HSRA:
High-Speed, Hierarchical Synchronous Reconfigurable Array

William Tsu, Kip Macy, Atul Joshi, Randy Huang
Norman Walker, Tony Tung, Omid Rowhani, Varghese George
John Wawrzynek, and André DeHon
Berkeley Reconfigurable, Architectures, Software, and Systems
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776
contact: <andre@acm.org>

Abstract

There is no inherent characteristic forcing Field Programmable Gate Array (FPGA) or Reconfigurable Computing (RC) Array cycle times to be greater than processors in the same process. Modern FPGAs seldom achieve application clock rates close to their processor cousins because (1) resources in the FPGAs are not balanced appropriately for high-speed operation, (2) FPGA CAD does not automatically provide the requisite transforms to support this operation, and (3) interconnect delays can be large and vary almost continuously, complicating high frequency mapping. We introduce a novel reconfigurable computing array, the High-Speed, Hierarchical Synchronous Reconfigurable Array (HSRA), and its supporting tools. This package demonstrates that computing arrays can achieve efficient, high-speed operation. We have designed and implemented a prototype component in a 0.4μm logic design on a DRAM process which will support 250MHz operation for CAD mapped designs.

1 Introduction

A common myth about FPGAs is that they are inherently 10× slower than processors. We see no physical limitations which would make this true, but there are some good reasons why this myth persists.

Looking at raw cycle times, we see that the potential operating frequencies for FPGAs are comparable to processors in the same process (See Table 1). The cycle time on a processor represents the minimum interval at which a new operation on new data can be initiated or completed. That is, it defines how fast we can clock the computational and memory units and *reuse* them to perform subsequent operations. Since traditional FPGAs are not synchronous, it is not as obvious what the native cycle time is for an FPGA. However, if we also take the FPGA cycle time as the minimum interval at which we can launch a new datum for computation, then we can identify a cycle time. For example, the XC4000XL-09 family has a logic evaluation to clock setup time of 1.6 ns, and a clock-to-Q time of 1.5 ns. If we take the minimum clock low and high-times of 2.3 ns each, we can define a cycle of 4.6 ns which leaves (4.6-1.5-1.6)=1.5 ns for interconnect on each cycle. Similarly, Von Herzen defined a 4 ns cycle on XC3100-09 and designed his signal processing applications to this cycle time [11]. In Table 1, we see that these cycle times are within a factor of two of processors in the same process.

In practice, however, the applications we see running at 200MHz+ on these FPGAs are few and far between. While the basic cycle time for an FPGA is small, most contemporary FPGA designs run much slower—more typically in the 25-70MHz range. Why do designs run this much slower than the conceivable peak? We conjecture there are several factors which contribute to the low frequency of most FPGA designs:

1. **no reason to run faster** — Often the limited speed is all the user wants or needs, and there is no application reason to run at a higher cycle rate. For example, if the application is sample rate limited at a modest sample rate, there is no requirement to process data at a higher rate. Furthermore, when data rates are limited by system components outside of the FPGA or standards, the application may have no cause to run at a faster rate. However, when such external or application limits appear, it is often possible to reduce the hardware required by running a more serialized design in less space (fewer gates, smaller FPGA component) at the higher cycle rate achievable by the FPGA.

2. **cyclic data dependencies limit pipelineability** – Cycles in the flow graph define a minimum clock cycle time. We cannot pipeline down to the LUT level within such cycles. We can, however, run the design C-slow [14] at the LUT-cycle rate, allowing us to solve C-independent problems simultaneously in the hardware space. If we do not have a number of independent problems to solve, we can reuse gates and interconnect at the LUT-cycle rate to solve the problem in less area when the device has multiple contexts (*e.g.* DPGA [6]).

3. **inadequate tool support** – Reorganizing a design to run at this tight cycle rate can be a tedious task. While the basic technology is known in the design automation world, typical FPGA tools and design flows do not provide support for aggressive retiming. In part this results from the traditional glue-logic replacement philosophy which lets the user define the base cycle and what has to happen within a cycle, rather than taking a computational view which says that the user defines a task and the tools are free to transform the problem as necessary to map the user's task onto the computing platform.

4. **interconnect delays dominate** – Interconnect delays depend on the distance between source and sink and can easily dominate all other delays. We were only able to define the tight cycle times we did above by assuming very local communications. If we allowed even one cross chip delay time in the

FPGA 99 Monterey CA USA
Copyright ACM 1999 1-58113-088-0/99/02...$5.00

Design	Feature	Cycle	Reference
XC4000XL-09	0.35μm	4.6 ns	[20]
A10K100A-1	0.35μm	† 5.0 ns	[1]
Strong Arm	0.35μm	5.0 ns	[15]
Alpha	0.35μm	2.3 ns	[10]
SPARC	0.35μm	3.0 ns	[9]
Pentium	0.35μm	3.3 ns	[4]
Alpha	0.35μm	1.7 ns	[7]
HSRA	0.40μm	4.0 ns	

† 5.0 ns cycle based on $t_{CH_{\min}} = t_{CL_{\min}} = 2.5$ ns

Table 1: Cycle Rate comparison at $\approx$0.35μm

Number of Registers	1	2	3	4	5	6	7	8	9	10
Percentage	72	16	4.5	2.2	1.3	0.96	1.2	0.46	0.12	0.11

Table 2: Benchmark-Wide Distribution of Registers Required between LUTs

cycle, the cycle time would increase significantly. This leads us to believe that either we have to accept a much larger cycle time, or we must limit all communications to local connections, as in [11]. As long as we must traverse an entire long interconnect line in a single cycle, we are left where we can only achieve the tight cycle for very stylized problems or with heroic personal effort to design and layout the computation entirely using local connections.

5. **pipelining becomes expensive** – In order to pipeline the device heavily enough to run at this cycle rate, the design needs a larger number of flip-flops for proper retiming. While flip-flops are "relatively" cheap in many FPGAs, the typical balance is roughly one flip-flop per 4-LUT. However, for a fully pipelined design, the number of flip-flops required may exceed the number of LUTs by a factor of 2–7 as we will see in Section 2.

