A State Machine Block for High-Level Synthesis

Shadi Assadikhomami
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
Email: shadiassadi@ece.ubc.ca

Jennifer Ongko
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
Email: jennifer.angelica@alumni.ubc.ca

Tor M. Aamodt
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
Email: aamodt@ece.ubc.ca

***Abstract*—FPGAs are being deployed in datacenters to enable improved energy efficiency and application acceleration. This paper explores whether FPGA designs can be improved to make them more effective in this new role. We explore the properties of applications after high-level synthesis has been applied and note that for irregular applications, a large fraction of FPGA resources may be consumed implementing finite state machines. For many applications the resulting state machines have states with a single successor and limited fan-out degree. We propose a mixed-grained logic block architecture exploiting these properties that can be integrated into current FPGA architectures, which reduces the area of the next state calculation in FSMs by more than $3\times$ in average without impacting performance.**

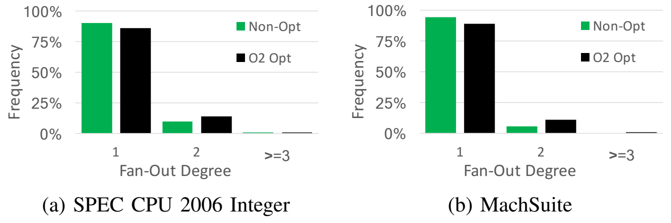


Fig. 1. Histograms showing number of successors in application *schedule flow graph* (a proxy for HLS generated FSMs using some existing HLS tools). Methodology in Section II.

I. Introduction

Technology scaling challenges motivate use of hardware accelerators, such as Field Programmable Gate Arrays (FPGAs), for cloud computing [1], [2]. Microsoft and Baidu have deployed FPGAs [3], [4], [5], and Amazon is offering FPGA instances in Amazon Web Services [6]. The acquisition of Altera by Intel [7], leading to FPGAs closely tied to CPUs, will likely encourage this trend.

The flexibility of FPGA lookup-tables (LUTs) incurs area, performance and power overheads relative to ASIC designs [8]. To bridge the gap, FPGAs include hard blocks, such as fixed and/or floating-point multiplier/accumulator and SRAM blocks [9]. The hard blocks in current FPGAs predate the trend towards using FPGAs inside datacenters. Moreover, while traditionally FPGAs were programmed with hardware design languages (HDLs), high-level synthesis (HLS) now achieves usable quality-of-results and is increasingly used for hardware design [10]. Thus, the question arises whether new coarse-grained operations would benefit future FPGAs. This paper argues finite state machines (FSMs) are good candidate for hard blocks as, depending upon workload and HLS tool, FSMs can represent a large portion of a circuit. While HLS from OpenCL input can efficiently use dataflow pipelines with small FSMs when sufficient thread-level parallelism exists, it is unclear if OpenCL is a good match for all datacenter workloads. Instead, this paper focuses on HLS from C/C++. FSMs for such workloads can represent a large fraction of total area in cases where controlling the datapath requires a large number of states and control signals [11]. FSMs generated during synthesis from C/C++ tend to have many states with a single successor as a consequence of their control data flow graph (Figure 1). To exploit this we propose an FSM block including a small RAM coupled to an adder.

Other works have also looked at accelerators for FSMs. Dlugosch et al. [12] propose a memory-based automata processor for automata-driven applications such as pattern matching and regular expressions. These applications are entirely defined as state machines. In contrast, this work focuses on FSMs that are generated by high-level synthesis tools. Here the FSM is only a portion of the application and acts as a controller for a datapath. These FSMs have different characteristics resulting in different architectures. Garcia-Vargas et al. [13] propose using memory units to implement next state and output calculation for every state. They reduce the size of the memory by multiplexing the FSM inputs to choose the set of active inputs at each state. Our FSM block also reduces memory size by exploiting the fact that typically only a subset of inputs are relevant in a given state. We further optimize memory size using an input encoder that exploits the fact that not all the combinations of active inputs contribute to different choices of next state.

We evaluate our proposed FSM block by detecting and extracting FSMs as standalone circuits from applications and compare them against baseline FSMs implemented purely in FPGA soft logic with/without memory blocks. We show that our proposed FSM block can reduce the area of the next state generation logic in FSMs by more than $3\times$ without impacting performance. An expanded version of this paper appears in the corresponding masters thesis [14]. We make the following contributions:

- We highlight dominant HLS generated FSM properties.
- We propose a state encoding technique and FSM block architecture to exploit these properties.
- We evaluate these and find area and critical path delay reductions of 70% and 45% respectively averaged across the evaluated FSMs.

II. HLS Generated Finite State Machines

High-level synthesis tools use the control/data flow graph (CDFG) of a given program to generate an RTL design composed of datapath and controller. The datapath corresponds to operations and data flow in the program while taking the available FPGA resources into account [10]. The controller is an FSM constructed after performing scheduling and binding. Below, we define and analyze specific characteristics of FSMs that can be exploited to design a custom FSM block that better utilizes the silicon area. In Section IV, we present HLS generated RTL designs showing that these characteristics are prevalent in HLS generated FSMs.

A. FSM Properties

1) Low Fan-Out Degree: The degree of a graph's vertex is the number of edges incident to the vertex. In directed cyclic graphs (DCGs) vertex degree can be broken into *fan-in degree* (the number of incoming edges of a vertex) and *fan-out degree* (the number of outgoing edges of a vertex).

