Synthesis and FPGA-based Implementation of Hierarchical Finite State Machines from Specification in Handel-C

VALERY SKLYAROV
Electronics and Telecommunications Department
Aveiro University, IEETA
Campus Universitário de Santiago, 3810-193 Aveiro
PORTUGAL

Abstract: - The paper suggests a technique, which permits to describe modular, hierarchical and parallel algorithms in Handel-C. This opportunity has been provided by generating the required control sequences with the aid of a hierarchical finite state machine. The proposed specification in Handel-C is synthesizable and it can be translated (for example, in DK2 environment of Celoxica) to EDIF format. The latter can be converted to a bit-stream for commercially available FPGAs. An example of sorting procedure was described in detail and implemented in Xilinx Spartan II XC2S200 FPGA available on Celoxica RC100 prototyping board.

Key-Words – hierarchical finite state machines, system-level specification, Handel-C, FPGA

1. Introduction

Hierarchical finite state machines (HFSMs) [1] allow modular, hierarchical and recursive algorithms to be implemented. These facilities are very useful and we will show just a few examples.

It is known that a digital system can be composed of an execution (EU) and a control (CU) unit. EU is responsible for operations over data and for data transfer between registers in order to perform the given algorithms. CU forces the required sequence of operations. Any operation can be elementary and non elementary. The first one can be executed by the relevant circuit of EU. The second one can be composed of elementary and/or non elementary operations. This permits a hierarchical specification to be built. Thus support for modularity and hierarchy is very helpful. Any module can be seen a specification of a non elementary operation. Hierarchy makes possible to construct new modules from elementary operations and existing modules. A significant benefit of such an opportunity is that any complex algorithm can be developed step by step, so that our efforts can be concentrated at each stage on a specified level of abstraction (that is, on a particular module). Each module is usually relatively simple, and can be checked and debugged independently. Modular specification provides support for either top-down or bottom-up design. Top-down design is based on extending given modules by supplying more and more detail. Bottom-up design enables us to use more complicated components that are defined as modules and to organize the design process with the aid of libraries of modules. Any module can be reused. For many practical applications extending or modifying a module does not change the existing specification.

Generally speaking, an opportunity of describing algorithms at different levels of complexity makes possible to concentrate efforts of the designer at the desired level and to abstract from all non essential details. Note that some modules might be virtual [1] that permits to redefine them later if necessary. Besides, for testing some ideas it is not required to implement all modules and we can deal with incompletely specified functionality, which is very helpful for debugging purposes [1].

For some applications a module might call itself. For example, many problems can be solved through traversing binary trees [2,3]. The nodes of the tree are maintained so that at any node N, the left sub-tree contains only values belonging to a range A, and the right sub-tree contains only values belonging to a range B ($A \cap B = \varnothing$). The value written in the node N makes possible the ranges A and B to be calculated. Possible extra fields in the node N keep additional information, for example, the number of occurrences of the value associated with the respective node N. It is known, for example, that such a tree can be constructed and used for sorting various types of data [3]. In order to build this tree for a given set of values, we have to find the appropriate place in the current tree for incoming items. In order to sort the data, we can apply a special technique [3] using forward and backtracking propagation steps that are exactly the same for each node. Thus a recursive procedure is very efficient and support for this is very helpful. Sorting of this type will be considered in the paper as a working example and all the required additional details can be found in [4,5].

Note that existing specification tools do not provide direct support for majority of the considered above features. On the other hand these features can be realized if the control sequence is generated by an HFSM and it was shown in [4], where VHDL-based implementation of modular, hierarchical and recursive algorithms has been proposed and tested in Spartan IIE XC2S300E/XC2S400E FPGAs on examples of recursive sorting and data compression. This paper suggests the method of description of modular, hierarchical, recursive and parallel algorithms in Handel-C, which is a system level specification language.

The remainder of this paper is organized in six sections. Section 2 shows how to describe modular and hierarchical algorithms. Section 3 explains various opportunities allowing modules to be executed in parallel. Section 4 considers recursive algorithms. Section 5 describes a Handel-C project for sorting algorithms, which provides support for modularity, hierarchy and recursive module invocations. The conclusion is in section 7.

