Chapter 2

Instructions: Language of the Computer

Note regarding the previous example

- In the previous example, since the caller does not expect registers $t0 and $t1 to be preserved across a procedure call, we can drop two stores and two loads from the code.

- We still must save and restore $s0, since the callee must assume that the caller needs its value.

Nested Procedures

- Suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register $a0 and then using jal A.

- Then suppose that procedure A calls procedure B via jal B with an argument of 7, also placed in $a0.

- Since A hasn't finished its task yet, there is a conflict over the use of register $a0.

- Similarly, there is a conflict over the return address in register $ra, since it now has the return address for B.

- Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

Nested Procedures

- Solution for previous Example: push registers that must be preserved to the stack

1. **The caller pushes** any argument registers ($a0–$a3) or temporary registers ($t0–$t9) that are needed after the call.
2. **The callee pushes** the return address register $ra and any saved registers ($s0–$s7) used by the callee.
3. **The stack pointer $sp is adjusted** to account *for the number of registers placed on the stack.*
4. **Upon the return, the registers are restored from memory and the stack pointer is readjusted.**

Example- compiling a recursive c procedure, showing nested procedure linking

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

What is the MIPS assembly code?

The parameter variable n corresponds to the argument register $a0.

```
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
```

The next two instructions test whether n is less than 1, going to L1 if n ≥ 1. If n is less than 1, fact returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in $v0.

Before popping two items off the stack, we could have loaded $a0 and $ra. Since $a0 and $ra don't change when n is less than 1, we skip those instructions.

```
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
```

Example- compiling a recursive c procedure, showing nested procedure linking

If n is not less than 1, the argument n is decremented and then fact is called again with the decremented value:

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

The next instruction is where fact returns. Now the old return address and old argument are restored, along with the stack pointer:

```
lw   $a0, 0($sp)  # return from jal: restore argument n
lw   $ra, 4($sp)  # restore the return address
addi $sp, $sp, 8  # adjust stack pointer to pop 2 items
```

Next, the value register $v0 gets the product of old argument $a0 and the current value of the value register.

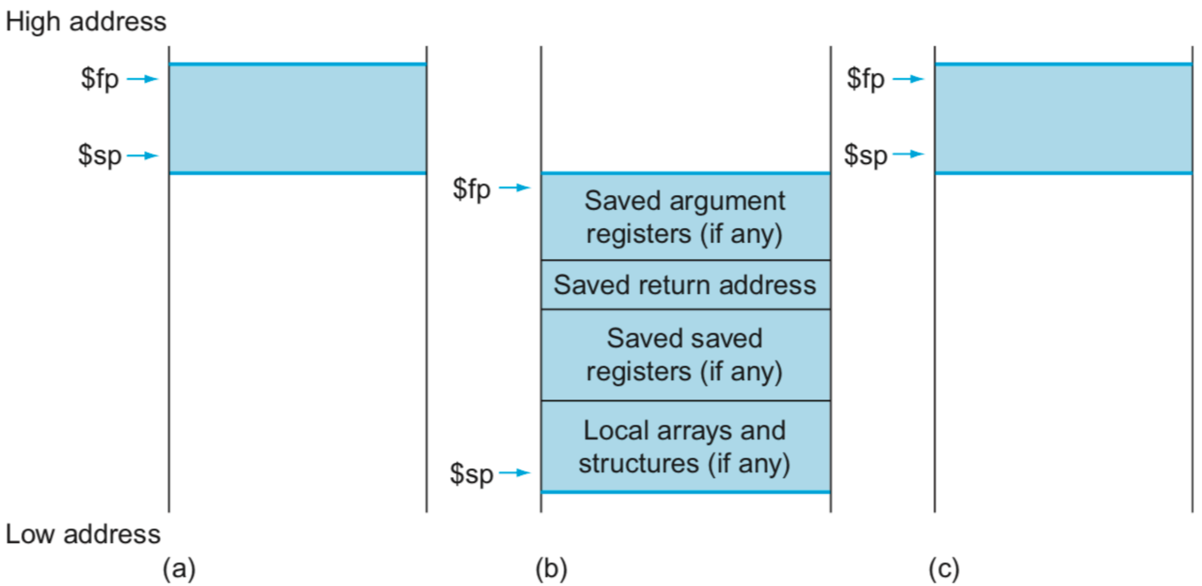
```
mul  $v0,$a0,$v0   # return n * fact (n - 1)
```

