MIPS ASSEMBLY PROGRAMMING LANGUAGE PART VI

Ayman Hajja, PhD

CALLING A FUNCTION

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

MIPS FUNCTION CALL CONVENTIONS

1. Registers faster than memory, so use them
2. $a0 to $a3; four argument registers to pass parameters
3. $v0–$v1; two value registers to return values
4. $ra; one return address register to return to the point of origin

OPTIMIZED FUNCTION CONVENTIONS

- To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

1. Preserved across function calls:
   - Caller can rely on values being unchanged
   - $ra, $sp, $gp, $fp, and saved registers ($s0 to $s7)
2. Not preserved across function calls:
   - Caller cannot rely on values being unchanged
   - Return values registers $v0, $v1, argument registers ($a0 to $a3), and temporary registers ($t0 to $t7)

USING PRESERVED REGISTERS IN A FUNCTION

- If we'd like to use any of the preserved registers in a function that's being called, we need to:
    1. Push the current value (of register) to the stack
    2. Use the register
    3. Pop the value from the stack and back to the register before leaving the function
- Example; inside the callee:
    - int s0 = arg0 + arg1;
    - int s1 = arg2 + arg3;
    - return s0 - s1;

USING PRESERVED REGISTERS IN A FUNCTION

Example;

```
void main()
{
        calc(4, 6, 2, 3);
}
int calc(int val1, int val2, int val3, int val4)
{
        int s0 = val1 + val2;
        int s1 = val3 + val4;
        return s0 - s1;
}
```

```
.text   # Instruction section
    MAIN:


    CALC:
```

USING PRESERVED REGISTERS IN A FUNCTION

Example;

```
void main()
{
        calc(4, 6, 2, 3);
}
int calc(int val1, int val2, int val3, int val4)
{
        int s0 = val1 + val2;
        int s1 = val3 + val4;
        return s0 - s1;
}
```

```
.text  # Instruction section
    MAIN:
    addi $a0, $zero, 4      # Fill arguments
    addi $a1, $zero, 6      # Fill arguments
    addi $a2, $zero, 2      # Fill arguments
    addi $a3, $zero, 3      # Fill arguments
```

CALC:

USING PRESERVED REGISTERS IN A FUNCTION

Example;

```
void main()
{
        calc(4, 6, 2, 3);
}
int calc(int val1, int val2, int val3, int val4)
{
        int s0 = val1 + val2;
        int s1 = val3 + val4;
        return s0 - s1;
}
```

```
.text  # Instruction section
    MAIN:
    addi $a0, $zero, 4      # Fill arguments
    addi $a1, $zero, 6      # Fill arguments
    addi $a2, $zero, 2      # Fill arguments
    addi $a3, $zero, 3      # Fill arguments
    jal CALC                # Jump to function

CALC:
```

BEFORE WE CAN USE $S0 & $S1, WE NEED TO STORE PREVIOUS VALUES IN STACK. RECALL THAT $S0 & $S1 ARE PRESERVED REGISTERS

Example;

```
void main()
{
        calc(4, 6, 2, 3);
}
int calc(int val1, int val2, int val3, int val4)
{
        int s0 = val1 + val2;
        int s1 = val3 + val4;
        return s0 - s1;
}
```

```
.text  # Instruction section
    MAIN:
    addi $a0, $zero, 4      # Fill arguments
    addi $a1, $zero, 6      # Fill arguments
    addi $a2, $zero, 2      # Fill arguments
    addi $a3, $zero, 3      # Fill arguments
    jal CALC                # Jump to function

    CALC:
```

USING PRESERVED REGISTERS IN A FUNCTION

Example;

```
void main()
{
        calc(4, 6, 2, 3);
}
int calc(int val1, int val2, int val3, int val4)
{
        int s0 = val1 + val2;
        int s1 = val3 + val4;
        return s0 - s1;
}
```

```
.text  # Instruction section
    MAIN:
    addi $a0, $zero, 4      # Fill arguments
    addi $a1, $zero, 6      # Fill arguments
    addi $a2, $zero, 2      # Fill arguments
    addi $a3, $zero, 3      # Fill arguments
    jal CALC                # Jump to function
```

```
CALC:
addi $sp, $sp, -8       # Allocate 8 bytes
sw $s0, 0($sp)          # Push $s0
sw $s1, 4($sp)          # Push $s1
```