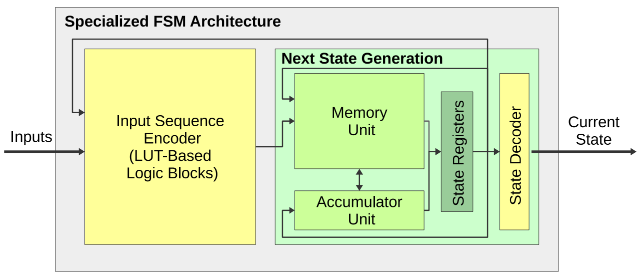
2) Abundance of Branch-Free Paths: We define a *branch-free path* of a DCG to be a directed path where each vertex (with any number of fan-in edge) has exactly one fan-out edge. For FSMs with long branch-free paths, consecutive states can be assigned consecutive state encoding values so that only an increment operation is needed for next state calculation.

B. Sources of Branch-free Paths

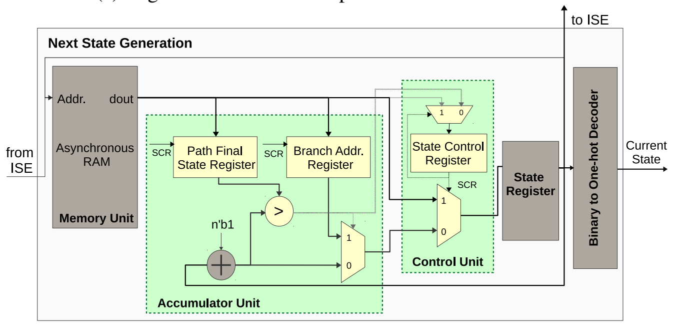
1) Operation Latency: We find that multi-cycle operations, such as divide and multiply, lead to states that belong to branch-free paths as often they are pipelined to run in parallel with independent operations. Directing these operations requires control signals at each cycle, which leads to additional branch-free states since there are no conditional operations and the operation latency is known in advance.

2) Data Dependency: To understand the impact of data dependency on scheduling, we performed an experiment to mimic the behaviour of an HLS scheduler by constructing a simplified equivalent FSM of a given program from its CDFG. We evaluated 12 SPEC CPU 2006 Integer benchmarks [15] and 19 MachSuite benchmarks [16] using LLVM [17].

First, the program's CDFG is constructed. Then, as-soon-as-possible (ASAP) scheduling is performed under the assumption that there is unlimited hardware and that all instructions execute in a single cycle. This results is what we call a *schedule flow graph*, which is a simplified equivalent FSM of the input program where each schedule cycle corresponds to an FSM state. Finally, we measure the fan-out degree.



(a) High-level view of the specialized FSM architecture.



(b) Next state generation block.

Fig. 2. Specialized FSM architecture.

The results are shown in Figure 1. Over 85% of states in SPEC CPU 2006 Integer and MachSuite benchmarks (both non-optimized and -O2 optimized) have only one possible next state. Thus, their HLS generated FSMs would contain many branch-free paths.

III. A Mixed-Grained Architecture for Finite State Machines Implementation

A. FSM Hard Block Architecture

Our proposed architecture (Figure 2a) is made up of fine- and coarse-grained logic connected via hard (or flexible FPGA) routing. Next state calculation is implemented in the coarse-grained logic using accumulator and memory units. The accumulator unit handles next state calculations for branch-free paths. The memory unit stores the remaining states' next state information along with metadata described below.

The following sections describe each component of the FSM block in more detail. A detailed view of the next state generation block is shown in Figure 2b.

1) Input Sequence Encoder Unit: Often only a subset of input signals impact the state transitions in each state of an FSM. These inputs are referred to as *active inputs*. Each state in FSMs extracted from our benchmarks (Table I) have between 3 and 56 inputs, but only 0 to 5 are active inputs. Moreover, the FSMs we examined have a maximum fan-out degree of 4, which means some combinations of active input values lead to the same next state. Thus, for these FSMs the choice of next state can be made with only 2 bits ($\log_2$(max number of reachable states)) instead of 5 (max number of active inputs). To achieve this we implement an encoder as a boolean function that maps a large FSM input sequence to a smaller *encoded input sequence* that is sufficient to select the

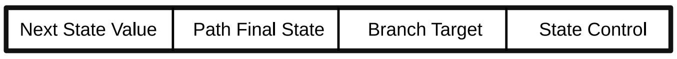


Fig. 3. Memory content

reachable next states. This results in a significant reduction in the size of the memory unit used for next state calculations as we avoid storing *don't care* data for unreachable states. The configurable input sequence encoder is implemented using soft logic on a LUT-based cluster as part of the conventional FPGA architecture.

2) Accumulator Unit: After applying our proposed state encoding, the next state calculation for branch-free states can be performed using an accumulator as described below:

- **Adder**: An adder increments the current state to calculate the next state in the branch-free path.
- **Control logic**: As memory addresses used in the FSM block RAM are distinct from the state encoding for branch-free paths two metadata registers are used to determine the end of a branch-free path and the next state after the final state in each branch-free path. In detail, the control logic contains the following: A **Path Final State Register** used to mark the ending state of a branch-free path. A **Branch Address Register** set to the memory address of the next state following the end of the branch-free path. A **Comparator** used to compare the path final state register with the accumulator. If they are equal, the next state is set to the Branch Address Register.

3) Memory Unit: For states that do not belong to a branch-free path, next state values and metadata are stored in the memory unit. To avoid adding a cycle for next state calculation, an asynchronous memory block is used. Figure 3 shows the content of a row in the memory unit. It has four main fields: (1) Next State Value, (2) Path Final State, (3) Branch Address, and (4) State Control bit. Fields (1) and (4) always have a valid value, however, fields (2) and (3) will only be valid if the next state is a part of a branch-free path. In this case, the contents of these two fields are inserted into the registers in the accumulator unit, as described previously. The last field (4) is used to determine if the source of the next state value should be the accumulator unit or the memory unit and is inserted into the control unit register described below.

