COMP1521 21T3 — MIPS Functions

https://www.cse.unsw.edu.au/~cs1521/21T3/
When we call a function:

- the arguments are evaluated and set up for 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, $a2, $a3.
- **jal function** set $ra to PC+4 and jumps to function
- function puts return value in $v0
- returns to caller using **jr $ra**
**Function with No Parameters or Return Value**

- **jal hello** sets $ra to address of following instruction, and transfers execution to **hello**
- **jr $ra** transfers execution to the address in $ra

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

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

```asm
main:
    ...  
    jal hello
    ...

hello:
    la $a0, string
    li $v0, 4
    syscall
    jr $ra
.data
string:
    .asciiz "hi\n"
```
By convention, function return value is passed back in $v0

```c
int main(void) {
    int a = answer();
    printf("%d\n", a);
    return 0;
}

int answer(void) {
    return 42;
}
```
Function with a Return Value and Parameters

By convention, first 4 parameters are passed in $a0, $a1, $a2, $a3; if there are more (or more complex) parameters, they are passed on the stack

```c
int main(void) {
    int a = product(6, 7);
    printf("%d\n", a);
    return 0;
}

int product(int x, int y) {
    return x * y;
}
```

```mips
main:
    ...  
    li $a0, 6
    li $a1, 7
    jal product
    move $a0, $v0
    li $v0, 1
    syscall
    ...  
product:
    mul $v0, $a0, $a1
    jr $ra
```
A function that calls another function must save $ra.

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

```c
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"
```
void f(void);
int main(void) {
    printf("calling function f\n");
    f();
    printf("back from function f\n");
    return 0;
}
void f(void) {
    printf("in function f\n");
}
Simple Function Call Example - broken MIPS

```
la $a0, string0  # printf(“calling function f\n”);
li $v0, 4
syscall
jal f          # set $ra to following address
la $a0, string1 # printf(“back from function f\n”);
li $v0, 4
syscall
li $v0, 0      # fails because $ra changes since main called
jr $ra          # return from function main
f:
la $a0, string2 # printf(“in function f\n”);
li $v0, 4
syscall
jr $ra          # return from function f
.data
```
The Stack: Where it is in Memory

Data associated with a function call placed on the stack:

![Diagram of the stack with memory segments labeled as follows:
- **Reserved (kernel)**: 0x7FFFFFFFC
- **Stack frames for function calls**
- **Data**: 0x10000000
- **Program data**
- **Program code**
- **Text**

https://www.cse.unsw.edu.au/~cs1521/21T3/
The Stack: Allocating Space

- \$sp (stack pointer) initialized by operating system
- always 4-byte aligned (divisible by 4)
- points at currently used (4-byte) word
- grows downward (towards smaller addresses)
- a function can do this to allocate 40 bytes:
  \[
  \text{sub} \quad \$sp, \$sp, 40 \quad \# \text{move stack pointer down}
  \]

- a function **must** leave \$sp at original value
- so if you allocated 40 bytes, before return \((\text{jr} \quad \$ra)\)
  \[
  \text{add} \quad \$sp, \$sp, 40 \quad \# \text{move stack pointer back}
  \]
The Stack: Saving and Restoring Registers

f:

```
    sub   $sp, $sp, 12       # allocate 12 bytes
    sw    $ra, 8($sp)        # save $ra on $stack
    sw    $s1, 4($sp)        # save $s1 on $stack
    sw    $s0, 0($sp)        # save $s0 on $stack

    lw     $s0, 0($sp)       # restore $s0 from $stack
    lw     $s1, 4($sp)       # restore $s1 from $stack
    lw     $ra, 8($sp)       # restore $ra from $stack
    add    $sp, $sp, 12      # move stack pointer back
    jr      $ra              # return
```
The Stack: Growing & Shrinking

How stack changes as functions are called and return:

```
main() calls f()
  Stack frame for main()
  Stack frame for f()

f() calls g()
  Stack frame for main()
  Stack frame for f()
  Stack frame for g()

h() returns
  Stack frame for main()
  Stack frame for f()
  Stack frame for g()
  Stack frame for h()

h() returns
  Stack frame for main()
  Stack frame for f()
```
A function that calls another function must save $ra.

```assembly
main:
    sub $sp, $sp, 4  # move stack pointer down
    # to allocate 4 bytes
    sw $ra, 0($sp)  # save $ra on $stack

    jal hello        # call hello

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

    li $v0, 0        # return 0
    jr $ra
```

