Computer Systems Fundamentals

call_return.c

C Function with No Parameters or Return Value

```c
#include <stdio.h>

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");
}
```

call_return.broken.s

simple example of returning from a function loops because main does not save return address

```
main:
    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
string0:
    .asciiz "calling function f\n"
string1:
    .asciiz "back from function f\n"
string2:
    .asciiz "in function f\n"
```

call_return_raw.s

simple example of placing return address on stack note stack grows down

```
main:
    addi $sp, $sp, -4    # move stack pointer down to make room
    sw   $ra, 0($sp)     # save $ra on $stack

    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


    .data
string0:
    .asciiz "calling function f\n"
string1:
    .asciiz "back from function f\n"
string2:
    .asciiz "in function f\n"
```

call_return.s

simple example of placing return address on stack begin, end, push pop are pseudo-instructions provided by mipsy but not spim

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

    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


    .data
string0:
    .asciiz "calling function f\n"
string1:
    .asciiz "back from function f\n"
string2:
    .asciiz "in function f\n"
```

return_answer.c

simple example of returning a value from a function

```
#include <stdio.h>

int answer(void);

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

int answer(void) {
    return 42;
}
```

return_answer.s

simple example of returning a value from a function note storing of return address $ra

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               #
```

more_calls.c

example of function calls

```
#include <stdio.h>

int sum_product(int a, int b);
int product(int x, int y);

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;
}
```

more_calls.s

example of function calls note storing of return address $a0, $a1 and $ra on stack

```
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  $s0               # save $s0 onto stack
    push  $s1               # save $s1 onto stack

    move  $s0, $a0          # preserve $a0 for use after function call
    move  $s1, $a1          # preserve $a1 for use after function call

    li    $a0, 6            # product(6, 7);
    li    $a1, 7
    jal   product



    add   $v0, $v0, $s0     # add a and b to value returned in $v0
    add   $v0, $v0, $s1     # and put result in $v0 to be returned

    pop   $s1               # recover $s1 from stack
    pop   $s0               # recover $s0 from stack
    pop   $ra               # recover $ra from stack
    end                     # move frame pointer back

    jr    $ra               # return from sum_product


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                #
```

two_powerful.c

recursive function which prints first 20 powers of two in reverse

```
#include <stdio.h>

void two(int i);

int main(void) {
    two(1);
}

void two(int i) {
    if (i < 1000000) {
        two(2 * i);
    }
    printf("%d\n", i);
}
```

two_powerful.s

simple example of placing return address ($ra) and $a0 on the stack recursive function which prints first 20 powers of two in reverse

```
main:
    begin                   # move frame pointer
    push  $ra               # save $ra onto stack

    li    $a0, 1
    jal   two               # two(1);

    pop   $ra               # recover $ra from stack
    end                     # move frame pointer back

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


two:
    begin                   # move frame pointer
    push  $ra               # save $ra onto stack
    push  $s0               # save $s0 onto stack

    move  $s0, $a0

    bge   $a0, 1000000, two_end_if
    mul   $a0, $a0, 2
    jal   two
two_end_if:

    move  $a0, $s0
    li    $v0, 1            # printf("%d");
    syscall

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

    pop   $s0               # recover $s0 from stack
    pop   $ra               # recover $ra from stack
    end                     # move frame pointer back

    jr    $ra               # return from two
```

squares.c

store first 10 squares into an array which is a local variable then print them from array

```
#include <stdio.h>

int main(void) {
    int squares[10];
    int i = 0;
    while (i < 10) {
        squares[i] = i * i;
        i++;
    }
    i = 0;
    while (i < 10) {
        printf("%d", squares[i]);
        printf("%c",'\n');
        i++;
    }
    return 0;
}
```

squares.s

i in register $t0 registers $t1, and $t2, used to hold temporary results

```
main:
    begin                   # move frame pointer
    addi $sp, $sp, -40      # move stack pointer down to make room to store array numbers on stack

    li   $t0, 0             # i = 0
loop0:
    bge  $t0, 10, end0      # while (i < 10) {

    mul  $t1, $t0, 4        #    calculate &numbers[i]
    add  $t2, $t1, $sp      #
    mul  $t3, $t0, $t0      #    calculate i * i
    sw   $t3, ($t2)         #    store in array

    addi $t0, $t0, 1        #    i++;
    b    loop0              # }
end0:

    li   $t0, 0             # i = 0
loop1:
    bge  $t0, 10, end1      # while (i < 10) {

    mul  $t1, $t0, 4
    add  $t2, $t1, $sp      #   calculate &numbers[i]

    lw   $a0, ($t2)         #   load numbers[i] into $a0
    li   $v0, 1             #   printf("%d", numbers[i])
    syscall                 #

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

    addi $t0, $t0, 1        #   i++
    b    loop1              # }
end1:

    addi $sp, $sp, 40       # move stack pointer back up to what it was when main called
    end                     # move frame pointer back

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

frame_pointer.c

example of function where frame pointer useful because stack grows during function execution