2. Modular and hierarchical algorithms

It is known [1] that CU algorithms can be constructed from modules and described hierarchically with the aid of the language that is called hierarchical graph-schemes (HGS). Fig. 1,a demonstrates an example of a binary tree, which keeps a set of unsigned integers. Fig. 1,b shows how to describe a simple algorithm (HGS) for finding an integer with minimum value. Here the logic condition x_1 tests if the node has a left sub-tree, the micro-operation y_1 selects the left sub-tree through its number (see *Italic* digits in fig. 1,a) and the micro-operation y_2 copies the value from the selected node to the result. Let us designate the HGS in fig. 1,b as z_1. Fig. 1,c depicts a fragment of an HGS of a higher hierarchical level, which invokes the HGS z_1 in the rectangular node a_m. This enables us more complicated HGSs to be constructed from elementary operations and other HGSs designated as modules $z_0, z_1, z_2, \ldots$.

Any HGS can be described in Handel-C using a finite state machine (FSM) notation [1,4] such as the following (see also fig. 1,b):

```
do {                      // module z1 in fig. 1,b
  CS = NS;
  switch(CS)
  {  case 0:              // state a0
       if (RAM[reg][2] != 31)             // x1
         par { reg=RAM[reg][2]; NS=1; }  // y1
       else NS=2;
     break;
     case 1:              // state a1
       if (RAM[reg][2] != 31)
         par { reg=RAM[reg][2]; NS=1; }  // y1
       else NS=2;
     break;
     case 2:              // state a2
       result=RAM[reg][0];               // y2
  }
   }
while(CS != 2);
```

Here CS/NS is a variable, which keeps current/next FSM state, the labels a_0, a_1, a_2 (see fig. 1,b) are considered to be FSM states, *reg* is the RAM address register, which was initially set to 0. The tree shown in fig. 1,a is stored in a RAM block as follows:

```
RAM[0][0]=5;  RAM[0][1]=1;  RAM[0][2]=2;
RAM[1][0]=4;  RAM[1][1]=3;  RAM[1][2]=31;
RAM[2][0]=9;  RAM[2][1]=4;  RAM[2][2]=9;
RAM[3][0]=3;  RAM[3][1]=5;  RAM[3][2]=31;
RAM[4][0]=6;  RAM[4][1]=31; RAM[4][2] = 8;
RAM[5][0]=1;  RAM[5][1]=31; RAM[5][2] = 6;
RAM[6][0]=2;  RAM[6][1]=31; RAM[6][2]=31;
RAM[7][0]=7;  RAM[7][1]=31; RAM[7][2]=31;
RAM[8][0]=8;  RAM[8][1]=7;  RAM[8][2]=31;
RAM[9][0]=10; RAM[9][1]=31; RAM[9][2]=31;
```

The value 31 indicates an absence of the respective (either left or right) sub-tree. Note that for simplicity the RAM is declared as:

```
unsigned int 5 RAM[32][3];
```

i.e. it keeps 5-bit unsigned integers. It enables us to sort values from 0 to 30 (because the value 31 is used as a flag). However this declaration can easily be changed in order to hold data with the required size. Here RAM[i][0] is the saved value (unsigned integer), RAM[i][1] is a pointer to the left sub-tree, RAM[i][2] is a pointer to the right sub-tree and i = 0,1,…,9.

Note that the considered above Handel-C code does not support any feature mentioned in the introduction. We want to make this code reusable in such a way that the module z_1 might be called from any other HGS much like we can invoke any elementary operation.

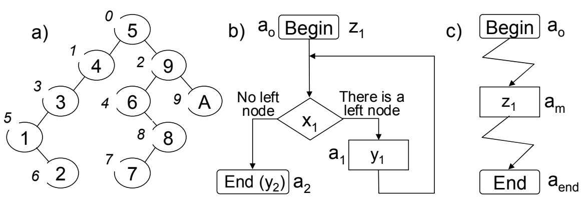

Fig. 1. Examples of HGSs (b and c) for operations over a binary tree (a)

It is known that hierarchical invocations of different modules (HGSs) $z_0,z_1,z_2,\ldots$(where z_0 is a top-level HGS) can be provided by HFSM [1], which contains two stack memories for modules (*M_stack*) and states (*FSM_stack*) (see [4] for details). A stack pointer is the same for the both stacks. Top-level registers of the stacks keep the codes for an active module and an active state. Any hierarchical call causes the stack pointer to be incremented and new top-level stack registers to be appropriately set. As a result these registers enable the module specified by the relevant algorithm to be selected and the required functionality of the module to be established. The code in the register of the *M_stack* points to the active module and codes in the register of the *FSM_stack* correspond to different states allowed for providing the module functionality. Thus an HGS for the new module can now be executed. Any hierarchical return forces the stack pointer decrement. As a result the control flow is returned to the module, from which the terminating module was called.

