

Computer Architecture

Instructions: the Language of Computer

The MIPS instruction set, Part 2

Agenda

- Logical operations and shift operations
- Conditional operation and loop statements
- Procedure calling
 - Register usage
 - Local data on the stack
- Memory layout

Logical Operations

- Instruction for bitwise manipulation
 - The previous instructions operate on full words, but it is useful to operate on fields of bits within a word on individual bits, e.g., extracting and inserting groups of bits in a word.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `shamt`: how many positions to shift
- shift left logical (`sll`)
 - shift left and fill with 0 bits
 - `sll` by i bits \Rightarrow multiplies by 2^i
- shift right logical (`srl`)
 - shift right fill with 0 bits
 - `srl` by i bits \Rightarrow divides by 2^i (unsigned only)

```
4  int main(){
5      int a = 4;
6      printf("%d\n", a>>1);
7  }
```

Shift Logical Example

- `sll $t2, $s0, 4` # $\$t2 = (\text{value in } \$s0) * 2^4$

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

0000 0000 0000 0000 0000 0000 0000 1001₂ = 9₁₀

shifting left by 4:

0000 0000 0000 0000 0000 0000 1001 0000₂ = 144₁₀

i.e., $144 = 9 \times 2^4$

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0 \Rightarrow isolating fields where are 1s in both binaries.

and \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged

`or $t0, $t1, $t2`


\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction (it is equivalent to NOT if one operand is zero.)
 - $a \text{ NOR } b == \text{NOT } (a \text{ OR } b)$

```
nor $t0, $t1, $zero
```

Register 0: always read as zero



\$t1

0000 0000 0000 0000 0011 1100 0000 0000

\$t0

1111 1111 1111 1111 1100 0011 1111 1111

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq rs, rt, L1`
 - `if(rs==rt)` branch to instruction labeled `L1`;
- `bne rs, rt, L1`
 - `if(rs!=rt)` branch to instruction labeled `L1`;
- `j L1`
 - unconditional jump to instruction labeled `L1`

Notes:

- Traditionally, `beq` and `bne` are called conditional branches.
- Avoiding unnecessary branching to make the execution more efficient

Compiling If statement

- C code: (assume f, g, \dots in $\$s0, \$s1, \dots$)

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

```
else f = g-h;
```

- Compiled MIPS code:

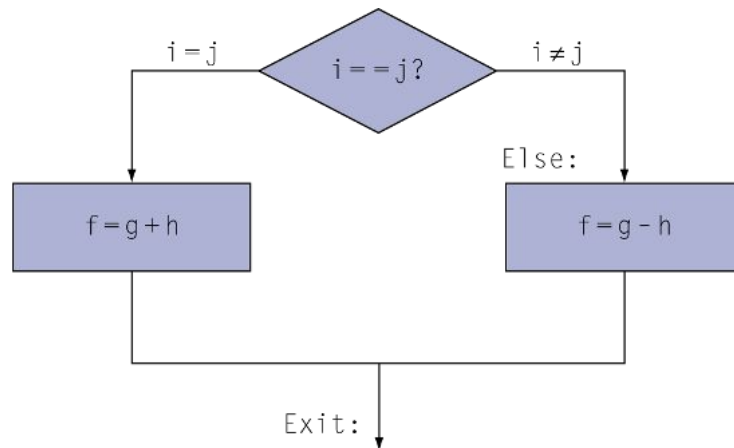
```
bne $s3, $s4, Else
```

```
add $s0, $s1, $s2
```

```
j      Exit
```

```
Else: sub $s0, $s1, $s2
```

```
Exit: ...
```



We assume that $i == j$ is the common case; we use `bne` to reduce the possibility of branching (after all, branching needs to change the address in PC, which takes a little bit more time.)

