FPGA '96 -- ACM/SIGDA Fourth International Symposium on FPGAs
February 11-13, 1996, Monterey, CA

Entropy, Counting, and Programmable Interconnect

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

Abstract

Conventional reconfigurable components have substantially more interconnect configuration bits than they strictly need. Using counting arguments we can establish loose bounds on the number of programmable bits actually required to describe an interconnect. We apply these bounds in crude form to some existing devices, demonstrating the large redundancy in their programmable bit streams. In this process we review and demonstrate basic counting techniques for identifying the information required to specify an interconnect. We examine several common interconnect building blocks and look at how efficiently they use the information present in their programming bits. We also discuss the impact of this redundancy on important device aspects such as area, routing, and reconfiguration time.

1 Introduction

Despite the fact that programmable devices (*e.g.* FPGAs) are marketed by the number of gates they provide, a device's interconnect characteristics are the most important factors determining the size of the programmable device and its usability. When designing the interconnect for a programmable device we must simultaneously address several important, and often conflicting, requirements:

1. Provide adequate flexibility, allowing the realization of a sufficiently large space of interesting interconnections
2. Efficiently specify interconnect behavior, minimizing the space and time required to configure the interconnect
3. Balance device bisection bandwidth with available or allowable spatial constraints
4. Minimize interconnection delays

In this paper we focus on the size and redundancy of our programmable interconnect description (requirement number 2). In the process of discussing interconnect descriptions, the paper broadly addresses interconnect flexibility.

Fundamentally, interconnect description memory need not grow as quickly as wire and switch requirements. This leaves us with design points which are generally wire and switch limited. We may, consequently, encode our interconnect description sparsely when memory area is nearly free and wiring channels are determining die size. Since interconnect memory grows slowly compared to switch and wire requirements, interconnect memory need never dictate die size for large, single-context, reconfigurable components.

As we begin to make heavy use of the reconfigurable aspects of programmable devices, device reconfiguration time becomes an important factor determining the performance provided by the part. In these rapid reuse scenarios, interconnect encoding density can play a significant role in determining device area and performance.

1. One technique for reducing the reconfiguration time is to store multiple, on-chip contexts. Since designs tend to be wire and switch limited, multiple on-chip contexts need not substantially increase die area. Multi-context designs, however, generally merit more dense interconnect encodings than single-context designs in order to minimize the impact which configuration memory has on die size. For small numbers of contexts, one is, in effect, sacrificing some encoding sparcity for extra, on-chip contexts. For large numbers of contexts, configuration memory, and hence encoding density, can dictate device size.
2. Off-chip context reloads for single- or multi-context devices are slow because a large amount of configuration data (typically, $> 10^5$ bits) must be transfered across a limited bandwidth i/o path. Transferring sparsely encoded bitstreams across this i/o bottleneck exacerbates already poor reload performance dictated by i/o bandwidth limitations.

This paper starts out by identifying a simple metric for characterizing interconnectivity – a count of the number of "useful" and distinct interconnection patterns. While simple, this metric turns out to be difficult to calculate in the general case. We can, nonetheless, analyze a number of common structures to obtain bounds on the number of patterns provided by more complicated topologies. From this analysis we observe that conventional, programmable architectures have highly redundant programming bit streams. The analysis helps us identify opportunities to save programmable memory area and lower reconfiguration time by reducing that bit stream redundancy.

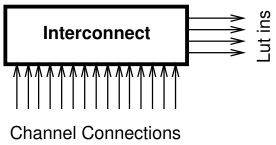

Figure 1: DPGA LUT Input Connection

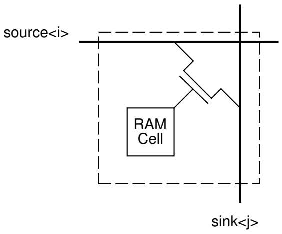

Figure 2: Crossbar Crosspoint with Memory

In the next section, we open with a motivational example to illustrate the issue and analysis. In Section 3, we define the interconnection metric more precisely. In Section 4, we look at some basic interconnection building blocks which can be easily analyzed in isolation. Section 5 looks at some ideal or pedagogical network structures by composing the primitives from Section 4. Section 6 touches briefly on the role of placement in interconnect flexibility. Section 7 looks at conventional interconnect examples using the results of Section 5 to estimate the amount of redundancy these devices exhibit. We discuss the implications and impact of this redundancy in more detail in Section 8 before concluding in Section 9.

