The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing

Jonathan Rose[1], Jason Luu[1], Chi Wai Yu[4], Opal Densmore[1], Jeffrey Goeders[3], Andrew Somerville[2], Kenneth B. Kent[2], Peter Jamieson[5] and Jason Anderson[1]

[1]Dept. Electrical and Computer Engineering, University of Toronto
[2]Dept. Computer Science, University of New Brunswick
[3]Dept. Electrical and Computer Engineering, University of British Columbia
[4]Dept. Electrical Engineering, City University of Hong Kong
[5]Dept. Electrical and Computer Engineering, Miami University

ABSTRACT

To facilitate the development of future FPGA architectures and CAD tools – both embedded programmable fabrics and pure-play FPGAs – there is a need for a large scale, publicly available software suite that can synthesize circuits into easily-described hypothetical FPGA architectures. These circuits should be captured at the HDL level, or higher, and pass through logical and physical synthesis. Such a tool must provide detailed modelling of area, performance and energy to enable architecture exploration. As software flows themselves evolve to permit design capture at ever higher levels of abstraction, this downstream full-implementation flow will always be required. This paper describes the current status and new release of an ongoing effort to create such a flow - the 'Verilog to Routing' (VTR) project, which is a broad collaboration of researchers. There are three core tools: ODIN II [10] for Verilog Elaboration and front-end hard-block synthesis, ABC [16] for logic synthesis, and VPR [13] for physical synthesis and analysis. ODIN II now has a simulation capability to help verify that its output is correct, as well as specialized synthesis at the elaboration step for multipliers and memories. ABC is used to optimize the 'soft' logic of the FPGA. The VPR-based packing, placement and routing is now fully timing-driven (the previous release was not) and includes new capability to target complex logic blocks. In addition we have added a set of four large benchmark circuits to a suite of previously-released Verilog HDL circuits. Finally, we illustrate the use of the new flow by using it to help architect a floating-point unit in an FPGA, and contrast it with a prior, much longer effort that was required to do the same thing.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Automatic Synthesis, Optimization

General Terms

Algorithms, Design, Architecture, Measurement, Performance

1. INTRODUCTION

The exploration of new programmable architectures, and the development of innovative algorithms required to synthesize circuits into FPGAs requires a robust software flow that permits experimentation. In order to model modern and future architectures, such a flow is necessarily quite complex, and largely beyond the capacity of any single academic enterprise to create, evolve and maintain. By contrast, the related commercial flows are supported by hundreds of full-time engineers. Equally important, to serve the same needs, is a set of relevant large-scale circuit benchmarks that can be used to test architectures and algorithms. This paper describes the status of a global collaboration attempting to provide such a framework - including several innovations within the three main parts of the tool flow, new work to create robust benchmarks, and an illustration of the flow's capability to explore a new kind of hard logic block.

One of the goals of this project is to provide a managed repository where enhancements of the flow by others can be more easily integrated into the suite of tools. Without such an effort, innovations are often orphaned, preventing our field from progressing by building on each other's work. These goals are challenging, because making a flow that is robust requires more effort than a typical academic project and publication requires.

In this paper we first describe several advancements in the core set of tools - Section 2 describe new features of the ODIN II tool [10] which takes in the Verilog description of the circuit and elaborates it into a BLIF netlist. Section 3 describes the use of the ABC tool [16], which performs logic optimization and technology mapping on the soft-logic portion of the BLIF. Section 4 describe advances in the VPR tool [13] which takes the synthesized BLIF and performs physical synthesis and timing analysis. Section 5 gives the basic flow's result for the set of previous and some new Verilog benchmark circuits that are a part of the related release. Section 6 provides a case study of the use of the flow to replicate previous work (done on a branch of the original VPR 4.3[3] flow) to model floating-point logic blocks. Section 7 gives the details of the full release of software, architecture files and benchmark circuits. Section 8 outlines the extensive set of additional features we see as necessary to continue this work, while Section 9 concludes.

2. ODIN II: ELABORATION

The Odin II Verilog elaboration front end [10] has four key roles in the VTR framework:

1. To interpret and convert some of the Verilog syntax into a logical netlist targeting the 'soft logic' on the FPGA.

2. To synthesize other constructs directly into 'hard logic' blocks on the FPGA, making specific use of the logical properties of those blocks to ensure that the logical netlist is physically realizable.

3. To be responsive to the architecture description of the FPGA. This is provided in the architecture description file which contains an extensive description of physical properties of the FPGA, together with a small amount of the logical properties. This includes the routing architecture of the FPGA [3], the internal structure of the logic blocks [13], the global pattern of logic blocks, and the I/O structure. Examples of a portion of an architecture file are given in Section 6.

4. To provide a framework for the verification of the correctness of the software flow.

This section reports on several new capabilities in these roles, including synthesis for memories, multipliers and other hard logic, a macro pre-processor, and a new verification infrastructure.

2.1 Compilation

It is essential for the elaboration step to be aware of the higher-level functionality of hard blocks in existing and hypothetical FPGAs. The Odin II Verilog compiler has been improved to enable more sophisticated mapping of hard multipliers and memories on FPGAs, which we describe in detail in this section, in addition to other 'generic' hard blocks that the architect can model.

2.1.1 Multipliers

Multiplication is a very common operation in digital circuits, and so modern FPGAs have included hard multipliers [2][1] for area-efficiency and higher performance. Odin II detects the multiplication operator (*) and synthesizes it directly into a hard multiplier block on the FPGA, if one exists in the architecture description file. Multiplication can appear in Verilog either as an explicit instantiation, or implicitly as a multiplier operation, as shown in Figure 1. Currently, Odin II can only synthesize unsigned multipliers.

The key issue with the elaboration and synthesis of multiplication is the transformation between the logical specification of the operation and its implementation using the available physical hard blocks [19]. The input circuit can contain any size of multiplier operation, whereas the FPGA architecture will typically have a physical hard block that is fixed in size. For example, a design could contain a logical 128-bit x 128-bit multiply that produces a 256-bit result, while the physical FPGA may contain only 8-bit x 8-bit multiplication hard blocks (as specified in the architecture description file) that provide a 16-bit result. This requires

```
multiply my_mult (a, b, out1);  // explicit

always @(c,d)
begin
    out2 <= c * d;  // implicit
end
```

Figure 1: Explicit & Implicit Multiply in Verilog

a 'splitting' of the large logical multiplier into many smaller multiplications that make use of the hard multipliers, and the synthesis of some soft logic, between the hard multipliers, to create the correct arithmetic function. This splitting operation can best be described through an example: Consider a logical multiplication operation that splits perfectly in half (into two multipliers of exactly half the size) the result is four multiplication operations (each of 50% the size) and three addition operations, as illustrated in Figure 2. The additions must be implemented in generated soft logic. Figure 2 shows the long multiplication form of A x B. If the resulting multiplication operations remain too large to implement in a hard multiplier, this splitting process is repeated recursively until the multiplication operation is small enough to fit.

