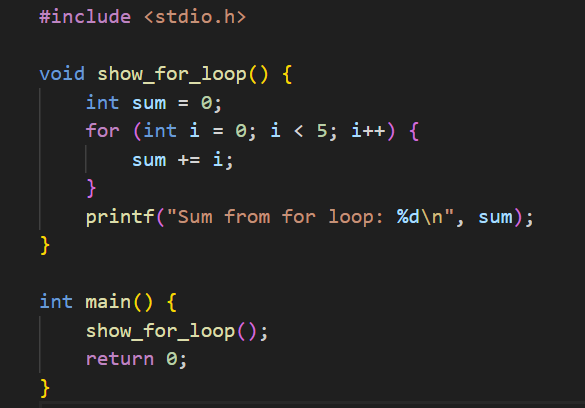
CPL ASSIGNMENT – 4

NAME: KAVYA NAMBURI

ENROLLMENT NO.: BT23CSE004

1. LANGUAGE FEATURE: FOR LOOP



* Without Optimization(-O0):

1. The for loop is preserved. The loop runs at runtime with cmp, jmp, add, and inc instructions.
2. Variables sum and i are stored in **stack memory** (using %ebp offsets).
3. Reserves **40 bytes** on the stack using subl $40, %esp.
4. Repeated additions: sum += i is performed at runtime inside a loop.
5. Long function with **20+ instructions**, including jumps, loads, adds.
6. Slower — does more work **at runtime**.
7. No constant folding — the loop is preserved even though its result is fixed.

SIMPLIFIED ASSEMBLY CODE(-O0):

show\_for\_loop:

push %ebp

mov %esp, %ebp

sub $40, %esp ; // Reserve 40 bytes stack space

mov $0, -12(%ebp) ;// int sum = 0

mov $0, -16(%ebp) ; //int i = 0

L2:

cmp $4, -16(%ebp) ; //if i <= 4

jg L4 ; // jump if i > 4

mov -16(%ebp), %eax

add %eax, -12(%ebp);// sum += i

add $1, -16(%ebp);// i++

jmp L2; //repeat loop

L4:

mov -12(%ebp), %eax

mov %eax, 4(%esp)

mov $LC0, (%esp)

call \_printf ; printf("Sum from for loop: %d\n", sum)

leave

ret

* With Optimization(-O2):

1. The compiler calculates the result (0 + 1 + 2 + 3 + 4 = 10) **during compilation**. The loop is removed entirely.
2. No memory is used for sum or i. The value 10 is **hardcoded directly into printf**.
3. Only **28 bytes** of stack space are used for the printf call. No extra space for loop variables.
4. No additions at all. The result is known, so it's just loaded as a constant.
5. Short function with only **a few instructions** — mostly preparing arguments for printf.
6. Faster — result is **precomputed**.
7. Constant folding is applied — the sum is evaluated at compile time.

SIMPLIFIED ASSEMEBLY CODE(-O2):

show\_for\_loop:

sub $28, %esp

mov $10, 4(%esp) ; //sum = 10

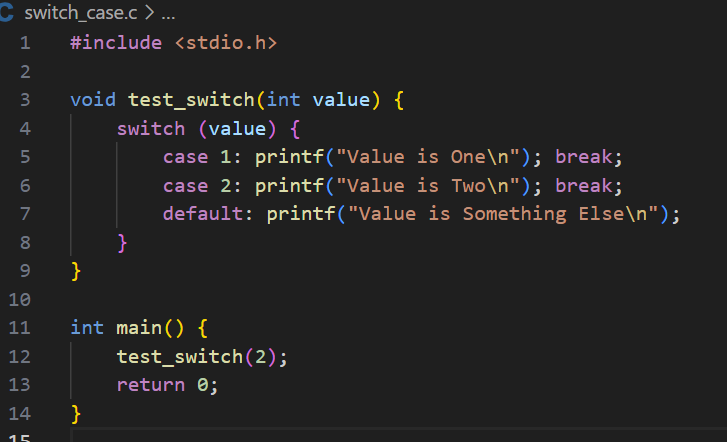
mov $LC0, (%esp)

call \_printf ; //printf("Sum from for loop: %d\n", 10)

add $28, %esp

ret

1. LANGUAGE FEATURE: SWITCH CASE



* Without Optimization(-O0):

