Activity I – Instruction Set and Compiler

Exercise 1 - Instruction Analysis

This exercise will familiarize you with several aspects of the instruction set and the fundamentals of the compiler. Given max.c (below), please use `gcc -S max.c` to compile the code into assembly code. (The result will be in max.s.) From the result, answer the following questions.

```c
// max.c
int max1(int a, int b) {
    return (a > b) ? a : b;
}

int max2(int a, int b) {
    int isaGTB = a > b;
    int max;
    if (isaGTB) {
        max = a;
    } else {
        max = b;
    }
    return max;
}
```

```
// max.s
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0 sdk_version 14, 4
    .globl  _max1                           ; -- Begin function max1
    .p2align    2
_max1:                                  ; @max1
    .cfi_startproc
; %bb.0:
    sub sp, sp, #16
    .cfi_def_cfa_offset 16
    str w0, [sp, #12]
    str w1, [sp, #8]
    ldr w8, [sp, #12]
    ldr w9, [sp, #8]
    subs    w8, w8, w9
    cset    w8, le
    tbnz    w8, #0, LBB0_2
    b   LBB0_1
LBB0_1:
```

```
    ldr w8, [sp, #12]
    str w8, [sp, #4]                    ; 4-byte Folded Spill
    b   LBB0_3
LBB0_2:
    ldr w8, [sp, #8]
    str w8, [sp, #4]                    ; 4-byte Folded Spill
    b   LBB0_3
LBB0_3:
    ldr w0, [sp, #4]                    ; 4-byte Folded Reload
    add sp, sp, #16
    ret
    .cfi_endproc
                                        ; -- End function
    .globl  _max2                           ; -- Begin function max2
    .p2align    2
_max2:                                  ; @max2
    .cfi_startproc
; %bb.0:
    sub sp, sp, #16
    .cfi_def_cfa_offset 16
    str w0, [sp, #12]
    str w1, [sp, #8]
    ldr w8, [sp, #12]
    ldr w9, [sp, #8]
    subs    w8, w8, w9
    cset    w8, gt
    and w8, w8, #0x1
    str w8, [sp, #4]
    ldr w8, [sp, #4]
    subs    w8, w8, #0
    cset    w8, eq
    tbnz    w8, #0, LBB1_2
    b   LBB1_1
LBB1_1:
    ldr w8, [sp, #12]
    str w8, [sp]
    b   LBB1_3
LBB1_2:
    ldr w8, [sp, #8]
    str w8, [sp]
    b   LBB1_3
LBB1_3:
    ldr w0, [sp]
    add sp, sp, #16
    ret
    .cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

What does the code hint about the kind of instruction set? (e.g. Accumulator, Register Memory, Memory Memory, Register Register) Please justify your answer.

> **Answer**:
> Register-Register (Load-Store) instruction set architecture.
>
> - Register-Register architectures operate primarily by loading values from memory into registers, performing operations on those registers, and then storing the results back into memory.
> - In the assembly code, we can see multiple load (ldr) and store (str) instructions. For example, the variables w0, w1, etc., are being loaded from memory into registers and stored back into memory. This is a characteristic of the load-store architecture.

Can you tell whether the architecture is either Restricted Alignment or Unrestricted Alignment? Please explain how you come up with your answer.

> **Answer**:
> The architecture seems to be Restricted Alignment. This conclusion is drawn from the fact that the code aligns the stack pointer with a multiple of 16 bytes using sub sp, sp, #16 before any operations. Restricted Alignment architectures often require that memory accesses are aligned to specific byte boundaries (e.g., 16-byte alignment for 64-bit architectures). This alignment ensures that memory accesses are efficient and prevents potential faults.

Create a new function (e.g. testMax) to call max1. Generate new assembly code. What does the result suggest regarding the register saving (caller save vs. callee save)? Please provide your analysis.