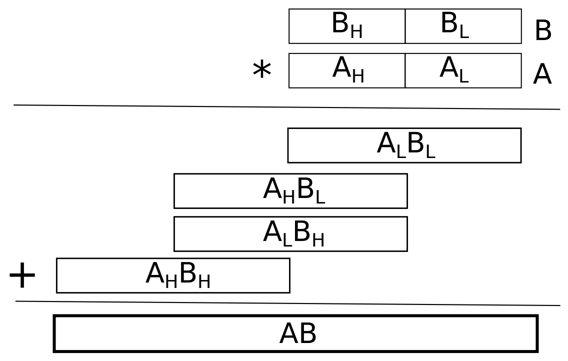


Figure 2: Long multiplication of variables A and B.

It is important to note that, in working with hard and soft multipliers, it has become clear that it is very inefficient to use larger physical hard multipliers to implement small logical multipliers - the resulting circuit is faster and smaller when small logical multipliers are built in soft logic. For this reason, Odin II has a parameter that sets the size of the logical multiplier that is too small to be implemented in a hard physical multiplier. This parameter is also applied to the remaining multiplier after the recursive splitting process described above. This parameter should be a function of the size of the smallest hard multiplier available on the FPGA; however we do not yet automate that setting.

2.1.2 Memories

Memory is a key component of digital circuits, and embedded memories are commonly found in many commercial FPGA architectures [1][2]. Odin II identifies explicit instantiations of memory (implicit memory coded as Verilog arrays is not yet supported) in the input Verilog circuit. Explicit memories, both single and dual port, can be specified by the designer as illustrated in Figure 4. These logical memories are synthesized into the hard blocks that exist in the target FPGA architecture, as specified in the architecture description file. Figure 3 illustrates part of the definition of a hard

```
<model name="single_port_ram">
  <input_ports>
  <port name="we"/>     <!-- control -->
  <port name="addr"/>  <!-- address lines -->
  <port name="data"/>  <!-- data lines -->
  <port name="clk" is_clock="1"/>
  </input_ports>
  <output_ports>
  <port name="out"/>   <!-- output -->
  </output_ports>
</model>
```

Figure 3: Excerpt of FPGA Architecture Description File Showing Model of Single Port Memory

```
single_port_ram my_mem(we, data, addr, clk, out);

dual_port_ram my_mem2(we1, we2, addr1, addr2, data1,
     data2, clk1, out1, out2);
```

Figure 4: Example Verilog Instantiations of Single and Dual Port Memories

block single-port RAM in the FPGA architecture description file, which is used by ODIN II to perform the synthesis. The keywords **single_port_ram** and **dual_port_ram** along with the ports for each shown in Figure 4 have specific meanings that are necessary to perform the memory synthesis within Odin II. A later physical layout section of the description file gives the size of the memory ports, which is also required.

Similar to multipliers, memories will appear in application circuits in many different logical sizes, and some will have to be split to fit into the physical size of memories on the FPGA. This may also require the generation of additional soft logic. The depth of a memory can be split in half by utilizing one of the address lines to select one of two smaller memories to access. The width of a memory can be split by utilizing two memories where the entries are concatenated together to achieve the required data width. Furthermore, most FPGA memories have the ability to trade depth for width, making the full memory synthesis problem somewhat intricate [9]. Odin II can be set to split the logical memories in two ways:

1. It can split a logical memory down into the size of the smallest physical memory, or

2. It can split all memories into the smallest data size possible (ie. 1-bit data). This method relies on downstream tools to pack memories together into the physical hard block size; we describe that more fully in Section 4. This can potentially render better results as multiple logical memories can share a single physical hard block.

2.1.3 Generic Hard Blocks

Multipliers and memories are common entities that are found in many digital circuits. A key goal of the VTR project is to enable exploration of other dedicated hard blocks, in order to explore their effectiveness. Odin II supports the detection and synthesis of such generic hard blocks. To use this feature, the design must explicitly specify the precise usage of the generic hard block (similar to explicit specification of memories in Figure 4) where the logical size is identical to the physical size. The specification of a generic hard block in the FPGA architecture is identical to that of memories and multipliers. A model must be described that specifies the name of the block along with the input and output port names. Subsequently in the physical layout of the FPGA the size of the ports must be provided. No splitting of generic hard blocks can be provided as this is dependent on the block's functionality. It is an open question as to whether there is a method of specification of a hard block that can concisely include how it can be split.

2.2 Pre-processor

An addition to Odin II in this release is a Verilog preprocessor that can support greater Verilog language coverage. The pre-processor performs an additional parsing of the Verilog circuit description prior to Verilog compilation. The pre-processing step provides support of Verilog compiler directives (i.e. `define`, `ifdef`, `else`, `endif` and `include`). This support has substantially increased the flexibility in writing Verilog benchmarks for the VTR project and has increased the likelihood that an existing circuit description can pass through the VTR flow without requiring modification.

2.3 Verification

As we have developed the VTR flow, it has become important to ensure that the output of Odin II, and ultimately the downstream tools, is correct. It is particularly difficult to verify that the output of Odin II is correct when the user defines new hard blocks with new functionality. This requires an ability to separately specify the logical functionality of those blocks to any kind of verification tool. To address this issue, a logic simulator was developed [17] inside ODIN II. The simulator exercises either the Verilog input (after elaboration into ODIN II's internal data structures) or a BLIF netlist file, as the specification of the circuit. It can use either an optional set of input test vectors that stimulate the circuit, or the simulator will generate a specified number of random vectors.

The logic functionality multiplication and memory hard blocks is built in to the Odin II simulation, as they are the most common blocks. When the FPGA architect wants to create custom hard blocks with unique functionality, the architect must also provide the simulator with a C-language description of the logical functionality. At simulation runtime, these blocks are simulated by loading these compiled codes as run-time libraries.

The output of the simulator is the computed result of the input vectors for the circuit, from either the Verilog or BLIF versions of the circuit. Once complete, any output vectors can be compared for equality to achieve a level of confidence that a circuit is synthesized correctly. In addition, the output of the post-synthesis simulation can be compared against the pre-synthesis simulation of commercial simulators, such as Modelsim from Mentor. In the benchmarks presented in Section 5, we have employed this latter method of verification.

3. ABC: LOGIC SYNTHESIS

We use the ABC logic synthesis framework [16] for technology independent logic synthesis and technology mapping to LUTs and flip-flops. ABC has evolved over the years to produce higher-quality results with a number of innovations. It has also evolved to handle both hard structures on FPGAs, as well as predefined soft structures, as discussed below. ABC represents a logic circuit using a network of two-input AND gates and inverters: an AND-inverter graph (AIG). We use ABC's `resyn2` script for technology independent optimization, which iteratively calls ABC commands that optimize the AIG to reduce the number of nodes and balance the lengths of its paths, thereby minimizing the maximum number of AND gates on any combinational path.

