DPST1092 24T3 — MIPS Functions

https://www.cse.unsw.edu.au/~dp1092/24T3/

Functions in MIPS

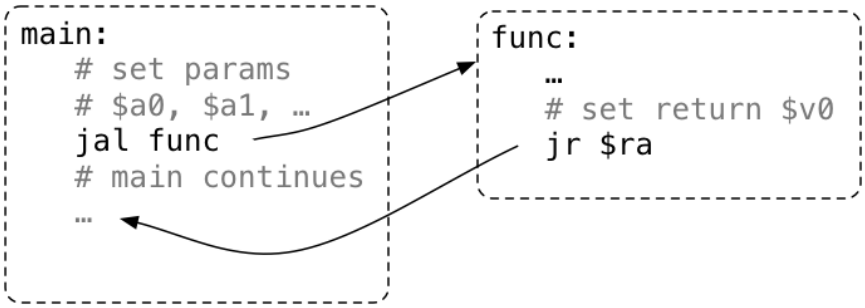
When we call a function (e.g. `fun(x+1,5,&y)`):

- the arguments (actual parameters) are evaluated
- argument values are made available for the function
- control is transferred to the code for the function
- local variables are created
- the function code is executed in this environment
- the return value is set up
- control transfers back to where the function was called from
- the caller receives the return value

Function Calls

Simple view of function calls:

- load argument values into `$a0`, `$a1`, ...
- invoke `jal`: loads the address of the next instruction (PC+4) into `$ra`, jumps to function (by setting the PC to the address of the first instruction of the function)
- function puts return value in `$v0`
- returns to caller using `jr $ra`



Function Calls: Return Address

The `jr $ra` in main below will fail, because `jal hello` changed `$ra`

A function that calls another function must save $ra.

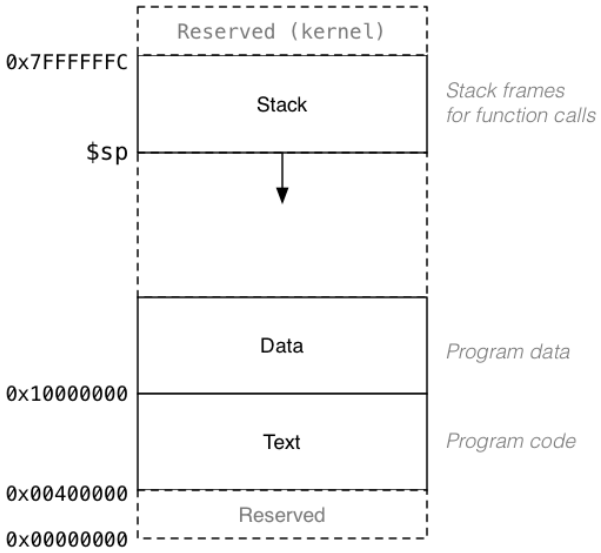
```
int main(void) {
    hello();
    return 0;
}

void hello(void) {
    printf("hi\n");
}
```

```
main:
    jal  hello
    li $v0, 0
    jr $ra # THIS WILL FAIL
hello:
    la $a0, string
    li $v0, 4
    syscall
    jr $ra
    .data
string: .asciiz "hi\n"
```


Function Calls: Return Address

Data associated with function calls is placed on the MIPS stack.



Function Calls: Return Address

Each function allocates a small section of the stack (a *frame*)

- used for: saved registers, local variables, parameters to callees
- created in the function *prologue* (pushed)
- removed in the function *epilogue* (popped)

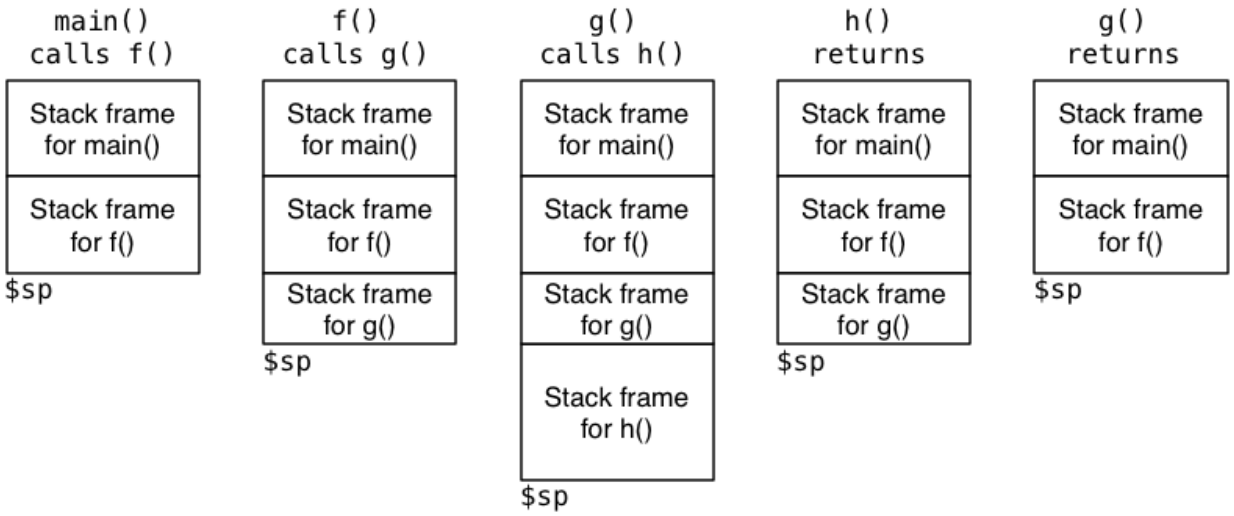
Why we use a stack:

- function `f()` calls `g()` which calls `h()`
- `h()` runs, then finishes and returns to `g()`
- `g()` continues, then finishes and returns to `f()`

i.e. last-called, exits-first (last-in, first-out) behaviour

Function Calls: Return Address

How the stack changes as functions are called and return:



Pushing and Popping a Value to and from the Stack

Pushing value of e.g. `$ra` onto stack means:

```
addi $sp, $sp, -4       OR      sw   $ra, -4($sp)
sw   $ra, ($sp)                 addi $sp, $sp, -4
```

Popping a value of e.g. `$ra` from the stack means:

```
lw   $ra, ($sp)
addi $sp, $sp, 4
```

Or the easy way if we are using mipsy is simply the pseudo instructions push and pop

```
push $ra

pop $ra
```


Saving the Return Address

A function calling another function must save the $ra

```
main:
                        # Push $ra onto the stack
    addi  $sp, $sp, -4  # move stack pointer down
                        # to allocate 4 bytes
    sw    $ra, 0($sp)   # save $ra on $stack

    jal   hello         # call hello

    li    $v0, 0        # set return value to 0

                        # Pop $ra from the stack
    lw    $ra, 0($sp)   # recover $ra from stack
    addi  $sp, $sp, 4   # move stack pointer back up
                        # to what it was when main called

    jr    $ra           # return
```

Saving the Return Address Using push and pop

A function calling another function must save the $ra

```
main:
    push $ra            # Push $ra onto the stack

    jal  hello          # call hello

    li   $v0, 0         # set return value to 0

    pop $ra             # Pop $ra from the stack

    jr   $ra            # return
```


MIPS Register usage conventions

- `$a0..$a3` contain first 4 arguments
- `$v0` contains return value
- `$ra` contains return address
- if function changes `$sp, $fp, $s0..$s8` it restores their value
- callers assume `$sp, $fp, $s0..$s8` unchanged by call (jal)
- a function may destroy the value of other registers e.g. `$t0..$t9`
- callers must assume value in e.g. `$t0..$t9` changed by call (jal) and save them themselves if needed after the function call.

MIPS Register usage conventions (not covered in DPST1092)

- floating point registers used to pass/return float/doubles
- similar conventions for saving floating point registers
- stack used to pass arguments after first 4
- stack used to pass arguments or return values which do not fit in register eg. argument or return value can be a struct, which is any number of bytes

Structure of Functions

Functions in MIPS have the following general structure:

```
# start of function
FuncName:
   # function prologue
   #   save $ra register
   #   save other relevant registers (any s register used in the function)
   #   update $sp
   ...
   # function body
   #   perform computation using $a0, $a1 etc.
   #   leaving result in $v0
   ...
   # function epilogue
   #   restore other saved registers (eg $s0 etc).
   #   restore $ra
   #   clean up stack frame (update $sp)
   jr  $ra
```

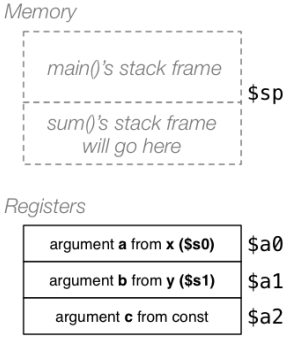

Example: Simple function call

```
int main()
{
   // x is $s0, y is $s1, z is $s2
   int x = 5; int y = 7; int z;
   ...
   z = sum(x,y,30);
   ...
}

int sum(int a, int b, int c)
{
   return a+b+c;
}
```


