JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 337-351 (2018)
DOI: 10.6688/JISE.201803_34(2).0002

Short Paper

An Approach to the Design of Specific Hardware Circuits From C Programs

CHENG-JUEI YU, YI-HSIN WU AND SHENG-DE WANG
Department of Electrical Engineering
National Taiwan University
Taipei, 116 Taiwan
E-mail: sdwang@ntu.edu.tw

This article presents an approach that helps convert a given C program into a hardware implementation for a digital circuit design. Based on and extended from the concept of hierarchical finite-state machines (HFSMs), four built-in HFSM templates, namely *Seq*, *Par*, *Loop* and *Atomic*, are proposed and used as the elementary components of a hardware design. A guideline on the refinement of a C program is also proposed; the refined C functions are compiled into HFSMs that in turn generate synthesizable hardware description language (HDL) code as the final design. A set of HFSMs is viewed as an intermediate representation between C and HDL and can be functionally simulated. Two modeling levels, *i.e.* cycle-accurate and cycle-approximated, are supported. A compilation technique based on syntax-directed translations is used to automate the proposed approach. Experimental results on several well-known algorithmic benchmarks show the effectiveness of the proposed approach.

Keywords: finite state machine, intermediate representation, hardware synthesis, hardware description language

1. INTRODUCTION

Recently, ever-growing complexity of hardware design poses great challenges. Today many embedded systems are initially described using an intuitive, high-level behavioral language, such as C. Although the general-purpose processors and software compilers can run C programs in an easiest and cheapest way, well-designed customized hardware can always do the job faster, using fewer transistors and less energy. Thus, generating efficient hardware from C becomes an ongoing topic that has attracted many researchers. However, there is a great gap between software and hardware designs. The infeasibility of describing concurrency model, communications, data types, and detailed timing are the fundamental problems when using C to specify hardware [1].

Since the late 1980s, there are many C-like hardware languages proposed. Related tools can be found in [2-9]; they in general perform the transformation from a behavioral level specification of hardware to an implementation in terms of circuits, and their users have to learn a new language with extended constructs/syntaxes that vaguely resemble C (*e.g.* BDL [3]; Handel-C [4]; SystemC [5]; SpecC [9]) to describe details of the circuits.

Received April 6, 2010; revised July 18 & September 7, 2010; accepted October 18, 2010.
Communicated by Chung-Ping Chung.

The idea presented in this paper is that, rather than having to learn a new language, the designers refine the given C program (according to the guideline proposed in this paper) such that each refined C function can be mapped to a separate hardware block in an effective way; more details are provided later in this paper.

Considering the implementation of the hardware, finite state machines (FSMs) are probably the most widely used components included in almost all the available automatic design tools. Theoretically, most of the digital systems can be realized from a formal FSM specification (extended with storage elements if needed); but the growing complexity of hardware design makes it nearly impossible to create the design from scratch and ensure its quality in a reasonable time. Moreover, the basic FSM, which is flat and sequential, has a major weakness; most practical systems have a very large number of states and transitions. Representation and analysis become difficult [15]. One of the solutions to this problem is based on the concept of hierarchy [16, 17]. In a hierarchical FSM, a state may be further refined into another FSM; that state at one level of the hierarchy is interpreted as being in one of several states at the lower level of the hierarchy, representing a FSM is *calling* another FSM to execute. Our idea presented in this paper is to map each refined C function to a specific HFSM that in turn represents a specific hardware block.

In order to bridge the gap between the behavioral C and the HFSM specification, we propose four built-in HFSM templates: *Seq*, *Par*, *Loop* and *Atomic*. Each template has a number of pre-defined variables (wires or registers) and actions (state transitions) used to communicate with other HFSMs and/or represent its internal status, and can be further edited to perform the behavior specified in C. Once completed, a HFSM is used to produce the corresponding synthesizable HDL code as a hardware block; a set of connected HFSMs can be viewed as an intermediate representation between C and hardware and can be functionally simulated. All of the HFSMs are basically cycle-accurate; an extension to the template of *Atomic* is the mode of cycle-approximated, which brings the support of two-level modeling of the design.

