrighted material of which has not been specifically authorized by the copyright owner. The use of copyrighted materials are solely for educational purpose. If you wish to use this copyrighted material for other purposes, you must first obtain permission from the original copyright owner.

Chapter 4 - 4

MIPS PROGRAMMING (PART 2)

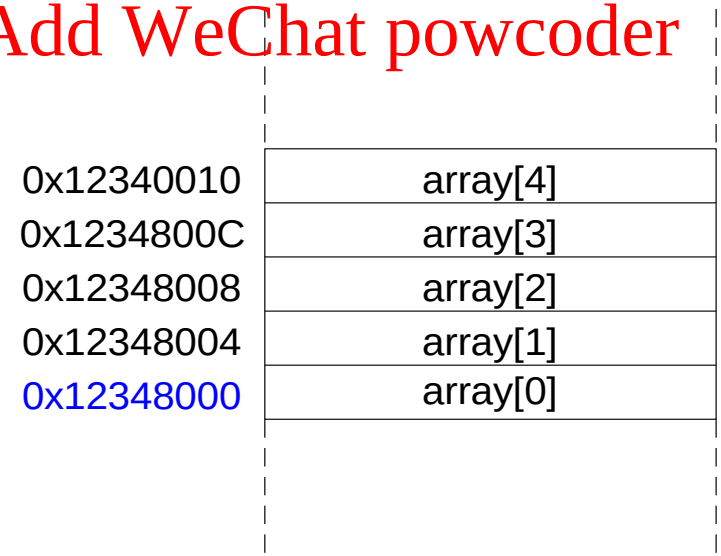
ARRAYS, STACK, FUNCTION CALL & PSEUDOINSTRUCTIONS

Arrays

- Access large amounts of similar data
- Index: access each element
- Size: number of elements

- Example: 5-element array
  - Base address = 0x12348000

    (address of first element, array[0])
  - First step in accessing an array: load base address into a register



Accessing Arrays

```
// C Code
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;
```

Accessing Arrays

```
// C Code
   int array[5];
   array[0] = array[0] * 2;
   array[1] = array[1] * 2;

# MIPS assembly code
# $s0 = array base address
   lui  $s0, 0x1234              # 0x1234 in upper half of $s0
   ori  $s0, $s0, 0x8000         # 0x8000 in lower half of $s0

   lw   $t1, 0($s0)              # $t1 = array[0]
   sll  $t1, $t1, 1              # $t1 = $t1 * 2
   sw   $t1, 0($s0)              # array[0] = $t1

   lw   $t1, 4($s0)              # $t1 = array[1]
   sll  $t1, $t1, 1              # $t1 = $t1 * 2
   sw   $t1, 4($s0)              # array[1] = $t1
```

Recall: Generating constants

Arrays using For Loops

```
// C Code
  int array[1000];
  int i;

  for (i=0; i < 1000; i = i + 1)
      array[i] = array[i] * 8;

# MIPS assembly code
# $s0 = array base address, $s1 = i
```

Arrays using For Loops

```
# MIPS assembly code
# $s0 = array base address, $s1 = i
# initialization code
  lui  $s0, 0x23B8         # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000    # $s0 = 0x23B8F000
  addi $s1, $0, 0          # i = 0
  addi $t2, $0, 1000       # $t2 = 1000
loop:
  slt  $t0, $s1, $t2       # i < 1000?
  beq  $t0, $0, done       # if not then done
  sll  $t0, $s1, 2         # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0       # address of array[i]
  lw   $t1, 0($t0)         # $t1 = array[i]
  sll  $t1, $t1, 3         # $t1 = array[i] * 8
  sw   $t1, 0($t0)         # array[i] = array[i] * 8
  addi $s1, $s1, 1         # i = i + 1
  j    loop                # repeat
done:
```

Bytes and Characters

- English characters are often represented by bytes.
  - The C language uses the type char to represent a byte or character.

- *American Standard Code for Information Interchange (ASCII)* assigns each text character a unique byte value.
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