USING PRESERVED REGISTERS IN A FUNCTION

Example;

```
void main()
{
        calc(4, 6, 2, 3);
}
int calc(int val1, int val2, int val3, int val4)
{
        int s0 = val1 + val2;
        int s1 = val3 + val4;
        return s0 - s1;
}
```

```
.text  # Instruction section
    MAIN:
    addi $a0, $zero, 4      # Fill arguments
    addi $a1, $zero, 6      # Fill arguments
    addi $a2, $zero, 2      # Fill arguments
    addi $a3, $zero, 3      # Fill arguments
    jal CALC                # Jump to function

CALC:
addi $sp, $sp, -8       # Allocate 8 bytes
sw $s0, 0($sp)          # Push $s0
sw $s1, 4($sp)          # Push $s1
add $s0, $a0, $a1       # Perform function tasks
add $s1, $a2, $a3       # Perform function tasks
sub $v0, $s0, $s1       # Put return value in $v0
```

NOW THAT WE'RE DONE USING $S0 & $S1, WE CAN LOAD THEIR OLD VALUES FROM THE STACK BEFORE QUITTING THE FUNCTION

Example;

```
void main()
{
        calc(4, 6, 2, 3);
}
int calc(int val1, int val2, int val3, int val4)
{
        int s0 = val1 + val2;
        int s1 = val3 + val4;
        return s0 - s1;
}
```

```
.text  # Instruction section
    MAIN:
    addi $a0, $zero, 4      # Fill arguments
    addi $a1, $zero, 6      # Fill arguments
    addi $a2, $zero, 2      # Fill arguments
    addi $a3, $zero, 3      # Fill arguments
    jal CALC                # Jump to function
```

```
CALC:
addi $sp, $sp, -8       # Allocate 8 bytes
sw $s0, 0($sp)          # Push $s0
sw $s1, 4($sp)          # Push $s1
add $s0, $a0, $a1       # Perform function tasks
add $s1, $a2, $a3       # Perform function tasks
sub $v0, $s0, $s1       # Put return value in $v0
```

USING PRESERVED REGISTERS IN A FUNCTION

Example;

```
void main()
{
        calc(4, 6, 2, 3);
}
int calc(int val1, int val2, int val3, int val4)
{
        int s0 = val1 + val2;
        int s1 = val3 + val4;
        return s0 - s1;
}
```

```
.text  # Instruction section
    MAIN:
    addi $a0, $zero, 4      # Fill arguments
    addi $a1, $zero, 6      # Fill arguments
    addi $a2, $zero, 2      # Fill arguments
    addi $a3, $zero, 3      # Fill arguments
    jal CALC                # Jump to function

CALC:
addi $sp, $sp, -8       # Allocate 8 bytes
sw $s0, 0($sp)          # Push $s0
sw $s1, 4($sp)          # Push $s1
add $s0, $a0, $a1       # Perform function tasks
add $s1, $a2, $a3       # Perform function tasks
sub $v0, $s0, $s1       # Put return value in $v0
lw $s0, 0($sp)          # Pop $s0
lw $s1, 4($sp)          # Pop $s1
addi $sp, $sp, 8        # Free stack
jr $ra                  # Go back to 'main'
```

USING PRESERVED REGISTERS IN A FUNCTION

Example;

```
void main()
{
        calc(4, 6, 2, 3);
}
int calc(int val1, int val2, int val3, int val4)
{
        int s0 = val1 + val2;
        int s1 = val3 + val4;
        return s0 - s1;
}
```

```
.text  # Instruction section
    MAIN:
    addi $a0, $zero, 4      # Fill arguments
    addi $a1, $zero, 6      # Fill arguments
    addi $a2, $zero, 2      # Fill arguments
    addi $a3, $zero, 3      # Fill arguments
    jal CALC                # Jump to function
    addi $v0, $zero, 10     # End program
    syscall                 # End program
    CALC:
    addi $sp, $sp, -8       # Allocate 8 bytes
    sw $s0, 0($sp)          # Push $s0
    sw $s1, 4($sp)          # Push $s1
    add $s0, $a0, $a1       # Perform function tasks
    add $s1, $a2, $a3       # Perform function tasks
    sub $v0, $s0, $s1       # Put return value in $v0
    lw $s0, 0($sp)          # Pop $s0
    lw $s1, 4($sp)          # Pop $s1
    addi $sp, $sp, 8        # Free stack
    jr $ra                  # Go back to 'main'
```

