Computer Organization

Lecture 7 -
Instruction Sets: Software Concerns

Photo Credit: Kamala via Flickr

Reading:

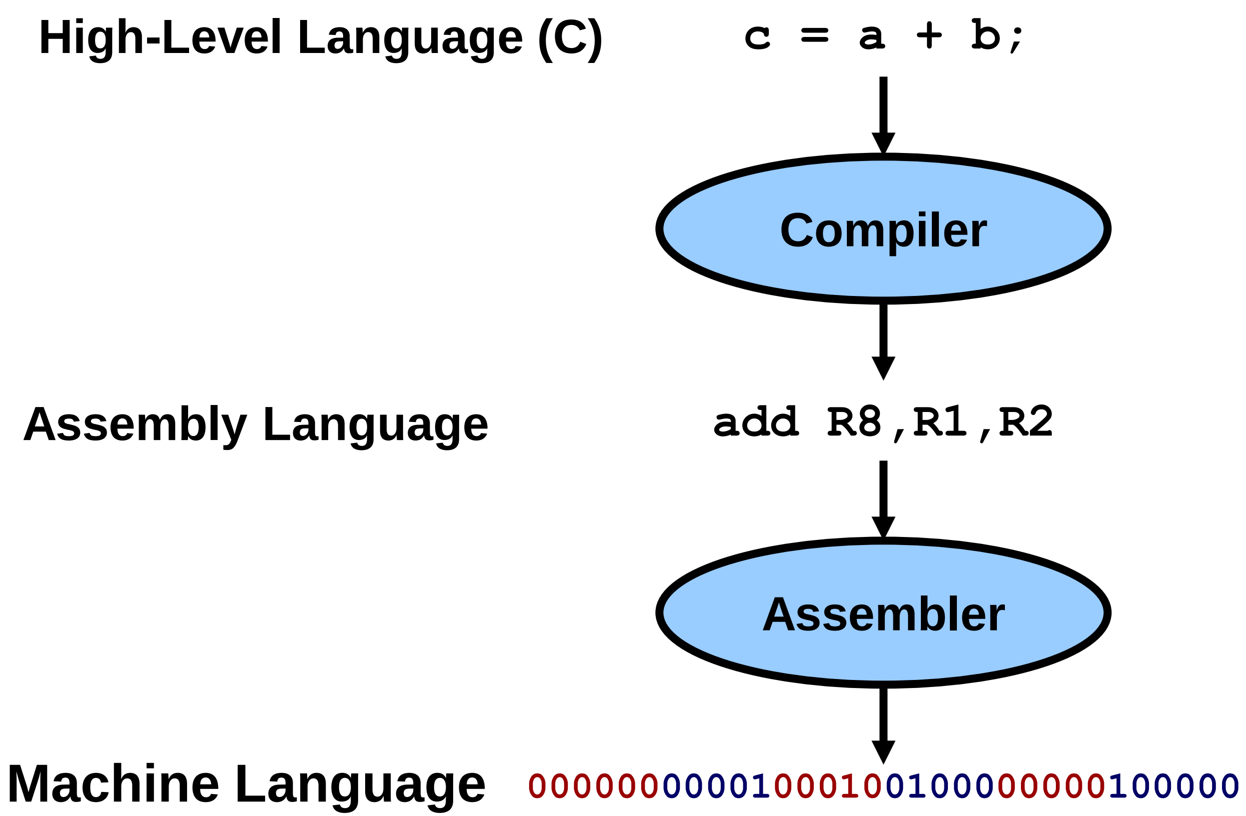
Homework: see uclass

*use corrected code for 2.4.4-2.4.6 and 2.13.1 (see Lecture 5)

Outline - Instruction Sets

- Instruction Set Overview
- MIPS Instruction Set
- Software Concerns
  - Compiling C Constructs
  - Procedures (Subroutines) and Stacks
- Summary

Review - Compiler & Assembler



Using MIPS Instructions

- **How does a compiler generate code for…**
  - **`if` statements?**
  - **Loops?**
  - **`switch` Statements?**

Example: Compiling C `if-then-else`

- **Example**

C Code

```
if (i==j)
    f = g + h;
else
    f = g - h;
```

Assume the following:
(1) i and j are in registers $s3 and $s4, respectively.
(2) g and h are in registers $s1 and $s2, respectively

Assembly

```
       bne $s3, $s4, Else
       add $s0, $s1, $s2
       j Exit;              # unconditional jump
Else:  sub $s0, $s1, $s2
Exit:
```

Compiling C Loops

- Simple Loop Example - C Code

```
Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
```

$s1 ← 0 //g
$s2 ← h
$s3 ← 0 //i
$s4 ← j
$s5 ← &A[0]

	0	1	2	3	4	5	6	7	8	9
A[10]	10	15	20	25	30	35	40	45	50	55

j = 2
h = 8

- Assembly Language

g = g + A[i]

```
Loop: sll $t1,$s3,2      # scale i for byte offset
      add $t1,$t1,$s5    # calculate address of A[i]
      lw $t0,0($t1)      # $t0 = A[i]
      add $s1,$s1,$t0    # g = g + A[i]
      add $s3,$s3,$s4    # i = i + j
      bne $s3,$s2, Loop
```

Compiling C Loops (cont'd)

