CS382 Computer Organization and Architecture — Fall 2021

Lab 10 · Floating Point Calculation

Lecturer: Philippos Mordohai, Shudong Hao — *Date: November 9*

Contents

1 Objective

Get to know and use floating point numbers in ARMv8 assembly.

2 Task

Given a math function $y = f(x)$ and an interval $[a, b]$, use the rectangle rule to approximate the integral.

Your data segment should contain at least these variables:

```
1 .data
2 a:      .double ... // left limit
3 b:      .double ... // right limit
```

```
4 n:      .double ... // number of rectangles under the curve
5
6 .bss
7 result: .skip 8     // the result of integral
```

Using the example shown in Figure 1, the data segment should look like this:

```
1 .data
2 a:      .double 0.0 // left limit = 0
3 b:      .double 2.0 // right limit = 2
4 n:      .double 10  // 10 rectangles
5 .bss
6 result: .skip 8     // the result of integral
```

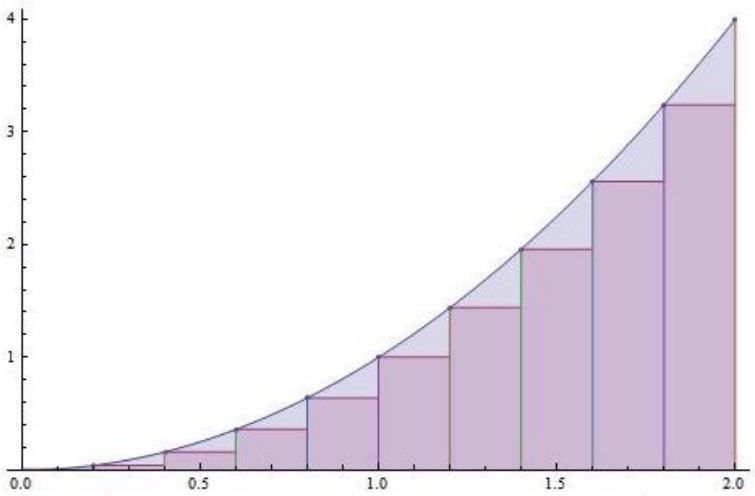

Figure 1: Example of approximation.

You can assume the curve, at least the curve between $[a, b]$ is always above x-axis.

3 Requirements

- The math function you're going to use is $y = 2.5x^3 - 15.5x^2 + 20x + 15$, and the integral is between $[-0.5, 5]$;
- The specific number of rectangles is up to you, as long as your calculation is correct, and the difference between the approximation and the actual integral is no larger than $1e - 3$;

- Please use `printf` to print out: your approximation result, the actual value of the integral (ok to calculate by hand), and the difference between them;
- You'll get zero point if you only calculated based on the integral formula instead of approximation.

What to Submit

A single assembly source code file `.s`.

A Architecture

Real numbers are a different species than integer numbers: they're encoded in a different way (if you're curious, check out Chapter 3); they are stored in a separate group of registers, instead of `X0` – `X31`; they are calculated in a unit called FPU (floating point unit), not ALU (!); and they use a totally different set of instructions.

Registers. Real numbers in ARM have 5 types of precision, but we mostly use 2 of them: single (encoded in 32 bits), and double (64 bits). Single precision numbers correspond to `float` type in C, and double precision numbers are, well, `double` type. Therefore, correspondingly, the registers that hold them are `Hn` (half precision), `Sn` (single), and `Dn` (double), where `n` ranging from 0 to 31 is the register number.

Usage. Like integer registers, we use `D0` – `D7` to pass parameters to procedures as well as store return values. `D8` – `D15` are preserved across calls, and `D16` – `D31` can be used as temporary registers. The same applies to single and half precision registers.

B Basic Instructions

For real numbers, they have a separate set of instructions, but thankfully they are very similar to the ones we know so far.

B.1 Arithmetic

Typically, when using floating point registers, the instruction has an `F` prefix. For example:

```
1 FADD  S0, S2, S3    // S0  = S2 + S3
2 FADD  D19, D12, D12 // D19 = D12 + D12
```

However, load and store are the same as before.

B.2 Moving Real Numbers

You can move an immediate real number to an `S` or `D` register using `FMOV`, but there's a restriction on that. Based on ARMv8 manual, only numbers that can be expressed as $\pm\frac{n}{16}\times 2^r$ where $n \in [16, 31]$ and $r \in [-3, 4]$ can be moved. Numbers such as `x.0` (integers), `x.5`, `x.25` are fine.

Therefore, we recommend that you just store all real numbers in the `data` segment, and load them into registers, and use `FMOV` between registers. Assume we have a number declared as such:

```
.data
pi: .float 3.1415
```

Then to move this number into a register, we should do:

```
ADR   X0, pi
LDUR  S0, [X0]
FMOV  S1, S0
```

B.3 Converting Precisions

Sometimes we'd like to cast a floating point to a double precision number. We can't just `FMOV` an `S` register to a `D` register. The instruction we need to use is `FCVT`:

```
FCVT    D0, S0   // Upcast, gain precision
FCVT    S1, D1   // Downcast, lose precision
SCVTF   D2, X2   // Convert an integer to a real number
FCVTZS  X3, D3   // Convert a real number to an integer (the fraction part␣
 ↪is removed)
```

For other instructions, the best resource out there is ARM64's reference sheet: https://developer.arm.com/documentation/100076/0100/a64-instruction-set-reference/a64-floating-point-instructions.

C Printing Using `printf`

Printing is a little bit tricky. As usual, we need to load a string to `X0`, but the rest of the parameters are passed from `D0` – `D7`.

```
.data
fmt_str: .ascii "%lf %lf\n\0"
number1: .double 3.1415
number2: .double 10
...
ADR   X0, fmt_str    // Load address of the string
ADR   X1, number1    // Load address of number1
LDUR  D0, [X1]       // Load number1 to D0
ADR   X1, number2
LDUR  D1, [X1]
BL    printf
```

When `printf` recognizes `%lf`, it doesn't go to `X1` to fetch the number; instead it goes to `D0`.

If you use `printf` to print both integer values (`int`, `char`, `long int`) and floating point values, move corresponding numbers into their registers in order:

```
.data
fmt_str: .ascii "%d = %lf, %d = %lf\n\0"
...
ADR    X0, fmt_str
LDUR   X1, [...]    // integer #1
LDUR   D0, [...]    // floating point #1
LDUR   X2, [...]    // integer #2
LDUR   D1, [...]    // floating point #2
```

!Caution! In case you declare your number as `.float` like following:

```
.data
fmt_str:    .ascii    "%f\n\0"
fpnum:      .float    3.14
```

you would need to cast `fpnum` from `float` to `double` because `printf` doesn't check `S` registers at all, and automatically uses double precision as output format (even if your format is `%f` instead of `%lf`):

```
ADR     X0,  fmt_str
ADR     X1,  fpnum
LDUR    S0,  [X1]
```

```
4 FCVT    D0,  S0
5 BL      printf
```

And if you declare your number as `.float`, you cannot load it directly into `D` registers.

Thus, the easiest way to avoid those situations is just declare your number as `.double`, and load it into `D` registers all the time.

D Debugging

The floating point registers cannot be viewed in the register panel, so we would have to use `p` to print out:

```
1 p/f $d0
```

where `/f` is to print out the register value as a floating point.