High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systems*

Tsuyoshi Isshiki and Wayne Wei-Ming Dai

Applied Sciences Building,
Computer Engineering
University of California
Santa Cruz, CA 95064, USA
isshiki@cse.ucsc.edu, dai@cse.ucsc.edu

1 Introduction

Field-programmable hardware exhibits a new trend towards computation-intensive applications. The basic idea is to completely customize the hardware architecture for the very given application in order to allocation the logic resources efficiently and effectively, improving the performance several orders of magnitude greater than general-purpose processor implementation [1][3][5][6]. And at the same time, it still covers a wide variety of applications for their reconfigurability.

Of the challenges of developing this field-programmable hardware, the software support for programming the hardware has long been recognized as one of the most crucial field of study.

1. The programming of the field-programmable hardware should be easy enough for the application developers to handle. This requires the support for high-level design capture without the need for digital system design skills.

2. Together with the easy programming environment, the compiler has to be sophisticated enough to produce area-efficient, high-speed datapath circuits.

The development of *High-Level Synthesis* in ASIC-DSP community [9] cannot simply be applied to the datapath designs on FPGA. Since the routing resource and logic resource are physically fixed, it is often hard to control or even predict the outcome of the layout synthesis on FPGA. If the routing resource is saturated in a certain location, this may lead to underutilization of the logic resource, in which most case is more or less unavoidable. Or when the design has to be partition across multiple FPGA chips, limitation of the number of IO pins can lead to underutilization of the logic resource as well. Therefore, the amount of hardware needed to implement a certain circuit can be hard to predict. Delay prediction can also be very difficult especially when pass-transistors are used as interconnects of the routing wires. Unable to predict these important low-level information makes the tradeoff decisions in the high-level synthesis less effective. Because of these reasons, some engineers still prefer manually handling the circuit designs and layout, or at least a part of them [2][3]. We believe that these problems come from the large growing gap between the logic synthesis techniques and the logic implementation technology. Although logic implementation technology have diverged into full-custom, standard cells, gate arrays and FPGA implementations, logic synthesis techniques have not been evolving accordingly. Routing area occupies significant portion of the chip area in VLSI designs in general, and silicon resource is thus underutilized. The situation becomes worse as the logic implementation becomes more robust from standard cells towards FPGAs. Layout synthesis studies have long been subjected to this routing-hungry logic circuits. Our approach towards this problem is to narrow the gap of logic synthesis and logic implementation, by designing logic circuits finely tuned for the specific logic implementation, in which our case is the FPGA. In this paper, we demonstrate that by using bit-serial circuits, we are no longer limited by the available routing resource or the IO resource of the FPGA. And this opens up a totally new opportunity for performance-driven partitioning and placement algorithms for FPGA layout, where conventionally we were subjected to maximum routable placement and minimum-cut partition. We then have more control of the outcome of layout synthesis, being able to predict the performance and the required amount of hardware, and thus making wiser trade off decisions on higher level of abstraction. The first part of this paper describes this bit-serial circuit designs and demonstrates the effectiveness of the bit-serial architecture on FPGA.

The second part of the paper deals with the actual programming environment developed upon C language for easy design verification and completely automated circuit synthesis using the bit-serial datapath modules

*This work is supported in part by ARPA under ONR Grant N00014-93-1-1334.

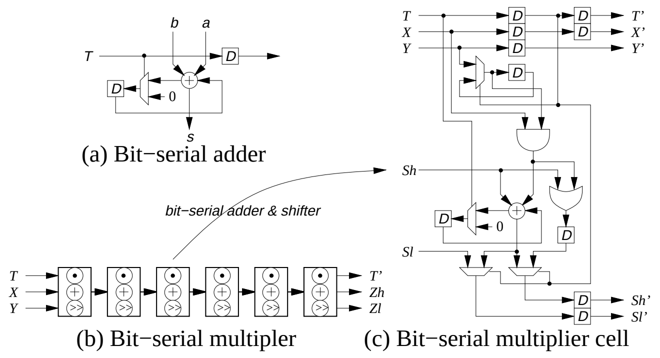


Figure 1: Bit-serial multiplier module cells

library.

