Electronic Design

NASSIM ABDERRAHMANE, AI Product Manager,
Menta, https://www.menta-efpga.com/

Maximize Flexibility in SoC Design with eFPGAs

An eFPGA is an FPGA that's embedded into an ASIC to provide one or more programmable-logic fabrics for flexibility and cost/performance benefits.

Designers continue to face greater challenges in the development of advanced embedded systems. Functionality and connectivity create layers of added integration and complexity that often make it difficult to provide an optimum logic architecture to manage a given system, especially if it's a system-on-chip (SoC). Let's explore the anatomy of an eFPGA and how to achieve the best optimization of silicon resources with the maximum amount of flexibility.

If we strip an FPGA into its component parts, we find primarily a core, typically containing the logic, memory, other macros such as DSP, interconnect grid of wires, switches, and other compute elements, arranged in a grid or matrix. The I/O ring usually contains high-speed interfaces to the physical world, such as SerDes, LVDS, CMOS, and TTL interfaces. An eFPGA is essentially just the core of an FPGA without the I/O ring.

When it comes to the integration and functions required in an IC design, we find that many IP blocks must be integrated. Typically, there would be things such as a microprocessor (or 16!), PLLs, memory controllers, various buses, cache controllers, and the like. However, with the larger integration of logic functions and IP blocks, there needs to be a way of changing or updating an IC after production in a cost-effective manner.

Advanced IC Designs Replace Board-Level Systems

We're entering an age where lots of legacy PCB-mounted ICs are getting mopped up and placed into a single monolithic IC or chiplets as an SoC. One issue is that IC design teams could miss a market area or a timeline if they don't incorporate the correct features, or end up finding bugs in portions of the design. An FPGA has been traditionally used to either prototype an IC or add flexible functionality to a design on a PCB, or to just integrate all of the simpler I/O and control functions (and fix/add any last-minute requirements).

However, since the advent of higher levels of integration, we're starting to hit massive bandwidth bottlenecks and I/O constraint issues, such as the inability to physically bond out enough I/Os in the space given on an IC package. We're seeing BGA packages of 2048+ I/Os at 0.5-mm pitch and below on a PCB, which causes a whole number of issues. These include physical pin density, routing congestion, layer counts with lots of micro-vias and stackup/lamination issues, and speed issues leading to signal-integrity problems such as simultaneous switching noise and crosstalk.

In addition, I/Os burn power and consume lots of silicon real estate, while requiring separate power rails for all of the different I/O standards. Though every effort has been made to ensure the accuracy of information contained herein, features, specifications, and technical information are subject to change without notice.

With all of this in mind, we must think about how to push past design limitations going forward. Offering a solution to these issues, an eFPGA is a matrix of LUTs/memory/DSP/compute elements that can be configured to be any size within the limits of the semiconductor die and real-estate requirements. It also provides a nearly limitless number of I/O interface pins, as many as allowed by the semiconductor design rules.

How an eFPGA Can Empower a System's Design

It's quite common to see a square design with 1,000 I/Os per side, for a total of 4,000 user I/O's. An implicit advantage of using an eFPGA is that you can run at internal IC speeds, with no I/O bound interface limitations through LVDS, SerDes, CMOS, and others, along with very wide bus interfaces.

This greatly benefits designers, as they can run at system speeds on the IC die and have large buses to push data in or out of a definable logic/compute element. Such definable logic can be designed after the IC has gone for tapeout and be updated in the field or at production time, customizing

1. Menta's eFPGA consists of smaller elements, with the I/Os and components that make up the Matrix.

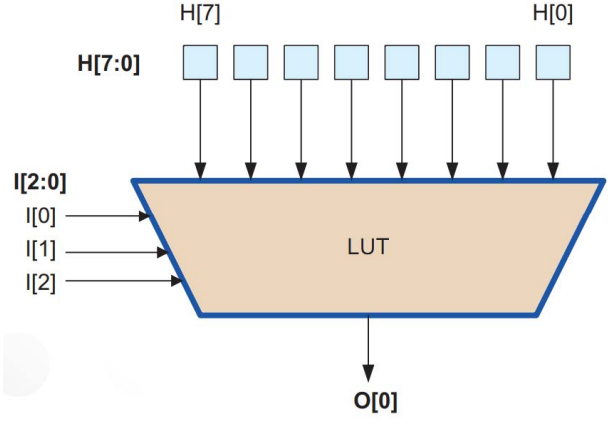
the product as required.