The depth of the memory unit depends on the number of non-branch-free states while the width depends on the next state plus metadata that corresponds to Figure 3.

4) Control Unit: Responsible for selecting the source of the next state between the accumulator unit and memory unit.

5) State Decoder: Provides an optional binary to one-hot decoder at the output of this block to enable more efficient conversion if required by the rest of the circuit and reduce the decoding logic for the output calculation.

B. FSM Soft Implementation

In addition to the proposed FSM hard block, we propose a soft IP core that uses FPGA soft-logic and existing embedded memories on FPGAs to implement the equivalent functionality of the FSM hard block. In this approach, the synthesis tool is responsible for mapping the FSM description to the FSM soft IP. However, as will be discussed in Section IV, this approach is not as efficient as using our proposed hard block due to the overhead of large and configurable FPGA block RAMs.

C. State Assignment

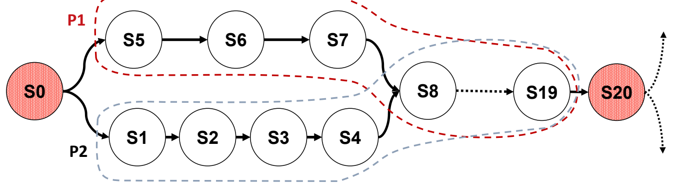
The state assignment problem involves determining the binary representation of states in an FSM such that each state has a unique value [18]. The state encoding directly affects the circuit area and performance as different encodings result in different circuit complexity. We propose a novel state assignment technique for FSMs targeting our FSM block, which aims to minimize the FSM area by mapping as many states to the accumulator logic as possible and by minimizing the number of states that reside in memory, hence reducing the complexity of the input encoder logic.

The proposed state assignment algorithm consists of two main parts: (1) identifying the state categories and (2) performing state encoding separately on each state category. These state categories are described below:

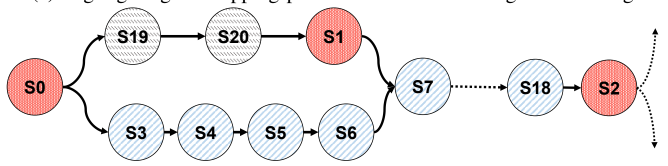
- **Branch-free states**: States with a fan-out degree of one in non-overlapping branch-free paths. States belonging to the overlapping portion of multiple branch-free paths will have multiple state encodings. Thus, we refine any overlapping paths such that each path is at least one *independent state* away from each other.
- **Independent states**: All remaining states that either have a fan-out degree greater than one (divergent states), or refined states that initially belonged to overlapping branch-free paths. These states are stored in memory to keep the metadata required for transitioning to/from divergent states or between branch-free paths.

Algorithm 1 for state assignment is described below:

- **Step 1 (Identify divergent vertices)**: Add vertices with a fan-out degree greater than one to $\mathbb{D}$ (line 9).
- **Step 2 (Identify branch-free paths)**: Find all of the branch-free paths between every two divergent vertices and add them to $\mathbb{P}_{nr}$ (lines 10-15). Branch-free paths are identified by traversing all consecutive vertices (by definition, with a fan-out degree of one) from the successor of one divergent vertex to the next divergent vertex.
- **Step 3 (Group branch-free paths)**: Add branch-free paths that share a common termination divergent vertex d_k to $\mathbb{P}_{k-nr}$, since this is a precondition for potential overlapping paths (lines 16-19).
- **Step 4 (Apply path refinement)**: Apply the path refinement algorithm on each group of branch-free paths with a common termination vertex in $\mathbb{P}_{k-nr}$, giving priority to the longest paths (line 21). The result is the subset of refined branch-free paths, $\mathbb{P}_{k-r}$, and the subset of independent states, $\mathbb{I}_k$, that are no longer part of the refined paths (details of this algorithm are omitted due to the page limit).
- **Step 5 (Update state categories-1)**: Add the subset of refined paths, $\mathbb{P}_{k-r}$, and independent vertices, $\mathbb{I}_k$, to the



(a) Highlighting overlapping paths P1 and P2 with original encoding.



(b) After applying path refinement and state assignment.

Fig. 4. State encoding and path refinement example.

final set of refined branch-free paths, $\mathbb{P}_r$ and independent vertices, $\mathbb{I}$ (line 22).

- **Step 6 (Update state categories-2)**: Add the divergent vertices, $\mathbb{D}$, to the list of independent vertices $\mathbb{I}$. Set $\mathbb{I}$ indicates all of the vertices (states) that will be mapped to the memory unit (line 24).
- **Step 7 (State assignment-1)**: For the final set of independent vertices, $\mathbb{I}$, assign incremental values to the vertices (states) starting from zero (lines 25-28).
- **Step 8 (State assignment-2)**: For each branch-free path in $\mathbb{P}_r$, assign incremental values to the consecutive vertices (states). The starting state value for the first branch-free path is one greater than the last independent state assigned in step 7. All remaining paths start one greater than the last state value of the previous path (lines 29-33).