We see that items 1 and 2 are application oriented and can often be mitigated by appropriate design. Items 3, 4, and 5 are architecture and CAD related. The HSRA architecture, which we introduce here, represents a direct attack on these architecture and CAD problems. The result is a reconfigurable computing array capable of running profitably on feed-forward and data-parallel designs at a cycle rate competitive with microprocessor designs in the same process. For designs with data-dependent cycles, the architectural techniques studied here should mix well with multicontext FPGA techniques (*e.g.* [6] [2] [13] [17]).

In the next two sections, we present additional evidence supporting the architectural issues of retiming balance (5) and interconnect delays (4). In Section 4, we introduce our architecture and show how it addresses these problems. Section 5 provides highlights from our prototype HSRA design. Section 6 addresses the new retiming problems introduced by our architecture and looks at the net retiming requirements which it implies.

2 Retiming Requirements

A necessary, but not sufficient, requirement for running at a minimum, single-LUT delay cycle, as developed in the previous section, is that the design be retimed so that there is at most one LUT evaluation between any pair of registers. For fully feed-forward designs (designs with no data dependency cycles), we simply add registers at the outputs and retime [14] them until the pipelined design meets this requisite property.

For cyclic designs, we can only pipeline up to the cycle rate implied by the design. If the cycle rate is C, we then run the design C-slow by replacing every register in the original design with a bank of C registers. The resulting design can now be pipelined and retimed down to a cycle rate of one as required. The C-slow design will execute C independent streams of computation, taking a new input for each of the C computations every C'th cycle and similarly producing a new output every C'th cycle.

To understand the minimum register requirements necessary to retime designs to run at the single-LUT cycle rate, we performed these retiming transformations on a series of benchmark circuits. That is:

1. Start with the delay targeted SIS [18] and Flowmapped [5] 4-LUT networks for the IWLS93 benchmarks under 250 4-LUTs.
2. Determine cycle rate, C, using SIS's implementation of retiming. (In concept, calculate the cycle bound by counting the delays and registers around every internal loops and finding the largest ratio of delays to registers in any loop. Effectively, we make an arbitrary number of registers available at the output of the network so that only the inherent cycle bound determines the cycle rate of the circuit.)
3. Replace every register in the original design with C registers.
4. Add sufficient registers on output of design for pipelining.
5. Use SIS to retime registers in the network.
6. Collect statistics on the properly retimed design.

Table 2 shows the distribution of register depths between LUTs across this benchmark set. Table 3 shows the general results of this mapping exercise. While unit depth between registers does remain the single largest depth, there are a significant number of cases which require greater depth. Benchmark wide, the designs require, on average, 1.7 registers between adjacent LUT pairs.

Earlier results for non-retimed circuits [16] suggested that a single register was worthwhile to integrate into the functionality of every logic block. This data alone suggests that the balance of one flip-flop per LUT on average is not bad. For some cases the pipeline penalty for such a distribution is moderately small, but for some it can be a factor of 2–7 greater. Since flip flops are small (*e.g.* 4Kλ^2, maybe 10Kλ^2 with switching and configuration to allow optional selection) compared to the total amortized area for a LUT with reasonable interconnect (500Kλ^2–1Mλ^2), it may be worthwhile to tip the balance to more than one flip-flop per LUT, as long as this

Design	LUTs	orig. cycle	C-slow C	add pipe stg	Total Regs
C17	2	2	1	1	2
shiftreg	2	1	1	0	3
daio	3	2	1	1	5
lion	3	2	1	1	3
train4	3	2	1	1	3
b1	4	2	1	1	4
cm82a	4	3	1	2	7
majority	4	3	1	2	4
lion9	4	2	1	1	4
dk27	5	2	1	1	5
parity	5	3	1	2	5
con1	6	3	1	2	6
s27	6	3	2	2	9
cm151a	8	4	1	3	13
mc	9	3	2	2	16
bbtas	9	2	2	1	14
cm138a	9	3	1	2	12
cm42a	10	2	1	1	10
cm152a	11	4	1	3	11
tav	12	3	1	2	16
cm150a	15	5	1	4	24
cm162a	15	4	1	3	32
tcon	16	2	1	1	16
cm163a	17	4	1	3	33
cm85a	19	4	1	3	22
xor5	19	5	1	4	23
cmb	20	5	1	4	35
train11	20	4	2	3	25
i1	22	1	4	3	46
beecount	22	4	3	3	31
x2	23	4	1	3	39
misex1	23	4	1	3	37
cu	25	4	1	3	48
dk512	26	3	3	2	39
s208	28	5	1	7	65
pm1	29	4	1	3	57
pcle	29	4	1	3	54
ex5	30	3	3	2	35
dk17	30	4	3	3	45
mult16b	31	2	1	1	31
my_adder	32	17	1	16	392
decod	32	3	1	2	34
unreg	32	3	1	2	49
cc	33	3	1	2	54
dk15	34	4	3	3	45
mult16a	35	16	2	1	167
ex3	36	3	3	2	39
count	37	7	1	6	158
pcler8	40	5	1	4	106
rd53	40	5	1	4	50
ex7	40	4	3	3	53
squar5	41	5	1	4	58
comp	41	6	1	5	61
bbara	42	3	3	2	43
s298	42	4	3	3	80
ex4	44	4	3	3	67
i3	46	4	1	3	50
sqrt8	47	5	1	4	65
o64	48	6	1	5	62
sqrt8ml	50	11	1	10	107
mux	53	7	1	6	79
dk14	55	4	3	3	66
opus	56	5	3	4	78
misex2	57	4	1	3	87
mark1	58	4	3	3	88
s344	58	4	4	1	105
s349	58	4	4	1	105
ldd	58	4	1	3	87
s382	60	3	3	1	89
s444	60	3	3	1	89
s400	60	3	3	1	89
b9	61	5	1	4	117
mult32b	62	2	1	1	62
mult32a	67	32	2	1	591
cht	68	4	1	3	133
sct	72	4	1	3	110
C499	74	5	1	4	178
C1355	74	5	1	4	178
lal	76	4	1	3	118
s420	76	6	3	5	134
i2	77	6	1	5	94
bbsse	78	4	4	3	97
sse	78	4	4	3	97
s386	84	4	4	3	119
z4ml	85	5	1	4	97
ex6	87	5	4	4	115
9symml	87	7	1	6	124
5xp1	88	5	1	4	119
c8	91	5	1	4	150
s526	92	4	3	4	156
s641	95	9	5	8	399
s526n	95	4	4	3	168
s713	96	9	5	8	406
ex2	102	4	4	2	116
f51m	105	5	1	4	137
i4	110	5	1	4	200
s510	110	4	4	3	156
C432	119	11	1	10	355
apex7	126	6	1	5	300
mm9a	128	18	8	17	701
sao2	136	6	1	5	164
C880	141	9	1	8	462
s838	141	14	3	13	584
kirkman	142	5	4	4	171
i6	144	3	1	2	175
cse	144	5	4	4	169
mm4a	160	11	6	10	294
C1908	162	10	1	9	459
example2	169	5	1	4	383
mm9b	169	24	14	23	1170
9sym	172	7	1	6	190
s820	180	5	5	3	235
s832	182	5	5	3	239
i5	185	4	1	3	367
s953	198	5	5	4	329
dk16	199	5	4	4	218
ex1	202	5	4	4	256
s1423	207	18	14	16	1430
keyb	208	5	5	4	231
i7	212	3	1	2	236
rd73	220	7	1	6	254
C2670	221	8	1	7	622
ttt2	232	6	1	5	338
s1196	244	8	1	8	498
clip	252	6	1	5	299