We also employ the WireMap [11] technqiue for technology mapping the AIG into K-input LUTs. WireMap produces depth-optimal mappings for a given AIG, while attempting to minimize the number of *used* inputs in the resulting LUTs. This reduction in LUT inputs benefits both routability and power [11], as well as facilitating a more efficient packing of these smaller LUTs into dual-output fracturable LUT architectures [13].

While soft logic is represented as an AIG in ABC, hard blocks, such as memories and multipliers, are received from the Odin II front-end modelled as *black boxes* in ABC. More recent versions of ABC permit the modelling of timing paths through black boxes which allows the synthesis and mapping steps to optimize the surrounding soft logic while taking that timing into account. We do not yet take advantage of this feature of ABC, but hope to do so in the near future. To do so will require experimentation and measurement of its effectiveness. We will also work similarly to move to modelling hard blocks as *white boxes*, wherein their internal logic functionality is exposed. The logic circuits in the transitive fanin and fanout of the white box can then be further optimized, for example, by leveraging don't-cares arising from the white box logic functionality.

4. VPR: PHYSICAL SYNTHESIS

This release of VTR includes an important new release of the VPR tool that is based on the VPR 6.0 beta release described in [13]. That version introduced new constructs that allow the description and packing of far more complex logic blocks. The architecture description file can now describe an essentially arbitrary interconnection of primitives inside the block, together with unlimited layers of hierarchy and multiple modes of operation within each piece of the hierarchy. That version, however, suffered from a lack of timing-driven physical synthesis - none of the packing, placement or routing phases was timing driven.

In the present release, timing-driven functionality has now been implemented. To do this, the timing analyzer in VPR, and most particularly the timing graph generator inside the timing analyzer, needed to generate timing graphs that reflect the arbitrary graph of connectivity that is now possible inside the complex logic blocks. It must also correctly model the different modes of operation, with different timing numbers for each mode. This has now been done; since the VPR placement and routing algorithms only needed proper timing analysis to work, this enhancement was all that was required to make both of those steps timing-driven. There are two interesting issues that arose in this work that will be described in the two subsequent sections: first, we needed to create a timing-driven packing algorithm that could deal with the arbitrarily complex logic blocks. Second, we had to come up with a clean method for handling the complexity of specifying the timing inside hard and soft logic blocks

4.1 Timing-Driven Packing

We begin by describing the area-driven packing algorithm in [13] and then describe how it was enhanced. The area-driven algorithm first selects a complex logic block, and populates it with primitives from the netlist, one at a time, until the complex block is full. This is repeated until the entire netlist is packed into complex blocks. This algorithm resolves the mode and hierarchy requirements of a complex logic block by using a depth-first traversal of the hierarchy and mode tree to find the appropriate location and mode settings for each netlist block. For each such selection, the algorithm invokes a router to ensure that the specific choice is feasible - in the general case it is necessary to perform routing within the block because there is no guarantee that any arbitrary network of switches will allow the candidate packed primitives to connect. The selection of which netlist primitive to pack next is based on an attraction function that was entirely area driven.

To make this algorithm timing-driven, the attraction function was modified based on a new timing model. The timing model used is a simplified version of the model used by T-VPack [15]. All blocks are modelled to have a logical depth of one. Nets are modelled as having zero delay. The timing analysis engine calculates the normalized criticality of each netlist primitive by dividing by the longest-path logical depth. Tie breakers are employed to determine which block should have higher criticality in the event that two blocks share the same depth criticality. These tie breakers are the same as those found in [15]; they are based on counting the number of critical/near critical paths that pass through a particular block.

The attraction function was modified to account for timing as follows:

$$Attr = \alpha \cdot criticality(B) + (1-\alpha) \cdot area\_attraction(B) \quad (1)$$

Where B is the candidate netlist block B and *area_attraction* is the attraction function used in [13]. The parameter *alpha* defaults to 0.75 to place a heavier emphasis on criticality.

As with earlier works on packing, the criticality of blocks serves as a good proxy for packing critical edges into the cluster. Due to the strong weighting in the attraction function in favour of criticality, there is a bias for packed netlist blocks to have equal or higher criticality than candidates. If a packed block is critical and the candidate block it is connected to is critical, then the edge to the candidate from the cluster is critical too. Thus, block criticalities alone achieves the goal of absorbing critical edges most of the time while avoiding the larger development effort needed to implement edge criticalities.

We performed an experiment to compare the impact of timing-driven and area-only packing in VPR 6.0 using the legacy T-VPack/VPR 5.0 flow as the baseline. The largest 20 MCNC circuits were mapped onto a transistor optimized homogeneous FPGA from the iFar repository [12] with clusters of ten 6-LUTs. The timing-driven VPR 6.0 flow gave, on average, the same critical path delay as T-VPack/VPR 5.0. The area-driven packer followed by timing-driven placement and routing, on the other hand, produced circuits that were 5% slower on average.

4.2 Timing Specification

The goal of the VPR 6.0 release was to enable the architectural exploration of very complex logic blocks. With the addition of a fully-timing driven flow, it was necessary to ensure that proper timing modelling and analysis was performed at the various stages of the flow. It was necessary to rethink how the timing of the primitives are described in the architecture description file, so that a wide variety of new kinds of blocks can be correctly modelled. Timing paths through a primitive can be either purely combinational, or have registered inputs, and/or have registered outputs, and/or have an internal pipeline, or some combination of these. It turns out that for certain primitives, it is not necessary to completely specify all its timing paths. For example, Figure 5 shows the timing for a primitive with an internal pipeline. Notice that, for the purposes of physical synthesis, the primitive is indivisible so it is not necessary to specify the timing in this level of detail.

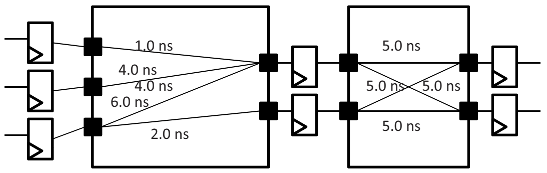


Figure 5: Example of an internal pipeline within a primitive.

Rather than have the architecture file specify the full timing graph itself, we decided that there was sufficient flexibility being able to specify delays in these four ways, dealing only with the input pins and output pins of the primitive:

1. Fully combinational paths - a different combinational delay can be specified from every input pin to every output pin in a timing matrix. If all of the delays to a specific output pin are the same, then this can be specified more concisely.

2. Input pins that feed flip-flops can be specified as having a set-up time to reflect the actual set-up time and any extra combinational delay in the path from the pin to the D input of the flip-flop.