```
#include <stdio.h>

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

frame_pointer.broken.s

example stack growing during function execution breaking the function return

```
f:
    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

    li   $v0, 5           # scanf("%d", &length);
    syscall

    mul  $v0, $v0, 4      # calculate array size
    sub  $sp, $sp, $v0    # move stack_pointer down to hold array

    # ...

                          # breaks because stack pointer moved down to hold array
                          # so we won't restore the correct value
    lw   $ra, 4($sp)      # restore $ra from $stack
    addi $sp, $sp, 8      # move stack pointer back up to what it was when main called

    jr   $ra              # return from f
```

frame_pointer.s

using a frame pointer to handle stack growing during function execution

```
f:
    begin                 # move frame pointer
    push  $ra             # save $ra on stack

    li    $v0, 5          # scanf("%d", &length);
    syscall

    mul   $v0, $v0, 4     # calculate array size
    sub   $sp, $sp, $v0   # move stack pointer down to hold array

    # ... more code ...

    pop   $ra             # restore $ra
    end
    jr    $ra             # return
```

strlen_array.c

calculate the length of a string using a strlen like function

```
#include <stdio.h>

int my_strlen(char *s);

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;
}
```

strlen_array.simple.c

calculate the length of a string using a strlen like function

```c
#include <stdio.h>

int my_strlen(char *s);

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;
}
```

strlen_array.s

simple example of placing return address ($ra) on the stack calculate the length of a string using a strlen like function

```
main:
    begin                   # move frame pointer
    push  $ra               # save $ra onto stack

    la   $a0, string    # my_strlen("Hello");
    jal  my_strlen

    move $a0, $v0           # printf("%d", i);
    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                #


my_strlen:                  # length in t0, s in $a0
    li   $t0, 0
loop:                       # while (s[length] != 0) {
    add  $t1, $a0, $t0  #   calculate &s[length]
    lb   $t2, ($t1)     #   load s[length] into $t2
    beq  $t2, 0, end    #
    addi $t0, $t0, 1    #   length++;
    b    loop           # }
end:
    move $v0, $t0           # return length
    jr   $ra                #


    .data
string:
    .asciiz "Hello"
```

strlen_pointer.c

calculate the length of a string using a strlen like function

```c
#include <stdio.h>

int my_strlen(char *s);

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

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

strlen_pointer.simple.c

calculate the length of a string using a strlen like function

```c
#include <stdio.h>

int my_strlen(char *s);

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

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

strlen_pointer.s

simple example of placing return address ($ra) on the stack calculate the length of a string using a strlen like function

```
main:
    begin                   # move frame pointer
    push  $ra               # save $ra onto stack

    la   $a0, string        # my_strlen("Hello");
    jal  my_strlen

    move $a0, $v0           # printf("%d", i);
    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                #


my_strlen:                  # length in t0, s in $a0
    li   $t0, 0
loop:                       #
    lb   $t1, ($a0)     # load *s into $t1
    beq  $t1, 0, end        #
    addi $t0, $t0, 1        # length++
    addi $a0, $a0, 1        # s++
    b    loop               #
end:
    move $v0, $t0           # return length
    jr   $ra                #


    .data
string:
    .asciiz "Hello Andrew"
```

