CHAPTER 31

THE IMPLICATIONS OF FLOATING POINT FOR FPGAS

Keith D. Underwood, K. Scott Hemmert
Sandia National Laboratories

FPGA-based computing has a long history of accelerating assorted types of computations in integer and fixed-point arithmetic. Until recently, however, applications based on floating-point arithmetic have been a relative rarity. This stems from early work [6, 12, 13] that indicated that IEEE-754 standard [11] floating point was a poor match for field-programmable gate array (FPGA) technology. This led directly to numerous efforts that created libraries using specialized floating-point formats [1, 3, 7], where the width of the exponent and the width of the mantissa could be specified. Unfortunately, many scientific applications require compliance with the IEEE standard. While seemingly an arbitrary requirement, it is driven by several factors. Foremost, some scientific applications have data with high dynamic ranges and high precision requirements. A good example is a typical linear solver that needs high precision to guarantee convergence of the algorithm. Second, application developers rely on the portability of their applications and the reproducibility of their results. Put another way, it is difficult to trust results that differ on every platform that runs them.

Fortunately, recent work indicates that FPGAs are viable competitors in IEEE-compliant floating-point arithmetic [14], and there has been an explosion of interest in mapping floating-point kernels to FPGA platforms [2, 5, 8–10, 15, 17, 18]. However, while FPGAs are now capable of implementing floating-point applications, the use of floating point in FPGAs still requires a great deal of care. This chapter introduces the IEEE floating-point standard and discusses implementations of compliant floating-point units for FPGAs. Section 31.2 contains case studies of three floating-point application kernels and their implementation on FPGAs.

31.1 WHY IS FLOATING POINT DIFFICULT?

Floating-point arithmetic is fundamentally different from typical integer or fixed-point arithmetic. Where integer and fixed-point values are typically stored in 2's

Note: Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

complement, floating-point numbers are typically stored in signed-magnitude format. Floating-point numbers also add an exponent field to control the position of the decimal point in the value. The most widely used floating-point format is the IEEE-754 standard. As an example, the IEEE double-precision floating-point format is shown in Figure 31.1. The mantissa (fraction part) is 52 bits, the exponent is 11 bits, and the sign is a single bit. A simple picture, however, cannot tell the full story of the complexities of the IEEE format.

First, as the figure suggests, the exponent in the IEEE format is maintained in *biased* notation. That is, rather than being in a signed-magnitude or 2's complement format, a *bias* is added to the true exponent to store it. For double precision, the bias is 1023 (approximately half the range). This means that an exponent of -1022 is stored as a 1. The second complication in the format is the use of an *implied 1*. An implied 1 means that the stored number is maintained in a normalized format such that there is a 1 immediately to the left of the decimal and the decimal is immediately to the left of the stored value. This allows the format to have an extra bit of precision without having to store it. Thus, the value can be extracted as shown in equation 31.1.

$$(-1)^S \times 2^{exp-bias} \times 1.mantissa \tag{31.1}$$

The format, as discussed so far, would have a major shortcoming. The number 0 would be impossible to represent. Since humanity has had the use of 0 for a few millennia now, the format inventors thought it best to include it by reserving a special value. They also saw fit to include representations for ∞, $-\infty$, and not-a-number (NaN), which is used as the result of meaningless operations (e.g., $\infty \times 0$). The reserved special values are summarized in Table 31.1.

As the table implies, both positive and negative 0 are possible (0 and 1 for the sign bit, respectively) as are positive and negative infinity. Several values require that the maximum possible value be loaded into the exponent field (i.e., all bits are set to 1 in the field). Finally, there is a set of values known as denormals.

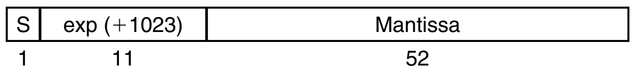


FIGURE 31.1 ■ IEEE double-precision floating-point format.

TABLE 31.1 ■ Special values in the IEEE-754 format

Special value	Sign	Exponent	Mantissa
Zero	0/1	0	0
∞	0	MAX	0
$-\infty$	1	MAX	0
NaN	0/1	MAX	nonzero
Denormal	0/1	0	nonzero

Denormals are a special form of IEEE floating-point numbers that provide a small amount of extra precision as the result of an operation approaches underflow. Unlike most IEEE floating-point numbers, they do not include the implied 1. Instead, they have an exponent of 0, keep the decimal immediately to the left of the stored value, and allow the first 1 to fall anywhere in the stored value. Denormals are particularly useful for code such as: if `(x!= y) z = 1/(x - y)`. This code should never cause an exception, but without denormal support it can easily cause a divide by 0 when x and y are small enough and close enough that the format cannot represent the difference. Floating-point hardware within a microprocessor typically implements denormals with an exception that then computes the value via software. However, in an FPGA-based implementation, to support full IEEE floating point we must generally add denormal support into the hardware itself. Thus, for denormal numbers, the value is extracted as in equation 31.2.

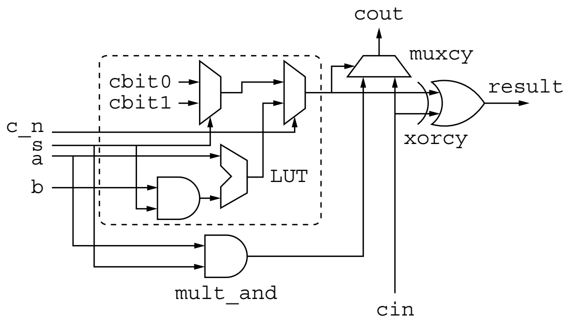
$$(-1)^S \times 2^{exp-bias} \times 0.mantissa \tag{31.2}$$

31.1.1 General Implementation Considerations

To produce the smallest, fastest circuits, it is necessary to efficiently use the structure of the FPGA. This comes up in two areas: (1) It is necessary to fully utilize every lookup table (LUT), whenever possible and (2) it is advantageous to provide an optimized layout for each unit. The floating-point units presented here have been written using JHDL—a structural design tool that provides a clean mechanism for mapping and relationally placing logic.

The units were optimized by identifying opportunities to combine logic into the LUT architecture of the FPGA. This can be challenging, particularly for operations that use the carry-chain logic. However, the special values in the IEEE format make it vital that carry-chain and other logic be mixed. For example, there are many instances where the output of the exponent logic is either the result of an arithmetic operation or a constant. For FPGA architectures, such as the Xilinx Virtex family, it is possible to map the arithmetic operation and the constant generation into the same LUT (along with its associated carry logic).

Take, for example, the `passAddOrConstant` circuit. It has four possible outputs: $a+b$, a, $c0$, or $c1$, where a and b are variables and $c0$ and $c1$ are constants. The inputs to the circuit are a, b, s, and c_n. When $c\_n = 0$, the output is one of the two constants, which is selected by the s input. Otherwise, the result is $a+b$ when $s = 1$ and a when $s = 0$. The logic used for each bit is shown in Figure 31.2(a). The circuit is only possible because of the `mult_and` added in the Virtex family of FPGAs. `mult_and` was originally intended for use in multipliers built from logic, but it enables many other useful optimizations. The same basic logic can also create a `passSubOrConstant`, and if the AND gate before the arithmetic operation is left off, the circuit is simply an `addOrConstant` or `subOrConstant`. These circuits are used to reduce the amount of logic and the logic delay required to compute the exponents. The JHDL code used to generate each bit of this circuit is shown in Figure 31.2(b). Note that all the logic is



(a)

```
// Produce the constant bit. The Xilinx tools believe
// that gnd and vcc are inputs to the LUT, so we can't
// use them. Instead, use c_n, which will be 0
// when the constant is selected.
Wire cbit0 = ((c0 >> i) & 1) == 1 ? not(c_n) : c_n;
Wire cbit1 = ((c1 >> i) & 1) == 1 ? not(c_n) : c_n;
Wire constant_result = mux(cbit0,cbit1,s);

// Generate the sum bit.
Wire sum = mux(constant_result,
          xor(a.gw(i),and(s,b.gw(i))),c_n);

// Map all the above logic in a single LUT
Cell x = map(c_n,s,a.gw(i),b.gw(i),s_partial);
place(x,0,virtex ? maxrow - i/2 : i/2);

Wire mult_and_out = wire(1,"mult_and_out" +i);
x = new mult_and(this,c_n,a.gw(i),mult_and_out);
place(x,0,virtex ? maxrow - i/2 : i/2);

x = new muxcy(this,mult_and_out,cin,s_partial,cout);
place(x,0,i/2);

x = new xorcy(this,s_partial, cin, output.gw(i));
place(x,0,i/2);
```

(b)

FIGURE 31.2 ■ Logic (a) and JHDL code (b) for the *i* th bit of the passAddOrConstant.

first mapped into LUTs using the map function, then relationally placed, using the place function. The same place function is used to relationally place the lower-level blocks at each level of hierarchy. The overall unit is placed into a rectangular area so that it can be easily tiled in a design (see the descriptions of the adder and multiplier in Sections 31.1.2 and 31.1.3).