As discussed in [13], any digital system consists of two major parts – an operational unit and a control unit. The logic design of operational units such as counters, adders, memory elements, *etc.*, is well developed and does not present special difficulties; in contrast, the control unit, which determines the information processing dynamics in a digital system, is the most complex part of the whole system. It can be said that the logic design of a digital system is reduced as a rule to the design of its control unit. Considering our approach, the first three types of the proposed templates represent the components of the control units (or control paths, more generally); the last one, *Atomic*, is used for the design of the operational units (or datapaths). We focus on the designs of control paths and datapaths separately, as discussed later in this paper.

2. THE BUILT-IN HFSM TEMPLATES

We propose four HFSM templates that represent the starting points and are used as the elementary components of a hardware design. Since one HFSM will be mapped to a specific hardware block in the final stage of the design process (as discussed later in Section 6), for convenience but without ambiguities we will use concrete hardware blocks

throughout this paper to demonstrate the proposed HFSM templates and their properties. As shown in Fig. 1, the proposed HFSMs have a common configuration; they are synchronously clocked with the global initialization signal *reset*, communicating with others through the input signal *req*, the output *busy*, and/or some data pins. The register *state* is used to represent a block's current state, and initialized as *IDLE* (a state parameter). The input *req* is used to activate a block; the output *busy* is set on when *state*≠*IDLE*, indicating that this block has been activated; otherwise off indicating this block is available to be called. The notation $\ominus(x)$ is used to group the signals of *req* and *busy* of a block *x*; if there exist some pins being data inputs or outputs, they are also grouped. This common configuration is fixed across all the different HFSMs such that they can pass messages to each other following a communication protocol, as illustrated later in this section. Note that in this common configuration the state transitions are not set yet and should depend on different templates or specific HFSMs implemented.

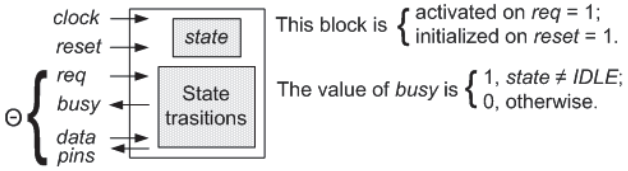

Fig. 1. The common configuration of the proposed HFSM templates.

Based on and extended from the configuration, four proposed HFSM templates, *Seq*, *Par*, *Loop* and *Atomic*, are specified as follows. For the demonstration purposes, when a set of HFSMs may be called by a parent HFSM, they are denoted as {A, B, …, F} or {A-F}; if only one HFSM can be invoked, it is denoted as Z.

As shown in Fig. 2 (a), the templates of *Seq* and *Par* have the same configuration; they have a set of HFSMs {A-F} at the lower level of the design hierarchy that are to be called. The block connects {A-F} through $\ominus(x)$; a set of registers, {*req_x*}, is used to call the lower-level block *x*, where $x \in$ {A-F}. The difference between the templates of *Seq* and *Par* is the state transitions defined. As shown in Fig. 2 (b), *Par* concurrently calls all of its connected lower level blocks, waiting for their executions to be completed. Thus, the process-level parallelism can be specified in a *Par* HFSM with each connected block being the executor of an independent process. Different from *Par* HFSMs, a *Seq* HFSM can only sequentially call {A-F} one after another, as the state transitions specified in Fig. 2 (c). During the state transitions represented by filled arrows, some actions may need to be taken. For instance and see Fig. 2 (c), during the transition from *IDLE* to *Cal- ling A*, the action is one that sets high the request variable *req_A*. On the other hand, the status of staying at the same state waiting for some event is represented by a rounded arrow, say *Awaiting* that waits for the execution of a block to be completed. Note that once a *Par* HFSM is activated, all of its connected lower level blocks must be called; whereas in a *Seq* block several calls to the lower level can be skipped according to the state transitions specified in the corresponding C function. Also note that for a *Seq* or *Par* HFSM every call to a lower-level block can be made only once during the activation of the HFSM. Finally, different from *Seq* and *Par*, a *Loop* HFSM can call another several times, but only *one* lower-level HFSM can be connected with it and thus be called.