- **`while` loop example - C Code**

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

save[10]
k = 10

0	1	2	3	4	5	6	7	8	9
10	10	10	10	10	20	20	20	20	20

$s3 ← 0 // i
$s5 ← k
$s6 ← &save[0]

- **Assembly Language**

save[i] == k

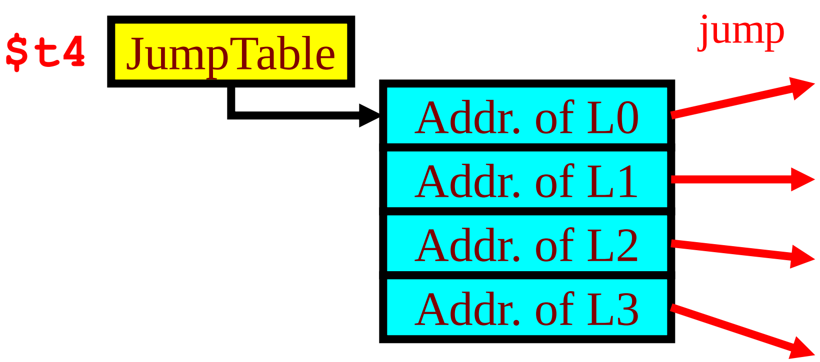
```
Loop: sll $t1,$s3, 2      # $t1 = i * 4
      add $t1,$t1,$s6     # calculate addr of save[i]
      lw $t0,0($t1)       # $t0 = save[i]
      bne $t0,$s5, Exit   # if (save[i] != k)
      addi $s3,$s3,1      # i = i + 1
      j Loop
Exit: ...
```

Compiling C `switch` statements

- `switch` statement selects one of many alternatives

```
switch(k) {
        case 0: f=i+j; break;
        case 1: f=g+h; break;    // if k = 1?
        case 2: f=g-h; break;
        case 3: f=i-j; break;
}
```

$t4 ← base address of jump table




```
L0:    add $s0,$s3,$s4     # f=i+j;
       j Exit;             # break;
L1:    add $s0,$s1,$s2     # f=g+h;
       j Exit;             # break;
L2:    sub $s0,$s1,$s2     # f=g-h;
       j Exit;             # break;
L3:    sub $s0,$s3,$s4     # f=i-j;
Exit:  …
```

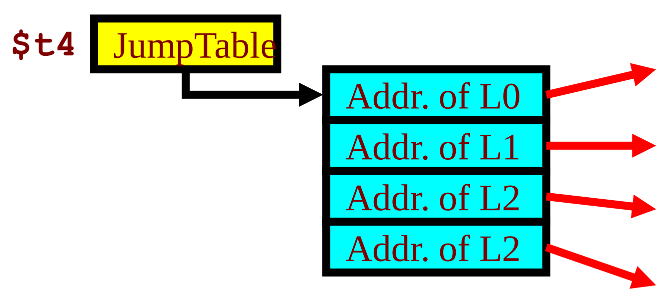
allows k=0, 1, 2, 3

$s5 = k
$t4 = jumpTable

Compiling C `switch` statements

- `switch` implementation: jump table access

```
slt  $t3,$s5,$zero   # test if k<0 (bounds check)
bne  $t3,$zero,Exit
slti $t3,$s5,4       # test if k≥4 (bounds check)
beq $t3,$zero,Exit

sll $t1,$s5,2        # scale k for instr. ofset
add $t1,$t1,$t4      # t4 - jump table base address
lw $t0,0($t1)
jr $t0               # new instruction: "jump reg."
                                            (R-format)
L0:   . . .
L1:   . . .
L2:   . . .
L3:   . . .
Exit:
```



Outline - Instruction Sets

- Instruction Set Overview
- MIPS Instruction Set
- Software Concerns
  - Compiling C Constructs
  - Procedures (Subroutines) and Stacks
  - Case Study: Internet Worms
  - Multicore Support: Synchronization
  - Compiling, Linking, and Loading
  - Example: String Processing / SPIM Demo
- Summary

Procedures

- Common tool to structure programs
  - Break program up into pieces
  - "Main program" calls procedures
  - Procedures may call other procedures
  - Recursion: procedure calls itself

- Other names for procedures
  - **Functions** (usually with a return value)
  - **Subroutines**
  - **Methods** (when associated with a class/object e.g. Java)

Example - C Functions

```
main(char argc, *argv[]) {
  DTYPE data;

  data = read_input(argv[0]));

  process_data(stuff);

  write_output(stuff)

}
```