For example, *Figure 1* shows Menta's eFPGA that's composed of smaller elements, with the I/Os and the components that make up the Matrix. The I/O Block contains optional registered I/Os with D-FFs so that the design can be clocked and timing can be closed at the interface level.

The embedded Custom Block (eCB) is a customer-specific definable function or hard macro that can be integrated into the matrix. This might be some definable AI or convolution function that's proprietary, or a mixed signal Digital/Analog block with a Digital interface (which is a feature of Menta's product offering).

The Configuration I/O is the interface in which users program the eFPGA bitstream with their customizable logic, and the DFT I/O is the Design For Test interface that enables the eFPGA to be fully checked for design flaws after manufacturing. There's also an embedded Logic Block (eLB). The DSP block is a DSP element containing a pre-processing FIR/IIR filter block. And the usual Multiply/Accumulate functionality can be cascaded to make up larger DSP elements.

The embedded Memory Block (eMB) is an instantiated memory from the silicon foundry or a third party and can easily be integrated into the eFPGA definition. In addition, an Interconnect grid of wires and switches connect these elements together to make up the defined circuits and function. This isn't shown in *Figure 1* for brevity, but it can be assumed to be embedded where the grid lines are located. (Definable in an eFPGA means at design time/specification time and not after production.)



2, An example of a simple LUT.

Addressing Programmable Logic

When it comes to programmable logic, the key is in the lookup tables (LUTs), which are part of the eLBs. A LUT is really a set of multiplexers that allows for the individual addressability of any input bit. It can create any combinational logic required by the user. This is done by hardwiring the inputs of a LUT to a predetermined value and then using the inputs to the LUT to provide the correct logic output by addressing the appropriate individual bit.

Figure 2 shows an example of a simple LUT that has three inputs I[2:0], eight hardwired inputs H[7:0], and one output O[0]. If we create a Truth Table *(Fig. 3)* of this, whereby we want to implement an AND gate (O[0] = I[2].I[1].I[0]), we can see that by addressing the element I[2:0] = 3'b111, we return bit H[7] of the LUT.

Figure 4 looks at a more complicated function: O[0] = (/I[2]./I[1])+I[0].

As can be seen in *Figures 2, 3, and 4*, a LUT can be configured to map to any logic function as required. In essence,

I[2]	I[1]	I[0]	H[7:0]	O[0]
0	0	0	00000000	0
0	0	1	00000000	0
0	1	0	00000000	0
0	1	1	00000000	0
1	0	0	00000000	0
1	0	1	00000000	0
1	1	0	00000000	0
1	1	1	10000000	1

3. The LUT's Truth Table.

I[2]	I[1]	I[0]	H[7:0]	O[0]
0	0	0	00000001	1
0	0	1	00000010	1
0	1	0	00000000	0
0	1	1	00001000	1
1	0	0	00000000	0
1	0	1	00100000	1
1	1	0	00000000	0
1	1	1	10000000	1

4. A more complicated function for the Truth Table.

a lookup table is a one-data-bit ROM configured to create logic functions. The LUT can be mapped to any Truth Table mapping function. The inputs to the LUT are the logic function's inputs, and the contents from the lookup from the ROM give the logic equivalent outputs.

In this example, we only have a three-input LUT. However, in more complex devices, there are four-, five-, six-, seven-, and even eight-input LUTs that make the LUT sizes increase with inherently longer delays.

LUTs have various optimal sizes, which are chosen by the manufacturers of the eFPGA and FPGAs to best fit the tradeoff between the logic density requirements and speed. The "H" bits are known as "configuration bits" since they configure the function of the eFPGA's logic mapping.

The LUT is the combinational logic part of the puzzle in the eLB. However, another part of the puzzle is the D-FF, because most designs are synchronous (clocked) designs. All that's required is to add a D-FF to the output of the LUT block (O[0]) so that the logic can be registered.

There are other elements within the eLB, but that requires much more explanation and isn't relevant to this level of description. The D-FFs have the ability to be SET and RESET and the choice of which edge is used to clock can also be assigned. More 'configuration bits'!

At this point, you should understand how an eFPGA (and FPGA) gets its programmability. One other important element is the Switch Box. It's basically a cross-point switch that can connect any input to any output with configuration bits to program its function.

