FPD'96 -- Fourth Canadian Workshop of Field-Programmable Devices
May 13-14, 1996, Toronto, Canada

Dynamically Programmable Gate Arrays: A Step Toward Increased Computational Density

André DeHon
andre@mit.edu
(617) 253-5868
MIT Artificial Intelligence Laboratory
NE43-791, 545 Technology Sq., Cambridge, MA 02139
FAX: (617) 253-5060

Abstract

Field-Programmable Gate Arrays are interesting, general-purpose computational devices because (1) they have high computational density and (2) they have fine-grained control of their computational resources since each gate is independently controlled. The earlier provides them with a potential 10× advantage in raw peak performance density versus modern microprocessors. The later can afford a 32× advantage on random bit-level computations. Nonetheless, typical FPGA usage seldom extracts this full density advantage. DPGAs are less computationally dense than FPGAs, but allow most applications to achieve greater, yielded computational density. The key to unraveling this potential paradox lies in distinguishing instruction density from active computing density. Since the storage space for a single instruction is inherently smaller than the computational element it controls, packing several instructions per computational unit increases the aggregate instruction capacity of the device without a significant reduction in computational density. The number of different instructions executed per computational task often limits the effective computational density. As a result, DPGAs can meet the throughput requirements of many computing tasks with 3-4× less area than conventional FPGAs.

1 Computational Area

"How big is a computation?"

The design goal for "general-purpose" computing devices is to develop a device which can:

- implement desired computational tasks
- perform the computation at the desired latency or throughput
- realize the implementation at minimal cost – usually silicon area

As device designers we are concerned with the area which a computational element occupies and its latency or throughput.

We know, for example, that a four input Lookup Table (4-LUT) occupies roughly 640Kλ^2 (*e.g.* 0.16mm^2 in a 1μ CMOS processor ($\lambda = 0.5\mu$)) [1] [9]. Thus, we get a 4-LUT density of 1.6 4-LUTs per one million λ^2 of area.

At the same time, we notice that the *descriptive density* of 4-LUT designs can be much greater than the 4-LUT density just observed. That is, the LUT configuration is small compared to the network area so that an idle LUT can occupy much less space than an active one.

For illustrative purposes, let us assume that it takes 200 bits to describe the configuration for one 4-LUT, which is typical of commercial FPGA devices. A 64Mb DRAM would hold 335K such configurations. Since a typical 64Mb DRAM is 6Gλ^2, we can pack 56 4-LUT descriptions per one million λ^2 of area – or about 35× the density which we can pack 4-LUTs. In fact, there is good reason to believe that we can use much less than 200 bits to describe each 4-LUT computation [3], making even greater densities achievable in practice.

Returning to our original question, we see that there are two components which combine to define the requisite area for our general-purpose device:

1. N_d – the total number of 4-LUTs in the design – the descriptive complexity
2. N_a – the total number of 4-LUTs which must be evaluated simultaneously in order to achieve the desired task time or computational throughput – the parallelism required to achieve the temporal requirements

In an *ideal* packing, a computation requiring N_a active

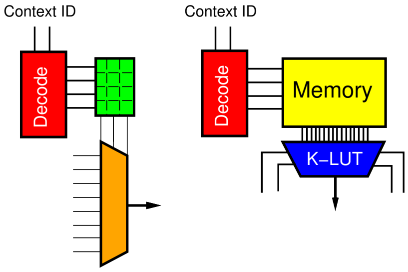


Figure 1: DPGA LUT and Interconnect Primitives

compute elements and N_d total 4-LUTs, can be implemented in area:

$$A_{compute} = N_a \cdot A_{LUT} + N_d \cdot A_{LUT\_config\_mem} \quad (1)$$

In practice, a perfect packing is difficult to achieve due to connectivity and dependency requirements such that $N'_d > N_d$ configuration memories are required.

