ECE 271 – Microcomputer Architecture and Applications Lecture 20

Vince Weaver

`http://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

5 April 2022

Announcements

- Read Chapter 12.1, 12.4
- Midterm next Tuesday, review in class Thursday

Floating Point / Fixed Point

- We have been working with integers, signed and unsigned.
- How can you represent fractional numbers?
- How does it work in base 10?
  $1234.56 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1} + 6 \times 10^{-2}$
- You can do the same thing in binary (base2)
  $1010.10 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2}$
  This is 10.5 in decimal

- You can do this for arbitrary bases.
  You have to keep track of the decimal or “radix” point handled

Fixed Point

- Fix the decimal point somewhere inside the number
- In decimal, note that 123.45 + 12.51 is the same as 12345+1251, just you move the decimal point.
- So we can have fractional parts of integers by just moving the decimal point.

Fixed Point – Notation

- UQm.n = Unsigned fixed point, m bits to left of point, n bits to right
- Qm.n = Signed fixed point, m bits to left (one is sign bit) n bits to right

Fixed Point – Size

- Tradeoff in m vs n values
- Accuracy – how close it is to the number you are trying to represent
- Resolution – the smallest change that will give you another value

Fixed Point – UQ16.16

- Q16.16 – 16 bits of integer, 16 bits of fraction
- Use regular integer register and regular math
- Limited range, you now have smallest max value you can have
- Also need to track the radix point yourself
- Binary example
- $101.111 = 5 + 0.25 + 0.125 = 5.375$

Fixed Point – Q16.16

- Two’s complement signed
- Note: TI-Style Q notation sign bit isn’t counted: Q15.16
- AMD variant sign bit is: Q16.16
- -4.25, first do 4.25 0100 0100, twos complement whole thing 1011 1100

Addition

- Straightforward. Make sure Q for both is the same and just add as normal
- $0101.1 + 0101.1 = 1011.0$

Subtraction

- Just like addition

Multiplication

- Think about decimal. 10.1 * 2.0 = 20.2
  but how do you do it

```
    10.1
     2.0
   =====
     000
    202
 =========
    2020
```

```
Then you shift the point left by the number to
20.2
```

- What you are doing is 101×10^{-1} times 20×10^{-1} so you can do the first, then do the second
- Regular multiply
- Need to adjust radix point back
- Example, 2.5 * 2.5
  - 0010.1 * 0010.1 $=$ 0000 0000 0000 0010 1000 0000 0000 0000
  - 0x28000*0x28000 $=$ x6 4000 0000 Q16.16*Q16.16 $=$ Q32.32

◦ Need to shift right by 16 ($>>$ 16) to get final result $=$ 0x6 4000 $=$ 6.25

Multiplication – Assembly Language

- A 32x32 multiply gives a 64-bit result
- How does this work on 32-bit processor?
- ARM MUL (multiply) instruction only updates the lower 32-bits of result
- ARM has special SMULL/UMULL instructions that take two 32-bit destinations so you can get the whole result
- To convert the result of a 16.16 multiply back from 32.32 you need to combine the bottom 16 bits of the top with the top 16 bits of the bottom

- You can do that with two shifts and an OR

Division

- Similar to multiply
- 0x28000 / 0x28000 = 1 Q16.16 / Q16.16 = Q1. $<<$ 16
  What happens to fraction part?
  Shift dividend by $<<$ 16 first before divide to not lose a lot of precision

Converting to int

- Just shift right by Q to truncate
- Rounding is straightforward, add one if the first fraction bit is 1 (meaning the fraction is 0.5 or higher)
- Easy to do because shifts will put the last shifted off bit in the carry.

Overflow

- Can be a problem

Why ARM is good at fixed point

- barrel-shift instructions

Can you exactly represent all numbers?

- In decimal, 1/3? No
- In binary, only combinations of powers of 2. So even things like 1/5 (0.2) you can't represent exactly.
- Irrational numbers like Pi?

Arbitrary Precision Number Libraries

- If you need *exact* values
- Tend to be slow and use lots of RAM, but give exact results

Fixed Point Limited Range

- What if you want to operate on numbers with different Q values
- What if you want to add very large or very small numbers

Fixed Point Benefits

- Can be faster than floating point
- Uses existing integer registers / ALU
- Works on machines without floating-point unit
- For certain ranges of numbers can have better accuracy (no bits are wasted on exponent)

Floating Point Drawbacks

- special hardware
- power hungry, if not commonly used
- chip area, expense
- back in day, special chip
- rounding issues
- money calcs. 1/10 only approximate. .0001100110011
- trouble near zero

Floating Point on Cortex-M4

- Optional on Cortex-M. Our boards do have it though.
- Thirty-two 32-bit (float) registers S0 to S31
- Four special registers
  - CPACR – coprocessor access control reg
  - FPCCR – floating point context control
  - FPCAR – floating point context address
  - FPSCR – floating-point status and control

Double Precision?