In addition to concerns about efficiently using the LUT and providing good placement directives, there are concerns about where to pipeline the units. The major concern that largely determined the pipelining of the units presented here involves the carry-chain logic. In the Virtex family, the times to initalize and finalize the carry chain are large relative to the per-bit propagation time on the carry chain. Thus, it is necessary to avoid having cascaded carry chains in the same stage. In most cases, this constraint determines the stage mapping.

31.1.2 Adder Implementation

The most noticeable difference between integer operations and floating-point operations is in the implementation of the adder. A 64-bit registered integer adder requires 64 4-LUTs, 64 flip-flops, and the associated carry-chain logic. It can be packed into 32 *slices* in a Xilinx Virtex-4[1] or similar family. In stark contrast, a 64-bit floating-point adder requires hundreds of 4-LUTs, hundreds of flip-flops, and nearly 700 *slices*. The core of the differences can be seen in Figure 31.3(a).

The fundamental problem is that two numbers of the form

$$(-1)^{S0} \times 2^{exp0-bias} \times 1.mantissa0 \qquad (31.3)$$

and

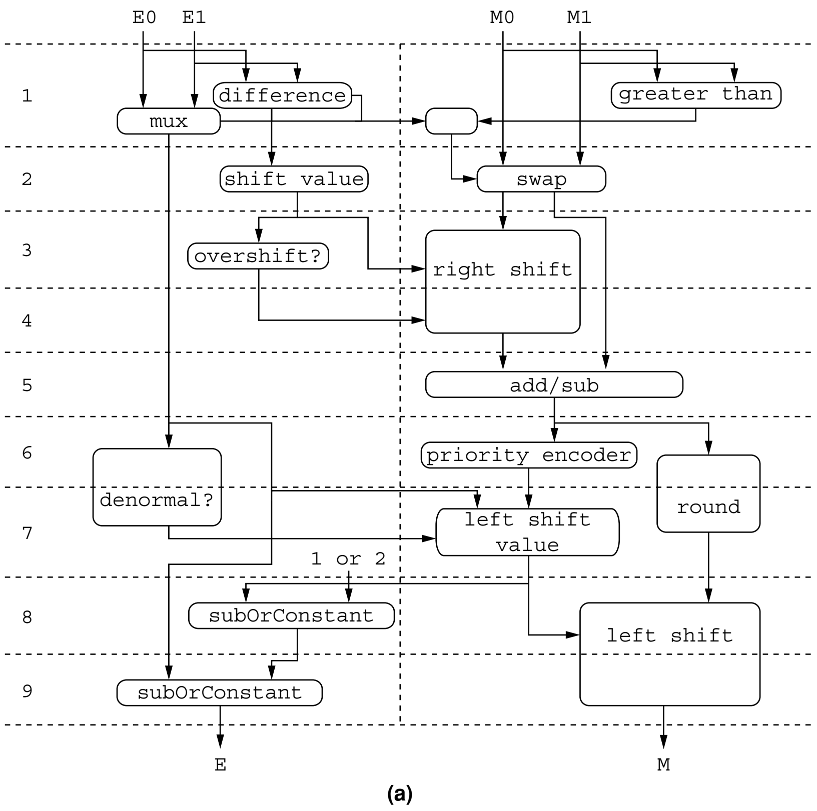
$$(-1)^{S1} \times 2^{exp1-bias} \times 1.mantissa1 \qquad (31.4)$$

must be added together. The signs can be the same or different, so the actual operation may be an addition or a subtraction. Worse, the exponents can differ (dramatically), so the two mantissas must be aligned before the operation can proceed. When the two are combined (different signs and different exponents), it becomes necessary to determine which number is larger so that they are subtracted in the right order. If the exponents are the same but the signs are different, the result can yield a very small mantissa, which must be normalized (i.e., the leftmost one is moved to the leftmost position) before it can be stored.

Looking again at Figure 31.3(a), we can see the impact of the extra format. Each horizontal dashed line represents a register, and the vertical dashed line separates the exponent path from the mantissa path. Note that the first two stages are spent inspecting and preparing the numbers and determining whether either of the inputs is one of the special values. The third and fourth stages are needed to align the mantissas, and it is not until the fifth stage that the actual *operation* occurs. In the exponent path, stages six through nine clean up the exponent to handle a variety of exception conditions. The sixth and seventh mantissa stages have two parallel paths: one for rounding the result and one for computing the shift value if the result must be renormalized. The last two stages are used to renormalize the result (if needed).

Figure 31.3(b) shows the approximate layout of the logic used in an implementation of the floating-point adder. For the adder implementation, it is possible to place all pipelining registers in the same slices as the logic, though some registers are placed in slices with unrelated logic. Of the total area, approximately 39 percent is used to align the mantissas prior to the actual add or subtract operation; this area includes right-shift logic and swap logic. These operations would be required for any floating-point format; however, the left-shift on the backend is only required because of the existence of the implicit 1 in the format. This case arises during a loss of precision when two numbers with

[1] A slice is two 4-LUTs, two flip-flops, and the associated carry-chain logic in this generation.



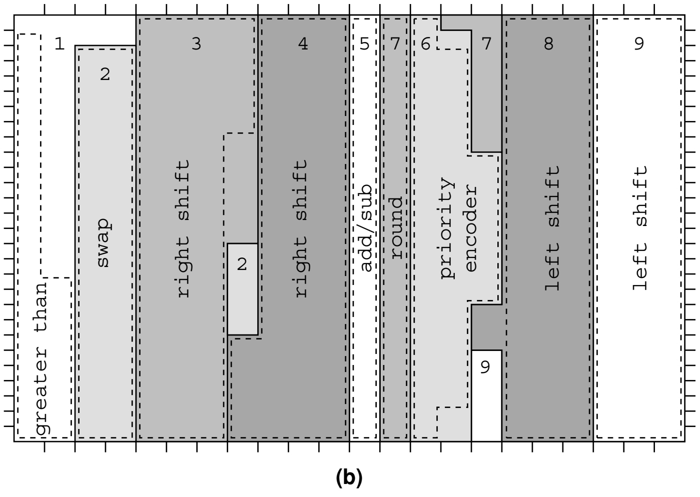


FIGURE 31.3 ■ Adder block (a) and adder layout (b) diagrams.

identical, or very close, exponents are subtracted and require normalization. The normalization logic, including a priority encoder to locate the first 1, uses another 39 percent of the logic. For comparison, the actual add and round logic consumes only 9 percent of the area.

31.1.3 Multiplier Implementation

The relationship between a floating-point multiplication and a fixed-point multiplication is a little more unusual. A fixed-point multiplier grows with the square of the width of the input. At the core of a floating-point multiplier is a fixed-point multiplier that multiplies the mantissas. Since the mantissa is significantly narrower than the floating-point number, a 64-bit fixed-point multiplier actually has a much larger core operation than a 64-bit floating-point multiplier because the floating-point multiplier only has to multiply two 53-bit mantissas. It does, however, have a lot of other work to do that more than makes up for the difference.