INSTRUCTION SUPPORT FOR FUNCTIONS

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

Parameter variables g, h, i, and j in argument registers $a0, $a1, $a2, and $a3

Say we can use one temporary register $t0

MIPS CODE FOR LEAF_EXAMPLE

```
addi $sp, $sp, -4        # adjust stack for 1 item ($s0)
sw $s0, 0($sp)           # save $s0 for use afterwards
```

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

MIPS CODE FOR LEAF_EXAMPLE

```
addi $sp, $sp, -4       # adjust stack for 1 item ($s0)
sw $s0, 0($sp)          # save $s0 for use afterwards

add $s0, $a0, $a1       # f = g + h
add $t0, $a2, $a3       # t0 = i + j
sub $v0, $s0, $t0       # return value (g + h) – (i + j)
```

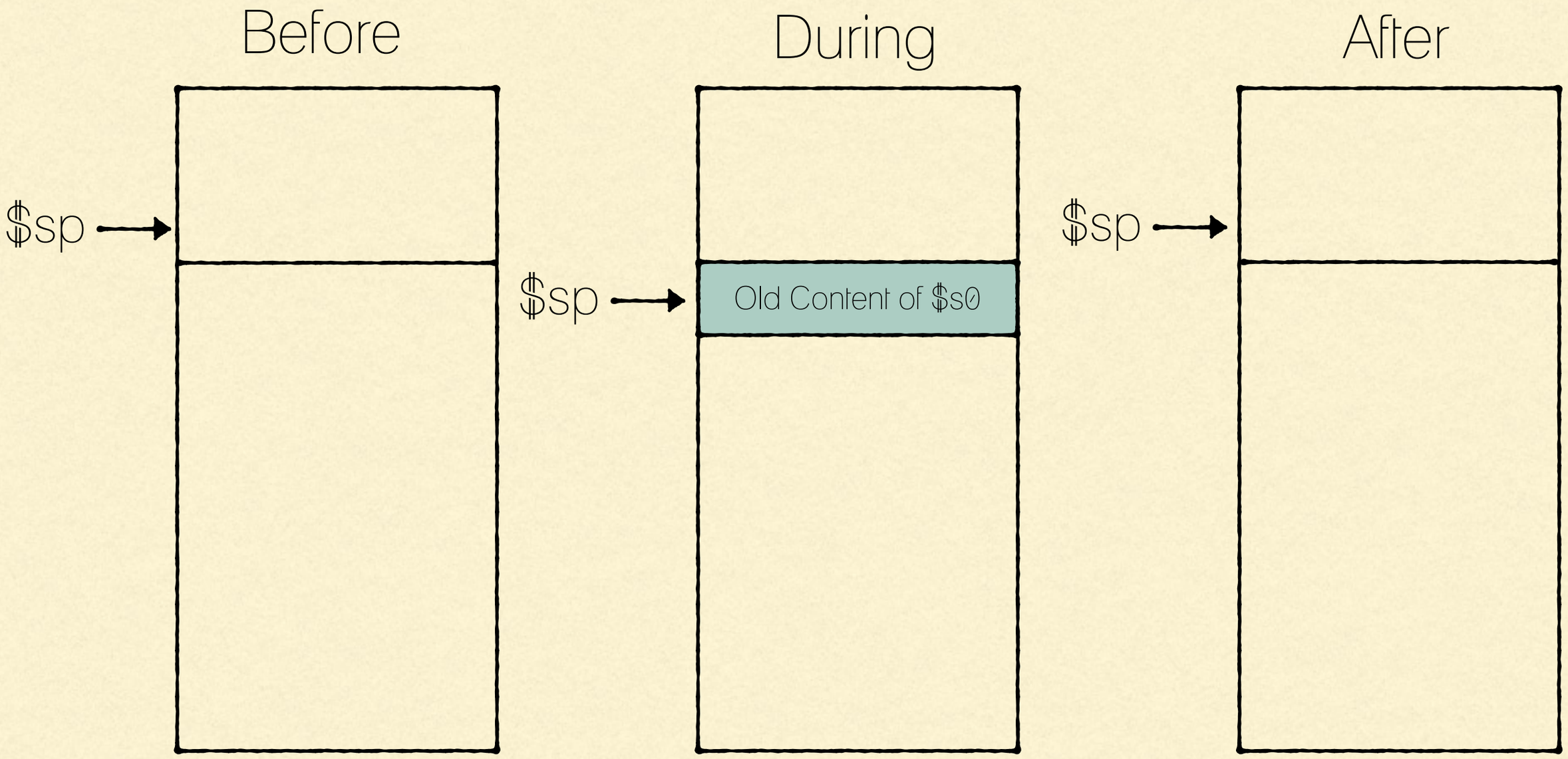
```
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) – (i + j);
    return f;
}
```