Let us consider an example. Fig. 2,a depicts an HGS z_2, which permits an integer with the maximum value to be found. Fig. 2,b shows how a new HGS z_0 sequentially determines the minimum and the maximum values to be kept on the tree in fig. 1,a. This might be described in Handel-C as follows:

```
do
{ par  {  module=get_module();
          state=get_state(); }
  switch(module)      // select module
  { case 0:                     // module z0
     switch(state)
     { case 0:  par {             // state a0
                      next_state(1);  reg=0;
                      new_module(1); }  break;
        case 1:  par {            // state a1
                      next_state(2); reg=0;
                      new_module(2); }   break;
       case 2:                    // state a2
               end_module();
   }  break;
     case 1:                      // module z1
      switch(state)
      { case 0:                   // state a0
           if(RAM[reg][2]!=31)
              par {   reg = RAM[reg][2];
                      next_state(1);         }
           else next_state(2);
        break;
        case 1:                   // state a1
           if(RAM[reg][2]!=31)
           par {    reg = RAM[reg][2];
                    next_state(1);           }
           else next_state(2);
        break;
        case 2:  par {            // state a2
                      result[0]=RAM[reg][0];
                      end_module();          }
   }  break;
     case 2:                      // module z2
       switch(state)
       { case 0:                  // state a0
            if (RAM[reg][1]!=31)
              par {   reg = RAM[reg][1];
                      next_state(1);         }
           else next_state(2);
         break;
         case 1:                  // state a1
            if(RAM[reg][1]!=31)
               par {  reg = RAM[reg][1];
                      next_state(1);         }
           else next_state(2);
         break;
         case 2:  par {           // state a2
                      result[1] = RAM[reg][0];
                      end_module();  }
       }
   } done=test_ends();    // z0 is terminated
 }  while(!done);
```

There are two levels of *switch-case* statements in the code. The first level permits an active module to be selected and the second level makes possible the required FSM functionality for the selected module to be established.

The micro operation y_3 in fig. 2 selects the right sub-tree through its number. There are some functions in the Handel-C code above, which look like the following:

```
unsigned int module_size get_module()
{       return M_stack[stack_ptr];         }
unsigned int state_size get_state()
{       return FSM_stack[stack_ptr];     }
void next_state(unsigned int state)
{       FSM_stack[stack_ptr] = state;    }
```

```
void new_module(unsigned int module)
{  if(stack_ptr != (MAX_H_INV-1))
   par {  stack_ptr++;
          M_stack[stack_ptr+1] = module;
          FSM_stack[stack_ptr+1] = 0;    }
}
void end_module(void)
{     if(stack_ptr == 0) ends = TRUE;
      else  stack_ptr--;                 }
unsigned int 1 test_ends()
{     return ends;     }
```

Here *MAX_H_INV-1* is a predefined constant, *stack_ptr* is the stacks (*M_stack* and *FSM_stack*) pointer. The size of returned values (such as *module_size*) is specified through Handel-C *#define* directives.

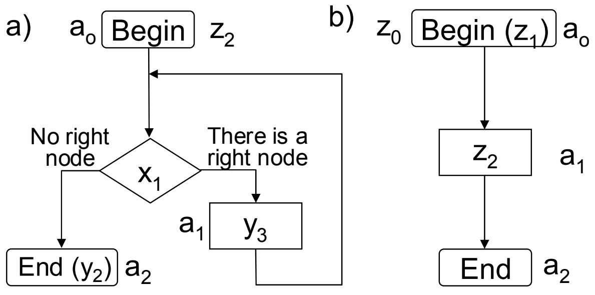

Fig. 2. An example of hierarchical module invocations

3. Parallel algorithms

If two or more modules are activated at the same time they will be executed in parallel [6]. For example, z_1 and z_2 can be invoked in the same rectangular node of an HGS, which enables us to find integers with minimum and maximum values. In this case two modules z_1 and z_2 will be executed in parallel.

Suppose a node a_m contains a subset Z_m of modules. In such situation the transition to the next after a_m node is allowed to be performed only after all modules from Z_m have been terminated. Let q be the maximum number of modules that might be executed in parallel. In this case we have to build q stacks of modules and states. This can be done in Handel-C by allocation of q-element arrays for the objects *M_stack* and *FSM_stack*, duplicating resources that are required for parallel branches and taking into account such additional issues as synchronizations mechanisms [6] and parallel algorithms correctness [7]. For relatively simple applications it is easier to construct autonomous FSMs for each module and to execute them at the same time using Handel-C *par* statements. For example, two concurrent FSMs can be built for the modules z_1 and z_2, which will be run in parallel. The respective example is given in [5]. Note that the resources of Xilinx XC2S200 [8] FPGA for parallel FSMs comparing with sequential FSMs are increased just by 9 slices.