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

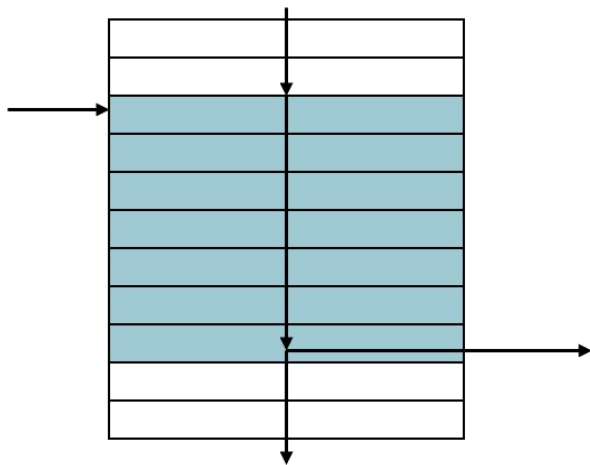
- Compiled MIPS code: (i in \$s3, k in \$s5, address of save in \$s6)

```
Loop: sll    $t1, $s3, 2      # $t1 = reg $s3 << 2 bits, i.e., $s3*4
      add    $t1, $t1, $s6    # $t1 is the address of the i-th element
      lw     $t0, 0($t1)      # load $t0 from the address in $t1
      bne    $t0, $s5, Exit   # if $t0 != $s5 brach to Exit
      addi   $s3, $s3, 1      # increment $s3 by 1
      j      Loop            # jump to Loop

Exit: ...
```

Basic Blocks

- A basic block is a sequence of instructions with
 - No embedded branches (except at end)
 - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

The following code is a basic block:

```
add    $t0, $t1, $t2    # Instruction 1
sub     $t3, $t4, $t5    # Instruction 2
and     $t6, $t7, $t8    # Instruction 3
beq     $t0, $t1, label1 # Branch, ending the block
```

More Conditional Operations

Set on less than: `slt`, `slti`

- Set result to 1 if a condition is true; otherwise, set to 0
- `slt rd, rs, rt`
 - if (`rs < rt`) `rd = 1`; else `rd = 0`;
- `slti rt, rs, constant`
 - if (`rs < constant`) `rt=1`; else `rt = 0`;
- Use in combination with `beq`, `bne`

```
slt $t0, $s1, $s2  # if($s1<$s2)
```

```
bne $t0, $zero, L  # branch to L
```

Branch Instruction Design

- MIPS compilers use `slt`, `slti`, `beq`, `bne`, and the fixed value of 0 (i.e., `$zero`) to create all relative conditions: equal, not equal, less than, less than or equal, greater than or equal.
 - Why not `blt`, `bge`, etc?
 - Hardware for `<`, `>=`, ..., slower than `=`, `!=`; actually, combining with branch involves more work per instruction, requiring a slower clock.
 - `beq` and `bne` are the common cases

Q: How can we implement `if (i >= j) f = g + h`?

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `tui`

- Example

- `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
- `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
- `slt $t0 $s0, $s1 #signed`
 - $-1 < +1 \Rightarrow \$t0=1$
- `sltu $t0, $s0, $s1 #unsigned`
 - $+4,294,967,295 > +1 \Rightarrow \$t0=0$

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: (“figment of the assembler’s imagination”)
 - convenient shorthand notations in assembly language that simplify programming.
 - they are translated into one or more actual MIPS instructions by the assembler.
- Examples:

Pseudoinstructions	Convert to instructions
<code>move \$t0, \$t1</code>	<code>add \$t0, \$zero, \$t1</code>
<code>blt \$t0, \$t1, L</code>	<code>slt \$at, \$t0, \$t1</code> <code>bne \$at, \$zero, L</code>

The assembler temporary

- `$at` (register 1) is used as assembler temporary
- When the assembler encounters a pseudoinstruction that requires more than one machine instruction to implement, it might need to use `$at` to hold intermediate values.
- Example,

```
bgt $t0, $t1, label    #if $t1 > $t0, branch to label
```

```
⇒  slt $at, $t1, $t0    #set $at to 1 if $t1<$t0
```

```
bne $at, $zero, label  #branch to label if $at!=0
```

More Pseudoinstructions

Pseudoinstructions	Equivalent instructions
<code>li \$t0, 10 # load immediate 10</code>	<code>addi \$t0, \$zero, 10 #adds 10 to \$zero and stores the result in \$t0</code>
<code>la \$t0, array #load address</code>	<code>lui \$t0, upper_part ori \$t0, \$t0, lower_part</code>
<code>bge \$t0, \$t1, label #be greater than</code>	<code>slt \$at, \$t0, \$t1 #load upper 16 bits of the address beq \$at, \$zero, label #combine with lower 16 bits</code>

