

A Design Strategy for Fixed-Word-Length Data Paths

Gregory W. Donohoe, David Buehler, and Stephen Bruder*
UNM Microelectronics Research Center
*New Mexico Institute of Mining and Technology

Abstract – In this paper we present an approach to automating portions of the design of applications built on fixed-word-length data paths. The goal of this research is to produce a software tool capable of automatically generating the data formatting steps of a fixed-width data path so as to maximize precision while preventing overflow.

1. Introduction

Designers of embedded systems for sensor signal processing and control must pay careful attention to data formats, in order to minimize information loss in finite-width data paths [4]. For many applications, the availability of floating point processors simplifies data path design. However, with an increasing trend toward highly parallel systems using multiple processing elements in Application-Specific Integrated Circuits (ASICs), Complex Programmable Logic Devices (CPLDs), and reconfigurable computing, implementation costs often render floating point data paths impractical. Historically, arithmetic operations have been implemented in fixed-point data paths on an *ad hoc* basis: the designer must insert information-reducing operations such as clipping, shifting and truncation into the data path to prevent overflow. The challenge to the designer is to decide which width-reducing operations to insert, and where to place them. This paper presents a step toward automating this process.

2. Data Representation

Typically, an application designer knows the range of values a parameter can take on, but must map this range onto a fixed-width binary representation. For example, in a particular finite impulse response (FIR) filter, we may want all of the filter coefficients to fit within the range of -1 to $+1$, or $b_i \in [-1,1]$. In a video signal calibration application, we may need to represent an calibration parameter in the range $[-2.5,+3.0]$. Given the range of values a parameter can take, the designer must convert the range into strings of bits, and provide a set of operations so that arithmetic operations produce sensible results.

Fixed Point

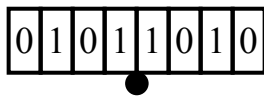
The simplest non-integer data representation is fixed point, in which the binary point is fixed at one location in the bit string. Fixed point has the advantages of simple hardware

implementation and high speed. In addition, for a fixed dynamic range and a given number of bits, fixed point allows the maximum precision: all of the bits are used for data, none for scaling. Digital hardware does not explicitly represent the binary point; rather, from the point of view of the hardware, arithmetic is computed on integers, usually in two's complement form. Interpretation of the results, and placement of the binary point, is left to the application designer. We place the binary point of an N -bit, two's complement number at index p , where p is the number of bits to the right of the binary point. The value of the fixed point number in two's complement is given by:

$$V = -b_{N-1} \times 2^{N-p-1} + \sum_{i=0}^{N-2} b_i \times 2^{i-p}$$

where $i = 0$ at the rightmost end of the string.

In the 8-bit example below, the black dot marks the binary point.



This is a positive number, and the leftmost bit is zero, so the first term vanishes and we have:

$$V = 2^{-3} + 2^{-1} + 2^0 + 2^2 = 5.625$$

A favorable property of the fixed point representation is that the spacing of numbers is uniform over the representable range. This makes numerical behavior very predictable, and simplifies error analysis.

While it is fast and accurate, fixed point has limited dynamic range, and suffers from a “fail hard” behavior on overflow. Operations like addition and multiplication produce results with more bits than the input operands. If the result is too large to fit in the width of the data path, it either saturates or wraps around, producing a nonsense value, depending on the hardware implementation. The designer must follow these operations with data-path-reducing scaling operations, such as clipping, shifting, or truncation. Since the semantics of the numbers are not explicit in the representation – i.e., their position on the real number line depends on interpretation -- this “design time scaling” must be done carefully to make sure that numbers are appropriately aligned, and ensure proper interpretation of the results.

Floating Point

While fixed point arithmetic requires “design time scaling”, floating point provides “run time scaling”. In a manner analogous to scientific notation, the floating point

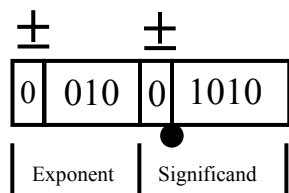
representation includes an exponent, to provide scaling, and a significand, which encodes the scalar data itself, as shown below [2].



In computer hardware, a floating point has five elements: the number of bits in the exponent, the number of bits in the significand, the radix of the exponent, and the signs of the significand and exponent. The value V_{FP} of floating point number is

$$V_{FP} = V_S \times r^{V_E}$$

where V_S is the value of the significand, and V_E , the value of the exponent. In the example below, let us assume a floating point system with a three-bit exponent, a five-bit significand, and a radix of 2. The significand is normalized with the binary point to the left. The sign bits are marked with ‘±’.