2 DPGAs

When $N_d > N_a$, it is advantageous to associate multiple, ideally $\frac{N'_d}{N_a}$, LUT descriptions with each active LUT. Dynamically Programmable Gate Arrays (DPGAs), do just that, packing several configuration memories for each active LUT and switching element (Figure 1).

From our own experience with the DPGA [9]:

$$\begin{aligned} A_{LUT} &\approx 560\text{K}\lambda^2 \\ A_{LUT\_config\_mem} &\approx 20\text{K}\lambda^2 \end{aligned}$$

The base LUT area is generally consistent with other FPGA implementations. Our $A_{LUT\_config\_mem}$ is based on a DRAM cell design which is $600\lambda^2$ per memory bit, making the memory cells $2\times$ smaller than an SRAM implementation. A static memory would be closer approximated as:

$$A_{LUT\_static\_config\_mem} \approx 40\text{K}\lambda^2$$

An aggressive DRAM cell should realize a $300\lambda^2$ memory cell, making:

$$A_{LUT\_dynamic\_config\_mem} \approx 10\text{K}\lambda^2$$

In all these cases the configuration size is at least an order of magnitude smaller than the active LUT:

$$14 \leq \frac{A_{LUT}}{A_{LUT\_config\_mem}} \leq 56$$

```
if (c >= 0x30 && c <= 0x39)
    res = c-0x30;
else if (c >= 0x40 && c <= 0x46)
    res = c - 0x40 + 10;
else if (c >= 0x60 && c <= 0x66)
    res = c - 0x60 + 10;
else
    res = 0;
```

Table 1: ASCII Hex→Binary Task Description

Single context devices running designs where the descriptive complexity, N_d, is large compared to the number of requisite parallel operations, N_a, are, consequently, much larger than they need to be since they require $N_d - N_a$ more active LUTs than are necessary to support the task at the desired computational speed.

3 ASCII Hex→Binary: Extended Example

Of course, there are many issues and details associated with multicontext computing devices. In this section, we will walk through a specific design example in order to make the previous discussion more concrete and in order to motivate additional issues. We will take an ASCII Hex→binary converter as our example.

Figure 1 describes the basic conversion task. Assuming we care about the latency of this operation, a mapping which minimizes the critical path length using SIS [8] and Chortle [4] has a path length of 3 and requires 21 4-LUTs. Figure 2 shows the LUT mapping both in equations and circuit topology.

3.1 Traditional Pipelining for Throughput

If we cared only about achieving the highest throughput, we would fully pipeline this implementation such that it took in a new character on each cycle and output its encoding three cycles later. This pipelining would require an additional 7 LUTs to pipeline data which is needed more than one pipeline stage after being generated (*i.e.* 4 to retime `c<3:0>` for presentation to the second stage and 3 to retime `c<3>`, `c<1>` and `i1` for presentation to the final stage). Consequently, we effectively evaluate a design with $N_d = N_{LUT_{design}} = 21$ 4-LUTs with $N'_d = N_a = 28$ physical 4-LUTs. Typical LUT delay, including a moderate amount of local interconnect traversal, is 7 ns [9] [1]. Assuming this is the only limit to cycle time, the im-

FPD'96 -- Fourth Canadian Workshop of Field-Programmable Devices
May 13-14, 1996, Toronto, Canada

```
INORDER = C<7> C<6> C<5> C<4> C<3> C<2> C<1> C<0> ;
OUTORDER = O<3> O<2> O<1> O<0> ;
# stage 1 – 8 LUTs [C<3:0> pass through]
i0 = !C<1> * !C<2> ;
i1 = C<4> * C<5> * !C<6> * !C<7> ;
i3 = C<0> * C<1> * !C<2> ;
i4 = !C<3> * !C<4> * C<6> * !C<7> ;
i6 = !C<0> * C<2> ;
i7 = !C<0> * C<1> ;
i8 = C<0> * !C<1> ;
i11 = !C<7> * C<6> * !C<4> * !C<3> ;
# stage 2 – 9 LUTs [i1,C<3>,C<1> pass through]
i5 = i0 * i1  + i3 * i4 ;
i9 = i6 * i4 + i7 * i4 + i8 * i4 ;
i10 = C<3> + i3 * i4 ;
i12 = i3 * i4 + i6 * i4;
i13 =  i1 * !C<3>  * C<2> ;
i14 = C<2> * !C<1> * i11 ;
i15 = i8 * i4 + i7 * i4 ;
i16 = i7 * i4 + i6 * i4 ;
i17 = i1 * !C<3> * C<0> + C<0> * i0 * i1 ;
# stage 3 – 4 LUTs
O<3> = (i10+i9)*(i5+i9);
O<2> = i12 + i13 + i14 ;
O<1> = i1 * !C<3> * C<1>  + i15 ;
O<0> = i16 + i17 ;
```

