Adding Floating Point Support to CertiKOS in ARM

JASON LIU*
Yale University
jason.liu@yale.edu

Abstract

Floating point operations were successfully demonstrated on an ARM version of the CertiKOS operating system, with both software emulation and hardware instructions. A representation of such numbers was successfully displayed through several methods, including decimal floating point and scientific notation. The floating point unit of an Arndale Octa 5420 chip was enabled and instructions were compared through usage of the unit and through software abstractions by enabling the PMCCNTR, the Performance Monitors Cycle Count Register. This register allows measurement of the number of CPU cycles for calculations. The software floating point instructions finished operations in 270,000 cycles for addition and division, 216,000 cycles for multiplication, and 894,000 cycles for division. The floating point unit finished all operations in approximately 12,000 cycles regardless of the operation, resulting in a 2000% speedup. Subsequently, a test suite was enabled such that these operations could be sufficiently validated to be working. Additional math operations were tested for their validity and efficiency in relevance to speed.

I. INTRODUCTION

An important focus in today’s world is verified software, which is a machine program with a formal proof that the software does not have bugs and is reliable. It was long thought that verified software was only implementable on a theoretical scale: if the kernel and operating system were not also verified, any claim would be rendered useless by underlying deficiencies in the chain. Recently, many groups have been working on the problem of solving and verifying low-level code, including formal proof systems, verified assembly code, and even verified compilers such as CompCert\(^1\). However, very few have attempted the challenge of working on a completely integrated system from the kernel level. The overall goal of such tooling is to create programs that will not malfunction: bugs in software tooling and research can have a disastrous effect on operators or users. CertiKOS (Certified Kit Operating System) is a hypervisor that is formally verified\(^2\) to prevent against information leakage and also ensures correctness. As a result, you could say that CertiKOS is the most advanced operating system currently available in terms of mathematical proficiency. Currently, it works on x86 hardware only; since software reliability is especially important in embedded systems, there is an ongoing effort to port it to the ARM architecture. Porting the system will cause it to be used on systems worldwide in mission-critical applications such as reconnaissance.

One overall goal of this project is to determine if the software can be ported onto a vehicle such as a quadcopter, an airborne vehicle capable of hovering and autonomous or manual control. Often, many complaints about such devices revolve around the software being unreliable, causing unpredictable crashes and possible malfunctions in control. Such occurrences can represent a substantial monetary loss for the consumer and thus deter subsequent buyers from the product. The port of CertiKOS to the ARM architecture is not

\(^{*}\)Department of Computer Science, Yale University, P.O. Box 208285, New Haven, Connecticut 06520-8285, United States
fully complete, so it isn’t currently possible to load the quadcopter’s control software onto the hardware. One substantial barrier to this effort is the availability of floating point calculations. These calculations, including angle manipulations such as sine, cosine, and tangent amongst other significant operations enhance stable and safe flight patterns.

Floating point calculations are computers’ best approximation for real numbers at the moment, combining range and precision. Currently, we can only do integer calculations, sans decimal points, which is a sizeable hindrance to complicated trajectories. Other embedded systems often neglect floating point calculations entirely for ease of implementation, but this is a necessity for ongoing complications in diverging applicatory fields. While numbers cannot be calculated to complete precision due to memory and storage requirements, floating point numbers involve representing real numbers with several significant digits and an exponent power, similar to scientific notation. The first step in this process is enabling the floating-point unit (FPU), a processor used specifically for such routines. Current FPUs perform simple operations such as addition, subtraction, multiplication, division, square roots, and shifting bits. Software math libraries, such as LibM, handle most of the rest of the operations such as exponential and trigonometric functions. In older FPUs, these operations were performed in hardware but recent advances in numerical methods of algorithm solving have enabled these to be much faster in the software level.