Finally, fact jumps again to the return address:

```
jr   $ra           # return to the caller
```

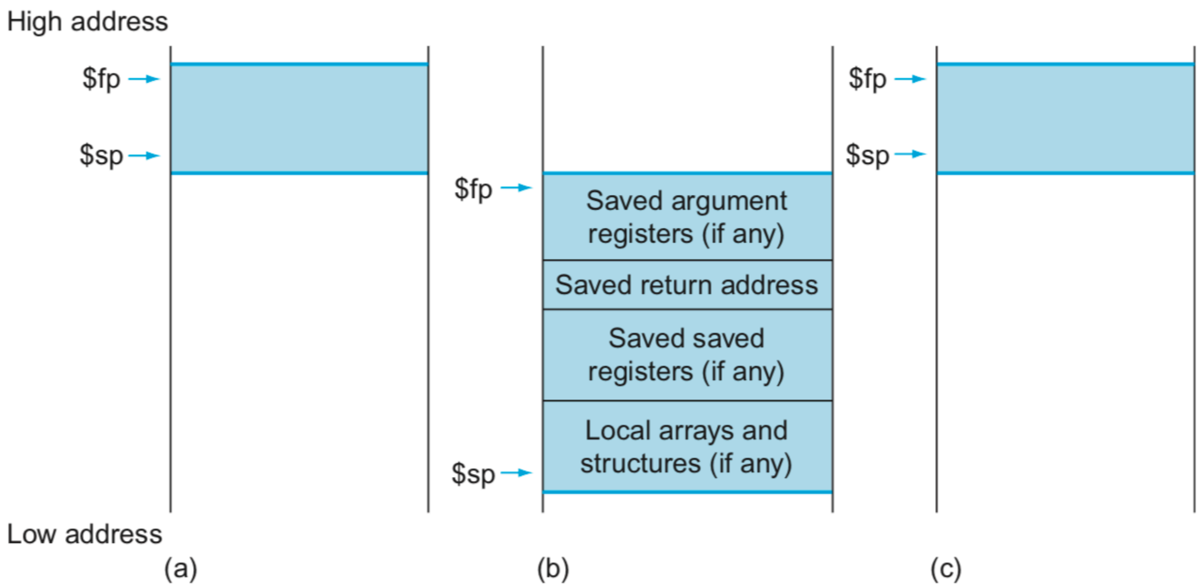
Allocating space for new data on the stack

- Stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures.
- The segment of the stack containing a procedure's saved registers and local variables is called a **procedure frame** or **activation record**.
- MIPS software uses a **frame pointer** ($fp) to point to the first word of the frame of a procedure.



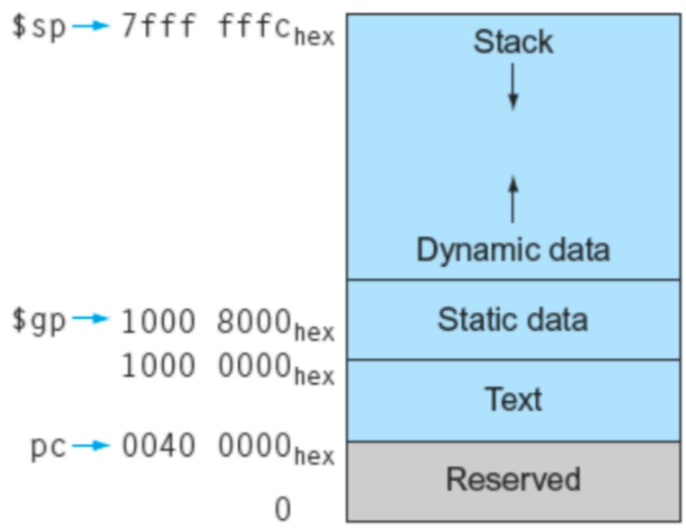
Allocating space for new data on the stack

- A stack pointer might change during the procedure
- Alternatively, a frame pointer offers a stable base register within a procedure for local memory-references.
- We've been avoiding using $fp by avoiding changes to $sp within a procedure: in our examples, the stack is adjusted only on entry and exit of the procedure.



