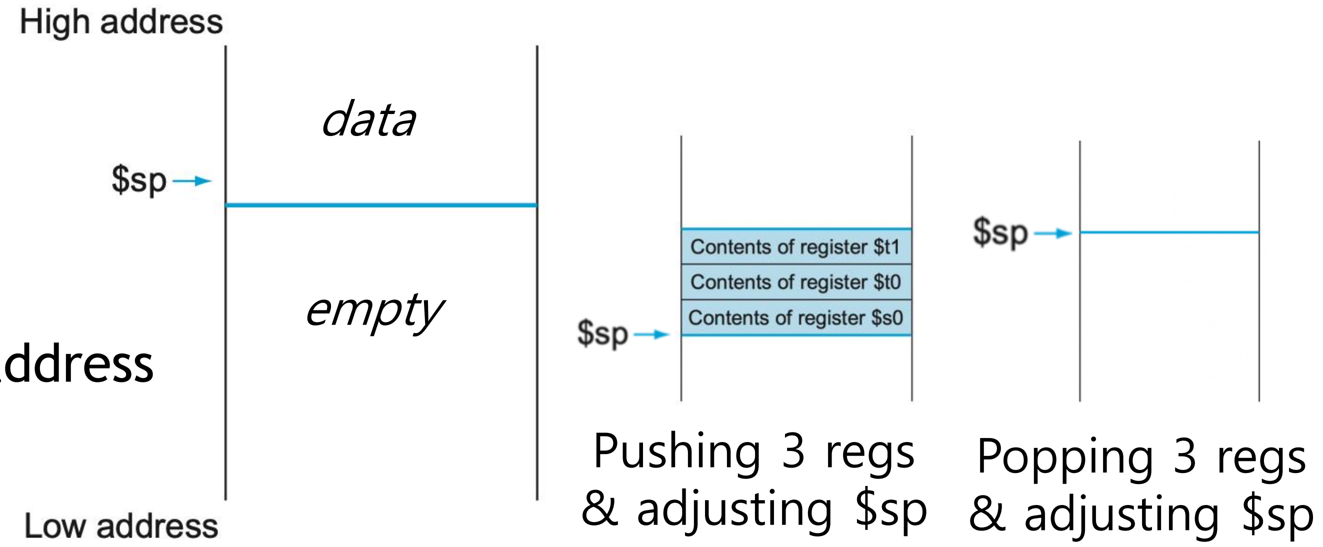
Computer Architecture
(ENE1004)

Lec – 7: Instructions: Language of the Computer (Chapter 2) - 6

Review: Using Stack for Procedure Call

- Question: Are **$a0—$a3** and **$v0—$v1** enough for a callee to work with?
  - What happens if callee uses **$s** or **$t**, which are being used by caller?
  - If so, once the procedure is returned, such registers (**$s** or **$t**) may be polluted
  - Registers must be restored to the values that they contained before the procedure was invoked
- Solution: Such register values are kept in an area of memory, called stack
  - Stack grows from higher to lower addresses
  - A last-in-first-out queue
    - Push: placing (storing) data onto the stack
    - Pop: removing (deleting) data from the stack
  - Stack pointer holds most recently allocated address
    - MIPS reserves **$sp** (register 29) for stack pointer
    - **$sp** is adjusted when pushing and popping
    - **$sp** is decremented by 4 when pushing a register
    - **$sp** is incremented by 4 when popping a register



Stack area of memory

Pushing 3 regs & adjusting $sp

Popping 3 regs & adjusting $sp

Review: Using Stack for Procedure Call: Example

```
int leaf_example (int g, int h, int i, int j)
{
     int f;

     f = (g + h) - (i + j);
     return f;
}
```