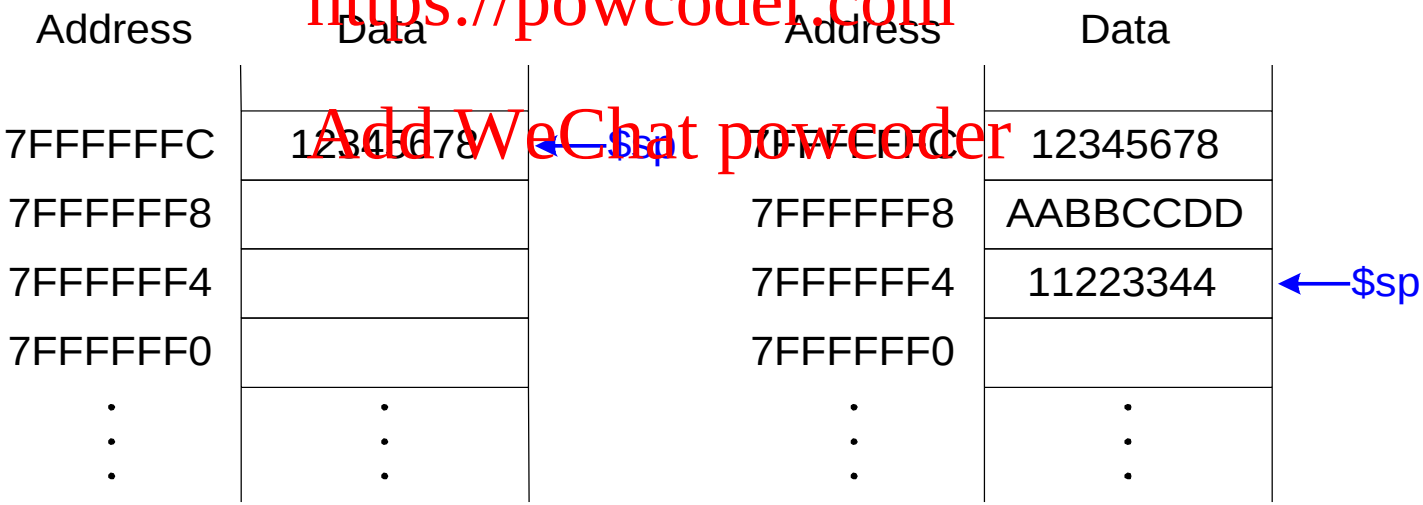
- A series of characters is called a *string*.

Cast of Characters

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

The Stack

- Memory used to temporarily save local variables within a function.
- Follow last-in-first-out (LIFO) queue, but grows down.
  - ***Expands***: uses more memory when more space needed
  - ***Contracts***: uses less memory when the space is no longer needed
- The stack pointer, `$sp`, is a special MIPS register that points to the top of the stack (most recently allocated space)



Function Calls

- A ***procedure*** or ***function*** is one tool programmers use to structure programs.
  - Easier to understand
  - Allow code to be reused.

- Parameters act as an interface between the **caller** and **callee**, since they can pass values and return results.
  - **Caller:** calling function (in this case, `main`)
  - **Callee:** called function (in this case, `sum`)

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee

- **Callee:**
  - performs the function
  - returns result to caller
  - returns to point of call
  - must not overwrite registers or memory needed by caller

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Function Calls

A ***function*** call must follow these six steps:

1. Put parameters in a place (registers) where the callee can access them.
2. Transfer control to the callee.
3. Acquire the storage resources needed for the callee.
4. Perform the desired task.
5. Put the result value in a place (registers) where the caller can access it.
6. Return to place of call.

MIPS Function Conventions

- **Arguments**: `$a0 – $a3`
  - Four argument registers in which to pass parameters
- **Return value**: `$v0`
  - Two value registers in which to return values
- **Return address**: `$ra`
  - The return address register to return to the point of origin
- **Call Function:** jump and link (`jal`)
  - A special instruction for function call
  - Jumps to an address and simultaneously saves the address of the following instruction (*PC*+4) in register `$ra`
  - Example: `jal FunctionAddress`
- **Return** from function: jump register (`jr`)
  - Needed for function return or switch statements.
  - Example: `jr $ra`

    Copies `$ra` to program counter.
- **Program Counter** (*PC*)
  - A special register to hold the address of the current instruction being executed.

C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

MIPS assembly code

```
# $s0 = y

main:
   ...
   addi $a0, $0, 2     # argument 0 = 2
   addi $a1, $0, 3     # argument 1 = 3
   addi $a2, $0, 4     # argument 2 = 4
   addi $a3, $0, 5     # argument 3 = 5
   jal  diffofsums     # call function
   add  $s0, $v0, $0   # y = returned value
   ...

# $s0 = result
diffofsums:
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # result = (f + g) - (h + i)
   add $v0, $s0, $0    # put return value in $v0
   jr  $ra             # return to caller
```