Next, we present an example highlighting proposed path refinement and state assignment algorithms on a given FSM. Figure 4a illustrates part of an FSM state transition graph between two divergent vertices with the original state encoding. The dotted arrow between states S8 and S19 indicates the existence of more branch-free states in between. In step 1, we add the states with a fan-out greater than one, S0 and S20, to the set of divergent states (shown in red in Figure 4a). In step 2, we find all the branch-free paths that start from the successors of S0 and end at S20. This step results in two paths, $P1 =< S5, S6, ..., S19 >$, and $P2 =< S1, ..., S4, S8, ..., S19 >$. In step 3, P1 and P2 are identified as potential overlapping paths, since they share the common terminating state, S20. We apply the path refinement algorithm in step 4, which results in preserving the longer path, P2, and terminating P1 by cutting the path at S7, one state before it starts to overlap with P2. S7 becomes an independent state that will be mapped to memory to hold the metadata to transition between P1 to P2. In step 5 and 6, the refined paths will be added to the list of non-overlapping paths and S7 is added to the independent states (shown in Figure 4b). Finally, we perform the state assignment on each category according to steps 7 and 8 of the algorithm. The result of the state assignment algorithm is shown in Figure 4b.

Algorithm 1 State Assignment

```
Input: $G_{fsm} = (V, E) \rightarrow$ FSM state transition graph
Output: $G_{e-fsm} = (V, E) \rightarrow$ FSM state transition graph
    with the applied state encoding
 1: $\mathbb{P}_{nr} \rightarrow$ Set of non-refined branch-free paths
 2: $\mathbb{P}_r \rightarrow$ Set of refined branch-free paths
 3: $\mathbb{P}_{k-nr} \rightarrow$ Set of non-refined branch-free paths that share
    common terminating divergent vertex $d_k$ ($\mathbb{P}_{k-nr} \subset \mathbb{P}_{nr}$)
 4: $\mathbb{P}_{k-r} \rightarrow$ Set of refined branch-free paths after applying
    refinement algorithm on $\mathbb{P}_{k-nr}$ ($\mathbb{P}_{k-r} \subset \mathbb{P}_r$)
 5: $\mathbb{I} \rightarrow$ Set of independent vertices
 6: $\mathbb{I}_k \rightarrow$ Set of independent vertices found after applying
    path refinement algorithm on $\mathbb{P}_{k-nr}$
 7: $\mathbb{D} \rightarrow$ Set of divergent vertices
 8: $\mathbb{S}_i \rightarrow$ Set of successors of divergent vertex $d_i$

 9: $\mathbb{D} = find\_divergent\_vertices(G_{fsm})$
10: for all $d_i \in \mathbb{D}$ do
11:     for all $s_j \in \mathbb{S}_i$ do
12:         $path\_s_j = trav\_path\_until\_divergent(s_j, \mathbb{D})$
13:         $\mathbb{P}_{nr} = \mathbb{P}_{nr} \cup path\_s_j$
14:     end for
15: end for
16: for all $d_k \in \mathbb{D}$ do
17:     $path\_group\_d_k = paths\_share\_end\_vertex(\mathbb{P}_{nr}, d_k)$
18:     $\mathbb{P}_{k-nr} = \mathbb{P}_{k-nr} \cup path\_group\_d_k$
19: end for
20: for all $d_k \in \mathbb{D}$ do
21:     $(\mathbb{P}_{k-r}, \mathbb{I}_k) = path\_refinement(\mathbb{P}_{k-nr})$
22:     $\mathbb{P}_r = \mathbb{P}_r \cup \mathbb{P}_{k-r}$;    $\mathbb{I} = \mathbb{I} \cup \mathbb{I}_k$
23: end for
24: $\mathbb{I} = \mathbb{I} \cup \mathbb{D}$
25: $state\_val = 0$
26: for all $v_i \in \mathbb{I}$ do
27:     $encode\_memory\_states(v_i, state\_val{++})$
28: end for
29: for all $P_i \in \mathbb{P}_r$ do
30:     for all $v_j \in P_i$ do
31:         $encode\_branch\text{-}free\_states(v_j, state\_val{++})$
32:     end for
33: end for
```

IV. Evaluation

This section presents our methodology and evaluation of our proposed FSM architecture.

A. Experimental Setup

1) Benchmarks: Two sets of C/C++ benchmark sets developed for use with HLS tools are used to assist with the design and evaluation of our proposed architecture. The first, *MachSuite* [16], is a collection of benchmarks for evaluating accelerator design and customized architectures. The second, *HLS datacenter benchmarks*, is an in-house benchmark set consisting of selected functions identified as representing a significant portion of execution time in the open source applications Lucy [19] (search), SQLite [20] (database), and BZIP [15] (compression). The functions in this set are meant to be somewhat representative of workloads one would encounter inside a datacenter. To accommodate our HLS synthesis flow, portions of these functions were re-written to replace unsupported language features. We convert the benchmarks from C/C++ to Verilog HDL using Vivado HLS and use one-hot encoding for all generated FSMs to ensure that the baseline FSMs have the most area-efficient implementations.

2) FSM Extraction: To evaluate our proposed mix-grained architecture, we extract the FSMs from each benchmark while preserving the original state encoding. This is achieved as follows: We use the Yosys synthesis tool [21] to synthesize each benchmark to an RTL netlist. We then use the *FSM detection* and *FSM extraction* passes provided in Yosys to detect and extract the FSMs in KISS [22] format. We have developed an FSM generator in C++ which, given an FSM described in KISS, generates the Verilog HDL code for the FSM that conforms to the standard syntax used by Vivado HLS. Using this flow, we can extract the FSM from any benchmark and generate a stand-alone RTL design that describes this state machine. We preserve the original one-hot encoding generated by Vivado HLS during both steps of the FSM extraction such that the extracted standalone FSM has the identical state encoding as the original HLS-generated encoding. Statistics of the FSMs extracted from MachSuite and data center benchmarks are shown in Table I.

