Project 1: Fibonacci Series

Fall 2019 - EE 466 Computer Architecture

Lewis Collum (0621539), colluml@clarkson.edu, EE/CE

October 29, 2019

Contents

Abstract

We will be implementing a simple iterative Fibonacci algorithm in ARM assembly. While doing so, we will explore the aarch64 ARM toolchains and QEMU emulation of an bare-metal ARM device.

1 Design

1.1 Fibonacci Sequence Basics

For any given element, f_n, in a series of fibonacci numbers,

$$f_n = f_{n-1} + f_{n-2}.$$

Also,

$$f_0 = 0, \text{ and } f_1 = 1.$$

Overall, a given element in series, except the first two elements of the series, is equal to the sum of the previous two elements.

1.2 Software Environment

We will be using the ARM64 GNU cross compiler toolchain. Our project source code will be maintained with a simple GNU makefile. Additionally, our makefile has a rule for QEMU simulation. That is, when we run `make qemu`, it will build the project, emulate an ARM64 device, and run the program.

Since the Makefile is not vital to understanding this exercise, it will not be explained in this report, but is included in the appendix (section 6.3).

1.3 Interface Between C and Assembly

We want to build a fibonacci series procedure in assembly, and call the procedure from C. Our fibonacci procedure will take in the size of the series to be calculated and a pointer to the beginning of a series array. In turn, the procedure will modify the array, and we will see those modifications in C, by printing the array.

1.4 Fibonacci Design

While there are many ways to design a Fibonacci sequence algorithm, we are using a simple iterative approach and storing each calculated element into an array of defined size. The algorithm can be seen in the flowchart (section 2).

2 Flow Chart

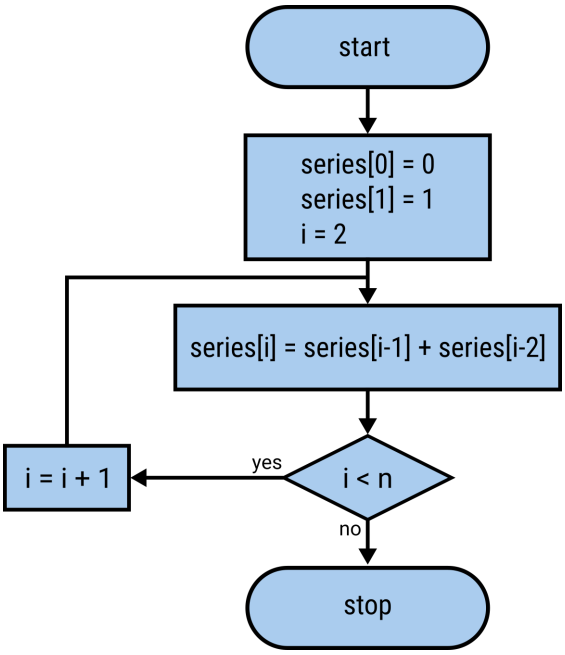

Figure 1: Flowchart depicting fibonacci series algorithm, where fibonacci numbers are stored in an array of size `n`. `i` is used as a counter variable, to exit the loop when the `series` array is full. `n` is the size of the `series` array.

3 Program

3.1 Fibonacci in Assembly

At the level of assembly, we are working with registers, as opposed to named variables. To improve readability of our assembly, we will start by assigning aliases to the registers we are going to use.

ARMv8 uses registers x0-x7 for procedure parameters. We know from our flowchart, that we need the size, `n`, of the series we want to calculate, and the pointer to the location of the `series` array. These parameters will be accessible from registers `x0` and `x1`, respectively.

We will use three other temporary registers — `x11`, `x12`, `x13` — for the current fibonacci number

being calculated, the index of the number being calculated (`count`), and the fibonacci number at position `count-2` in the `series` array.

So we have at the beginning of our `fibonacci.S` file,

```
#define seriesSize x0
#define seriesPointer x1

#define fibonacciNumber x11
#define count x12
#define backTwo x13
```

Since our procedure is to be used externally, in C, we declare the procedure symbol name as global with the `global` directive.

```
    .global fibonacci
fibonacci:
    ;; procedure body here
```

A fibonacci number in a series is calculated by $f_n = f_{n-1} + f_{n-2}$. In order to begin calculating a series of fibonacci numbers, f_0 and f_1 must be defined so that $f_2 = f_1 + f_0$ is valid. f_0 is defined as 0, and f_1 is defined as 1. In assembly, we must define f_0 and f_1 in the `series` array. We start by setting the `fibonacciNumber` register to 0 (for f_0). Then we store the value of the `fibonacciNumber` register in the address pointed to by the `seriesPointer` register. Next, we increment the `seriesPointer` register by a doubleword (`#8`). At this point, we have the value, 0, at index 0 in the `series` array. We can do the same actions to store the value, 1, at index 1 in the `series` array. Finally, we need to set the `count` register to the value, 2, representing index 2 of the `series` array. The `count` register is used to ensure we do not exceed the size of the `series` array, while looping.

```
    .global fibonacci
fibonacci:
    mov fibonacciNumber, 0
    stur fibonacciNumber, [seriesPointer, #0]
    add seriesPointer, seriesPointer, #8

    mov fibonacciNumber, 1
    stur fibonacciNumber, [seriesPointer, #0]
    add seriesPointer, seriesPointer, #8

    mov count, 2
```

