[4] M. Diaz, P. Azema, and J. M. Ayache, "Unified design of self-checking and fail-safe combinational circuits and sequential machines," *IEEE Trans. Comput.*, vol. C-28, pp. 276–281, Mar. 1979.
[5] T. Nanya and M. Uchida, "The design of strongly fault-secure and strongly code-disjoint combinational circuits," in *Proc. Joint Fault-Tolerant Computing Symp.*, 1989, pp. 245–250.
[6] M. Nicolaidis, "Shorts in self-checking circuits," *J. Electron. Testing: Theory Appl.*, vol. 1, pp. 257–273, 1991.
[7] D. Nikolos, "Theory and design of t-error correcting/d-error detecting $(d > t)$ and all unidirectional error detecting codes," *IEEE Trans. Comput.*, vol. 40, pp. 132–142, Feb. 1991.
[8] T. Nanya and T. Kawamura, "On error indication for totally self-checking systems," *IEEE Trans. Comput.*, vol. C-36, pp. 1389–1391, Nov. 1987.
[9] R. A. Parekhji, G. Venkatesh, and S. D. Sherlekar, "A methodology for designing optimal self-checking sequential circuits," in *Proc. IEEE Int. Test Conf.*, 1991, pp. 283–291.
[10] I. Jansch and B. Courtois, "Design of SCD checkers based on analytical hypotheses," in *Proc. 10th Eur. Solid-State Circuits Conf.*, 1984, pp. 109–112.
[11] J. Shen, W. Maly, and F. Ferguson, "Inductive fault analysis of MOS integrated circuits," *IEEE Design Test Comput.*, pp. 26–33, Dec. 1985.
[12] A. Casimiro, M. Simoes, M. Santos, I. Teixeira, and J. P. Teixeira, "Experiments on bridging fault analysis and layout-level DFT for CMOS designs," in *Proc. IEEE Int. Workshop Defect and Fault Tolerance in VLSI Syst.*, 1993, pp. 109–116.
[13] J. D. Lesser and J. J. Schedletsky, "An experimental delay test generator for LSI logic," in *Proc. IEEE Int. Test Conf.*, 1980, pp. 235–248.
[14] G. L. Smith, "Model for delay faults based upon path," in *Proc. IEEE Int. Test Conf.*, 1985, pp. 342–349.
[15] H. Hao and E. J. McCluskey, "Resistive shorts within CMOS gates," in *Proc. IEEE Int. Test Conf.*, 1991, pp. 292–301.
[16] B. Bose, "On unordered codes," *IEEE Trans. Comput.*, vol. 40, pp. 125–131, Feb. 1991.
[17] J. F. Wakerly, "Detection of unidirectional multiple errors using low-cost arithmetic codes," *IEEE Trans. Comput.*, vol. C-24, pp. 210–212, Feb. 1975.
[18] C. Lin and S. M. Reddy, "On delay fault testing in logic circuits," in *Proc. IEEE Int. Test Conf.*, 1986, pp. 148–151.
[19] G. Smith, "Model for delay faults based upon paths," in *Proc. IEEE Int. Test Conf.*, 1985, pp. 845–856.
[20] R. Lisanke, "Logic synthesis and optimization benchmarks user guide v. 2.0," Tech. Rep., Microelectron. Cen. North Carolina.
[21] E. M. Sentovich *et al.*, "SIS: A system for sequential circuit synthesis," Tech. Rep., Univ. California, Berkeley.
[22] N. K. Jha and S.-J. Wang, "Design and synthesis of self-checking VLSI circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 878–887, June 1993.
[23] A. Greiner and F. Pecheux, "Alliance: A complete set of CAD tools for teaching VLSI design," Tech. Rep., Laboratorie MASI/CAO-VLSI, Univ. Pierre et Marie Curie, Paris, France.

A Phase Assignment Method for Virtual-Wire-Based Hardware Emulation

Hsiao-Pin Su and Youn-Long Lin

Abstract— **In a hardware emulator consisting of multiple field-programmable gate arrays (FPGA's), the utilization of the FPGA logic resource is usually very low due to the limitation on the number of I/O pins. *Virtual wire* technology not only increases the inter-FPGA communication capability, but it also increases the logic resource utilization by means of time division multiplexing (TDM). TDM allows one physical wire to be shared by multiple logical wires. For TDM to be effective, each transportation of an inter-FPGA signal must be carefully assigned to a slot of the time division. In this note, we show that the phase assignment problem is exactly same as the *resource-constrained operation scheduling* problem. We adopt the static-list scheduling heuristic for the task, and present some experimental results on a set of benchmark circuits from the MCNC. The experiments show that the proposed method can increase the number of effective I/O pins as many as ten times.**

I. Introduction

Design verification is essential to virtually any very large scale integration (VLSI) design project. One of the popular verification methods is logic simulation. Logic simulation software reports on how the circuit under design will respond to a sequence of input vectors, so the designer can judge whether the circuit behaves as expected over the input sequence. The more vectors simulated, the greater the designer has confidence in the correctness of the designing circuit. As the circuit complexity increases and the time to market shortens, inadequate simulation speed becomes a major bottleneck in the design process. Therefore, several special-purpose machines [1], [3], [6] have been built to accelerate the simulation task. Despite their having achieved a speed-up of one–two orders of magnitude, simulation accelerators still cannot fully meet the need of today's designs. Presently, the fastest simulation-based verification is done with a hardware emulator such as QuickTurn [14].