Table 3: Benchmark statistics after retiming to single LUT delay per cycle

Path	Total Delay	Interconnect % of cycle	
		5 ns	2.1 ns
LUT-local-LUT	2.1 ns	42%	100%
LUT-row-local-LUT	3.6 ns	72%	170%
LUT-column-local-LUT	7.6 ns	150%	360%
LUT-row-column-local-LUT	9.1 ns	180%	430%
LUT-row-fanout-local-LUT	10.6 ns	210%	500%

Table 4: Interconnect Delays for Altera 10K100A-1

can be done without increasing interconnect requirements.

If our requisite interconnect and LUT delay both fit within the cycle rate, then this data tells the complete story. However, if we have interconnect delays which are greater than our cycle rate, then the retiming requirements can be even greater than this, as we will see below (Section 6).

3 Interconnect Delays

The variation and absolute size of interconnect delays is one of the challenges to using an FPGA at a high clock rate. Table 4 shows the clocked LUT→LUT delays in a modern FPGA. From this data we immediately see:

- Interconnect delay varies considerably, such that the LUT→LUT cycle time we might define may vary from 2.1 ns (fully local assuming no other limitations on cycle rate) to 10.6 ns, a difference of 5×.
- Absolute interconnect delay, for all but the most local connections, dominates logic delay.

If we are stuck traversing arbitrary interconnect at our cycle rate, even if we retime to a single LUT between registers, we still have a moderately large cycle (>10 ns) compared to processors in a comparable process (See Table 1). This means that shifting register balance alone in conventional FPGA architectures is not sufficient to make high cycles rates achievable. Longer interconnect paths would become unusable in order to maintain a high clock rate. If we want to consistently achieve high clock rates, it will also be necessary to pipelined long interconnect paths so that they will not limit our clock cycle.

4 Architecture

We have seen that small cycle times are possible, but hard to exploit with current FPGA architectures. Our new architecture is designed to overcome these problems. The basic principles are:

1. Pipeline the architecture itself:
    - Fix a single cycle time and define everything in terms of that cycle time.
    - Add mandatory pipeline registers to break delays up into clock cycle segments, including long interconnect runs.

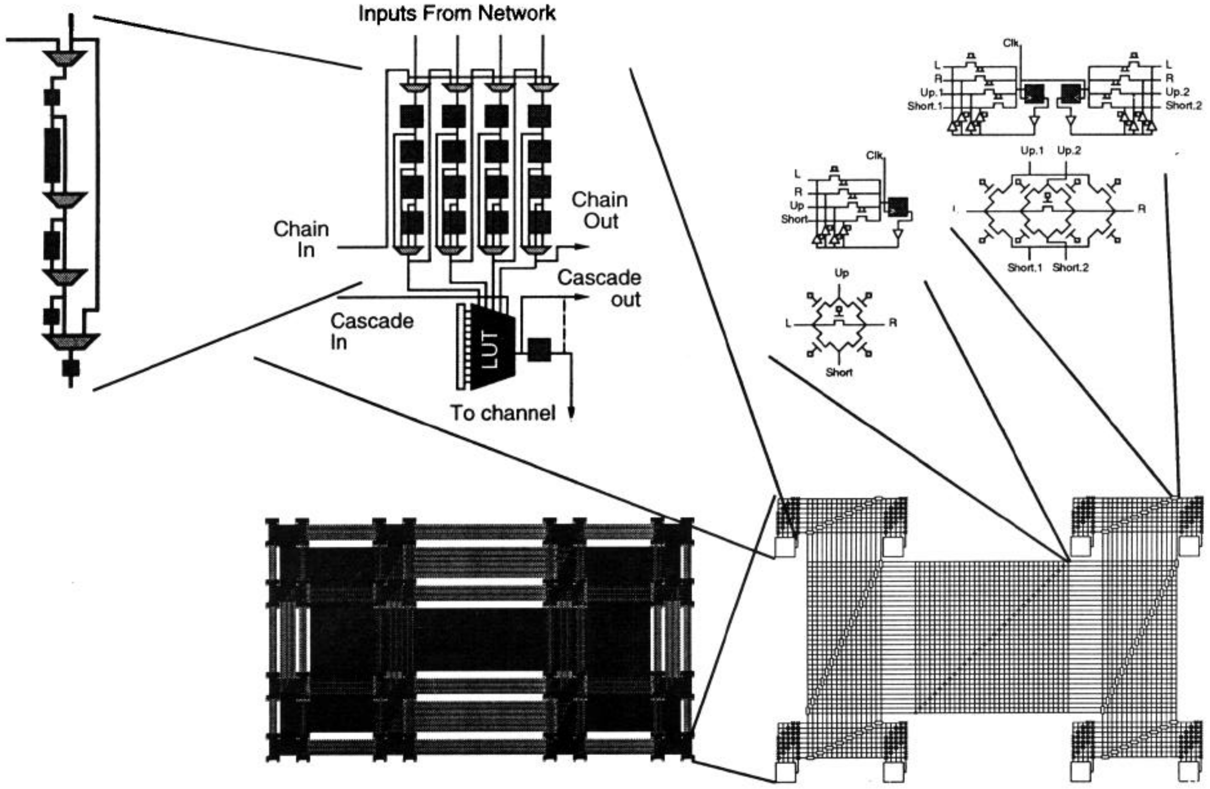
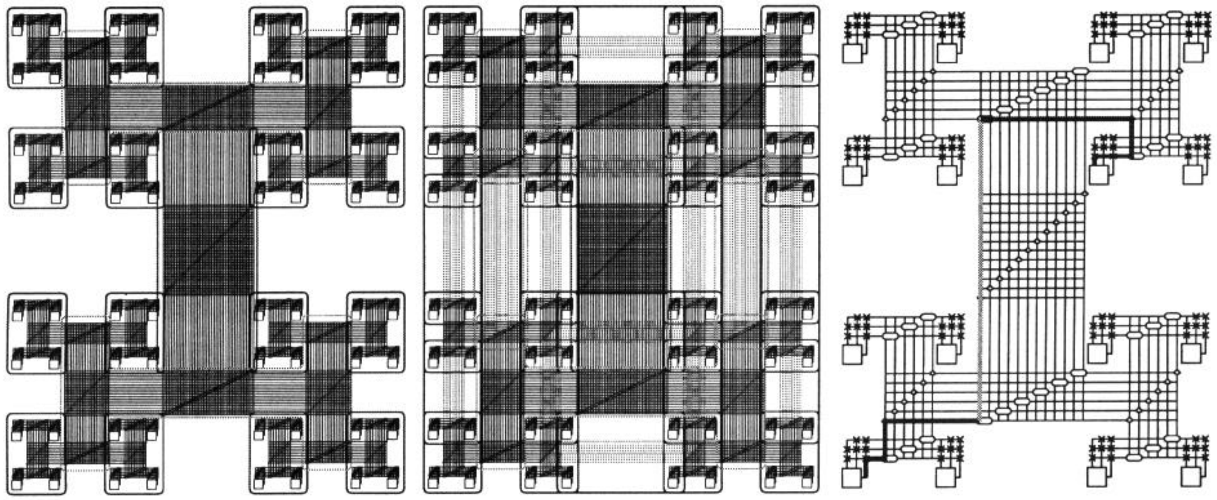