MIPS CODE FOR LEAF_EXAMPLE

```
addi $sp, $sp, -4       # adjust stack for 1 item ($s0)
sw $s0, 0($sp)          # save $s0 for use afterwards

add $s0, $a0, $a1       # f = g + h
add $t0, $a2, $a3       # t0 = i + j
sub $v0, $s0, $t0       # return value (g + h) – (i + j)

lw $s0, 0($sp)          # restore register $s0 for caller
addi $sp, $sp, 4        # adjust stack

jr $ra                  # jump back to calling routine
```

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

STACK BEFORE, DURING, AFTER FUNCTION



WHAT IF A FUNCTION CALLS ANOTHER FUNCTION?

- What if you're calling a function that will need to call a second function. What do you need to do to $a0 to $a3 and $ra?

WHAT IF A FUNCTION CALLS ANOTHER FUNCTION?

- What if you're calling a function that will need to call a second function. What do you need to do to $a0 to $a3 and $ra?

We need to push $ra to the stack; and we need to push any argument ($a0 to $a4) that we're using after the function call, since the callee is allowed to (and may) change these arguments.

NESTED PROCEDURES

```
int firstFunction(int x, int y)
{
    int result = secondFunction(x, x) + x + y;
    return result;
}
```

main() is calling:
firstFunction(4, 3);

- Something called 'firstFunction', now 'firstFunction' is calling 'secondFunction'
- So there's a value in $ra that 'firstFunction' wants to jump back to, but this will be overwritten by the call to 'secondFunction'

What can we do?

NESTED PROCEDURES

```
int firstFunction(int x, int y)
{
    int result = secondFunction(x, x) + x + y;
    return result;
}
```

main() is calling:
firstFunction(4, 3);

- Something called 'firstFunction', now 'firstFunction' is calling 'secondFunction'
- So there's a value in $ra that 'firstFunction' wants to jump back to, but this will be overwritten by the call to 'secondFunction'

We need to save 'firstFunction' return address before we call 'secondFunction'

```
int firstFunction(int x, int y)
{
        int result = secondFunction(x, x) + x + y;
        return result;
}
```

main() is calling:
firstFunction(4, 3);

```
            main:
1000        …
…           …                   # Other code in main()
2000        jal firstFun        # We're still in main, now we call firstFunction(4, 3)
2004        …                   # do something with $v0
…           …
_____________________________________
            firstFun:           # What should the value of $a0, $a1, and $ra be?
4000        …                   # Some code in firstFunction
…           …                   # Some code in firstFunction
4040        jal secondFun       # Now we call secondFunction(4, 4)
4044        …
…           jr ra
_____________________________________
            secondFun:          # What should the value of $a0, $a1, and $ra be?
8000        …
…           jr ra
_____________________________________
```

```
int firstFunction(int x, int y)
{
        int result = secondFunction(x, x) + x + y;
        return result;
}
```