Example: Simple function call

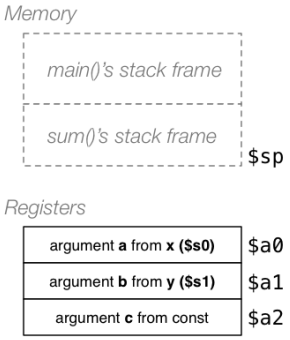
Simple function call:




```
# …
# z = sum(x,y,30);
# …
# set up args
move $a0, $s0
move $a1, $s1
li   $a2, 30
# call sum()
jal  sum
# $s2 = return value
move $s2, $v0
```

Example: Simple function call

Execution of `sum()` function:



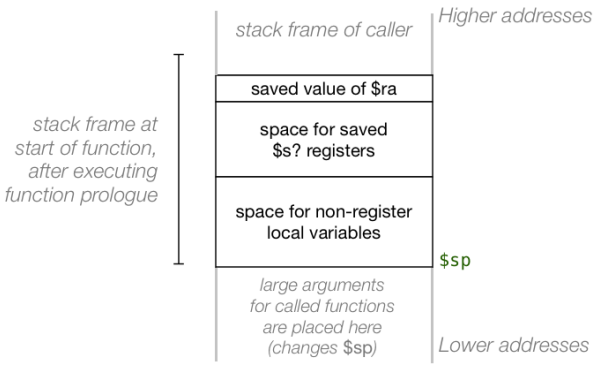

```
sum:
# set up stack frame

move $v0, $a0
add  $v0, $v0, $a1
add  $v0, $v0, $a2

# remove stack frame
jr   $ra
```


Example: Simple function call

Contents of a typical stack frame:



Exercise: Function to compute 1+2+3+...+n

Implement the function `sumTo()`

```
int main(void) {
   int max;
   print("Enter +ve integer: ");
   scanf("%d", &max);
   printf("Sum 1..%d = %d\n", max, sumTo(max));
   return 0;
}

int sumTo(int n) {
   int sum = 0;
   for (int i = 1; i <= n; i++)
       sum += i;
   return sum;
}
```

Exercise: Function to compute 1+2+3+...+n recursively

Implement the function `sumTo()` recursively

```
int main(void) {
   int max;
   print("Enter +ve integer: ");
   scanf("%d", &max);
   printf("Sum 1..%d = %d\n", max, sumTo(max));
   return 0;
}

int sumTo(int n) {
   if(n == 0){
       return 0;
   }
   return n + sumTo(n-1);
}
```


Exercise: Function to sum values in array

Implement a MIPS version of the following:

```
int array[10] = {5,4,7,6,8,9,1,2,3,0};

int main(void) {
   printf("%d\n", sumOf(array,10));
   return 0;
}

int sumOf(int a[], int n) {
   int i;  int sum = 0;
   for (i = 0; i < n; i++)
       sum += a[i];
   return sum;
}
```


Exercise: Recursive function to sum values in array

Implement a MIPS version of the following:

```
int array[10] = {5,4,7,6,8,9,1,2,3,0};

int main(void) {
   printf("%d\n", sumOf(array,0,9));
   return 0;
}

int sumOf(int a[], int lo, int hi) {
   if (lo > hi)
      return 0;
   else
      return a[lo] + sumOf(a,lo+1,hi);
}
```

Local Variables

Some local (function) variables must be stored on stack e.g. variables such as arrays and structs

```
int main(void) {
    int squares[10];
    int i = 0;
    while (i < 10) {
        squares[i] = i * i;
        i++;
    }
```

```
main:
    sub  $sp, $sp, 40
    li   $t0, 0
loop0:
    bge  $t0, 10, end0
    mul  $t1, $t0, 4
    add  $t2, $t1, $sp
    mul  $t3, $t0, $t0
    sw   $t3, ($t2)
    add  $t0, $t0, 1
    j    loop0
end0:
    addi $sp, $sp, 40
```


The Frame Pointer

frame pointer $fp is a second register pointing to stack

by convention, set to point at start of stack frame

provides a fixed point during function code execution useful for functions which grow stack (change $sp) during execution

makes it easier for debuggers to forensically analyze stack e.g if you want to print stack backtrace after error

frame pointer is optional (in CP1521 and generally)

often omitted when fast execution or small code a priority

The Frame Pointer

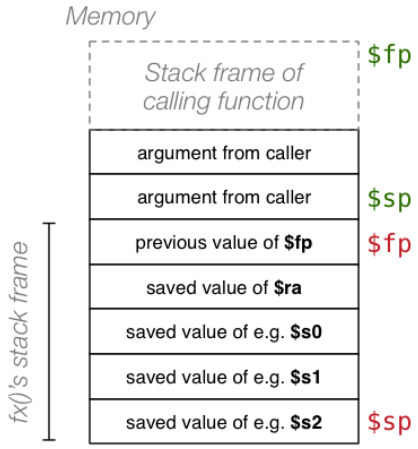
If you are using mipsy you can do it the easy way using **`begin`** and **`end`**

```
#prolog
begin       #create new frame
push $ra
push $s0
push $s1
push $s2
....

#epilog
pop $s2
pop $s1
pop $s0
pop $ra
end          #remove frame
```