Figure 1: HSRA Architecture Overview



No Shortcut

w/ Shortcuts

Sample Route

Figure 2: Timing Domains in Interconnect

2. Balance resources to application needs—provide more retiming registers so device can be run profitably at the defined cycle rate.

3. Integrate data retiming into CAD flow to match fixed cycle rate and accommodate pipelined interconnect delays.

Figure 1 shows an overview of the HSRA architecture. The computational element is a conventional 4-LUT design. The interconnect is a full 2-ary hierarchical array with shortcut connections like a Fat-Pyramid [8]. What differs from a traditional FPGA design are the balance and implementation of retiming resources and the presence of pipeline registers in the interconnect.

While we have worked out a design in detail for a hierarchical interconnect, the key techniques exploited here should be applicable to any pipelineable interconnect structure.

4.1 Timing Domains

As noted above, long interconnect runs are a possible impediment to a high clock cycle rate. In order to avoid this obstacle, we place pipeline registers directly in the interconnect as needed. That is, for a given target clock cycle rate, we calculate (at HSRA design time), the length of interconnect we can travel in a clock cycle. At the end of this length, we place a register and start over. As a result, we break the design into timing domains as shown in Figure 2. That is, we can go between any two LUTs or switching blocks in a given domain within a single cycle. If two LUTs are not in the same timing domain, it requires additional clock cycles to traverse the intervening interconnect. For example, Figure 2 (right) shows a route which takes 3 clock cycles to get between LUTs.

As shown in the overview (Figure 1), the timing domains are implemented by populating a given switchbox with either pass transistor (or non-latched, rebuffered) switch point or a registered switch point. In the pass-transistor case, the switchboxes look similar to conventional switchpoint implementations. For small clock cycles and long interconnect segments in the upper part of the hierarchy, it will become necessary to place registers in the middle of long wires, breaking them into multiple clock cycles.

The time to evaluate a single 4-LUT is actually moderately small, easily 1 ns in the 0.35μm processes used for comparison, and possible to implement in much less time. If we target a very aggressive cycle comparable to LUT and flip-flop delay (*e.g.* Altera's 2.1 ns local delay), then the LUT evaluation, itself, makes a nicely balanced timing domain. However, if we relax the cycle time, we are either left with an imbalanced pipeline, or we need to rebalance the computational portion of the cycle. Two possibilities to consider are:

1. Place the LUT evaluation in a timing domain with a small amount of interconnect.

2. Use a cascade of LUTs as the basic logic cycle time. For example, a cascade chain made of hardwired inputs between multiple LUTs is shown in Figure 3.

For our detailed architecture development, we have been focusing on the LUT cascade.

4.2 Logic Block Retiming

Our Basic Logic Block (BLB) differs from conventional logic blocks primarily in the way we handle retiming. Instead of having a single, optional output-register on the LUT, we have a bank of flip-flops on the input which we can set to any depth between 1 and d, the architected maximum input retiming depth for a particular device. This both gives us the richer retiming which we identified as necessary in Section 2 and gives us the retiming we need to balance out interconnect path delay difference introduced by the pipelined interconnect.

Figure 4 shows an example of how the input retiming bank is typically used. The unplaced design has 3 LUTs providing inputs to a third. Once placed, the 3 LUTs are all a different number of clock cycles away from the consumer. In order to get the timing correct, it is necessary for all three signals to show up at the consumer on the same logical cycle. Therefore, we must add additional delay to the nearby paths so that they arrive at the same time as the far away paths. These delays are added from the input retiming bank. Of course, if we are fortunate, it is sometimes possible to logically lag the evaluation time of the closer LUTs so that less padding is necessary (See Figure 5).