A hardware emulator consists of a large number of field-programmable gate arrays (FPGA's) interconnected either directly or indirectly through field-programmable interconnect chips (FPIC's) [18]. A hardware emulator can be configured as the circuit under design. Even running at a few hundred kilohertz, a hardware emulator still evaluates input vectors much faster than a simulation software does (as much as 10^5 times faster). Therefore, FPGA-based hardware emulation is becoming indispensable in many state-of-the-art VLSI design projects [4], [8].

Given a circuit netlist, many tasks need to be done before the emulator evaluates the input vectors. First, the netlist must be partitioned into smaller parts if it is too large to fit into a single FPGA, which is usually the case. Second, each smaller part is then technologically mapped into a set of configurable logic blocks (CLB's).[1] A CLB is the basic logic unit of an FPGA. Finally, parts are assigned to FPGA's, inter-FPGA routing is performed according to

Manuscript received June 7, 1995; revised July 12, 1996 and July 17, 1997. This paper was recommended by Associate Editor C.-K. Cheng. This work was supported in part by the National Science Council of R.O.C. under Contracts NSC-86-2221-E-007-049.

The authors are with the Department of Computer Science, Tsing Hua University, Hsin-Chu, Taiwan 30043 R.O.C.

Publisher Item Identifier S 0278-0070(97)07657-4.

[1] Other approaches may do mapping before partitioning.

the connectivity among parts, and placement and routing of CLB's are done within each FPGA.

One problem with multiple-FPGA-based emulation is the mismatch between the large logic density and the limited number of I/O pins of an FPGA chip. Typically, an FPGA chip has a logic capacity of more than 10 000 gates, but it has only 100–200 signal pins. According to Rent's rule [9] and some empirical measurement we have performed, when partitioning a large netlist under the pin constraints that each partition can use no more than 200 external connections, a partition can, on average, hold only 1000 gates. Therefore, the logic resources within the FPGA's are extremely underutilized. This problem will worsen in the future as semiconductor fabrication technology outpaces packaging technology in its growth.

Although the FPGA's can be clocked very fast (50–100 MHz), a hardware emulator usually runs very slowly (a few megahertz) because, during every emulation cycle, signals must travel through combinational paths across multiple FPGA's. *Virtual wire* technology [2], [5], [11] takes advantage of the high FPGA speed, with respect to the low emulation speed. By means of time division multiplexing (TDM), it uses each physical inter-FPGA wire to transport several logical signals during each emulation cycle.

In a virtual-wire-based emulator, an emulation cycle is divided into phases. Each phase is capable of transporting multiple logic signals across every inter-FPGA physical wire. Each logic I/O signal is assigned to one of the phases such that, within an emulation cycle, all signals can correctly travel through all combinational paths of the circuit. Effective assignment allows each physical wire to carry more logic wires, resulting in greater communication capability of the FPGA's, and hence, greater utilization of the logic resources.

In this note, we show that the phase assignment problem is exactly equivalent to the *resource-constrained operation scheduling* problem in high-level synthesis (HLS). Because the problem has been extensively studied in the HLS community, we propose adopting the static-list scheduling method [10] for the phase assignment problem.

The rest of the note is organized as followed. In Section II, we describe a typical hardware emulator architecture. The concept of virtual wire is described in Section III. Section IV defines the phase assignment problem, and proposes the assignment approach. Section V extends the method to a special case where each phase consists of only a single FPGA cycle. Section VI presents some experimental results. Concluding remarks are drawn and directions for future research are pointed out in Section VII.

II. Hardware Emulation

From the target system's point of view, the hardware emulator functions exactly as a slowed-down version of the chip under design. Two popular architectures for hardware emulation are mesh-connected [13] and partial-crossbar-connected [14] as depicted in Fig. 1. In this note, we concentrate on the mesh architecture. The proposed technique is also applicable to the other architecture.

To emulate the circuit under design, the software running on the host computer must do the following.

1) Partition the circuit into subcircuits, each satisfying both the logic capacity and the I/O pin constraints of an FPGA.
2) Assign each partitioned subcircuit to an FPGA, and complete inter-FPGA routing using inter-FPGA wires (i.e., board-level routing).
3) Technologically map gates into CLB's within each FPGA.
4) Perform intra-FPGA placement and routing of CLB's.

Besides configuring the emulator as the circuit under design, the host also inserts some interface circuitry for monitoring some of the interior of the circuit.

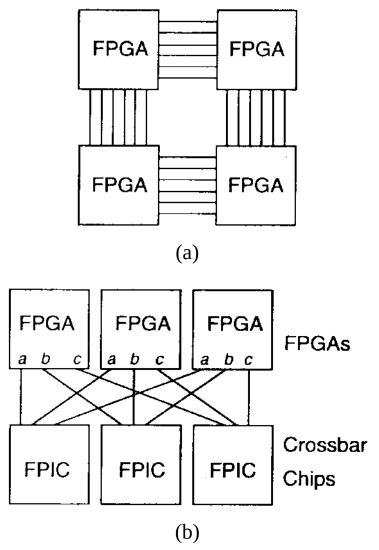


Fig. 1. Typical hardware emulator architectures: (a) mesh and (b) partial crossbar.

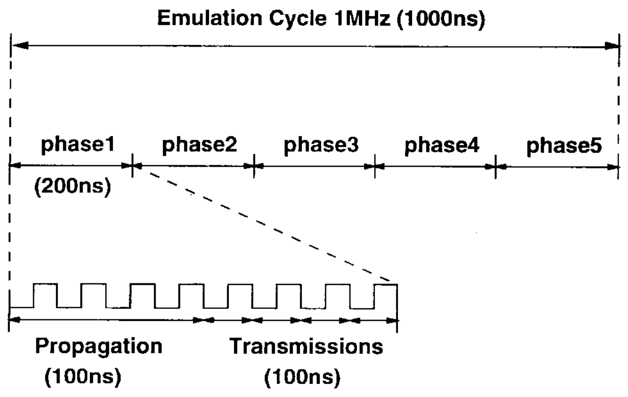


Fig. 2. Timing diagram for emulation cycle, phases, and FPGA cycle.