https://www.cse.unsw.edu.au/~cs1521/21T3/
Simple Function Call Example - correct MIPS

```
la    $a0, string0     # printf("calling function f\n");
li    $v0, 4
syscall
jal   f                # set $ra to following address
la    $a0, string1    # printf("back from function f\n");
li    $v0, 4
syscall
lw    $ra, 0($sp)     # recover $ra from $stack
addi  $sp, $sp, 4     # move stack pointer back to what it was
li    $v0, 0          # return 0 from function main
jr     $ra

f:
la    $a0, string2    # printf("in function f\n");
li    $v0, 4
syscall
jr     $ra            # return from function f
```

source code for call_return.s

https://www.cse.unsw.edu.au/~cs1521/21T3/
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 ($jal$)
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 which do not fit in register

stack used to return values which do not fit in register

for example C argument or return value can be a struct, which is any number of bytes
int answer(void);
int main(void) {
    int a = answer();
    printf("%d\n", a);
    return 0;
}
int answer(void) {
    return 42;
}
Example - Returning a Value - MIPS

main:

```mips
addi $sp, $sp, -4  # move stack pointer down to make room
sw $ra, 0($sp)    # save $ra on $stack
jal answer        # call answer, return value will be in $v0
move $a0, $v0     # printf("%d", a);
li   $v0, 1
```

```mips
syscall
li $a0, '\n'       # printf("\n");
li $v0, 11
```

```mips
syscall
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            #
```

answer: # code for function answer

```mips
li   $v0, 42       #
jr   $ra           # return from answer
```
void two(int i);
int main(void) {
    two(1);
}
void two(int i) {
    if (i < 1000000) {
        two(2 * i);
    }
    printf("%d\n", i);
}
Example - Argument & Return - MIPS (main)

```assembly
main:
    addi $sp, $sp, -4  # move stack pointer down to make room
    sw $ra, 0($sp)    # save $ra on $stack
    li $a0, 1         # two(1);
    jal two           # recover $ra from $stack
    lw $ra, 0($sp)    # move stack pointer back up to what it was when main called
    addi $sp, $sp, 4  # return from function main
    jr $ra
```

source code for two_powerful.s

https://www.cse.unsw.edu.au/~cs1521/21T3/
Example - Argument & Return - MIPS (two)

two:

```
addi $sp, $sp, -8  # move stack pointer down to make room
sw $ra, 4($sp)    # save $ra on $stack
sw $a0, 0($sp)    # save $a0 on $stack
bge $a0, 1000000, print
mul $a0, $a0, 2   # restore $a0 from $stack
jal two
```

print:

```
lw $a0, 0($sp)    # restore $a0 from $stack
li $v0, 1        # printf("%d");
syscall
li $a0, \
li $v0, 11
syscall
lw $ra, 4($sp)   # restore $ra from $stack
addi $sp, $sp, 8 # move stack pointer back up to what it was when main calle
jr $ra           # return from two
```

source code for two_powerful.s

https://www.cse.unsw.edu.au/~cs1521/21T3/
Example - More complex Calls - C

```c
int main(void) {
    int z = sum_product(10, 12);
    printf("%d\n", z);
    return 0;
}

int sum_product(int a, int b) {
    int p = product(6, 7);
    return p + a + b;
}

int product(int x, int y) {
    return x * y;
}
```

source code for more_calls.c
Example - more complex Calls - MIPS (main)

main:

    addi $sp, $sp, -4    # move stack pointer down to make room
    sw $ra, 0($sp)      # save $ra on $stack
    li $a0, 10          # sum_product(10, 12);
    li $a1, 12
    jal sum_product
    move $a0, $v0       # printf("%d", z);
    li $v0, 1
    syscall
    li $a0, '\n'        # printf("%c", '\n');
    li $v0, 11
    syscall
    lw $ra, 0($sp)      # recover $ra from $stack
    addi $sp, $sp, 4    # move stack pointer back up to what it was when main called
    li $v0, 0            # return 0 from function main
    jr $ra                # return from function main

source code for more_calls.s

https://www.cse.unsw.edu.au/~cs1521/21T3/
Example - more complex Calls - MIPS (sum_product)

```assembly
sum_product:
    addi $sp, $sp, -12      # move stack pointer down to make room
    sw $ra, 8($sp)          # save $ra on $stack
    sw $a1, 4($sp)          # save $a1 on $stack
    sw $a0, 0($sp)          # save $a0 on $stack
    li $a0, 6               # product(6, 7);
    li $a1, 7
    jal product
    lw $a1, 4($sp)          # restore $a1 from $stack
    lw $a0, 0($sp)          # restore $a0 from $stack
    add $v0, $v0, $a0       # add a and b to value returned in $v0
    add $v0, $v0, $a1       # and put result in $v0 to be returned
    lw $ra, 8($sp)          # restore $ra from $stack
    addi $sp, $sp, 12       # move stack pointer back up to what it was when main called
    jr $ra                  # return from sum_product
```

source code for more_calls.s

https://www.cse.unsw.edu.au/~cs1521/21T3/
a function which doesn’t call other functions is called a leaf function
its code *can* be simpler...

```c
int product(int x, int y) {
    return x * y;
}
```

```assembly
mul $v0, $a0, $a1
jr $ra
```

