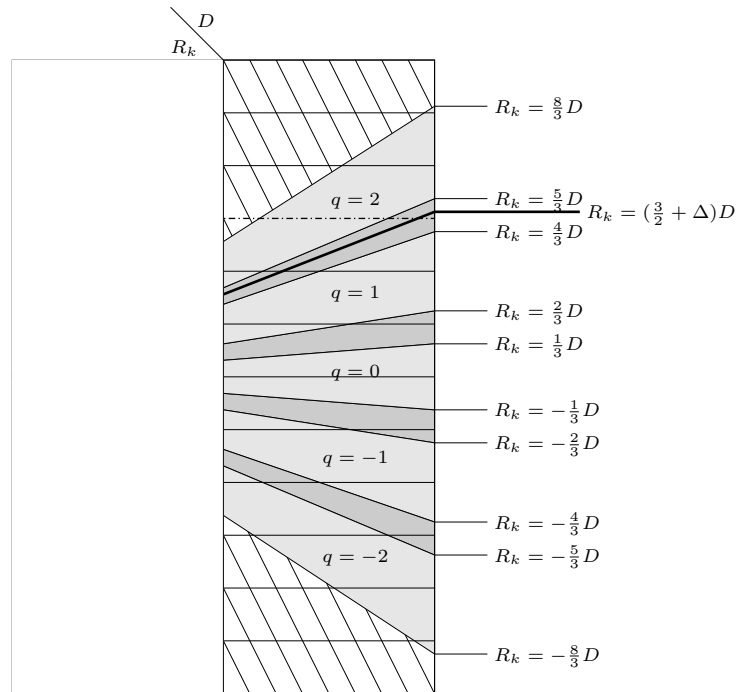


# Homework 3 Solutions

Prof. Loh  
 CS3220 - Processor Design - Spring 2005

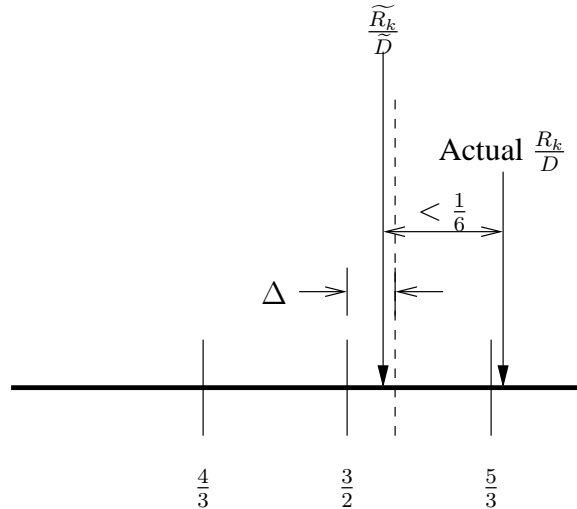
## 1. SRT Division

Let us suppose that the engineer in charge of the SRT lookup table is not good at plotting straight lines, and ends up using a line that is slightly “higher” than  $\frac{R_k}{D} = \frac{3}{2}$  (shown as a bold line in the figure below – the “correct” line would have terminated slightly lower where the dash-dot horizontal line hits the end of the table). That is, he uses the line  $\frac{R_k}{D} = \frac{3}{2} + \Delta$  (see the figure below). How would this affect the correctness of the divider? Assuming you cannot change the lookup table, how would you modify the algorithm to fix the problem?



**Solution:**

- The problem uses a value  $\Delta$  which is said to be only “slightly” off. If  $\Delta$  is too large, then there is not too much we can do about it. The divider will generate incorrect answers.
- Assuming that  $\Delta$  is small, consider what this does to the number line:



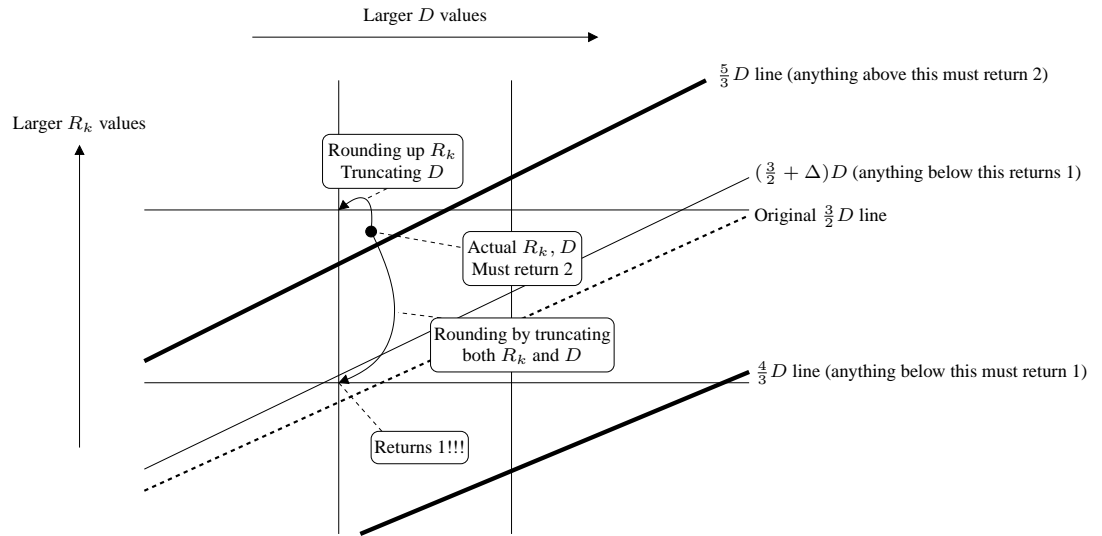
Using the example values in this figure, the actual value of  $\frac{R_k}{D}$  falls in the range where the SRT table *must* return a 2. The error in the approximation of  $\frac{R_k}{D}$  is less than  $\frac{1}{6}$ . With the boundary line between returning a value of 1 and 2 shifted by  $\Delta$ , it is now possible that the approximated value falls to the left of our new boundary, which causes the table to return an erroneous value. After this point, the error is unrecoverable and the divider will return an incorrect value.