Note that we have chosen to put the variable depth retiming bank on the inputs to the logic block rather than on its outputs. By placing the extra retiming on the inputs, we place no greater requirements on the interconnect network than an unpipelined, unretimed design. The data is simply routed to the target BLB as soon as possible where it is delayed appropriately for presentation to the LUT. Had we placed the retiming register on the output (See Figure 6), it would have been necessary to route a separate output over the network for each different retiming of the output needed by consumers, as was the case in Xilinx's Time-Multiplexed FPGA [19]. For similar reasons, we avoided the complete decoupling of extra flip-flops from logic blocks as explored for time-multiplexed architectures by Bhat [2] and Chang [3].

The use of input retiming instead of output retiming does come at a cost. Naively, we need $k\times$ (k number of inputs to LUT) as many registers as the output case. From the experiment described in Section 2, we can also calculate the input depths required. Benchmark-wide, we need an average of 1.3 registers per input or a total of 4.4 registers per LUT. As future work, we should revisit this question to see if the additional expense of extra flip flops was suitably compensated by the reduced interconnect requirements.

5 Implementation Highlights

We have implemented a prototype HSRA in a 0.4μm DRAM process. As noted, the architecture can be targeted to a range of cycle times by selecting where to populate the switchboxes with registered switches. Early design studies suggested we could build an array for a cycle time as low as 2 ns, but at moderately large cost in area (See Figure 7), power, and cross-chip latency (See Figure 8). Consequently, we settled on a 250MHz design point for our detailed implementation.

Latency Cost A primary cost of pipelining, from a latency standpoint, is the additional clock-to-Q delay which is inserted on every clock cycle. That is, a pipelined design of depth P_d pays $P_d \times t_{\mathrm{clk}\rightarrow\mathrm{Q}}$ overhead delay. In this process, our clock-to-Q delay was just under 1 ns. Crudely speaking, at 500 MHz, this meant half of our cycle went to clocking overhead, or our pipelining latency cost was 100%. At 250 MHz, we are only spending a quarter of the cycle in pipeline overhead. In addition to clock-to-Q delay, the fixed cycle granularity along with the restriction that clock domain boundaries occur at switchboxes also imposes a fragmentation overhead since all delays are rounded up to the cycle time by the fixed clocking regime.

At 250 MHz, we nominally spend 3 ns in interconnect and compute and 1 ns in clock overhead each cycle. This increases end-to-end or data cycle latency by 33%. For feed-forward designs, the additional latency does not harm throughput and is generally tolerable. For data-dependent designs, the latency can expand the time around the critical cycle and reduce the single data stream throughput. In data-parallel cases multiple data streams can share the com-