Return Type

Arguments

```
DTYPE read_input(char* filename) {
    ...
  return data;
}
```

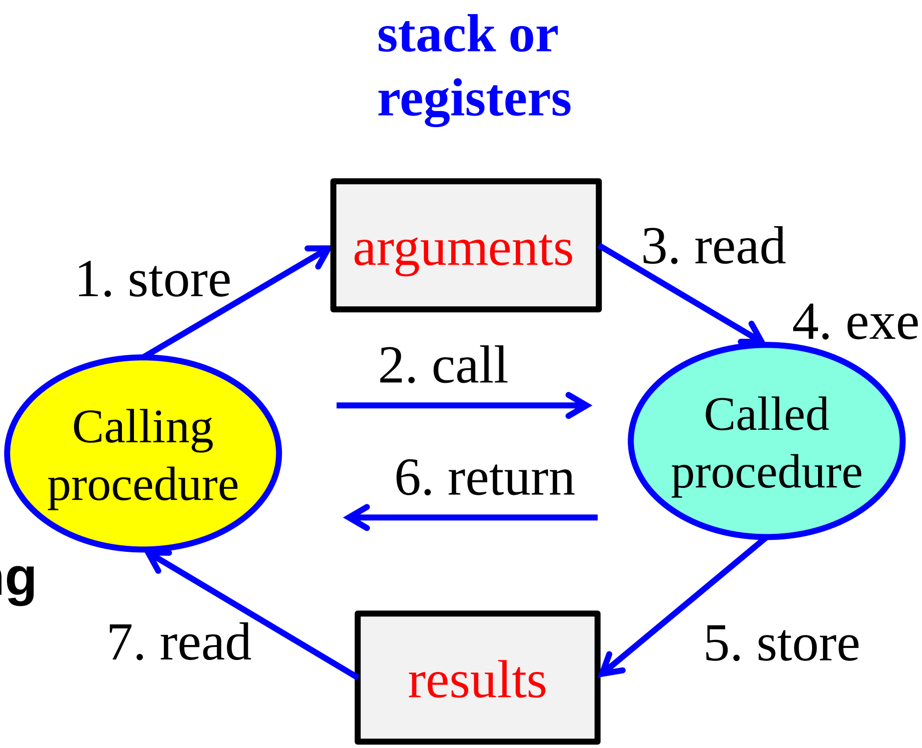
Return Value

```
void process_data(DTYPE pdata) {

    ...

}
```

```
void write_data(DTYPE wdata) {
    ...
}
```

Steps Required to Call a Procedure

1. **Store arguments where called procedure can access them**
2. **Transfer control to called procedure**
3. **Acquire arguments needed for procedure**
4. **Perform desired task**
5. **Place return value where calling procedure can access**
6. **Return control to calling procedure**
7. **Read the results**



전역변수의 의미?

Procedure Calls in MIPS

- Register conventions to support procedures:
  - `$a0-$a3` - four argument registers (`$4-$7`)
  - `$v0-$v1` - two return value registers (`$2-$3`)
  - `$t0-$t9`: temporary registers for variables (`$8-$15`)
    - Can be overwritten by callee
  - `$s0-$s7`: saved registers for variables (`$16-$23`)
    - Must be saved/restored by callee
  - `$gp`: global pointer for static data (`$28`)
  - `$sp`: stack pointer (`$29`)
  - `$fp`: frame pointer (`$30`)
  - `$ra` - one return address register (`$31`)

Procedure Calls in MIPS

- **Calling procedures - jump and link (jal)**

```
jal ProcedureAddress          // J-Format instr.
...
```

effect:

```
$ra = PC + 4
PC = ProcedureAddress
```

PC + 8 in "real" MIPS due to delay slot

- **Returning from procedure - "jump register (jr)**

```
jr $ra
```

effect:

```
PC = $ra
```

Procedure Calls - Additional Concerns

- Q: What happens when…
  - We need more storage than registers can provide?
    - More than four arguments `(arg. registers: $a0-$a3)`
    - Local array variables
    - Data “spilled” from registers (see example - p. 114)
  - We have <u>nested</u> procedure calls (`f()` calls `g()`)?
  - We have <u>recursive</u> procedure calls (`f()` calls itself)?