```c
// max.c, testMax()
int testMax() {
    int result = max1(5, 10);
    return result;
}
```

```
// max.s
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0 sdk_version 14, 4
    .globl  _max1                           ; -- Begin function max1
    .p2align    2
_max1:                                  ; @max1
    .cfi_startproc
; %bb.0:
    sub sp, sp, #16
    .cfi_def_cfa_offset 16
    str w0, [sp, #12]
```

```
    str w1, [sp, #8]
    ldr w8, [sp, #12]
    ldr w9, [sp, #8]
    subs    w8, w8, w9
    cset    w8, le
    tbnz    w8, #0, LBB0_2
    b   LBB0_1
LBB0_1:
    ldr w8, [sp, #12]
    str w8, [sp, #4]                        ; 4-byte Folded Spill
    b   LBB0_3
LBB0_2:
    ldr w8, [sp, #8]
    str w8, [sp, #4]                        ; 4-byte Folded Spill
    b   LBB0_3
LBB0_3:
    ldr w0, [sp, #4]                        ; 4-byte Folded Reload
    add sp, sp, #16
    ret
    .cfi_endproc
                                        ; -- End function
    .globl  _max2                           ; -- Begin function max2
    .p2align    2
_max2:                                  ; @max2
    .cfi_startproc
; %bb.0:
    sub sp, sp, #16
    .cfi_def_cfa_offset 16
    str w0, [sp, #12]
    str w1, [sp, #8]
    ldr w8, [sp, #12]
    ldr w9, [sp, #8]
    subs    w8, w8, w9
    cset    w8, gt
    and w8, w8, #0x1
    str w8, [sp, #4]
    ldr w8, [sp, #4]
    subs    w8, w8, #0
    cset    w8, eq
    tbnz    w8, #0, LBB1_2
    b   LBB1_1
LBB1_1:
    ldr w8, [sp, #12]
    str w8, [sp]
    b   LBB1_3
LBB1_2:
    ldr w8, [sp, #8]
    str w8, [sp]
    b   LBB1_3
LBB1_3:
    ldr w0, [sp]
    add sp, sp, #16
    ret
```

```
    .cfi_endproc
                                        ; -- End function
    .globl  _testMax                        ; -- Begin function testMax
    .p2align    2
_testMax:                               ; @testMax
    .cfi_startproc
; %bb.0:
    sub sp, sp, #32
    .cfi_def_cfa_offset 32
    stp x29, x30, [sp, #16]             ; 16-byte Folded Spill
    add x29, sp, #16
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    mov w0, #5
    mov w1, #10
    bl  _max1
    stur    w0, [x29, #-4]
    ldur    w0, [x29, #-4]
    ldp x29, x30, [sp, #16]             ; 16-byte Folded Reload
    add sp, sp, #32
    ret
    .cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

> **Answer**:
> The analysis suggests that this system follows a caller-save convention for general-purpose registers (w0, w1, etc.). This means that the caller is responsible for saving any registers that it needs to preserve across function calls. However, special registers like x29 (frame pointer) and x30 (link register) follow a callee-save convention, meaning the callee is responsible for preserving these registers' values.
>
> - Caller-Save Convention Observed:
>   - In the testMax function, before calling max1, the stack is adjusted, and the registers x29 (frame pointer) and x30 (link register, used to store the return address) are stored on the stack using the stp instruction. This ensures that the current state of the function can be restored after max1 is called. The fact that these registers are saved by the caller (testMax) suggests a caller-save convention.
>   - The general-purpose registers w0 and w1 are used to pass arguments to max1, and max1 places the return value in w0. These registers are not explicitly saved and restored by the callee (max1), indicating that if testMax needed to preserve the values in these registers across the function call, it would be responsible for saving them. This confirms that w0 and w1 are caller-saved registers.

> - Callee-Save Convention for Specific Registers:
>   - The registers x29 (frame pointer) and x30 (link register) are saved on the stack at the beginning of testMax and restored before returning. These are examples of callee-saved registers, meaning that functions are expected to preserve their values across calls.

How do the arguments be passed and the return value returned from a function? Please explain the code.

> **Answer**:
>
> - Arguments Passing:
>
> ```
> mov    w0, #5        ; Move the value 5 into register w0 (first argument
> mov    w1, #10       ; Move the value 10 into register w1 (second argume
> bl     _max1         ; Branch to the function max1
> ```
>
> The first two arguments are passed via the w0 and w1 registers. If there were additional arguments beyond what can fit in registers, they would be passed via the stack.
>
> - Return Value:
>   i. Return Value in Register:
>
> ```
> ldr    w0, [sp, #4]  ; Load the result from the stack into w0
> ```
>
> The return value of the function is stored in the w0 register.
> ii. Handling the Return Value:
>
> ```
> stur   w0, [x29, #-4]  ; Store the result of max1 in memory
> ldur   w0, [x29, #-4]  ; Load the result back into w0 (to return from te
> ```
>
> The caller (testMax) retrieves the return value from w0 and can use or return it as necessary.