stack_inspect.c

```
#include <stdio.h>
#include <stdint.h>

/*
$ clang --version
Ubuntu clang version 11.0.0-2~ubuntu20.04.1
$ clang stack_inspect.c
$ a.out
 0: Address 0x7ffe7d80defc contains        3     <- e[0]
 1: Address 0x7ffe7d80df00 contains        7     <- e[1]
 2: Address 0x7ffe7d80df04 contains        5     <- d
 3: Address 0x7ffe7d80df08 contains        9     <- c
 4: Address 0x7ffe7d80df0c contains       2a     <- b
 5: Address 0x7ffe7d80df10 contains 7d80df30     <- saved frame pointer of main (lower 32 bits)
 6: Address 0x7ffe7d80df14 contains     7ffe     <- saved frame pointer of main (upper 32 bits)
 7: Address 0x7ffe7d80df18 contains   4011d3     <- saved return address of main (lower 32 bits)
 8: Address 0x7ffe7d80df1c contains        0     <- saved return address of main (upper 32 bits)

 9: Address 0x7ffe7d80df20 contains 7d80e020     <- STACK pointer (???) (lower 32 bits)
10: Address 0x7ffe7d80df24 contains     7ffe     <- STACK pointer (???) (upper 32 bits)
11: Address 0x7ffe7d80df28 contains        9     <- a
12: Address 0x7ffe7d80df2c contains        0     <- padding (for next 64-bit pointer ^)
13: Address 0x7ffe7d80df30 contains        0     <- saved frame pointer of _start (lower 32 bits)
14: Address 0x7ffe7d80df34 contains        0     <- saved frame pointer of _start (upper 32 bits)
15: Address 0x7ffe7d80df38 contains 9fbb30b3     <- saved return address of _start (lower 32 bits)
16: Address 0x7ffe7d80df3c contains     7f98     <- saved return address of _start (upper 32 bits)
17: Address 0x7ffe7d80df40 contains       71     <- ???
18: Address 0x7ffe7d80df44 contains        0     <- ???
19: Address 0x7ffe7d80df48 contains 7d80e028     <- source information
20: Address 0x7ffe7d80df4c contains     7ffe     <- source information

$ gcc --version
gcc (Ubuntu 10.3.0-1ubuntu1~20.04) 10.3.0
$ gcc stack_inspect.c
$ a.out                                          # GCC stores local arrays first in the stack, so some
things are missing from the start of the output.
                                                 9     <- c
                                                2a     <- b
                                                       <- TEXT pointer (???)
                                                       <- TEXT pointer (???)
                                                 0     <- padding (for next 64-bit pointer ^)
                                                 5     <- d
 0: Address 0x7ffccc6e6e90 contains        3     <- e[0]
 1: Address 0x7ffccc6e6e94 contains        7     <- e[1]
 2: Address 0x7ffccc6e6e98 contains f08e2f00     <- ???
 3: Address 0x7ffccc6e6e9c contains 8c526a01     <- ???
 4: Address 0x7ffccc6e6ea0 contains cc6e6ec0     <- saved frame pointer of main (lower 32 bits)
 5: Address 0x7ffccc6e6ea4 contains     7ffc     <- saved frame pointer of main (upper 32 bits)
 6: Address 0x7ffccc6e6ea8 contains 11c6a221     <- saved return address of main (lower 32 bits)
 7: Address 0x7ffccc6e6eac contains     5582     <- saved return address of main (upper 32 bits)

 8: Address 0x7ffccc6e6eb0 contains cc6e6fb0     <- STACK pointer (???) (lower 32 bits)
 9: Address 0x7ffccc6e6eb4 contains     7ffc     <- STACK pointer (???) (upper 32 bits)
10: Address 0x7ffccc6e6eb8 contains        0     <- padding (for next 64-bit pointer ^)
11: Address 0x7ffccc6e6ebc contains        9     <- a
12: Address 0x7ffccc6e6ec0 contains        0     <- saved frame pointer of _start (lower 32 bits)
13: Address 0x7ffccc6e6ec4 contains        0     <- saved frame pointer of _start (upper 32 bits)
14: Address 0x7ffccc6e6ec8 contains 6f6d70b3     <- saved return address of _start (lower 32 bits)
15: Address 0x7ffccc6e6ecc contains     7fba     <- saved return address of _start (upper 32 bits)
16: Address 0x7ffccc6e6ed0 contains       71     <- ???
17: Address 0x7ffccc6e6ed4 contains        0     <- ???
18: Address 0x7ffccc6e6ed8 contains cc6e6fb8     <- source information
19: Address 0x7ffccc6e6edc contains     7ffc     <- source information
20: Address 0x7ffccc6e6ee0 contains 6f898618     <- ??? (upper 32 bits)
*/

void f(int b, int c) {
    int d = 5;
    uint32_t e[2] = { 3, 7 };

    for (int i = 0; i < 20; i++)
```

```
        printf("%2d: Address %p contains %8x\n", i, &e[i], e[0 + i]);
}

int main(void) {
    int a = 9;
    f(42, a);
    return 0;
}
```

invalid0.c

```
Run at CSE like this

$ clang -Wno-everything invalid0.c -o invalid0
$ ./invalid0
42 77 77 77 77 77 77 77 77 77
```