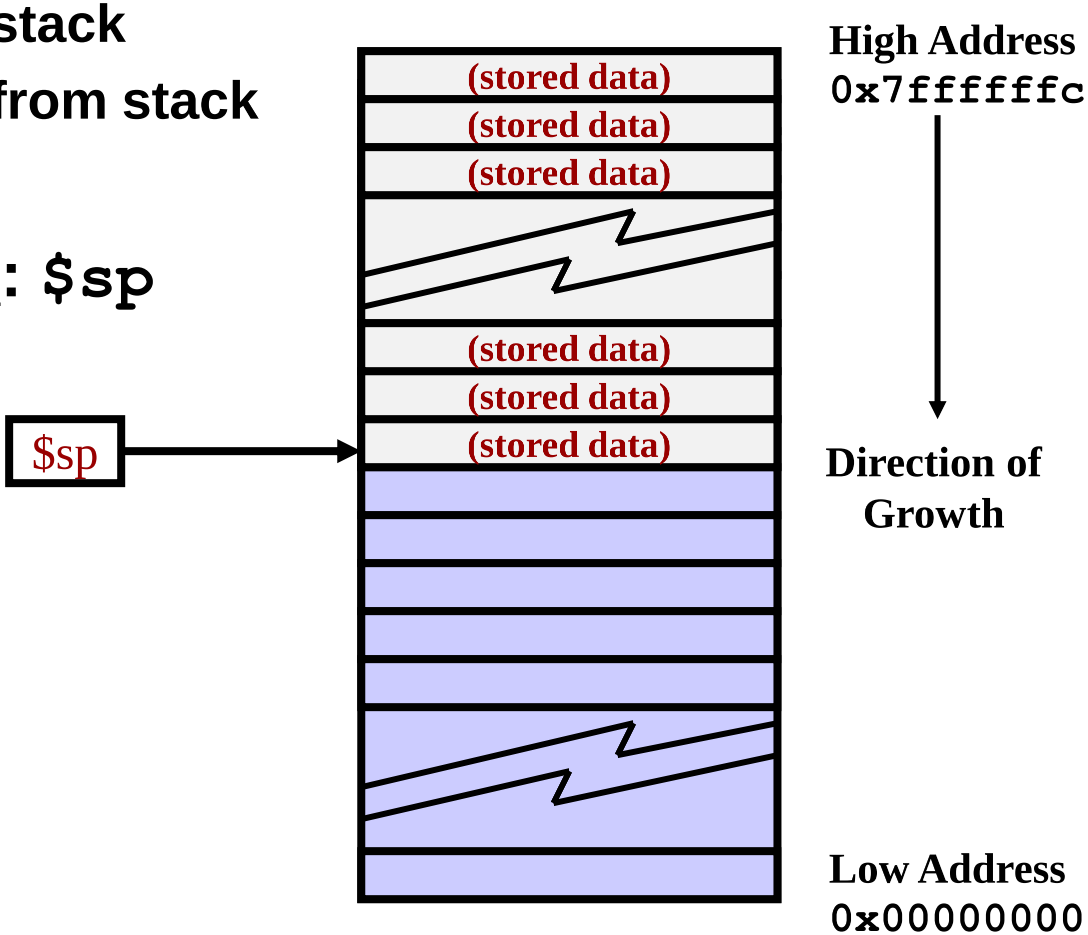
- A: We use the <u>stack</u>

공유메모리?
- register file
- stack

Using the Stack

- **Stack operations**
  - **Push - add data to stack**
  - **Pop - remove data from stack**

- **MIPS <u>stack pointer</u>: `$sp`**