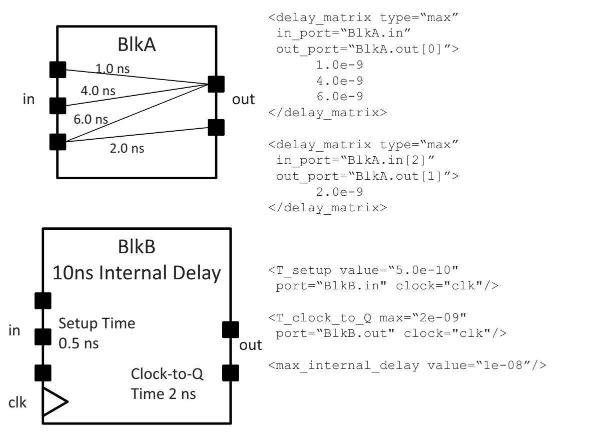
3. Output pins that are fed by the output of flip-flops can be specified as having a clock-to-Q delay, which can also include any extra combinational delay in the path from the Q output to the pin.

4. In the case that there are internal pipeline stages that are not visible with the above specifications, the architect can also specify the minimum clock period for the primitive. By having only one possible specification here, we limit the flexibility of the timing analysis. However, if more accuracy is needed, then a more detailed set of primitives can be used, essentially creating the timing graph as part of the complex block.

Figure 6 provides two examples to illustrate these scenarios. The timing for *BlkA* illustrates a primitive with fully combinational paths. The timing for *BlkB* illustrates the timing information for primitive with an internal pipeline. This primitive has a setup time at the input pins, a clock-to-Q delay at the output pins, and a longest combinational path delay of 10 ns.

5. BENCHMARKS AND FLOW RUN

An important part of this software release is the concomitant release of benchmarks circuits that can be processed through the circuit flow. We have come to realize that the benchmarks associated with a major software effort such as this are both as important as the software itself, and are themselves a form of software. As the language coverage of the ODIN II tool improves, it is clear to us that specific versions of benchmarks must be associated with specific releases of the tool flow. In this section we describe the benchmarks and give the results of the running of the new flow on each.



```
<delay_matrix type="max"
 in_port="BlkA.in"
 out_port="BlkA.out[0]">
      1.0e-9
      4.0e-9
      6.0e-9
</delay_matrix>

<delay_matrix type="max"
 in_port="BlkA.in[2]"
 out_port="BlkA.out[1]">
      2.0e-9
</delay_matrix>

<T_setup value="5.0e-10"
 port="BlkB.in" clock="clk"/>

<T_clock_to_Q max="2e-09"
 port="BlkB.out" clock="clk"/>

<max_internal_delay value="1e-08"/>
```

Figure 6: Examples of the different timing scenarios that can be expressed for a primitive.

5.1 The Circuits

Table 1 list the set of 19 benchmarks provided with this release, including four new circuits and several modifications to previously-released circuits. The table gives the circuit name, the number of primary inputs and outputs, the number of 6-input lookup tables in the combinational soft logic, the number of flip-flops, the number of 36x36 multiplier equivalents in each circuit, the number of logical memories, the maximum data width of all the logical memories in the circuit and the number of address bits in the deepest logical memory.

Each benchmark is coded in the Verilog HDL, and in many cases been recoded to meet the language coverage restrictions of ODIN II. There are three significant new additions to the benchmark suite from [13]: The MCML circuit, which is an application that uses Monte Carlo simulation of photons that could be used as part of a Photo-Dynamic Therapy-based cancer treatment plan [14]. The second circuit has provided two separate benchmarks - LU8 and LU32. This is a scalable linear system solver that makes use of the LU Decomposition Method [20]. The third circuit, bgm, is a Monte Carlo simulation for a financial application that uses the BGM interest rate model to price derivatives [8]. The other benchmarks come from a variety of sources: Opencores (or1200, sha), various university research projects (blob_merge, raygentop, boundtop, diffeq1 and diffeq2 ch_instrinsics and stereovisionX) and an FPGA consultant (mkDelayWorker32B, mkSMAdapter5B, and mkPktMerge).

The circuits range in size from 170 to 99,700 6-LUTs. Three of the new circuits added to the benchmark set are of significant size, although we note that even so, we are not keeping up with the size of the modern, largest FPGAs, which contain roughly to 500,000 6-LUTs.

Circuit	# In	# Out	# 6-LUTs	# FFs	# Mult	# Mem	Max width	Max Addr bits
bgm	257	32	30089	5362	11	0	0	0
blob_merge	36	100	6016	735	0	0	0	0
boundtop	275	192	2921	1671	0	1	32	14
ch_intrinsics	99	130	413	233	0	1	8	14
diffeq1	162	96	434	193	5	0	0	0
diffeq2	66	96	277	96	5	0	0	0
LU8PEEng	114	102	21954	6630	8	9	256	14
LU32PEEng	114	102	75530	20898	32	9	1024	14
mcml	36	33	99700	53736	30	10	36	16
mkDelayWorker32B	511	553	5580	2491	0	9	313	14
mkPktMerge	311	156	226	36	0	3	153	14
mkSMAdapter4B	195	205	1977	983	0	3	61	14
or1200	385	394	2963	691	1	2	32	14
raygentop	239	305	2134	1423	18	1	21	14
sha	38	36	2212	911	0	0	0	0
stereovision0	157	197	11462	13405	0	0	0	0
stereovision1	133	145	10366	11789	152	0	0	0
stereovision2	149	182	29849	18416	564	0	0	0
stereovision3	10	30	174	102	0	0	0	0

Table 1: Benchmarks and Data

Circuit	Pack Time (s)	Place Time (s)	MinW Route Time (s)	Route Time (s)	Min W (Tracks)	Crit Path Delay (ns)
bgm	2314	1768	12265	77	168	29.3
blob_merge	624	107	475	7	100	14.2
boundtop	522	37	17	2	72	7.57
ch_intrinsics	52	4	2	0	48	3.72
diffeq1	25	5	7	2	62	19.1
diffeq2	17	3	7	1	52	17.7
LU8PEEng	2526	1322	2564	116	136	149
LU32PEEng	9395	9983	109313	926	204	149
mcml	12674	5901	14173	168	144	109
mkDelayWorker32B	1115	154	1368	36	110	7.43
mkPktMerge	11	8	53	6	50	3.48
mkSMAdapter4B	481	25	44	2	80	7.80
or1200	373	55	92	4	90	24.0
raygentop	431	29	16	2	74	6.46
sha	516	21	20	2	64	15.6
stereovision0	1538	172	93	6	78	4.54
stereovision1	2500	209	539	18	120	5.89
stereovision2	3160	923	157888	276	172	16.9
stereovision3	20	1	0	0	30	3.51

Table 2: Data from Basic Flow Run

One key contribution of this work is the more careful verification of the elaboration stage (through Odin II) of the Verilog HDL code of these benchmarks. For 14 of the 19 circuits, there is an exact simulation match (of randomly generated vectors) between the output of the ODIN II simulator, and Modelsim simulation of the same code and vectors. For the other 5 circuits, (raygentop, boundtop, bgm, mkSMAdapter4B, and mkDelayWorker32B) the variations were minor.