Example: Switch/Case statement

High-Level Code

```
switch (amount) {  
    case 20:    fee = 2; break;  
  
    case 50:    fee = 3; break;  
  
    case 100:   fee = 5; break;  
  
    default:    fee = 0;  
}
```

```
// equivalent function using if/else statements
if (amount == 20) fee = 2;
else if (amount == 50) fee = 3;
else if (amount == 100) fee = 5;
else fee = 0;
```

MIPS Assembly Code

```

# $s0 = amount, $s1 = fee

case20:
    addi $t0, $0, 20          # $t0 = 20
    bne $s0, $t0, case50     # amount == 20? if not,
                                # skip to case50
    addi $s1, $0, 2          # if so, fee = 2
    j     done                # and break out of case

case50:
    addi $t0, $0, 50          # $t0 = 50
    bne $s0, $t0, case100     # amount == 50? if not,
                                # skip to case100
    addi $s1, $0, 3          # if so, fee = 3
    j     done                # and break out of case

case100:
    addi $t0, $0, 100         # $t0 = 100
    bne $s0, $t0, default     # amount == 100? if not,
                                # skip to default
    addi $s1, $0, 5          # if so, fee = 5
    j     done                # and break out of case

default:
    add  $s1, $0, $0           # fee = 0

done:

```

switch/case statements
execute one of several
blocks of code depending
on the conditions.

A case statement is equivalent to a series of nested `if/else` statements.

Procedure/Function Calling

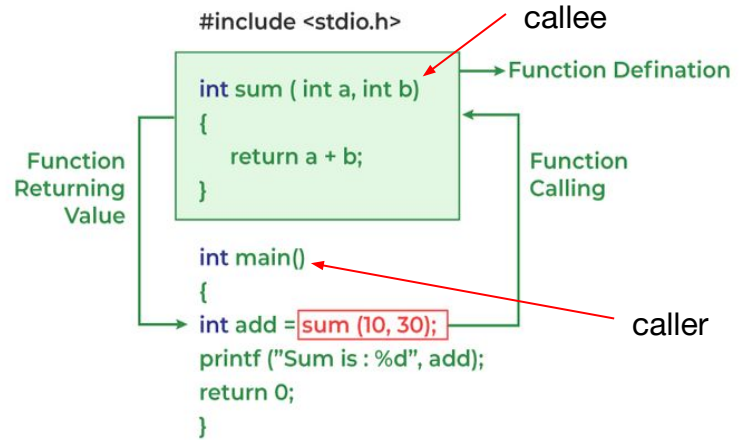
Steps required:

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call

- **Caller:** the calling program, e.g., the `main()`
- **Callee:** the procedure/function being called, e.g., the `sum()`

When it runs the `main()`, the arguments 10, and 30 will be placed into registers, and then, runs the statements in `sum()`. The result of `sum()` will be placed in a register for the `main()` to get it. Finally, it will go back to the place where the `sum()` is called.

Working of Function in C



Recall the registers in MIPS

- MIPS has 32 registers and each can store exactly one word.
- Each register has its own status or function; they are critical for function calls.

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

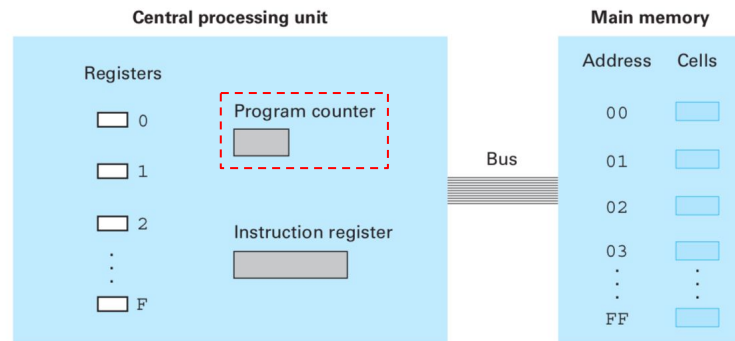
- puts address of the **following** instruction in `$ra`, and,
- jumps to target address

- Procedure return: jump register

```
jr $ra
```

- Copies `$ra` to *program counter*

- `jal` links one procedure to another, and
`jr` traces back to the start procedure.