Allocating space for new data on the heap

- The first low part of memory is reserved, followed by MIPS machine code, called text segment
- Static data is a place for static data, like array
- Dynamic data is place that its size changed dynamically, such as link list. We call this section heap



Communicating with people

- Most computers offer 8 bit bytes to represent characters, with American Standard Code for Information Interchange (ASCII)

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Communicating with people

- We could represent numbers as strings of ASCII digits instead of as integers. How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

- One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be (10 *8) /32 or 2.5.

Communicating with people

- **A series of instructions can extract a byte from a word,** so load word and store word are sufficient for transferring bytes as well as words
- ***Load byte* (lb)** loads a byte from memory, placing it in the rightmost 8 bits of a register.
- ***Store byte* (sb)** takes a byte from the rightmost 8 bits of a register and writes it to memory.
- Thus, we copy a byte with the sequence

```
lb $t0,0($sp)     # read a byte from memory
sb $t0,0($sp)     #  write a byte to memory
```

Communicating with people

- **There are three choices for representing a string:**

1. The first position of the string is reserved to give the length of a string,
2. An accompanying variable has the length of the string (as in a structure),
3. The last position of a string is indicated by a character used to mark the end of a string.

- **C uses the third choice, terminating a string with a byte whose value is 0 (named null in ASCII). Thus, the string “Cal” is represented in C by the following 4 bytes, shown as decimal numbers: 67, 97, 108, 0. (As we shall see, Java uses the first option.)**

Compiling a string copy procedure, showing how to use C strings

- The procedure strcpy copies string y to string x using the null byte termination convention of C. What is the MIPS assembly code?

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
    i += 1;
}
```

- Assume that base addresses for arrays x and y are found in $a0 and $a1, while i is in $s0. strcpy adjusts the stack pointer and then saves the saved register $s0 on the stack:

```
strcpy:
    addi   $sp,$sp,-4    # adjust stack for 1 more item
    sw     $s0, 0($sp)   # save $s0
```

- To initialize i to 0, the next instruction sets $s0 to 0 by adding 0 to 0 and placing that sum in $s0:

```
    add    $s0,$zero,$zero # i = 0 + 0
```

- This is the beginning of the loop. The address of y[i] is first formed by adding i to y[]:

```
L1: add    $t1,$s0,$a1  # address of y[i] in $t1
```

Compiling a string copy procedure, showing how to use C strings

- To load the character in y[i], we use load byte unsigned, which puts the character into $t2:

```
lbu   $t2, 0($t1)  # $t2 = y[i]
```

- A similar address calculation puts the address of x[i] in $t3, and then the character in $t2 is stored at that address.

```
add   $t3,$s0,$a0  # address of x[i] in $t3
sb    $t2, 0($t3)  # x[i] = y[i]
```

- Next, we exit the loop if the character was 0. That is, we exit if it is the last character of the string:

```
beq   $t2,$zero,L2 # if y[i] == 0, go to L2
```

- If not, we increment i and loop back:

```
addi  $s0, $s0,1    # i = i + 1
j     L1            # go to L1
```

- If we don't loop back, it was the last character of the string; we restore $s0 and the stack pointer, and then return

```
L2: lw     $s0, 0($sp)  # y[i] == 0: end of string.
                        # Restore old $s0
    addi   $sp,$sp,4    # pop 1 word off stack
    jr     $ra          # return
```

Reading Assignment

- Reading assignment: Read 2.8 and 2.9 of the textbook