Example

MIPS assembly code

```
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

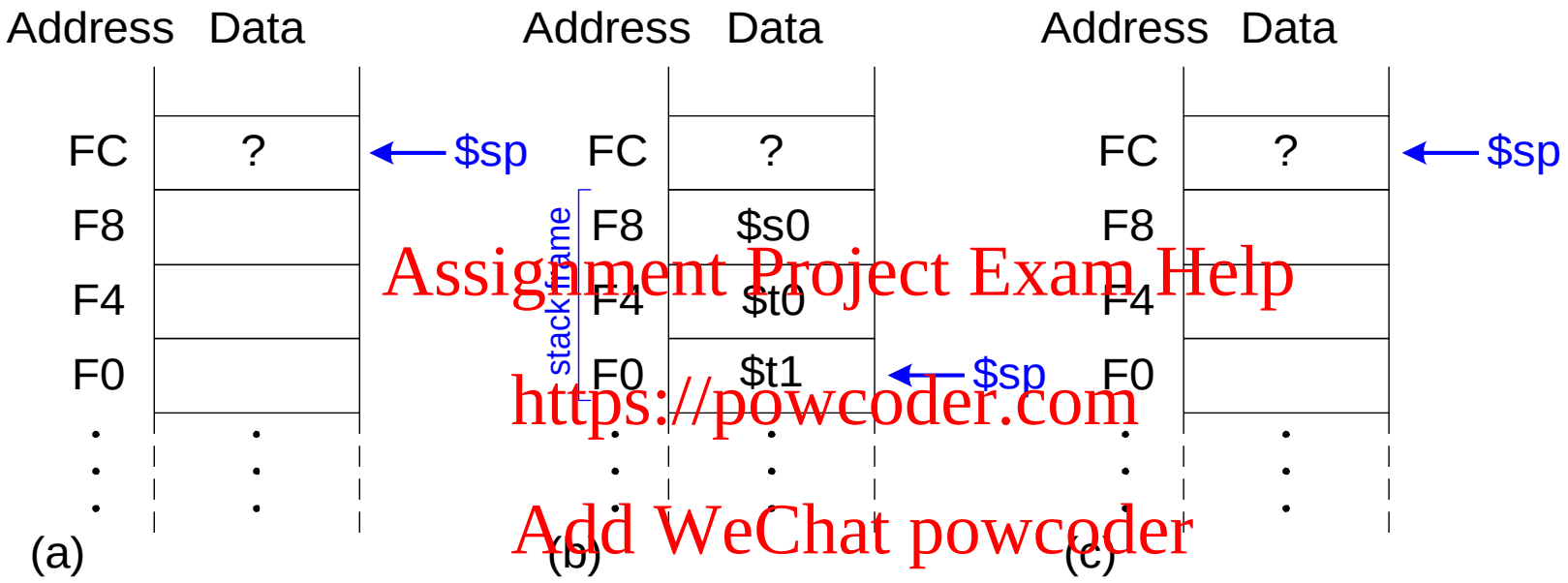

* However, note that

- `diffofsums` overwrote 3 registers: `$t0`, `$t1`, `$s0`
- Called functions must have no unintended side effects
- `diffofsums` can use *stack* to temporarily store registers

Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12   # make space on stack
                       # to store 3 registers
  sw   $s0, 8($sp)     # save $s0 on stack
  sw   $t0, 4($sp)     # save $t0 on stack
  sw   $t1, 0($sp)     # save $t1 on stack
  add  $t0, $a0, $a1   # $t0 = f + g
  add  $t1, $a2, $a3   # $t1 = h + i
  sub  $s0, $t0, $t1   # result = (f + g) - (h + i)
  add  $v0, $s0, $0    # put return value in $v0
  lw   $t1, 0($sp)     # restore $t1 from stack
  lw   $t0, 4($sp)     # restore $t0 from stack
  lw   $s0, 8($sp)     # restore $s0 from stack
  addi $sp, $sp, 12    # deallocate stack space
  jr   $ra             # return to caller
```

The stack during `diffofsums` Call



Registers

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
$s0-$s7	**$t0-$t9**
$ra	**$a0-$a3**
$sp	**$v0-$v1**
stack above **$sp**	stack below **$sp**

- The stack above `$sp` is preserved simply by making sure the callee does not write above `$sp`.

Recursive Function Call

- A function that does not call others is called a ***leaf*** function
  - Example: diffofsums.

- A function that does call others is called a ***nonleaf*** function.
  - More complicated.
  - Need to save non-preserved registers on the stack before they call another function.

- A **recursive function** is a nonleaf function that calls itself.
  - Example: factorial
    - *Factorial(n)* = $n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$