Note that the option to return more than one value only applies to values of  $R_k$  and  $D$  that fall into the “overlap” regions. Any value that maps to a location in the table where only one distinct interval covers it *must* return the corresponding value.

Assuming we cannot change the entries in the SRT table, the way to fix this problem is to control the error in the approximation. Normally, reducing the error would require using more bits for  $\tilde{R}_k$  and  $\tilde{D}$ , but adding more bits to these approximations would require to increase the size of the table, which violates the assumption of not being able to modify the table.

When computing  $\tilde{R}_k$  and  $\tilde{D}$  by truncating the less significant bits, we shift the original point  $R_k, D$  down and to the left in the lookup table (see figure below). For a value of  $R_k$  and  $D$  that *must* return 2, this truncation can place us below the  $\Delta$  offset causing an erroneous 1 to be returned.

Rounding by truncation is equivalent to rounding down to the next allowable number. Truncating  $D$  moves the point to the left. Instead of truncating  $R_k$ , we can instead round it *up* to the next allowable value. This has the effect of shifting the value up and away from the “danger zone”, thus avoiding the return of an erroneous value.



## 2. Floating Point

In class we covered the IEEE single- and double-precision floating point formats. Let us consider a “half-precision” format:

Sign	Exp	Mantissa
s	e e e e e	m m m m m m m m m m m

Except for the different field sizes, assume that the format is in all other respects the same as the single- and double-precision FP formats.

- (a) What is the largest representable real number using this format? (not counting  $+\infty$ )

$$\begin{aligned}
 &0111111111111111 = \\
 &1.1111111111_2 \times 2^{(11110_2 - 01111_2)} = \\
 &(2 - 2^{-10}) \times 2^{15} = \\
 &1.9990234375 \times 2^{15} = 65504
 \end{aligned}$$

- (b) What is the smallest representable (normalized) non-zero number?

$$\begin{aligned}
 &0000010000000000 = \\
 &1.0000000000_2 \times 2^{(00001_2 - 01111_2)} = \\
 &1.000 \times 2^{-14} = 6.10351... \times 10^{-5}
 \end{aligned}$$

- (c) What is the base-10 value of the half-precision number 1110011001011000?

$$\begin{aligned}
 &1110011001011000 = \\
 &(-1)^1 \times 1.1001011000_2 \times 2^{(11001_2 - 01111_2)} = \\
 &-1.5859375 \times 2^{10} = -1624
 \end{aligned}$$

- (d) What is the half-precision binary representation of  $\frac{13}{32}$ ?

$$\begin{aligned}
 \frac{13}{32} &= \frac{8}{32} + \frac{4}{32} + \frac{1}{32} \\
 &\rightarrow 0.01101 \\
 &= 1.101000 \times 2^{-2} \\
 &\rightarrow 0011011010000000
 \end{aligned}$$

### 3. ISA Design Tradeoffs

There are multiple correct answers for this section. You need not have all of them to receive full credit.

- (a) What advantages does a VLIW architecture like IA64 have over a CISC architecture like x86?
- explicit concurrency, find it once at compile time (as opposed to repeating the work over and over again in hardware).
  - fixed instruction length: easy to decode
  - individual insts are RISC-like: easy to decode
  - large number of registers
- (b) What advantages does a VLIW architecture have in common with a RISC architecture like MIPS or Alpha?

The answers for this are similar to the previous part because the sub-instructions that comprise a VLIW instruction are basically RISC-like.

- fixed instruction length: easy to decode
  - individual insts are RISC-like: easy to decode
  - large number of registers
- (c) What complexities does VLIW introduce over RISC architectures?
- large burden on software (compiler) to find parallelism to deliver performance
  - if not much parallelism in code, must insert many nop's to fill the blanks
  - IA64 template bits are extra "overhead" (less code density)
  - IA64 template bits add decode complexity
- (d) What advantages does a CISC architecture hold over VLIW?
- code density (especially when considering the nop's needed for VLIW padding)
  - simpler compiler
- (e) Is VLIW a good idea? Why and/or why not?

There isn't really a correct answer for this question. There are pros and cons to VLIW. Besides IA64, there are other domains that make extensive use of VLIW architectures (such as in the digital signal processing area).

The idea is that the compiler (a) has a lot of time to search for parallelism, and (b) has a larger scope to search for parallelism (i.e. it can look at the whole program all at once whereas typical non-VLIW machines only look at a few instructions at a time). The other component is that by pushing the complexity to the compiler, the hardware can be made simpler. The hardware need not explicitly check for dependent instructions.

The compiler is a potential strength, but it can also be a major weakness. The compiler is responsible for generating correct code, and that forces it to be conservative which sacrifices performance. For example, it may be that two load and store instructions never access the same address. Unless the compiler can find some way to *prove* this, it cannot allow these two instructions to execute in parallel because there is still some small non-zero chance that there is a memory dependency.

The compiler sets the instruction schedule at compile time, and therefore the ordering of the instructions cannot be modified in reaction to run-time events (such as cache misses).