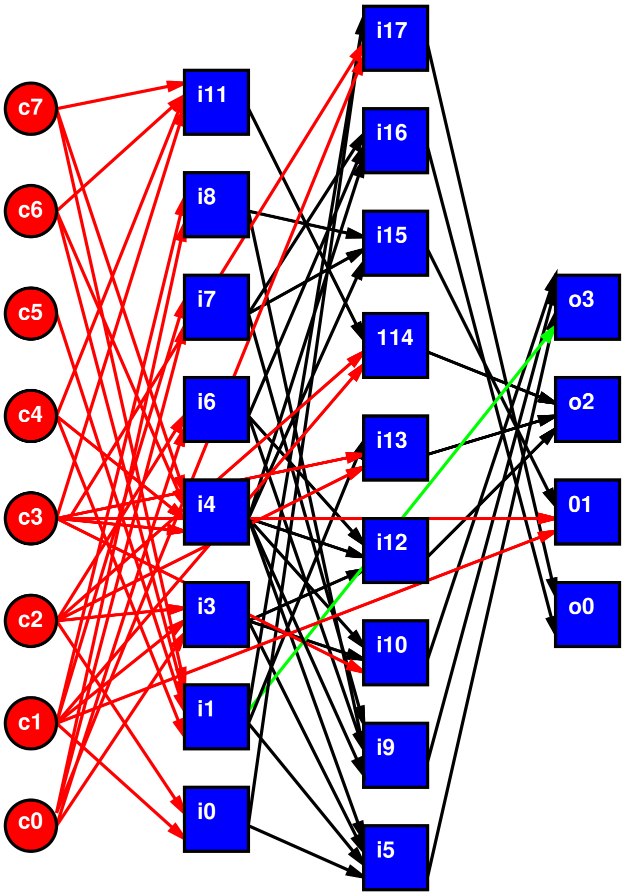


Figure 2: 4-LUT Mapping of ASCII Hex→Binary

plementation could achieve 140 MHz operation. Notice that the only reason we had to have any more LUTs or LUT descriptions than strictly required by the task description was in order to perform signal retiming based on the dependency structure of the computation. Spatially, this implementation requires:

$$A_{pipe} = 28 \cdot 560\mathrm{K}\lambda^2 + 28 \cdot 20\mathrm{K}\lambda^2 = 16240\mathrm{K}\lambda^2$$

3.2 Multicontext Implementation – Temporal Pipelining

If, instead, we cared about the latency, but did not need 140 MHz operation, we could use a multicontext device with 3 LUT descriptions per active element. To achieve the target latency of 3 LUT delays, we need to have enough active LUTs to implement the largest stage – the middle one. If the inputs are arriving from some other circuit which is also operating in multicontext fashion, we must retime them as before. Consequently, we require 3 extra LUTs in the largest stage, making for a total $N_a = 12$. Note that the 4 retiming LUTs added to stage 1 also bring its total LUT usage up to 12 LUTs. We end up implementing $N_d = 21$, with $N_a = 12$ and $N_d' = 36$. If `c<7:0>` were inputs which did not change during these three cycles, we would only need one extra retiming LUT in stage 2 for `i1`, allowing us to use $N_a = 10$.