2 Bit-Serial Datapath Circuit Designs on FPGA

2.1 Bit-Serial Arithmetic Operator Designs

Bit-serial arithmetic operators have a very simple and area efficient structure [7][8]. They operate at each bit, either in least-significant-bit-first or most-significant-bit-first order, and generated carry is fed back to itself. Fig.1 shows the design of bit-serial adder and 2's complement multiplier. Bit-serial multiplier can produce a $2N$-bit double precision product every N clock cycles for N-bit inputs. Bit-serial double precision data are represented by two wires and can easily be rounded down to single precision by various rounding functions. Control logic which controls the feedback of the carry bits is basically a chain of shift registers. Each bit-serial data accompanies a *tail bit* whose value is 1 when the data bit is the tail of a data word, and 0 otherwise. Each bit-serial operator is responsible for generating the *tail bit* of the output data as well as the output data itself. Bit-serial operators can be connected in a systolic array fashion to implement a very fine grain pipeline network. This network of bit-serial operator cells can easily be partitioned across multiple FPGA chips because of their sparce interconnections, and still maintain the high throughput by simply inserting pipeline latches along the chip-to-chip connections.

Table 1 shows some figures of other bit-serial operators. We should note here that the logic depth of the bit-serial circuits is basically independent of the word length (1 to 2 LUTs) which leads to a very high frequency operation and also accurate performance prediction.

Table 1: Statistics of bit-serial datapath modules. Word length = N

Modules	Area	Logic depth
Multiplier		
(1-input 1-constant)	$N \sim 4N$ CLBs	1 LUT
(2-inputs) type I	$5N$ CLBs	2 LUTs
(2-inputs) type II	$5.5N$ CLBs	1 LUT
Adder		
(single precision)	1 CLB	1 LUT
(double precision)	3 CLBs	1 LUT
Rounder		
(truncate)	1 CLB	1 LUT
(round-to-nearest-even)	4 CLBs	1 LUT
Absolute operator	2 + N/2 CLBs	1 LUT
Max-min selector		
(least-significant-bit-first)	1 + N/2 CLBs	1 LUT
(most-significant-bit-first)	4 CLBs	1 LUT

2.2 Comparison of Bit-Serial Modules Against Bit-Parallel Modules

Let us now discuss how these bit-serial circuits compare with bit-parallel circuits in terms of area and performance under the Xilinx FPGA architecture. Comparison is done on three parameters: T = time, A = area, and $A \cdot T$ = area × time. T refers to the data sampling period or the inverse of the throughput (ns). Area is measured by the number of logic blocks (CLBs). $A \cdot T$ describes the efficiency of the circuits. Following delay parameters are used to estimate the performance:

T_{LUT} :	Combinational delay	(4.5 ns)
T_{FF} :	Flip-flop delay (clock to output)	(3.0 ns)
T_{carry} :	Bypass carry logic delay	(0.75 ns)
T_{sum} :	Carry-chain overhead delay (Operands to outputs)	(10.0 ns)

Actual numbers are given according to the parameters of Xilinx XC4000-5 series FPGA chip [10].

2.2.1 Adders

- Bit-parallel ripple-carry adder using bypass carry generator:

$$\begin{aligned} T &= N \cdot T_{carry} + T_{sum} + T_{FF} \\ A &= \lceil \tfrac{N}{2} \rceil + 1 \end{aligned}$$

- Bit-parallel ripple-carry adder without bypass carry generator:

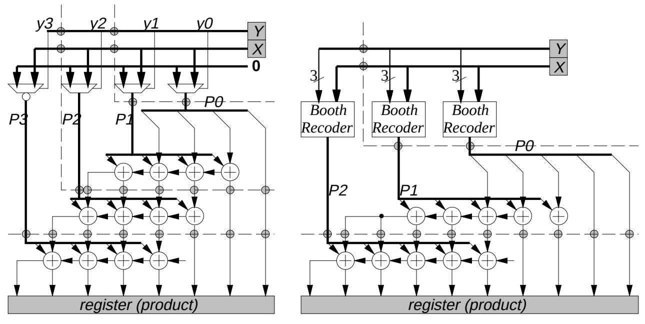
$$\begin{aligned} T &= N \cdot T_{LUT} + T_{FF} \\ A &= N \end{aligned}$$

- Bit-serial adder:

$$\begin{aligned} T &= (T_{LUT} + T_{FF}) \cdot N \\ A &= 1.5 \end{aligned}$$

Table 2: Comparison of ripple-carry adder and bit-serial adder on thoughput period and area. Data word length N is assumed to be 16 bits.