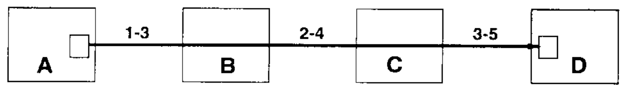


Fig. 3. A combinational path across four FPGA's.

The size of the circuit that an emulator is capable of emulating depends on the number of FPGA's of the system and the number of gates each FPGA can hold. The latter, in turn, depends on the quality of the partitioner as well as on the number of I/O pins with which each FPGA can communicate off chip.

State-of-the-art FPGA's can easily have a logic capacity of more than 10 000 gates. However, the number of I/O pins is restricted to a few hundred. For example, the largest chip from Xilinx (XC4025) [19] has 25 000 gates housed in a 299-pin package, of which only 256 are signal pins.

III. Virtual Wiring

Babb *et al.* [2], [5], [11] proposed using a TDM scheme to increase the effective inter-FPGA communication capability during hardware emulation. We use Fig. 2 to illustrate the operation principle of virtual wiring. Suppose the emulator is to run at 1 MHz, and the longest combinational path travels across six FPGA's. An emulation cycle consists of five phases. Assume that the FPGA's can be clocked at 40 MHz, and that the longest propagation of signals through any intra-

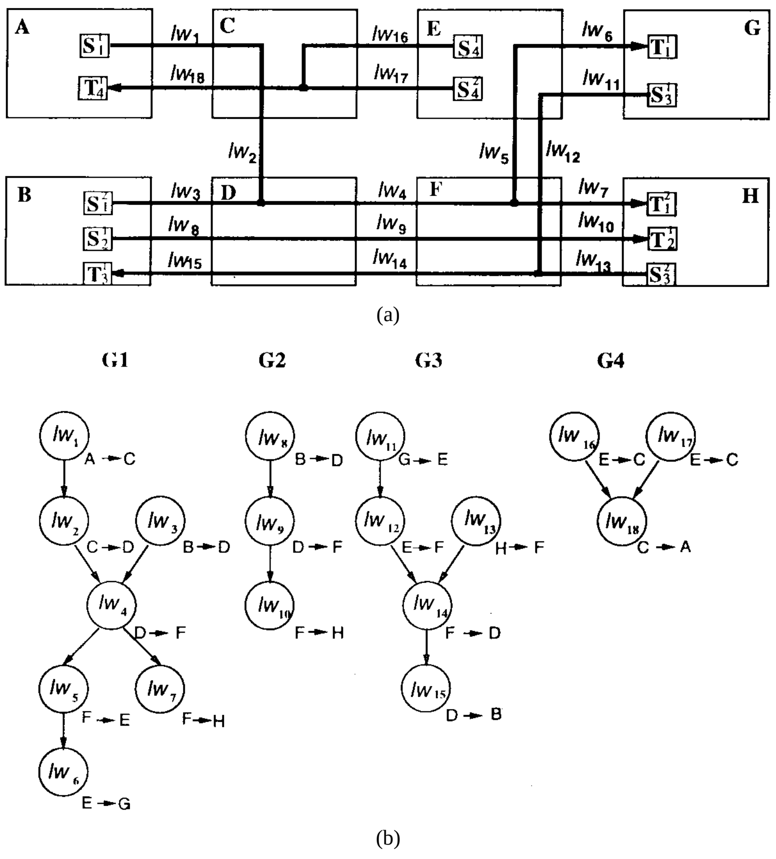


Fig. 4. (a) Combinational paths. (b) DFG's.

FPGA combinational path (from an input pin or a flip-flop output to a flip-flop input or an output pin) needs four FPGA cycles. So, we reserve the first four FPGA cycles in a phase for intra-FPGA propagation. Therefore, we have four FPGA cycles for transmitting four logical signals over a physical wire to a neighboring FPGA. In the best case, each physical wire can be used as 20 logical wires. Hence, we could have as much as a 20-fold increase in the effective communication capability.

The assignment of logic wires to phases cannot be done arbitrarily. Consider the case depicted in Fig. 3. There is a combinational path originating from a flip-flop in FPGA A, passing through FPGA's B and C, and terminating at a flip-flop in FPGA D. Let $T_{X \to Y}$ denote the transportation from FPGA X to FPGA Y. During each emulation cycle, the signal must be able to propagate through these three FPGA boundaries. Furthermore, $T_{A \to B}$ must occur at least one phase earlier than $T_{B \to C}$, which in turn must occur at least one phase earlier than $T_{C \to D}$. Clearly, there are dependency relationships among signal transportations. It is also clear that to carry out the three ordered transportations in five phases, $T_{A \to B}$ can only occur during phases one, two, or three. Similarly, the possible phases for other transportations are shown in Fig. 3.