Find the part of code (snippet) that does comparison and conditional branch. Explain how it works.

> **Answer**:
>
> ```
> //_max1:
> subs    w8, w8, w9      ; Subtract w9 (b) from w8 (a), setting flags
> cset    w8, le          ; Set w8 to 1 if the result of the subtraction i
> ```

```
tbnz    w8, #0, LBB0_2  ; Test bit 0 of w8, branch to LBB0_2 if w8 is nc
b       LBB0_1          ; Otherwise, branch to LBB0_1
```

```
//_max2:
subs    w8, w8, w9      ; Subtract w9 (b) from w8 (a), setting flags
cset    w8, gt          ; Set w8 to 1 if a > b, otherwise set to 0
and     w8, w8, #0x1    ; Mask w8 to ensure it's either 0 or 1
str     w8, [sp, #4]    ; Store the result (isaGTB) in memory
ldr     w8, [sp, #4]    ; Load the result (isaGTB) back from memory
subs    w8, w8, #0      ; Compare isaGTB to 0
cset    w8, eq          ; Set w8 to 1 if isaGTB == 0 (i.e., a <= b)
tbnz    w8, #0, LBB1_2  ; Test bit 0 of w8, branch to LBB1_2 if w8 is nc
b       LBB1_1          ; Otherwise, branch to LBB1_1
```

If max.c is compiled with optimization turned on (using `gcc -O2 -S max.c`), what are the differences that you may observe from the result (as compared to that without optimization). Please provide your analysis

```
// max.s
// via gcc -O2 -S max.c
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0 sdk_version 14, 4
    .globl  _max1                           ; -- Begin function max1
    .p2align    2
_max1:                                  ; @max1
    .cfi_startproc
; %bb.0:
    cmp w0, w1
    csel    w0, w0, w1, gt
    ret
    .cfi_endproc
                                        ; -- End function
    .globl  _max2                           ; -- Begin function max2
    .p2align    2
_max2:                                  ; @max2
    .cfi_startproc
; %bb.0:
    cmp w0, w1
    csel    w0, w0, w1, gt
    ret
    .cfi_endproc
                                        ; -- End function
    .globl  _testMax                        ; -- Begin function testMax
    .p2align    2
_testMax:                               ; @testMax
    .cfi_startproc
; %bb.0:
    mov w0, #10
```

```
    ret
    .cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

> **Answer**:
>
> - Fewer Instructions: The optimized code is much more concise, reducing the instruction count.
>
> ```
> cmp   w0, w1                ; Compare w0 (a) with w1 (b)
> csel  w0, w0, w1, gt        ; Conditionally select w0 if a > b, otherwis
> ret                         ; Return the result in w0
> ```
>
> Instead of using conditional branches, the compiler optimizes the comparison and selection process with the cmp and csel instructions. The csel instruction conditionally selects between two values based on the result of the comparison (cmp). This avoids branching entirely, making the code faster and reducing potential branch mispredictions.
>
> - No Memory Accesses: All operations are performed in registers, eliminating unnecessary memory loads and stores.
> - No Branching: The optimized code uses conditional selection instead of branching, which can reduce latency and improve performance.
> - Inlining: Functions with constant arguments are evaluated at compile time, and their results are inlined into the code.c
>
> ```
> // _testMax:
> mov    w0, #10  ; Move the constant value 10 into w0
> ret             ; Return the result
> ```
>
> The optimized version of testMax no longer calls max1. Instead, it directly returns the constant result 10. This is an example of constant folding and inlining, where the compiler evaluates the result at compile-time since it knows the values being passed to max1 are constants (5 and 10). The function becomes a simple mov and ret sequence:

Please estimate the CPU Time required by the max1 function (using the equation CPI=ICxCPIxTc). If possible, create a main function to call max1 and use the time command to measure the performance. Compare the measure to your estimation. What do you think are the factors that cause the difference? Please provide your analysis. (You may find references online regarding the CPI of each instruction.)

Answer:

```
// _max1:
cmp   w0, w1        ; Compare w0 and w1 (1 instruction)
csel  w0, w0, w1, gt  ; Conditionally select between w0 and w1 (1 instru
ret                 ; Return the result (1 instruction)
```

- Total instruction count (IC) = 3 instructions
- Cycles per Instruction (CPI):
  - cmp (compare)= 1 cycle,
  - csel (conditional select)= 1 cycle,
  - ret (return)= 1 cycle
  - Average CPI = 1 cycle per instruction
- Clock Cycle Time (Tc):
  For an Apple M2 CPU, clock frequency = 3.49 GHz
  - Tc = 1 / (3.49 GHz) = 0.286 ns per cycle
- CPU Time
  = IC×CPI×Tc
  = 3 instructions × 1 cycle/instruction × 0.286×10-9 seconds/cycle
  = 0.858 ns

Estimated CPU time per call to max1() is approximately 0.858 nanoseconds.

Measure Actual CPU Time

```c
// max.c
#include <stdio.h>

int max1(int a, int b) {
    return (a > b) ? a : b;
}

int max2(int a, int b) {
    int isaGTB = a > b;
    int max;
    if (isaGTB) {
        max = a;
    } else {
        max = b;
    }
    return max;
}
```

```c
int testMax() {
    int result = max1(5, 10);
    return result;
}

int main() {
    int result = 0;
    for (int i = 0; i < 1000000000; i++) {  // Call max1 a billion(10^9) tim
        result = max1(5, 10);
    }
    printf("Result: %d\n", result);
    return 0;
}
```

```
// ! gcc -O2 -o max_time max.c
// ! time ./max_time
Result: 10
./max_time  0.00s user 0.00s system 1% cpu 0.475 total
```

Time per call = 0.475 s / 10^9 calls = 0.475 ns per call

The difference between the estimated and measured CPU time can be attributed to:

- CPU Optimizations: Pipelining, out-of-order execution, and branch prediction reduce the effective CPI.
- Efficient Execution: The optimized code benefits from advanced CPU features that minimize the time per instruction.
- Measurement Precision: The time command captures the total execution time including all overheads, but the measured time per call is still impressive due to efficient processor execution.

Exercise 2 - Optimization

We will use simple fibonacci calculation as a benchmark. Please measure the execution time (using thetime command) of this given program when compiling with optimization level 0 (no optimization), level 1, level 2 and level 3. (Note that some compilers do similar optimization for all level 1, level 2 and level 3. If that is the case, you will see no difference after level 1.) You may want to run each program a few times and use the average value as a result.

```c
// fibo.c
#include <stdio.h>

long fibo(long a)
```

```
{
    if (a <= 0L)
    {
        return 0L;
    }
    if (a == 1L)
    {
        return 1L;
    }
    return fibo(a - 1L) + fibo(a - 2L);
}
int main(int argc, char *argv[])
{
    for (long i = 1L; i < 45L; i++)
    {
        long f = fibo(i);
        printf("fibo of %ld is %ld\n", i, f);
    }
}
```

Answer:

```
# Average
O0 = 10.576 s
O1 = 5.977 s
O2 = 6.043 s
O3 = 5.920 s
# raw
./fibo_O0  10.54s user 0.07s system 95% cpu 11.146 total
./fibo_O0  10.26s user 0.04s system 99% cpu 10.304 total
./fibo_O0  10.24s user 0.03s system 99% cpu 10.278 total
./fibo_O1  5.67s user 0.02s system 93% cpu 6.087 total
./fibo_O1  5.70s user 0.02s system 99% cpu 5.753 total
./fibo_O1  5.89s user 0.06s system 97% cpu 6.072 total
./fibo_O2  5.85s user 0.02s system 93% cpu 6.266 total
./fibo_O2  5.86s user 0.02s system 99% cpu 5.912 total
./fibo_O2  5.88s user 0.02s system 99% cpu 5.951 total
./fibo_O3  5.75s user 0.03s system 94% cpu 6.129 total
./fibo_O3  5.67s user 0.02s system 99% cpu 5.695 total
./fibo_O3  5.81s user 0.04s system 98% cpu 5.935 total
```

Exercise 3 - Analysis

As suggested by the results in Exercise 2, what kinds of optimization are used by the compiler in each level in order to make the program faster? To answer this question, use `gcc -S` to generate the assembly code for each level (e.g. `gcc -S -O2 fibo.c`) and use this result as a basis for your analysis.(Depending on your version of the compiler, the result may vary.)