# product doesn't call other functions
# so it doesn't need to save any registers
# return argument * argument 2
#
Example - strlen using array - C

```c
int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (s[length] != 0) {
        length++;
    }
    return length;
}
```

Simple C

```c
int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    loop:
        if (s[length] == 0) goto end;
        length++;
        goto loop;
    end:
        return length;
}
```

Source code for strlen_array.c

Source code for strlen_array.simple.c
int main(void) {
    int i;
    int *p;
    p = &answer;
    i = *p;
    printf("%d\n", i); // prints 42
    *p = 27;
    printf("%d\n", answer); // prints 27
    return 0;
}
Example - pointer - MIPS

main:
    la $t0, answer       # p = &answer;
    lw $t1, ($t0)        # i = *p;
move $a0, $t1          # printf("%d\n", i);
    li $v0, 1
    syscall
    li $a0, '\n'         # printf("%c", '\n');
    li $v0, 11
    syscall
    li $t2, 27            # *p = 27;
    sw $t2, ($t0)         #
    lw $a0, answer        # printf("%d\n", answer);
    li $v0, 1
    syscall
    li $a0, '\n'         # printf("%c", '\n');
    li $v0, 11
    syscall
    li $v0, 0             # return 0 from function main
    jr $ra

https://www.cse.unsw.edu.au/~cs1521/21T3/
int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (s[length] != 0) {
        length++;
    }
    return length;
}
Example - strlen using pointer - MIPS (my_strlen)

```assembly
la $a0, string  # my_strlen("Hello");
jal my_strlen
move $a0, $v0    # printf("%d", i);
li $v0, 1
syscall
li $a0, \n     # printf("\n");
li $v0, 11
syscall
lw $ra, 0($sp)  # recover $ra from $stack
addi $sp, $sp, 4 # move stack pointer back up to what it was when main called
li $v0, 0       # return 0 from function main
jr $ra
```

source code for strlen_array.s

[https://www.cse.unsw.edu.au/~cs1521/21T3/](https://www.cse.unsw.edu.au/~cs1521/21T3/)
Storing A Local Variables On the Stack

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

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

Source code for squares.c

```asm
main:
    sub $sp, $sp, 40
    li $t0, 0

loop0:
    mul $t1, $t0, 4
    add $t2, $t1, $sp
    mul $t3, $t0, $t0
    sw $t3, ($t2)
    add $t0, $t0, 1
    b loop0

end0:
```

Source code for squares.s
Example - strlen using pointer - C

```c
int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (s[length] != 0) {
        length++;
    }
    return length;
}
```

source code for strlen_array.c

https://www.cse.unsw.edu.au/~cs1521/21T3/
What is a 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 COMP1521 and generally)
- often omitted when fast execution or small code a priority
Example of Growing Stack Breaking Function Return

```c
void f(int a) {
    int length;
    scanf("%d", &length);
    int array[length];
    // ... more code ...
    printf("%d\n", a);
}
```

```assembly
f:
    sub $sp, $sp, 4
    sw $ra, 0($sp)
    li $v0, 5
    syscall
    # allocate space for
    # array on stack
    mul $t0, $v0, 4
    sub $sp, $sp, $t0
    # ... more code ...
    # breaks because $sp
    # has changed
    lw $ra, 0($sp)
    add $sp, $sp, 4
    jr $ra
```

source code for frame_pointer.c

source code for frame_pointer.broken.s

https://www.cse.unsw.edu.au/~cs1521/21T3/
Example of Frame Pointer Use

```c
void f(int a) {
    int length;
    scanf("%d", &length);
    int array[length];
    // ... more code ... 
    printf("%d\n", a);
}
```

Source code for frame_pointer.c

```mips
f:

sub    $sp, $sp, 8
sw     $fp, 4($sp)
sw     $ra, 0($sp)
add    $fp, $sp, 8
li     $v0, 5
syscall
mul    $t0, $v0, 4
sub    $sp, $sp, $t0
# ... more code ...
lw     $ra, -4($fp)
move   $sp, $fp
lw     $fp, 0($fp)
jr     $ra
```

Source code for frame_pointer.s

https://www.cse.unsw.edu.au/~cs1521/21T3/