Hardware Implementation of Recursive Algorithms

Dmitri Mihhailov
Computer Department,
TUT,
Tallinn, Estonia
d.mihhailov@ttu.ee

Valery Sklyarov
DETI/IEETA,
University of Aveiro,
Aveiro, Portugal
skl@ua.pt

Iouliia Skliarova
DETI/IEETA,
University of Aveiro,
Aveiro, Portugal
iouliia@ua.pt

Alexander Sudnitson
Computer Department,
TUT,
Tallinn, Estonia
alsu@cc.ttu.ee

***Abstract*—The paper presents new results in the hardware implementation and optimization of recursive sequential and parallel algorithms using the known and a new model of a hierarchical finite state machine. Applicability and advantages of the proposed methods are confirmed through numerous examples of the designed hardware circuits that have been analyzed and compared. The results of experiments and FPGA-based prototyping demonstrate clearly that the proposed innovations enable the required hardware resources to be decreased achieving at the same time better performance of recursive sorting algorithms compared to known implementations both in hardware and in software.**

I. Introduction

Recursion is a powerful problem-solving technique [1] that may be applied to problems that can be decomposed into smaller sub-problems that are of exactly the same form as the original problem. Such technique is not always appropriate, particularly when an efficient iterative solution exists. This is primarily due to the large number of states that are accumulated during deep recursive calls. Additionally, a function call incurs a bookkeeping overhead in most high-level programming languages. Recursive functions magnify this overhead because a single initial call to a function can generate a large number of recursive invocations of the function. It has been shown [2] however that recursion can be implemented much more efficiently in hardware. This is because any activation of a recursive sub-sequence of operations can be combined with the execution of the operations that are required by the respective algorithm. The same is true when any recursive sub-sequence is being terminated, i.e. when control has to be returned to the point following the last recursive call and an operation of the executing algorithm that follows the last recursive call has to be activated. The number of states required for the execution of recursion in hardware can be further reduced compared with software by applying a number of methods, such as those reviewed in [3]. Besides, such states are accumulated on stacks that are typically implemented on built-in memory blocks, which are relatively cheap. The results obtained for some known methods (see, for example, [2,4,5]) for implementing recursive calls have shown that many hardware circuits are faster than software programs executing on general-purpose computers.

Recursive algorithms are employed most often for various kinds of binary search [1, 2]. Let us consider an example of using a binary tree for sorting data [1]. Suppose that the nodes of the tree contain three fields: a pointer to the left child node; a pointer to the right child node; and a value (*e.g.* an integer or a pointer to a string). The nodes are maintained so that at any node, the left sub-tree only contains values that are less than the value at the node, and the right sub-tree contains only values that are greater. In order to build such a tree and to sort data, standard techniques can be applied (based on forward and backward propagation steps that are exactly the same for each node). Thus, a recursive procedure is very helpful. Sorting of this type will be considered in the paper as a case study to demonstrate the proposed innovations.

A brief summary of what is new and distinctive in this paper is given below:

- HFSM with implicit modules (sub-section II, B);
- Improved methods for sequential recursive sorting over tree-based data structures (sub-section III, A);
- Parallel recursive sorting algorithms over tree-based data structures (sub-section III, A);
- Prototyping in FPGA, experiments, profound analysis and comparison of alternative and competing techniques clearly demonstrating advantages of the proposed model and methods (sub-section III, B).

II. Hierarchical Finite State Machines

We will base the approach considered here on hierarchical finite state machines (HFSM) [2] with some useful modifications and improvements..

A. HFSM with explicit modules

The main feature of any recursive algorithm is the capability to activate (to call) itself. There are a variety of synthesizable specifications for recursive algorithms and we will use hierarchical graph-schemes (HGS) [2]. A HGS can easily be converted to a HFSM and then formally coded in a

hardware description language (VHDL will be used). The coding is done using the VHDL templates proposed in [2], which are easily customizable for the given set of HGSs. The resulting VHDL code is synthesizable and permits the hardware to be designed in commercially available CAD systems, such as the Xilinx ISE. Let us briefly demonstrate all the steps mentioned above applied to the synthesis of a simple recursive circuit that outputs data from a binary tree for data sort described in section I. Suppose, that the recursive function *treesort* is written in the C++ language as follows:

```
template<class T> void treesort(treenode<T> *node) { // <a0>
  if (node != 0)                  // if the node exists   (X1)
  { treesort (node->lnode); // Sort left sub-tree (Z) <a1>
    // Display value after the hierarchical return
    cout << "value - " << node->val << endl; <a2>
    treesort (node->rnode); // Now sort right sub-tree (Z) <a3>
  }                                                    } // <a4>
```

where the *treenode* is a structure, described something like the following:

```
template <class T> struct treenode {
  T val;                          // Value of an item of type T
  int count;                      // Number of items with value val
  treenode <T> *lnode; // Pointer to left node (sub-tree)
  treenode<T> *rnode;  // Pointer to right node (sub-tree)
};
```