The exponent has a decimal value of 2, and the significand a value of $\frac{1}{2} + \frac{1}{8} = 0.625$.

Thus, the value of the number is $0.625 \times 2^2 = 2.5$.

The floating-point hardware adjusts the exponent automatically to ensure that the numbers are properly aligned for each arithmetic operation. Scaling and normalization take place according to a pre-determined set of rules built into the hardware. As a floating point number grows, the exponent expands, allowing the number to encompass a larger dynamic range. This is a “fail soft” mechanism; instead of overflowing, large numbers are accommodated by decreasing the precision, or increasing the significance, of each bit in the significand. This complicates floating point error analysis. Behrooz describes floating point arithmetic in detail [2].

The run-time scaling of floating greatly simplifies the management of dynamic range, and this makes it very popular with designers. In addition, the semantics of the representation are explicitly encoded in the representation and embedded in the hardware, so converting between the hardware representation and the application is trivial. However, this convenience comes at a significant cost in hardware complexity and execution speed, rendering floating point impractical for large pipelines with many processing elements. In addition, the need for an exponent leaves fewer bits for data.

3. Automated Design-Time Scaling

For embedded systems with simple processors lacking floating point arithmetic, and for pipelined data paths, it would be useful to have the hardware efficiency of fixed point with the design-time convenience of floating point. The designer could specify a range of data values and a scaling policy; an application compiler would allocate bits accordingly, and insert scaling operations such as shifts, truncations, and clipping, to maintain acceptable precision while avoiding overflow. In order to implement such a compiler, we need a notation system.

SIF Notation

The Sign-Integer-Fraction (SIF) format notation described here is adapted from the block floating point notation described in [2,3]. In a fixed-point data word, each bit can represent a sign bit, an integer bit, or a fraction bit. In two's complement, sign bits are replicated from the left. Thus, the 8-bit two's complement representation of +2, 00000010, has six positive sign bits ('0'), of which five are redundant. Likewise, the representation of -2, 11111110 has six negative sign bits ('1'). We divide the bit string into sign, integer and fraction fields, and use the format specifier:

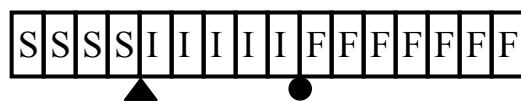
$$x \approx (\pm S / I / F)$$

where: *S* is the number of sign bits
 I is the number of integer bits
 F is the number of fraction bits
 + or - means the number is positive or negative (if known)

The symbol '≈' reminds us that this notation describes a number *format*, not a value.

Unless otherwise stated, we assume a policy of left-aligned data: that is, during intermediate stages of arithmetic processing, the bits are shifted as far left as possible to minimize the number of redundant sign bits, and to retain the maximum number of data bits. Raw input data, such as that from an analog-to-digital converter, is usually right-justified.

In the 16-bit example below, the representation is (4/5/7). The triangle marks the boundary between the sign and integer fields, and the dot marks the boundary between the integer and fraction fields. The letters 'S', 'I' and 'F' mark sign, integer and fraction bits, respectively.



The purpose of this notation is to provide information to the data path compiler about the range of the numbers and the available bits, to enable the compiler to format the data in

an optimal way. Thus, the presence of the optional + or – informs the designer that this is guaranteed to be a positive or negative number, allowing more freedom in determining how it will be manipulated. Since the sign bit is replicated from the left, the value S , the size of the sign field, indicates how many sign bits of headroom we have to work with.

It is important to note that the bit string alone is not enough to specify a number to the application; both the bit string and the format information are required. With these, we can establish a procedure for specifying a data path.

Rules for SIF Data Manipulation

SIF notation allows us to format the data with several goals in mind: (1) to ensure that bits are properly aligned so that operations like addition produce a sensible result; (2) to guide us in adjusting the data after a data-path-widening operation, to reformat it for subsequent processing; (3) to enable us to adjust the data so that it exits the processing chain in a format that makes sense for the application.