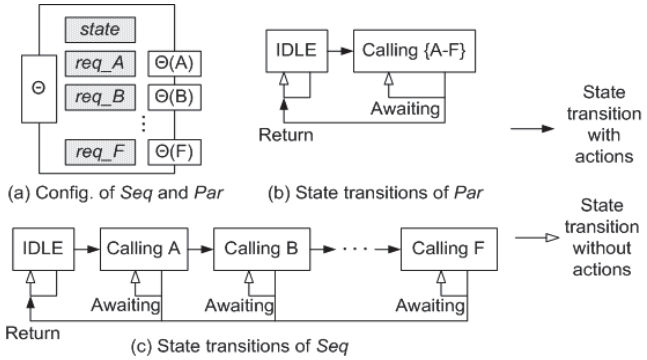

Fig. 2. (a) The configuration of the templates of *Seq* and *Par*; we have registers *req_x* connected to $\Theta(x)$ used to call block *x,* where $x \in \{\text{A-F}\}$; (b-c) Their state transitions with or without actions.

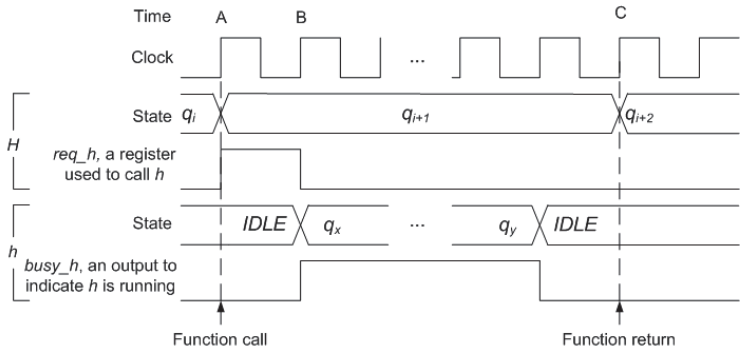

Fig. 3. The waveform of the communication protocol between HFSM *H* and its child *h*.

The template of an *Atomic* HFSM is simple; it is basically the common configuration as shown in Fig. 1. It cannot call any other blocks; thus it lies at the lowest level of the design hierarchy, representing the design of pure datapaths. The state transitions, declarations of registers and wires are determined according to the algorithmic procedure that is to be performed; the only fixed property is that it has to follow the communication protocol for its callers. The protocol, also followed by other types of HFSM templates, is illustrated in Fig. 3 and as follows. Suppose HFSM *H* in state q_i is calling the connected lower-level HFSM *h* by setting high the request register *req_h* at time A. One clock cycle later, at time B, *h* is activated by changing its state from *IDLE* to q_x. During the invocation, *req_h* is set low; an output *busy_h* indicating *h* is running is set high; and *H waits* in state q_{i+i} until *h* finishes its task by returning to state *IDLE* and setting *busy_h* low, which in turn notifies *H* that the function call is completed, at time C. *H* then leaves state q_{i+i}. Notice we spend two fixed clock cycles on the call and the return of the lower-level HFSM, implying that the greater the number of cycles needed to execute a subroutine, the smaller the communication overhead.

With a specified algorithmic procedure, an *Atomic* HFSM can be configured by using one of the many famous scheduling algorithms and heuristics (*e.g.* as-soon-as-possi-

ble (ASAP); force-directed [18]; list-scheduling [19]; exact scheduling [20, 21]) to determine the sequences of the operations execution under various constraints (*e.g.* resource and/or timing constraints, pipelined and/or multicycle operations). In addition, an *Atomic* HFSM can be viewed as a functional black box. When the algorithmic procedure is not yet determined in the early stage of a design process, designers can just feed the desired outputs according to its inputs on the data pins, possibly after a delay of a number of clock cycles. This brings the support of the cycle-approximated simulation, as given later in Section 5.

3. HFSM DESIGN HIERARCHY AND SHARED MEMORIES

The design hierarchy of a set of connected HFSMs can be visualized as a tree structure as shown in Fig. 4. A *Seq* HFSM is represented by a thin-circle node; *Par* by a bold circle; *Loop* by double circles; and triangle nodes are leaves with no successors representing *Atomic* HFSMs. Two nodes are connected through ⊖ represented by an arc between them. Each node is labeled; nodes with the same label represent the same reusable HFSM, say the nodes labeled *d*, which will lead to resource savings since only one hardware block will be generated for them. Recall that the datapath design is within *Atomic* HFSMs; storage elements such as a shared memory may be needed, say *RAM* shared by *e* and *h* in the figure. In this section, we consider the hardware implementation of this design hierarchy including shared components, *i.e.* the reusable HFSMs and shared memories.