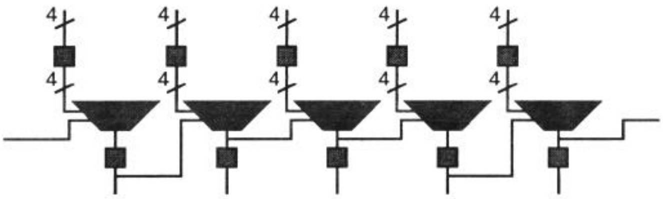


Figure 3: LUT timing with 3 logic block long Cascade Chain

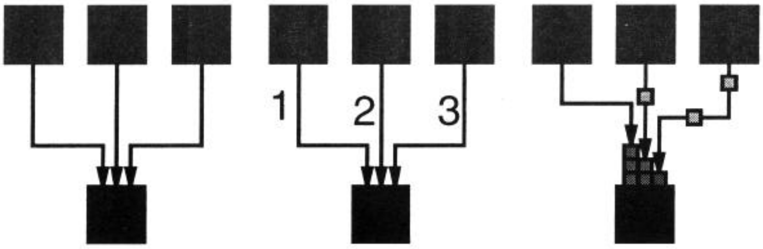


Figure 4: Example: Input retiming registers balancing mapped path lengths

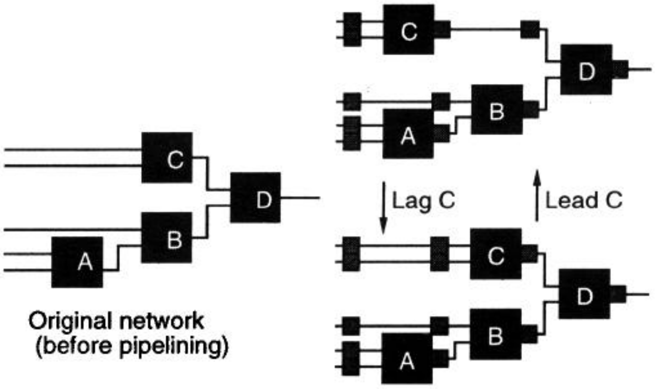


Figure 5: Example of Retiming Freedom and Lead/Lag of Node

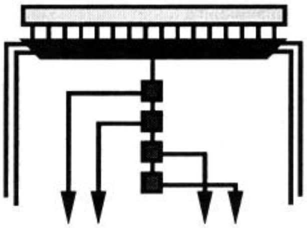


Figure 6: Output Retiming—An alternative to Input Retiming

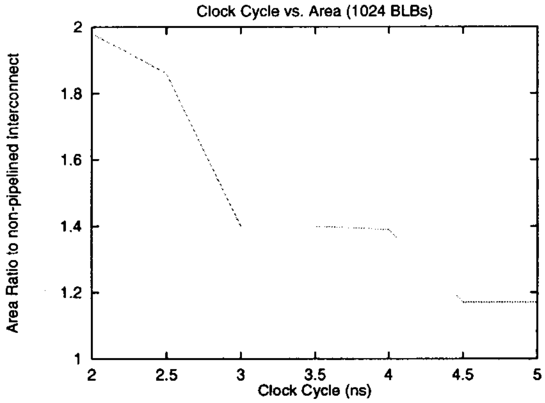
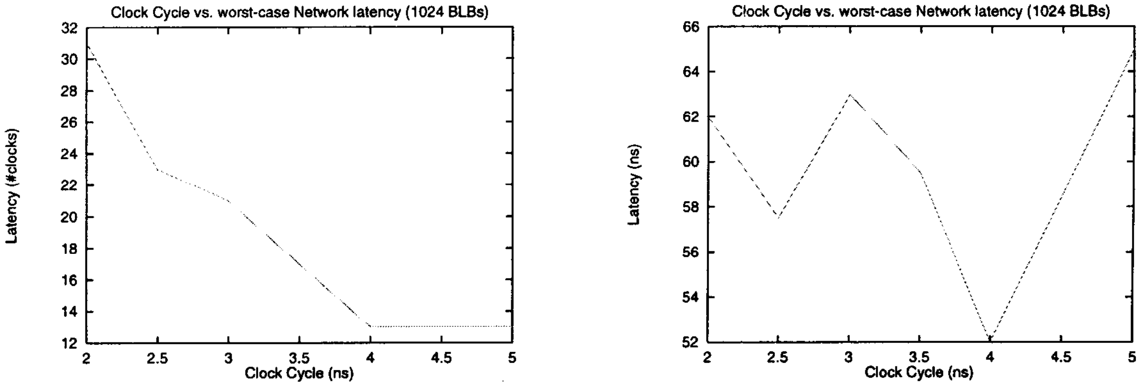


Figure 7: Area as a function of Target Clock Cycle



N.B. non-monotonicity in absolute time arises because we force clock breaks to occur at switchboxes (or at regular intervals between a pair of clocked switchboxes).

Figure 8: Latency as a function of Target Clock Frequency

putation (C-slow); even running only two data streams through the hardware can more than compensate for the 25% reduction in single stream throughput. When no data-parallelism exists, the latency represents a net loss in performance to the application compared to the unpipelined case.

Area Cost The addition of pipelining resources does have a cost in area. Table 5 compares several design points from our implementation. The table shows the effects of the input retiming bank and pipelined interconnect. Our unpipelined design without retiming requires over $4M\lambda^2$ per BLB which is extremely large compared to conventional 4-LUT-based FPGAs; we expect that several factors in the current network design and layout are responsible for this discrepancy and merit serious redesign. We see that adding both an 8-deep input retiming bank and pipelining the interconnect to support a 4 ns cycle cost us 50% area overhead per BLB compared to the design with no pipelining or input retiming. Note that our registered switchpoints are $62.5K\lambda^2$, or about 5× the area of our pass transistor switchpoints at $12.5K\lambda^2$. The difference in switch area, however, does not translate directly into array area at all levels of the tree. At the lowest tree levels switch area dominates wire areas so the difference is seen directly. However, as we get higher in the tree, wires dominate switches so differences in switch areas have a less significant impact on overall device size.

Latency Hindsight In hindsight, it is clear that the particular design point we have been studying pays an excessive latency penalty. From Figure 8, we see that crossing a 1024 logic block array can take 52 ns, or about 5× the worst-case delay in the Altera architecture (10.6 ns from Table 4). We focussed initially on a complete 2-ary tree rather than a flatter tree. As a consequence, our routes pass through many serial switches crossing this network (21), while the Altera design uses a flatter hierarchy where it needs to only cross through 3 or 4 series switches. Consequently, we believe the current absolute latency penalty is a function of the detailed network architecture rather than pipelining and can be tamed if given proper attention during design. This latency penalty, in turn, forces us to register more often to achieve the high cycle rate and hence is partially responsible for the overhead area required in the network for pipelining.

Power With all the added pipelining resources, clock power is a potential concern. From our proptotype layout, we burn 900pJ in each 64 BLB subarray on each clock cycle. On average, this means each BLB consumes 15pJ/cycle; this figure includes all of the BLB's input retiming registers, its output registers, and an amortized portion of the clocked interconnect. For comparison, a Xilinx XC4000XL flip-flop consumes 8-9pJ of clock cycle when in use [12]. For large HSRA arrays, the clock energy per BLB per cycle

	X	Y	BLB	No Registers	Register 250MHz
BLB composition	μm	μm	Area in λ^2	λ^2/BLB	λ^2/BLB
no input retiming	80	160	320K	4.3M	6.3M
8-deep input retiming	160	200	800K	4.6M	6.3M

Table 5: Effects of Interconnect Pipelining and Input Retiming on Area/BLB (Comparison based on 1024 BLB array)

will be higher due to the amortized contribution of additional, upper level switching stages, but it should remain in this basic ballpark for this technology. The Xilinx device can conserve power by disconnecting unused flip-flops from the clock tree while our prototype cannot.

6 Retiming and Pipelined Interconnect

After placing a design on the physical interconnect, we usually discover that there are more register delays between some LUT pairs in the mapped design than there were in the original netlist. To obtain correct logical behavior, we must now retime the design so that we can accommodate the presence of these delays in the design. That is, we must find a logically correct way to move registers around in the design, adding registers as necessary, so that the logical net will have adequate registers between every pair of LUTs to cover the necessary pipeline delays in interconnect. Our retiming problem becomes: *retime all LUTs such that the number of registers between any two LUTs is greater than or equal to the number of registers required by the interconnection network.* We can transform this problem to a traditional retiming problem as follows (See Figure 9):

1. For every LUT→LUT connection over the network, place a linear cascade of buffers between the LUTs of length equal to the number of registers required by the interconnection network. Note that these added buffers are purely a logical construct used to model the delay associated with each segment of piece of pipelined interconnect.
2. Retime the resulting network to run at single-LUT (or buffer) delay between registers using conventional retiming algorithms as in Section 2.
3. Remove the extra buffers, but keep the additional registers added to cover interconnect delays and provide data alignment. As we saw in Figure 4, a common outcome of this retiming is that one input to a LUT will arrive just as needed, requiring no additional input retiming registers. Other inputs to the LUT will be forced to await the arrival of this final LUT input. As a consequence, these inputs will end up with more registers between the source and sink LUT than there are interconnect pipeline delays (See rightmost case in Figure 9).
4. Set the input retiming registers on the BLB to accommodated any of the extra registers required to align timing on LUT inputs.