5.2 Running the Flow

These Verilog circuits were run through the VTR flow targeting a hypothetical 40 nm FPGA architecture, which contains soft logic clusters of 10 fracturable LUTs. In this architecture, each fracturable LUT can operate as either a single 6-input LUT or two 5-input LUTs that share all five inputs, similar to the Virtex 6 FPGA [2]. The delays for this cluster were scaled from a 45 nm 6-LUT FPGA found in the iFar repository [12]. The routing architecture consists of segments of only length 4 wires, with Fc(In) set to 0.15 and Fc(Out) = 0.1 [3]. Its delay model was taken from the same iFar model used for the soft logic. The memory block in this architecture is similar to the Altera Stratix IV M144K memory block [1]. It contains 144K bits, and can act either as a single-port or dual port RAM. In single-port mode, the largest data width is 72 bits, and the smallest width is 9 bits; the maximum depth is 16K words. In dual-port mode the maximum width is 36 bits, and the maximum depth is 16K words. The memory speed was based on the speed of the Stratix IV M144K block. Each multiplier in the architecture can operate as one 36x36 or two independent 18x18 multipliers, which in turn can operate as two independent 9x9 multipliers. The multiplier delays were set to be the same as the Stratix IV DSP block.

The circuits are run through the flow in the following way: first through Odin II and ABC to create the pre-packing netlist. Odin II is set to target the specific physical memory and multipliers described above through a related description in the architecture file. Then the minimum channel width (the number of tracks per channel, as is often measured) is determined by running VPR's packing, placement and routing in non-timing-driven mode. (Here, as usual, the router is run repeatedly to find the smallest number of tracks per channel, W, which will succeed in routing.) Finally, the VPR flow is again invoked, using timing-driven routing with the channel width set to 1.3 times W. The latter measurement is used to determine the final critical-path delay of the circuit. The results of this flow are shown in Table 2. The first column lists the circuits in the benchmark. In the VTR flow, the VPR stage dominates the runtime so the next four columns that follow are the packing, placement, minimum channel width routing, and final routing runtimes for VPR in seconds. The last two columns show the key circuit statistics - minimum channel width and critical path delay.

One aspect of these results stand out - the runtime for packing, compared to placement and routing, is very large. This is caused by the part of the packing algorithm that invokes a router to determine if a specific primitive can be connected correctly within the logic block. This feature allows the packer to handle any arbitrary internal routing structure within the logic block, which we feel is important. However, we plan to reduce this runtime when specific flexibile structures, such as crossbars, are present. This is left as future work.

To illustrate the new timing-driven nature of the VPR portion of the flow, we measure the effect of each stage's timing-driven algorithm for packing, placement and routing. To do so, each circuit was run through VPR, holding its channel width at 1.3 * min W given in Table 2 (as is fairly common to create a low-stress routing [3]), but turning the timing-driven setting for each of placement and routing on and off. The results are shown in Table 3. The first column lists the circuit name followed by the critical path delays for each run normalized to the default, fully timing-driven run. The stages of the flow that have timing turned off are labelled after the NT prefix. For example, the column labelling NT PackPlace is a flow with non-timing-driven packing and placement, but with timing-driven routing.

The last row shows the geometric mean of these ratios. We see that turning off timing-driven placement results in the least impact on critical path delay with only a 3% increase on average. We also see that the stages are not independent: turning off timing for all stages results in a 22% increase to critical path delay on average but multiplying the individual

Circuit	Full Timing	NT Route	NT Place	NT Pack	NT PackPlace	NT
bgm	1.00	1.06	1.01	1.14	1.24	1.30
blob_merge	1.00	1.02	1.08	1.14	1.18	1.20
boundtop	1.00	1.04	1.05	1.15	1.34	1.44
ch_intrinsics	1.00	1.02	0.96	1.02	1.09	1.17
diffeq1	1.00	1.06	1.00	1.04	1.04	1.07
diffeq2	1.00	1.05	1.01	1.03	1.00	1.06
LU8PEEng	1.00	1.02	1.01	1.08	1.10	1.13
LU32PEEng	1.00	1.03	1.05	1.09	1.14	1.17
mcml	1.00	1.01	1.02	1.13	1.20	1.23
mkDelayWorker32B	1.00	1.04	1.06	1.16	1.28	1.36
mkPktMerge	1.00	1.00	1.04	1.06	1.02	1.02
mkSMAdapter4B	1.00	1.07	1.02	1.03	1.03	1.12
or1200	1.00	1.01	1.02	1.17	1.20	1.22
raygentop	1.00	1.06	1.00	1.28	1.30	1.30
sha	1.00	1.07	1.03	1.16	1.18	1.22
stereovision0	1.00	1.06	1.03	1.15	1.25	1.33
stereovision1	1.00	1.00	1.06	1.32	1.42	1.43
stereovision2	1.00	1.02	1.15	1.05	1.26	1.29
geomean	1.00	1.04	1.03	1.12	1.17	1.22

Table 3: Impact of timing-driven algorithms in different stages of CAD flow

delay increases from different stages results in a lower gain of 20%.

6. EXAMPLE: FLOATING-POINT BLOCKS

In this section we illustrate the power of the VTR framework showing how a hard block modelled in a previous research project (at great effort) can be modelled in the VTR framework with far less effort. The goal of the previous project [4] was to improve the computational efficiency of floating-point-heavy applications on FPGAs. It explored the architecture of a floating-point hard block, and showed that, for certain floating-point intensive applications, an FPGA employing the new block consumed a factor of 25 times less area and the speed of the resulting circuit increased by four times. In the following we describe the floating-point block, and show how it can be captured in the new complex block architecture description language. We then run the VTR flow and compare the results with the previous work, and comment on the relative effort required.

6.1 Architecture of Floating-Point Block

Figure 7 illustrates the generic architecture of the floating-point block that was explored in [4] that we will model in the VTR flow. The block consists of three basic elements: floating-point multipliers (FMs), floating-point adders (FAs) and wordblocks (WBs). The wordblocks are used for fixed-point arithmetic and logical operations. The FAs, FMs and WBs are connected in series using bus-based routing, which is provided by the multiplexers shown in the figure.

For the purpose of architecture exploration, several parameters are used to explore the architecture of the block, including bus width *(N)*, number of input buses *(M)*, number of output buses *(R)*, number of feedback paths *(F)*, number of blocks *(D)* and number of FA and FM *(P)*. The work in [6] determined good choices for these parameters.