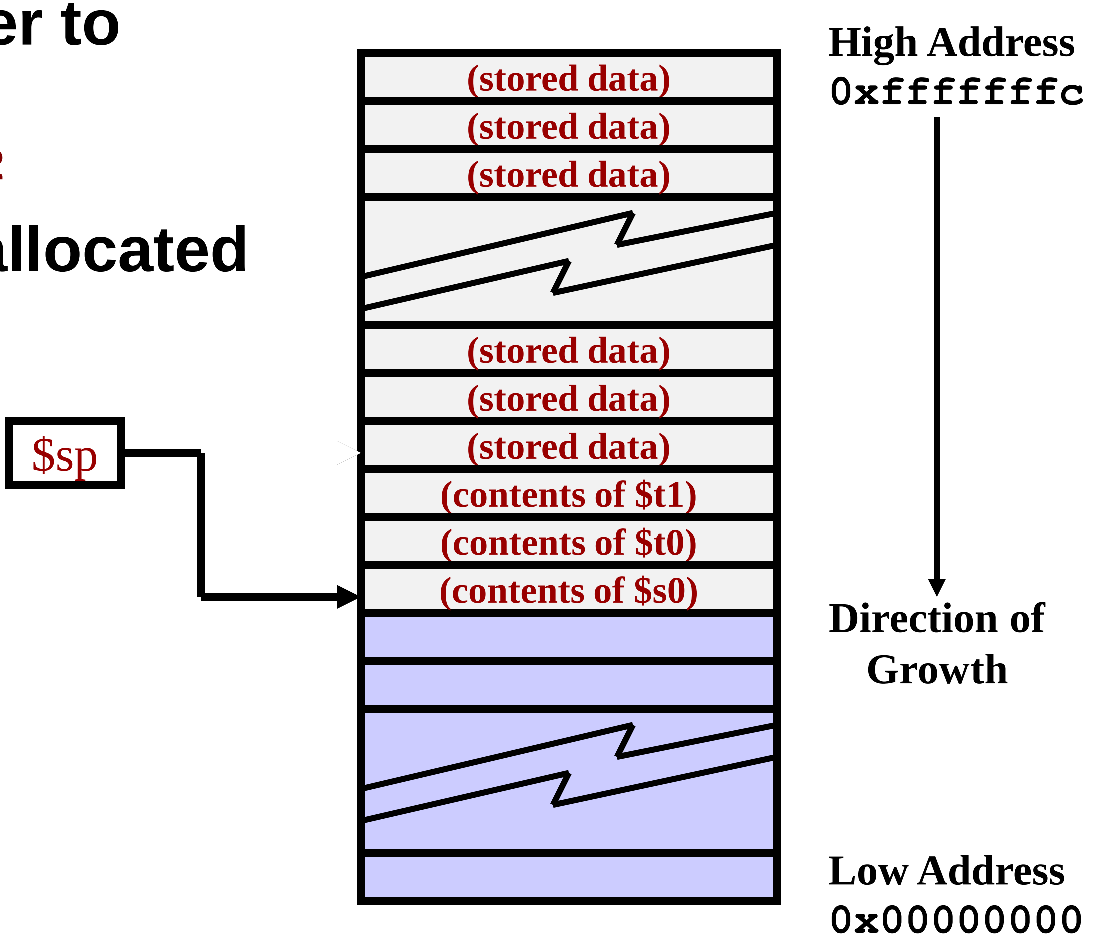
"Push" - Adding Data to the Stack

- **Adjust stack pointer to allocate space**

```
addi $sp,$sp,-12
```

- **Store registers in allocated space**

```
sw $t1,8($sp)
sw $t0,4($sp)
sw $s0,0($sp)
```