After the FPU has been enabled, the main challenge is porting over a sufficient floating point library. Some libraries currently available in the public domain include GNU MPFR, FLIP, and LibM. The most important part for implementing the library is the number of dependencies contained in such a library; CertiKOS has isolation mechanisms for each guest application and processor resources are strictly controlled. As a result, a dependence on many standard libraries is not guaranteed, especially in the ARM architecture. As developing a new version of a math library specific for this application is quite an ambitious goal, a current library must be sufficiently robust and have a suitable propensity for porting. Current investigations involve finding suitable libraries and looking at architectural differences in possible target platforms to gauge feasibility. Three possibilities investigated were the Julia LibM, the NetBSD LibM, and the Muskel LibM (forked from the NetBSD LibM).

II. Printing Floating Point

There are only four basic data types in C: char, a single byte or character, int, an integer, float, a single-precision floating point number, and double, a double-precision floating point number. The qualifiers short and long can be applied to the int type as necessary to provide different lengths of integers, and the qualifier signed or unsigned can be applied to the char or int type. In addition, long can also be applied to double to specify extended-precision floating point.

As the float and double types are currently defined as primitive types in the C language, no additional support is needed to have them compile correctly. However, there currently was no method for printing such numbers — printf is implemented as a variadic function, which can take an unlimited number of arguments and then parses them case by case. To this, a new case was added to the formatting printf function such that calls of the value

```c
printf("%f", num);
```

were valid.

To understand how to correctly represent a floating point number, it is important to recognize how to correctly print an integer. I/O functions in standard libraries are often very obfuscated, but the key here is that the print function happens character by character: each char of the string to be printed is analyzed and then an action is taken if necessary. For normal characters in the ASCII character set (with the exception of the % character), they are printed normally to the screen without issue. When the % character is encountered, initial processing characters are looked at (for example, in the case %3.2f, the 3.2 specifies the number
of decimals and precision necessary) and then the type of the constant to be printed is determined. The most common types include d and i (decimal), o (unsigned octal), u (unsigned decimal), x (unsigned hexadecimal), s (string), c (character), p (pointer), f (float or double), e (scientific notation), and g (either e or f, whichever is convenient).

For both hexadecimal and decimal numbers, the printing process is straightforward. After receiving the value of the number from the arguments of the variadic function (and parsing whether it is a long or long long value), a function is recursively called that determines whether the number is greater than the current base, which is 10 for decimal and 16 for hexadecimal. If the number is too large, the function is recursively called; otherwise padding characters as specified by the format above are printed to the screen. In each call of the function, the number is taken modulo the base (to obtain the rightmost digit in base 10), and that is printed to the screen.

The initial approach taken was to design a function similar to that used to print decimals. One obvious problem was that the concept of modulo did not easily apply to float numbers, as the operator itself only was applicable to decimal numbers. With this in mind, one approach was to convert the floating point number into a integer through a cast and subtract the integer from the floating point number to receive the part after the decimal point. Now, the integer part could be printed with the existing function to print decimals and the portion after the decimal part could be printed by itself. Another approach involved repeatedly dividing the number by 10 until it was less than 10, then the digit itself would be printed. Afterwards, the number would be repeatedly multiplied by 10, subtracting the floor of the number, and repeating this process until the number was successfully printed. A hybrid approach was taken, printing the floor of the original number using the existing function and then printing the digits after the decimal point to 16 digit precision. One interesting quirk in the function is that the va_arg(ptr, type) function does not work with the float type. The parameters of functions (...) are promoted before being passed to a variadic function. The char and short types are promoted to int, and the float type is promoted to double.

Another approach considered was to print the floating point number in a native representation instead of the “decimal” approach. Floating point numbers are typically represented in binary, as the form:

$-1^{\text{Sign}} \times \text{Mantissa} \times 2^{\text{Exponent}}$  \hspace{1cm} (1)

While this was possible by reading the specific bits of the floating point representation, it would not be as easy to look at for a human reader as binary exponentiation would be difficult by hand. To create a better self-contained operating system, passing the numbers to an outside calculator or representation was discouraged so an approximation of the printed value would serve the purpose of conveying the current value of the number.