2 Motivational Example

Consider a case where we wish to drive any of n sources onto each of m sinks. In our DPGA prototype [7], for example, we needed to drive the 4 inputs to the 4-LUT from the 15 lines which physically converged upon the LUT cell ($m = 4$, $n = 15$, See Figure 1). This kind of structure is typical when connecting logic block inputs to a routing channel.

We could provide full connectivity by building a full crossbar between the $n = 15$ sources and $m = 4$ sinks. This requires $m \times n$, 60 in this case, crosspoints. If each crosspoint is built with its own memory cell, as shown in Figure 2, this arrangement entails equally many memory cells.

At this point it is worth noting that there are $2^{60} > 10^{18}$ ways to program 60 memory cells, but many of those connections are not useful. In fact, of the n memory cells along each input wire, at most one should enable its crosspoint at any point in time. Since each output should be driven by one source, there are, in fact, only n^m, $(15)^4 = 50,625$ in this case, combinations which may be useful. We can, in fact, control the $n \times m$ crosspoints with only $\lceil m \log_2(n) \rceil$ memory cells, or 16 in this case. Here, each group of $\lceil \log_2(n) \rceil$ bits serves to select which of the n inputs should drive onto each of the m output lines (See Figure 3).

If, as is the case for the DPGA, the logic block is a k-input lookup table, even this arrangement provides more routing flexibility than is necessary. There is, for instance, no need for any of the k inputs

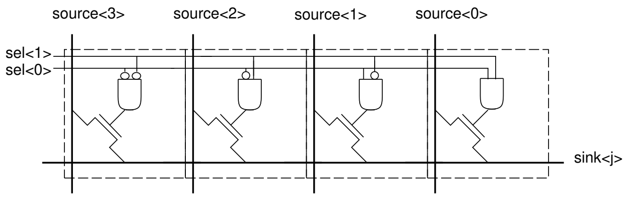

Figure 3: Encoded Crossbar Control

to be the same. Strictly speaking there are at most $\binom{n}{m}$ (in this example, $\binom{15}{4} = 1365$) distinct selections of the $k = m$ inputs from the n sources. This implies we could get away with as few as 11 ($\lceil \log_2(1365) \rceil$) control bits if we included heavy decoding.

From this calculation we can also conclude that any combinations provided beyond the 1365 distinct interconnect patterns entailed by the 15 choose 4 combinations are redundant and offer us no additional functionality. This bound is often useful in understanding the gross characteristics of an interconnect. All interconnects which provide all these combinations are functionally equivalent. Above this point we can compare the number of specification bits to the number of requisite specification bits to understand the redundancy in the encoding scheme. For interconnects providing less distinct interconnect patterns, we can compare the number of achievable interconnect patterns to the maximum number of distinct interconnect patterns to get a scalar metric of the flexibility of the interconnect.

In practice, the DPGA prototype used 4, 8-input muxes. This gave it only 32 crosspoints requiring $4 \times \log_2(8) = 12$ bits to specify the input. Due to the limited size input muxes, it only supported 1217 distinct input combinations. Our encoding is within one bit of the theoretical minimum of 11 and routes 89% of the 1365 combinations identified above.

With respect to memory bits, we see that dense encoding of the combinations yields a more significant reduction in requisite memory cells than depopulating the crossbar in a memory-cell-per-crosspoint scenario. In an unencoded case, 12 memory bits would not even support a single connection from each of the 15 potential inputs.

Of course, minimizing the number of memory bits does not, necessarily, produce the smallest layout since dense encodings must be decoded and routed. We will visit this issue in Section 8.