The multicontext implementation requires:

$$A_{time-pipe} = 12 \cdot 560\mathrm{K}\lambda^2 + 36 \cdot 20\mathrm{K}\lambda^2 = 7440\mathrm{K}\lambda^2$$

In contrast, a non-pipelined, single-context implementation would require $N_d' = N_a = 21$ LUTs, for an area of:

$$A_{nonpipe} = 21 \cdot 560\mathrm{K}\lambda^2 + 21 \cdot 20\mathrm{K}\lambda^2 = 12180\mathrm{K}\lambda^2$$

If we assume that we can pipeline the configuration read, the multicontext device can achieve comparable delay per LUT evaluation to the single context device. The total latency then is 21 ns, as before. The throughput at the 7 ns clock rate is 48 MHz. If we do not pipeline the configuration read, as was the case for the DPGA prototype [9], the configuration read adds another 2.5 ns to the LUT delay, making for a total latency of 28.5 ns and a throughput of 35 MHz.

3.3 Further Temporal pipelining

If the 35-48 MHz implementations are still faster than required by the application, we can reduce N_a even further. Ideally, if we did not have to worry about retiming issues:

$$N_a = \frac{N_d}{\left(\frac{T_{target}}{t_{LUT\_cycle}}\right)}$$

That is, if we can afford $n_{cycle} = \frac{T_{target}}{t_{LUT\_cycle}}$ LUT delays, and we can evaluate N_a LUTs per cycle, we only need enough active LUTs such that $n_{cycle} \cdot N_a = N_d$. *E.g.* if we required less than 4 MHz operation, we should need only one active LUT.

In practice, as long as we have to retime intermediates through LUTs as noted above, there will be a limit to our ability to reduce the maximum stage size by adding temporal pipeline stages. Even if we only had to arrange for the four LUT inputs needed for a single LUT evaluation in stage n, we would have to deploy at least 4 LUTs in stage $n-1$ to carry the signals needed by that single LUT in stage n. *If the only place where intermediate signals can exist is on LUT outputs, as assumed so far, we eventually reach a lower limit on our ability to serialize active resource usage – decreasing active resource requirements by reusing them to evaluate different LUTs in sequence.* The limit arises from signal connectivity requirements. For the ASCII Hex→binary mapping considered above, the 3 cycle, $N_a = 12$ case is the effective limit.

To achieve further reductions it is necessary to relax the restriction that all inputs to an evaluation stage be delivered from active LUT outputs from the immediately prior evaluation stage. One way to achieve this is to associate a pool of memory with each active LUT. This pool serves the same role as a register file in a traditional processor. VEGA [7] uses this approach achieving:

$$A_{compute} \approx N_d \cdot A_{LUT\_descript}$$

Where:

$$120\text{K}\lambda^2 \leq A_{LUT\_descript} \leq 170\text{K}\lambda^2$$

when:

$$256 \leq \frac{N_d}{N_a} \leq 2048$$

A similar alternative is to move the flip-flop which normally lives at the output of a LUT to its inputs and to replicate this for each described LUT rather than only associating such flip-flops with active LUTs. Figure 3 contrasts this scheme with the scheme used in the original DPGA prototype and with traditional LUT organizations. The basic idea is to latch each LUT input on the cycle which it is produced even when the LUT will not be evaluated on immediately subsequent cycle. Figure 4 elaborates upon this configuration. Since each virtual LUT in a group shares each input line, it is not possible to load two different signals into the same input location of different LUTs on the same cycle. Also, since all the inputs to a virtual LUT are latched, the LUT output can be produced on multiple cycles if routing restrictions such as these require it to be generated multiple times.