For example, a 16 I/O switch box would have four I/O on either side of a square and be able to route any of the points to another point. Switch Boxes are custom-crafted to the architecture of the eFPGA/FPGA with various optimizations for speed/power/routability. The Switch Boxes are wired to metal lines and to the inputs and outputs of the eLBs (LUTs, etc.) to provide a routing path between the LUTs, Memory, DSP, and Compute elements.

Arranging all of these elements on a grid of wires, the LUTs, Memory, DSP, and Compute elements as well as the Switch Boxes make it possible to build an eFPGA. There's also the storage of the "Configuration bits," which is where manufacturers differ in their strategies. Most eFPGA and FPGA providers use SRAM bitcells, while Menta decided to use D-FFs.

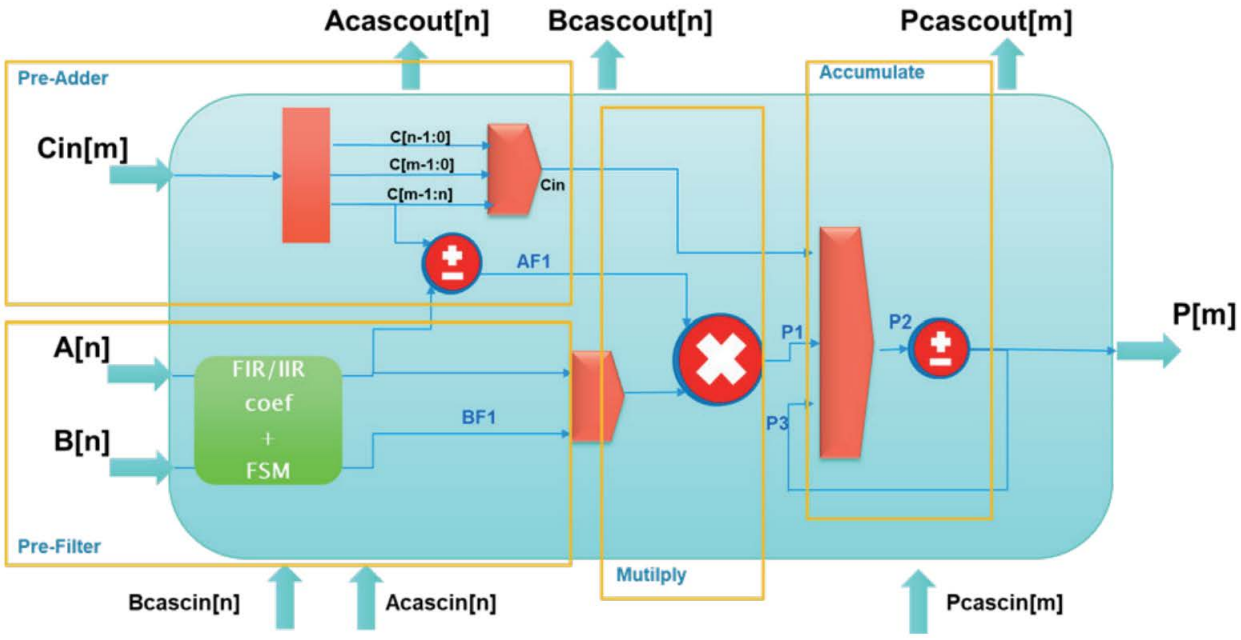
There are some very good reasons for using D-FFs. They're typically more radiation tolerant due to their nature as a master-slave system—if a stray particle hits one of the elements, it's unlikely to cause corruption. An SRAM cell has a dense array, susceptible to a stray particle that can flip the value of a bit.

As process geometries shrink, this is becoming more of an issue due to the thickness of the oxides and metal not being able to disperse the stray particles. Since Menta has standard D-FF at the heart of its technology, there's no reason why a user can't use triple module redundant (TMR) techniques with the understanding that an area and speed penalty occurs when implementing a design this way.

Lastly, SRAMs require proprietary libraries from a silicon foundry which, most of the time, forces eFPGA providers to design their own bitcells. This creates another path for potential issues in the design.