Type	T (time)	A (area)	$A \cdot T$
Ripple-carry adder (with carry logic)	25.0 ns	9 CLBs	225
Ripple-carry adder (without carry logic)	75.0 ns	16 CLBs	1200
Bit-serial adder	120.0 ns	1.5 CLBs	180



(a) Ripple−carry adder based parallel multipler.

(b) Ripple−carry adder based Booth's parallel multipler.

Figure 2: Parallel multipliers. In order to increase the thoughput, we can insert pipeline latches along the nets crossing the dotted line.

Table 2 shows the actual numbers based on the above parameter values. For $N = 16$, we can see that bit-serial adder is 1.25 times more efficient than bit-parallel ripple-carry adder with bypass carry generators, and 6.67 times more efficient than ripple-carry adder without the bypass carry generator. These gaps grow larger as N increases.

2.2.2 Multipliers

Here, we will compare three types of bit-parallel multiplier against two types of our bit-serial multiplier. Fig.2(a) shows a conventional parallel multiplier composed of ripple-carry adders, and Fig.2(b) shows the Booth's parallel multiplier. Although carry-save adders are more popular than ripple-carry adders for parallel multipliers in VLSI implemetation, parallel multipliers using ripple-carry adders are same in both area and performance on FPGAs utilizing fast carry logic. These parallel multipliers are limited in their performance since the critical paths include long paths which do not go through the bypass carry generators. In order to increase the throughput, we can divide these paths by inserting pipeline latches. One drawback of this approach is that the overhead circuit of pipeline retiming is very large.

Fig.3 shows another design style where the area is

(

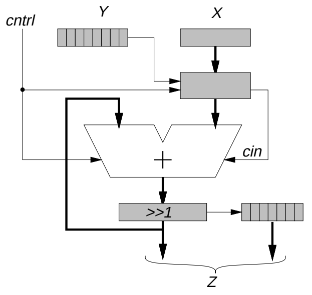


Figure 3: Sequential shift & add multiplier.

significantly reduced at the cost of performance. This shift & add multiplier can perform N-bit 2's complement multiplication every N cycles.

- Parallel multiplier using bypass carry generator:

$$\begin{aligned} T &= (2N-2)\cdot T_{carry} + (N-1)T_{sum} \\ &+ T_{LUT} + T_{FF} \\ T_{pipe} &= N \cdot T_{carry} + T_{sum} + T_{FF} \\ A &= (2N-1)\cdot \lceil \tfrac{N}{2} \rceil + N - 1 \\ A_{pipe} &= A + \lceil \tfrac{N}{4}(3N-5) \rceil \end{aligned}$$

- Booth's parallel multiplier using bypass carry generator:

$$\begin{aligned} T &= (2N-1)\cdot T_{carry} + \lceil \tfrac{N}{2} \rceil \cdot T_{sum} \\ &+ 3T_{LUT} + T_{FF} \\ T_{pipe} &= (N+1)\cdot T_{carry} + T_{sum} + T_{FF} \\ A &= \lceil \tfrac{N}{2} \rceil \cdot (\lceil \tfrac{3N}{2} \rceil + 5) \\ A_{pipe} &= A + \lceil (\lceil \tfrac{N}{2} \rceil - 1)(\lceil \tfrac{N}{2} \rceil + N)/2 \rceil \end{aligned}$$

- Shift & add multiplier using bypass carry generator:

$$\begin{aligned} T &= N(N \cdot T_{carry} + T_{sum} + T_{FF}) \\ A &= 3\lceil \tfrac{N}{2} \rceil + N + 1 \end{aligned}$$

- Bit-serial multiplier using bit-serial adders and shifters (type I and type II):

$$\begin{aligned} T_I &= N(2T_{LUT} + T_{FF}) \\ A_I &= 5N \\ T_{II} &= N(T_{LUT} + T_{FF}) \\ A_{II} &= 5.5N \end{aligned}$$

Table 3 shows the actual numbers for the various types of multipliers discussed above. As we can see, bit-serial multipliers are in fact faster than many of the bit-parallel multipliers. Pipelined parallel multipliers which can outperform the bit-serial multiplier (II) by a factor of 5 are unlikely to be a design option in the actual design because of their large circuits requiring 300 to 400 CLBs. We should add that while shift & add

Table 3: Comparison of parallel multipliers, shift & add multipliers and bit-serial multipliers on thoughput period and area. Data word length N is assumed to be 16 bits.