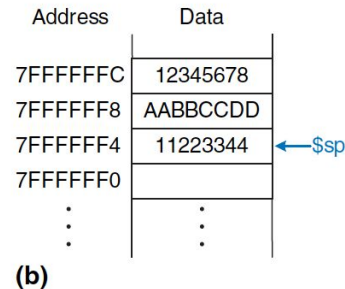
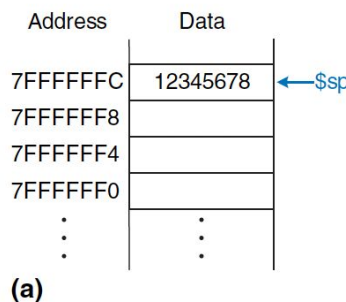


Program counter (PC): a register to hold the address of the current instruction being executed, i.e., the processor is loading and executing the instruction from the address it pointed to.

- The `jal` instruction actually saves $PC+4$ in register `$ra` to link the following instruction to set up the procedure return.

Using More Registers

- MPIS has four argument registers (\$a0-\$a3) and two return value registers (\$v0, \$v1);
- But a procedure may need more registers than those ones. \Rightarrow We need to spill registers to memory.
 - e.g., in recursion, the arguments of the current self-call should be preserved before it goes into the next self-call.
- We use a stack to spill registers



Using the stack

- We use a **stack**, a last-in-first-out queue, for spilling registers.
 - `$sp` (Register 29): the stack pointer, MIPS software reverses it for the stack pointer.
 - It is adjusted by one word (4 bytes) for each saved or restored register.
 - **Push**: Placing data into the stack; **Pop**: removing data from the stack
- In practice, saving and restoring are necessary for some values but not all values in registers
 - e.g., the variable `temp` in the function `swap(a, b)`.
- To avoid saving and restoring a register whose value is never used, MIPS software separates 18 of the registers into two groups:
 - `$t0-$t9`: temporary registers that are not preserved by the callee on a procedure call
 - `$s0-$s7`: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

Leaf Procedure Example

- Leaf procedure: procedures that do not call others.

C code:

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

We assume:

- Arguments g, \dots, j in $\$a0, \dots, \$a3$
- f in $\$s0$ (hence, need to save $\$s0$ on stack before we store f in it, then restore it)
- Result in $\$v0$

Leaf Procedure Example

- MIPS code:

```
leaf_example:
    addi $sp, $sp, -4  ← push
stacksw    $s0, 0($sp)    #save $s0 on
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3    #procedure body
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero  #result
    lw   $s0, 0($sp)      #restore $s0
    addi $sp, $sp, 4      ← pop
    jr   $ra              #return
```

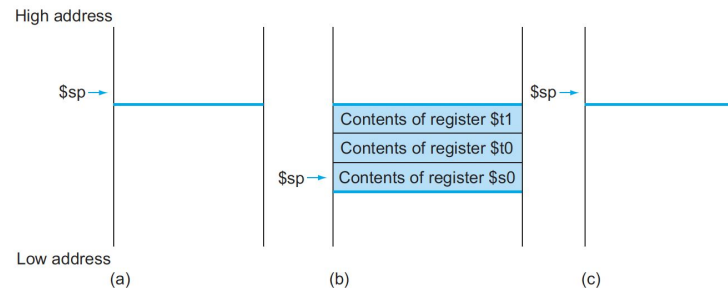


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the “top” of the stack, or the last word in the stack in this drawing.

MIPS set stacks “grow” from higher addresses to lower address.

- Push values into stack ⇒ Subtracting from the stack pointer
- Pop values out of stack ⇒ Adding to the stack pointer

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save the following on the stack because they might be overwritten by the callee:
 - Its return address
 - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

C code: (factorial of n , $n! = n \times (n-1) \times \cdots \times 1$)

```
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

- 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
```