Floating-point multiplication starts with two numbers:

$$(-1)^{S0} \times 2^{exp0-bias} \times 1.mantissa0 \tag{31.5}$$

and

$$(-1)^{S1} \times 2^{exp1-bias} \times 1.mantissa1 \tag{31.6}$$

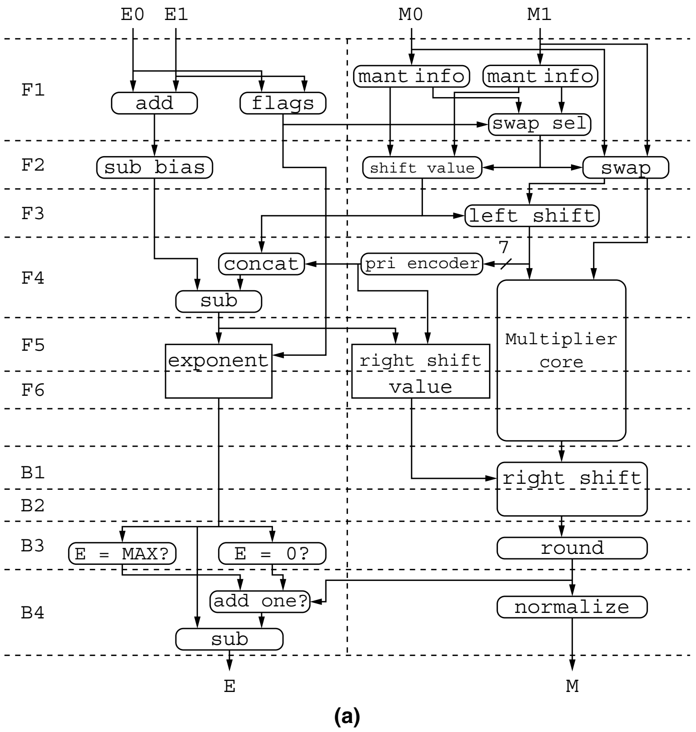
that produce the result:

$$(-1)^{(S0 \oplus S1)} \times 2^{(exp0-bias)+(exp1-bias)} \times 1.mantissa0 \times 1.mantissa1 \tag{31.7}$$

Conceptually, the dataflow shown in Figure 31.4(a) is quite simple. The first three stages unpack the IEEE format looking for special cases and preparing a possible denormal mantissa for the multiplier core. Stages F4 through F6 operate concurrently with the multiplier core and compute the resulting exponent and determine whether the result is denormal. The four backend stages provide shifting for creating denormal numbers, rounding, and normalization, which includes adjusting the exponent when required.

Figure 31.4(b) gives the approximate layout of the logic for the front- and backends of the multiplier. The multiplier core (not shown in the figure) uses nine 17×17 multiplier blocks plus additional logic to sum the partial products to create a 53×53 multiplier core. The logic used in the core is about 40 percent of the total multiplier logic. Unlike the adder, it is not possible to place all of the required pipelining registers in slices used by the logic. The black regions in Figure 31.4(b) are either unused or used by pipelining registers.

The logic required to support the IEEE format is nontrivial. Support for denormals consumes 40 percent of the multiplier area and includes logic to gather information about the mantissa, swap the mantissa, and shift the mantissa. Thus, supporting denormals requires approximately the same amount of logic resources as the multiplier core. An additional 7 percent of the area is used for rounding and normalization to put the number back into the IEEE format.



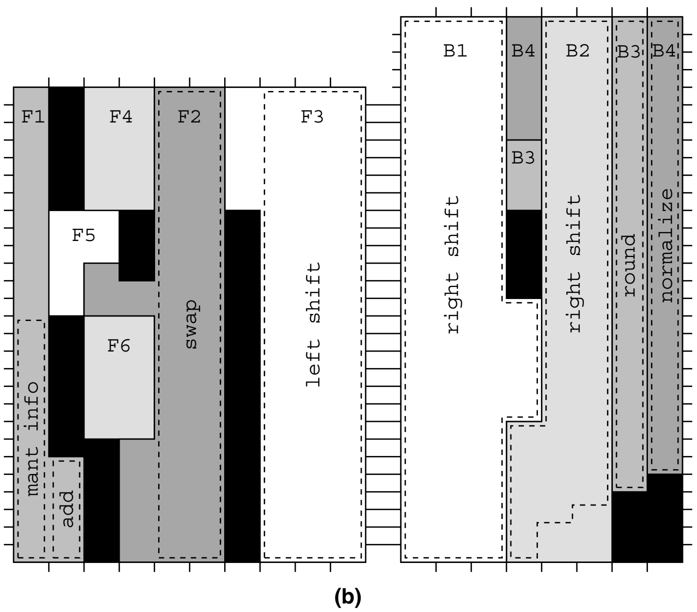


FIGURE 31.4 ■ Multiplier block (a) and multiplier layout (b) diagrams.

31.2 FLOATING-POINT APPLICATION CASE STUDIES

Floating-point applications that are appropriate to map to FPGAs differ dramatically from integer applications that are typically mapped to FPGAs. The differences can be understood by realizing that a single floating-point operation can easily consist of 30 integer operations; thus, where a 2005-era FPGA can easily implement 1000 integer operations, it is more likely that it can only implement 32 double-precision floating-point operations. Furthermore, floating-point operations are much higher latency than corresponding integer operations, which significantly affects designs.

This section considers three kernel operations implemented with double-precision floating point to demonstrate three important considerations when using floating point operations on FPGAs. The first operation is matrix multiply, which demonstrates the FPGA's ability to exploit high degrees of parallelism and to programmably manage local storage to significantly reduce the amount of external RAM bandwidth needed. The second kernel is a vector dot product, which highlights the ability of the FPGA to provide large amounts of RAM bandwidth; plus it highlights limitations introduced by the high latency of the floating-point units. The third kernel is the fast Fourier transform (FFT), which can find similar advantages in mitigating the need for memory bandwidth as the matrix multiply, but has similar limitations from the latency of the floating-point units to the dot product.

31.2.1 Matrix Multiply

The standard matrix multiply (the `DGEMM` BLAS routine) is defined as:

$$\mathbf{C}_{ij} += \sum_{k=0}^{N-1} \mathbf{A}_{ik}\mathbf{B}_{kj} \qquad (31.8)$$

The operation multiplies two matrices and adds it to a third (in place). Conceptually, this means performing the dot product of a single row of A with a single column of B and adding the result to a single point of C. Each dot product is completely independent, which means there are N^2 independent dot products. In practice, neither microprocessors nor FPGAs implement it this way because of the nature of modern memory hierarchies. In all modern systems (including FPGAs), main memory is "far away" and there is one or more caches significantly "closer."

The primary performance characteristic of matrix multiply is that it does $O(N^3)$ operations on $O(N^2)$ data. Thus, for every data item loaded from memory, it should be hypothetically possible to do $O(N)$ operations. Performing matrix multiplication as a series of independent dot products would throw away this advantage; thus, all matrix multiply implementations attempt to exploit some form of locality within the cache structure.

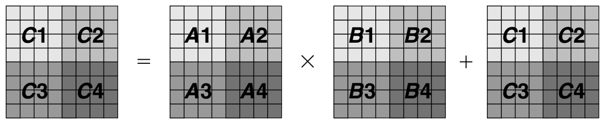


FIGURE 31.5 ■ Block decomposition of a matrix multiply.

FPGA implementation

To understand an FPGA implementation of matrix multiply, it helps to first understand how it is done on a microprocessor. To exploit (or rather compensate for) the nature of modern memory hierarchies, the typical approach to matrix multiplication on a microprocessor breaks the matrices into smaller $S \times S$ blocks [16]. A given block from each matrix is loaded into the processor, a matrix multiply is performed on the block, and partial results are stored. An example for an 8×8 matrix multiply is shown in Figure 31.5. Each matrix is broken into four regions that are 4×4. A row of these blocks is then multiplied by a column of these blocks to create a 4×4 block of the result; thus, $C1 = A1 * B1 + A2 * B3 + C1$. In the process, the partial result (a 4×4 block) is updated two times (although typically in local storage or cache).

The same approach can be used on FPGAs. After all, FPGAs and microprocessors are similar in that they have a small amount of local memory with high bandwidth and a large amount of external, slower memory. FPGAs differ, however, in that they have a drastically large number of floating-point units that should be kept fully utilized. Whereas microprocessors must supply inputs to two functional units per cycle, FPGAs must supply inputs to 32 functional units (in a 2005 FPGA).

A matrix multiply can be decomposed into a series of multiply–accumulate (MACC) operations that multiply the individual elements of a row with elements of a column and accumulate the result into one element of the final matrix. The MACC unit has a multiplier, an adder, and a feedback path. In an FPGA, 16 MACC units are operating concurrently. Unfortunately, the latency of the adder is very high (10 cycles). This means that we must keep at least 10 concurrent operations ($row \times column$ operations) in progress at all times to hide the latency of the adder. In a perfect world, each unit could work on a block of the matrix, with the concurrent operations happening on the independent row–column dot product in that block. Unfortunately, this would require far more internal memory than is available in typical FPGAs.