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int a[10];
    int b[10];
    printf("a[0] is at address %p\n", &a[0]);
    printf("a[9] is at address %p\n", &a[9]);
    printf("b[0] is at address %p\n", &b[0]);
    printf("b[9] is at address %p\n", &b[9]);

    for (int i = 0; i < 10; i++) {
        a[i] = 77;
    }

    // loop writes to b[10] .. b[12] which don't exist -
    // on CSE servers  (clang 7.0 x86_64/Linux)
    // b[12] is stored where a[0] is stored
    // on CSE servers  (clang 7.0 x86_64/Linux)
    // b[10] is stored where a[0] is stored

    for (int i = 0; i <= 12; i++) {
        b[i] = 42;
    }

    // prints 42 77 77 77 77 77 77 77 77 77 on x86_64/Linux
    // prints 42 42 42 77 77 77 77 77 77 77 at CSE
    for (int i = 0; i < 10; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");

    return 0;
}
```

invalid1.c

```
Run at CSE like this


$ clang -Wno-everything invalid1.c -o invalid1
$ ./invalid1
i is at address 0x7ffe2c01cd58
a[0] is at address 0x7ffe2c01cd30
a[9] is at address 0x7ffe2c01cd54
a[10] would be stored at address 0x7ffe2c01cd58


doesn't terminate
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int i;
    int a[10];
    printf("i is at address %p\n", &i);
    printf("a[0] is at address %p\n", &a[0]);
    printf("a[9] is at address %p\n", &a[9]);
    printf("a[10] would be stored at address %p\n", &a[10]);

    // loop writes to a[10] .. a[11] which don't exist -
    // but on CSE servers  (clang 7.0 x86_64/Linux)
    // i would be stored where a[11] is stored

    for (i = 0; i <= 11; i++) {
        a[i] = 0;
    }

    return 0;
}
```

invalid2.c

```
Run at CSE like this


$ clang -Wno-everything  invalid2.c -o invalid2
$ ./invalid2
answer=42
```

```c
#include <stdio.h>

void f(int x);

int main(void) {
    int answer = 36;
    printf("answer is stored at address %p\n", &answer);

    f(5);
    printf("answer=%d\n", answer); // prints 42 not 36

    return 0;
}

void f(int x) {
    int a[10];

    // a[18] doesn't exist
    // on CSE servers  (clang 7.0 x86_64/Linux)
    // variable answer in main happens to be where a[19] would be

    printf("a[18] would be stored at address %p\n", &a[18]);

    a[18] = 42;
}
```

invalid3.c

```
Run at CSE like this


$ clang -Wno-everything invalid3.c -o invalid3
$ ./invalid3

I hate you.
$
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

void f(void);

int main(void) {
    f();

    printf("I love you and will never say,\n");
    printf("I hate you.\n");

    return 0;
}

void f(void) {
    uint64_t a[1];

#ifdef __clang__
#if __clang_major__ >= 16
#error "Don't know the correct offset for this clang version"
#elif __clang_major__ >= 14
    a[2] += 14;
#elif __clang_major__ >= 4
    a[2] += 17;
#elif __clang_major__ == 3
#if __clang_minor__ >= 5
    a[2] += 17;
#elif __clang_minor__ >= 4
    a[2] += 15;
#else
// clang 3.0-3.3 are broken on godbolt.org
#error "Don't know the correct offset for this clang version"
#endif
#else
// clang <3.0 are not avalible on godbolt.org
#error "Don't know the correct offset for this clang version"
#endif
#else
#error "This program must be compiled with clang"
#endif

    // function f has it return address on the stack
    // the call of function f from main should return to
    // the next statement which is:  printf("I love you and will never say,\n");
    //
    // on CSE servers  (clang 11.0 x86_64/Linux)
    // f's return address is stored where a[2] would be
    //
    // so changing a[2] changes where the function returns
    //
    // adding 17 to a[12] happens to cause it to return 2 statements later
    // at:  printf("I hate you.\n");
}
```

invalid4.c