is translated into

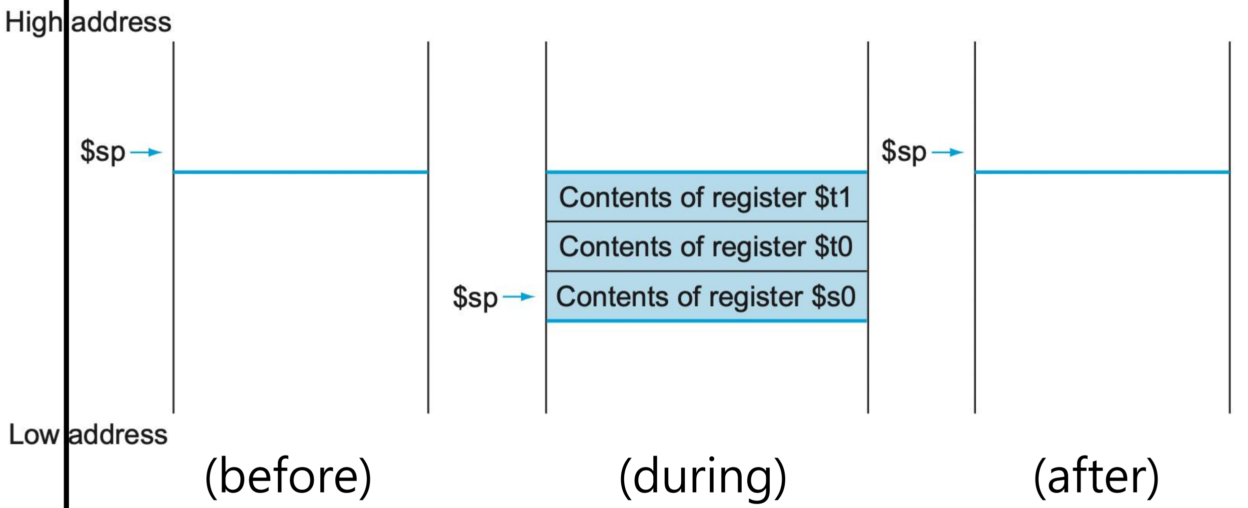
```
leaf_example:

add  $t0,  $a0,  $a1        # register $t0 contains g + h
add  $t1,  $a2,  $a3        # register $t1 contains i + j
sub  $s0,  $t0,  $t1        # f = $t0 - $t1

add  $v0,  $s0,  $zero      # returns f

jr $ra                      # jump back to caller
```

- Assumption
  - g, h, i, and j correspond to **$a0, $a1, $a2, and $a3**
  - f corresponds to **$s0**
- Caller sets argument registers
  - E.g., **add $a0, $t0, $zero**
  - E.g., **addi $a1, $zero, 6**
- Caller executes **jal leaf_example**
  - **$ra** ← PC + 4
  - PC ← leaf_example

Review: Using Stack for Procedure Call: Example

```
leaf_example:

addi  $sp,  $sp,  -12      # adjust stack to make room for 3 items
sw  $t1,  8($sp)           # save $t1 for use afterwards
sw  $t0,  4($sp)           # save $t0 for use afterwards
sw  $s0,  0($sp)           # save $s0 for use afterwards

add  $t0,  $a0,  $a1       # register $t0 contains g + h
add  $t1,  $a2,  $a3       # register $t1 contains i + j
sub  $s0,  $t0,  $t1       # f = $t0 - $t1

add  $v0,  $s0,  $zero     # returns f

lw  $s0,  0($sp)           # restore $s0 for caller
lw  $t0,  4($sp)           # restore $t0 for caller
lw  $t1,  8($sp)           # restore $t1 for caller
addi  $sp,  $sp,  12       # adjust stack to delete 3 items

jr $ra                     # jump back to caller
```

- What if **$t0, $t1, $s0** are holding data needed by caller afterwards?
  - After returning, program malfunctions
- The three register data can be protected by keeping them in stack
  - Pushing the values before using them
  - Popping them when returning
- **$sp** must be adjusted correspondingly



Saved vs Temporary Registers

- Then, do we need to save and restore all the registers whenever a function is called?
  - Actually, we do not have to save and restore registers whose values are never used
- To avoid such unnecessary saving/restoring, MIPS separates registers into two groups
- Temporary registers ($t0—$t9)
  - These registers are not preserved by the callee on a procedure call
- Saved registers ($s0—$s7)
  - These registers must be preserved on a procedure call
  - If used, the callee saves and restores them

addi $sp, $sp, ~~-12~~ -4
~~sw $t1, 8($sp)~~
~~sw $t0, 4($sp)~~
sw $s0, 0($sp)