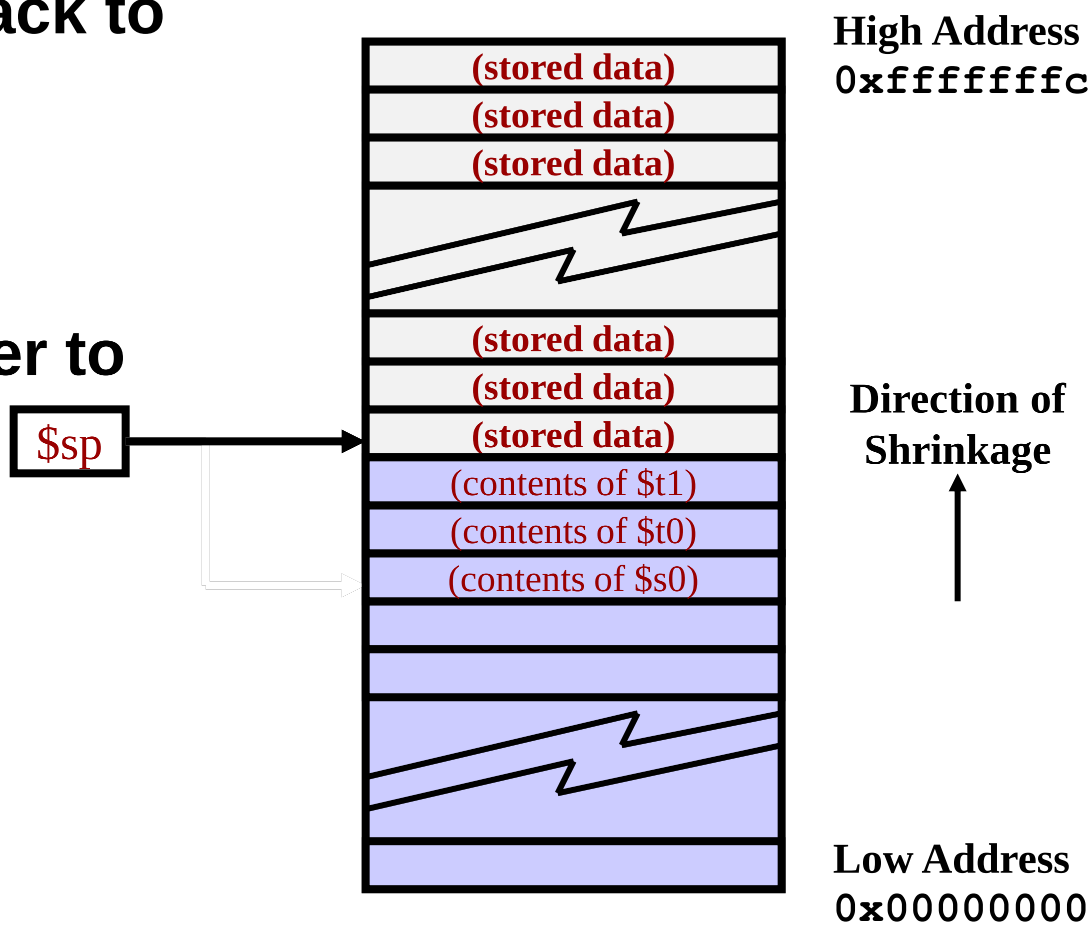
Pop - Removing Data from Stack

- Move data from stack to register if needed

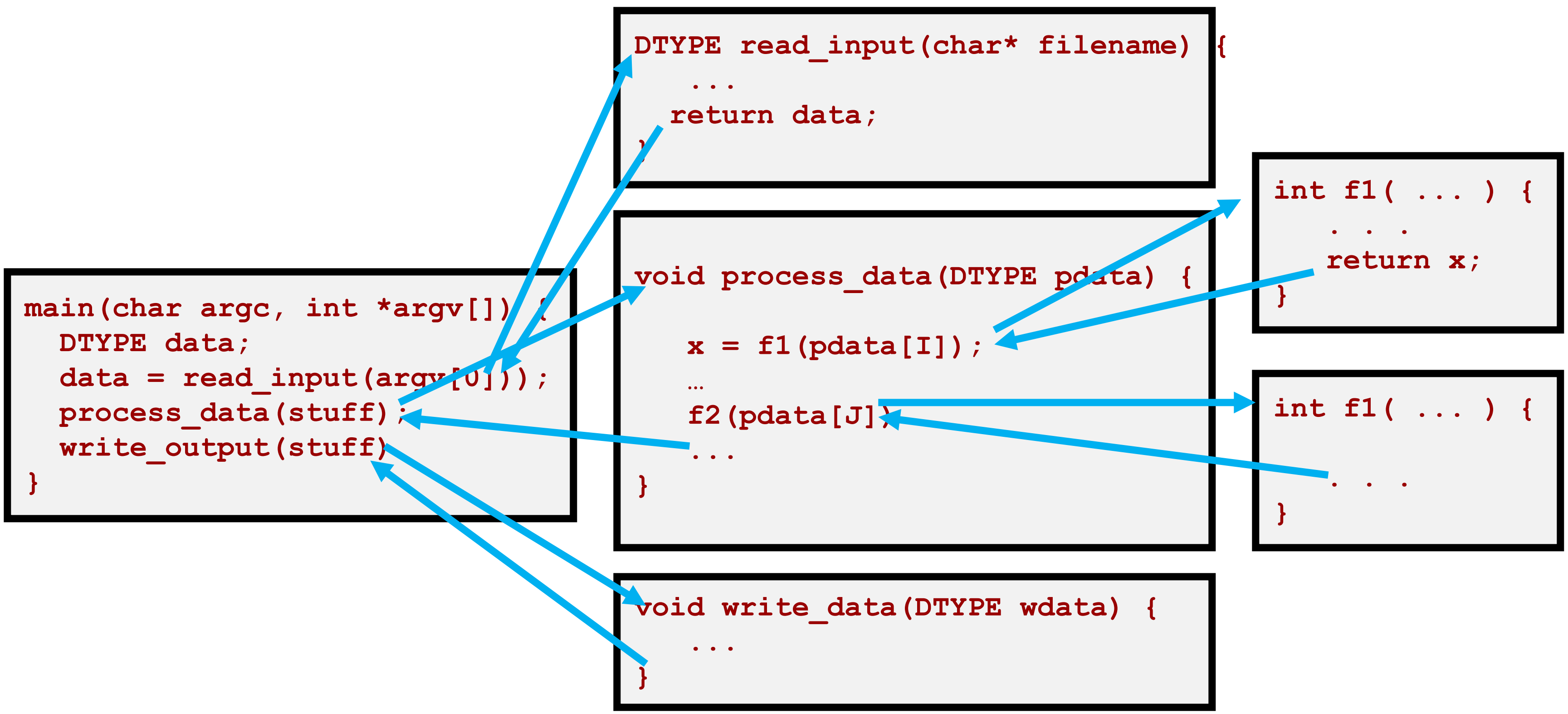
```
lw $s0,0($sp)
lw $t0,4($sp)
lw $t1,8($sp)
```