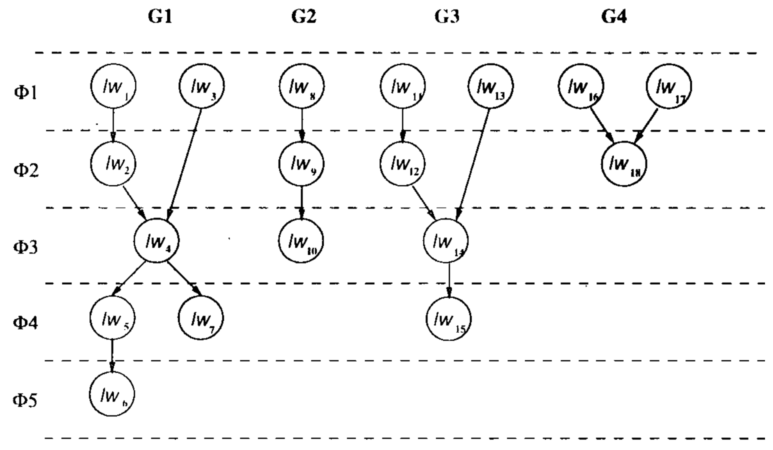
Because there is a limitation on the number of transportations over each physical wire during each phase, the number of physical wires needed from an FPGA X to its neighboring FPGA Y is $\lceil M/N \rceil$, where M is the maximum number of transportations from X to Y assigned to a phase, and N the maximum number of transportations per phase. In reality, the number of physical wires is fixed by the number of signal I/O's on each side of an FPGA chip. Hence, the phase assignment problem is to assign each logic wire to a phase, while preserving the dependency relationship, such that, under the I/O pin constraints for each path, the maximum number of phases needed is minimized so as to increase the emulation speed.

IV. Phase Assignment

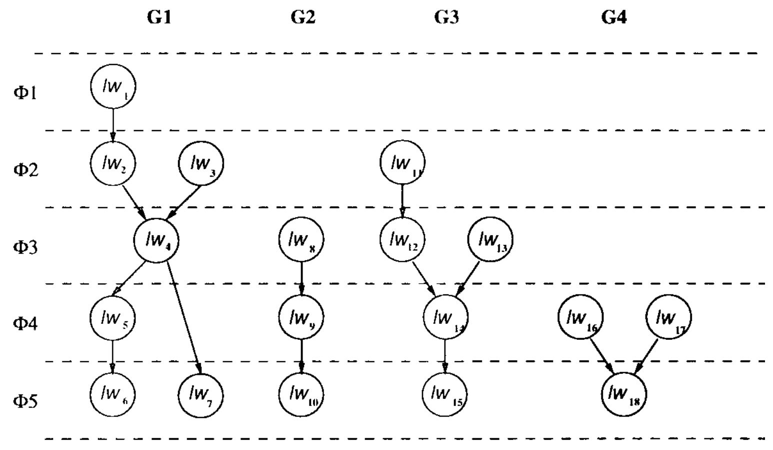
The phase assignment problem is equivalent to the *resource-constrained scheduling* (RCS) problem in high-level synthesis [7]. Inputs to the RCS problem are a data flow graph (DFG) and the number of resources available. The DFG is a directed acyclic graph (DAG) in which each vertex represents an operation and each edge a data dependency. To show that a phase assignment (PA) problem instance is exactly an RCS one, we view each logic wire transmission as an operation, each dependency relationship between a pair of transmissions as a data dependency between the corresponding pair of operations, each direction of each FPGA boundary as an operator type, and the number of physical wires for each direction per boundary as the resource constraints. Then we can build the DFG and resource constraints for an instance of the phase assignment problem.

Consider the illustrative example depicted in Fig. 4(a). The emulator consists of a 4 × 2 array of eight FPGA's labeled from A to H. There are four disjointed combinational (acyclic) paths in the circuit under design, and each could have one or more sources and one or more sinks. The dependency relationship of each combinational path is captured as a DFG, as depicted in Fig. 4(b). Consider graph $G1$. In the graph, each vertex represents a logic wire annotated with its boundary identity. For instance, notation $A \to C$ beside the vertex representing lw_1 means that lw_1 is from FPGA A to FPGA C. The directed edge from lw_1 to lw_2 denotes that lw_1 must be assigned at least one phase earlier than lw_2 is.

The essential difference between operation scheduling and phase assignment is their problem sizes. While almost all HLS benchmarks in the open literature consist of only a few tens of operations, the number of logic wires in an emulation instance is typically in the range of tens of thousands or more. However, we still can apply the same algorithm to solve the larger ones by properly managing the data structure.

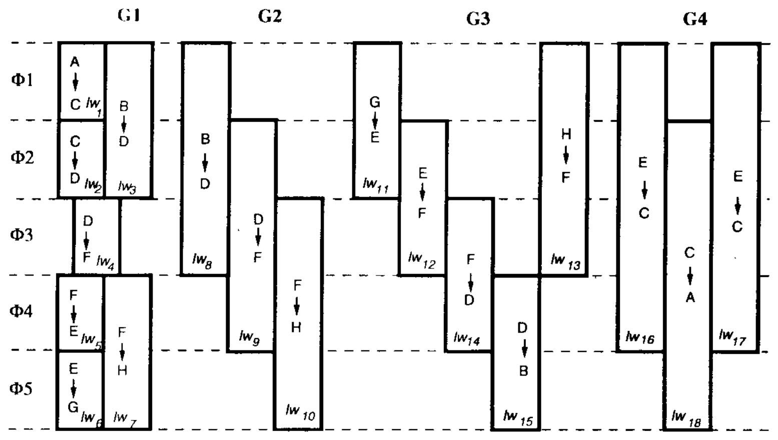


(a)



(b)

Fig. 5. (a) ASAP Scheduling. (b) ALAP scheduling. (c) The priority heap for phase ϕ_1.