6.2 Architecture Description in VTR

The description of the block shown in Figure 7 was rendered in the new complex block architecture description language described in [13]. As described in Section 2, the first part of the architecture file gives the atomic primitive constructs that must be instantiated in the Verilog input code,

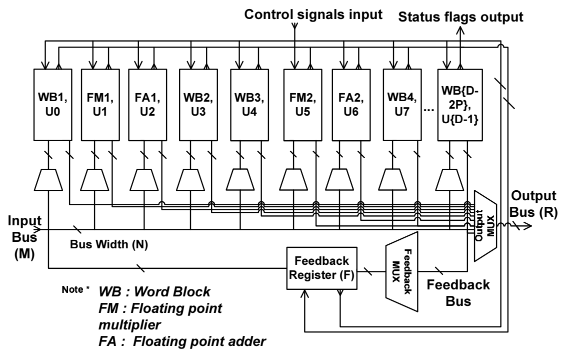


Figure 7: Architecture of the floating-point block

```
<model name="fpu_mul">
<input_ports>
    <port name ="clk" is_clock="1"/>
    <port name="opa"/>
    <port name="opb"/>
</input_ports>
<output_ports>
    <port name="out"/>
    <port name="control"/>
</output_ports>
</model>

<model name="fpu_add">
<input_ports>
    <port name ="clk" is_clock="1"/>
    <port name="opa"/>
    <port name="opb"/>
</input_ports>
<output_ports>
    <port name="out"/>
    <port name="control"/>
</output_ports>
</model>
```

Figure 8: Architecture Description of Floating Point Block ("fpu_mul" is the primitive of FM and "fpu_add" is the primitive of FA)

and will appear in the netlist that are sent to the packing stage, and is shown in Figure 8. Here the model "fpu_mul" describes the primitive of the floating point multiplier, FM, and "fpu_add" describes the primitive of the adder, FA. In both primitives, the port *clk* provides the clock signal to the registers in the block, and ports *opa* and *opb* are two floating-point inputs of the primitives. The port *out* is the floating-point addition or multiplication output. The port *control* emits status flags that indicate such things as the result is not a number value and overflow.

The FPU block contains the primitives illustrated in Figure 7, and the synthesis flow produces a netlist containing those primitives. The packing step packs those primitives into the physical block. For this to work, there must be a clear description of the contents and interconnection between the primitives in the architecture description file. Figure 9 gives an excerpt of the architecture description file that corresponds to the FAs, FMs and WBs and some of their interconnect. The physical block name of the floating-point block is "block_FPU" and is given in line 1. The "FPU" on line 12 is one of the configuration modes of this block, as different modes can be set for the overall "block_FPU". The bus width for all the floating point quantities is 32 bits (*N=32*) as shown by the **num_pins** construct on lines 2 to 9. There are four input busses (*M=4* from in1 to in4) and three output busses (*R=3* from out1 to out3). There are a total of eight primitives (D=8), including two floating-point adders and two floating-point multipliers (P=2). The description of the first FM (FM1) is from line 25 to 36 and the description of first FA (FA1) is from line 37 to 48. Within those descriptions are the netlist designations of the primitives that must appear in the pre-packing BLIF netlist - for example ".subckt fpu_mul" is required to indicate an FM primitive. The <T_setup> tag (on line 31) provides the setup time of the input port of the primitive block and <T_clock_to_Q> tag (on line 33) is the output port delay time. The maximum delay of the block is specified by <max_internal_delay value> tag (on line 35), as described in Section 4.2. The four wordblocks are simply registers as described from line 18 to 24, which contain 32 registers in each wordblock. As such, these registers appear in the pre-packing netlist as the standard BLIF primitive ".latch." There are three feedback registers named "feedback_reg1", "feedback_reg2" and "feedback_reg3". The description of "feedback_reg1" is from line 51 to 57.

After providing the definitions of the atomic primitive constructs, the interconnection of FAs, FMs and WBs are described in the interconnect section beginning on line 60. Line 62 defines the direct connection of internal control signals. Line 69 describes a single bit of the input multiplexer of "WB1", which allows selection of the input from the four block inputs and the outputs of feedback registers.

6.3 Experiment

In this section we describe an experiment that replicates a prior research effort that explored the architecture of the floating point block [6, 7]. The goal of that experiment was to measure the area and delay impact of the presence (vs. absence) of the floating point block. We employ the same set of eight circuits, shown in Table 4, that were used in the prior research. The table gives the number of FA and FM instances used in each circuit, and the general nature of the circuit. Note that all of these circuits can be completely expressed with just these primitives, so in the case of a FPU-based FPGA, there is almost no need of soft logic blocks to implement these circuits.

Circuit	# of FA	# of FM	Nature
bfly	4	4	DSP kernel
dscg	2	4	DSP kernel
fir	3	4	DSP kernel
mm3	2	3	Linear Algebra kernel
ode	3	2	Linear Algebra kernel
bgm	9	11	Finance application
syn2	5	4	Synthetic circuit
syn7	25	25	Synthetic circuit

Table 4: Floating-point benchmark circuits

In the case that the floating point block is not present, the floating point operations are implemented in the soft logic of the FPGA; in this experiment and the prior research, the soft-logic implementation does not make use of hard integer multipliers. For both FPGAs (with and without the floating point unit), the soft logic CLB consists of four 4-input LUTs, similar to the Virtex-II. The routing architecture used for both FPGAs was Fc(In) = 0.15, Fc(Out) = 0.25, and all length four wire segments. The number of tracks per channel, W, was set to be 72.