lw $s0, 0($sp)
~~lw $t0, 4($sp)~~
~~lw $t1, 8($sp)~~
addi $sp, $sp, ~~12~~ 4

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));
}
```

is translated into

- 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

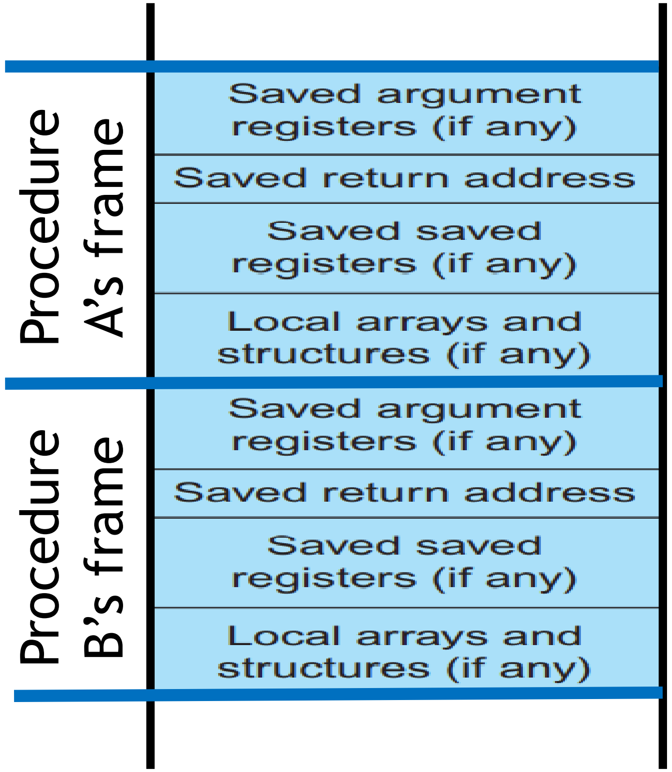
with an argument of n * fact (n-1)

```
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

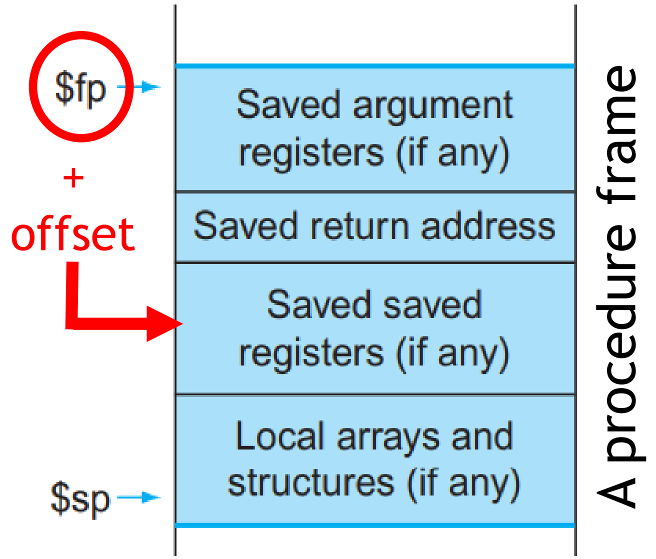
- Procedures may use local arrays or structures, which do not fit in registers
  - Such variables can be stored in the stack (in addition to the registers)
- Stack data can be segmented into procedure frames (or activation records)
  - Procedure frame (activation record) is a segment containing a procedure's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