To exploit the parallelism available in FPGAs without exhausting the limited internal memory, we can further decompose the view of the problem. A simple way to view one block-level matrix multiplication is as a collection of S matrix–vector multiplications. As such, significantly more parallelism is obvious. Figure 31.6 shows an FPGA-based implementation that first decomposes the problem into blocks and then distributes portions of the work to multiply the two blocks as matrix–vector multiplications.

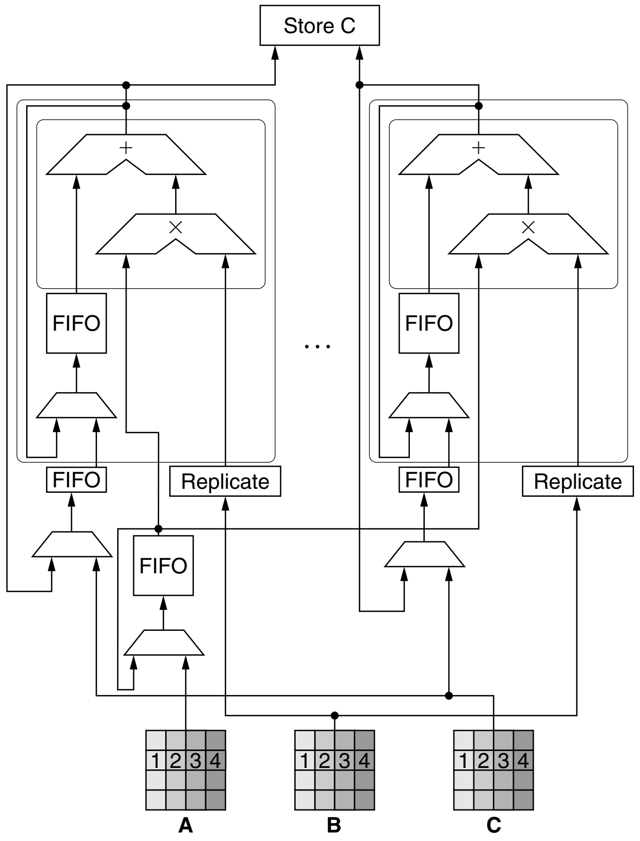


FIGURE 31.6 ■ Matrix multiply implementation.

To perform the full matrix multiplication, each matrix is decomposed into $S \times S$ blocks. In Figure 31.6, S is 4, but in practice, S is typically set large enough to cover the adder's latency (currently 10 cycles). Blocks of **B** are broken into m columns, where m is the number of MACC units (m is assumed to be 4 in the figure); thus, independent columns of a block of **B** go to each MACC unit. All the blocks of **A** are *broadcast* to all MACC units. Thus, in Figure 31.6, one column of block **B** is multplied by all four rows from the **A** block. This requires that four copies (in the general case S copies) of the **B** block be made by the replicate unit. This creates the concurrency needed to cover the latency of the adder.

Matrix **C** is managed similarly. A block of **C** is loaded and distributed in the same *order* as the block of **B**, but there is no need to replicate it. In addition, taking the example from Figure 31.5, two **A** blocks and two **B** blocks are needed for each **C** block. Thus, $A1$, $B1$, and $C1$ are loaded and used to create an intermediate product $C1-2$ that is used as the C block when $A2$ and $B2$ are multiplied. Overall, this requires no more than $6S^2$ elements of storage at 8 bytes per element. This includes two copies of each matrix block—one to operate on and one to change it from row-major to column-major order.

Performance

By nature, a matrix multiply requires at least $4N^2$ memory accesses[2] and performs $2N^3$ floating-point operations. This yields $\frac{N}{2}$ floating-point operations for each element retrieved from memory, but it assumes that two matrices (**A** and **B**) can be kept resident in the chip (processor or FPGA) for the entire operation. In the perfect scenario, the maximum sustainable floating-point rate would be

$$FLOPs = \frac{\frac{N}{2} \times BW}{8} \tag{31.9}$$

where BW is the memory bandwidth in bytes per second, N is the dimension of the matrix, and 8 bytes are required to store a double-precision floating-point number.

While this is unrealistic for all but relatively small matrices, using blocking techniques [16] to manage the local storage makes it possible to sustain a high percentage of peak performance with relatively low memory bandwidth. The result is that the matrices are fetched several times more than would otherwise be necessary. For blocks of dimension S, this yields a factor of $\frac{N}{S}$ increase in accesses to the **A** and **B** matrices, leading to $2N^2 + \frac{2N^3}{S}$ memory accesses. For large matrices, this approaches a floating-point rate of

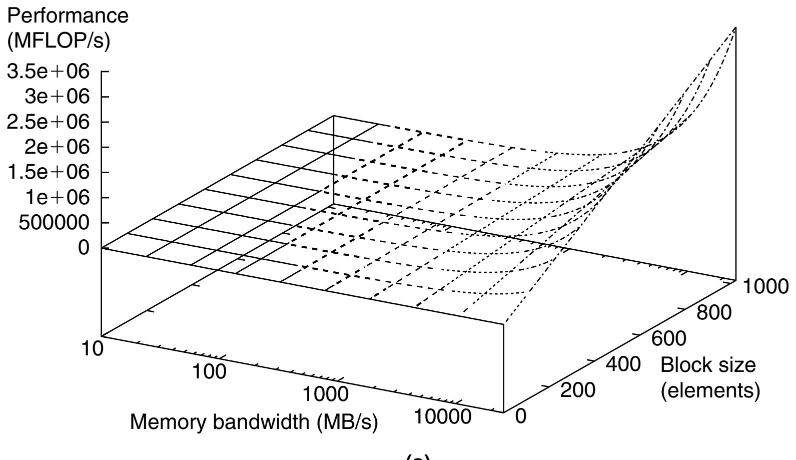
$$FLOPs = \frac{S \times BW}{8} \tag{31.10}$$

This is shown in Figure 31.7(a) as MFLOP/s versus MB/s on a log–log graph. Delineations that map memory bandwidth needs to the generation of FPGAs are provided for clarity, based on earlier work [14, 15].

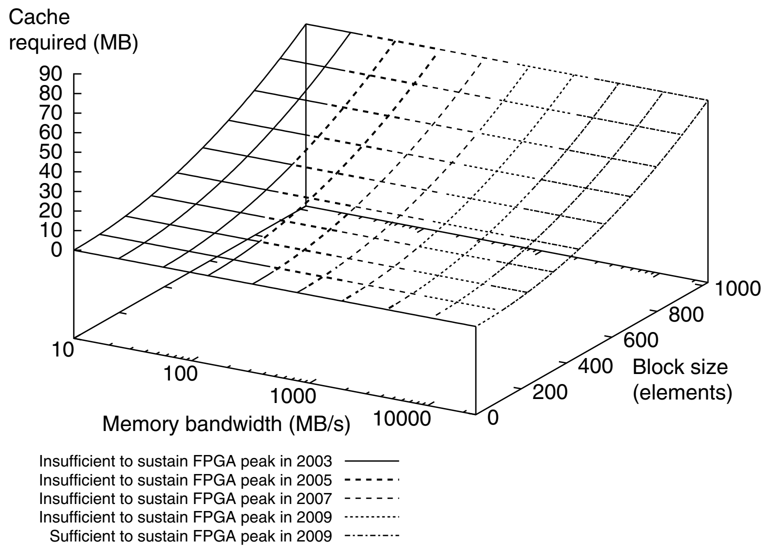
A slightly different perspective is presented in Figure 31.7(b) where the total amount of on-chip memory needed to sustain peak performance is graphed. What is notable about these graphs is the relatively small amount of memory and relatively small amount of memory bandwidth needed to sustain peak performance on FPGAs. This stands in stark contrast to modern microprocessors (2005 era) that only sustain 85 to 90 percent of peak performance on a matrix multiply using several times as much on-chip memory and off-chip memory bandwidth. This is a product of the ability of the FPGA to directly manage local storage and to separate data prefetching from computation.

We can also compare performance over time using data from 2004 [see 14,15]. Table 31.2 shows parts used for comparison. The performance of FPGAs gained rapidly on microprocessors during this era, as shown in Figure 31.8.

[2] This assumes square matrices and includes retrieving three matrices and storing one matrix.



