COMP1521 22T2 — MIPS Functions

https://www.cse.unsw.edu.au/~cs1521/22T2/
When we call a function:

- in code at the start of the function, called the **prologue**
  - the arguments are evaluated and set up for function
  - control is transferred to the code for the function
  - local variables are created
- the code for the function body is then executed
- in code at the end of the function, called the **epilogue**
  - 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

```
main:
    # set params
    # $a0, $a1, ...
    jal func
    # main continues
    ...

func:
    ...
    # set return $v0
    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();
    hello();
    hello();
    return 0;
}

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

```assembly
main:
    ...  
    jal hello
    jal hello
    jal hello
    ...

hello:
    la $a0, string
    li $v0, 4
    syscall
    jr $ra

.data
string:
    .asciiz "hi\n"
```
Function with a Return Value but No Parameters

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;
}
```

```
main:
    ...
    jal answer
    move $a0, $v0
    li $v0, 1
    syscall
    ...

answer:
    li $v0, 42
    jr $ra
```
Function with a Return Value and Parameters

By convention, first 4 parameters are passed in $a0, $a1, $a2, $a3
If there are more parameters they are passed on the stack
Parameters too big to fit in a register, such as structs, also 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;
}
```
Function calling another function ... DO NOT DO THIS

Functions that do not call other functions - *leaf functions* - are simpler to implement.

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");
}
```

```assembly
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

```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
```

.source code_for_call_return.broken.s
The Stack: Where it is in Memory

Data associated with a function call placed on the stack:

- **Reserved (kernel)**
- **Stack frames for function calls**
- **Program data**
- **Program code**
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:

```assembly
sub $sp, $sp, 40    # move stack pointer down
```

- a function **must** leave $sp$ at original value
- so if you allocated 40 bytes, before return (jr $ra)

```assembly
add $sp, $sp, 40    # move stack pointer back
```
The Stack: Saving and Restoring Registers - the Hard Way

f:

```
# function prologue code
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

...  # function body code

# function epilogue code
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: Saving and Restoring Registers - the Easy way

```plaintext
f:

    # function prologue code
    push $ra          # save $ra on $stack
    push $s1          # save $s1 on $stack
    push $s0          # save $s0 on $stack

    ...                  # function body code

    # function epilogue code
    pop $s0            # restore $s0 from $stack
    pop $s1            # restore $s1 from $stack
    pop $ra            # restore $ra from $stack

note must pop everything push-ed, must be in reverse order

push & pop are pseudo-instructions available only on mipsy
```
The Stack: Growing & Shrinking

How stack changes as functions are called and return:

- `main()` calls `f()`
  - `f()` calls `g()`
    - `g()` calls `h()`
      - `h()` returns
  - `f()` returns

- `main()` calls `f()`
  - `f()` calls `g()`
    - `g()` returns

- `main()` calls `f()`
  - `f()` returns
Function calling another function ... how to do it right

A function that calls another function must save $ra.

```
main:
    # prologue
    push $ra  # save $ra on $stack

jal hello  # call hello

    # epilogue
    pop $ra    # recover $ra from $stack
    li $v0, 0  # return 0
    jr $ra     #
```
Simple Function Call Example - correct hard way

```
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_raw.s
Simple Function Call Example - correct easy way

```
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
pop $ra             # recover $ra from $stack
li $v0, 0           # return 0 from function main
jr $ra              #

# f is a leaf function so it doesn't need an epilogue or prologue

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
MIPS Register usage conventions

- $a0..a3$ contain first 4 arguments
- $v0$ contains return value
- $ra$ contains return address
- If function changes $sp$, $fp$, $s0..s7$ it restores their value
- Callers assume $sp$, $fp$, $s0..s7$ 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)
MIPS Register usage conventions (not covered in COMP1521)

- 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, a struct can be an C function argument or function return value, but a struct can be 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