Type	T (time)	A (area)	$A \cdot T$
(with carry logic)			
Parallel	180.0 ns	263 CLBs	47340
Parallel (pipe)	25.0 ns	435 CLBs	10875
Booth's parallel	119.75 ns	232 CLBs	27782
Booth's parallel (pipe)	25.75 ns	316 CLBs	8137
Shift & add	400.0 ns	41 CLBs	16400
(without carry logic)			
Parallel	210.0 ns	368 CLBs	77280
Parallel (pipe)	75.0 ns	540 CLBs	40500
Booth's parallel	187.5 ns	288 CLBs	54000
Booth's parallel (pipe)	79.5 ns	372 CLBs	29574
Shift & add	1200.0 ns	49 CLBs	58800
Bit-serial (I)	192.0 ns	80 CLBs	15360
Bit-serial (II)	120.0 ns	88 CLBs	10560

multiplier is smaller than our bit-serial multipliers, its operations are bit-serial in nature and therefore can be changed into bit-serial multiplier simply by modifying the input and output registers into shift registers.

These observations show that bit-serial operators are more efficient in terms of area $\times$ time measure for 16-bit word data. Area-time complexity of bit-parallel adders and multipliers are $O(N^2)$ and $O(N^3)$, respectively, whereas bit-serial adder and multipliers are $O(N)$ and $O(N^2)$, respectively, which means that as the word size grows, bit-serial operators become more and more efficient compared to bit-parallel operators. It is important to note that these observations are based upon the existing FPGA architecture utilizing fast carry logic which is biased toward bit-parallel operations. Bit-serial operators require large number of storage elements, and large portion of the logic resource is used only for latching data. If we were to strip away the fast carry logic and put more flip-flops inside the logic blocks instead, this would further favor the bit-serial operators, requiring even fewer number of logic blocks.

2.3 Impact on the Circuit Layout on FPGA Architecture

The advantage of bit-serial circuits on the physical layout on FPGA architecture is just as appealing. The unique features of the FPGA architectures (as for our interest, Xilinx FPGAs in particular) makes the layout problem very different from ASIC designs:

- Limited routing resource
- Limited IO resources
- Large routing delays

Table 4: Layout result of 5×5 2D FIR bit-serial filter. There are 25 multipliers and 25 adders. Word length is 8 bits (external) and 16 bits (internal). Circuit includes 5 parallel-to-serial converters and 1 serial-to-parallel converters for communicating with the outside. Delay estimation assumes XC3042-100 parts.

	# of CLBs	Logic utilization	# of IOBs	Critical path delay		
				Pad to setup	Clock to pad	Clock to setup
1	137	95.1%	44	17.0ns	21.2ns	38.8ns
2	119	82.6%	10	17.0ns	19.0ns	26.4ns
3	123	85.4%	7	17.0ns	11.0ns	24.8ns
4	136	94.4%	22	17.0ns	20.4ns	24.2ns
5	113	78.5%	7	17.0ns	11.0ns	26.5ns
6	114	79.2%	22	17.0ns	19.0ns	23.0ns

Under these circumstances, bit-serial circuits are far more suited for mapping on FPGAs compared to bit-parallel circuits.

1. Partition:
   (a) IO pin limitation is a major problem in bit-parallel datapath circuits. Also the large size of the module cluster can leave a lot of unused spaces and make the logic resources underutilized.
   (b) Bit-serial datapath modules are easy to partition since cell-to-cell connections are sparse and would not lead to IO pin limitation problem. Chip-to-chip communication penalty can be totally eliminated by simply adding pipeline latches on the partitioned inter-cell connections. Logic resources can be utilized to their maximum limits for the small size of the bit-serial operator cells.
2. Routing:
   (a) Wiring distance can be considerably long for some large fanout nets. Routability of such modules are hard to predict, and their routing delays are also unpredictable.
   (b) Bit-serial modules consisting of systolic array cells only has local connections. Since the distance of those wires are all short, the propagation delays of those wires can be highly predictable. Also because of the local connections, routing wires tend to be evenly distributed throughout the chip, naturally avoiding routing congestions.