Addition

Encompassing addition, subtraction, and comparison, the addition operation requires the binary points to be aligned. Assume that the two operands x and y contain the same number of bits, and that they have the formats $x \approx (S_x / I_x / F_x)$ and $y \approx (S_y / I_y / F_y)$. Before we can compute the sum $r = x + y$, we must adjust x and y so that $S_x + I_x = S_y + I_y$. If both operands have the same sign, there is the possibility of a carry from the integer field into the sign field. To prevent overflow, we require that $S_x \geq 2$ and $S_y \geq 2$. If the operands are properly aligned, then the format of the result can be derived from:

$$S_r = \min(S_x, S_y) - 1$$

$$I_r = \max(I_x, I_y) + 1$$

$$F_r = \max(F_x, F_y)$$

Multiplication

Multiplication has the following properties: (1) if we multiply two N -bit strings, the result contains $2N$ bits; (2) the number of sign bits in the result is the sum of the two sign bits. Summarizing, the format of the product $r = x \times y$ can be obtained from:

$$S_r = S_x + S_y$$

$$I_r = I_x + I_y$$

$$F_r = F_x + F_y$$

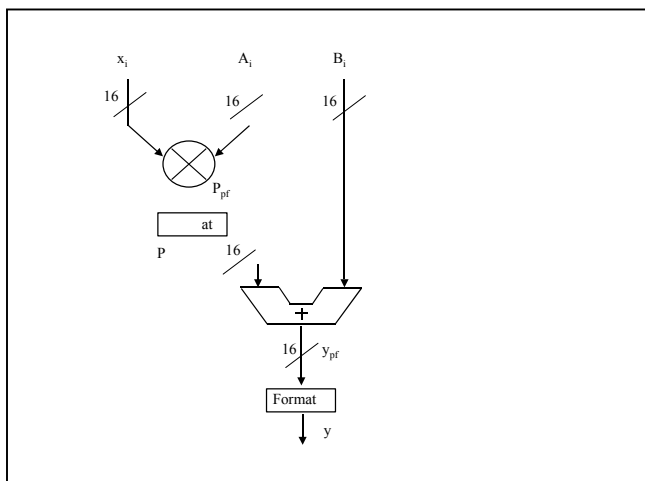
4. Example: Pixel Readout Correction

Imaging focal plane arrays typically exhibit pixel-by-pixel variation due to manufacturing tolerances; in particular, each pixel has a brightness offset due to leakage or dark current, and a gain variation. To obtain accurate data, the sensor must be calibrated. In a calibration phase, we estimate the offset and gain factor for each pixel, storing these in a memory. In operation, we correct each pixel by multiplying by its corrective gain factor, and adding its corrective offset. Thus, the output pixel y is computed from input pixel x by $y = Ax + B$.

The data path to carry out this correction in the operation phase is shown below. In this example, we make the following assumptions:

- Data paths are 16 bits wide
- Input x : 12 bits, unsigned, or $x \in [0,4095]$
- Gain factor: $A \in [0.5,2.0]$
- Offset: $B \in [-63,63]$

The units of x and B are counts; A is dimensionless. In SIF notation, we have $x \approx (+4/12/0)$. Next, we need a representation for A that maximizes the precision of the multiplication.



Since the integer part of A requires at most two bits, we assign one bit for the sign, two for the integer, and the rest for the fractional part, giving $A \approx (+1/2/13)$. Performing the multiplication and applying the multiplication rules, we have a raw or pre-formatted product with the format $P_{pf} \approx (+5/14/13)$, producing a 32 bit result.

The offset B is represented with 5 integer bits and no fraction bits: $B \approx (11/5/0)$. To format product P for the 16 bit addition, leaving two sign bits to account for possible overflow, we shift P left 3 places and truncate to remove the lower 16 bits, yielding a formatted product: $P \approx (+2/14/0)$. Now summing P and B , we have a pre-formatted output $y_{pf} \approx (+1/15/0)$. Finally, so that output y will have the same format as input x , we perform an arithmetic right shift three places, to get $y \approx (+4/12/0)$.

5. Conclusion

9th NASA Symposium on VLSI Design, Albuquerque, NM, Nov 8-9, 2000.

This paper has outlined a notation scheme and scaling rules, which will serve as a step toward developing a software tool for compiling applications into machines or pipelined with fixed-width data paths. The goal is to provide the designer with the convenience of floating point with the economical hardware implementation of fixed point. The description here is not exhaustive, merely illustrative of the approach. The details are the subject of research by the authors in developing a software tool called *SIFtool*, to be used as a design aid for data path generation for pipelined processors.

6. References

Ahmed, Nasir, and T. Natarajan, *Discrete-Time Signals and Systems*, Reston Pub. Co., 1983.

Behrooz, Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.

Fogler, R.J., *On a Block Floating-Point Implementation of an Intrusion-Detection Algorithm*, M.S. Thesis, Kansas State University, October, 1979.

Mitra, Sanjit K., *Digital Signal Processing: a Computer-Based Approach*, McGraw-Hill, 1998.