This example underscores several issues which are worthwhile to understand when designing programmable interconnect.

- Independently programming each crosspoint leads to highly redundant interconnect programming, including a large space of non-useful configuration specifications.
- The number of bits required to configure an interconnect can be much lower than the number of pass gates or crosspoints in the interconnect.
- The number of useful configurations provided by a piece of interconnect cannot be determined by looking at the interconnect in isolation. When we consider the context in which the interconnect is used, many of its configuration may become redundant.

- Simple bounds on the theoretical encoding density and required flexibility are helpful in evaluating the gross merits of various proposed interconnect solutions.

3 Interconnection Metric

The interconnection metric we are focusing on here is to count: *the number of unique and useful connection patterns which the network can realize.*

Useful Connections With sparse encoding, many bit combinations may be parasitic or non-useful. In the “bit-per-crosspoint” example above, we saw that it was not useful to have more than one crosspoint enabled along an output line. In these cases, a large portion of the entire bit stream encoding space refers to interconnection patterns which are at best nonsensical and may even be damaging to the part.

Unique Connections An interconnection pattern which provides the *same* functionality as another specifiable pattern adds no capability to the interconnect. That is, the flexibility which allows both specifications provides no additional interconnect power. In the example above, we saw that a full crossbar provided $(15)^4 = 50,625$ specifiable connections. However, with the connections feeding into a 4-LUT, a pattern which bring inputs 15, 13, 10 and 7 into the LUT is no different from a pattern which brings inputs 10, 15, 7 and 13 into the LUT. The full crossbar thus provides $50,625 - 1,365 = 49,260$ redundant patterns which add no functionality to the interconnect.

It is worthwhile to note that interconnect “flexibility” as used here can be applied to any interconnection network or family of graphs. As such it is very different from the interconnect flexibility defined in [6] [3] which is used to describe the level of population of switches in a particular interconnect family.

Interconnection Patterns It is important that we look at the interconnection patterns as a whole to understand which patterns are functionally identical. Again, looking at the k-LUT input selector in isolation from the logic block, we can identify more distinct patterns than we see when viewed in an ensemble.

It is also significant that the metric looks at the ensemble of interconnections feasible rather than looking at the interconnect flexibility afforded to a single input in isolation. Since resources are typically shared, the focus on patterns accounts for the limited availability of shared resources.

4 Basic Primitives

In this section we identify a few basic primitives used in building interconnection networks. We count the number of interconnection patterns they provide, as well as relating the abstract primitives to their more physical implementations.

Muxes An n-input multiplexor (mux) can connect any of its n inputs to its output. In isolation, the multiplexor realizes n distinct interconnect patterns, requiring $\lceil \log_2(n) \rceil$ bits to specify its behavior.

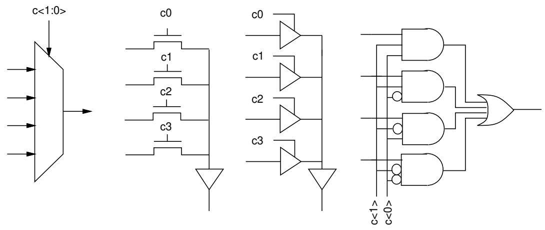

Figure 4: Multiplexor Symbol with Three Potential Implementations

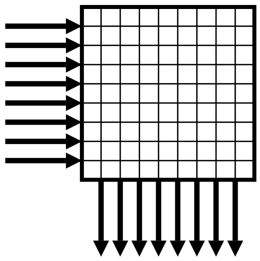

Figure 5: 8×8 Crossbar

Figure 4 shows a multiplexor and several possible implementations. Note that a series of tristate drivers attached to a single, physical wire serves logically as a multiplexor and can be treated as one for the purposes of analysis. This remains true whether the tristate drivers have separate or central control. The series of tristate drivers with distributed control has the additional ability to drive no value onto the line, but this offers no additional logical functionality since we can just as easily drive any value onto the output in such a case. The tristate drivers could also have multiple values driven simultaneously onto the output, but, unless this is used to perform a logical function, the multiple drive cases are parasitic rather than useful behavior.