The loop inside of our fibonacci procedure is where the rest of our series is calculated. The structure of the loop is as follows:

```
    mov count, 2
loop:

    ;; algorithm code here

    cmp count, seriesSize
    add count, count, #1
    ble loop
```

This is analogous to a `for` loop in C:

```
for (int count = 2; count < seriesSize; ++count) {
  // algorithm code here
}
```

The algorithm code that goes within the loop stores a newly calculated fibonacci number into the `series` array. To elaborate, fibonacci numbers are calculated as, $f_n = f_{n-1} + f_{n-2}$. To obtain f_{n-2}, we load the fibonacci element from the memory address pointed to by `seriesPointer` minus two doublewords into the register, `backTwo`. As an aside, "minus two doublewords" is expressed in ARMv8 as `#-16`. To continue, for each loop iteration in the assembly code, `fibonacciNumber = fibonacciNumber + backTwo`. The result, `fibonacciNumber`, is stored at the memory address pointed to by the `seriesPointer` register. Finally, the `seriesPointer` is incremented to the next doubleword, which represents the next element in the series. The full loop code is as follows:

```
loop:
    ldur backTwo, [seriesPointer, #-16]
    add fibonacciNumber, fibonacciNumber, backTwo
    stur fibonacciNumber, [seriesPointer, #0]
    add seriesPointer, seriesPointer, #8

    cmp count, seriesSize
    add count, count, #1
    blt loop

    br lr
```

Once `count` exceeds `seriesSize`, the looping is finished, and the procedure branches back to the main program flow. This is accomplished with the instruction `br lr`.

The `fibonacci.S` code can been seen in the appendix (section 6.2).

3.2 Interface in C

We have our Fibonacci procedure in assembly, now we will use C to create create an interface for the procedure. Namely, we need a way to **print** the results of the procedure. The following code is contained within a new `main.c` file.

Ultimately, the `main` function within the `main.c` file contains:

```
void main() {
  int n = 20;
  unsigned long long series[n];
  fibonacciBounded(n, series);
  printArrayWithSize(series, n);
  while(1);
}
```

In `main`, we are printing the series array, which contains `n` amount of fibonacci numbers, provided by our `fibonacciBounded` function. We will go through how we called the fibonacci procedure we wrote in assembly, in C, and how we added basic bounds checking for the function. Once we've done so, we will have a `fibonacciBounded` function as seen in the `main` above.

We begin by declaring the Fibonacci function.

```
extern void fibonacci(int n, unsigned long long* series);
```

The `extern` keyword indicates that the function definition may exist in a separate object file. Note that the `extern` keyword is not necessary, since functions are `extern` by default in C; but, including the keyword makes it explicit to the reader of the code that the definition of this function exists elsewhere.

The `fibonacci` function takes two parameters, the size `n`, of the series that will be calculated, and an array pointer, to the series array. The `series` array is what will contain our fibonacci numbers, which we want to display to the user.

Since we did not do bounds checking (for `n < 1`) in the Fibonacci assembly procedure, we will create a wrapper function, named `fibonacciBounded` that will do a bounds check (disregarding the upper-bound).

We will forward-declare this function and make it static (so that it is only visible in the scope of this file). Note that it has the same parameters as the `fibonacci` function.

```
static void fibonacciBounded(int n, unsigned long long* series);
```

The definition of `fibonacciBounded` checks if `n < 1` and, if true, prints an error message, otherwise calls the `fibonacci` function. While this method does not actually handle the error, it suffices for this exercise.

```
void fibonacciBounded(int n, unsigned long long* series) {
  n < 1? print("ERROR: n must be larger than 0."): fibonacci(n, series);
}
```

The rest of the code is available in the appendix (section 6.1).