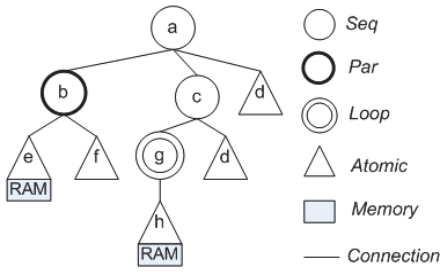

Fig. 4. The design hierarchy of a set of connected HFSMs.

In hardware design, data are usually shared using a shared memory accessible through different hardware blocks, or using message passing mechanisms. While the latter has been considered by our approach (through the data pins as shown in Fig. 1), we focus on the implementation of shared memories, or more generally shared hardware blocks.

In our approach, a memory can be modeled by a simplified HFSM; it has no *req* or *busy* signals; only data pins such as *addr*, *din*, *dout*, *we*, *etc.* are grouped into ⊖. It is viewed as a reusable HFSM with a fixed number of states, *i.e.* one or several clock cycle(s) to read or write a data. The blocks accessing the memory may use its data output after a delay of a fixed number of clock cycles, rather than setting and waiting for the handshaking signals, *i.e. req* and *busy*, in the communication protocol as specified in Fig. 3.

As shown in Fig. 5 and following Fig. 4, the physical connections of the hardware

blocks mapped from the design hierarchy of a set of connected HFSMs can be determined as follows. First, a nearest common parent for each type of the shared components is found. For instance, the common parent of the *RAMs* (the dashed rectangles) is node *a*. Next, the shared component is moved to the next level of the common parent and connects with it, say *RAM* is moved to level 1 and connects with *a*. Finally, the shared component and its callers are connected across the layers between them. As an example, the connection between *RAM* and *h* is across blocks *a*, *c*, and *g*; similarly, the connection between *d* and *c* is across *a* that is also the nearest common parent of the shared nodes *d*'s (including the dashed triangle). Now we have a new design hierarchy where the nodes represent distinct HFSMs/blocks and the arcs are physical links between them.

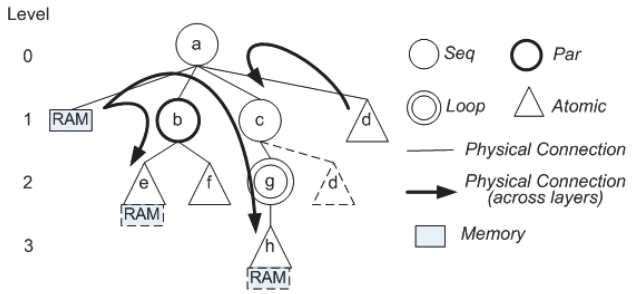

Fig. 5. The physical connections of the hardware blocks represented by distinct HFSMs.

One problem to the shared components is the access collisions that can occur if several blocks attempt to access/activate the same component at the same instant. We resolve this by using a multiplexor at their common parent to select the access rights assigned to one of the blocks. In general, the block that is busy (when its output signal *busy* is on) gets the rights; that is, the link between the shared component and the calling block is held by the common parent through the multiplexor until the calling block's *busy* signal is off. The design hierarchy shown in Fig. 5 and the mentioned links, multiplexors, and related signals will be automatically generated as RTL code during the final stage of the design process, as given later in Section 6.

4. REFINEMENTS OF C FUNCTIONS

For a given C program, our idea is to firstly refine it and then map each refined C function to a specific HFSM that in turn represents a hardware block. In this section we give the refinement guideline on a C program such that the structure of the refined C functions is similar to the design hierarchy of a set of connected HFSMs as shown before in Fig. 4. In other words, the behavior of one refined C function should be similar to one of the proposed HFSM templates. As listed in Table 1, we outline the types of the refinement problems and their solutions in terms of HFSM templates and properties. As can be seen, types (A)-(D) consider the mapping of a refined function and a HFSM template. Specifically, only four function types, *Seq*, *Loop*, *Par* and *Atomic*, are allowed in a refined C program. A *Seq* function sequentially invokes other functions in a specific order; between two consecutive subroutines are simple assignments and *if-then-else* state-

ments rather than complex arithmetic and memory operations that should be refined into an *Atomic* function. A *Loop* function iteratively calls another function; similar to *Seq*, between two consecutive calls to the subroutine are simple statements. In a *Par* function, no statements between two consecutive subroutines are allowed, since the subroutines will be executed concurrently in the mapped *Par* HFSM. Thus, we can specify process-level parallelism in a *Par* function with each subroutine being an independent process. Finally, there are no special regards to an atomic function except that it cannot call and execute any other functions; it represents the design of pure data flow, usually composed of a sequence of memory and arithmetic operations.