Crossbars A crossbar has a set of n inputs and a set of m outputs and can connect any of the inputs to any of the outputs with the restriction that only one input can be connected to each output at any point in time. Logically, an $n \times m$ crossbar is equivalent to m n-input multiplexors. The crossbar can realize n^m distinct interconnect patterns and requires $\lceil m \log_2(n) \rceil$ bits to specify its behavior. The crossbar represents the most general kind of interconnect and can often be used to calculate an upper bound on the flexibility which could be offered by a piece of interconnect.

A small crossbars is shown in Figure 5. We have already seen some potential implementations in Figures 2 and 3.

Subset Selection With subset selection, we select a group of m outputs from n inputs. We saw this selection when routing the k inputs to a k-LUT. This kind of selection is also typical when concentrating outputs from one region down to a limited size channel connecting to another region of the component. As we saw in Section 2, this function is not as trivially implemented as the crossbar, but the function is frequently desired making this primitive highly useful for analysis. Subset selection entails $\binom{n}{m}$ distinct interconnection patterns and requires $\left\lceil \log_2 \binom{n}{m} \right\rceil$ bits to specify its behavior.

Pass gates Individual pass gates are too primitive to generally be useful from an analysis standpoint. Alone, each pass gate has 2 interconnection states which can be specified with one bit. From an analysis standpoint it is generally more useful to group collections of pass gates together into a larger structure. As noted above, when the direction of signal flow is apparent, groups of pass gates used to selectively drive onto a single line effectively form a logical multiplexor.

5 Composition Examples

In this section we apply and compose the primitives of the previous section to examine a few families of netlists and networks of general interest.

Netlists of k-input logic blocks Let us consider the family of netlists with n logic blocks where each logic block has at most k inputs. Further, the netlist has i inputs and o outputs. To get an upper bound, we assume all logic blocks may provide a distinct function. In general, each of the logic block inputs may want any of the n logic block outputs or any of the i inputs. Each output may come from any of the n logic block outputs. There are $(i+n)^{(n\cdot k)}$ possible interconnect patterns for the $(n \cdot k)$ logic block inputs and n^o interconnection patterns for the outputs. All together, there are $n^o \cdot (i+n)^{(n\cdot k)}$ interconnection patterns which the netlist may exhibit. Without exploiting placement (See Section 6), if all interconnects are given equally long descriptions, it will require at least $\lceil o \log_2(n) \rceil + \lceil n \cdot k \log_2(i+n) \rceil$ bits to describe each interconnection patterns.

Network for k-input logic blocks In a similar manner, we can look at a device or interconnect and bound the number of interconnection patterns it provides. Again, assuming we have n logic blocks with k inputs each along with i inputs and o outputs, if we do not allow inputs to be directly connected to outputs, we have a total number of interconnection patterns of, at most:

$$N_{no\_direct\_io\_patterns} \leq n^o \cdot (i+n)^{(n\cdot k)} \qquad (1)$$

If inputs can be directly connected to outputs, we have $(i+n)^o$ possible output patterns for a total number of interconnection patterns bounded by:

$$\begin{aligned} N_{direct\_io\_patterns} &\leq (i+n)^o \cdot (i+n)^{(n\cdot k)} \\ &\leq (i+n)^{(n\cdot k+o)} \end{aligned} \qquad (2)$$

For many devices, each inter-chip i/o pin can serve as either an output with enable or an input. We thus let $o = 2i$, assuming we may have to route both the output and the enable for each physical pin, and have a total of at most $(i+n)^{(n\cdot k+2i)}$ interconnection patterns. The interconnect can thus be specified with $\lceil (n \cdot k + 2i) \log_2(i+n) \rceil$ configuration bits.