(a)



(b)

FIGURE 31.7 ■ Maximum achievable performance versus memory bandwidth and block size (a); on-chip memory needed versus memory bandwidth and block size (b).

31.2.2 Dot Product

The standard vector dot product (the DDOT BLAS routine) is the sum of the pairwise products of two vectors, or

$$p = \sum_{i=0}^{N-1} \mathbf{x}_i \mathbf{y}_i \tag{31.11}$$

TABLE 31.2 ■ Parts used for performance comparison

Year	FPGA	CPU
1997	XC4085XLA-09	Pentium 266 MHz
1999	Virtex 1000-5	
2000	Virtex-E 3200-7	Athlon 1.2 GHz
2001	Virtex-II 6000-5	
2003	Virtex-II Pro 100-6	Pentium-4 3.2 GHz

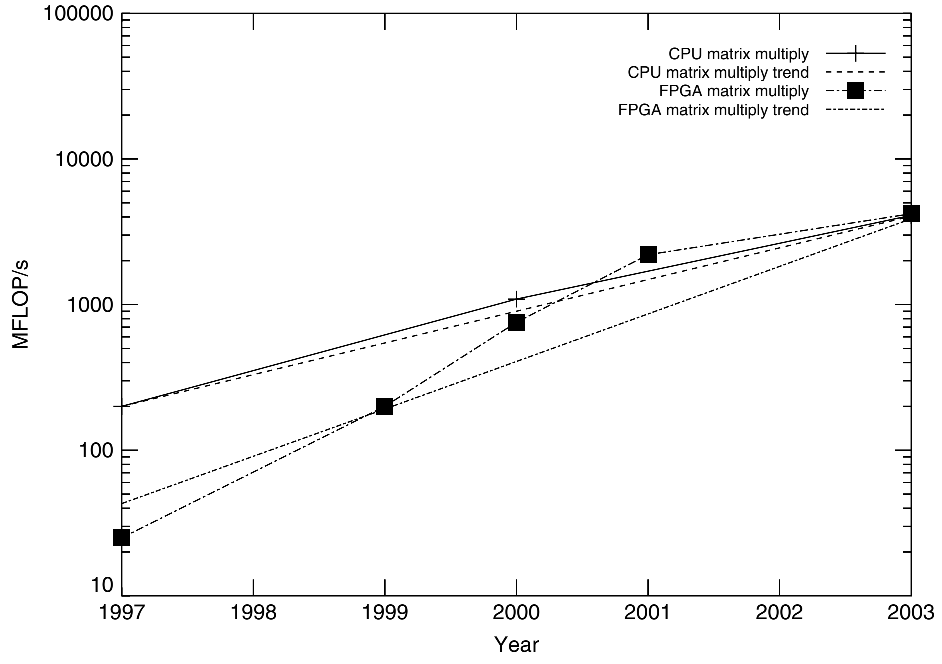


FIGURE 31.8 ■ Matrix multiply performance of FPGAs and microprocessors from 1997–2003.

which requires $2N$ memory accesses to perform $2N$ floating-point operations. This means that a double-precision floating-point number (8 bytes) must be fetched from memory for every floating-point operation that will be done. Modern processors are not built with this type of balance between memory bandwidth and floating-point capability. A processor capable of providing five GFLOP/s may only have 6.4 GB/s of memory bandwidth. Streaming problems (like this one) provide FPGAs an opportunity to excel—processors have a fixed-memory bandwidth that is configured based on a balance between the requirements for various markets and the cost of providing that bandwidth. In contrast, each board containing an FPGA can decide how many FPGA pins are used for memory bandwidth, including dedicating almost all available user pins to memory connections.

FPGA implementation

Although the potential for increased memory bandwidth on an FPGA gives it a distinct advantage, it also faces significant challenges imposed by the large number of functional units and the high latency of the units. Like many BLAS routines, DDOT is based on multiply–accumulate operations; however, it differs from many BLAS routines in that it exposes a relatively limited amount of parallelism. Where a DGEMM operation computes N^2 independent results and a DGEMV operation computes N independent results, a DDOT operation produces a single number as the final result. This means than any partial products must be reduced through a long, slow pipeline. The nature of the problem is best realized through a comparison to microprocessors.

Current microprocessors typically have a floating-point pipeline depth of four to six cycles for the functional unit running at 2 GHz or more. Obviously, we would not want every addition to depend on the previous addition, so the microprocessor can easily keep six running sums in progress and then reduce those sums to one result. This leads to several pipeline stalls in the final reduction, but the total time is a small number of nanoseconds. In contrast, FPGAs differ in three dramatic ways:

- The adder pipeline is deeper.
- Multiple MACC units are required to fully utilize high bandwidth memory.
- The clock rate is lower.

A modern FPGA would have tens of functional units with a pipeline depth of 10-cycles running at approximately 300 MHz. Assuming 16 adders with a pipeline depth of 10 cycles means that there must be 160 concurrent summations. This is impossible for short vectors and challenging even for longer vectors. Furthermore, the process of reducing these partial sums to a single result is slow and cumbersome.

To achieve reasonable performance, additional control logic is required inside and outside the multiply–add and MACC units. First, a multiplier bypass multiplexer (labeled MB) is required in the multiply–add (Figure 31.9(b)) to reuse the adder for portions of the final summation. Second, the adder has a 10-cycle latency; thus, the MACC must perform 10 concurrent operations to keep the adder pipeline filled. This requires a second feedback path (with associated control) through the FP multiplexer in the MACC (Figure 31.9(c)) to sum the 10 results. The added logic is shown with dashed lines in Figure 31.9(b) and (c).

Performance

If we work from the memory bandwidth as the typical limiting factor, the maximum sustainable floating-point rate is

$$FLOPs = \frac{BW}{8} \tag{31.12}$$

where BW is the memory bandwidth in bytes per second and 8 bytes are required to store a floating-point number. This is graphed in Figure 31.10(a)

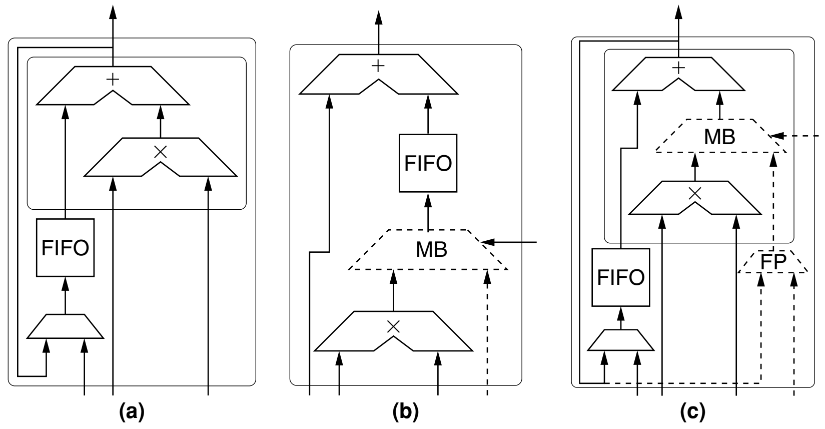


FIGURE 31.9 ■ A standard multiply–accumulate (a); a modified multiply–add for the dot product (b); a modified multiply–accumulate for the dot product (c).

on a log–log graph. Like Figure 31.8, Figure 31.10(b) compares performance projections for both FPGAs and microprocessors [14, 15]. In this case, however, the FPGA shows a much more dramatic advantage over a microprocessor. This is because large FPGAs provide sufficient I/O resources to obtain much higher memory bandwidths than commodity microprocessors offer. Since this is a memory bandwidth-limited problem, the platform with the most memory bandwidth wins.

The other notable feature of Figure 31.10(b) is that it is somewhat more crowded than the matrix–multiply comparison. This is because FPGAs face a second challenge in implementing the dot product operation: the latency of the floating-point unit. Thus, the size of the vector has a much greater impact on sustained performance on the FPGA than the microprocessor. The top FPGA line represents a scenario whereby the FPGA achieves 90 percent of its peak performance, but this requires a nearly 6000-element vector.[3] The second FPGA line shows the FPGA achieving 50 percent of peak performance by using an 800-element vector. Despite this hefty penalty, the FPGA still has a remarkable advantage (4× in 2003) over the microprocessor.