Besides the four function types, problems (E)-(G) in Table 1 focus on the data shared by different functions. (E) and (F) represent the message passing mechanisms for a function call; the arguments and return value of a function are mapped to the data pins of the corresponding HFSM. In problem type (G), the allocated memory (array) in a C program is regarded as a shared component that can be accessed by different HFSMs through an address offset signal (and other signals, *e.g.* data, but not mentioned here) being a specific output on one HFSM's data pins.

Table 1. The refinement problems in C and their solutions.

Type	Characteristics of C functions	Mapped HFSM template
(A)	Invokes others in a specific order	*Seq*
(B)	Iteratively calls another function	*Loop*
(C)	Performs a specific task itself	*Atomic*
(D)	Same as (A) but the order is arbitrary	*Par*
(E)	Arguments to the function	Inputs or outputs on the data pins
(F)	Return value of the function	An output on the data pins
(G)	Allocated memories (arrays)	Shared components

One can gradually refine a given C program according to the refinement problems and their solutions shown above. More formally, we have four main refinement steps stated as follows. The details of each step are given in the rest of this section.

1. Rewrite each of the C functions to one of the types {A-D} listed in Table 1. We may need to create a new function or delete one in this step.
2. Identify the pointers that represent pass-by-reference arguments to functions. They will be used as outputs on the data pins of the mapped HFSMs.
3. Replace dynamic memory allocations to static memory allocations. Create the memory reading and writing functions. The allocated memories will be mapped to the shared components in the HFSM design hierarchy.
4. Add comments above the declaration of each refined function with information of the HFSM type and the widths of the declared variables in terms of a number of bits. These comments will be read to configure the mapped HFSM.

We firstly focus on step 1 since the rewriting procedure plays a key role that determines the structure of the design hierarchy, and thus directly affects the performance of the hardware implemented. Based on the operation of Extract Method in the field of code refactoring [14] that is the process of changing a software system so that it does not alter

the external behavior of the code yet improves its internal structure, we change a given C program in the following ways. When a method is too long or a fragment of code needs a comment to understand its purpose, that fragment is extracted into its own method. The newly created methods are usually short (if still long, extract them again if possible) and well-named. This operation increases the chances that other components can use a method when the method is finely grained. Since the fragment of code extracted is replaced by a line of subroutine call, it allows the higher-level methods to read more like a series of comments. Regarding the hardware implementation, a reusable method synthesized as a shared block can lead to resources savings; a shorter, clearer method can also potentially lead to improved readability and maintainability of the low-level descriptions.

In addition to the typical Extract Method, we further consider simplifying the control flow of a method that has loops. The intuition is that a method with a complex control flow might result in a much more complicated controller design when realizing it in hardware. In order to exclude the statements of *continue* and *break* that eventually complicate the control flow, we force a method to have only one subroutine that can iteratively be called; no statements are allowed outside the loop except *return* (see Fig. 6 (b) for a concrete example). In this method, the statement of *break* is replaced by *return*; *continue* is replaced by *if-then-else*. While this approach leads to simpler control path design that can be directly mapped to digital circuits, the overhead is the refactoring tasks that have to be regarded by designers. Fortunately, these tasks will be performed at the highest design level; it is much easier to verify the correctness of the refined functions (*e.g.* using gcc) and to change the overall structure of the design (*e.g.* using another refinement paradigm).

There are methods (functions) that cannot be extracted anymore. In this paper we call them *atomic* functions (similar to *atomic* HFSMs as having been discussed in Section 2). An atomic function is usually composed of a sequence of memory and arithmetic operations. For example, a 256-point FFT program has atomic functions such as butterfly calculation and samples re-ordering; both of them read data from arrays, compute the desired values, and then write the data back. Considering the statements in an atomic function, it is usually plausible to exposing fine-grained parallelism using the two approaches discussed in [1].