4. Recursive algorithms

An algorithm becomes recursive, when it calls itself. The following Handel-C code shows how to describe an HGS z_3 for recursive data sorting extracted from the given tree (such as that is depicted in fig. 1,a).

```
case 3:                  // module z3
  switch(state)  {
    case 0:              // state a0
      if (reg != 31) next_state(1);
      else next_state(4);
    break;
    case 1:              // state a1
      par {
        next_state(2);
        local_stack[local_sp]= reg;
        local_sp++;
        reg=RAM[reg][2];
        new_module(3); } // recursive call
     break;
     case 2:             // state a2
       par {
         next_state(3);
         output_stack[output_sp]
                         = 000@RAM[reg][0];
         output_sp++;  }
     break;
     case 3:             // state a3
       par  {
         next_state(4);
         local_stack[local_sp]=reg;
         local_sp++;
         reg = RAM[reg][1];
         new_module(3);  } // recursive call
      break;
      case 4:            // state a4
        par  {
          next_state(4);
          end_module();
          if (local_sp>0)  local_sp--;
          reg = local_stack[local_sp-1]; }      }
```

Here *local_stack* and *output_stack* are exactly the same as in [4]; *local_sp* and *output_sp* are local and output stack pointers respectively. All the details of recursive sorting can be found in [4] and we won't discuss them in order to keep the description short. The complete synthesizable Handel-C project and the relevant tutorial are available in [5].

It is important to note that the considered modules (HGSs) can easily be reused. For example, the module z_2 (see the node a_1 in fig. 2,b) can be replaced with the module z_3. This permits data in the node a_1 to be sorted.

5. Handel-C project for sorting algorithm

The detailed description of a recursive sorting algorithm was considered in [4]. This section shows how to specify a similar algorithm in Handel-C using the proposed above technique.

Let us design a circuit, which has to receive and sort unsigned integers from an external source. As soon as a new item is available from the source it has to be included in the sorted sequence. The following module z_4 constructs the tree (such as that is shown in fig. 1,a) from incoming unsigned integers:

```
case 4:          // module z4
  switch(state)  {
   case 0:       // state a0
     if (reg == 31) next_state(1);
     else if (ROM[ROM_address]
                 == RAM[reg][0]) next_state(6);
     else if (ROM[ROM_address]
                 > RAM[reg][0])   next_state(3);
     else next_state(2);
   break;
   case 1:       // state a1
     par {  next_state(7);
            RAM[RAM_w+1][0]
                 = ROM[ROM_address];
            RAM[RAM_w+1][1] = 31;
            RAM[RAM_w+1][2] = 31;
            RAM_w++;
            local_stack[local_sp] = RAM_w+1;
         }  break;
   case 2:       // state a2
     par  {  next_state(4);
              local_stack[local_sp]=reg;
              local_sp++;
              reg = RAM[reg][1];
              new_module(4); // recursive call
         }  break;
    case 3:      // state a3
      par {  next_state(5);
              local_stack[local_sp]= reg;
              local_sp++;
              reg=RAM[reg][2];
              new_module(4); // recursive call
          }  break;
     case 4:     // state a4
      par   {  next_state(7);
              RAM[reg][1]
                    = local_stack[local_sp+1];
            }  break;
   case 5:     // state a5
     par  {  next_state(7);
              RAM[reg][2]
                 = local_stack[local_sp+1];
          }  break;
   case 6:     // state a6
     par  {  next_state(7);
              ROM_address++;
              local_stack[local_sp] = reg;
           }  break;
   case 7:     // state a7
    par  {  next_state(7);
            end_module();
            if (local_sp>0)  local_sp--;
            reg = local_stack[local_sp-1]; }
          }   break;
```

There are three modules in the Handel-C project. The first module z_0 calls the module z_4 for each incoming integer and z_4 incrementally constructs the tree (such as that is shown in fig. 1,a), i.e. for any new integer either a new node is allocated on the tree (if the integer is unique) or a counter for a node is incremented (if the value of the node is the same as the integer). As soon as a new node has been included in the tree, z_0 calls the module z_3 (see section 4). Thus incoming data are kept permanently sorted. Note that two modules z_3 and z_4 are recursive.