Element	Growth
Number of Logic Blocks	n
Logic Block Area	$O(n)$
Logic Block Configuration	$O(n)$
Interconnect Configuration	$O(n\log(n))$
Crosspoints/switches	$O(n^2)$
Wire	$O(n^2)$

Table 1: Growth Rates for Key Elements of a Fully Flexible Programmable Device

From this and the previous example, it is worth observing that network configuration requirements are growing as $O(n\log(n))$ — assuming k is fixed and i grows at most linearly in proportion to n. We can compare this to the configuration requirements for the logic which are growing as $O(n)$. We can also compare this with the number of crosspoints or total length of interconnect wire which is growing as $O(n^2)$ if we are going to provide the full interconnect to support any of the possible n-node netlists.

These general observations, summarized in Table 1, are worthwhile to remember. Asymptotically, they tell us wires and crosspoints will be the pacing items for offering flexibility in programmable devices. They also tell us to expect interconnect programming to grow faster than logic cell programming. We can describe any interconnect much more cheaply than we can spatially route it, underscoring the need to reuse physical crosspoints and wires in time as we build larger programmable devices and systems.

Network of k-LUTs If a device's network is built entirely out of k-LUTs, we can get a slightly tighter bound on the number of interconnection patterns provided. Again, our source of inputs is the n LUT outputs and the i inputs. Each LUT must choose k inputs from $(i+n)$ sources. Together, this makes for $\binom{i+n}{k}^n$ total interconnection patterns. If we also assume $o = 2i$ and outputs can come directly from inputs, the total number of interconnection patterns is at most:

$$N_{lut\_interconnect\_patterns} \leq \binom{i+n}{k}^n (i+n)^{(2i)} \qquad (3)$$

The difference between this case and the previous is that the network, sans output connections, has at most $\binom{i+n}{k}^n$ patterns rather than $\left((i+n)^k\right)^n$. The ratio between these two expressions is:

$$\left(\frac{(i+n)^k}{\binom{i+n}{k}}\right)^n$$

For large $(n+i)$ and small k, $(n+i)-1 \approx (n+i)-k \approx (n+i)$. This ratio is then roughly $(k!)^n$. In terms of configuration bits, this amounts to $n\log_2(k!)$ — a small savings linear in n for fixed k. *e.g.* for $k=4$, one can save at most 4-5 network configuration bits per LUT by exploiting the input pin equivalences. Referring back to

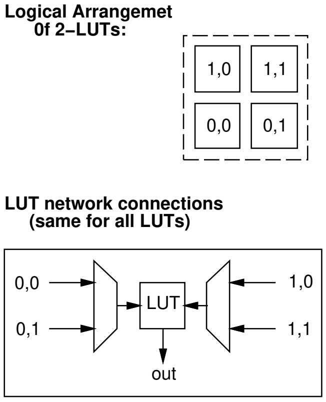

Figure 6: Limited Interconnect Network of 4, 2-LUTs

Table 1, this tells us that exploiting LUT input equivalences saves us $O(n)$ configuration bits, but does not, fundamentally, change the growth rate for interconnect configuration.

6 Placement Freedom

Once we identify a level of desired interconnect flexibility, it is not necessary for the physical interconnection network to solely provide that flexibility. In programmable devices, we also have freedom in where we place functions and results within the interconnection network. The net result of this freedom is that we can achieve a given flexibility $N_{patterns}$ with a network which provides fewer than $N_{patterns}$ interconnection patterns. This also means we could, in theory at least, use less than $\log_2\left(N_{patterns}\right)$ bits to specify the interconnect pattern when we observe that the placement of functions and results relative to the network also gives us a degree of specification freedom.

To make this concrete, let us consider a very simple interconnection problem where we wish to interconnect 4 2-input LUTs ($n = 4$, $k = 2$). We know there are $\begin{pmatrix} 4 \\ 2 \end{pmatrix}^4 = 1296$ possible interconnection networks for this small example.