4 Result

4.1 Case `n = 0`:

```
~/course/computer_architecture/p1_fibonacciSeries/release $ make qemu
n = 0:
ERROR: n must be larger than 0.
```

4.2 Case `n = -1`:

```
~/course/computer_architecture/p1_fibonacciSeries/release $ make qemu
n = -1:
ERROR: n must be larger than 0.
```

4.3 Case `n = 1`:

```
~/course/computer_architecture/p1_fibonacciSeries/release $ make qemu
n = 1:
0
```

4.4 Case `n = 2`:

```
~/course/computer_architecture/p1_fibonacciSeries/release $ make qemu
n = 2:
0, 1
```

4.5 Case `n = 5`:

```
~/course/computer_architecture/p1_fibonacciSeries/release $ make qemu
n = 5:
0, 1, 1, 2, 3
```

4.6 Case `n = 10`:

```
~/course/computer_architecture/p1_fibonacciSeries/release $ make qemu
n = 10:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

5 Self-Evaluation

One technical difficulty we had was emulating ARM64 hardware. QEMU is often used for doing this. It was not designed to emulate bare-metal though. We choose to emulate an ARM cortex-a53 processor, since it uses the A64 instruction set for an ARMv8 architecture. In order to get the device to "boot", we had to provide a simple linker script and startup script.

Linker script (`kernel.ld`)

```
OUTPUT_FORMAT("elf64-littleaarch64")
OUTPUT_ARCH(aarch64)
TARGET(binary)

STACKTOP = 0x51000000;

SECTIONS
{
        . = 0x40010000;
        .text : { *(.text) }
        .data : { *(.data) }
        .bss : { *(.bss) }

        . = STACKTOP;
        stacktop = .;
}
```

Startup script (`startup.S`)

```
    .text
    .globl _start
_start:
        ;; configure stack
        adrp x0, stacktop
        mov sp, x0

        b main
```

We also had difficulties getting the standard library I/O functions to work (possibly due to improper communication between the emulated devices UART and our terminal stdio). What this means is that we could not emulate stdin, since we believe it would require us to write our own UART driver for the emulated device. For this reason, we could not complete the project requirement to take user input.

6 Appendix: Code

6.1 main.c

```
#define UART_BASE 0x09000000
#include <stdio.h>

extern void fibonacci(int n, unsigned long long* series);
static void fibonacciBounded(int n, unsigned long long* series);
static void print(const char * string);
static void printArrayWithSize(unsigned long long* const array, int size);
static void itoa(unsigned int n, char* const buffer);

void main() {
  int n = 20;
```

```
  unsigned long long series[n];
  fibonacciBounded(n, series);
  printArrayWithSize(series, n);
  while(1);
}

void fibonacciBounded(int n, unsigned long long* series) {
  n < 1? print("ERROR: n must be larger than 0."): fibonacci(n, series);
}

void printArrayWithSize(unsigned long long* const array, int size) {
  for (int i = 0; i < size; ++i) {
    char buffer[10];
    itoa(array[i], buffer);
    print(buffer);
    if (i != size-1) print(", ");
  }
  print("\n");
}

void print(const char* string) {
  while (*string)
    *((unsigned int*) UART_BASE) = *string++;
}

void itoa(unsigned int n, char* const buffer) {
  if (n == 0) {
    buffer[0] = '0';
    buffer[1] = '\0';
    return;
  }

  int size = 1;
  int nCopy = n;

  while (nCopy != 0) {
    nCopy /= 10;
    ++size;
  }

  for (int i = size-2; i >= 0; --i) {
    buffer[i] = n % 10 + '0';
    n /= 10;
  }

  buffer[size-1] = '\0';
}
```

6.2 fibonacci.S

```
#define seriesSize x0
#define seriesPointer x1

#define fibonacciNumber x11
#define count x12
#define backTwo x13

    .global fibonacci
fibonacci:
    mov fibonacciNumber, 0
    stur fibonacciNumber, [seriesPointer, #0]
```

```
    add seriesPointer, seriesPointer, #8

    mov fibonacciNumber, 1
    stur fibonacciNumber, [seriesPointer, #0]
    add seriesPointer, seriesPointer, #8

    mov count, 2

loop:
    ldur backTwo, [seriesPointer, #-16]
    add fibonacciNumber, fibonacciNumber, backTwo
    stur fibonacciNumber, [seriesPointer, #0]
    add seriesPointer, seriesPointer, #8

    cmp count, seriesSize
    add count, count, #1
    blt loop

    br lr
```

6.3 Makefile

```
IMAGE := kernel.elf

CROSS_COMPILE = aarch64-linux-gnu-
CC = $(CROSS_COMPILE)gcc
CFLAGS = -Wall -fno-common -O0 -g -nostartfiles -ffreestanding -march=armv8-a

OBJS = startup.o main.o fibonacci.o

all: $(IMAGE)

$(IMAGE): kernel.ld $(OBJS)
        $(CC) $(OBJS) -T $< -o $(IMAGE) $(CFLAGS)

qemu: $(IMAGE)
        @qemu-system-aarch64 \
                -machine virt -cpu cortex-a53 \
                -smp 4 -m 4096 \
                -nographic -serial mon:stdio \
                -monitor telnet:127.0.0.1:1234,server,nowait \
                -kernel $(IMAGE)

clean:
        rm -f $(IMAGE) *.o

.PHONY: all qemu clean
```