Figure 1, a) depicts the HGS, which is designated by Z, built at the first step from the function *treesort*. There are two recursive module calls (HGS calls) in Figure 1, a).

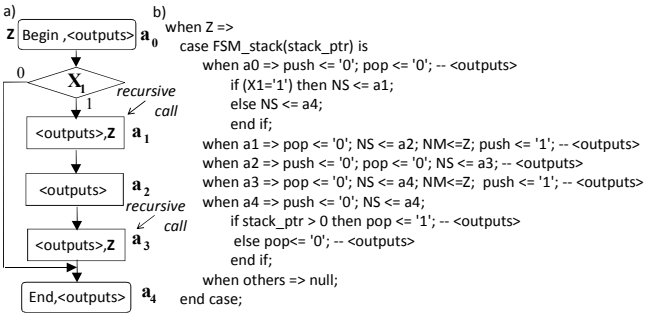

Figure 1. C function *treesort* described in a HGS (a) and in VHDL (b).

At the second step, the HGS has to be converted to a HFSM and this is done in two sub-steps: marking the HGS rectangular nodes with labels $a_0,a_1,\ldots,a_4$ (considered further as HFSM states) using the rules [2] (Figure 1, a); and customizing the HFSM template [2] with state transitions extracted from the HGS. The labels $a_0,a_1,\ldots,a_4$ as well as the recursive call Z and the condition X_1 are also shown in the C++ code. This is done to make clearer a relationship between the C++ code and Figure 1, a). General HFSM template is shown in Figure 2 and includes two processes describing: 1) reusable stacks for modules (*M_stack*) and states (*FSM_stack*); 2) a structure of combinational circuits allowing transitions at the level of modules and states to be executed (see example in Figure 1, b). Here *NS*/*NM* is the next state/the next module; *stack_ptr* is a stack pointer common to both *M_stack* and *FSM_stack*; signals *push* and *pop* increment and decrement the *stack_ptr*. All the details can be found in [2]. We will call the known model *HFSM with explicit modules*. It has the following distinctive features. There are two stack memories with $\lceil \log_2 Q \rceil$ bits for modules and $\lceil \log_2 R \rceil$ bits for states (Q is the number of modules and R is the maximum number of states in a module). States in different modules can be assigned the same labels (the same codes).

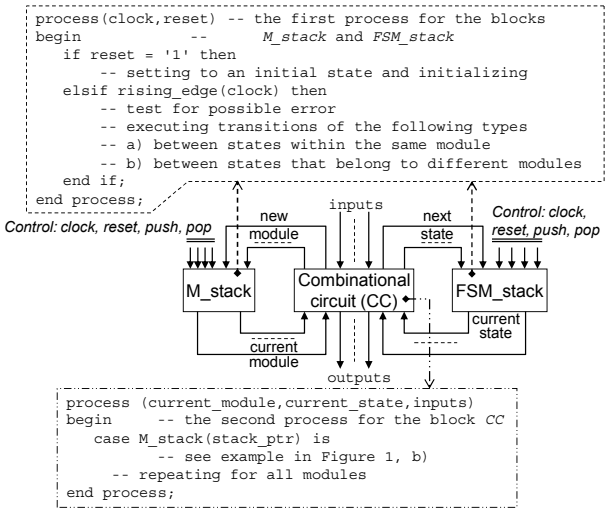

Figure 2. HFSM with explicit modules.

B. HFSM with implicit modules

Figure 3 depicts a new proposed HFSM model called *HFSM with implicit modules*. The HFSM in Figure 3 behaves like an ordinary FSM and a single stack of states is used just for returns from called modules.