The first one, instruction-level parallelism (ILP), dispatches groups of nearby instructions simultaneously. The second one, pipelining, dispatches instructions in sequence but overlaps them – the second instruction starts before the first completes. Both of the approaches can be identified automatically by using various scheduling algorithms such as as-soon-as-possible (ASAP), force-directed [18], list-scheduling [19], and exact scheduling [20, 21]. From this point of view, if a fragment of code is not long but looks like atomic or has potential to be scheduled effectively, it is worth creating a new method for it.

When the refinement step 1 is done, we will roughly have the design hierarchy similar to the tree structure as shown before in Fig. 4 that is originally used for HFSMs. Now each node represents a refined C function. We then perform the refinement step 2 that considers the return values of a refined function. Due to the limits of the C language, only one return value is permitted for a declared function. If one needs another return value, it is usually put as a pass-by-reference argument to the function, and will be used as an output on the data pins of the mapped HFSM.

In the refinement step 3, the dynamic memory allocations are replaced by static

ones. Two functions to read and write an allocated memory (array) are created. If a lowest-level function needs an access to the memory, an address offset is given as an argument to the function and used to read a data from the memory.

As shown in Fig. 6 and for each of the refined functions, the final step is to insert the information that is required by hardware designs such as HFSM types and variables widths in terms of a number of bits. We use Doclet annotations (supported in JAVA API) inserted above each function declaration such that they can be read to configure the mapped HFSM. As an example, on line 2 in Fig. 6 (c) we annotate this function is type *Par*, which infers a process-level parallelism for its subroutines; on line 3 we annotate the width of *x* and *y* is 8. For the demonstration purposes, we also insert one state after each of the subroutine calls in the function. The name of a state is the name of the corresponding subroutine appended to a meaningful word, say the state *await_b* in Fig. 6 (a) on line 7 with respect to the function call *b*( ). A state inserted in this way is used to wait for the execution of the subroutine to be completed. A state *IDLE* is inserted after the variables declaration (on line 6), used to initialize the registers and indicate the mapped HFSM is available to be called. Whenever the statement of *return* is reached, the HFSM should change its state to *IDLE*. When the call to a subroutine is completed, some actions must be taken and the state may transit to another. Take the *Loop* function in Fig. 6 (b) as an example, when the call to *h*( ) (on line 8) is completed, the assignments and conditions specified on lines 9-14 and 7 are the actions that must be done consecutively in one clock cycle during the state transition. Note that since most of the complex arithmetic and memory operations are already refined into the lowest-level function as the design of pure datapaths, in such a control-intensive function the actions would be simple and not expose other critical issues. Besides, in a *Par* function as shown in Fig. 6 (c), we have only two fixed states inserted: *IDLE* and *WAIT*. In the mapped *Par* HFSM (refer to Fig. 2 (b)), the subroutines, say *e*( ) and *f*( ) in this figure, will be called concurrently, although in C they are still sequentially be called. In the mapped HFSM the state *WAIT* is used to wait for all of their executions to be completed.

```
   // … Doclet comments …
   void a ( ) {
6     …(var. dclr.)        // IDLE
7     x = b ( );           // await_b
      if (x) {
         y = c (n, m);     // await_c
         if (y) return;
         else …
      } else {
         d ( );            // await_d
      }
      …
   }
```

(a) *Seq* function

```
   // … Doclet comments …
   int g (int n, int m) {
      i = n; x = 0;      // IDLE
      while (i < m) {
         t = h (i);      // await_h
         if (t == -1)
7           return x;
8        else {
9           x += t;
            i += 1;
         }
      }
      return x;
14 }
```

(b) *Loop* function

```
   /**
2   * @type Par
3   * @width x y 8;
    **/
   int b (int* x, int* y) {
      …(var. dclr.) // IDLE
      *x = e (…);
      *y = f ( );     // WAIT
      return *x + *y;
   }
```

(c) *Par* function

Fig. 6. Examples of *Seq* (a), *Loop* (b), and *Par* (c) functions illustrating the information of HFSM types, variables widths, and states that are inserted above the function declarations and after the subroutine calls. Note that in a *Par* function we have a fixed state *WAIT* that awaits the executions of all the subroutines to be completed.