III. Enabling the Floating Point Unit

After printing of the floating point numbers was implemented successfully, it was possible that simple calculations such as addition, subtraction, multiplication, and division could be carried out successfully. Some examples of successful simple calculations are shown below:

```c
float a = 50.12938;
float e = a/10.1;
int f = (int) (e + 0.5);
int c = (int) (a + 0.5);
double d = 3.1415;
double b = 3.14;
double g = 0.123;
double db = d + b;
double dg = d * g;
float Qa = Q_rsqrt(a);
float Qe = Q_rsqrt(e);
printf("c: %d", c);
printf("f: %d", f);
printf("float: %f", a);
printf("float2: %f", e);
```
printf("floatq: %f", Qa);
printf("floatq2: %f", Qe);
printf("dbl: %f", d);
printf("dbl2: %f", b);
printf("dbl3: %f", g);
printf("dbl4: %f", db);
printf("dbl5: %f", dg);

However, these were all “software” floating point instructions, using regular ARM assembly instructions or GCC-defined implementations of the functions. To accelerate these operations and provide the opportunity for hardware versions of functions such as the square root function, the floating point unit (FPU), a coprocessor specifically designed for these calculations, was enabled. The Cortex-A7 FPU, part of the Arndale Octa 5420 chip, supports single-precision and double precision floating point, conversion between the two, and addition, subtraction, multiplication, division, multiplication and accumulation, fused multiplication and accumulation, and square root operations. To enable it, several ARM registers needed to have their values checked. When the system is booted, the FPU is disabled by default: the instructions below describe how to enable the FPU for future calculations. Some of the relevant ARM registers are the FPSID, the floating-point system ID register, FPSCR, the floating-point status and control register, CPACR, the coprocessor access control register, MVFR0, the media and VFP feature register 0, MVFR1, the media and VFP feature register 1, and FPEXC, the floating-point exception register.

// Set CPACR for access to CP10 and CP11
asm ("LDR r0, =(0xF << 20)");
asm ("MCR p15, 0, r0, c1, c0, 2");

// Set FPEXC EN to enable FPU
asm ("MOV r3, #0x40000000");
asm ("VMSR FPEXC, r3");

After this was enabled, an additional compilation flag needed to be added to ensure that floating point operations were correctly enabled. This was -mfloat-abi=softfp, which specifies which floating-point ABI to use. The possible options were soft (the default), softfp, and hard. The soft option causes the output from GCC to use library calls for floating point operations, in essence an emulation of floating point support through software. The softfp option allows hardware floating point instructions, but still using the conventions of the soft mode. This was the desired mode for operation of the chip, as the hardware was already configured to use prototypes of the soft mode. The hard mode allows generation of hardware floating point instructions but using FPU-specific calling conventions. However, it is not directly compatible with the soft mode and the current set of libraries, so it was not used.

IV. TESTING THE FLOATING POINT UNIT

To test the floating point unit, a library called Paranoia\(^5\) was used to test many floating point operations for their accuracy. Originally, there was an issue with compiling the library as it had a main() function. There could only be one main function in a set of C programs, and there already existed one in the shell that was launched when the environment ran. As a result, it was moved and converted to a library function that was simply called.

The source of the program is a BASIC program written by Professor W. M. Kahan, which was subsequently converted to Pascal and then C. The program first tests small integers, to see if they are properly represented. Then the precision of the radix is determined through several successive floor() functions. Subtraction is checked to see if it is normalized, and guard digits are checked in multiplication, division, and subtraction. Rounding on multiplication, division, addition, and subtraction is checked along with the sticky bit. Commutation of multiplication, the square root function, and the power function are investigated. Underflow and overflow are checked, and finally division by zero is checked to ensure that inf and nan are properly implemented.

In addition, a generic timer was initialized
so that the number of cycles per function call could be counted. This was a relative scale of each operation’s performance: by using the clock speed of the processor, the total time could be determined from the number of cycles. Initially, the generic timer was used with the CNTFRQ, or Counter Frequency Register. The CNTFRQ had to be set with the system clock frequency so that the generic timer would be available to software. However, this approach had one caveat: the generic timer was usually used for interrupts happening a specific amount of time in the future, and wouldn’t work specifically for more of a stopwatch application for measuring a function call.