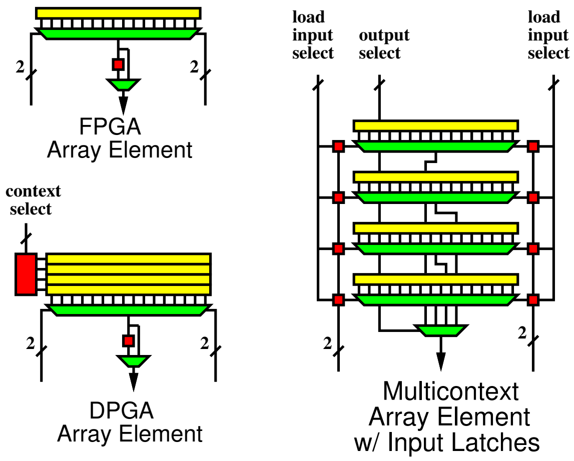


Figure 3: Input Latches versus Output Latches

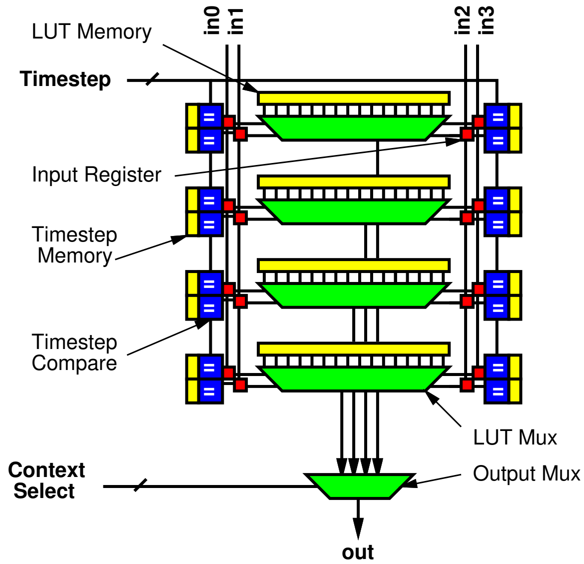


Figure 4: Separately Latched Inputs

Adding these matching units and input flip-flops will, of course, make the described contexts larger.

$$\begin{aligned} A_{ctx} &= A_{LUT\_config\_mem} + A_{LUT\_mux} \\ &\quad + k \cdot A_{input\_match} \\ A_{compute} &= N_a \cdot A_{fixed\_net} + N_d \cdot A_{ctx} \end{aligned}$$

From our own experience, we estimate:

$$\begin{aligned} A_{input\_match} &\approx 20\text{K}\lambda^2 \\ A_{LUT\_MUX} &\approx 10\text{K}\lambda^2 \\ A_{fixed\_net} &\approx 500\text{K}\lambda^2 \\ A_{LUT\_config\_mem} &\approx 40\text{K}\lambda^2 \end{aligned}$$

Together, this gives us:

$$A_{compute} \approx N_a \cdot 500\text{K}\lambda^2 + N_d \cdot 130\text{K}\lambda^2$$

In the most extreme serial case, $N_a = 1$, $N_d = 21$. This implementation requires area:

$$A_{serial} = 1 \cdot 500\text{K}\lambda^2 + 21 \cdot 130\text{K}\lambda^2 = 3230\text{K}\lambda^2$$

Running at a 9.5 ns LUT cycle, this implementation has a 200 ns latency achieving 5 MHz operation.

3.4 Interleaved Computation

Excessive serialization is expensive since we need to add resources to hold intermediate values. Another way to share resources while meeting a lower computational rate is to overlay multiple, independent computations into the same space. This allows each computation to have wide stages as in Section 3.2, while still taking advantage of low throughput requirements.