First, we consider a limited interconnection network where we arrange the four LUTs into a 2×2 array. Each LUT has one input which may come from either LUT in the first column and a second input which may come from either LUT in the second column (See Figure 6). Each of the $(2 \cdot 4) = 8$ inputs in this restricted interconnect can come from one of two places, making for a total interconnect flexibility of $2^8 = 256$. It is also worthwhile to note that all 256 interconnect combinations are distinct. If we had fixed the placement of the 4 logic functions in the array, then we could only realize these 256 interconnect patterns. Allowing the functions to be arranged within the array allows greater flexibility. We know there are $4! = 24$ ways to place 4 logic functions in the 2×2 array. Because of the high symmetry of the physical network, it turns out that groups of 8 permutations are equivalent with respects to routing. As a result there are 3 distinguishable placement classes. This could provide us with at most $3 \times 256 = 768$ interconnection patterns if all of the permuted interconnects were distinct. In practice, this gives us 720 of the 1296 possible patterns.

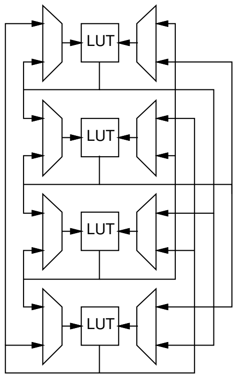

Figure 7: Less Symmetric, Limited Interconnect Network of 4, 2-LUTs

Using an alternate interconnection scheme with less symmetry, shown in Figure 7, we can get 6 distinguishable placement classes and achieve 1104 of the 1296 possible patterns. The 1104 interconnection patterns are, of course, over a factor of four more patterns than the 8 bits of interconnect programming could specify, alone.

This example underscores the fact that placement in asymmetric networks allows us to expand the number of realizable networks for a given, limited, physical interconnect. Additionally, we see that successfully routing networks in this scheme requires picking the correct permutation for the network (placement) and then the correct interconnection pattern (routing). Asymmetry in the network can have the effect of both increasing the flexibility, by making more interconnection patterns realizable, and of making the routing problem harder, by offering a larger space of distinct placement classes to explore during routing. In general, the extent to which placement can reduce the demand for interconnect resources, including wires, switches, and configuration bits, remains an open issue.

7 Some Conventional Architectures

Table 2 summarizes the major characteristics for several contemporary programmable devices. Additionally, a pure 4-LUT design is included for sake of comparison with the industrial offerings. We can make a rough, back-of-the-envelope-style computation on the bits required to configure the network by making the assumption that the network does support all potential networks without exploiting placement flexibility. That is, we assume that the basic logic blocks are fully connected. Since the conventional offerings are far from being fully interconnected, this gives us an upper bound on the number of network configuration bits. Adapting Equation 2 and taking the base two logarithm to convert to bits, we get:

$$\begin{aligned} N_{net\_bits} = & \\ & \lceil (n_{block\_ins} \cdot n_{blocks} + n_{io\_ins} \cdot n_{io}) \\ & \times \log_2\left(n_{block\_outs} \cdot n_{blocks} + n_{io\_outs} \cdot n_{io}\right) \rceil \end{aligned} \quad (4)$$

We can also calculate the number of bits required to specify the logic block functions in the obvious manner:

$$N_{logic\_bits} = n_{block\_logic\_bits} \cdot n_{blocks} \quad (5)$$

Part	n_{blocks}	n_{io}	Programming Bits	Reference
Xilinx xc4013	578 CLBs	192 IOBs	240K	[9]
Xilinx xc5210	324 CLBs	196 IOs	160K	[10]
Altera EPF81188	1008 LEs	184 IOEs	192K	[2] [1]
Pedagogical Reference	1024 4-LUTs	200	–	

Family		n_{block_ins}	n_{block_ous}	n_{io_ins}	n_{io_outs}	$n_{block\_logic\_bits}$
XC4K	CLB	13	4	4	2	$2 \times 2^4 + 2^3 = 40$
XC5K	CLB	4×4	4×2	2	1	$4 \times 2^4 = 64$
Altera 8K	LE	4	1	1	1	$2^4 = 16$
Reference	4-LUT	4	1	2	1	$2^4 = 16$

Table 2: Parameters for a Sampling of Contemporary Programmable Devices

Part	N_{net_bits}	N_{logic_bits}	Programming Bits	Unaccounted Control Bits
Xilinx xc4013	92K	23K	240K	non-logic CLB/IOB configuration, edge decoders
Xilinx xc5210	62K	20K	160K	non-logic CLB/IOB configuration
Altera EPF81188	42K	16K	192K	LAB control and Peripheral Bus
Pedagogical Reference	45K	16K	–	–

Table 3: Configuration Space versus Bit Stream Size

Table 3 summarizes the results of these basic calculations for the identified components.