```
Run at CSE like this


$ clang invalid4.c -o invalid4
$ ./invalid4
authenticated is at address 0xff94bf44
password is at address 0xff94bf3c

Enter your password: 123456789

Welcome. You are authorized.
$
```

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int authenticated = 0;
    char password[8];

    printf("authenticated is at address %p\n", &authenticated);
    printf("password[8] would be at address %p\n", &password[8]);

    printf("Enter your password: ");
    int i = 0;
    int ch = getchar();
    while (ch != '\n' && ch != EOF) {
        password[i] = ch;
        ch = getchar();
        i = i + 1;
    }
    password[i] = '\0';

    if (strcmp(password, "buffalo") == 0) {
        authenticated = 1;
    }

    // a password longer than 8 characters will overflow the array password
    // on CSE servers  (clang 7.0 x86_64/Linux)
    // the variable authenticated is at the address where
    // where password[8] would be and gets overwritten
    //
    // This allows access without knowing the correct password

    if (authenticated) {
        printf("Welcome. You are authorized.\n");
    } else {
        printf("Welcome. You are unauthorized.  Your death will now be implemented.\n");
        printf("Welcome. You will experience a tingling sensation and then death. \n");
        printf("Remain calm while your life is extracted.\n");
    }

    return 0;
}
```

func.c

```c
#include <stdio.h>

int func(int n);

int main(void) {
    printf("main is calling func(42)\n");
    int f = func(42);
    printf("func has returned back to main. func(42) returned %d\n", f);

    return 0;
}

int func(int n) {
    printf("hello from func!\n");
    return 2 * n;
}
```

factorial.c

```c
#include <stdio.h>

int factorial(int);

int main(void) {
    int input;
    scanf("%d", &input);

    int f = factorial(input);

    printf("%d", input);
    printf("! = ");
    printf("%d", f);
    putchar('\n');

    return 0;
}

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

factorial.simple.c

```c
#include <stdio.h>

int factorial(int);

int main(void) {
    int input;
    scanf("%d", &input);

    int f = factorial(input);

    printf("%d", input);
    printf("! = ");
    printf("%d", f);

    putchar('\n');

    return 0;
}

int factorial(int n) {

    int retval;

    if (n == 0) goto factorial__n_eq_0;

    retval = n * factorial(n - 1);
    goto factorial__epilogue;


factorial__n_eq_0:
    retval = 1;

factorial__epilogue:
    return retval;



}
```

factorial.s

```
main:
        # Args: void
        # Returns:
        #       - $v0: int
        #
        # Locals:
        #       - $s0: int input
        #       - $t0: int f
        #
        # Stack:        [$ra, $s0]
        # Uses:         [$ra, $s0, $v0, $t0, $a0]
        # Clobbers:     [$v0, $t0, $a0]
        #
        # Structure:
        # -> main
        #       -> [prologue]
        #       -> [body]
        #       -> [epilogue]

main__prologue:
        begin
        push    $ra
        push    $s0

        # The following two instructions are equivalent to `push $s0`
        # addi  $sp, $sp, -4
        # sw    $s0, ($sp)

main__body:
        li      $v0, 5                  # syscall 5: read_int
        syscall                         #
        move    $s0, $v0                # scanf("%d", &input);

        move    $a0, $v0
        jal     factorial               #
        move    $t0, $v0                # int f = factorial(input);

        li      $v0, 1                  # syscall 1: print_int
        move    $a0, $s0                #
        syscall                         # printf("%d", input);

        li      $v0, 4                  # syscall 4: print_string
        la      $a0, main__result_msg   #
        syscall                         # printf("!= ");


        li      $v0, 1                  # syscall 1: print_int
        move    $a0, $t0                #
        syscall                         # printf("%d", f);

        li      $v0, 11                 # syscall 11: print_char
        li      $a0, '\n'               #
        syscall                         # putchar('\n');

        # Restore the original value of $s0 from memory
        # lw    $s0, ($sp)
        # addi  $sp, $sp, 4

main__epilogue:
        pop     $s0
        pop     $ra
        end

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

factorial:
        # Args:
        #       - $a0: int n
        # Returns:
        #       - $v0: int
```