3) Baseline FPGA architecture: We use the architecture file *k6_frac_N10_40nm* provided in VTR [23] as the baseline FPGA architecture. We selected the simple architecture without any hard block as the baseline to minimize the area overhead of unused hard blocks that will not benefit the FSM.

4) Area and Delay Model: For the hard FSM block, the memory unit is modelled using the Artisan synchronous SRAM compiler [24]. The RTL design has been synthesized using the Synopsis Design Compiler (DC) vH-2013.03-SP5-2 [25] with the TSMC 65nm library. The area estimation obtained from DC are pre place-and-route. We estimate the routing area of the next state generation block, which is not calculated by the Synopsys DC as follows: We exclude the area of the RAM (since the internal routing is modelled by the SRAM compiler), then we multiply the area of the remaining units, which is reported by DC, by a factor of 2×. Note that with this approach, we are overestimating the area of the block, since the routing inside the next state generation unit is very limited. Thus, the presented area estimations are conservative.

The delay of the next state generation block is obtained from the DC and multiplied by a factor of $1.6\times$ to reflect the impact of place and route [26]. The area and delay of the input sequence encoder is obtained from VTR by mapping the encoder onto the baseline architecture. We then use the following formula (also used by VTR) to convert the logic and routing area reported by VTR in Minimum Width Transistor Area (MWTA) to um^2 for 65nm technology ($\lambda = 65$):

$$1 * MWTA = 60 * (\lambda)^2,$$

TABLE I
CHARACTERISTICS OF THE FSMS EXTRACTED FROM MACHSUITE AND HLS DATACENTER BENCHMARKS.

Benchmark	Abbrev.	States	Frac. of memory state	Inputs	Max fanouts
aes_fsm1	as1	47	0.75	6	2
aes_fsm2	as2	76	0.19	14	2
bckp_fsm1	bp1	11	0.81	11	2
bckp_fsm2	bp2	158	0.05	10	2
bckp_fsm3	bp3	69	0.06	6	2
bfs_b_fsm	bb1	8	0.38	7	3
bfs_q_fsm	bq1	8	0.38	6	2
fft_st_fsm	fs1	24	0.23	5	2
fft_tr_fsm1	ft1	17	0.30	8	2
fft_tr_fsm2	ft2	24	0.13	6	2
fft_tr_fsm3	ft3	219	0.06	14	2
fft_tr_fsm4	ft4	10	0.40	6	2
fft_tr_fsm5	ft5	66	0.04	5	2
gemm_fsm1	gm1	10	0.50	8	2
kmp_fsm1	kp1	7	0.30	4	2
kmp_fsm2	kp2	10	0.40	6	2
md_gr_fsm	mg1	15	0.47	10	2
md_knn_fsm	mk1	98	0.02	5	2
sort_m_fsm1	sm1	4	0.75	5	2
sort_m_fsm2	sm2	7	0.43	5	2
sort_r_fsm1	sr1	15	0.60	11	2
sort_r_fsm2	sr2	6	0.33	4	2
sort_r_fsm3	sr3	6	0.33	4	2
spmv_crs_fsm	spc	10	0.30	6	2
spmv_elpk_fsm	spe	9	0.33	6	2
stencil_fsm	ste	4	0.50	4	2
viterbi_fsm	vit	8	0.50	6	2
lucy_sh_fsm	lh1	71	0.02	3	2
sql_ln_fsm1	sl1	508	0.08	56	4
sql_ln_fsm2	sl2	7	0.29	6	3
sql_ln_fsm3	sl3	5	0.60	6	3
sql_ln_fsm4	sl4	10	0.60	10	3
sql_ln_fsm5	sl5	4	0.50	4	2
sql_ln_fsm6	sl6	4	0.40	4	2
lucy_sn_fsm	ln1	25	0.12	5	2
lucy_sv_fsm	lv1	12	0.67	10	4
bzip_fsm1	bz1	72	0.16	19	3
bzip_fsm2	bz2	41	0.17	11	2
bzip_fsm3	bz3	67	0.21	28	4
bzip_fsm4	bz4	17	0.35	9	3
bzip_fsm5	bz5	43	0.05	4	2
bzip_fsm6	bz6	61	0.16	19	3
bzip_fsm7	bz7	36	0.36	13	2
bzip_fsm8	bz8	117	0.21	34	3
sql_gt_fsm1	sg1	61	0.49	48	4
sql_gt_fsm2	sg2	12	0.50	9	2

For the soft FSM implementation, we use the architecture file *k6_frac_N10_mem32k_40nm* from VTR which provides embedded memory units. We use the same formula above to convert the VTR area numbers from MWTA to um^2.

B. Sizing of the FSM Block

The first design decision is to select an FSM block size that will accommodate the common FSM size, while reducing the amount of wasted resources for smaller than average FSMs. While evaluating different memory sizes, we found that the memory unit is always the main contributor to the block area.

TABLE II
THE SIZING CONFIGURATION OF THE ELEMENTS OF THE NEXT STATE GENERATION BLOCK.

Total Memory size	128x22 bits
Adder size	8 bits
State Register size	8 bits
Encoded Input Sequence size	2 bits

Thus, we can minimize the total area of our proposed FSM block by choosing the proper memory size.

For our evaluated benchmarks, we collected the required memory depth in terms of number of entries (independent states multiplied by the maximum number of next reachable states per state). For our workloads, 98% of the FSMs fit into a depth of 128. Thus, for the remainder of our evaluation, we selected a memory size with a depth of 128 entries.