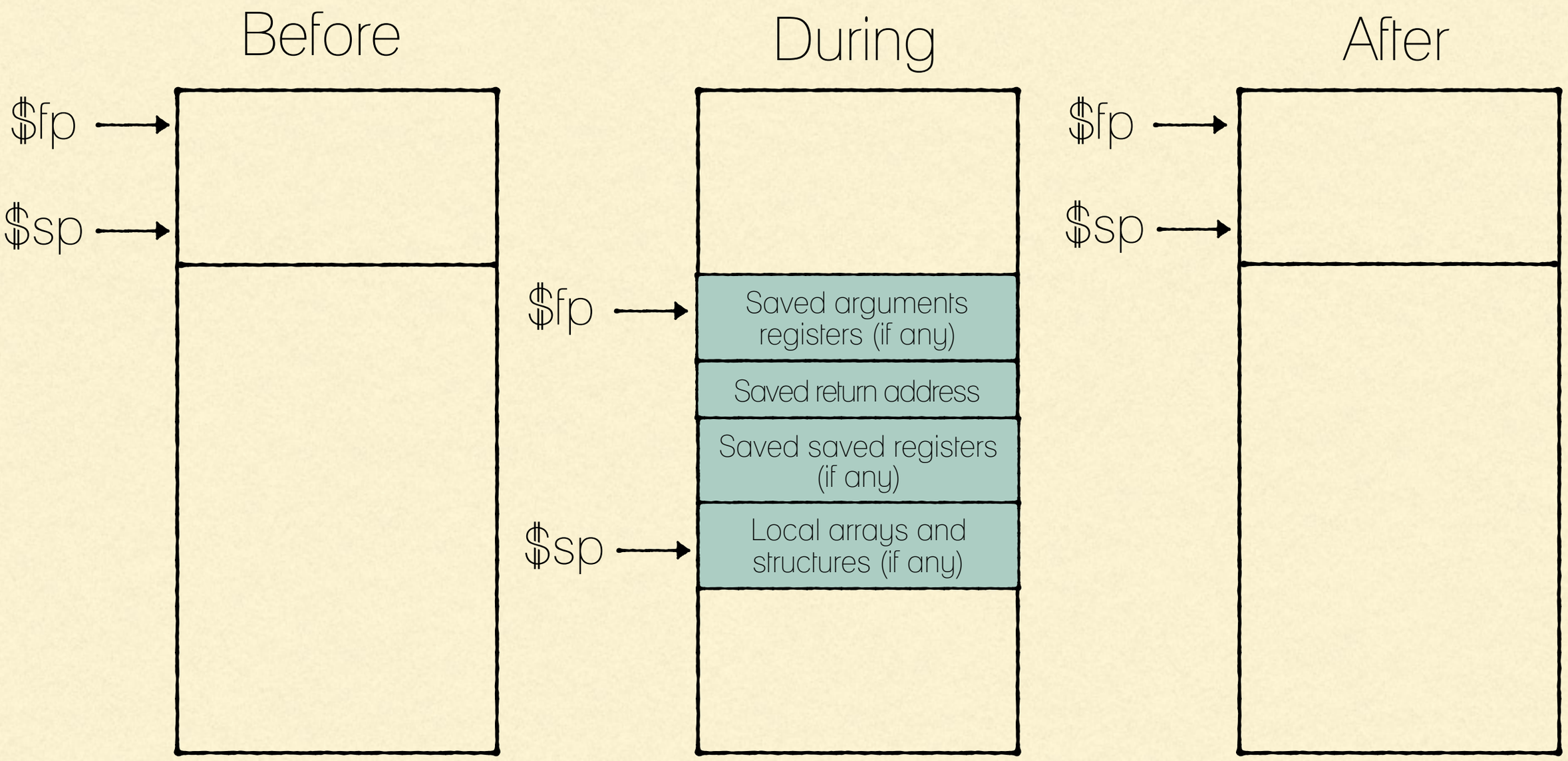
main() is calling:
firstFunction(4, 3);

```
            main:
1000        …
…           …                   # Other code in main()
2000        jal firstFun        # We're still in main, now we call firstFunction(4, 3)
2004        …                   # do something with $v0
…           …
```

```
            firstFun:           # What should the value of $a0, $a1, and $ra be? 4, 3, & 2004
4000        …                   # Some code in firstFunction
…           …                   # Some code in firstFunction
4040        jal secondFun       # Now we call secondFunction(4, 4)
4044        …
…           jr ra
```

```
            secondFun:          # What should the value of $a0, $a1, and $ra be? 4, 4, & 4044
8000        …
…           jr ra
```

STACK BEFORE, DURING, AFTER FUNCTION



NESTED PROCEDURES

```
int firstFunction(int x, int y)
{ int result = secondFunction(x, x) + x + y; return result; }
  FirstFunction:
```

```
        jal SecondFunction      # Call SecondFunction(x, x)
```

```
SecondFunction:
        . . .
```

NESTED PROCEDURES

```
int firstFunction(int x, int y)
{ int result = secondFunction(x, x) + x + y; return result; }
  FirstFunction:
```

```
        add $a1, $a0, $zero     # a1 = x (second argument is now x)
        jal SecondFunction      # Call SecondFunction(x, x)
```

```
SecondFunction:
        . . .
```