In our example above, we might run 12 LUTs on the same 21 cycle round to achieve 5 MHz throughput for the ASCII conversion. The conversion itself only occupies these 28 LUTs for 3 cycles, leaving 18 cycles during which the array can be used for other tasks. Array costs are now in line with Equation 1. The amortized cost for this implementation is then:

$$\begin{aligned} A_{interleave} &= \left(\frac{3}{21}\right) \cdot \left(12 \cdot 560\text{K}\lambda^2 + 21 \cdot 20\text{K}\lambda^2\right) \\ &= 1020\text{K}\lambda^2 \end{aligned}$$

3.5 Area Comparison Summary

Table 2 summarizes the area for various implementation points described in this section. Also included is the effective area required for two different processor implementations of this task. The MIPS-X [6] assembly code is shown in Figure 5; for the sake of comparison, we assume the 4 cycle path which occurs when converting an ASCII decimal digit. The DEC Alpha [5] implementation assumes that the conversion is performed by table lookup and the table is already loaded into the on-chip data cache; the conversion takes one add and one lookup, each of which take up half a cycle since the Alpha can issue two instruction per cycle. The processor area comes from scaling the actual area to meet the cycle time – this assumes that we either deploy multiple processors to achieve a higher rate of execution than the base processor cycle time or that the task shares the processor with other tasks and area is amortized accordingly.

```
atoh:
//c in r1
//res in r2
//2-9 instructions
      blt r1,#48,zero
      bgt r1,#57,try40
      sub r1,#48,r2
      bne r0,r0,done
try40:
      and r1,#223,r1
      blt r1,#65,zero
      bgt r1,#70,zero
      sub r1,#55,r2
      bne r0,r0,done
zero:
      addi r0,r0,r2
done:
```

Figure 5: ASCII Hex→binary MIPS-X Assembly

In this section, we have examined one task in depth. See [2] for a broader look at multicontext applications.

4 Device Sizing

One thing which is clear from Table 2 is that there is an optimal ratio between N_d and N_a which varies with the desired throughput requirements. Unfortunately, we cannot expect to have a distinct DPGA design for every possible number of contexts.

We can define a throughput ratio as the ratio of the

Throughput	**Optimal**		**FPGA**		**MIPS-X**		**Alpha**	
(chars/sec)	Area	Note	Area	Ratio	Area	Ratio	Area	Ratio
435M	48.7Mλ^2	3$\times$ pipe FPGA	48.7Mλ^2	1.0	5.9Gλ^2	0.0082	6.8Gλ^2	0.0072
140M	16.2Mλ^2	1$\times$ pipe FPGA	16.2Mλ^2	1.0	1.9Gλ^2	0.0084	2.3Gλ^2	0.0072
35M	7.4Mλ^2	3 ctx DPGA	12.2Mλ^2	0.61	480Mλ^2	0.016	550Mλ^2	0.014
5M	3.2Mλ^2	21 ctx DPGA$'$	12.2Mλ^2	0.27	68Mλ^2	0.048	78Mλ^2	0.041
5M	1.0Mλ^2	21 ctx DPGA interleaved	12.2Mλ^2	0.084	68Mλ^2	0.015	78Mλ^2	0.013

Values used for comparison

Parameter	**Value**
A_{MIPS-X}	68Mλ^2
A_{Alpha}	6.8Gλ^2
A_{FPGA_LUT}	580Kλ^2
A_{DPGA}	$560\text{K}\lambda^2 + \left(\frac{N_d'}{N_a}\right) \cdot 20\text{K}\lambda^2$
$A_{DPGA'}$	$500\text{K}\lambda^2 + \left(\frac{N_d'}{N_a}\right) \cdot 130\text{K}\lambda^2$
T_{MIPS_cycle}	50 ns
$n_{MIPS\_cyles/conversion}$	4
T_{Alpha_cycle}	2.3 ns
$n_{Alpha\_cyles/conversion}$	1
T_{FPGA}	7 ns
T_{DPGA}	9.5 ns
$T_{DPGA'}$	9.5 ns

$DPGA'$ – DPGA with separate input latches (Figure 4)

Table 2: ASCII Hex$\rightarrow$Binary Implementation Comparison

desired task throughput to the raw LUT throughput:

$$R = \frac{Th_{LUT}}{Th_{task}} \qquad (2)$$

Here, we have been assuming $Th_{LUT} \approx$ 140 MHz for conventional devices. When $R >> 1$, single contexts are very inefficient since far more active LUTs are provided than necessary. When $c >> R$, where c is the number of contexts supported, area is unnecessarily consumed by context memory. Figure 6 plots R versus c showing the relative efficiency of running a task with throughput ratio R on a device with c contexts for both the DPGA context sizes and DPGA$'$ sizes.