The configuration SRAM needs some fairly complex logic to shift data bits through them as well as address them. SRAM cells are also more prone to manufacturing errors: They push the boundaries of semiconductor manufacturing technologies because they use the highest packing density techniques (very small geometries). On a side note, yielding SRAM requires redundancy, which is cumbersome for eFPGAs. All in all, and thanks to architecture patents Menta's D-FF implementation doesn't lead to a silicon area penalty compared to SRAM-based eFPGA IPs.

Another unseen advantage of using D-FF is that Menta's eFPGA may be synthesized very quickly into a customer's product. All of the elements are standard cells; you can use any semiconductor foundry library and synthesize your custom eFPGA IP into your design quickly.



5. Shown is a DSP block and its constituent parts.

Other vendors provide "hard" macros, and they must pre-qualify their IP blocks in a particular process/foundry geometry. This can take lots of time since they have to do layout, a netlist comparison, and timing and parasitic extraction before being able to commit a design to a particular process/foundry geometry.

Menta can provide the eFPGA verilog/VHDL netlist immediately to the design team so that they can start integrating and floorplanning, as well as optimize their IP requirements (eLB/DSP/MEM/eCB mix). Obviously, there are guidelines on how the layout is done; Menta provides this as part of the deliverables of what's called "soft" IP. The company can also provide a service for layout of the IP if needed.

DSP Blocks

The DSP blocks in eFPGAs comprise a synchronous multiply-accumulate architecture that's wholly based on logic. The iterative nature of these devices can be incredibly fast, and unlike a CPU/GPU, they don't need instruction streams to direct how the iterations are performed. In a CPU/GPU architecture, there must be a Fetch-Execute-Store process where the CPU/GPU fetches an instruction, decodes the instruction, performs the operation in an ALU, and then stores the data back into memory or registers.

Doing DSP operations in a CPU/GPU comes with a heavy cost, which basically involves the memory loads and stores along with instruction decodes. A CPU doesn't lend itself to a streaming DSP capability very well. The way in which hardware-based DSP architectures differ is that they perform bitwise mathematics in native standard-cell gates.

Figure 5 shows a DSP block and its constituent parts. On the left are the inputs A and B as bit vectors to do the math on, as well as Cin as a supplemental Carry input. We'll ignore the green "FIR" box for now (assume the signals pass straight through it).

A and B will eventually be passed onto a multiplier as AF1and BF1, which will produce an intermediate product—let's call it P1. P1 is then taken into a Math block, where it can be added to P3; P3 was the last result (i.e., a value held in an accumulator). The output of P2 can have its sign changed, too, to provide the desired output.

In effect, the simple math function described so far is:

$P = (A * B) + LAST\_VAL$, or

$P = (A * B) - LAST\_VAL$

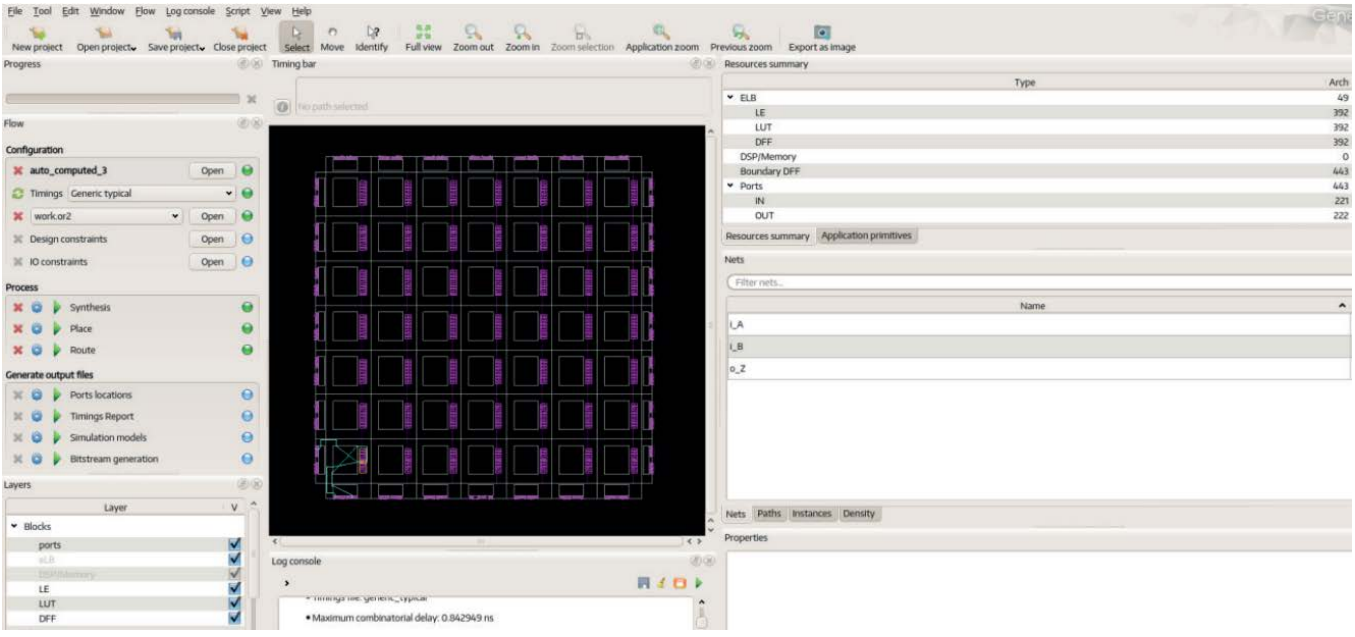
Regarding the Carry input Cin, by adding a carry input to the addition, we can take into account overflows from other previous calculations. Because the Carry can be shifted, we're also able to change the carry's significance to the input. This Cin is called a "Pre-adder" since it can have different weightings/significance.