```
        # Locals:
        #       - $s0: int n
        #
        # Stack:        [$ra, $s0]
        # Uses:         [$ra, $s0, $v0, $a0]
        # Clobbers:     [$v0, $t0, $a0]
        #
        # Structure:
        # -> main
        #       -> [prologue]
        #       -> [body]
        #               -> n_eq_0
        #       -> [epilogue]
factorial__prologue:
        begin
        push    $ra
        push    $s0

factorial__body:
        move    $s0, $a0

        beqz    $a0, factorial__n_eq_0          # if (n != 0) {

        addi    $a0, $a0, -1
        jal     factorial                       #   factorial(n - 1)

        mul     $v0, $v0, $s0                   #   return n * factorial(n - 1);
        j       factorial__epilogue             # }

factorial__n_eq_0:
        li      $v0, 1                          # return 1;

factorial__epilogue:
        pop     $s0
        pop     $ra
        end

        jr      $ra


        .data
main__result_msg:
        .asciiz "! = "
```

function_example.c

A simple example of a program with functions
These functions have no parameters or return values (except for main that returns 0)

```
#include <stdio.h>

void f(void);
void g(void);

int main(void){
    printf("I am in the main\n");
    f();
    printf("I am about to return from the main function\n");
    return 0;
}

void f(void){
    printf("I am in the f function\n");
    g();
    printf("I am about to return from f the function\n");
}

void g(void){
    printf("I am in the g function\n");
}
```

function_example_broken.s

Live coding exercises from recap lecture

A simple example of a program with functions

These functions have no parameters or return values (except for main that returns 0)

This code is broken!

```
        .data
msg_main1:      .asciiz "I am in the main\n"
msg_main2:      .asciiz "I am about to return from the main function\n"
msg_f1:         .asciiz "I am in the f function\n"
msg_f2:         .asciiz "I am about to return from the f function\n"
msg_g1:         .asciiz "I am in the g function\n"

        .text
main:
        li      $v0, 4          #printf("I am in the main");
        la      $a0, msg_main1
        syscall

        jal     f               # f()

        li      $v0, 4          #printf("I am in the main");
        la      $a0, msg_main2
        syscall

        li      $v0, 0          # set return value for main to be 0
        jr      $ra             # return from main to (__start)

f:
        li      $v0, 4          #printf("I am in the f function");
        la      $a0, msg_f1
        syscall

        jal     g               # g()

        li      $v0, 4          #printf("I am in the f function");
        la      $a0, msg_f2
        syscall
        jr      $ra             #return

g:
        li      $v0, 4          #printf("I am in the g function");
        la      $a0, msg_g1
        syscall
        jr      $ra             #return
```

function_example.s

Live coding exercises from recap lecture

A simple example of a program with functions

These functions have no parameters or return values (except for main that returns 0)

```
        .data
msg_main1:      .asciiz "I am in the main\n"
msg_main2:      .asciiz "I am about to return from the main function\n"
msg_f1:         .asciiz "I am in the f function\n"
msg_f2:         .asciiz "I am about to return from the f function\n"
msg_g1:         .asciiz "I am in the g function\n"

        .text
main:
main__prologue:
        push    $ra
main__body:
        li      $v0, 4          #printf("I am in the main");
        la      $a0, msg_main1
        syscall

____
        jal     f               # f()

____
        li      $v0, 4          #printf("I am in the main");
        la      $a0, msg_main2
        syscall

____
        li      $v0, 0          # set return value for main to be 0
main__epilog:
        pop     $ra
        jr      $ra             # return from main to (__start)

f:
f__prologue:
        push    $ra
f__body:
        li      $v0, 4          #printf("I am in the f function");
        la      $a0, msg_f1
        syscall

____
____    jal     g               # g()
____
____    li      $v0, 4          #printf("I am in the f function");
        la      $a0, msg_f2
        syscall

f__epilog:
        pop     $ra
        jr      $ra             #return

____
g:
g__prologue:
        push    $ra
g__body:
        li      $v0, 4          #printf("I am in the g function");
        la      $a0, msg_g1
____    syscall
g__epilog:
        pop     $ra
____    jr      $ra             #return
```

sum_to.c

An example of a C program with a function with one input and a return value

```c
#include <stdio.h>

int sum_to(int n);

int main(void) {
    int max = 10;
    int result = sum_to(max);
    printf("Sum 1.. %d = %d\n", max, result);
    return 0;
}