NESTED PROCEDURES

```
int firstFunction(int x, int y)
{ int result = secondFunction(x, x) + x + y; return result; }
  FirstFunction:
        addi $sp, $sp, -12      # Create space on stack
        sw $ra, 8($sp)          # Push return address to the stack
        sw $a0, 4($sp)          # Push a0 (x) to the stack since we need it and the callee may change it
        sw $a1, 0($sp)          # Push a1 (y) since we're changing it (also, callee is allowed to change it)
        add $a1, $a0, $zero     # a1 = x (second argument is now x)
        jal SecondFunction      # Call SecondFunction(x, x)
```

```
SecondFunction:
        ...
```

NESTED PROCEDURES

```
int firstFunction(int x, int y)
{ int result = secondFunction(x, x) + x + y; return result; }
  FirstFunction:
        addi $sp, $sp, -12      # Create space on stack
        sw $ra, 8($sp)          # Push return address to the stack
        sw $a0, 4($sp)          # Push a0 (x) to the stack since we need it and the callee may change it
        sw $a1, 0($sp)          # Push a1 (y) since we're changing it (also, callee is allowed to change it)
        add $a1, $a0, $zero     # a1 = x (second argument is now x)
        jal SecondFunction      # Call SecondFunction(x, x)
        lw $a0, 4($sp)          # Bring the value of a0 (x) from the stack
        add $v0, $v0, $a0       # Value returned from secondFunction(x, x) + x (or a0)
```

```
SecondFunction:
        ...
```

NESTED PROCEDURES

```
int firstFunction(int x, int y)
{ int result = secondFunction(x, x) + x + y; return result; }
  FirstFunction:
        addi $sp, $sp, -12      # Create space on stack
        sw $ra, 8($sp)          # Push return address to the stack
        sw $a0, 4($sp)          # Push a0 (x) to the stack since we need it and the callee may change it
        sw $a1, 0($sp)          # Push a1 (y) since we're changing it (also, callee is allowed to change it)
        add $a1, $a0, $zero     # a1 = x (second argument is now x)
        jal SecondFunction      # Call SecondFunction(x, x)
        lw $a0, 4($sp)          # Bring the value of a0 (x) from the stack
        add $v0, $v0, $a0       # Value returned from secondFunction(x, x) + x (or a0)
        lw $a1, 0($sp)          # Bring the value of a1 (y) from the stack
        add $v0, $v0, $a1       # Value returned from secondFunction(x, x) + x + y
```

```
SecondFunction:
        ...
```

NESTED PROCEDURES

```
int firstFunction(int x, int y)
{ int result = secondFunction(x, x) + x + y; return result; }
  FirstFunction:
        addi $sp, $sp, -12      # Create space on stack
        sw $ra, 8($sp)          # Push return address to the stack
        sw $a0, 4($sp)          # Push a0 (x) to the stack since we need it and the callee may change it
        sw $a1, 0($sp)          # Push a1 (y) since we're changing it (also, callee is allowed to change it)
        add $a1, $a0, $zero     # a1 = x (second argument is now x)
        jal SecondFunction      # Call SecondFunction(x, x)
        lw $a0, 4($sp)          # Bring the value of a0 (x) from the stack
        add $v0, $v0, $a0       # Value returned from secondFunction(x, x) + x (or a0)
        lw $a1, 0($sp)          # Bring the value of a1 (y) from the stack
        add $v0, $v0, $a1       # Value returned from secondFunction(x, x) + x + y
        lw $ra, 8($sp)          # Bring the value of 'return address' from the stack
        addi $sp, $sp, 12       # Restore space on stack
        jr $ra
SecondFunction:
        ...
```

REGISTERS DESCRIPTION

Number	Name	Description
0	$zero	Hardwired to 0
1	$at	Reserved for pseudo-instructions
2-3	$v0 - $v1	Return values from functions
4-7	$a0 - $a3	Arguments to functions — Not preserved
8-15	$t0 - $t7	Temporary data — Not preserved
12-23	$s0 - $s7	Saved registers — Preserved
24-25	$t8 - $t9	More temporary registers — Not preserved
26-27	$k0 - $k1	Reserved for kernel — Do not use
28	$gp	Global area pointer (points to middle of static data)
29	$sp	Stack pointer (points to last location on stack)
30	$fp	Frame pointer (preserved across procedure calls)
31	$ra	Return address