The Frame Pointer - Advanced

Example of function `fx()`, which uses `$s0`, `$s1`, `$s2`

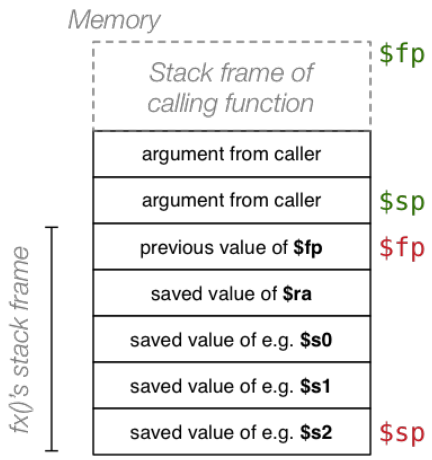



```
# start of fx() function
fx:
# fx()'s prologue
sw   $fp, -4($sp)
sw   $ra, -8($sp)
sw   $s0, -12($sp)
sw   $s1, -16($sp)
sw   $s2, -20($sp)
la   $fp, -4($sp)
add  $sp, $sp, -20
# rest of fx()'s code
```

```
initial values: $fp,$sp
final values: $fp,$sp
```


The Frame Pointer - Advanced

Alternatively ... (more explicit `push`)

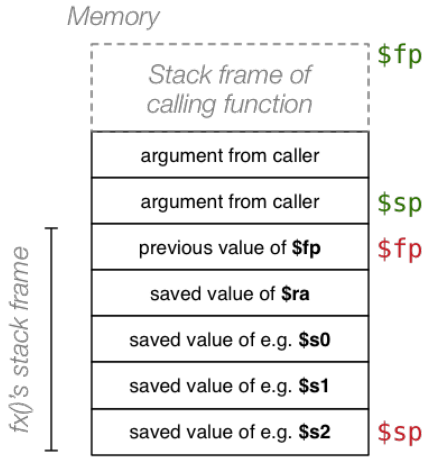



```
# start of fx() function
fx:
# fx()'s prologue
addi $sp, $sp, -4
sw   $fp, ($sp)
move $fp, $sp
addi $sp, $sp, -4
sw   $ra, ($sp)
addi $sp, $sp, -4
sw   $s0, ($sp)
addi $sp, $sp, -4
sw   $s1, ($sp)
addi $sp, $sp, -4
sw   $s2, ($sp)
# rest of fx()'s code
```

```
initial values: $fp,$sp
final values: $fp,$sp
```


The Frame Pointer - Advanced

Alternatively ... (relative to new `$fp`)

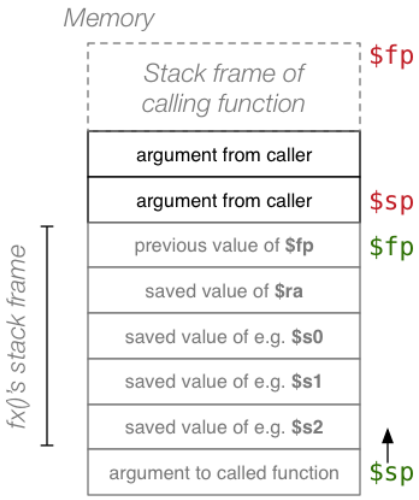



```
# start of fx() function
fx:
# fx()'s prologue
sw   $fp, -4($sp)
la   $fp, -4($sp)
sw   $ra, -4($fp)
sw   $s0, -8($fp)
sw   $s1, -12($fp)
sw   $s2, -16($fp)
add  $sp, $sp, -20
# rest of fx()'s code
```

```
initial values: $fp,$sp
final values: $fp,$sp
```

The Frame Pointer - Advanced

Example of function `fx()`, which uses `$s0`, `$s1`, `$s2`




```
# end of fx() function
lw   $v0, return value
# fx()'s epilogue
lw   $s2, -16($fp)
lw   $s1, -12($fp)
lw   $s0, -8($fp)
lw   $ra, -4($fp)
# remove stack frame
la   $sp, 4($fp)
lw   $fp, ($fp)
# return
jr   $ra

initial values: $fp,$sp
final values: $fp,$sp
```