To support the above argument, we have provided with some examples to demonstrate the high logic resource utilization of the bit-serial datapath modules

Table 5: Layout result of 8-point Inverse DCT bit-serial circuit. There are 20 multipliers and 28 adders. Word length is 16 bits (external) and 32 bits (internal). Circuit includes 8 parallel-to-serial converters and 8 serial-to-parallel converters. Delays estimation assumes XC3042-100 parts.

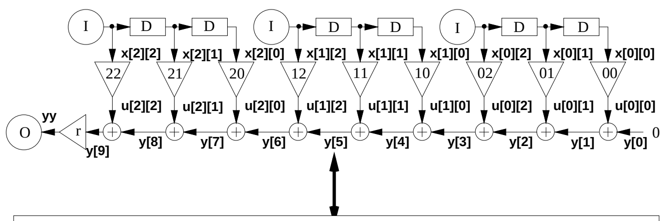
	# of CLBs	Logic utilization	# of IOBs	Critical path delay		
				Pad to setup	Clock to pad	Clock to setup
1	133	92.4%	39	17.0ns	12.4ns	37.5ns
2	138	95.8%	16	17.0ns	14.2ns	23.7ns
3	143	99.3%	36	17.0ns	24.6ns	26.6ns
4	130	90.3%	30	17.0ns	13.1ns	24.0ns
5	143	99.3%	37	17.0ns	13.1ns	26.4ns
6	121	84.0%	17	17.0ns	12.4ns	22.8ns
7	111	77.1%	19	17.0ns	18.1ns	22.5ns
8	80	55.6%	8	17.0ns	11.1ns	24.9ns
9	72	50.0%	8	17.0ns	14.2ns	27.6ns
10	136	94.4%	23	17.0ns	16.3ns	23.7ns

and their consistent propagation delay figures. Table 4 shows the layout result of a 5 × 5 2D FIR filter implemented by bit-serial circuits. 5 line-delays are implemented outside the FPGA implementation and the datapath contains 5 parallel-to-serial converters and 1 serial-to-parallel converters to communicate with the outside world which are implemented on FPGA as well. The external word length is 8 bits and internal word length is 16 bits using double precision adders and a rounder to convert to 8 bits. Table 5 shows the layout result of an 8-point Inverse Discrete Cosine Transform circuit with 8 parallel-to-serial input ports and 8 serial-to-parallel output ports. External and internal word lengths are 16 bits and 32 bits, respectively. Under a 20 MHz clock, the performance is 62.5 MOPS for 5 × 5 2D FIR filter and 60.0 MOPS for 8-point IDCT. With newer FPGA devices such as XC4000 and XC3100 series, clock is expected to run as fast as 50 MHz, where the performance would reach 156 MOPS for 5 × 5 2D FIR filter and 150 MOPS for 8-point IDCT.

As shown in the tables, the circuits are *always* 100% routed even with the very high logic utilization of over 99%. Also, the critical path delays are constantly in the same range for both algorithms. Usage of the IO pins are low, meaning that the IO pin limitation is no longer critical in the partitioning problem.

3 High-Level Datapath Synthesis in C-code

Let us now describe how these design examples are generated. We are currently developing a high-level datapath synthesis tool for multi-FPGA designs on C language using the bit-serial datapath module library. The



```
for(k = 0; k < tapY; k ++){
   for(l = 0; l < tapX - 1; l ++){
      FPGA_delay(x[k][l],x[k][l+1],WORD);
   }
   FPGA_write(x[k][tapX-1],in_frame[i][j],WORD);
}
FPGA_set(y[0],0);
for(k = 0; k < tapY; k ++){
   for(l = 0; l < tapX; l ++){
      FPGA_mult_c(u[k][l],x[k][l],coef[tapY-k-1][tapX-l-1],WORD);
      FPGA_dadd(y[tapY*k+l+1],y[tapY*k+l],u[k][l],WORD);
   }
}
FPGA_round_even(yy,y[tapX*tapY],WORD);
FPGA_read(&out_frame[i][j],yy,WORD);
```