The comparison is necessarily crude since vendors do not provide detailed information on their configuration streams. However, we expect the unaccounted control bits in Table 3 to not be more than 10% of the total device programming bits. With this expectation, we see that these devices exhibit a factor of two to three more interconnect configuration bits than would be required to provide full, placement-independent, logic block and i/o interconnect.

We can, of course, derive a tighter bound for the reference 4-LUT design by adapting Equation 3:

$$N_{net\_bits} = \left\lceil n_{blocks} \log_2 \left(\begin{array}{c} n_{blocks} + n_{io} \cdot n_{io\_outs} \\ k \end{array} \right) \right\rceil + \lceil n_{io} \cdot n_{io\_ins} \log_2 (n_{blocks} + n_{io} \cdot n_{io\_outs}) \rceil$$

For the 1,024 4-LUT case identified above, $N_{net\_bits} \approx 40K$, saving roughly 5 bits per LUT as suggested in Section 5. Eight of the 13 inputs on the XC4K go into 4-LUTs and all 16 of the XC5K inputs go into 4-LUTs. Consequently, all of these arrays require comparably fewer network configuration bits. The 4 inputs on the Altera 8K LE also go into a 4-LUT. Since one input may also be used in a control capacity, the reduction is slightly lower for the Altera 8K part.

8 Impact of Configuration Density

Area The total, on-chip configuration memory can be one of the major contributors to chip area (*e.g.* [4]). As such, redundancy in the configuration space may cost additional die area. The effect, however, is technology and design point dependent. When the design is active silicon area limited, the configuration size can play a large factor in determining design size. However, when the design is wire limited, the redundant configuration memory may be free or negligible.

As we see in Table 1, we ultimately expect devices to be wire or switch limited. In the wire limited case, we may have free area under long routing channels for memory cells. In fact, dense encoding of the configuration space has the negative effect that control signals must be routed from the configuration memory cells to the switching points. The closer we try to squeeze the bit stream encoding to its minimum, the less locality we have available between configuration bits and controlled switches. In Figure 3, for instance, we had to run additional control lines into the crossbar to control the crosspoints. These control lines compete with network wiring, exacerbating the routing problems on a wire dominated layout.

In a multiple context or switched interconnect case, the effects of memory density are more pronounced. Limited wire resources are reused in time, making more efficient use of the wires and minimizing the effects of bisection wiring limitations. In these cases the chip needs to hold many configurations worth of memory simultaneously. If one is not careful about the density of the interconnect configuration encoding, the configuration memory stores can dominate chip area.

In the aforementioned DPGA Prototype [7], for example, even with four on-chip contexts, wiring and switching accounted for over half of the die area. Network configuration memory made up about one fourth of the area. A factor of 2-4 increase in the configuration memory due to sparser encoding would have forced a substantial (20-60%) increase in die area.

Performance In conventional, single context FPGAs, the configuration inputs to switches are static during normal operation. Any time associated with decoding switch control from memories is not in the critical path and will not affect the timing on signal flow through the interconnect. In the rapidly switched, multiple context case, this is less true. Memory access and decoding time define the overhead required to change between context configurations.

Reconfiguration Time The reconfiguration time of both single- and multi-context reconfigurable devices is directly impacted by encoding density. As we have seen in Section 7, contemporary reconfigurable devices have very large configuration bit streams. Due to physical i/o limitations, changing bit streams is an expensive operation. In cases of heavy device reuse this reload time can have a significant effect on system performance (*e.g.* [5] [8]).