The second design decision is the bit-width of the adder, control registers, and encoding bits, which are dependent on the total number of FSM states. All but one of our evaluated benchmarks contains less than 256 states. As such, we use 8 bits to represent the states. Table II shows the size of the units in the next state generation block.

To efficiently accommodate FSMs with a different number of states and to avoid hard block under-utilization, we also propose fracturable FSM hard blocks. The main idea behind fracturable FSM blocks is to tailor the block size such that it accommodates average sized FSMs while supporting combination of multiple blocks such that they can accommodate larger FSMs that do not fit into a single block. To map a larger FSM to multiple smaller combined blocks, the FSM needs to be partitioned into multiple sub FSMs and the architecture should enable fast transition between these blocks. We use the Fiduccia-Matheyses partitioning algorithm [27] to partition the FSM, which due to the properties of the HLS generated FSMs, results in a very low cut size between partitions. In the case where there is a transition between two fracturable blocks, we require a multiplexer before the state register in each sub block to support inter-block state updates. We find that the area overhead of splitting a larger FSM over two fracturable blocks is negligible compared to the area improvement. The data shown in Figure 7 includes the area overhead of the added logic required to make the FSM blocks fracturable.

C. HLS Optimization Impact on FSMs

We evaluate the impact of HLS optimizations for area-delay product minimization on the generated FSMs using HLS directives obtained by Lo et al. following the model described in [28]. We applied the HLS directive settings to 3 MachSuite benchmarks: *aes*, *backprop* and *sort radix*. Figure 5 shows the results averaged across the 3 MachSuite benchmarks. In this figure, the x-axis indicates the number of fan-outs per state and the y-axis indicates the fraction of total states that have the corresponding fan-out degree. On average, the optimized designs (opt) have a higher number of branch-free paths (fan-out degree of 1) than the non-optimized designs (no-opt).

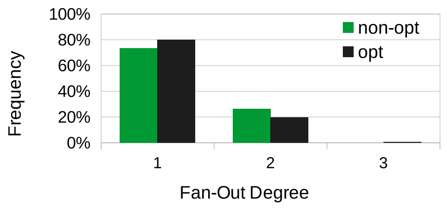
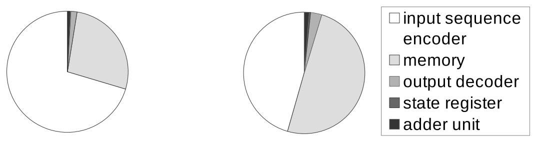


Fig. 5. Impact of applying HLS optimization directives on *backprop*, *aes*, and *radix sort* benchmarks from MachSuite.



(a) FSM $>$ 70 states (b) FSM $\leq$ 70 states

Fig. 6. Average area breakdown of the mix-grained FSM architecture for the FSMs in Table I.

We found that the addition of HLS directives did not significantly change the ratio of fan-out degrees 1 and 2 for specific FSMs, but instead generated additional FSMs for different parts of the same design, which, in the case of the 3 MachSuite benchmarks, contained a higher ratio of branch-free paths. This is due to many HLS directives attempting to exploit more parallelism, for example, by loop unrolling and loop pipe-lining. This results in an increase in the number of states to generate the control signals for the unrolled and pipe-lined loops, adding more branch-free states in between divergent states used to control the loops.

D. Area and Delay Improvement

First, we evaluate the area of each unit in the proposed architecture. Figure 6 illustrates the area of each unit as a fraction of the total area of the mix-grained architecture averaged across all of the FSMs extracted from our evaluated benchmarks. It can be seen that between 45% to 70% of the area is consumed by the input sequence encoder. This amount varies among benchmarks due to the variation in the number of states that reside in memory across FSMs.

We found that by using our proposed state assignment we were able to reduce the memory size used in the hard block by up to 2× over the original state encoding, when using the most area-efficient FSMs with one-hot encoding. Additionally, since the independent state values start from zero and are encoded before the branch-free states, there is a one-to-one mapping between the independent state encoding and the memory address space. Hence, no additional logic is required to calculate the memory address from the state encoding.

Second, we compare the area that our proposed architecture (hard block and soft FPGA IP core) consumes compared to the baseline LUT-based implementation. Figure 7 shows the result of this comparison (the x-axis is sorted in increasing order based on the number of states in each FSM). Overall, as

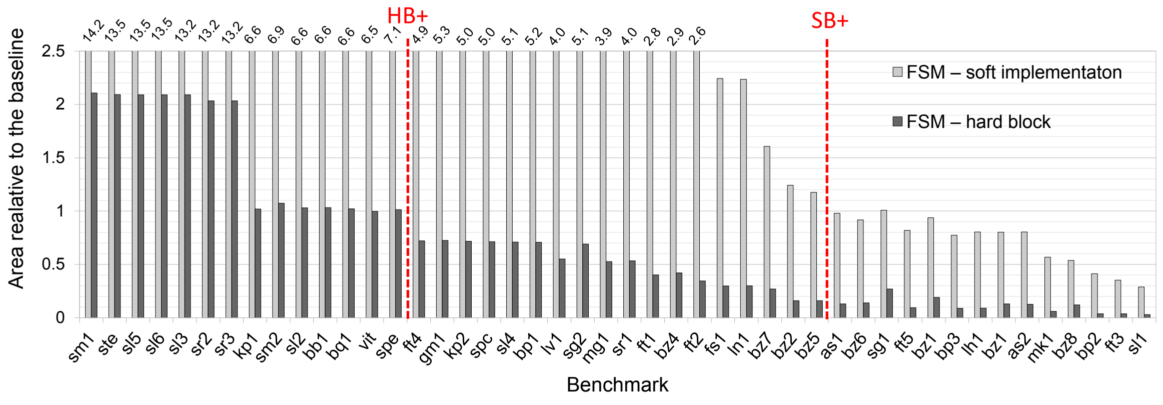


Fig. 7. Comparison between the area improvement of the soft vs hard specialized FSM architecture relative to the baseline.

the number of states in the branch-free paths increases, there is also an increase in area saving.