Figure 4: C description for 2D FIR filter. Function calls `FPGA_xxx()` denotes the hardware operations. Arguments of the function calls are the input and output variables of the operation and the word length of the data.

main features of the tool are:

1. High-level design capture in C.
2. Algorithm verification at behavioral level.
3. Fully automated circuit and layout synthesis. Functional and logic simulations are unnecessary due to this automated processes which assure correct functionality and timing constraints.
4. Bit-serial datapath module library written as C functions.

3.1 High-Level Design Capture in C

Fig.4 shows the C-code for 2D FIR filter. Programmer would describe their algorithm in a very similar way as writing a normal C-code. The only difference is that the programmer would explicitly specify which operations are to be performed on the field-programmable hardware. *Hardware operations* are specified by function calls `FPGA_xxx()`. Variables are also declared as either *software variables* or *hardware variables*. Communication between these two types of variables are explicitly performed by `FPGA_write()` and `FPGA_read()` function calls.

3.2 Behavioral Simulation

This hardware C-code can simply be a part of any C program. Each hardware operation function includes the behavioral model of the operation. Thus behavioral

```
read_image(image, in_frame);
read_filter_coefficient(coef_file, coef);

for(i = 0; i < sizeY + originY; i ++)
{
    for(j = 0; j < sizeX + originX; j ++)
    {
            Hardware C-code
            for 2D FIR filter
    }
}
display_image(out_image);
```

Figure 5: Hardware C-code in behavioral simulator

simulator can be written just by adding the hardware C-code in the routine (Fig.5), and is, of course, significantly faster than logic simulation. Design verification on lower levels are not needed due to fully automated circuit synthesis.

3.3 Bit-serial datapath module library

Bit-serial datapath module library is composed of groups of C functions. Each datapath module is associated with a set of C functions which establish the following hierarchical netlist data structures:

1. Module netlist:
   This can also be seen as the data-flow graph describing the algorithm. This netlist is a *logical* description of the given algorithm which consists of *modules, ports* and *variables.* This netlist is generated when hardware operation functions **FPGA_xxx()** are invoked. Ports and variables are associated with scheduling information such as *latency* of output ports of each module, *latest* and *earliest* time of each variable. This netlist is used for various high-level synthesis processes such as *pipeline synchronization* and *architecture synthesis.*

2. Cell netlist:
   This is the *physical* description of the given algorithm which consists of *cells, pins* and *nets.* Partition algorithm works on this netlist, and if needed, dedicated placement algorithm can also be developed on it. A cell is composed of the following primitives:

   (a) CLB: *name, BASE, CONFIG, EQUATE, pin.*

   (b) IOB: *name, CONFIG, pin.*

   (c) COMPONENT: *name, type* (TBUF, PULLUP for longlines, and BUF for global clock buffers)*, pin.*

Important characteristic of the bit-serial datapath cells is that the cell circuits are complete sequential logic circuits where all the output signals are latched. All the critical paths unaccounting the routing delays are known in advance, and therefore the maximum operational clock frequency can be associated with each cell. This greatly simplifies the early assessment of system performance during high-level synthesis.

3.4 Circuit Synthesis

1. Architecture Synthesis:

   Architecture synthesis routine, a translation of a logical description into a physical one, generates the cell netlist from the module netlist. As of now, a trivial algorithm is used for architecture synthesis. All logically allocated module instances are given distinct physical allocations. That is, all hardware operations are executed on seperate hardwares. Logical variables have a distinct physical allocation of nets. More sophisticated architecture synthesis routines involving scheduling, loop unfolding, resource assignment, register allocation and memory allocation will be implemented in the near future.

2. Flip-Flop Minimization:
   Some of the bit-serial datapath modules include large amount of shift registers. Data sampling delays are key component of convolution circuits such as FIR filter, and they are implemented only by shift registers. Since bit-serial multiplier includes these shift registers, they can be shared. After the initial architecture synthesis routine, register elements are shared whenever possible.