Of course, the real problem associated with reconfiguration time is the i/o bandwidth limitation. It is certainly not necessary for the bits stored in the configuration memories to be *identical* to the off-chip interconnect specification or the specification transmitted across the chip boundary. In wire limited cases, where local memory cells are inexpensive or free and routing control signals are expensive, the device could decompress the configuration specification during configuration reload time.

For example, let us revisit the LUT interconnection example from Section 2 in the wire limited case. Here, we suppose the technology costs dictate that the memory-cell-per-crosspoint design is more area efficient than the denser encoding schemes identified that section. We could build a single copy of the decoder for the $\left\lceil \log_2 \binom{15}{4} \right\rceil = 11$ bit encoding scheme. This single decoder could then be used to decode each 11 bits of LUT input configuration into the 60 bits required to program the input crossbar. As a result, we reduce the LUT input portion of the bitstream by a factor of $\frac{60}{11} \approx 5.4$ over the unencoded case.

It is worthwhile to note at this point that the compression we are discussing here is *design independent*. That is, the bounds on configuration size we have derived throughout this paper are applicable across all possible interconnection patterns which a network may provide. It is also possible to exploit redundancy within the design to further compress the configuration bit stream in a design dependent manner.

Safety One effect of dense encoding is to eliminate parasitic configurations. As noted, configuration specifications which allow multiple drivers sharing a common line to be enabled simultaneously are general parasitic. It is, in fact, possible to destroy many conventional devices by uploading parasitic configurations.

9 Conclusions

Sparse interconnect configuration encodings can result in bloated bit stream configurations. Asymptotic growth rates suggest that dense encodings grow more slowly than desired interconnect requirements, placing designs in a wire and switch limited domain. Consequently, some encoding density may be judiciously sacrificed at the on-chip storage level to decrease control interconnect in programmable devices. Multiple-context components, on the other hand, have a greater demand for on-chip storage and merit denser encoding in order to make effective use of silicon area.

Sparsely encoded bit streams have their largest impact on configuration load time. As device size increases and the reconfigurable aspects of programmable devices are heavily exploited, the i/o bandwidth limited reconfiguration time becomes a significant performance factor. Dense configuration encoding can be exploited to minimize the external storage space required for configurations and the transmission time required to communicate configurations.

At a broader level, we have focussed on interconnect flexibility to establish gross bounds on the information required to configure a device. We demonstrated simple building blocks and metrics for gauging the flexibility of an interconnect and apprising the level of redundancy in a particular interconnect description. These tools can be useful for first order analysis of programmable interconnect designs.

Acknowledgments

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] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020. *FLEX 8000 Handbook*, May 1994.

[2] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020. *Data Book*, March 1995.

[3] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts, 02061 USA, 1992.

[4] Richard Guo, Hung Nguyen, Adi Srinivasan, Quaid Nasir, Hong Cai, Steve Law, and Amar Mohsen. A Novel Reprogrammable Interconnect Architecture with Decoded RAM Storage. In *Proceedings of the IEEE 1994 Custom Integrated Circuits Conference*, pages 193–196. IEEE, May 1994.

[5] Chris Jones, John Oswald, Brian Schoner, and John Villasenor. Issues in Wireless Video Coding using Run-time-reconfigurable FPGAs. In Peter Athanas and Ken Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Los Alamitos, California, April 1995. IEEE Computer Society, IEEE Computer Society Press.

[6] Jonathan Rose and Stephen Brown. Flexibility of Interconnection Structures for Field-Programmable Gate Arrays. *IEEE Journal of Solid-State Circuits*, 26(3):277–282, March 1991.

[7] 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.

[8] Michael J. Wirthlin and Brad L. Hutchings. A Dynamic Instruction Set Computer. In Peter Athanas and Ken Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Los Alamitos, California, April 1995. IEEE Computer Society, IEEE Computer Society Press.

[9] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *The Programmable Logic Data Book*, 1994.

[10] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *XC5200 Logic Cell array Family Technical Data*, preliminary (v1.0) edition, April 1995.