// An iterative solution
int sum_to(int n) {
    int sum = 0;
    int i = 1;
    while (i <= n) {
        sum = sum + i;
        i++;
    }
    return sum;
}
```

sum_to.s

A live coding example done as a recap on functions.

This has a simple function with 1 parameter and a return value

```
        .data
msg_sum1:
        .asciiz "Sum 1.. "
msg_sum2:
        .asciiz " = "

        .text
main:
        # $s0 used for max since we set it before doing the jal
        # but need it again after the jal
        # If a function uses $s0 that same function must
        # push it to the stack and pop it to restore it later

        # $t0 was used for result since we are not doing another
        # jal after we set its value so we do not need to worry
        # about it getting overwritten by another function
main__prologue:
        push    $ra
        push    $s0
main__body:
        li      $s0, 10                 # int max = 10;
        move    $a0, $s0                # int max = 10;
        jal     sum_to                  # result = sum_to(max);
        move    $t0, $v0                # $t0 = result

        li      $v0, 4                  # printf("Sum 1.. ");
        la      $a0, msg_sum1
        syscall

        li      $v0, 1                  # printf("%d",max);
        move    $a0, $s0
        syscall

        li      $v0, 4                  # printf(" = ");
        la      $a0, msg_sum2
        syscall

        li      $v0, 1                  # printf("%d", result);
        move    $a0, $t0
        syscall

        li      $v0, 11                 # putchar('\n')
        li      $a0, '\n'
        syscall

        li      $v0, 0
main__epilog:
        pop     $s0
        pop     $ra
        jr      $ra


sum_to:
sum_to__prologue:
sum_to__body:
        li      $t0, 0                  # sum = 0
        li      $t1, 1                  # i = 1

sum_to__while:
        bgt     $t1, $a0, sum_to__while_end # while (i <= n)
        add     $t0, $t0, $t1           # sum = sum + i
        addi    $t1, $t1, 1             # i++
        j       sum_to__while
sum_to__while_end:

        move    $v0, $t0                # return sum
sum_to__epilog:
        jr      $ra
```

sum_to_r.c

An example of a C program with a recursive function

```c
#include <stdio.h>

int sum_to(int n);

int main(void) {
    int max = 10;
    int result = sum_to(max);
    printf("Sum 1.. %d = %d\n", max, result);
    return 0;
}

int sum_to(int n) {
   if(n == 0){
       return 0;
   } else {
       int result = sum_to(n-1);
       return n + result;
   }
}
```

sum_to_r.s

A live coding example done as a recap on functions.

This has a recursive function

```
        .data
msg_sum1:
        .asciiz "Sum 1.. "
msg_sum2:
        .asciiz " = "

        .text
main:
main__prologue:
        push    $ra
        push    $s0
main__body:
        li      $s0, 10                 # int max = 10;
        move    $a0, $s0                # int max = 10;
        jal     sum_to                  # result = sum_to(max);
        move    $t0, $v0                # $t0 = result

        li      $v0, 4                  # printf("Sum 1.. ");
        la      $a0, msg_sum1
        syscall

        li      $v0, 1                  # printf("%d",max);
        move    $a0, $s0
        syscall

        li      $v0, 4                  # printf(" = ");
        la      $a0, msg_sum2
        syscall

        li      $v0, 1                  # printf("%d", result);
        move    $a0, $t0
        syscall

        li      $v0, 11                 # putchar('\n')
        li      $a0, '\n'
        syscall

        li      $v0, 0
main__epilog:
        pop     $s0
        pop     $ra
        jr      $ra


sum_to:
        # We move our input n from $a0 to $s0
        # We need to do this as $a0 will be modified
        # so we need to store n somewhere safe
        # as we need to use it again in the last line of
        # the function.
sum_to__prologue:
        push    $ra
        push    $s0
sum_to__body:
        move    $s0, $a0                # n $s0
sum_to_if:
        bnez    $s0, sum_to_else        # if(n == 0)
        li      $v0, 0                  # return 0
        j       sum_to__epilog
sum_to_else:
        sub     $a0, $s0, 1             # n - 1
        jal     sum_to                  # int result = sum_to(n-1);
        add     $v0, $v0, $s0
sum_to_if_end:
        _
sum_to__epilog:
        pop     $s0
        pop     $ra
        jr      $ra
```

COMP1521 24T2: Computer Systems Fundamentals is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs1521@cse.unsw.edu.au