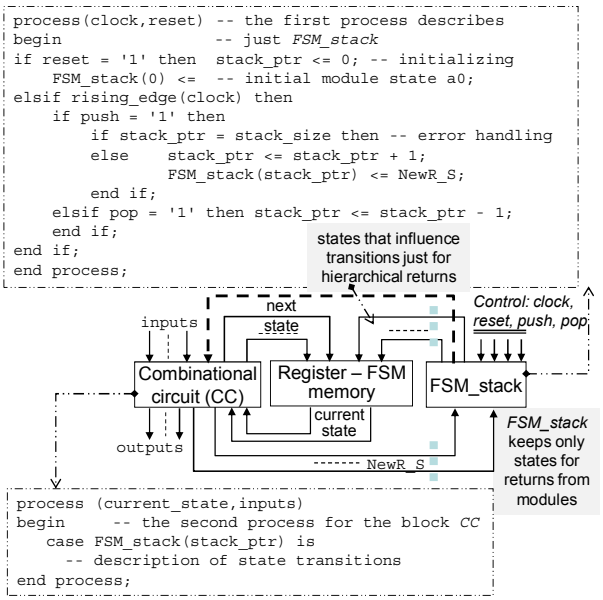

Figure 3. HFSM with implicit modules.

In this case states in different modules have to be assigned different labels (different codes). The *FSM_stack* in Figure 3 is needed just to know which state has to be the target of the transition when a called module is terminated. Note that the return from a called module (such as the return in the state a_4 in Figure 1, a) and the transition from the relevant state of the calling module (such as the transition from the state a_1 or a_3 in Figure 1, a) are executed within the same clock cycle. In any module all the necessary state transitions are realized through the register, much like it is done in a conventional FSM. Suppose that a new module Z has to be called in the state a_1 (see Figure 1, a). In this case the following operations are executed at the same time: 1) the state a_2 is saved in the *FSM_stack*; 2) the *stack_ptr* is incremented; and 3) the transition from a_1 to a_0 (the first state of Z) is performed in the register (see Figure 3). When the called module Z is terminated, the stack pointer is decremented and the *stack_ptr* points to the register of the stack with the state a_2 that has to be selected for the next state transition. There exist *two modes* of returns. In the first mode there are no conditional transitions from the state like a_1, a_3 in Figure 1, a). Thus, we can explicitly save in the stack the target state (such as a_2) for transition after return from the called module. This mode has to be used for our example in Figure 1, a). In the second (more complicated) mode there are conditional transitions from the states where we have to call other modules and conditions for such transitions can be influenced by the called modules. In this case the method based on the use of a special return flag (described in [2] with all the necessary details) can be applied directly. It is important to note that no delay is introduced. This is achieved through the technique described below. Let us consider pairs "$a_m z_n$", where a_m is the state of a calling module from which the module z_n is called (thus, z_n is a called module). All such pairs are unique. When z_n is called, the following steps are executed: 1) the code of a_m is saved in the top register of the stack; 2) the code of the first state of z_n is saved in the register (see Figure 3); 3) the *stack_ptr* is incremented. Thus, the stack keeps track of returns beginning with the currently active module. When any module is terminated, the following steps are performed: 1) the transition from the state indicated by *stack_ptr*-1 is executed; 2) the *stack_ptr* is decremented. Finally, no extra delay is introduced in the HFSM compared to a conventional FSM.

Note that module calls (such as Z) might appear in different states (*e.g.* a_1, a_3 in Figure 1, a), and, thus, the returns might also be to different states (*e.g.* a_2, a_4 in Figure 1, a) from the same called module. That is why the returned state (such as a_2 or a_4) has to be chosen in the calling modules (but not in the called modules). The called modules might change conditions that influence transitions from the states of calling modules. Such changes are taken into account with the aid of the methods proposed in [2]. The stack is needed just to know which state has to be the target of the transition when a called module is terminated. The number of states is increased over that of the previous model (Section II, A). However, the number of stacks and the size of the stack registers are reduced. Another feature of the new model is that it is directly applicable to all known optimization techniques that have been proposed for conventional FSMs.