1. A full stack frame is created for every function, even when not strictly needed.
2. Function arguments and local variables are accessed using the base pointer.
3. The switch logic is compiled as a sequence of comparisons and jumps.
4. After handling each case, the code explicitly jumps to the end to prevent fall-through.
5. The main function calls test\_switch(2) even though the value is constant.
6. All switch cases are preserved in the output even if some are never reached

SIMPLIFIED ASSEMBLY CODE(-O0):

\_test\_switch:

push %ebp

mov %esp, %ebp

sub $24, %esp ; //create stack space

mov 8(%ebp), %eax; // get function arg 'value'

cmp $1, %eax

je case\_1

cmp $2, %eax

je case\_2

jmp default

case\_1:

push $LC0; // "Value is One"

call \_puts

jmp end

case\_2:

push $LC1; //"Value is Two"

call \_puts

jmp end

default:

push $LC2; //"Value is Something Else"

call \_puts

end:

leave

ret

* With Optimization(-O2):

1. Stack frame setup is skipped for functions that don’t need it, improving performance.
2. Arguments are accessed directly from the stack without extra setup instructions.
3. The switch statement is optimized into a minimal number of comparisons and jumps.
4. The compiler aligns code blocks for faster branching using CPU-level optimizations.
5. Since the function is always called with the same value, the call is skipped, and only the result is used directly.
6. Unused code paths are optimized out wherever possible, reducing size and increasing speed.

SIMPLIFIED OPTIMIZED CODE(-O2):  
\_test\_switch:

mov 4(%esp), %eax ; //get argument

cmp $1, %eax

je case\_1

cmp $2, %eax

jne default ; //if not 2, go to default

case\_2:

mov $LC1, 4(%esp)

jmp \_puts

default:

mov $LC2, 4(%esp)

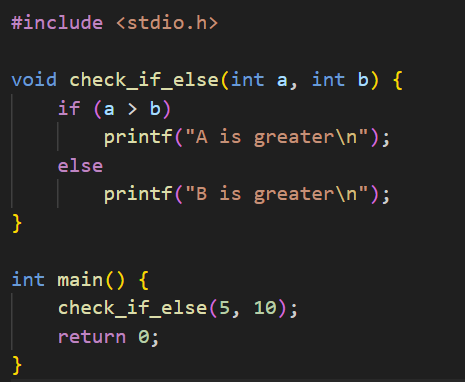
jmp \_puts

case\_1:

mov $LC0, 4(%esp)

jmp \_puts

3.LANGUAGE FEATURE: IF-ELSE STATEMENTS



* Without Optimization(-O0):

1. The compiler generates a full stack frame using the base pointer (%ebp) to store local variables and function arguments.
2. Function arguments are accessed via offsets from the base pointer, making the code easier to debug but less efficient.
3. The if-else condition is implemented using a basic comparison followed by conditional and unconditional jumps to different labels.
4. Both branches of the if-else statement are preserved in the assembly.
5. After evaluating the if or else, the function jumps to the end explicitly using a jmp to skip the other block.
6. In the main function, the check\_if\_else function is actually called with the values 10 and 5, even though the result is predictable.