Note that this algorithm does not place an upper limit on the number of registers which may be placed between LUTs. In practice, we will have a finite input buffer depth, so as a post pass it is necessary to break out retiming depths greater than the input depth and allocate additional logic blocks to cover these long inputs. If no BLB is available to place these extra retiming blocks, it may be necessary to iterate placement in order to create room for the extra retiming BLBs.

Post Placement Retiming Requirements Table 6 shows the resulting input depths after placing and retiming the benchmark set from Section 2 and subtracting out the registers which are covered by the network. Once again, single register depth is the single largest component. However, on average the input depth is 4.7. The distribution above one is much flatter with over 9% of the register depths greater than 10.

Finite Input Retiming Depth Figure 10 shows the effects of running a post pass to break up delays and allocate additional BLBs to accommodate deep data retiming requirements. We see, on average, a depth of 16 has a less than a 10% BLB overhead, with the worst case being 100% for one benchmark. To accommodate deep retiming, the input registers can be chained together to provide deeper retiming of fewer signals (See top middle of Figure 1). Using this feature a single d-deep, k-input logic block can act as a $(k \cdot d)$-input retiming chain. Figure 10 shows the BLB overhead both using and not using this feature. The data suggest that a modest input depth (4–8) with chaining typically leaves us with only 20-50% BLB overhead resulting from finite retiming buffer depth. We used an 8 deep retiming bank for our test chip design point since it did not result in a significantly larger design than 4 deep retiming bank.

Retiming Requirements Caveat There are two reasons to believe these retiming results represent a pessimistic upper bound on the the retiming requirements for a properly optimized design:

1. Our network has excessive latency which one should be able to reduce.
2. The placement routines used did not attempt to minimize interconnect delays between LUTs.

7 Assessment and Conclusions

We have shown that it is possible to engineer arrays which run at cycle rates comparable to processors. We use the same techniques common in the processor and logic design—pipelining and data retiming. We have further shown that with these techniques we can automatically map and run designs which use the devices profitably at this rate for feed-forward and data-parallel applications.

Our effort to prototype this architecture achieved a 4 ns cycle rate at a cost of roughly 1.5× area over an unpipelined design. Add to this the 20% BLB overhead, and we pay less than 2× the area of the unpipelined design. Since this allows us to run many pipelineable designs at frequencies of 2–17× their unpipelined frequency, this represents a superior area-time point for high-throughput designs. Furthermore, the architecture supports cyclic designs in C-slow manner, extracting greater functionality from the silicon when multiple independent tasks can be interleaved. There is, nonetheless, evidence that our current design point has excessively high area and latency; this suggests that a properly optimized architecture which uses these principles can achieve even greater density than our prototype.

Obvious areas for improvement and future work include:

- Quantify network costs of output retiming versus input retiming.
- Improve network design to reduce area and latency and re-evaluate effects on retiming requirements.

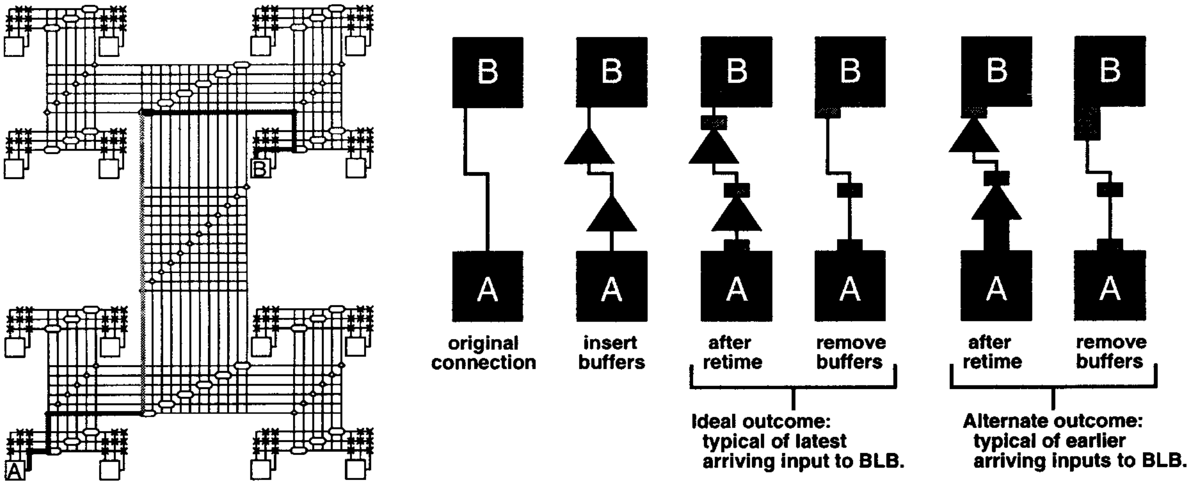

B
A
B
A
B
A
B
A
B
A
B
A
B
A
original connection
insert buffers
after retime
remove buffers
after retime
remove buffers
Ideal outcome: typical of latest arriving input to BLB.
Alternate outcome: typical of earlier arriving inputs to BLB.

Figure 9: Transforming network so that retiming will guarantee to cover interconnect delays

Number of Registers	1	2	3	4	5	6	7	8	9	10	> 10
Percentage	60	6.9	5.9	3.8	4.3	2.7	2.6	1.9	1.5	1.2	9.2

Table 6: Benchmark-Wide Distribution of Required Input Register Depth after Placement and Retiming

- Develop delay- or cycle-oriented placement and assess effects on retiming requirements.

Further, the important application questions remains:

- Can most tasks with low-throughput requirements be readily serialized to take commensurately less area with such an architecture (*e.g.* use bit-serial datapaths)?
- How often are cyclic constraints both unavoidable and not amenable to profitable C-slow execution (*i.e.* no available data parallelism)?

If low throughput tasks cannot be serialized, the area overhead paid for a high clock rate will go to waste. For data-dependent cycles, the clocking latency penalty will reduce single-stream throughput. Both of these cases would be mitigated by multiple context designs where the higher clock rate enabled by the pipelining techniques introduced here allows us to share interconnect and compute resources in time. Ultimately, these are the questions we need to understand in order to decide when it is best to build single-context architecture like the HSRA and when it is more profitable to couple these techniques with multicontext designs so that the active silicon can be rapidly reused to perform different tasks.