When looking at the Pre-adder block, you'll notice that in the path before AF1, there's an arithmetic block with unary sign "inversion" capabilities. This means that we can effectively change the input on Cin or A to become negative.

As a result, we can add the new math functions to the capability by the use of the pre-adder. The equation becomes:

$P = ((\pm A \pm Cin) * B) \pm LAST\_VAL$

Not to overcomplicate things, but the Pre-adder has more capabilities than I've shown here. However, that would take us beyond the scope of this article. The FIR/IIR engine is a



6. Many designers are accustomed to automation in their workflow using TCL/TK scripts.

patented Menta technology that provides finite impulse response (FIR) or infinite impulse response (IIR) filtering, a common requirement for DSP algorithms. It's a pre-process that can perhaps be used for harmonic or spur removal in the original signal; it comes for free as part of the DSP block. It can be bypassed if not needed.

The Menta DSP Block also provides a "Cascade" ability that enables the DSP blocks to be daisy-chained together to provide a larger/wider "DSP" elements. The obvious penalty is that the daisy-chained DSP function has been moved on to another pipeline stage. The application side has a wide number of use cases for DSP mathematics. This includes:

- Convolution
- Filtering
- Modulation and demodulation
- Mixing/summing/decimation
- Image processing (kernel mathematics)
- Matrix mathematics (real and complex number math is capable in the DSP block!)
- FFT, DFT, Correlation, etc.
- SDR (software-defined radio)—I/Q schemes

Tools and Workflow

Once an eFPGA IP has been designed, you'll want to do some FPGA work for your application. This would include writing RTL (the application at hand), synthesizing the logic, placing and routing the design and extraction of timing, and static timing analysis.