# code for function main

main:
  begin    # move frame pointer
  push $ra # save $ra onto stack
  jal answer # call answer(), return value will be in $v0
  move $a0, $v0 # printf("%d", a);
  li $v0, 1 #
  syscall #
  li $a0, '\n' # printf("%c", '\n');
  li $v0, 11 #
  syscall #
  pop $ra # recover $ra from stack
  end      # move frame pointer back
  li $v0, 0 # return
  jr $ra #

# code for function answer

answer:
  li $v0, 42 # return 42
  jr $ra #
Example - Argument & Return - C

```c
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)

```
main:
  begin
  push $ra
  li $a0, 1
  jal two
  pop $ra
  end
  li $v0, 0
  jr $ra
```

Source code for two_powerful.s
Example - Argument & Return - MIPS (two)

```mips
# Example code for two_powerful.s

two:
    begin            # move frame pointer
        push $ra      # save $ra onto stack
        push $a0      # save $a0 onto stack
        bge $a0, 1000000, two_end_if
        mul $a0, $a0, 2
        jal two

two_end_if:
        pop $a0       # restore $a0 from $stack
        li $v0, 1     # printf("%d");
        syscall
        li $a0, \n    # printf("%c", \n');
        li $v0, 11
        syscall
        pop $ra       # recover $ra from stack
        end            # move frame pointer back
        jr $ra         # return from two
```

Source code for two_powerful.s
```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;
}
```
main:

begin             # move frame pointer
push   $ra       # save $ra onto 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
pop    $ra       # recover $ra from stack
end            # move frame pointer back
li    $v0, 0     # return 0 from function main
jr    $ra        # return from function main
sum_product:
begin
    # move frame pointer
    push $ra
    # save $ra onto stack
    push $a0
    # save $a0 onto stack
    push $a1
    # save $a1 onto stack
    li $a0, 6
    # product (6, 7);
    li $a1, 7
    jal
    pop $a1
    # recover $a1 from stack
    pop $a0
    # recover $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
    pop $ra
    # recover $ra from stack
    end
    # move frame pointer back
    jr $ra
    # return from sum_product
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;
}
```

source code for more_calls.c

```
product:
    # product doesn't call other functions
    # so it doesn't need to save any registers
mul  $v0, $a0, $a1  # return argument * argument 2
jr   $ra
```

source code for more_calls.s
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;
}
```
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)

```
l a  $a0, string           # my_strlen("Hello");
 j a l  my_strlen
 m o v e  $a0, $v0            # printf("%d", i);
 l i  $v0, 1
 s y s c a l l
 l i  $a0, '\n'              # printf("%c", '\n');
 l i  $v0, 11
 s y s c a l l
 p o p   $ra                # recover $ra from stack
 e n d   # move frame pointer back
 l i  $v0, 0                # return 0 from function main
 j r   $ra                   #
```

source code for strlen arrays
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:
mul  $t1, $t0, 4
add  $t2, $t1, $sp
mul  $t3, $t0, $t0
sw   $t3, ($t2)
add  $t0, $t0, 1
b    loop0

dere

}
```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
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.
- Using a frame pointer is optional - both in COMP1521 and generally.
- A frame pointer is 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
", a);
}
```

```
f:
    # prologue
    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 ...

    # epilogue
    # breaks because $sp has changed
    lw $ra, 0($sp)
    add $sp, $sp, 4
    jr $ra
```

```
### Example of Frame Pointer Use - Hard Way

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

### Assembly Code

```assembly
f:
    # prologue
    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 ...

    # epilogue
    lw $ra, -4($fp)
    move $sp, $fp
    lw $fp, 0($fp)
    jr $ra
```
Example of Frame Pointer Use - Easy Way

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

```assembly
f:
    # prologue
    begin
    push $ra
    li $v0, 5
    syscall
    mul $t0, $v0, 4
    sub $sp, $sp, $t0
    # ... more code ...

    # epilogue
    pop $ra
    end
    jr $ra
```

- `begin` & `end` are pseudo-instructions available only on mipsy.