(a)

Priority heap : lw_1 lw_3 lw_{11} lw_8 lw_{13} lw_{16} lw_{17}

(b)

Fig. 6. List scheduling: (a) the mobility of DFG; (b) the priority heap for phase ϕ_1.

In HLS research, many approaches have been proposed for the RCS problem. Two best known ones are the list-based scheduling method and the static-list scheduling method [10].

We present a version of the static-list scheduling heuristic. Our goal is to minimize the number of phases used under the inter-FPGA pin constraints. A logic wire is scheduled as early as possible, subject only to available physical wires and logic wire dependencies. The algorithm maintains a priority heap to determine the order in which logic wires are scheduled. The priority heap is decided by sorting all logic wires using the ALAP label (L_i) in ascending order as

```
List($G$)
{
    Asap($G$);
    Alap($G$);
    $Priority\_heap$ = Construct($G$);
    /* Construct $Priority\_heap$ for all unscheduled logic wire without predecessor*/
    $now\_cstep = 0$ ;
    while there exist unscheduled logic wire do
        Update all $Channel\_band\_width$;
        while $Priority\_heap \neq \phi$ do
            if $Channel\_band\_width_k$ for $lw_i$ is enough then
                Schedule $lw_i$ in $now\_cstep$;
                Update $Channel\_band\_width_k$;
            else
                Insert($Temp\_heap$, $lw_i$);
            endif
        endwhile
        $Priority\_heap = Temp\_heap$;
        $now\_cstep = now\_cstep + 1$;
    endwhile
    return;
}
```

Fig. 7. List scheduling algorithm.

```
Asap($G$)
{
    for each logic wire $lw_i \in G$ do
        if $Pred_{lw_i} = \phi$ then
            $E_i = 1$;
            $G = G - \{lw_i\}$;
        else
            $E_i = 0$;
        endif
    endfor
    while $G \neq \phi$ do
        for each logic wire $lw_i \in G$ do
            if all predecessor already scheduled then
                $E_i = \mathrm{Max}(Pred_{lw_i}, E) + 1$;
                $G = G - \{lw_i\}$;
            endif
        endfor
    endwhile
    return;
}
```

Fig. 8. **Asap** scheduling algorithm.

```
Alap($G$)
{
    for each logic wire $lw_i \in G$ do
        if $Succ_{lw_i} = \phi$ then
            $L_i = T$; /* given $T$ phases */
            $G = G - \{lw_i\}$;
        else
            $L_i = 0$;
        endif
    endfor
    while $G \neq \phi$ do
        for each logic wire $lw_i \in G$ do
            if all successors already scheduled then
                $L_i = \mathrm{Min}(Succ_{lw_i}, L) - 1$;
                $G = G - \{lw_i\}$;
            endif
        endfor
    endwhile
    return;
}
```

Fig. 9. **Alap** scheduling algorithm.

the primary key and the ASAP label (E_i) in descending order as the secondary key. If both keys have the same value, an arbitrary ordering is used.

Fig. 5(a) and (b) depicts the ASAP and ALAP scheduling, respectively, for the DFG's of the illustrative example. With the ASAP (E_i) and ALAP (L_i) values, Fig. 6(a) shows the mobility of each logic wire in Fig. 5. Because logic wires $lw_1, lw_3, lw_8, lw_{11}, lw_{13}, lw_{16}$, and lw_{17} do not have any predecessor, the algorithm puts these logic wires in the priority heap for phase ϕ_1. Among these seven nodes, logic wire lw_1 has the lowest ALAP value, so it has the highest priority and should be scheduled first. The rest of the heap is formed in a similar manner. The priority heap for phase ϕ_1 is shown in Fig. 6(b). Logic wires lw_{16} and lw_{17} have the same ASAP and ALAP values so we arbitrary choose lw_{16} to be scheduled first. Once the priority heap is created, the logic wires are scheduled sequentially starting with the top of the heap.

The pseudocode descriptions of the list-scheduling algorithm, the ASAP algorithm, and the ALAP algorithm are given in Figs. 7–9, respectively.

V. Fine Grain Scheduling

Babb *et al.* later proposed TIERS [12]. The main difference between TIERS and PhaseRoute [2] is that TIERS overlaps propagation and transmission, whereas PhaseRoute requires a fixed worst case propagation time. TIERS eliminates the clustering of signal transmissions into phases. Instead, a signal is transmitted over the FPGA boundary as soon as its propagation is finished. Because transmission no longer needs to be synchronized to the phase boundary, the overall performance could be enhanced. Our scheduling algorithm is also applicable to this case by modeling the variable propagation time. The DFG model is modified by adding a delay node to each edge in the original DFG, as depicted in Fig. 10. The added nodes are annotated with the number of clock cycles needed for the signal to