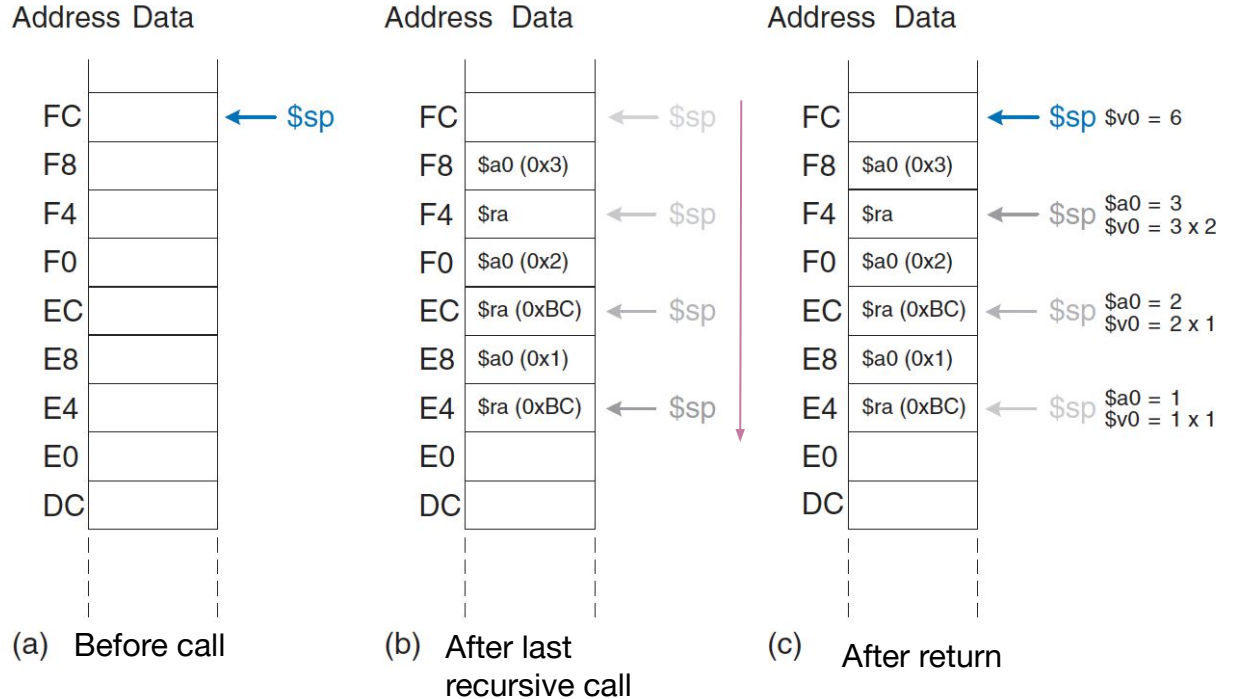
- push two words on stack
- save the current address and argument on the stack

- Check the stop condition; if **not** satisfied $\Rightarrow \$t0=0$;
- if $\$t0=0$ goto L1; otherwise, goto the next instruction

- add 1 to $\$v0$
- add 8 to $\$sp \Rightarrow$ pop 2 items
- Jump to $\$ra \Rightarrow$ finish instructions in the callee and back to the caller

- add -1 to $\$a0$
- Jump and link to fact; so, $\$ra=PC+4$
The `jr` in fact will jump to the following instruction `lw $a0, 0($sp)` which is store in the memory next to the `jal fact`.

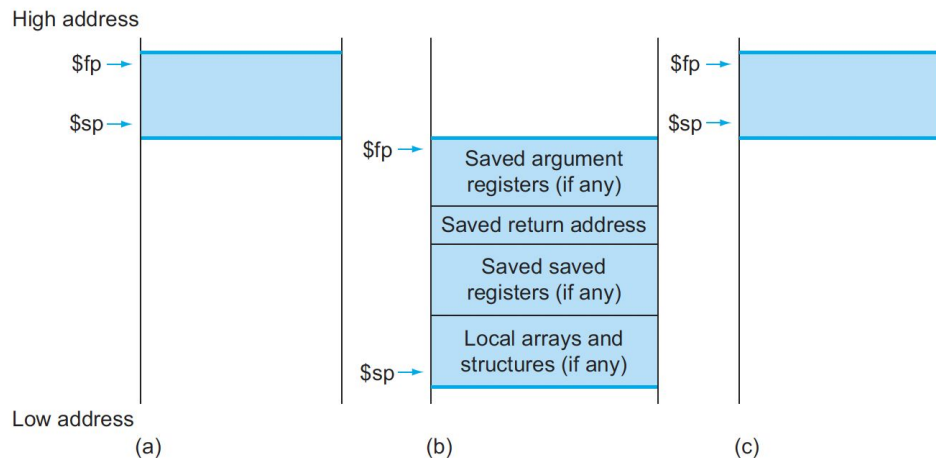
Stack during the factorial function call when n=3



Local Data on the Stack

- The 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 **activate record**.
- Some MPIS compilers use a frame pointer ($\$fp$) to point to the **first** word of the frame of a procedure.
- $\$fp$ offers a stable base register within a procedure for local memory-references.