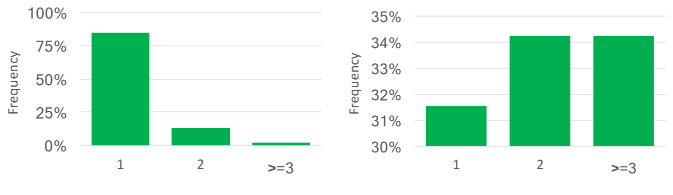
The two vertical dashed lines marked by *HB+* and *SB+* in Figure 7 indicate the FSM size after which mapping the FSM to the specialized FSM architecture is beneficial for the hard block and soft implementation respectively. *HB+* indicate that for FSMs with more than 10 states, the FSM hard block is beneficial. In the extreme cases where the FSM only has a few states, less than 10, the number of states on the branch-free paths and the number of states to be stored in memory are so limited that it does not justify the area overhead of using the FSM hard block with a memory depth of 128. A simple predictor based on the FSM size could be used during the synthesis to decide whether the FSM should be mapped to the proposed FSM hard block or should be implemented using the soft logic on FPGAs. For soft implementation, *SB+* indicates that for FSMs with more than 47 states, the FSM soft IP core is beneficial. The existing FPGA block RAMs provided in VTR FPGA models support up to 1024x32 bit memory which is much larger than the required memory unit for the FSM implementation. Additionally, block RAMs provide higher flexibility by allowing different memory configurations. Therefore, the overhead of using block RAMs compared to our hard blocks makes them a less efficient option. However, as can be seen in Figure 7, for the FSMs with large number of states, our proposed soft IP core can still offer up to 3× reduced area compared to the pure LUT-based baseline.

Similar to the area improvement, for FSMs with more than 10 states, the FSM hard block reduces the critical path delay of the next state calculation by an average of 45%.

V. Hand-Coded vs HLS Generated FSMs

So far, we have primarily focused on exploring the properties of HLS generated FSMs and exploiting these properties in our FSM block architecture to reduce the area of the next state calculations. In this section, we extend our discussion to the properties of FSMs hand-coded by hardware designers to evaluate the potential benefits from our proposed architecture.

First, we need to understand how hand-coded FSMs differ from HLS generated FSMs. We analyzed FSMs extracted from 21 hand-coded VTR benchmarks against the HLS generated benchmarks in Table I and recorded the number of possible next states for each state (Figure 8), as well as the type of inputs (e.g. module port input, datapath input) that each state transition depends on (Figure 9).



(a) HLS generated FSMs (b) Hand-coded FSMs

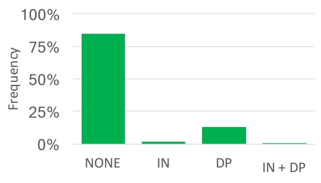
Fig. 8. Number of next possible reachable states per state.

As can be seen in Figure 8, over 80% of the total states in HLS generated FSMs only have a single next state in which the state transition does not depend on any input, whereas only 32% of the total states for hand-coded FSMs have a single next state. Additionally, almost 60% of the total state transitions in hand-coded FSMs depend on datapath input. This difference may be due to hardware designers partitioning FSMs into main FSMs and counters. Since these counters mostly reside in the datapath, we see a lower number of input-independent state transitions for hand-coded FSMs.

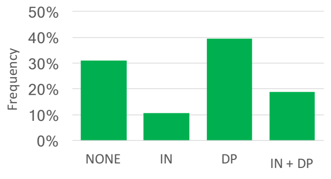
As part of our future work, we plan to develop an algorithm that detects and extracts explicit counters that may be present in hand-coded RTL designs. These extracted counters, along with the hand-coded FSM, can then be mapped to our proposed FSM block. By doing so, implementation of hand-coded FSMs can also benefit from our proposed architecture.

VI. Related Work

Related work has focused on FSM area reduction. Tiwari and Tomko mapped FSMs into SEMBs (synchronous embedded memory block) in FPGAs to reduce power consumption, area and routing overhead compared to flip-flop based implementations [29]. Kolopienczyk et al. use RAM-based implementations for Moore FSMs to reduce the number of LUTs by using additional variables to replace part of the



(a) HLS generated FSMs



(b) Hand-coded FSMs

Fig. 9. Input dependency of state transitions. IN is module port input and DP is datapath input.

logical conditions [30]. Sklyarov proposed a technique for FSM design with modifiable behavior requiring a limited amount of FPGA resources [31]. Cooke et al. [32] looked at overlay architectures for FSM implementation using memory units with a focus on improving compilation time.

None of the above works looked at the specific properties of FSMs generated by HLS tools that helped us reduce memory size and efficiently handle FSMs with a large number of states. This is important, as according to Senhadji-Navarro et al., the number of resources used in block-RAM based FSM implementations increases exponentially with increase number of inputs and state encoding bits of an FSM [33].

Recent work has examined programmable FSM units in network engines for packet processing and pattern matching [34], [35]. These works are based on the (B)FSM [36] concept which relies on the state transition rules to perform the next state calculation. This enables them to limit the memory growth by the number of transition rules. We target the similar problem of optimizing the storage size, however, they target a different class of applications. As such, our proposed FSM block can offer even more memory savings by exploiting the properties of the HLS generated FSMs.