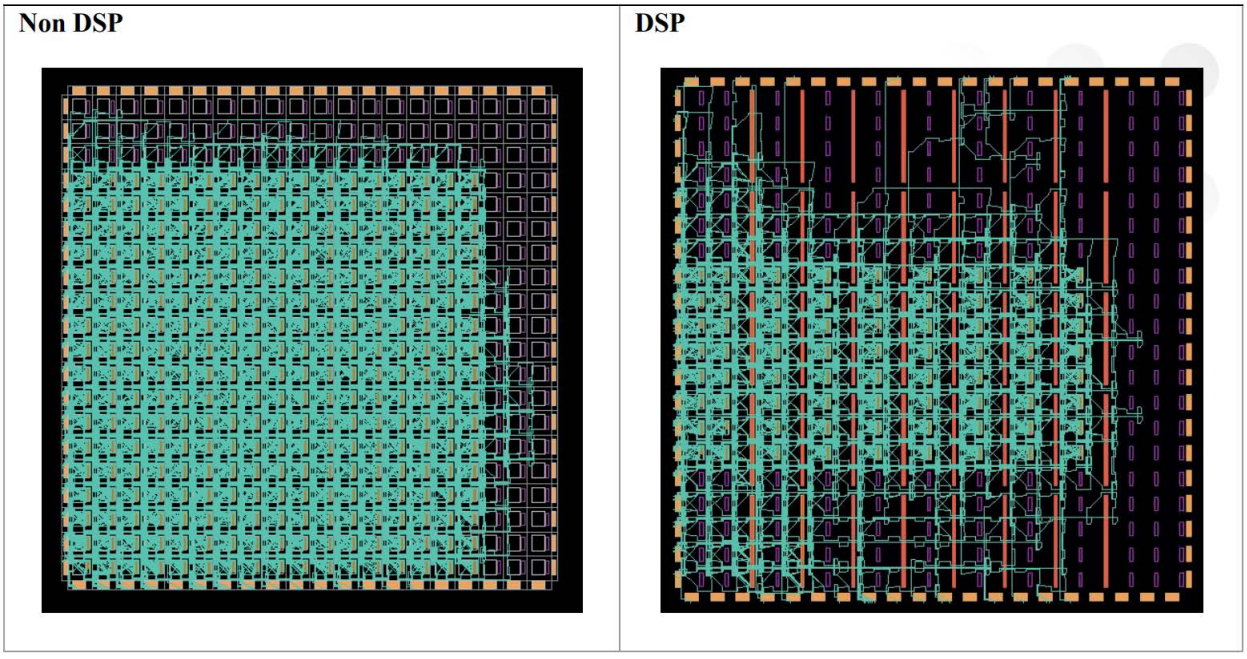
Menta has a set of internally developed tools called Origami, a full development suite that provides what's needed to implement a design. This suite of programs is integrated under an umbrella GUI, but it can also be run individually at the command prompt using TCL/TK scripts.

Many designers are accustomed to automation in their workflow using TCL/TK scripts. Origami can be easily integrated into this flow *(Fig. 6)*. Origami can read all RTL languages (Verilog, System Verilog, and VHDL in all of their defined standards), perform automatic hierarchy recognition, do synthesis into an internal netlist, perform optimized place and route (based on your I/O mapping), and generate a mapping file for the configuration bitstream data.

In addition, Origami does static timing analysis and will provide a back-annotated SDF netlist for post place-and-route simulation. The timings are extracted from sign-off EDA tools for the eFPGA IP design, thanks to the third-party standard cell libraries, and they remain under the full control of the customer.

Origami can optimize logic to also use DSP elements where it sees fit, and the user can choose what kind of optimization they want to map to DSP blocks to implement designs highly optimized for speed where math is concerned. Once Origami completes the workflow a bitstream file is produced and this bitstream would then be written to the programming interfaces of the eFPGA to configure the eFPGA for its end function as per the user's design/application. This enables the host to store multiple bitstreams with different functions and these may be chosen and written (downloaded) to the eFPGA as required providing a choice of hardware functions.

By carefully selecting your mix of DSP and LUTs, you can



7. By carefully selecting your mix of DSP and LUTs, you can achieve better packing ratios and functional speed.

achieve better packing ratios and functional speeds *(Fig. 7)*. This optimization process is iterative and depends heavily on the design. This is where you need to engage with the eFPGA vendor.

eFPGA Considerations

When choosing an eFPGA, you should be aware of your range of applications and what you want to achieve. eFPGA vendors provide a vehicle where you can add flexibility to your custom IC, but you must be aware of the limitations of what's achievable with an eFPGA. Clock speeds will be less than raw standard-cell ASIC design and porting a portion of your current ASIC IP (Verilog and VHDL) will likely require some custom work.

Typically, we find that ASIC IPs use gated clocks, which is a big NO for FPGAs, so expect some handcrafting. Optimization of code can significantly reduce logic use, too, because an eFPGA/FPGA has larger blocks to which the logic is mapped. FPGAs map into LUTs and D-FFs, whereas a custom ASIC maps into standard or full custom cells.

For DSP applications, you would need to carefully craft your DSP code and/or define an optimal architecture to take advantage of the DSP elements if you wish to get the highest performance/utilization of elements. You must also be aware of the programming interface and be able to send in a bitstream from your design (ROM, CPU, etc.), with a solid design-for-test plan and strategy.

The benefits of an eFPGA are flexibility, design reuse, the ability to make post tapeout changes to fix bugs or change algorithms, and the possibility for a customer to make a more general ASIC that can be customized for different products.