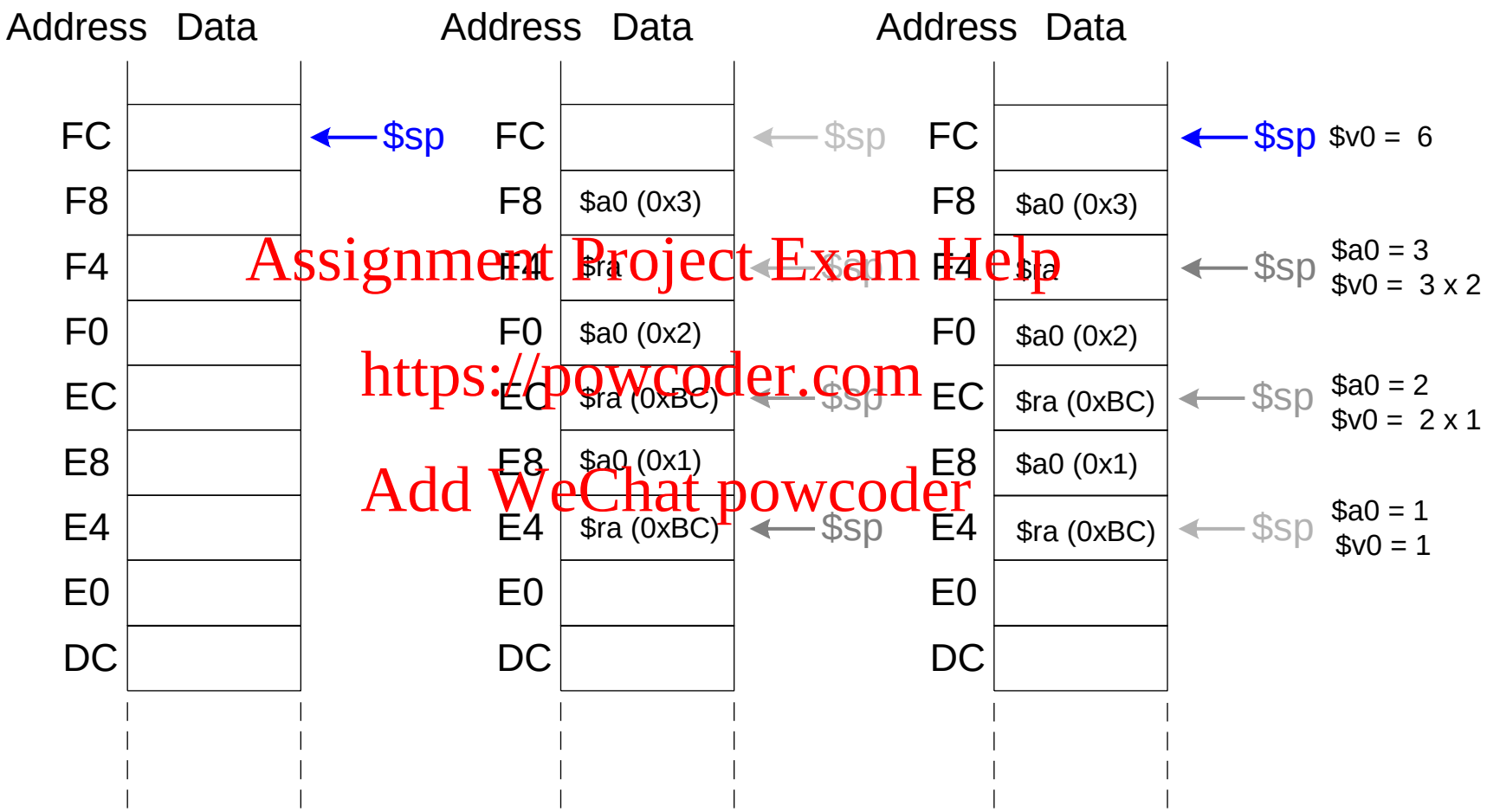
High-level code

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

MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0  # a <= 1 ?
0xA4            beq  $t0, $0, else  # no: go to else
0xA8            addi $v0, $0, 1     # yes: return 1
0xAC            addi $sp, $sp, 8    # restore $sp
0xB0            jr   $ra            # return
0xB4       else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial      # recursive call
0xBC            lw   $ra, 0($sp)    # restore $ra
0xC0            lw   $a0, 4($sp)    # restore $a0
0xC4            addi $sp, $sp, 8    # restore $sp
0xC8            mul  $v0, $a0, $v0  # n * factorial(n-1)
0xCC            jr   $ra            # return
```

Stack During Recursive Call



Function Call Summary

- **Caller**
  - Put arguments in `$a0-$a3`
  - Save any needed registers (`$ra`, maybe `$t0-t9`)
  - `jal callee`
  - Restore registers
  - Look for result in `$v0`
- **Callee**
  - Save registers that might be disturbed (`$s0-$s7`)
  - Perform function
  - Put result in `$v0`
  - Restore registers
  - `jr $ra`

Pseudoinstructions

- MIPS defines **pseudointructions** that are not actually part of the instruction set but are commonly used by programmers and compilers.
- When converted to machine code, pseudoinstructions are translated into one or more MIPS instructions.

Pseudoinstruction	MIPS Instructions
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, $s0, 0xAA77`
`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`