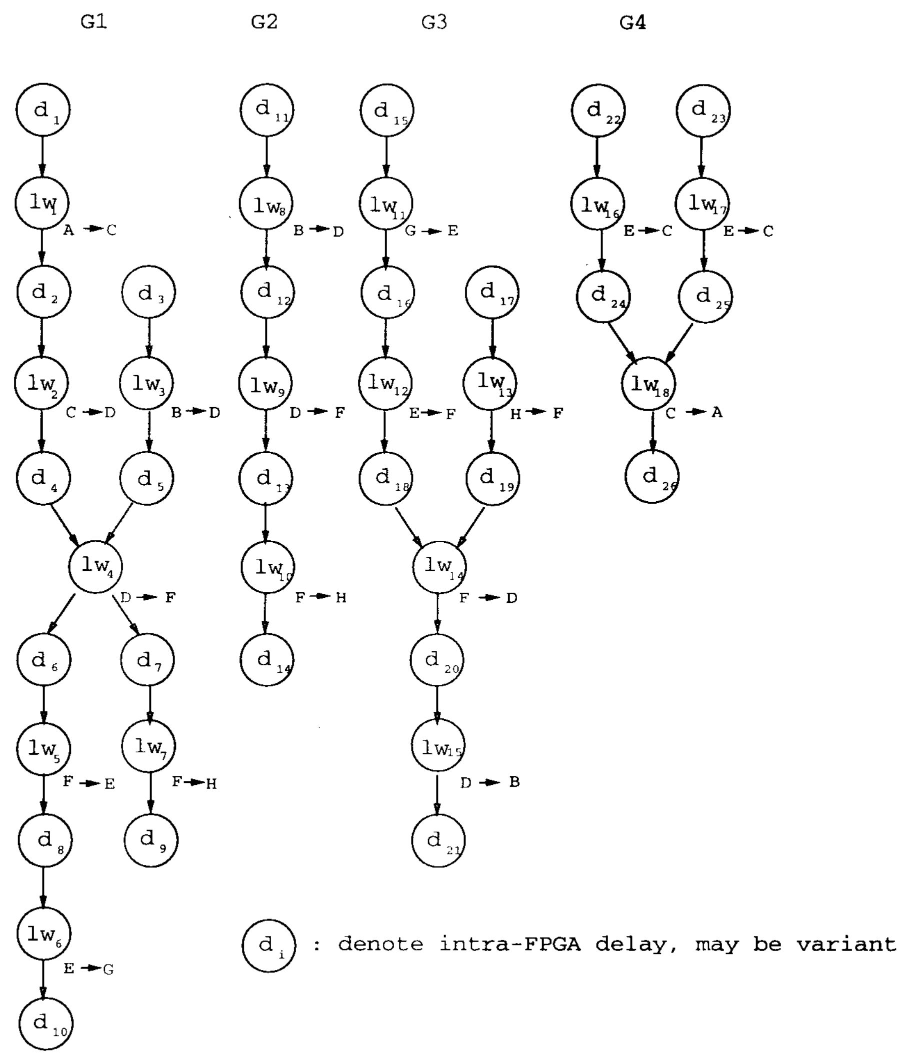


Fig. 10. Modified DFG with intra-FPGA delay inserted.

TABLE I
Comparison Between Without and with Virtual Wire Support

Circuit	#CLBs	W/O Virtual Wiring			W/ Virtual Wiring		
		#FPGAs	CLB Util.	I/O Util.	#FPGAs	CLB Util.	I/O Util.
s35932	1751	23 (XC3064)	34%	89%	16 (XC3064)	49%	100%
s38417	1921	26 (XC3064)	33%	85%	16 (XC3064)	54%	100%
s38584	2304	21 (XC3064)	49%	84%	16 (XC3064)	64%	100%
avq.small	6094	23 (XC4013)	46%	84%	16 (XC4013)	66%	100%
avq.large	6336	25 (XC4013)	44%	81%	16 (XC4013)	69%	100%

travel across the combinational path within the corresponding FPGA. After the change, we just need to view each FPGA cycle as a phase. Each logic wire is now scheduled into an FPGA clock cycle, and each intra-FPGA delay node is scheduled into one or more FPGA clock cycles.

VI. Experimental Results

We have implemented the proposed method in a C program. We test the effectiveness of the program using a set of benchmark netlists from the MCNC [16]. The target hardware emulator system consists of a 4 $\times$ 4 array of Xilinx FPGA's [19]. Each FPGA is only connected to its four neighbors. We assume that each phase is capable of transmitting two logic signals after four FPGA cycles of intrachip propagation delay. Therefore, an emulation cycle needs $(4+2) \times N = 6N$ FPGA cycles, where N is the number of phases. The partitioning of gates into FPGA's and the board-level routing are obtained by superimposing a 4 $\times$ 4 grid on a row-based layout generated by a simulated-annealing-based placement-and-routing tool, TimberWolf 6.0 [15]. That is, a net in the layout plan intersected by a grid line is mapped to a logical inter-FPGA connection in the corresponding boundary.

TABLE II
EXPERIMENTAL RESULTS FOR PHASE ASSIGNMENT AND FINE GRAIN SCHEDULING

Circuit	#Wires per FPGA boundary		#Clock per Emulation cycle		CPU(sec)		#PMF
	logical(max)	physical	phase assignment	fine grain	phase assignment	fine grain	
s35932	259	30	90	29	1.45	7.37	8.63
s38584	297	30	78	24	1.71	4.11	9.87
s38417	326	30	78	46	1.82	4.71	10.87
avq.small	316	48	66	28	1.35	4.28	6.58
avq.large	327	48	66	39	1.62	5.06	6.81

TABLE III
FEEDTHROUGH PERCENTAGE

Circuit	Total wires	Feedthrough wires	Feedthrough(%)
s35932	5359	2896	20.0%
s38584	5286	781	14.8%
s38417	5871	729	12.4%
avq.small	6439	1170	18.2%
avq.large	6799	1176	17.3%

The first experiment shows the effect of virtual wiring on the enhancement of the logic utilization of the FPGA's. We use a commercial tool [17] to partition a netlist into as few FPGA's as possible under both the logic capacity and I/O pin constraints. Different types of Xilinx FPGA's are used in different cases. Table I shows the number of FPGA's needed. Without virtual wire support, we need up to twice as many FPGA chips to emulate the circuits. Note that in the virtual wiring case, about 20% of the I/O pins are used for feedthrough routing, while in the partitioning case, it is assumed that no inter-FPGA routing will go through any other FPGA. Therefore, the virtual wiring would be more effective if more board-level routing resources are available.