III. FPGA-BASED IMPLEMENTATIONS AND EXPERIMENTS

A. Methods

The known method [2] (let us call it *S1*) was chosen as a base for comparison. Let us consider the basic ideas making it possible the method [2] to be improved. The embedded dual-port memory blocks (available for different FPGAs) permit two memory cells to be accessed simultaneously. Each cell holds *data+LA+RA* for the left (LA) and for the right (RA) nodes. Initially the input buffer register is loaded with *data+LA+RA* for the root of the sorting tree. Firstly, we examine left sub-trees. If a left sub-tree (node) exists then it is checked again to determine whether the left sub-tree also has either left or right sub-trees (nodes). If there is no sub-tree from the left node, then the value of the left node is the leftmost data value and can be output as the smallest. In the last case the node in the input buffer register holds the second smallest value and the relevant data value is sent to the output. Secondly, we perform similar operations for right nodes. The dual-port RAM makes possible LA and RA (for each cell in the dual-port RAM) to be examined independently. Finally, compared with [2], for some sub-trees of the tree we are able to check more than one node (*e.g.* a left node and a right node or a left node and a left node of the first left node) at the same time, which reduces the required processing time. Let us call the improved method *S2*.

Note that the known method [2] is sequential and does not permit parallel processing of different branches of the tree. We propose to traverse the left and the right sub-trees of the tree nodes in parallel using the method *S2* (let us call this method *S3*). It is achieved with the aid of the following technique. There are two simultaneously functioning HFSMs that are a master and a slave. The master HFSM: builds the tree; outputs the left sub-tree; and activates the slave HFSM when necessary. The slave HFSM outputs the right sub-tree. By adding a simple counter to the root that counts the number of left and right nodes during the construction of the tree, we can easily calculate the addresses for the sorted data in an output memory. Thus, processing of both sub-trees can be done simultaneously. Obviously, more parallel branches can be introduced using cascade structures. Intuitively we can guess that the result would depend considerably on the balance between the left and right sub-trees of the root. We would like to eliminate such dependency, and this can be achieved using the following technique. The first HFSM implements the algorithm *S2* and when the HFSM detects that for the current node the left and the right sub-trees are well balanced, the second HFSM is activated. In this approach each node of the tree includes an additional field indicating the number of child nodes on the left and on the right (such fields can easily be filled in during the construction of the tree). Let us call this method *S4*.

B. Experiments

The synthesis and implementation of the circuits from the specification in VHDL were done in Xilinx ISE 11 for FPGA Spartan3E-1200E-FG320 of Xilinx.

A random-number generator produces 2^{11} items of data with a length of 14 bits (i.e. values in an interval between 0 and 16383). Values greater than 9999 are removed leaving 1200-1300 items available for further processing. These items are sorted using the methods *S1*-*S4* and implementations in FPGA based on the known HFSM with explicit modules ($S1_e$-$S4_e$) and the new HFSM with implicit modules ($S1_i$-$S4_i$). The results are presented in Table I, where the *Data* column indicates the number of data items that were sorted, the column *Left/Right* shows the number of nodes in the left and right sub-trees from the root. The other columns indicate time of sorting in *ns* per data item. Table II presents the maximum achievable clock frequency (*F*) in MHz and FPGA resources (the number of slices - *Slices*, the number of LUTs – *LUTs*, the number of block RAMs - *B*) needed for two different implementations: based on HFSM with explicit modules ($HFSM_e$) and HFSM with implicit modules ($HFSM_i$). In four lines *S1*-*S4* of Table II not marked with *b* the stacks are built from LUTs. In two bottom lines of Table II marked with *b* the stacks are built from block RAMs. The results for $S1_i$-$S4_i$ are slightly better than $S1_e$-$S4_e$ because the new HFSM model achieves higher value of *F* (see Table II). Only the column $S1_e$ (from $S1_e$-$S4_e$) is shown in Table I. Ratios SI_e/SI_i (I=2,3,4) are similar to $S1_e/S1_i$ (see Table I), i.e. for all values I (1-4) the new model of HFSM gives slightly better results just because of higher achievable frequency *F* (see Table II).

Besides, the method *S1* [2] was described in C++ and implemented in software (other algorithms *S2*-*S4* are hardware-oriented and their advantages have appeared just in hardware). The same data (randomly generated) were used for the software implementations. The results were produced on HP EliteBook 2730p (Intel Core 2 Duo CPU, 1.87 GHz) computer and shown in column *SW* of Table I (also in *ns* per data item).