- The S registers can also be read as sixteen 64-bit registers D0 to D15. D0 contains S0 and S1, D1 contains S2 and S3
- You can't do 64-bit (double) math on the Cortex-M4 in hardware
- The C compiler will emulate in software behind your back

FPU – ABI Differences

- When passing fp arguments put them in the registers.
- Up to 16 32-bit or 8 64-bit can be passed
- If you mix and match S/D then it gets complicated
- What if you want to pass more? Goes on the stack
- If result is fp return in S0/D0

IEEE 754 Standard

Enabling Floating Point

- The FPU is disabled by default
- Have to enable CP10 and CP11 (CP10 is single, CP11 double)
- Need to use memory barriers
- Note this is Cortex-M4 specific, it's slightly different on other ARM processors

```
; Enable FPU (from Cortex-M4 Technical Reference, 7-1)
; CPACR is located at address 0xE000ED88
LDR.W   R0, =0xE000ED88
; Read CPACR
LDR     R1, [R0]
; Set bits 20-23 to enable CP10 and CP11 coprocessors
```

```
ORR     R1, R1, #(0xF << 20)
; Write back the modified value to the CPACR
STR     R1, [R0]
; wait for store to complete / reset pipeline (necessary?)
DSB
ISB
```

Why enable Floating Point?

- First, why disable? Can use power
- Enable it if you want to use FP
- Any downsides to using FP?
- One is that if you have an operating system and context switching it's more registers you have to save/restore
- Some code intentionally won't use it (Linux kernel for example, by default can't use FP in kernel. What to use instead? Fixed point often, milicelsius)

Any use of FPU if not doing math?

- memcpy() is a highly optimized routine
- Why not just `for(i=0;i<1000;i++)a[i]=b[i];`?
- Byte-by-byte is OK
- What if we copied integers (32-bit) instead? Would it be 4 times faster?
- Yes, but watch out for corner cases (i.e. starting/ending not on an aligned 4-byte boundary)
- Could we make it even faster? ARM FP allows loading/storing 64-bit doubles, 8 bytes at a time

- There’s even load/store multiple on ARM so can copy bigger chunks
- Why do you want fast memcpy? Lots of reasons. Talk a bit about 2D game programming for one.

Floating Point Status and Control Register (FPSCR)

- Has the N/Z/C/V bits set by the `VCMP` instructions Integer instructions cannot use these, have to cop it to the APSR using the `VMRS` instruction first
- Has control bits
  - Alt half-precision
  - Default NaN
  - Flush-to-zero
  - Rounding mode

- Has exception bits
  - Input Denormal
  - Inexact Cumulative
  - Underflow Cumulative
  - Overflow Cumulative
  - Division by Zero cumulative
  - Invalid Operation cumulative

Rounding Modes

- 00 Round to nearest (default)
- 01 Round to $+$infinity
- 10 Round to -infinity
- 11 Round to zero

Non-standard Modes

- Also supports some modes not in IEEE 754
  - Flush-to-zero (subnormals go to 0)
  - Default NaN
  - Alternative half-precision mode (16-bits, can get extra precision vs IEEE by not allowing infinity or NAN but gaining a bit)

Exceptions / Interrupts

- Underflow/Overflow – when number is too small/big to represent
- Inexact exception – result lies between two floating point numbers, had to be rounded
- Invalid operation – things like 0 times infinity, infinity - infinity, sqrt(-1)
- Divide-by-zero
- Denormal (value to small, flushed to zero)

Interrupt Stacking

- When get an interrupt, push the FP regs on the stack too
- Can do "lazy stacking", only saves FP regs if you are using the FPU
- How can you tell? FPCA (floating point context address register) can set bit when you do a FPU instruction
- Interrupt handler will save S0 .. S15 and FPSCR (status and control) on stack for you

VFP Instructions – Load and Store

- `VLDR.F32 Sd,[Rn]`
- `VLDR.F64 Dd,[Rn]`
- `VSTR` – store
- `VLDM` – load multiple
- `VSTM` – store multiple
- `VPUSH` – push
- `VPOP` – pop
- `VMOV` – move immediate or SP/DP, also R

VFP Instructions – Math

- `VADD.F32`
- `VSUB`
- `VDIV`
- `VMUL`
- `VNEG`
- `VABS` – absolute value
- `VSQR` – square root

VFP Instructions – Multiply/Add

- Can multiply then add in one instruction
- Regular multiply/add
  - `VMLA` – multiply add
  - `VMLS` – multiply subtract
- Fused multiply/add, does not round in between
  - `VFMA` – multiply add
  - `VFMS` – multiply subtract

VFP Instructions – Conversion

- `VCVT` convert between single double, or fp to integer
- Can specify rounding method (or use default)

VFP Instructions – Compare

- `VCMP` – compare
- note goes to FP cmp register, need `MVRS` to move to integer flags registers before you can use `BEQ` or similar

Vector Instructions

- Some modern processors have more advanced vector operations
- Neon on 32-bit ARM
- MMX/SSE/AVX on x86
- Can have 128-bit (or larger) register, hold 4 32-bit values for example
- Then one vector add can add all 4 in parallel, in theory 4 times faster than regular loop with floating point one by one

- Lots of complex stuff
- GPUs (graphics cards) are also good and doing floating point math in parallel like this