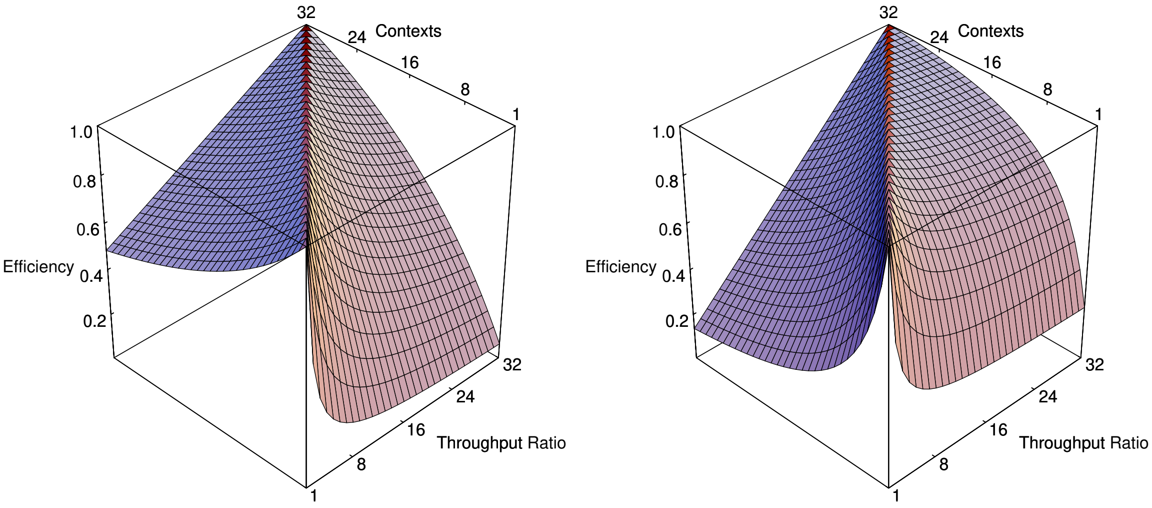
One interesting design point to notice on both graphs is that point where the area dedicated to the fixed LUT is equal to the area of the contexts. This occurs at $c = 28$ for the DPGA and $c = 4$ for DPGAs with input latches. At this point, the efficiency never drops below 50% for any values of R. This is interesting since the cross efficiencies at the extremes in the graph shown drop down to 13%.

5 Typical Application

In a quick review of the FPD'95 proceedings, we see FPGA task implementations running at frequencies between 6MHz and 80MHz, with most running between 10MHz and 20MHz. Thus, we see a decent number of FPGA tasks implemented with $7 < R < 14$. We might ask why these applications have such large throughput ratios, R. There are two main causes:

1. **Task throughput requirements are low** – an external timing requirement dictates that there is no point for the circuit to provide any higher throughput.
2. **Critical path with cyclic constraints** – throughput is often limited by the critical path, especially when the result from one round of computation are required at the start of the next round of computation.

Regardless of the detailed reasons, in this range of operation, single context FPGAs are paying a raw overhead factor of 6-10$\times$ for DPGAs or 3$\times$ for DPGAs with input latches. Even if the packing efficiency on the standard DPGAs is only 50%, this still amounts to a 3-5$\times$ area overhead.



Left – DPGA; Right – DPGA$'$ (input latches)

Figure 6: Task:LUT Throughput Ratio versus Number of Contexts

6 Conclusions

Multicontext FPGAs have the potential to pack typical computational tasks into less area, delivering more computation per unit area – *i.e.* increased computational density. The key properties they exploit are:

1. LUT throughput is often much greater than task throughput.
2. A LUT instruction is smaller than the active LUT.

Sharing the active portion of the LUT among multiple instructions yields increased device usage efficiency when the throughput ratio is large ($R >> 1$).