- Adjust stack pointer to remove space

```
addi $sp,$sp,12
```

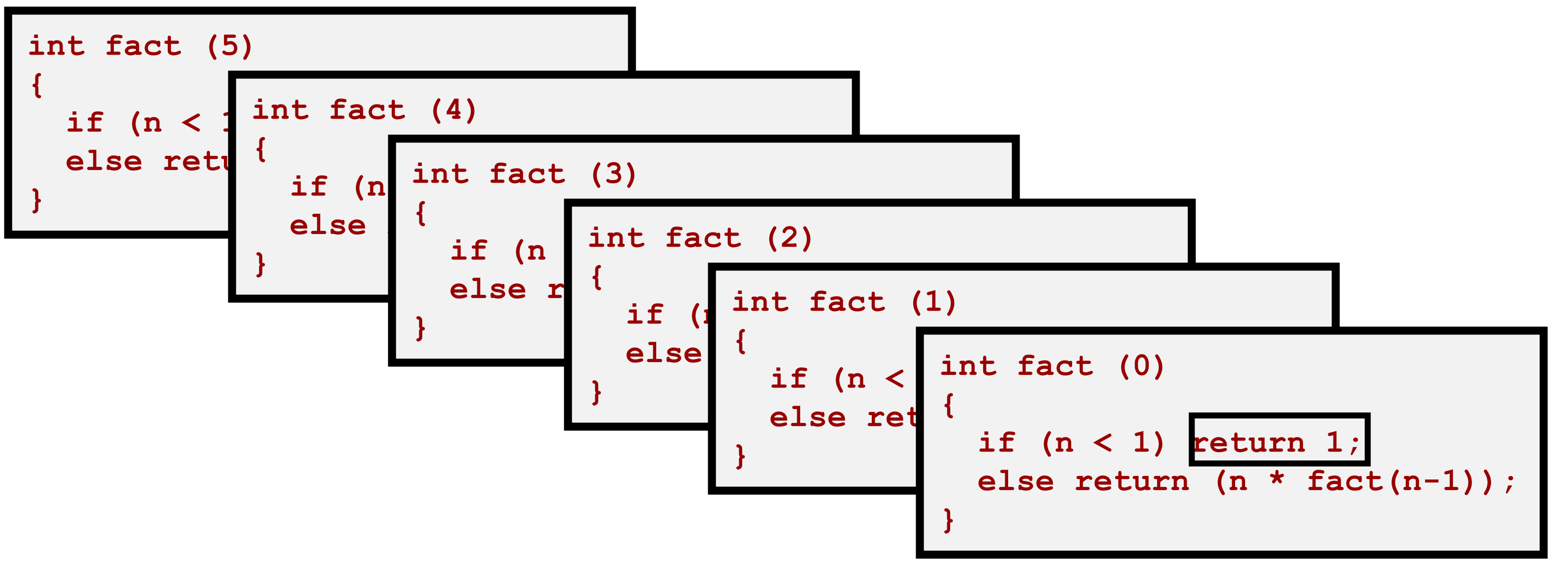


Nested Calls in C Functions



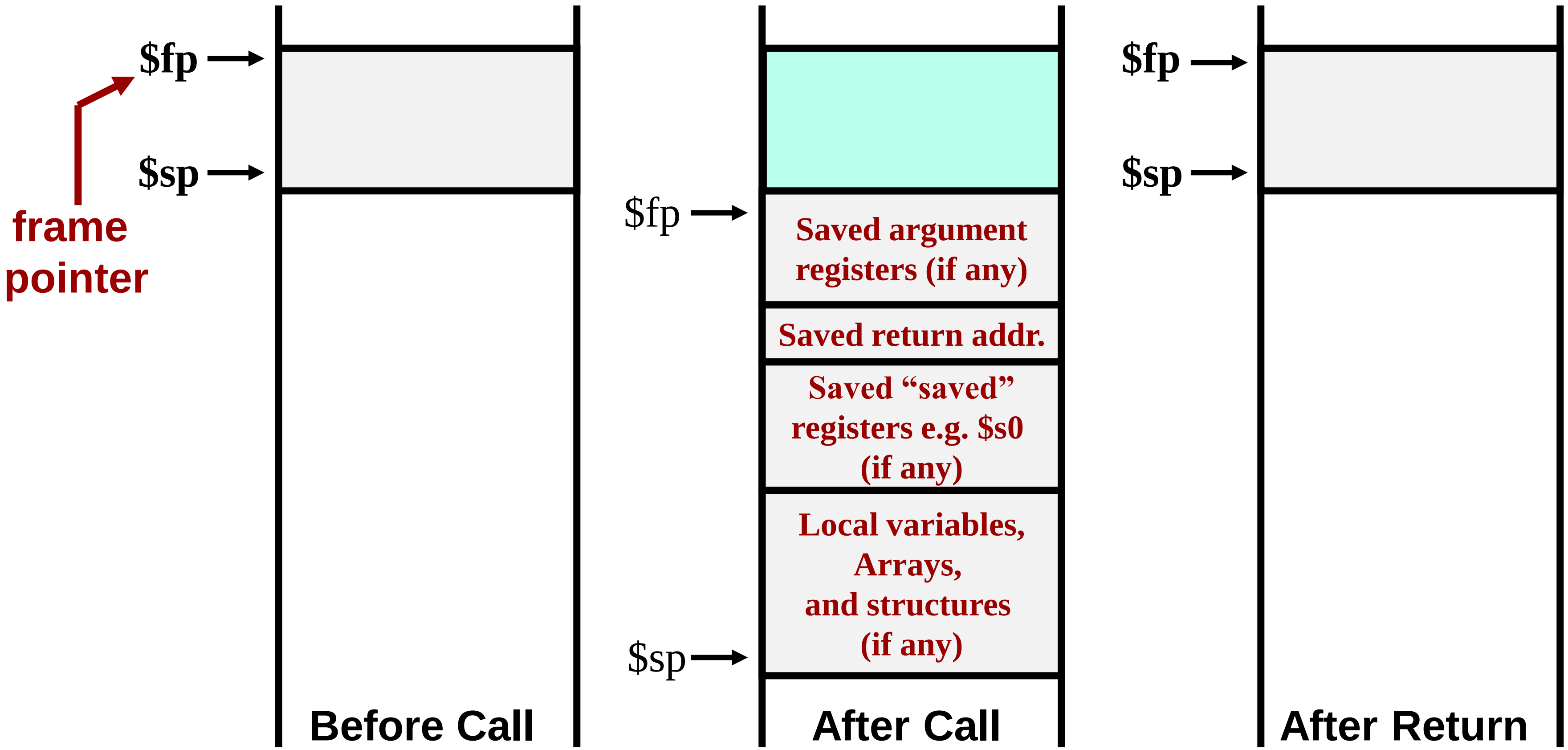
Recursion in C Functions

```
int fact (int n) /* see book p. 117)
{
  if (n < 1) return 1;
  else return (n * fact(n-1));
}
```