```
// O0
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0 sdk_version 14, 4
    .globl  _fibo                           ; -- Begin function fibo
    .p2align    2
_fibo:                                  ; @fibo
    .cfi_startproc
; %bb.0:
    sub sp, sp, #48
    .cfi_def_cfa_offset 48
    stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
    add x29, sp, #32
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    str x0, [sp, #16]
    ldr x8, [sp, #16]
    subs    x8, x8, #0
    cset    w8, gt
    tbnz    w8, #0, LBB0_2
    b   LBB0_1
LBB0_1:
    stur    xzr, [x29, #-8]
    b   LBB0_5
LBB0_2:
    ldr x8, [sp, #16]
    subs    x8, x8, #1
    cset    w8, ne
    tbnz    w8, #0, LBB0_4
    b   LBB0_3
LBB0_3:
    mov x8, #1
    stur    x8, [x29, #-8]
    b   LBB0_5
LBB0_4:
    ldr x8, [sp, #16]
    subs    x0, x8, #1
    bl  _fibo
    str x0, [sp, #8]                    ; 8-byte Folded Spill
    ldr x8, [sp, #16]
    subs    x0, x8, #2
    bl  _fibo
    mov x8, x0
    ldr x0, [sp, #8]                    ; 8-byte Folded Reload
    add x8, x0, x8
    stur    x8, [x29, #-8]
    b   LBB0_5
LBB0_5:
    ldur    x0, [x29, #-8]
    ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
    add sp, sp, #48
    ret
```

```
	.cfi_endproc
                                        ; -- End function
	.globl	_main                           ; -- Begin function main
	.p2align	2
_main:                                  ; @main
	.cfi_startproc
; %bb.0:
	sub sp, sp, #64
	.cfi_def_cfa_offset 64
	stp x29, x30, [sp, #48]             ; 16-byte Folded Spill
	add x29, sp, #48
	.cfi_def_cfa w29, 16
	.cfi_offset w30, -8
	.cfi_offset w29, -16
	stur	wzr, [x29, #-4]
	stur	w0, [x29, #-8]
	stur	x1, [x29, #-16]
	mov x8, #1
	str x8, [sp, #24]
	b	LBB1_1
LBB1_1:                                 ; =>This Inner Loop Header: Depth=1
	ldr x8, [sp, #24]
	subs	x8, x8, #45
	cset	w8, ge
	tbnz	w8, #0, LBB1_4
	b	LBB1_2
LBB1_2:                                 ;   in Loop: Header=BB1_1 Depth=1
	ldr x0, [sp, #24]
	bl	_fibo
	str x0, [sp, #16]
	ldr x10, [sp, #24]
	ldr x8, [sp, #16]
	mov x9, sp
	str x10, [x9]
	str x8, [x9, #8]
	adrp	x0, l_.str@PAGE
	add x0, x0, l_.str@PAGEOFF
	bl	_printf
	b	LBB1_3
LBB1_3:                                 ;   in Loop: Header=BB1_1 Depth=1
	ldr x8, [sp, #24]
	add x8, x8, #1
	str x8, [sp, #24]
	b	LBB1_1
LBB1_4:
	ldur	w0, [x29, #-4]
	ldp x29, x30, [sp, #48]             ; 16-byte Folded Reload
	add sp, sp, #64
	ret
	.cfi_endproc
                                        ; -- End function
	.section	__TEXT,__cstring,cstring_literals
l_.str:                                 ; @.str
```

```
    .asciz  "fibo of %ld is %ld\n"