SIMPLIFIED ASSEMBLY CODE(-O0):

check\_if\_else:

pushl %ebp

movl %esp, %ebp ; //Set up stack frame

subl $24, %esp ; // Allocate local space

movl 8(%ebp), %eax; //Load argument a into eax

cmpl 12(%ebp), %eax ; // Compare a with b

jle else\_branch ; Jump if a <= b

movl $LC0, (%esp) ; //Load "A is greater"

call \_puts

jmp end\_if

else\_branch:

movl $LC1, (%esp) ; // Load "B is greater"

call \_puts

end\_if:

leave

ret

* With Optimization(-O2):

1. The function avoids creating a full stack frame when it's not needed, leading to more efficient code.
2. Arguments are accessed directly via stack offsets without storing them in local variables.
3. The if-else logic is minimized to a more direct comparison and branch, reducing instruction count.
4. The function call in main is removed entirely — instead, the compiler directly inserts the result of the comparison and prints the appropriate string.
5. Only the necessary output for the condition 10 > 5 is retained; the other code path is completely optimized away.
6. Redundant instructions, such as saving/restoring the base pointer or unnecessary jumps, are removed.

SIMPLIFIED ASSEMBLY CODE(-02):

check\_if\_else:

movl 8(%esp), %eax; //Load b into eax

cmpl %eax, 4(%esp); // Compare a with b (a > b?)

jg print\_a ; // Jump if a > b

movl $LC1, 4(%esp) ; // Else: Load "B is greater"

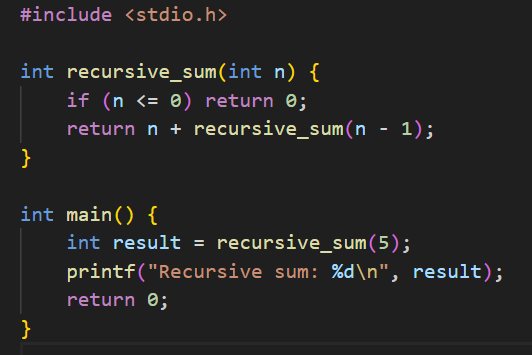
jmp \_puts

print\_a:

movl $LC0, 4(%esp) ; Load "A is greater"

jmp \_puts

4.LANGUAGE FEATURE: RECURSION



* Without Optimization(-O0):

1. The recursive function uses the base pointer and creates a full stack frame, storing arguments and local variables in memory.
2. Each recursive call is explicitly made using call \_recursive\_sum, and the result is combined after the function returns.
3. The logic strictly follows the original C code structure, including base-case check and recursive calculation with intermediate storage.
4. The main function calls the recursive function with the value 5 and stores the result in local memory before printing.
5. The return value is handled manually by moving it into registers and then into memory before the printf call

SIMPLIFIED ASSEMBLY CODE(-O0):  
recursive\_sum:

push %ebp

mov %esp, %ebp

sub $24, %esp

mov 8(%ebp), %eax; //Load n

cmp $0, %eax

jg L2 ; //If n > 0, jump to recursive part

mov $0, %eax ; //Base case return 0

jmp L3

L2:

mov 8(%ebp), %eax ; // Load n

sub $1, %eax

push %eax ; //Push n - 1 as argument

call recursive\_sum; // Recursive call

mov %eax, %edx ; // Store result

mov 8(%ebp), %eax ; // Load n again

add %edx, %eax ; // n + recursive\_sum(n-1)

L3:

leave

ret

* With Optimization(-O2):

1. The recursive function is converted into an iterative loop, eliminating actual recursion to improve performance and avoid stack overhead.
2. A loop is used to accumulate the sum of numbers from the input value down to 1, storing the result in %eax.
3. The compiler completely removes the stack frame setup and base pointer usage in the recursive function since it’s now a simple loop.
4. The main function skips the recursive call and directly passes the final result (which would be 15 for sum of 1 to 5) to printf.
5. This transformation is an example of **tail recursion optimization**, where the compiler recognizes the recursive pattern and replaces it with a loop

SIMPLIFIED OPTIMIZED CODE(-O2):  
recursive\_sum:

mov 4(%esp), %edx ; Load n

xor %eax, %eax ; Clear %eax (sum = 0)

test %edx, %edx

jle done ; if (n <= 0) jump to done

loop:

add %edx, %eax ; sum += n

sub $1, %edx ; n--

jne loop ; repeat until n == 0

done:

ret