31.2.3 Fast Fourier Transform

The fast Fourier transform (FFT) is a reduced-complexity implementation of the discrete Fourier transform (DFT), which takes as input N complex numbers and returns as output N complex numbers where each of the outputs is determined by the following equation:

[3] Earlier work by Underwood and Hemmert [15] specified a 7500-element vector, but the floating-point unit latency has been optimized since then.

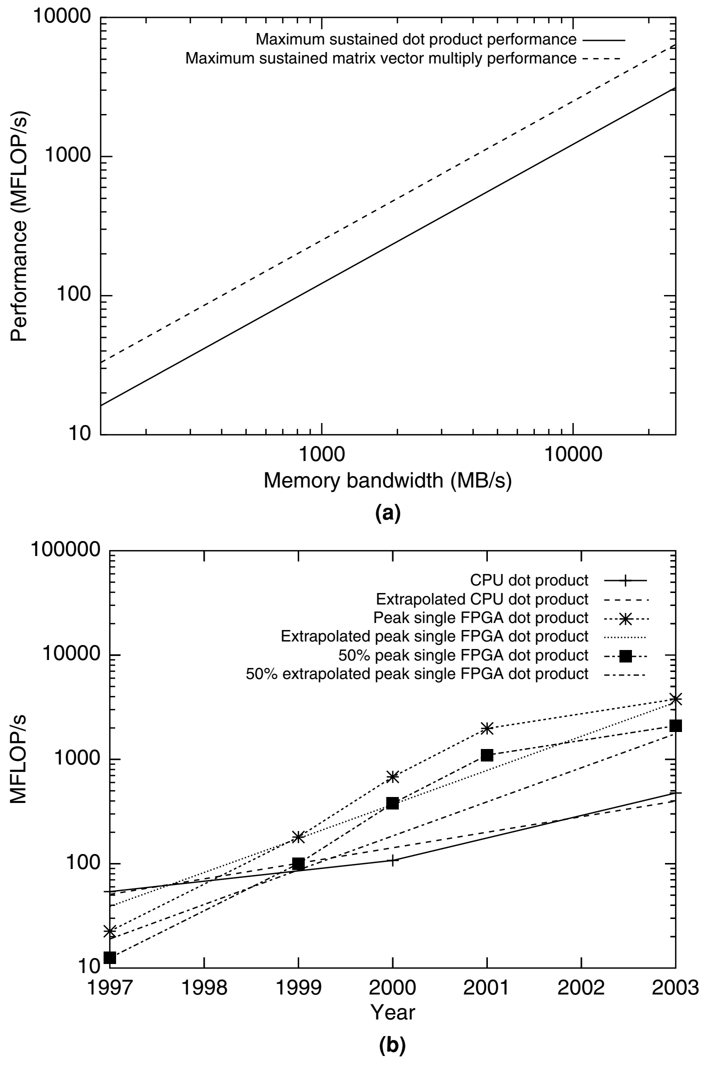


FIGURE 31.10 ■ Maximum achievable performance versus memory bandwidth (a) and dot product performance on FPGAs and microprocessors from 1997–2003 (b).

$$Y[j] = \sum_{k=0}^{N-1} X[k] W_N^{jk} \tag{31.13}$$

where $W_N^{jk} = e^{\frac{-i2\pi jk}{N}}$.

The FFT exploits symmetries in the DFT and is implemented in stages, where each stage combines r items to create r outputs. The value r is known as the *radix*. For the implementation discussed here, $r = 2$ (radix-2). For the radix-2

FFT, each stage operates pairwise on the data, although there are different formulations of the algorithm that determine how the data are combined. These operations are commonly referred to as *butterflies* and in the formulation used in this example, each pairwise operation is identical and consists of one complex multiply and two complex adds. This is shown graphically in Figure 31.11(a).

Even after selecting the formulation that gives the structure of the butterfly, there is some flexibility in the structure of the stages. The basic stage structures are shown in Figure 31.12. Both structures require data reordering, either on

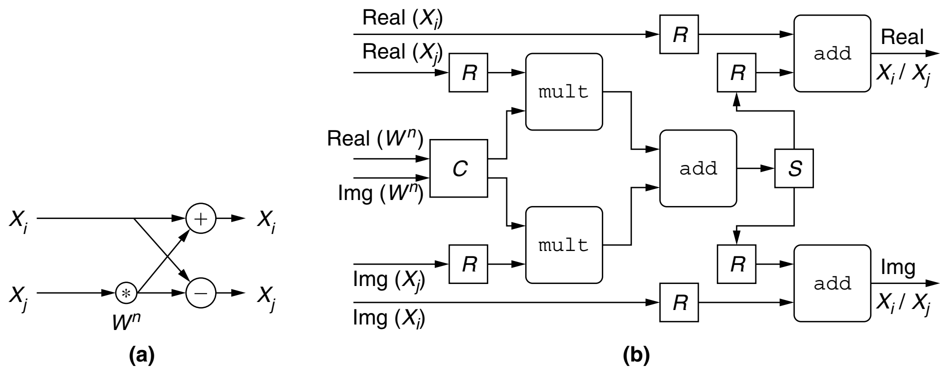


FIGURE 31.11 ■ Basic butterfly operation (a) and basic butterfly datapath (b). The component S is a switch that directs inputs to alternate outputs. The components marked as R replicate the input once and C is a crossover to facilitate the complex multiply.

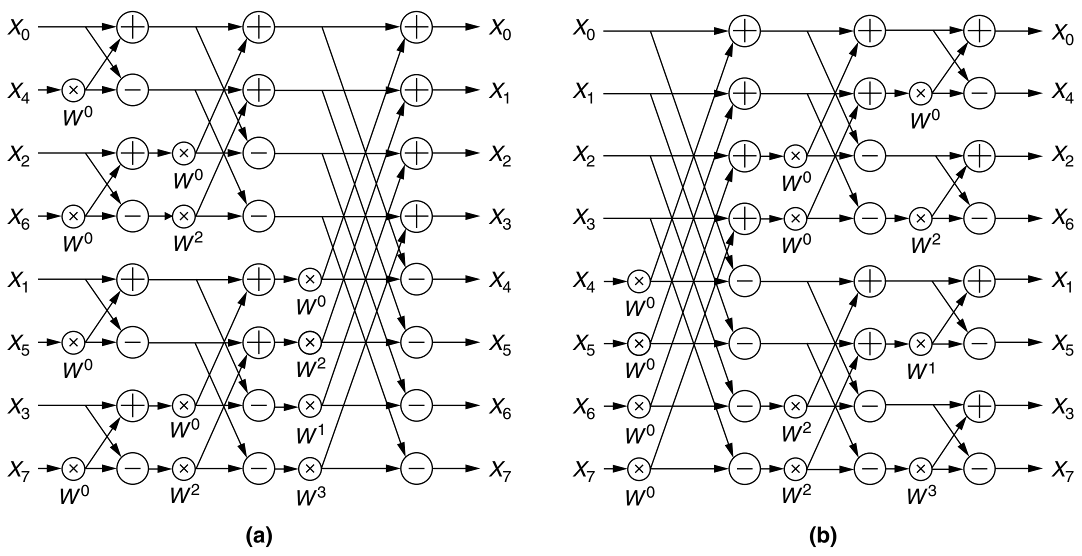


FIGURE 31.12 ■ Variations of the 8-point, radix-2 FFTs with reordered inputs (a) and reordered outputs (b).

the frontend or backend, and produce the identical set of computations (though in different orders). This example uses the ordering shown in Figure 31.12(b), because this structure provides an increasing number of independent datasets as the computation progresses. This approach is easier for implementations that use units in parallel to process data within a single stage since all interunit communication can reside at the front of the pipeline.

FPGA implementation

The butterfly computation requires four multiplications and six additions to implement one complex multiply and two complex adds. The hardware presented here uses two double-precision multiplies and three double-precision adds (see Figure 31.11(b)). Each floating-point unit is used twice for each set of inputs, which results in an average throughput of one data item per clock cycle. Although it is possible to design a datapath that accepts two data items per clock cycle, this design was chosen because it matches the available bandwidth of internal RAM blocks in the target architecture and because it provides the greatest flexibility when scaling the parallelism of the final implementation.

Parallelism in the FFT computation can be exploited in two ways: (1) pipelined units, or parallelism in the stages (S), and (2) parallel units, or parallelism (P) within a stage. Three architectures, which exploit the two types of parallelism to differing degrees, are explored.

