

Problem 6: Cache Performance (15 points) (L19)

It's 1987 and Apple has just released the Macintosh II featuring a Motorola 68030 processor. The 68030 contains a state of the art **128 byte** data-cache and uses **byte-addressable** memory. Having taken 370, you know there's more to cache design than just size, so you write a C program to help you determine the block size, number of blocks per set, and number of sets in the cache. You have a function `ACCESS(char *)` that loads the memory at the given pointer. Assume all variables other than the `data[]` array are stored in registers and that the cache has been cleared before each call to `miss_rate(...)`. Assume `data[0]` is at the start of the cache block.

```
#define CACHE_SIZE XX
double miss_rate(int stride) {
    char data[CACHE_SIZE * 2]; // note array is 2*cache size

    for (int i = 0; i < stride; i++){
        for (int j = i; j < (CACHE_SIZE * 2); j += stride) {
            ACCESS(data + j)
        }
    }
}
```

1. In order to test your code, you first run it on a **256 B fully-associative cache** with **32 B blocks**. You set the `CACHE_SIZE` macro to 256. Fill in the miss rate for each stride in the following table. Only answers in the table will be graded. Fractions are OK. **[6]**

Stride	Miss rate
1	1/32
2	1/16
4	1/8
8	1/4
16	1/2
32	1
64	1/2
128	1/4

You now run your program on the Motorola 68030 processor with a **128 byte** data-cache. You set the `CACHE_SIZE` macro to 128 and collect the following data

Stride	Miss rate
1	1/16
2	1/8
4	1/4
8	1/2
16	1
32	1
64	1/16

Using the above data answer the following questions and justify your answers.

2. What is the block size in bytes? **[3]**

At stride 16, the miss rate is 1 and at stride 1, the miss rate is 1/16, meaning the block size must be 16.

3. What is the associativity of the cache? **[3]**

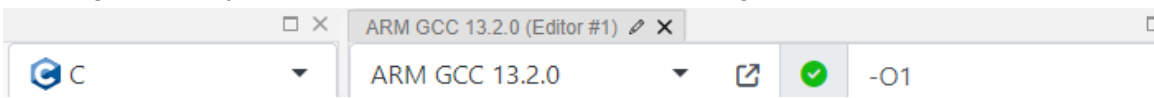
This is a 2 way associative cache.

4. How many sets are there in the cache? **[3]**

$128/16 = 8$ sets

Problem 7: Looking at a real compiler with a slightly different ISA (Group, 15 points) (L6)

Go to <https://godbolt.org/>. Select the ARM GCC 13.2.0 compiler, the C programming language, and set the compiler options to “-O1” (turning on the optimizer). Note, this is a 32-bit version of ARM and among other differences, the registers are listed with an r rather than an X (so r1 rather than X1). Enter the following code and then answer questions in parts a through d. If you’ve set things correctly, the website bar should look something like this:



```
#include<stdio.h>
int fib(int n)
{
    int a,b;
    if (n <= 1)
        return n;
    a=fib(n-2);
    b=fib(n-1);
    return(a+b);
}

int main ()
{
    int n = 7;
    printf("%d",fib(n));
    return 0;
}
```

You might find

<https://developer.arm.com/documentation/ddi0597/2023-09/Base-Instructions?lang=en> useful as a reference. Also, be sure to notice just how useful the colorization is.

- 1) Copy the assembly code for the function “fib”. Comment each line of assembly for that function explaining what it does. You are going to have to look up some of the instructions online. We are not looking for things that just restate what the assembly instruction does (such as “stores register 2” or “copies register 3 to memory”) but instead explains it in context (“puts the argument onto the stack” or “calls the function fib”). [7]

```
fib(int):
    push    {r3, r4, r5, lr} //Saves reg 3,4,5 and link reg to stack
    mov     r4, r0 //Moves/copies value of reg 0 into reg 4 - n
    cmp     r0, #1 //Compares the value of reg 0 (n) with 1
    ble     .L1 //Branches to .L1 label if value of reg 0 is less
                    than or equal to 1 (base case)
```

```

    subs    r0, r4, #2 //Subtracts 2 from value of reg 4 and saves
answer into reg 0 → represent (n-2) in preparation for fib(n-2) call
    bl      fib(int) //Branch and link to the fib function recursively
to calculate fib(n-2)
    mov     r5, r0 //Stores the result of fib(n-2) in reg 5
    subs    r0, r4, #1 //Subtracts 1 from value of reg 4 and saves
answer into reg 0 → represents (n-1) in preparation for fib(n-1) call
    bl      fib(int) //Branch and link to the fib function recursively
to calculate fib(n-1)
    add     r0, r0, r5 //Add the results of fib(n-1) in reg 0 with
results of fib(n-2) in reg 5 and stores in reg 0
.L1:
    pop     {r3, r4, r5, pc} //Restores the previously saved registers
reg3, reg4, reg5, and the pc from the stack.

```

- 2) What will be the maximum stack depth of this program (in bytes)? Include main and briefly justify your answer. [2]

For fib(int):

Each call to fib(int) pushes {r3, r4, r5, lr} onto the stack.\

Each push adds 16 bytes to the stack (4*4)

The recursive depth for fib(7) is 7 levels, so $7 \times 16 = 112$ bytes.

For main:

main pushes {r3, lr} onto the stack.

Each push adds 8 bytes to the stack

$112 \times 8 = 120$ bytes

- 3) Explain how arguments are being passed, where the return value is being placed, and how caller/callee save issues are being resolved. [2]

The argument n, which holds the number 7, is being passed by register 0, which also holds the return value. Caller/callee issues are resolved by the stack, as at the beginning of the fib function, registers 3,4,5 as well as the link register (which has the pc to return to one fib is done) are saved on the stack. At the end of the fib function, the registers 3,4, and 5 and the pc to return to are popped off the stack, ensuring that no register was accidentally overwritten and we know how to return to the main function.

- 4) Write a short C program which computes combinations recursively¹ and give it to the same compiler. Briefly provide the C code, the assembly code, and explain how arguments are passed. [4]

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

```
int combinations(int n, int k) {  
    if (k == 0 || k == n) {  
        return 1;    // Base case  
    }  
    return combinations(n - 1, k - 1) + combinations(n - 1, k);  
}
```

```
int main() {  
    int n = 5;  
    int k = 10;  
    int combination = combinations(n,k);  
    return 0;  
}
```

```
combinations(int, int):  
    cmp     r1, r0  
    it      ne  
    cmpne   r1, #0  
    bne     .L8  
    movs    r0, #1  
    bx      lr  
.L8:  
    push    {r4, r5, r6, lr}  
    mov     r6, r1  
    subs    r4, r0, #1  
    subs    r1, r1, #1  
    mov     r0, r4  
    bl      combinations(int, int)  
    mov     r5, r0  
    mov     r1, r6  
    mov     r0, r4  
    bl      combinations(int, int)
```

¹ The recursive definition to calculate n choose k can be defined as follows (for n, k non-negative integers):

$C(n, k) = 1$ If $k = 0$ or $n = k$;

Else $C(n, k) = C(n-1, k) + C(n-1, k-1)$

```
        add    r0, r0, r5
        pop    {r4, r5, r6, pc}
main:
        push   {r3, lr}
        movs   r1, #10
        movs   r0, #5
        bl     combinations(int, int)
        movs   r0, #0
        pop    {r3, pc}
```

In the assembly code, we can see how arguments are passed by register 0 which holds n, and register 1, which holds k. In the actual combinations function, the arguments are based back into combinations recursively, much like the fib function.