.subsections_via_symbols
```

```
// 01
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0 sdk_version 14, 4
    .globl  _fibo                           ; -- Begin function fibo
    .p2align    2
_fibo:                                  ; @fibo
    .cfi_startproc
; %bb.0:
    stp x22, x21, [sp, #-48]!           ; 16-byte Folded Spill
    .cfi_def_cfa_offset 48
    stp x20, x19, [sp, #16]             ; 16-byte Folded Spill
    stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
    add x29, sp, #32
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    .cfi_offset w19, -24
    .cfi_offset w20, -32
    .cfi_offset w21, -40
    .cfi_offset w22, -48
    mov x19, x0
    mov x20, #0
    mov w21, #1
    subs    x0, x19, #1
    b.lt    LBB0_3
LBB0_1:                                 ; =>This Inner Loop Header: Depth=1
    b.eq    LBB0_4
; %bb.2:                                ;   in Loop: Header=BB0_1 Depth=1
    bl  _fibo
    add x20, x20, x0
    sub x19, x19, #2
    subs    x0, x19, #1
    b.ge    LBB0_1
LBB0_3:
    mov x21, #0
LBB0_4:
    add x0, x20, x21
    ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
    ldp x20, x19, [sp, #16]             ; 16-byte Folded Reload
    ldp x22, x21, [sp], #48             ; 16-byte Folded Reload
    ret
    .cfi_endproc
                                        ; -- End function
    .globl  _main                           ; -- Begin function main
    .p2align    2
_main:                                  ; @main
    .cfi_startproc
```

```
; %bb.0:
    sub sp, sp, #48
    .cfi_def_cfa_offset 48
    stp x20, x19, [sp, #16]             ; 16-byte Folded Spill
    stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
    add x29, sp, #32
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    .cfi_offset w19, -24
    .cfi_offset w20, -32
    mov w19, #1
Lloh0:
    adrp    x20, l_.str@PAGE
Lloh1:
    add x20, x20, l_.str@PAGEOFF
LBB1_1:                                 ; =>This Inner Loop Header: Depth=1
    mov x0, x19
    bl  _fibo
    stp x19, x0, [sp]
    mov x0, x20
    bl  _printf
    add x19, x19, #1
    cmp x19, #45
    b.ne    LBB1_1
; %bb.2:
    mov w0, #0
    ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
    ldp x20, x19, [sp, #16]             ; 16-byte Folded Reload
    add sp, sp, #48
    ret
    .loh AdrpAdd    Lloh0, Lloh1
    .cfi_endproc
                                        ; -- End function
    .section    __TEXT,__cstring,cstring_literals
l_.str:                                 ; @.str
    .asciz  "fibo of %ld is %ld\n"

.subsections_via_symbols
```

```
// O2
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0 sdk_version 14, 4
    .globl  _fibo                           ; -- Begin function fibo
    .p2align    2
_fibo:                                  ; @fibo
    .cfi_startproc
; %bb.0:
    stp x22, x21, [sp, #-48]!           ; 16-byte Folded Spill
    .cfi_def_cfa_offset 48
```

```
    stp x20, x19, [sp, #16]             ; 16-byte Folded Spill
    stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
    add x29, sp, #32
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    .cfi_offset w19, -24
    .cfi_offset w20, -32
    .cfi_offset w21, -40
    .cfi_offset w22, -48
    subs    x22, x0, #1
    b.lt    LBB0_6
; %bb.1:
    mov x19, x0
    mov x20, #0
    mov w21, #1
LBB0_2:                                 ; =>This Inner Loop Header: Depth=1
    cbz x22, LBB0_5
; %bb.3:                                ;   in Loop: Header=BB0_2 Depth=1
    sub x0, x19, #1
    bl  _fibo
    sub x22, x22, #2
    add x20, x0, x20
    subs    x19, x19, #2
    b.hi    LBB0_2
; %bb.4:
    mov x21, #0
LBB0_5:
    add x0, x21, x20
    b   LBB0_7
LBB0_6:
    mov x0, #0
LBB0_7:
    ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
    ldp x20, x19, [sp, #16]             ; 16-byte Folded Reload
    ldp x22, x21, [sp], #48             ; 16-byte Folded Reload
    ret
    .cfi_endproc
                                        ; -- End function
    .globl  _main                           ; -- Begin function main
    .p2align    2
_main:                                  ; @main
    .cfi_startproc
; %bb.0:
    sub sp, sp, #48
    .cfi_def_cfa_offset 48
    stp x20, x19, [sp, #16]             ; 16-byte Folded Spill
    stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
    add x29, sp, #32
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    .cfi_offset w19, -24
```