VII. Conclusion

In this work, we analyzed the control-unit portion of RTL designs (modelled by FSMs) that are generated by HLS tools. HLS generated FSMs can account for a large fraction of the total design area in applications where a large number of states and control signals are required to control the datapath. We show that these FSMs demonstrate common properties and propose a novel mix-grained architecture that exploits these properties to improve the total area for implementing the next state calculation logic in FSMs. We introduce a new state assignment technique that enables FSMs to better map to our proposed architecture. We evaluate our proposed architecture on a group of RTL designs generated by a commercial HLS tool. Finally, we show that our proposed architecture can reduce the area of the FSM next state generation logic by more than 3× compared to a LUT-based FSM implementation without impacting performance.

Acknowledgment

The authors would like to thank the reviewers for their insightful feedback and Tayler Hetherington for his invaluable help throughout this research. They would also like to thank Dave Evans, Lotus Herrick Fenn, and Yulong Zheng for constructing the HLS datacenter benchmarks. This research was funded by an NSERC Discovery Grant.

References

[1] Xilinx. (2015, Oct.) Qualcomm and Xilinx Collaborate to Deliver Industry-Leading Heterogeneous Computing Solutions for Data Centers with New Levels of Efficiency and Performance.
[2] Xilinx. (2016, Apr.) Xilinx and IBM to Enable FPGA-Based Acceleration within SuperVessel OpenPOWER Development Cloud.
[3] Xilinx-Xcell Daily Blog. (2016, Oct.) Baidu Adopts Xilinx Kintex UltraScale FPGAs to Accelerate ML Applications in the Data Center.
[4] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, 2014, pp. 13–24.
[5] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *MICRO*, 2016, pp. 1–13.
[6] Amazon. (2017) Amazon EC2 F1 Instances.
[7] Intel. (2015, Dec.) Intel Completes Acquisition of Altera.
[8] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *TCAD*, vol. 26, no. 2, pp. 203–215, 2007.
[9] M. Langhammer and B. Pasca, "Floating-point DSP block architecture for FPGAs," in *FPGA*, 2015, pp. 117–125.
[10] J. Cong *et al.*, "High-level synthesis for FPGAs: From prototyping to deployment," *TCAD*, vol. 30, no. 4, pp. 473–491, 2011.
[11] C. Menn, O. Bringmann, and W. Rosenstiel, "Controller estimation for FPGA target architectures during high-level synthesis," in *ISSS*, 2002.
[12] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE TPDS*, vol. 25, no. 12, pp. 3088–3098, 2014.
[13] I. Garcia-Vargas *et al.*, "ROM-based finite state machine implementation in low cost FPGAs," in *IEEE ISIE*, 2007, pp. 2342–2347.
[14] S. Assadikhomami, "A mixed-grained architecture for improving HLS-generated controllers on FPGAs," MASc thesis, University of British Columbia, Vancouver, Canada, Jun. 2017.
[15] SPEC, "CPU 2006."
[16] B. Reagen *et al.*, "Machsuite: Benchmarks for accelerator design and customized architectures," in *IISWC*, 2014.
[17] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.
[18] G. De Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.
[19] Apache Software Foundation, "LuCy."
[20] Hwaci, "SQLite."
[21] C. Wolf, "Yosys open synthesis suite."
[22] E. Sentovich *et al.*, "SIS: A system for sequential circuit synthesis," EECS Department, University of California, Berkeley, Tech. Rep., 1992.
[23] J. Luu *et al.*, "Vtr 7.0: Next generation architecture and CAD system for FPGAs," *ACM TRETS*, vol. 7, no. 2, pp. 6:1–6:30, 2014.
[24] ArmDeveloper, "Artisan Memory Compiler."
[25] Synopsys, "Design Compiler."
[26] G. Zgheib *et al.*, "Revisiting and-inverter cones," in *FPGA*, 2014.
[27] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Twenty-five Years of EDA*, 1988.
[28] C. Lo and P. Chow, "Model-based optimization of high level synthesis directives," in *FPL*, 2016, pp. 1–10.
[29] A. Tiwari and K. A. Tomko, "Saving power by mapping finite-state machines into embedded memory blocks in FPGAs," in *DATE*, 2004.
[30] M. Kolopienczyk, A. Barkalov, and L. Titarenko, "Hardware reduction for RAM-based Moore FSMs," in *HSI*, 2014, pp. 255–260.
[31] V. Sklyarov, "Reconfigurable models of finite state machines and their implementation in FPGAs," *JSA*, vol. 47, no. 14, pp. 1043–1064, 2002.
[32] P. Cooke, L. Hao, and G. Stitt, "Finite-state-machine overlay architectures for fast FPGA compilation and application portability," *ACM TECS*, vol. 14, no. 3, pp. 54:1–54:25, 2015.
[33] R. Senhadji-Navarro, I. Garcia-Vargas, and J. L. Guisado, "Performance evaluation of RAM-based implementation of finite state machines in FPGAs," in *ICECS*, 2012, pp. 225–228.
[34] K. Septinus, P. Pirsch, H. Blume, and U. Mayer, "A fully programmable FSM-based processing engine for Gigabytes/s header parsing," in *IEEE SAMOS*, 2010, pp. 45–54.
[35] F. Abel, C. Hagleitner, and F. Verplanken, "Rx stack accelerator for 10 GbE integrated NIC," in *IEEE HOTI*, 2012, pp. 17–24.
[36] J. Van Lunteren, "High-performance pattern-matching for intrusion detection," in *IEEE INFOCOM*, 2006, pp. 1–13.