The second experiment measures the effectiveness of virtual wiring in increasing the communication bandwidth. Table II summarizes the results for phase assignment and fine grain scheduling, respectively. Both the number of logic and physical wires are counted for one chip to and from one of its four neighbors. #Clock denotes the number of FPGA cycles needed for an emulation cycle. Both techniques increase the communication bandwidth by about six–ten times. We measure the effectiveness of the communication bandwidth increasing in terms of the pin multiplication factor (PMF) [11]. Fine grain scheduling results in faster emulation speed at the expense of more CPU time. The table shows that, without virtualization, even if we use the largest FPGA (XC4025 with 64 I/O pins per side), none of the circuits can be emulated with 16 FPGA's due to the violation of the I/O pin constraints. On the other hand, all circuits can be emulated if virtualization is used.

For the largest circuit (avq.large), the maximum number of logic wires going from one FPGA to one of its neighboring FPGA's is 327. That is, there are, on average, 1500 off-chip connections per FPGA. The number seems unreasonably large because there are only about 396 CLB's per FPGA. Actually, as shown in Table III, up to 20% of the connections are feedthroughs linking nonadjacent FPGA's.

VII. CONCLUSION

We have shown the equivalence relationship between the phase assignment problem of virtual-wire-based hardware emulation and the *resource-constrained operation scheduling* problem of high-level synthesis. We have proposed adopting the static-list scheduling heuristic [10] for the problem, and have demonstrated its effectiveness through experiments over a set of MCNC benchmarks [16]. Although our layout-based approach for partitioning and routing is adequate for the demonstrated cases, we shall improve it for better virtualization. Also, we would like to apply the approach to emulators of different architectures such as the partial crossbar [14]. On the FPGA side, it would be useful if there were built-in hardware support for virtual wiring.

ACKNOWLEDGMENT

The authors would like to thank the reviewers and Dr. T.-Y. Wu for their suggestions and comments.

REFERENCES

[1] P. Agrawal, S. Goil, S. Liu, and J. A. Trotfer, "Parallel model evaluation for circuit simulation on the PACE multiprocessor," in *Proc. 7th Int. Conf. VLSI Design*, Jan. 1984, p. 485.

[2] J. Babb, R. Tessier, and A. Agarwal, "Virtual wires: Overcoming pin limitations in FPGA-based logic emulators," in *Proc. IEEE Workshop FPGA-Based Custom Computing Machines*, Napa, CA, Apr. 1993, pp. 142–151.

[3] D. K. Beece, G. Deiberg, G. Papp, and F. Villante, "The IBM engineering verification engine," in *Proc. 25th ACM/IEEE Design Automation Conf.*, June 1988, pp. 218–224.

[4] A. Cao *et al.*, "CAD methodology for the design of *UltraSPARC*^{TM} *I* microprocessor at Sun Microsystems Inc.," in *Proc. 32nd ACM/IEEE Design Automation Conf.*, June 1995, pp. 19–22.

[5] M. Dahl, J. Babb, R. Tessier, and S. Hanono, "Emulation of the sparcle microprocessor with the MIT virtual wires emulation system," in *Proc. IEEE Workshop FPGA-Based Custom Computing Machines*, Apr. 1994, pp. 14–22.

[6] A. T. Eiriksson, "Mixed-level simulation with a zycard simulation engine," in *Proc. 3rd Annu. IEEE ASIC Seminar and Exhibit*, Sept. 1990, pp. P5/1.1–5.

[7] D. Gajski, A. C.-H. Wu, N. Dutt, and Y.-L. Lin, *High-Level Synthesis—Introduction to Chip and System Design*. Boston: Kluwer Academic, 1992.

[8] J. Gateley *et al.*, "*UltraSPARC*^{TM}*-I* emulation," in *Proc. 32nd ACM/IEEE Design Automation Conf.*, June 1995, pp. 7–12.

[9] L. Hagen, A. B. Kahng, F. J. Kurdahi, and C. Ramachandran, "On the intrinsic rent parameter and spectra-based partitioning methodologies," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 27–37, Jan. 1994.

[10] R. Jain, A. Mujumdar, A. Sharma, and H. Wang, "Empirical evaluation of some high-level synthesis scheduling heuristics," in *Proc. 28nd ACM/IEEE Design Automation Conf.*, June 1991, pp. 210–215.

[11] R. Tessier, J. Babb, M. Dahl, S. Hanono, and A. Agarwal "The virtual wires emulation system: A gate-efficient ASIC prototyping environment" presented at the ACM Int. Workshop Field-Programmable Gate Arrays, Berkeley, CA, Feb. 1994.

[12] C. Selvidge, A. Agarwal, M. Dahl, J. Babb, "TIERS: Topology independent pipelined routing and scheduling for virtual wire compilation," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 1995, pp. 25–31.

[13] S. Matic, "Emulation of hypercube architecture on nearest-neighbor mesh-connected processing elements," *IEEE Trans. Comput.*, vol. 39, pp. 698–700, May 1990.

[14] J. Varghese, M. Butts, and J. Batcheller, "An efficient logic emulation system," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 171–174, June 1993.

[15] C. Sechen, *TimberWolf6.0: Mixed Macro/Standard Cell Floor Planning, Placement and Routing Package, User's Manual*, Yale Univ., Sept. 1991.