Procedures and the Stack

- Each called procedure maintains a **stack frame**

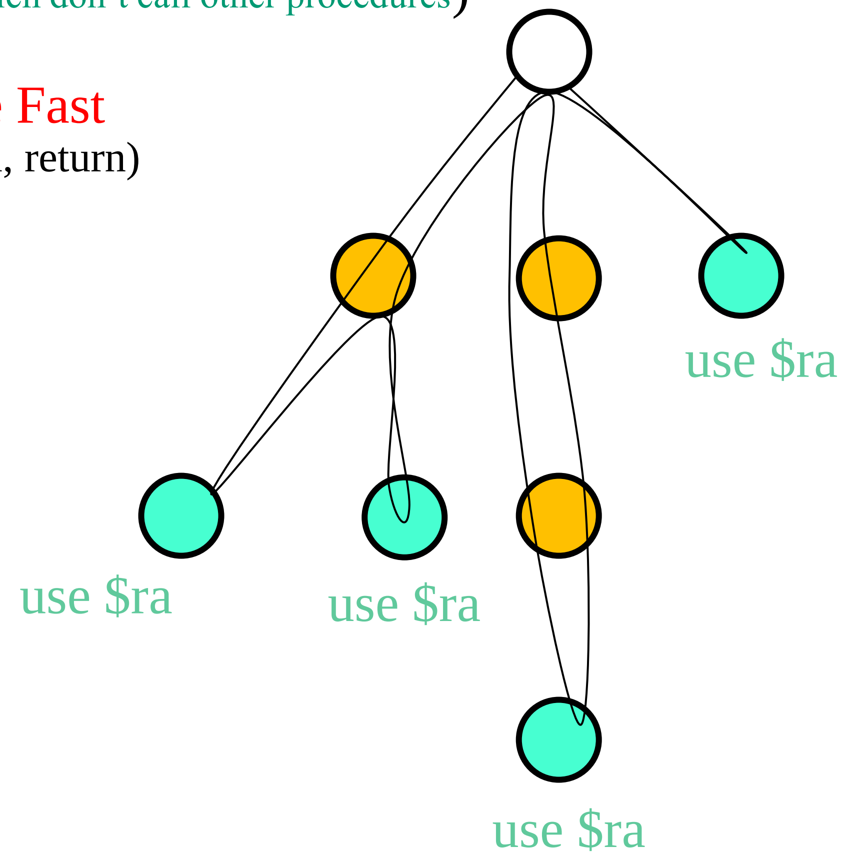


Transfer of Control in Procedures

- **Some architectures such as 68HC11 transfer control on procedures using:**
  - **jsr instruction**
    - **Pushes return address onto the stack**
    - **Changes PC to target address**
  - **rts instruction**
    - **Pops return address off the stack**
    - **Changes PC to popped address**

Transfer of Control in Procedures

- Why doesn't MIPS architecture do this?
  - ✓ Nested calls are not as common as leaf procedures
    (procedures which don't call other procedures)
  - ✓ Principle 4 - Make the Common Case Fast
    (stack overhead avoided for leaf procedure call, return)



Example: Leaf Procedure C Code

```
int leaf_example (int g, h, i, j) //callee
{
   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`

If this procedure wants to use the saved register $s0, it has to save $s0 before using it since the caller may have been using the register $s0. Then, at the end of this procedure, it has to recover $s0 before returning to the caller.

See the example in the next page!

Example: Leaf Procedure MIPS Code

```
#at the caller
jal  leaf_example     # $ra = PC+4,  PC = leaf_example
add $t3, $v0, $zero
```

$ra → add $t3, $v0, $zero

```
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
```

PC → addi $sp, $sp, -4

addi $sp, $sp, -4; sw $s0, 0($sp)	Save $s0 on stack
add $t0, $a0, $a1; add $t1, $a2, $a3; sub $s0, $t0, $t1	Procedure body
add $v0, $s0, $zero	Result in $v0
lw $s0, 0($sp); addi $sp, $sp, 4	Restore $s0
jr $ra	Return

Non-Leaf Procedures

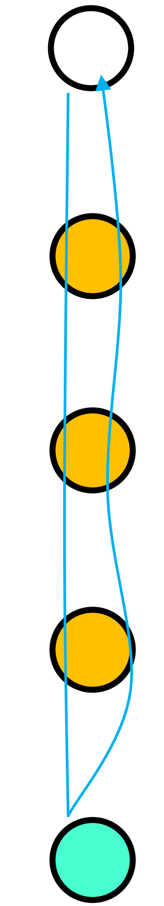
- **Procedures that call other procedures**
- **For nested call, caller needs to save on the stack:**
  - **Its return address**
  - **Any arguments and temporaries needed after the call**
- **Restore from the stack after the call**

Non-Leaf Procedure Example

- C code:

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

main() {
   …
   int r = fact(n); //call procedure
   printf(“fact value = %d\n”, r);
   …
}
```

return here



use $ra

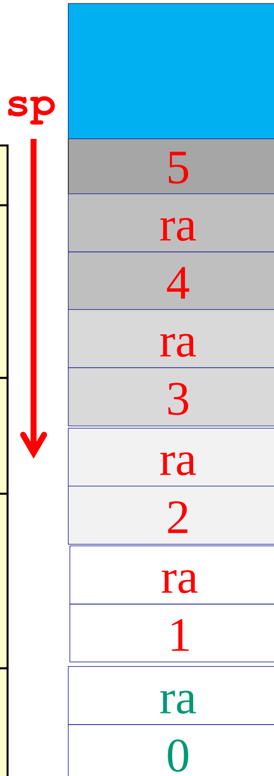
- Argument `n` in `$a0`
- Result in `$v0`

Non-Leaf Procedure Example

fact(5)?

```
addi $a0, $zero, 5
jal fact
xxxx
```

```
fact:
     addi $sp, $sp, -8      # adjust stack for 2 items
     sw   $ra, 4($sp)       # save return address (main)
     sw   $a0, 0($sp)       # save argument n
     slti $t0, $a0, 1       # test for n < 1
     beq  $t0, $zero, L1    # if n >= 1, go to L1
     addi $v0, $zero, 1     # if n < 1, 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)       # return address
     addi $sp, $sp, 8       # pop 2 items from stack
     mul  $v0, $a0, $v0     # multiply to get result
     jr   $ra               # and return
```



Summary

- Steps required to call a Procedure
- Make the common case fast : $ra
  - leaf and non-leaf nodes
- How the system stack is changed when a procedure is called?
- How the non-leaf procedure is implemented?