```
    .cfi_offset w20, -32
    mov w19, #1
Lloh0:
    adrp    x20, l_.str@PAGE
Lloh1:
    add x20, x20, l_.str@PAGEOFF
LBB1_1:                                 ; =>This Inner Loop Header: Depth=1
    mov x0, x19
    bl  _fibo
    stp x19, x0, [sp]
    mov x0, x20
    bl  _printf
    add x19, x19, #1
    cmp x19, #45
    b.ne    LBB1_1
; %bb.2:
    mov w0, #0
    ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
    ldp x20, x19, [sp, #16]             ; 16-byte Folded Reload
    add sp, sp, #48
    ret
    .loh AdrpAdd    Lloh0, Lloh1
    .cfi_endproc
                                        ; -- End function
    .section    __TEXT,__cstring,cstring_literals
l_.str:                                 ; @.str
    .asciz  "fibo of %ld is %ld\n"

.subsections_via_symbols
```

```
// O3
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0 sdk_version 14, 4
    .globl  _fibo                           ; -- Begin function fibo
    .p2align    2
_fibo:                                  ; @fibo
    .cfi_startproc
; %bb.0:
    stp x22, x21, [sp, #-48]!           ; 16-byte Folded Spill
    .cfi_def_cfa_offset 48
    stp x20, x19, [sp, #16]             ; 16-byte Folded Spill
    stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
    add x29, sp, #32
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    .cfi_offset w19, -24
    .cfi_offset w20, -32
    .cfi_offset w21, -40
    .cfi_offset w22, -48
    subs    x22, x0, #1
```

```
    b.lt    LBB0_6
; %bb.1:
    mov x19, x0
    mov x20, #0
    mov w21, #1
LBB0_2:                                 ; =>This Inner Loop Header: Depth=1
    cbz x22, LBB0_5
; %bb.3:                                ;   in Loop: Header=BB0_2 Depth=1
    sub x0, x19, #1
    bl  _fibo
    sub x22, x22, #2
    add x20, x0, x20
    subs    x19, x19, #2
    b.hi    LBB0_2
; %bb.4:
    mov x21, #0
LBB0_5:
    add x0, x21, x20
    ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
    ldp x20, x19, [sp, #16]             ; 16-byte Folded Reload
    ldp x22, x21, [sp], #48             ; 16-byte Folded Reload
    ret
LBB0_6:
    mov x0, #0
    ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
    ldp x20, x19, [sp, #16]             ; 16-byte Folded Reload
    ldp x22, x21, [sp], #48             ; 16-byte Folded Reload
    ret
    .cfi_endproc
                                        ; -- End function
    .globl  _main                           ; -- Begin function main
    .p2align    2
_main:                                  ; @main
    .cfi_startproc
; %bb.0:
    sub sp, sp, #48
    .cfi_def_cfa_offset 48
    stp x20, x19, [sp, #16]             ; 16-byte Folded Spill
    stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
    add x29, sp, #32
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    .cfi_offset w19, -24
    .cfi_offset w20, -32
    mov w19, #1
    mov w0, #1
    bl  _fibo
    stp x19, x0, [sp]
Lloh0:
    adrp    x19, l_.str@PAGE
Lloh1:
    add x19, x19, l_.str@PAGEOFF
```

```
	mov	x0, x19
	bl	_printf
	mov	w20, #2
	mov	w0, #2
	bl	_fibo
	stp	x20, x0, [sp]
	mov	x0, x19
	bl	_printf
	mov	w20, #3
	mov	w0, #3
	bl	_fibo
	stp	x20, x0, [sp]
	mov	x0, x19
	bl	_printf
	mov	w20, #4
	mov	w0, #4
	bl	_fibo
	stp	x20, x0, [sp]
	mov	x0, x19
	bl	_printf
	mov	w20, #5
	mov	w0, #5
	bl	_fibo
	stp	x20, x0, [sp]
	mov	x0, x19
	bl	_printf
	mov	w20, #6
	mov	w0, #6
	bl	_fibo
	stp	x20, x0, [sp]
	mov	x0, x19
	bl	_printf
	mov	w20, #7
	mov	w0, #7
	bl	_fibo
	stp	x20, x0, [sp]
	mov	x0, x19
	bl	_printf
	mov	w20, #8
	mov	w0, #8
	bl	_fibo
	stp	x20, x0, [sp]
	mov	x0, x19
	bl	_printf
	mov	w20, #9
	mov	w0, #9
	bl	_fibo
	stp	x20, x0, [sp]
	mov	x0, x19
	bl	_printf
	mov	w20, #10
	mov	w0, #10
	bl	_fibo
```