[16] Layout synthesis benchmark set, Microelectronics Center of North Carolina, Research Triangle Park, NC, May 1990.
[17] ACEO, Inc., *ACEO Migration and Partitioning Reference Manual*. ACEO, 1995.
[18] Aptix, Inc., *The Aptix FPID Data Book*. Aptix, Feb. 1993.
[19] Xilinx, Inc., *The Programmable Logic Data Book*. San Jose, CA: Xilinx, 1994.

Logic Synthesis of Multilevel Circuits with Concurrent Error Detection

Nur A. Touba and Edward J. McCluskey

***Abstract*—This paper presents a procedure for synthesizing multilevel circuits with concurrent error detection. All errors caused by single stuck-at faults are detected using a parity-check code. The synthesis procedure (implemented in Stanford CRC's TOPS synthesis system) fully automates the design process, and reduces the cost of concurrent error detection compared with previous methods. An algorithm for selecting a good parity-check code for encoding the circuit outputs is described. Once the code has been selected, a new procedure called *structure-constrained logic optimization* is used to minimize the area of the circuit as much as possible while still using a circuit structure that ensures that single stuck-at faults cannot produce undetected errors. It is proven that the resulting implementation is path fault secure, and when augmented by a checker, forms a self-checking circuit. The actual layout areas required for self-checking implementations of benchmark circuits generated with the techniques described in this paper are compared with implementations using Berger codes, single-bit parity, and duplicate-and-compare. Results indicate that the self-checking multilevel circuits generated with the procedure described here are significantly more economical.**

I. Introduction

Concurrent error detection is an important technique in the design of systems in which dependability and data integrity are important. Concurrent error detection circuitry has the ability to detect both transient and permanent faults, as well as to enhance off-line testability and reduce BIST overhead [1]–[3].

One general approach for concurrent error detection is to encode the outputs of a circuit with an error-detecting code, and to have a checker that monitors the outputs and gives an error indication if a noncodeword occurs. A *systematic code* is a code in which codewords are constructed by appending check bits to the normal output bits. Using a systematic code for concurrent error detection has the advantage that no decoding is needed to get the normal output bits. Fig. 1 shows the general structure of a circuit checked with a systematic code. There are three parts: function logic, check symbol generator, and checker. The function logic generates the normal outputs, the check symbol generator generates the check bits, and the checker determines if they form a codeword. Two types of systematic codes that are used for concurrent error detection are Berger codes and parity-check codes [4].

Manuscript received December 19, 1995; revised August 12, 1996 and November 22, 1997. This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate, and administered through the Department of the Navy, Office of Naval Research under Grant N00014-92-J-1782, by the National Science Foundation under Grant MIP-9107760, and by the Advanced Research Projects Agency under Prime Contract DABT63-94-C-0045. This paper was recommended by Associate Editor A. Saldanha.

N. A. Touba was with the Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 USA. He is now with the Computer Engineering Research Center, Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712-1084 USA.

E. J. McCluskey is with the Center for Reliable Computing, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 0278-0070(97)07563-5.

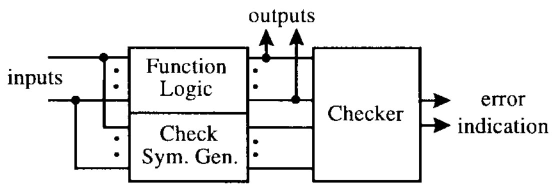


Fig. 1. Concurrent error detection using a systematic code.

While methods exist for designing PLA's and simple functional units (e.g., adders, multipliers, etc.) with concurrent error detection [4], the conventional approach for designing arbitrary multilevel circuits with concurrent error detection has been to use duplication. The circuit is simply duplicated, and the outputs are compared using a two-rail checker (equality checker). While this provides very high error-detection capability, it requires a large area overhead. Recently, research has been done on using automated logic synthesis techniques (such as those used in MIS [5]) to design multilevel circuits with concurrent error detection requiring less area overhead than duplication while still being able to detect all errors due to *internal single stuck-at faults* [6]–[8]. Internal single stuck-at faults are all single stuck-at faults, except those at the primary inputs (PI's). Note that for any concurrent error-detection scheme (including duplication), detection of stuck-at faults at the PI's cannot be guaranteed unless encoded inputs are used. However, if the inputs to the circuit are outputs of another concurrently checked logic block, then the only undetectable PI faults are break faults after the checker [9].

Jha and Wang [6] proposed a synthesis method in which the functional circuit is optimized using a MIS script with only algebraic operations such that the resulting circuit can be transformed so that it is *inverter free*, i.e., it has inverters only at the PI's. The primary outputs (PO's) are then encoded with a Berger code, which is a unidirectional error-detecting code. Since the inverters are only at the PI's, any error caused by an internal single stuck-at fault will produce a unidirectional error at the PO's, and therefore is guaranteed to be detected.

De *et al.* [7] have proposed two schemes for generating multilevel circuits with concurrent error detection. The first scheme uses a Berger code. It fully automates the synthesis method proposed in [6] by automatically adding the logic equations for the Berger check bits and checker, and then using a constrained technology mapping procedure that maintains the inverter-free property during technology mapping. The second scheme uses a parity-check code. A *parity-check code* is a code in which each check bit is a parity check for a group of output bits. Each group of outputs that is checked by a check bit is called a *parity group*, and corresponds to a row in the parity check matrix $\boldsymbol{H}$ [4]. Fig. 2 shows the parity check matrices $\boldsymbol{H}$ for a circuit with three outputs Z_1, Z_2, Z_3, encoded with single-bit parity and with duplication. In single-bit parity, there is one parity group which contains all the outputs. In duplication of a circuit with n outputs, there are n parity groups, each containing one of the outputs. The synthesis method proposed in [7] partitions the outputs to form