TABLE I. THE RESULTS OF EXPERIMENTS

Data	*Left/Right*	$S1_e$	$S1_i$	$S2_i$	$S3_i$	$S4_i$	*SW*
1286	1/1284	39.5	36.6	31.3	48.1	28.2	147.2
1278	53/1224	39.5	36.6	31.4	46.1	27.9	151.1
1248	143/1104	39.5	36.6	31.1	42.6	28.0	153.6
1211	185/1025	39.5	36.6	31.2	40.8	28.1	167.3
1216	266/949	39.5	36.6	31.3	37.6	28.0	154.1
1248	332/915	39.5	36.6	31.3	35.3	27.6	148.6
1203	460/742	39.5	36.6	31.2	29.7	26,3	155.6
1228	528/699	39.5	36.6	31.3	27.4	25.1	151.5
1212	556/655	39.5	36.6	31.0	26.0	24.7	148.5
1230	623/606	39.5	36.6	31.6	24.3	23.1	155.2
1305	742/562	44.7	41.5	30.9	26.1	24.7	147.8
1259	822/436	39.5	36.6	31.1	28.5	28.0	149.1
1230	799/430	39.5	36.6	31.1	28.4	27.8	150.0
1304	849/454	39.5	36.6	31.0	28.4	27.9	148.7
1276	963/312	39.5	36.6	31.3	31.4	28.1	148.0
1225	958/266	39.5	36.6	31.2	32.2	27.7	151.0
1225	986/238	39.5	36.6	31.3	32.9	28.0	148.8
1199	1051/147	39.5	36.6	31.0	35.0	27.9	157.6
1309	1288/20	39.5	36.6	31.6	38.1	28.6	149.6
1204	1203/0	39.5	36.6	31.5	38.5	27.8	157.0

From the results of the experiments it is clearly observable that the proposed in sub-section III,B HFSM model permits the hardware resources to be reduced and it supports faster clock frequency. This is achieved because the stacks (needed to support modularity and recursion) were simplified as much as possible. The improved methods of data sort (*S2*-*S4*) are faster than the known method *S1*. Although the method *S3* of parallel sorting gives the worst results for some lines of Table I, the improved version *S4* of this method (see sub-section III, A) is undoubtedly the fastest. Both methods *S3* and *S4* were implemented with two parallel communicating HFSMs.

TABLE II. IMPLEMENTATION DETAILS

Method	Known model ($HFSM_e$)				New model ($HFSM_i$)			
	F	Slices	LUTs	B	F	Slices	LUTs	B
S1	101.36	714	1391	5	109.24	365	708	5
S2	82.84	790	1548	6	89.27	435	855	6
S3	102.6	1115	2203	8	103,88	701	1397	8
S4	101.54	1256	2497	8	101.71	728	1346	8
S1b	70.44	149	277	7	97.33	131	241	6
S2b	59.91	233	449	8	69.27	197	379	7

IV. CONCLUSION

An analysis of experimental results clearly demonstrates the following: 1) the proposed new model of HFSM consumes almost two times less hardware resources and it is slightly faster than the known HFSM model; 2) the hardware implementations are faster than software implementations for all the experiments (in the best case in about 6 times) even though the clock frequencies of the FPGA and the PC differ significantly (the clock frequency of the PC is about 20 times faster than that of the FPGA); 3) the proposed methods of data sort are faster than the known methods in about 1.5 times for the best case (*S4*); 4) although the results of *S3* depend considerably on the balance between the left and right sub-trees of the root, the improved version *S4* of *S3* for parallel sorting eliminates such dependency and gives the best results.

ACKNOWLEDGMENT

This research was supported by the European Union through the European Regional Development Fund.

REFERENCES

[1] F.M. Carrano, Data Abstraction and Problem Solving with C++, The Benjamin/Cumming Publishing Company, Inc., 2006.

[2] V. Sklyarov, “FPGA-based implementation of recursive algorithms,” Microprocessors and Microsystems. Special Issue on FPGAs: Applications and Designs, vol. 28/5-6, pp. 197–211, 2004.

[3] I. Skliarova, V. Sklyarov, "Recursion in Reconfigurable Computing: a Survey of Implementation Approaches", Proc. 19th Int. Conference on Field Programmable Logic and Applications – FPL’2009, Prague, Czech Republic, 2009, pp. 224-229.

[4] T. Maruyama, M. Takagi, T. Hoshino, “Hardware implementation techniques for recursive calls and loops”, Proc. 9th Int. Workshop on Field-Programmable Logic and Applications - FPL’99, Glasgow, UK, 1999, pp. 450-455.

[5] S. Ninos, A. Dollas, “Modeling recursion data structures for FPGA-based implementation”, Proc.18th Int. Conference on Field Programmable Logic and Applications – FPL’08, Heidelberg, Germany, 2008, pp. 11-16.