We used a 130nm CMOS process technology for estimating the area and timing of both the soft logic CLB and the floating-point block. The area and delay of the soft logic CLB are based on the iFAR architecture file FPGA Repos-

```
 1 <pb_type name=”block_FPU” height=”8”>
 2 <input name=”in1” num_pins=”32”/>
 3 <input name=”in2” num_pins=”32”/>
 4 <input name=”in3” num_pins=”32”/>
 5 <input name=”in4” num_pins=”32”/>
 6 <output name=”out1” num_pins=”32”/>
 7 <output name=”out2” num_pins=”32”/>
 8 <output name=”out3” num_pins=”32”/>
 9 <output name=”control” num_pins=”32”/>
10 <clock name=”clk” num_pins=”1”/>

12 <mode name=”FPU”>
13  <pb_type name=”FPU_slice” num_pb=”1”>
14  <input name=”in1” num_pins=”32”/>
15  <input name=”in2” num_pins=”32”/>
16  .
17  .
18  <pb_type name=”WB1” blif_model=”.latch” num_pb=”32”
        class=”flipflop”>
19   <input name=”D” num_pins=”1” port_class=”D”/>
20   <output name=”Q” num_pins=”1” port_class=”Q”/>
21   <clock name=”clk” num_pins=”1” port_class=”clock”/>
22   <T_setup value=”3.88e−10” port=”WB1.D” clock=”clk”/>
23   <T_clock_to_Q max=”1.557e−10” port=”WB1.Q”
        clock=”clk”/>
24  </pb_type>
25  <pb_type name=”FM1” blif_model=”.subckt fpu_mul”
        num_pb=”1”>
26   <clock name=”clk” num_pins=”1”/>
27   <input name=”opa” num_pins=”32”/>
28   <input name=”opb” num_pins=”32”/>
29   <output name=”out” num_pins=”32”/>
30   <output name=”control” num_pins=”8”/>
31   <T_setup value=”3.88e−10” port=”FM1.opa” clock=”clk”/>
32   <T_setup value=”3.88e−10” port=”FM1.opb” clock=”clk”/>
33   <T_clock_to_Q max=”1.557e−10” port=”FM1.out”
        clock=”clk”/>
34   <T_clock_to_Q max=”1.557e−10” port=”FM1.control”
        clock=”clk”/>
35   <max_internal_delay value=”2.99e−9”/>
36  </pb_type>
37  <pb_type name=”FA1” blif_model=”.subckt fpu_add”
        num_pb=”1”>
38   <clock name=”clk” num_pins=”1”/>
39   <input name=”opa” num_pins=”32”/>
40   <input name=”opb” num_pins=”32”/>
41   <output name=”out” num_pins=”32”/>
42   <output name=”control” num_pins=”8”/>
43   <T_setup value=”3.88e−10” port=”FA1.opa” clock=”clk”/>
44   <T_setup value=”3.88e−10” port=”FA1.opb” clock=”clk”/>
45   <T_clock_to_Q max=”1.557e−10” port=”FA1.out”
        clock=”clk”/>
46   <T_clock_to_Q max=”1.557e−10” port=”FA1.control”
        clock=”clk”/>
47   <max_internal_delay value=”2.99e−9”/>
48  </pb_type>
49  .
50  .
51  <pb_type name=”feedback_reg1” blif_model=”.latch”
        num_pb=”32” class=”flipflop”>
52   <input name=”D” num_pins=”1” port_class=”D”/>
53   <output name=”Q” num_pins=”1” port_class=”Q”/>
54   <clock name=”clk” num_pins=”1” port_class=”clock”/>
55   <T_setup value=”3.88e−10” port=”feedback_reg1.D”
        clock=”clk”/>
56   <T_clock_to_Q max=”1.557e−10” port=”feedback_reg1.Q”
        clock=”clk”/>
57  </pb_type>
58   .
59   .
60  <interconnect>
61  <!−−Connection sequence:WB1−>FM1−>FA1−>
        WB2−>WB3−>FM2−>FA2−>WB4−−>
62   <direct name=”direct1” input=”FM1.control[7:0]”
63   output=”FPU_slice.control[7:0]”> </direct>
64   <direct name=”direct2” input=”FM2.control[7:0]”
        output=”FPU_slice.control[15:8]”></direct>
65   .
66   .
67   <!−−####### WB1 #######−−>
68   <!−− Input Mux WB1 in1 −−>
69   <mux name=”WB1_in1_mux1” input=”FPU_slice.in1[0:0]
        FPU_slice.in2[0:0] FPU_slice.in3[0:0] FPU_slice.in4[0:0]
        feedback_reg1[0:0].Q feedback_reg2[0:0].Q
        feedback_reg3[0:0].Q” output =”WB1[0:0].D”/>
70   .
71   .
```

Figure 9: Code for Architecture Description

itory [12]. The area included the routing resources at a channel width equal to 72 tracks, and the area of the CLB tile, including programmable routing was determined to be 5679 um^2.

The area and delay of the floating-point block (with parameters N=32, M=4, R=3, F=3, D=8 and P=2) was estimated, as in [7] by synthesizing the floating-point block into standard cells from UMC in their 130nm CMOS process, using the Synopsys Design Compiler. The area of the FPU block, including programmable routing was determined to be 498,847 um^2. Using these two values, we calculate that one FPU tile requires the same area as 88 CLB tiles. In the discussion that follows, area is expressed in equivalent CLB area units.

The total area consumed by a circuit is the sum of the area taken by the floating-point block (in equivalent CLBs) plus the number of soft logic CLBs used. The maximum delay of the floating-point block is 2.99ns which is specified in <max_internal_delay> in Figure 9.

All eight of the circuits described in Table 4 were implemented in two hypothetical FPGAs: one with a hard floating point hard block, and one without - the latter containing only soft logic CLBs as described above. We compared the speed and area consumed in both cases, and for the purposes of this paper, compared those results with the measurements in [4].

The speed comparison is shown in Table 5. The average speedup of the FPGAs with a hard floating-point block is 12.6 times. This number differs significantly from the average speed ratio measured in the previous work which was only 4 times [4]. In that work, the soft logic FPGA was a Xilinx Virtex-II device which employs fixed carry chains in the adders and the adders contained in the multipliers. Those dedicated carry chains are significantly faster than carry logic as implemented in the CLBs in this experiment - directly in the LUTs, and using the regular intra-CLB routing to connect. This speed difference shows up particularly in the soft multiplier implementation; the speed of the soft multipliers in most of the circuits alone was 35ns, accounting for the difference. This result clearly shows that our future work must include the modelling of high-speed carry logic to support these kinds of experiments.

Circuit	Soft-Only Critical Path (ns)	Hard-Logic Critical Path (ns)	Ratio
bfly	36.1	2.99	12.1
bgm	35.7	2.99	11.9
dscg	36.6	2.99	12.3
fir	36.0	2.99	12.0
mm3	35.3	2.99	11.8
ode	34.5	2.99	11.6
syn2	37.6	2.99	12.6
syn7	50.3	2.99	16.9
Geomean			12.6

Table 5: Delay Comparison of Hard v. Soft Logic

Table 6 shows the area comparison between the pure soft logic and FPU hard block FPGA, where area is measured in equivalent CLBs. On average, the the circuits implemented in the FPGA with the hard FPU block is 18 times smaller than the FPGA with only soft logic. This result is in the same ballpark as the result (25x) in [4].

6.4 Comparison of Effort

The prior research that explored the floating point block studied the optimization of its internal routing and logic architecture [6, 7] using a customized version of the VPR flow that was based on VPR 4.2, called VPH [5]. The development time of the VPH tool was roughly one year; the modelling using the new VTR flow took approximately 2 man-weeks of time, a significant reduction in effort. Overall, this experiment shows that VTR framework provides a platform to evaluate new complex blocks such as the floating-point block, and that it is a more efficient way to enable this kind of experiment. It is also useful to note that the prior flow used a commercial synthesis tool from Synplicity as the front end, which can only target existing FPGA architectures. This prevents the exploration of FPGA architecture parameters - for example changing the size of the LUT in the FPGA. The new VTR flow permits the changing of many more parameters in the FPGA architecture from synthesis through placement and routing.

Circuit	Soft-Only Area (CLBs)	Hard-Logic Area (Equiv CLBs)	Ratio
bfly	6405	264	24.3
bgm	16908	792	21.3
dscg	6371	440	14.5
fir	6215	352	17.7
mm3	4556	264	17.3
ode	3609	480	7.5
syn2	6553	264	24.8
syn7	39240	1584	24.8
Geomean			17.9

Table 6: Area Comparison of Hard v. Soft Logic

7. RELEASE

The release of this software and benchmarks can be found at the following location:

`http://www.eecg.utoronto.ca/vtr/`

It contains:

1. The source code for the specific versions of ODIN II, ABC, and VPR that are being released, which are compatible with the benchmarks being released.