Parallel architecture The *parallel implementation* exploits only parallelism within a stage (P). This is shown in Figure 31.13(a). In this implementation, data are read from external memory, processed iteratively, and written back to external memory. Each of the butterfly units operates on a subset of the data and is able to work independently of the other units for a large part of the computation (the datasets are completely independent after $log_2(P)$ stages).

The advantages of this architecture are that the utilization of the units is high because the pipeline depth is short. The parallel version can also take advantage of higher-memory bandwidths. The disadvantages of this architecture as implemented are that it requires a large amount of internal memory and it requires a parallelism that is a power of 2. This second restriction is important because it can limit the number of butterfly units that can be used. For example, if six butterfly units fit in an FPGA, the parallel architecture is still only able to use four.

Pipelined architecture At the other extreme, one butterfly unit can be dedicated to each of the stages of the FFT in a pipelined fashion, as illustrated in Figure 31.13(c). Data is read from memory and passed through a series of butterfly units before being written back to memory. Data delays and permutations are needed between each of the stages and between the pipelined FFT unit and DRAM memory. When the number of stages, S, that can be implemented in the FPGA is less than the number of stages needed by the FFT ($log_2(N)$), then $\frac{log_2(N)}{S}$ passes to memory are needed, with the final pass using a subset, R, of the stages. For each pass to memory, data must be read and written in a particular permutation to optimize the delay and storage requirements in the pipeline.

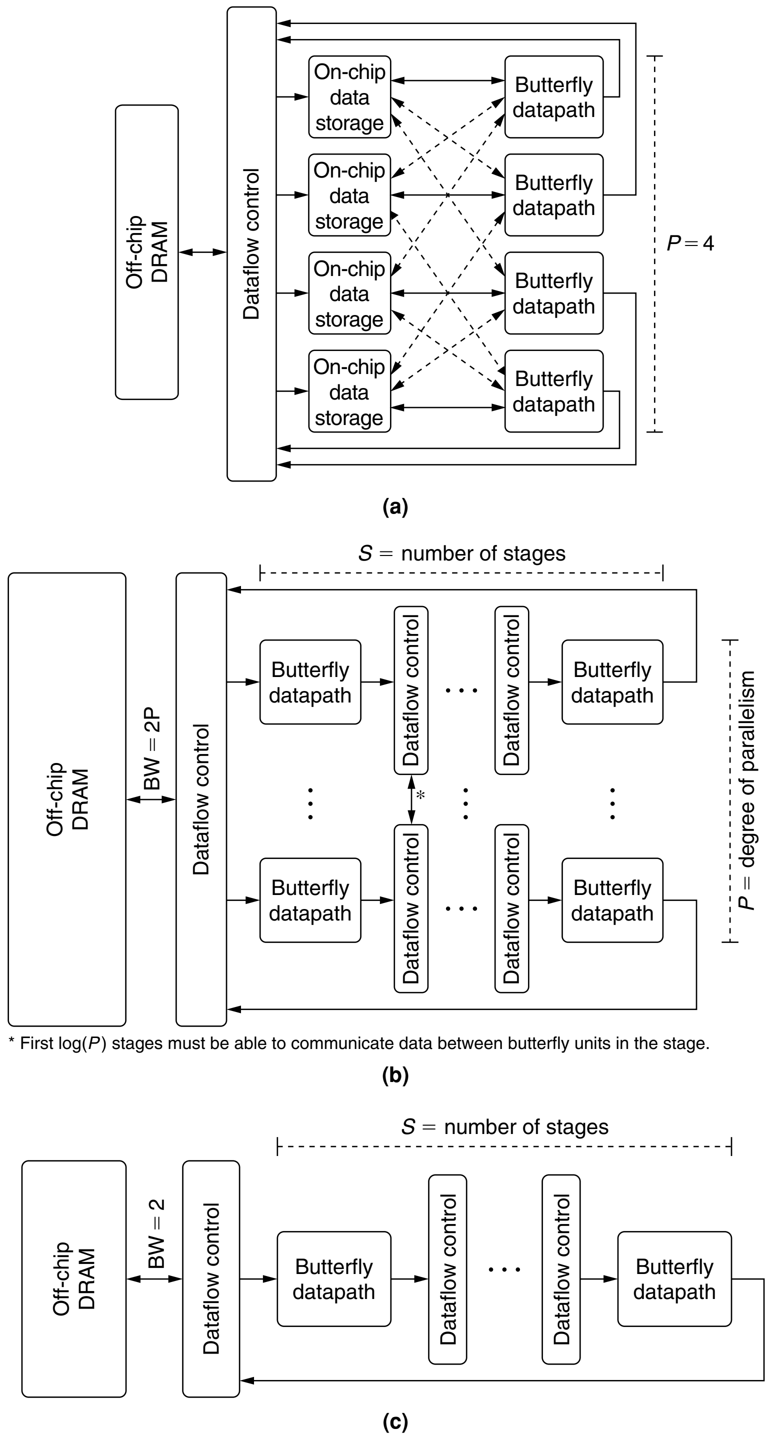


FIGURE 31.13 ■ Three architectures: (a) parallel, (b) parallel–pipelined, and (c) pipelined for exploiting parallelism in the FFT—from using all parallelism within a single stage to using all parallelism in the stages.

The pipelined architecture works well when streaming a large number of small FFTs. This is because the architecture gets good performance with minimal memory bandwidth requirements. Another benefit of this architecture is that it can take advantage of parallelism at a finer granularity than the parallel version (i.e., it can use a nonpower of 2 number of processors). However, there are some major disadvantages to this architecture. First, for single FFTs, the unit utilization is low because of the depth of the overall pipeline. Second, it is unable to take advantage of higher-memory bandwidth. Last, the buffer space required between stages for data reordering grow as 2^S, where S is the number of stages in the circuit. For a large number of stages, the memory required for buffering can easily exceed available on-chip memory.

Parallel–pipelined architecture Figure 31.13(b) is a cross between the two previous architectures. Data moves from external memory, through a set of P parallel pipelines—each with S stages—and back to external memory. The first $log_2(P)$ stages must have additional data exchange circuits (for the first pass through the pipeline) because these stages have data dependencies between the pipelines. This approach leverages the ability of the pipelined architecture to reduce bandwidth demands and the ability of the parallel architecture to tolerate shorter input vectors (as well as a wider variety of vector lengths) than the pure pipelined approach. In contrast, the parallel–pipelined hybrid has a higher bandwidth demand than the purely pipelined approach and less tolerance of short vectors than the parallel approach.

Performance

In evaluating the performance of the FFT, the floating-point operation count that is typically used is $5Nlog_2(N)$; there are $log_2(N)$ stages that each contain $5N$ computations (four multiplies and six additions for each pair of data). To determine performance, it is necessary to know how long it will take the FPGA to compute the FFT. For the parallel version, the number of cycles required to complete the FFT is given by the following equation:

$$T = \frac{32N}{BW} + BL + (\frac{N}{P} + BL)(log_2(N) - 2) \tag{31.14}$$

The first term of equation 31.14 is the time to read and then write N items based on the memory bandwidth, BW, in bytes per cycle. The usable bandwidth is limited to the number of units, P. The second term is the latency of passing through the butterfly units during the read from memory. The third term is the time to perform the iterations—using P butterfly units of latency BL for $log_2(N) - 2$ iterations, assuming that the first and last iterations are performed as part of reading and writing the data.

The pipelined and parallel–pipelined architectures share the same equation for determining the number of clock cycles required to complete the operation. The only difference is that the pipelined architecture is limited to a bandwidth (*BW*) of 2. The number of cycles to compute the FFT for these architectures is

$$T = P(S) \times \left\lfloor \frac{log_2(N)}{S} \right\rfloor + P(R) \tag{31.15}$$

$$P(J) = BL \times J + I(J) + \frac{2N}{BW} + (B-1) \times 2^J \tag{31.16}$$

$$I(K) = \sum_{i=0}^{K-1} B \times 2^i \approx B \times 2^K \tag{31.17}$$

$$R = log_2(N) \bmod S \tag{31.18}$$

Each pass, $P(J)$, through J butterfly stages (each having a latency of BL) requires the time shown in equation 31.16.