```
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #11
mov w0, #11
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #12
mov w0, #12
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #13
mov w0, #13
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #14
mov w0, #14
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #15
mov w0, #15
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #16
mov w0, #16
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #17
mov w0, #17
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #18
mov w0, #18
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #19
mov w0, #19
```

```
	bl  _fibo
	stp x20, x0, [sp]
	mov x0, x19
	bl  _printf
	mov w20, #20
	mov w0, #20
	bl  _fibo
	stp x20, x0, [sp]
	mov x0, x19
	bl  _printf
	mov w20, #21
	mov w0, #21
	bl  _fibo
	stp x20, x0, [sp]
	mov x0, x19
	bl  _printf
	mov w20, #22
	mov w0, #22
	bl  _fibo
	stp x20, x0, [sp]
	mov x0, x19
	bl  _printf
	mov w20, #23
	mov w0, #23
	bl  _fibo
	stp x20, x0, [sp]
	mov x0, x19
	bl  _printf
	mov w20, #24
	mov w0, #24
	bl  _fibo
	stp x20, x0, [sp]
	mov x0, x19
	bl  _printf
	mov w20, #25
	mov w0, #25
	bl  _fibo
	stp x20, x0, [sp]
	mov x0, x19
	bl  _printf
	mov w20, #26
	mov w0, #26
	bl  _fibo
	stp x20, x0, [sp]
	mov x0, x19
	bl  _printf
	mov w20, #27
	mov w0, #27
	bl  _fibo
	stp x20, x0, [sp]
	mov x0, x19
	bl  _printf
	mov w20, #28
```

```
mov w0, #28
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #29
mov w0, #29
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #30
mov w0, #30
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #31
mov w0, #31
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #32
mov w0, #32
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #33
mov w0, #33
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #34
mov w0, #34
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #35
mov w0, #35
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
mov w20, #36
mov w0, #36
bl  _fibo
stp x20, x0, [sp]
mov x0, x19
bl  _printf
```

```
    mov w20, #37
    mov w0, #37
    bl  _fibo
    stp x20, x0, [sp]
    mov x0, x19
    bl  _printf
    mov w20, #38
    mov w0, #38
    bl  _fibo
    stp x20, x0, [sp]
    mov x0, x19
    bl  _printf
    mov w20, #39
    mov w0, #39
    bl  _fibo
    stp x20, x0, [sp]
    mov x0, x19
    bl  _printf
    mov w20, #40
    mov w0, #40
    bl  _fibo
    stp x20, x0, [sp]
    mov x0, x19
    bl  _printf
    mov w20, #41
    mov w0, #41
    bl  _fibo
    stp x20, x0, [sp]
    mov x0, x19
    bl  _printf
    mov w20, #42
    mov w0, #42
    bl  _fibo
    stp x20, x0, [sp]
    mov x0, x19
    bl  _printf
    mov w20, #43
    mov w0, #43
    bl  _fibo
    stp x20, x0, [sp]
    mov x0, x19
    bl  _printf
    mov w20, #44
    mov w0, #44
    bl  _fibo
    stp x20, x0, [sp]
    mov x0, x19
    bl  _printf
    mov w0, #0
    ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
    ldp x20, x19, [sp, #16]             ; 16-byte Folded Reload
    add sp, sp, #48
    ret
```

```
    .loh AdrpAdd    Lloh0, Lloh1
    .cfi_endproc
                                        ; -- End function
    .section    __TEXT,__cstring,cstring_literals
l_.str:                                 ; @.str
    .asciz  "fibo of %ld is %ld\n"

.subsections_via_symbols
```

Answer:

- O0 (No Optimization): The assembly code is verbose, with lots of stack operations, redundant memory loads, and stores. No optimizations are applied, resulting in a straightforward translation of C code into assembly, prioritizing ease of debugging over performance.
- O1 (Basic Optimization): Basic optimizations are introduced, such as eliminating some redundant instructions and using registers more efficiently. Loop optimizations and function inlining start to appear, which reduces the number of instructions executed and improves performance.
- O2 (Further Optimization): More aggressive optimizations are applied, including loop unrolling and more efficient use of registers to minimize memory access. Dead code elimination and branch optimizations are performed to further reduce execution time.
- O3 (Maximum Optimization): At this level, the compiler performs the most aggressive optimizations, including function inlining and advanced loop transformations. The assembly code is highly optimized for speed, with minimal memory operations and efficient use of processor resources.