Task dataflow dependencies require that a number of signals be made available to each stage of computation. The retiming requirements associated with satisfying these dependencies can pose a limit to the effective serializability of the task. In some cases we can avoid these limitations by overlaying loosely or unrelated tasks on the same resources. Alternately, we can architect intermediate data storage to allow serialization, at the cost of larger context resources. Architectural design of these multicontext structures and synthesis for them is a moderately young area of research. Interesting research possibilities exist in the areas of:

- Temporally oriented synthesis to minimize retiming requirements
- Tighter structures for data retiming
- Hybrid structures with an intermediate amount of retiming resources

Acknowledgments:

Jeremy Brown, Derrick Chen, Ian Eslick, Ethan Mirsky, and Edward Tau made the DPGA prototype possible. The input latch ideas come from the TSFPGA architecture co-developed with Derrick Chen. Many of the quantitative details about the DPGA are the results of the collected efforts of this team.

This research is supported by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

References

[1] Paul Chow, Soon Ong Seo, Dennis Au, Terrence Choy, Bahram Fallah, David Lewis, Cherry Li, and Jonathan Rose. A 1.2μm CMOS FPGA using Cascaded Logic Blocks and Segmented Routing. In Will Moore and Wayne Luk, editors, *FPGAs*, pages 91–102. Abingdon EE&CS Books, 15 Harcourt Way, Abingdon, OX14 1NV, UK, 1991.

[2] André DeHon. DPGA Utilization and Application. In *Proceedings of the 1996 International Symposium on Field Programmable Gate Arrays*. ACM/SIGDA, February 1996. Extended version available as Transit Note #129, available via anonymous FTP `transit.ai.mit.edu:transit-notes/tn129.ps.Z`. Anonymous FTP `transit.ai.mit.edu:papers/dpga-use-fpga96.ps.Z`.

[3] André DeHon. Entropy, Counting, and Programmable Interconnect. In *Proceedings of the 1996 International Symposium on Field Programmable Gate Arrays*. ACM/SIGDA, February 1996. Extended version available as Transit Note #128, available via anonymous `transit.ai.mit.edu:transit-notes/tn128.ps.Z`. Anonymous FTP `transit.ai.mit.edu:papers/entropy-fpga96.ps.Z`.

[4] Robert Francis. *Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays*. PhD thesis, University of Toronto, 1992.

[5] Paul Gronowski, Peter Bannon, Michael Bertone, Randel Blake-Campos, Gregory Bouchard, William Bowhill, David Carlson, Ruben Castelino, Dale Donchin, Richard Fromm, Mary Gowan, Anil Jain, Bruce Loughlin, Shekhar Mehta, Jeanne Meyer, Robert Mueller, Andy Olesin, Tung Pham, Ronald Preston, and Paul Robinfeld. A 433MHz 64b Quad-Issue RISC Microprocessor. In *1996 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 222–223. IEEE, February 1996.

[6] Mark Horowitz, John Hennessy, Paul Chow, Glenn Gulak, John Acken, Anant Agarwal, Chorng-Yeung Chu, Scott McFarling, Steven Przybylski, Steven Richardson, Arturo Salz, Richard Simoni, Don Stark, Peter Steenkiste, Steven Tjiang, and Malcom Wing. A 32b Microprocessor with On-Chip 2K byte Instruction Cache. In *1987 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 30–31. IEEE, February 1987.

[7] David Jones and David Lewis. A Time-Multiplexed FPGA Architecture for Logic Emulation. In *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pages 495–498. IEEE, May 1995.

[8] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. UCB/ERL M92/41, University of California, Berkeley, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, May 1992.

[9] Edward Tau, Ian Eslick, Derrick Chen, Jeremy Brown, and André DeHon. A First Generation DPGA Implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pages 138–143, May 1995. Anonymous FTP `transit.ai.mit.edu:papers/dpga-proto-fpd95.ps.Z`.