Acknowledgements

This research is part of the Berkeley Reconfigurable Architectures Software and Systems effort supported by the Defense Advanced Research Projects Agency under contract numbers F30602-94-C-0252 and DABT63-C-0048.

Jason Cong's VLSI CAD Lab at UCLA provided the Flowmap implementation used to map LUTs here.

Thanks to Andreas Koch for providing feedback on an early drafts of this paper.

References

[1] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020. *FLEX 10K Embedded Programmable Logic Family*, ver. 3.11 edition, May 1998. `<http://www.altera.com/documents/ds/dsf10k.pdf>`.

[2] Narasimha B. Bhat, Kamal Chaudhary, and Ernest S. Kuh. Performance-Oriented Fully Routable Dynamic Architecture for a Field Programmable Logic Device. UCB/ERL M93/42, University of California, Berkeley, June 1993.

[3] Douglas Chang and Malgorzata Marek-Sadowska. Buffer Minimization and Time-multiplexed I/O on Dynammically Reconfigurable FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 142–148, February 1997.

[4] Mustafiz Choudhury and James Miller. A 300MHz CMOS Microprocessor with Multi-Media Technology. In *1997 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 170–171. IEEE, February 1997.

[5] Jason Cong and Yuzheng Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Transactions on Computer-Aided Design*, 13(1):1–12, January 1994.

[6] André DeHon. DPGA Utilization and Application. In *Proceedings of the 1996 International Symposium on Field Programmable Gate Arrays*. ACM/SIGDA, February 1996. `<http://www.ai.mit.edu/projects/transit/dpga/dpga-use-fpga96.ps.Z>`.

[7] Bruce Gieseke, Randy Allmon, Daniel Bailey, Bradley Benschneider, Sharon Britton, John Clouser, Harry Fair III, James Farrell, Michael Gowan, Christopher Houghton, James Keller, Thomas Lee, Daniel Leibholz, Susan Lowell, Mark Matson, Richard Mathew, Victor Pen, Michael Quinn, Donald Priore,

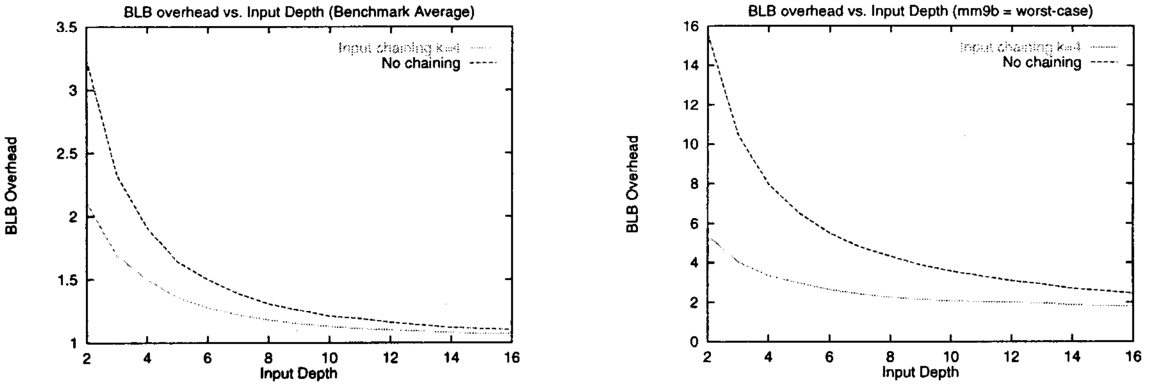


Figure 10: Effects of limited Input Depth

Michael Smith, and Kathryn Wilcox. A 600MHz Superscalar RISC Microprocessor with Out-of-Order Execution. In *1997 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 176–177. IEEE, February 1997.

[8] Ronald Greenberg. The Fat-Pyramid and Universal Parallel Computation Independent of Wire Delay. *IEEE Transactions on Computers*, 43(12):1358–1365, December 1994.

[9] David Greenhill, Eric Anderson, James Bauman, Andrew Charnas, Rakesh Cheerla, Hao Chen, Manjunath Doreswamy, Phillip Ferolito, Srinivasa Gopaladhine, Kenneth Ho, Wenjay Hsu, Poonacha Kongetira, Ronald Melanso, Vinita Reddy, Raoul Salem, Harikaran Sathianathan, Shailesh Shah, Ken Shin, Chakara Srivatsa, and Robert Weisenbach. A 330MHz 4-Way Superscalar Microprocessor. In *1997 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 166–167. IEEE, February 1997.

[10] 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, Digest of Technical Papers*, pages 222–223. IEEE, February 1996.

[11] Brian Von Herzen. Signal Processing at 250 MHz using High-Performance FPGA's. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 62–68, February 1997.

[12] Xilinx Inc. XC4000XL Power Calculation. *XCell*, 1998. <http://www.xilinx.com/xcell/xl27/xl27_29.pdf>.

[13] 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.

[14] Charles Leiserson, Flavio Rose, and James Saxe. Optimizing Synchronous Circuitry by Retiming. In *Third Caltech Conference On VLSI*, March 1993.

[15] James Montanaro, Richard Witek, Krishna Anne, Andrew Black, Elizabeth Cooper, Dan Dobberpuhl, Paul Donahure, Jim Eno, Alejandro Farell, Gregory Hoeppner, David Kruckemyer, Thomas Lee, Peter Lin, Liam Madden, Daniel Murray, Mark Pearce, Sribalan Santhanam, Kathryn Snyder, Ray Stephany, and Stephen Thierauf. A 160MHz 32b 0.5W CMOS RISC Microprocessor. In *1996 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 210–211. IEEE, February 1996.

[16] Jonathan Rose, Robert Francis, David Lewis, and Paul Chow. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency. *IEEE Journal of Solid-State Circuits*, 25(5):1217–1225, October 1990.

[17] Stephen Scalera and Jóse Vázquez. The Design and Implementation of a Context Switching FPGA. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 78–85, April 1998.

[18] 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, May 1992.

[19] Steve Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A Time-Multiplexed FPGA. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, April 1997.

[20] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *XC4000XL Field Programmable Gate Arrays*, version 2.0 edition, February 1998.