2. A few sample architecture files including various memory architectures with different combinations of size and flexibility, a suite of different fracturable LUT architectures, and a few heterogeneous architectures with realistic timing numbers.

3. The 19 benchmark circuits, which are compatible with the release of the software.

4. Example scripts for running experiments as well as regression tests for the software. These tests come with golden results and a range of error bands.

5. A web page with documentation on how to run the various versions of the flow on the released benchmarks.

6. An issue-tracking site that users can report software issues on the flow.

8. FUTURE WORK

There is a great deal of future work to be done on VTR system to include all of the innovations already done, and some new things in the future. These include:

1. Multi-clock timing analysis and optimization.

2. Cross-block Carry-Chains. VPR, with its new more complex logic blocks, can model carry chains (which provided augmented arithmetic speed and density) within a single complex logic block. However, to model the standard practice of building inter-block carry chains, the placement algorithm has to be capable of aligning (typically vertically) the blocks with connecting carry logic. In addition, the front-end synthesis flow must correctly capture and emit carry chains in the correct circumstances.

3. Clock tree architecture. There should be a separately described set of clock tree architectures that can be explored, and used for more realistic modelling of clocks.

4. Verilog language coverage. parameters. One of the most laborious conversion issues for benchmark circuits is the lack of parameters in ODIN II's coverage of the Verilog language.

5. Libraries. To help acquire more circuits, we need to have Verilog libraries of standard cores, such as dividers, square root units, and floating point arithmetic.

6. Bus-based routing. To help connect FPGAs to data-oriented blocks, it would be good to integrate the routing of multi-wire busses into the router.

7. Power/Energy modelling. The back-end flow, VPR, needs to model the energy consumption of all architectures, similar to [18]. This will necessitate a way to properly model the new, more complex logic blocks.

8. Transistor-level modelling. The most accurate way to model many aspects of the FPGA architecture is to have a transistor-level model of the logic and the routing. To be sensible, these models must have proper electrical design, including sizing.

9. White and Black box modelling in ABC. To enhance the quality of logic synthesis, ODIN II will need to transmit information, contained in the architecture file, to ABC, making use of its white and black box synthesis capabilities.

9. CONCLUSIONS

This paper has described new features and benchmarks of the Verilog-To-Routing (VTR) flow, a publicly available synthesis flow that permits exploration of hypothetical FPGA architectures and new CAD algorithms. The release is now fully timing-driven, and comes with a set of larger benchmarks. We have shown how it can be used to model new FPGA logic structures far more easily than previous tools. This is a ongoing, world-wide collaboration which has much more work to do to make the tool suite more viable.

10. REFERENCES

[1] Stratix IV Device Family Overview. `http://www.altera.com/literature/hb/stratix-iv/stx4_siv51001.pdf`, 2009.

[2] Xilinx Virtex-6 Family Overview. `http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`, 2009.

[3] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs.* Kluwer Academic Publishers, Norwell, Massachusetts, 1999.

[4] C.H. Ho, C.W. Yu, P.H.W. Leong, W. Luk and S.J.E. Wilton. Floating-Point FPGA: Architecture and Modeling. *IEEE Trans. on VLSI Systems*, 17(2):1709–1718, Dec 2009.

[5] C.W. Yu. A Tool for Exploring Hybrid FPGAs. In *Proc. International Conference on Field Programmable Logic and Applications (FPL), PhD Forum*, pages 509–510, 2007.

[6] C.W. Yu, A.M. Smith, W. Luk, P.H.W. Leong, S.J.E. Wilton. Optimizing Floating Point Units in Hybrid FPGAs. *IEEE Trans. on VLSI Systems*, to appear.

[7] C.W. Yu, W. Luk, S.J.E. Wilton, P.H.W. Leong. Routing Optimization for Hybrid FPGAs. In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 419–422, 2009.

[8] G.L. Zhang and P.H.W. Leong and C.H. Ho and K.H. Tsoi and C.C.C. Cheung, D. Lee, R.C.C. Cheung and W. Luk. Reconfigurable Acceleration for Monte Carlo Based Financial Simulation. In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 215–222, 2005.

[9] W. K. C. Ho and S. J. E. Wilton. Logical-to-physical memory mapping for fpgas with dual-port embedded arrays. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pages 111–123, London, UK, 1999. Springer-Verlag.

[10] P. Jamieson, K. Kent, F. Gharibian, and L. Shannon. Odin II-An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *IEEE Annual Int'l Symp. on Field-Programmable Custom Computing Machines*, pages 149–156. IEEE, 2010.

[11] S. Jang, B. Chan, K. Chung, and A. Mishchenko. WireMap: FPGA technology mapping for improved routability and enhanced LUT merging. *ACM Trans. on Reconfigurable Technology and Systems*, 2(2):1–24, 2009.

[12] I. Kuon and J. Rose. Automated transistor sizing for fpga architecture exploration. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 792–795, New York, NY, USA, 2008. ACM.

[13] J. Luu, J. Anderson, and J. Rose. Architecture description and packing for logic blocks with hierarchy, modes and complex interconnect. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 227–236, New York, NY, USA, 2011. ACM.

[14] J. Luu, K. Redmond, W. Lo, P. Chow, L. Lilge, and J. Rose. Fpga-based monte carlo computation of light absorption for photodynamic cancer therapy. *Field-Programmable Custom Computing Machines, Annual IEEE Symp. on*, 0:157–164, 2009.

[15] A. Marquardt, V. Betz, and J. Rose. Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density. *ACM Int'l Symp. on FPGAs*, pages 37–46, 1999.

[16] A. Mishchenko et al. ABC: A System for Sequential Synthesis and Verification. `http://www.eecs.berkeley.edu/alanmi/abc`, 2009.

[17] P. O'Brien, A. Furrow, B. Libby, and K. Kent. A simple tractable approach to design tool verification through simulation and statistics. In *to appear in IEEE Conference on Field Programmable Technologies*, FPT '11. IEEE, 2011.

[18] K. K. W. Poon, A. Yan, and S. J. E. Wilton. A flexible power model for fpgas. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, FPL '02, pages 312–321, London, UK, UK, 2002. Springer-Verlag.

[19] S. Srinath and K. Compton. Automatic generation of high-performance multipliers for fpgas with asymmetric multiplier blocks. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 51–58, New York, NY, USA, 2010. ACM.

[20] W. Zhang, V. Betz, and J. Rose. Portable and scalable fpga-based acceleration of a direct linear system solver. In *International Conference on Field-Programmable Technology, FPT 2008.*, pages 17 –24, dec. 2008.