Local Data on the Stack

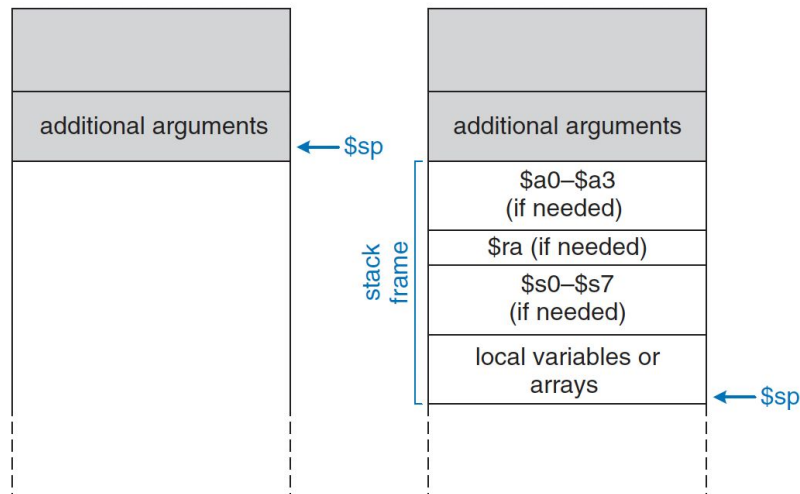


- The frame pointer (\$fp) points to the first word of the frame, often a saved argument register, and the stack pointer (\$sp) points to the top of the stack.
- The stack is adjusted to make room for all the saved registers and any memory-resident local variables.
- Since the stack pointer may change during the program execution, it's easier for programmers to reference variables via the stable frame pointer.

Stack allocation (a) before, (b) during, and (c) after the procedure call.

Additional arguments

By MIPS convention, if a function has more than four arguments, the first four arguments are passed in the argument registers (\$a0–\$a3), and the additional arguments are passed on the stack, just above \$sp.



In such cases, the **caller** must expand its stack to make room for the additional arguments.

To access additional input arguments, the callee can access stack data not in its stack frame.

Example: handling additional arguments

C code:

```
void foo(int arg1, int arg2, int arg3, int arg4, int  
arg5, int arg6);
```

We assume

- The first four arguments are passed in `$a0`, `$a1`, `$a2`, `$a3`;
- The caller pushes `arg 5` and `arg 6` onto the stack before calling `foo`.

MIPS Assembly code

```
1  # Assume the arguments are stored in registers and memory
2  # Register or memory where arguments are located:
3  # $t0 = arg1, $t1 = arg2, $t2 = arg3, $t3 = arg4, $t4 = arg5, $t5 = arg6
4
5  # First four arguments in registers
6  move $a0, $t0      # arg1
7  move $a1, $t1      # arg2
8  move $a2, $t2      # arg3
9  move $a3, $t3      # arg4
10
11 # Push additional arguments onto the stack
12 subu $sp, $sp, 8    # Make space on the stack for 2 additional arguments (arg5 and arg6)
13 sw $t4, 4($sp)      # Store arg5 on the stack
14 sw $t5, 0($sp)      # Store arg6 on the stack
15
16 # Call the function foo
17 jal foo
18
19 # After the function returns, you might need to restore the stack pointer
20 addu $sp, $sp, 8    # Restore the stack pointer
```

In the call of `foo`, `arg5` and `arg6` can be accessed from the stack using offsets relative to `$sp`. (e.g., `arg5` would be at `4($sp)` and `arg6` would be at `8($sp)` .

In such cases, the `$fp` can be useful as it provides a stable reference.

After the call, the caller should restore the stack pointer.