Instead, the PMCCNTR, or Performance Monitors Cycle Count Register, was investigated as a possible candidate for timing the performance of a function. This register holds the value of the CCNT, or processor cycle counter, which directly gives a measurement of processor clock cycles. An addition provision was made when booting the system so that the register could be accessed in user mode, since all of the current assembly calls were enabled only in kernel space.

```c
// Let user-mode access CCNT
asm("MCR p15, 0, %0, c9, c14, 0": "r" (1));

// Disable overflow interrupt in counter
asm("MCR p15, 0, %0, c9, c14, 2": "r" (0x8000000f));
```

Then, in user space, functions were added to initialize the performance counters and read the CCNT:

```c
// Enable user performance counters
// Performance counter control register
asm("MCR p15, 0, %0, c9, c12, 0": "r" (0x17));
// Enable all counters
asm("MCR p15, 0, %0, c9, c12, 1": "r" (0x8000000f));
// Clear overflows
asm("MCR p15, 0, %0, c9, c12, 3": "r" (0x8000000f));
```

The value 0x17 (23) in the first call to enable the performance counter control register is derived from enabling several constant timers. All counters and the cycle counter are reset, but the CCNT “by 64” divider is not enabled in the calculation. The initialization process proceeds with enabling all of the counters and clearing any overflows present. This process is only performed once, but the operation to read the CCNT is called whenever necessary.

One important measurement of note is that reading the CCNT in user-mode invokes a large overhead, but this overhead is static every time. Hence, it can be calculated by reading the CCNT twice consecutively and subtracting their values to obtain the overhead from the single function call. This is subtracted from future calculations to avoid bias in the development of system cycle counting.

V. Results and Discussion

For the four operations addition, subtraction, multiplication, and division, a timing cycle was done for 100 of the same operation with both software floating point and hardware floating point instructions.

![Figure 1. Comparison of primitive instructions for VFP (vector floating point) and software instructions.](image)

Here, the blue columns represent addition, the red columns represent subtraction, the green columns represent multiplication, and...
the purple columns represent division. On the vertical axis is a logarithmic scale of the number of CPU cycles needed to perform the operations: the floating point unit finished all operations in approximately 12,000 cycles regardless of the operation. However, the software floating point instructions finished operations in 270,000 cycles for addition and division, 216,000 cycles for multiplication, and 894,000 cycles for division.

For the operations addition, subtraction, multiplication, and division, the software instructions were: `addf3`, `aeabi_dsub`, `aeabi_dmul`, and `aeabi_ddiv` respectively. These were implementations defined by GCC in terms of software interpretations from libgcc. The vector floating point operations studied were `vadd.f64`, `vsub.f64`, `vmul.f64`, and `vdiv.f64`. In these was an implicit cast from float to double (single to double floating point precision), represented by `vcvt.f64.f32`. After the calculation was finished, there was a second implicit cast back to single precision, represented by `vcvt.f32.f64`. However, this did not directly impact the speed of the calculation as the floating point calculations finished 20-fold faster than the software implementation.

As a result, a drastic speedup of operations from simply enabling hardware operations can be seen. Unfortunately, the primitive hardware operation `vsqrt.f64`, for taking the square root of a number, could not be tested against a software operation because there is no simple operation for the square root in the ARM assembly language. Due to the fact that more complicated floating point operations are the combination of many simpler functions, any increase in speed in the performance of these operations will be directly propagated to the correctness and speed of complicated results.

VI. Conclusion

With this project, there will be substantial advancements in the types and varieties of software that can run on CertiKOS. As a result, this is a further step towards creating a fully-featured operating system suitable for daily use. In general, this will represent a improvement compared to the prior methods of computing, where frequent crashes and unexpected behavior are the norm, rather than an occasional happenstance. Ultimately, this research can serve as a case study which will further the ability of quadcopters to perform with greater precision, allowing advancement on both the scientific front and the consumer front. It would be interesting to implement such a system on a distributed platform to investigate the effects of load balancing and fault tolerance. In addition, detection of the ARM hardware and necessary architecture could allow for platform-independent code to enable the relevant hardware.

I would like to give thanks to Mengqi Liu and Joshua Lockerman for their advice in troubleshooting parts of the code.

References