6. Experiments

Handel-C projects for the considered above modular, hierarchical and parallel algorithms have been described in Handel-C and debugged in the Celoxica DK2 design suite [9]. After verifying the required functionality in software the relevant circuit was synthesized in the DK2 (for EDIF-based design flow). The resulting file in electronic design interchange format (EDIF) was converted (in the Xilinx ISE 6.2.01 [9]) to a bit-stream for the Xilinx XC2S200-5-FG456 FPGA [8]. Finally the bit-stream was loaded to the FPGA available on the RC100 [9] prototyping board with the aid of the Celoxica FTU2 utility [9] and examined in hardware.

All the considered above circuits have been designed, implemented and tested in the following Handel-C projects (P_1, P_2, P_3):

P_1 - for discovering the minimum and the maximum integers (see the modules z_1 and z_2 in section 2) on the basis of data in the given tree, such as that is shown in fig. 1,a;

P_1 – for sequential and parallel execution of the modules z_1 and z_2 (all these facilities have been provided within the same project P_1 and to test the required module(s) it is necessary to uncomment the relevant sections of the code);

P_2 – for sorting of data receiving from an internal source, which is an FPGA ROM block. Arbitrary data were preliminary saved in the ROM block and then they were read and the tree (like shown in fig. 1,a) was constructed with the aid of the module z_4 (see section 5). Finally the data were extracted from the tree, sorted by the module z_3 and displayed;

P_3 – for sorting of data receiving from a mouse connected to the prototyping board RC100 [9]. The left mouse button was used to get a value of one of the mouse coordinate. This value is considered to be an incoming integer. The right mouse button was used to withdraw all previously received data and to start a new data sequence. Obviously any other source (such as serial interface, keyboard, etc.) can be used.

All the considered projects are available in [5] and can be downloaded and tested. The Web site [5] contains also a set of very useful tutorials that have already been very widely used in educational process. There are also a number of papers describing various methods and tools for the design of FPGA-based digital circuits, including more than 10 student publications and many VHDL and Handel-C based projects.

Note, that in general the number of FPGA slices in case of synthesis from Handel-C is larger then in case of synthesis from VHDL. However, the time required for the design is significantly shorter.

It is very important that the considered modular specification is very flexible and can easily be retargeted for different criteria. For example, a trivial change to the module z_3 (the lines reg=RAM[reg][2]; and reg=RAM[reg][1]; are swapped) makes it possible to alter the order of sorting. Other changes permit finding all data within a given range, etc. The designed Handel-C modules are parameterizable which permits the circuit to be scaled very easy.

The modules considered, which form a hierarchical specification, can be reused for new circuits. This is especially interesting for FPGA-based applications. The basic Handel-C core of the HFSM can be seen as a template for synthesis. The number of modules and the number of states in each module can easily be customized. The respective Handel-C code can be inserted and used in any new project. This permits synthesizable code to be constructed from previously verified fragments, which can be easily modified in future, if required.

7. Conclusion

The paper suggests a novel approach for describing modular, hierarchical and recursive algorithms in Handel-C system-level specification language. The considered technique is very helpful and it permits complex solutions to be specified clearly and implemented on the basis of relatively simple circuits. This technique is used in software and it is rational to study its opportunities in hardware. The practicability of the proposed methods has been demonstrated on numerous examples Modular, hierarchical, parallel and recursive algorithms have been coded in Handel-C, implemented and tested in FPGA of Spartan-II family from Xilinx.

Acknowledgement

This work was partially supported by the grant POSI/43140/CHS/2001.

References

[1] V.Sklyarov, Hierarchical Finite-State Machines and Their Use for Digital Control. IEEE Transactions on VLSI Systems, 1999, Vol. 7, No 2, pp. 222-228.

[2] Frank M.Carrano. Data Abstraction and Problem Solving with C++. The Benjamin/Cumming Publishing Company, Inc., 1995.

[3] Brian W.Kernighan, Dennis M.Ritchie, The C Programming Language, Prentice Hall, 1988.

[4] V.Sklyarov, FPGA-based Implementation of Recursive Algorithms, Elsevier Journal Microprocessors and Microsystems, April, 2004.

[5] http://webct.ua.pt, the discipline Computação Reconfigurável for the 2nd semester. Login and password for access to the protected section can be provided via e-mail: skl@ieeta.pt.

[6] V.Sklyarov. Graphical Description and Hardware Implementation of Parallel Control Algorithms. Proceedings of PDPTA'99, June, Las Vegas, USA, 1999, pp. 1390-1396.

[7] A.Zakrevskij, V.Sklyarov. The Specification and Design of Parallel Logical Control Devices. Proceedings of PDPTA'2000, June, Las Vegas, USA, 2000, pp. pp. 1635-1641.

[8] http://www.xilinx.com.

[9] http://www.celoxica.com.