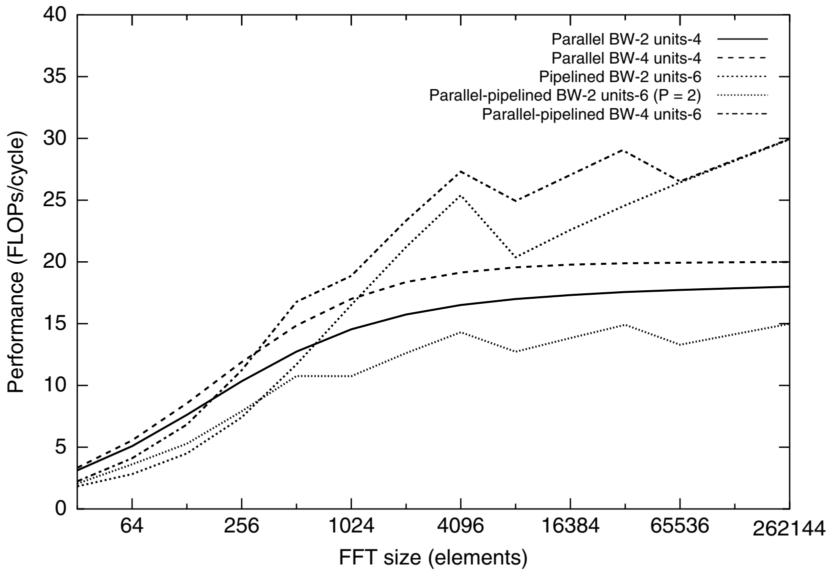
Data dependencies between the stages introduce a delay that doubles at each stage, and create a total interstage delay given by $I(K)$. Using standard DRAM memories introduces a penalty associated with the burst length (B) required to maintain full memory bandwidth to both the interstage delay and a backend reordering time. The time to retrieve the data from memory and write them back is defined by $\frac{2N}{BW}$. The final term represents the final pass through a subset of the stages, R, with the corresponding delays.

The preceding equations point to the fact that the *best* implementation for the FFT depends on many factors: memory bandwidth, size of the FFT, and size of the FPGA. The performance (in FLOPs per cycle) for a single FFT of the different FPGA architectures on a Xilinx Virtex-II Pro (a late 2005 part) are shown in Figure 31.14(a). For single short vectors, the parallel architecture provides the best performance. This is because of the high utilization of the floating-point units. For longer FFTs, all three units provide good performance, though the pipelined version requires less external memory bandwidth. Figure 31.14(b) shows that the FPGA implementations (running at 160 MHz) compare favorably to microprocessors for large FFTs.

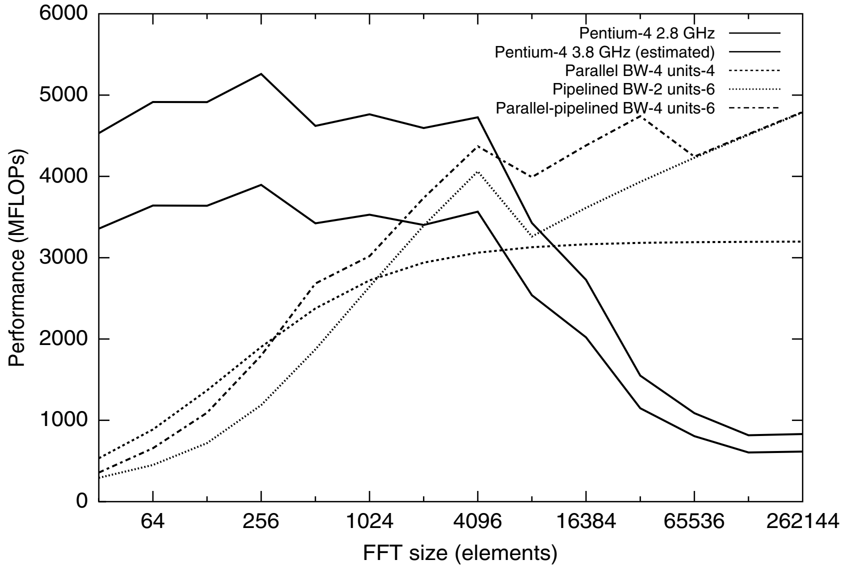
31.3 SUMMARY

Implementing floating-point arithmetic on FPGAs requires significant effort. Supporting the IEEE-754 standard poses particularly unique challenges, but much of the effort is expended in coping with the interaction between exponent logic and mantissa logic. Great care is required to minimize the latency through the unit without significantly decreasing clock rate by having two dependent carry chains in a single pipeline stage. Even with effort, floating-point operations are significantly bigger and have significantly deeper pipelines than their fixed-point counterparts. This adds additional challenges to the design of applications.

Although FPGAs can now deliver impressive performance on double-precision floating-point operations, it requires a very different mind-set from working with fixed-point arithmetic. Increased operation latency leads to a need to find more parallelism to exploit in paths with the cyclic data dependencies typical of



(a)



(b)

FIGURE 31.14 ■ A comparison of performance for different FFT architectures in FLOPs per cycle (a) and a comparison of FFT implementation on FPGAs and CPUs (b).

iterative solutions. Simultaneously, the increased size of a single operation reduces the portion of a given dataflow graph that can be implemented directly and pushes a designer toward more iterative solutions. The dot product is an excellent example because it is forced to reuse adders to compute a summation that would typically be done as a tree of adders in a fixed-point solution. The result is that only longer vectors make sense.[4]

Even simple feedforward paths incur a penalty from the high latency of the floating-point units. The FFT provides an example whereby the latency of a single butterfly path can approach the length of a short vector. Thus, if the FFT implementation in an FPGA is not used for a long FFT or a series of short FFTs, it cannot offer competitive performance.

There are, however, floating-point kernels that offer abundant parallelism for the FPGAs to exploit. Matrix–multiply (`DGEMM`), for example, is an N^3 operation with minimal data dependencies. Similar things can be said about LU solvers, which form the basis of the traditional Linpack benchmark [4]. Three-dimensional FFTs are another example in which hundreds of one-dimensional FFTs can be carried out simultaneously.

References

[1] P. Belanovic, M. Leeser. A library of parameterized floating-point modules and their use. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2002.

[2] M. deLorimier, A. DeHon. Floating point sparse matrix-vector multiply for FPGAs. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2005.

[3] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, D. Poirier. A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2002.

[4] J. J. Dongarra. The linpack benchmark: An explanation. *First International Conference on Supercomputing*, June 1987.

[5] Y. Dou, S. Vassiliadis, G. Kuzmanov, G. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2005.

[6] B. Fagin, C. Renard. Field-programmable gate arrays and floating point arithmetic. *IEEE Transactions on VLSI* 2(3), 1994.

[7] A. A. Gaar, W. Luk, P. Y. Cheung, N. Shirazi, J. Hwang. Automating customisation of floating-point designs. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2002.

[8] G. Govindu, S. Choi, V. K. Prasanna, V. Daga, S. Gangadharpalli, V. Sridhar. A high-performance and energy-efficient architecture for floating-point based LU decomposition on FPGAs. *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW)*, April 2004.

[9] G. Govindu, L. Zhuo, S. Choi, P. Gundala, V. K. Prasanna. Area and power performance analysis of a floating-point based application on FPGAs. *Proceedings of the Seventh Annual Workshop on High-Performance Embedded Computing,* September 2003.

[10] K. S. Hemmert, K. D. Underwood. An analysis of the double-precision floating-point FFT on FPGAs. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005.

[4] A long series of short vectors can also be made to work using an appropriate architecture.

[11] IEEE Standards Board. *IEEE Standard for Binary Floating-Point Arithmetic*. Technical Report ANSI/IEEE Std. 754-1985, The Institute of Electrical and Electronics Engineers, 1985.

[12] L. Louca, T. A. Cook, W. H. Johnson. Implementation of IEEE single precision floating point addition and multiplication on FPGAs. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[13] N. Shirazi, A. Walters, P. Athanas. Quantitative analysis of floating-point arithmetic on FPGA based custom computing machines. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[14] K. D. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2004.

[15] K. D. Underwood, K. S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[16] R. C. Whaley, A. Petitet, J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 27(1–2), 2001.

[17] L. Zhuo, V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. *18th International Parallel and Distributed Processing Symposium*, April 2004.

[18] L. Zhuo, V. K. Prasanna. Sparse matrix–vector multiplication on FPGAs. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, February 2005.