3. Pipeline Synchronization:
   For datapath modules with multiple inputs, all inputs have to be completely synchronized. That is, the arrival time of all the tail bits have to be the same. Since each module may have different latency, we have to retime the inputs in order to make the modules function properly. Register elements are inserted or deleted to achieve this synchronization.

4. IO Port Scheduling:
   IO ports sharing the same parallel bus to the outside world has to be statically scheduled in order to avoid bus conflicts. Retiming registers are inserted according to the scheduling.

3.5 Layout Synthesis

1. Partition:
   Currently implemented partitioning algorithm is

based on a classical min-cut hyper-graph bipartitioning heuristics by Fiduccia and Mattheyses [4] for its simplicity. The netlist is recursively partitioned into clusters of a given size. Although this simple algorithm achieves satisfactory results in terms of number of IO pins used as see in Table 4 and Table 5, we believe that carefully tuning this algorithm for our bit-serial datapath circuits would further improve the logic utilization. Since IO pin limitation is no longer a problem, it is now practical for us to search for a performance driven, maximum logic utilizing partitioning algorithm.

After the partition is completed, the tool will generate seperate mapped netlist files (`.xtf` files) with only CLB, IOB, TBUF, PULLUP and GCLK primitives.

2. Placement and Routing:
XNFMAP from Xilinx is performed to each `.xtf` files to perform design rule checking and elimination of loadless CLBs and IOBs. Placement and routing is performed by the APR also from Xilinx.

We believe that the placement algorithm used in APR is not well-suited for our bit-serial circuits where most of the nets are low fan-in low fan-out local connections. From the layout results of our bit-serial circuits where the designs were always completely routed *all* at the first try with logic utilization as high as 99%, we strongly believe that we have succeeded in designing a group of circuits which use substantially small amount of routing, and therefore a *performance driven placement* instead of a *maximum routability placement* is now a reality for FPGA layout.

4 Summary

In this paper, we first described our bit-serial datapath circuit designs and their advantages over bit-parallel circuits on FPGA implementation. We have given some circuit layout examples of 2D FIR filter and IDCT circuits which demonstrates the efficiency of bit-serial circuits in terms of logic resource utilization, routability, IO pin utilization and performance. Also, we have described our high-level datapath synthesis tool under development. It accepts high-level C-code description of the desired algorithm and generates the fully mapped and partitioned netlists to be fed to the APR tool. We have demonstrated that this high-level synthesis approach is now practical in generating highly efficient designs on multi-FPGA system using the bit-serial datapath circuits.

References

[1] P. Bertin, D. Roncin and J. Vuillemin, "Programmable Active Memories - Performance Measurements," *Proc. ACM/SIGDA Workshop on Field Programmable Gate Arrays*, 1992.

[2] P. Bertin and H. Touati, "PAM Programming Environments: Practice and Experience," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.

[3] C. E. Cox and W. E. Blanz, "GANGLION–A Fast Field-Programmble Gate Array Implementation of a Connectionist Classifier," *IEEE Solid-State Circuits* Vol. 27, No. 3, pp. 288–299, March 1992.

[4] C. M. Fiduccia and R. M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. 19th Design Automation Conference*, pp.241-247

[5] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich and D. Sweely, "Building and Using a Highly Parallel Programmable Logic Array," *IEEE Computers*, pp. 81-89, Jan. 1991.

[6] Dzung T. Hoang, "Searching Genetic Databases on Splash 2," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 185-191, April 1993.

[7] T. Isshiki and W. W.-M. Dai, "High-Performance Datapath Implementation on Field-Programmable Multi-Chip Module (FPMCM)," *Proc. 4th International Workshop on Field-Programmable Logic and Applications, FPL '94*, Sept. 1994.

[8] T. Isshiki and W. W.-M. Dai, "Field-Programmable Multi-Chip Module (FPMCM) for High-Performance DSP Accelerator," *IEEE Asia-Pacific Conference on Circuits and Systems*, Dec. 1994.

[9] J. Vanhoof, K. V. Rompaey, I. Bolsens, G. Goossens and H. De Man," "High-Level Synthesis for Real-Time Digital Signal Processing," *Kluwer Academic Publishers*, 1993.

[10] "The Programmable Logic Data Book," *Xilinx, Inc.*, 1994.