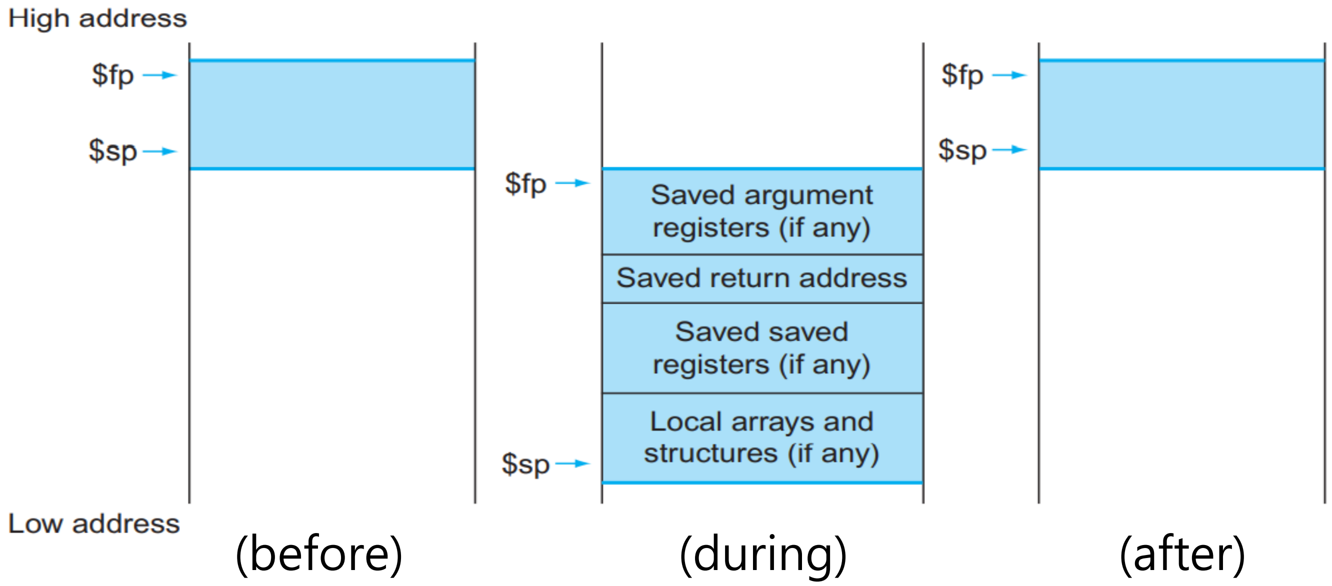


- Assumption: procedure A calls procedure B
- All the registers and local variables of a procedure are ke pt within its procedure frame
  - Argument registers (**$a0-$a3**)
  - Return address register (**$ra**)
  - Saved registers (**$s0-$s7**)
  - Local variables
- Whenever a procedure is invoked or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

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

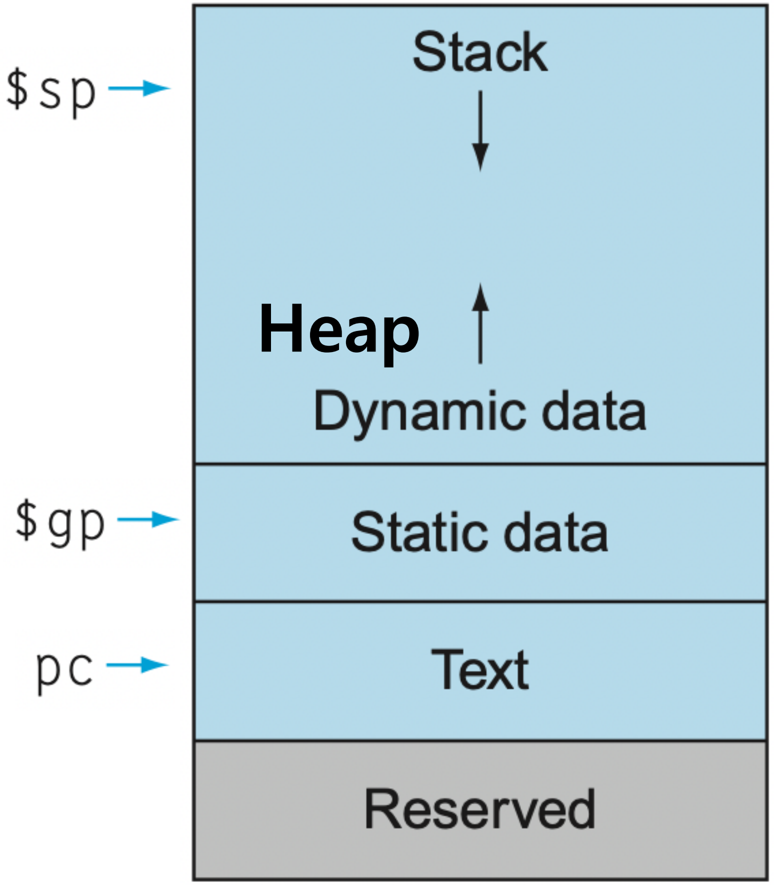
- It may be hard to use **$sp** to locate a desired data within a procedure frame
  - A data within a procedure frame can be located by "$sp + offset" - e.g., **4($sp)**
  - However, **$sp** might change during the procedure
- MIPS offers a frame pointer (**$fp**) that is a stable base register within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