5. CYCLE-APPROXIMATED SIMULATION

With a set of refined C functions, we are ready to perform a cycle-approximated simulation by inserting some code as shown in Fig. 7 marked by brackets. In order to record the cycles spent by a subroutine, a pass-by-reference argument is added to it, say *higher_level_cycles* on line 2 and *cycles* on line 5 in (a). In the begin of each refined function, *cycles* is initialized as 1 since the activation of a block spends one cycle; after each function call, *cycles* is incremented by 1 since a state change also spends one cycle. Note that the actions with respect to the state change are taken during that cycle so that they do not pose other cycle wastes. Before the return of a function, the cycles recorded by the higher level function, *higher_level_cycles*, are accumulated to the total cycles spent by the function (see lines 11 and 5 in (a) and (b), respectively). As can be seen in (a), for a *Seq* or *Loop* function the cycles spent by its subroutines are accumulated and stored in *cycles*. The cycle-approximated ones are specified in an atomic function as shown in (b) on lines 3 and 5. The parameter *K* on line 3 represents the approximated cycles spent by this atomic function; it can be adjusted by designers or according to some estimating rules. If the design of an atomic function has not yet been determined, one can just feed the desired outputs (the return value, the pass-by-reference arguments, or the memory data that might need to be updated) according to its inputs and also estimate the number of cycles it might spend on the task that is going to be performed. Finally, we can record the total approximated cycles spent by the whole set of the refined functions through a *main*( ) that calls the root-node function in the design hierarchy and verifies the results at the same time.

```
   /* Seq or Loop function */
2  int func (…, int* higher_level_cycles) {
   ⌜ int cycles = 1;
     …
5  ⌜ first_func (…, &cycles);
   ⌞ cycles += 1;
     …
11 ⌜ *higher_level_cycles += cycles;
   ⌞ return …;
   }
```

(a) A *Seq* or *Loop* function.

```
   /* Atomic function */
   int func (…, int* higher_level_cycles) {
3  ⌜ int approximated_cycles = K;
     …
5  ⌜ *higher_level_cycles +=
   ⌞    approximated_cycles;
     return …;
   }
```

(b) An *Atomic* function.

Fig. 7. Cycle-approximated simulation is supported by inserting the C code marked by brackets into each refined function.

6. VERILOG CODE GENERATION

With a set of refined C functions mapped to a set of connected HFSMs shown in Fig. 4 and further in Fig. 5, we are ready to generate HDL code as the final design. As shown in Fig. 8, the code fragments written in Verilog describe the state transitions specified in the refined C functions given in Fig. 6. In Fig. 8 (a), HFSM *a* calls its children HFSMs *b*, *c*, and *d* by setting high the request registers on lines 76, 86, and 89, respectively. When *b* is running, the state is waiting at *await_b* due to the high of *busy_b* on line 83; while *b* is finished, the state transits to *await_c* or *await_d* depending on the

output value, *x*, of *b* on lines 84 and 85. Note that blocking assignments are used to ensure the correct behaviors specified in the C program. In Fig. 8 (b) HFSM *g* (iteratively) calls *h* through setting on the registered output *req_h* on line 64. The two *return* statements in Fig. 6 (b) on lines 7 and 13 are accomplished here on lines 59 and 67, respectively. When one call to *h* is completed, the actions between the state *await_h* that waits for the call and the next state, possibly *await_h* or *IDLE*, are taken on lines 58-69. The code shown can be automatically generated from a given set of proposed HFSMs without ambiguities.

```
always@ (posedge clk or
posedge reset) begin
   if (reset) begin
      … // registers initialization
      state <= IDLE;
   end else begin
      case (state)
         IDLE: begin
            if (req) begin
76             req_b <= 1;
               state <= await_b;
            end
         end
80       await_b: begin
            if (req_b)
               req_b <= 0;
83          else if (!busy_b) begin
84             x = x_b;
85             if (x) begin
86                req_c <= 1;
87                state <= await_c;
               end else begin
89                req_d <= 1;
                  state <= await_d;
               end
            end
         end
         ... // other cases
      endcase
   end
end
assign busy = state != IDLE;
```

(a) *Seq* block

```
always@ (posedge clk or
posedge reset) begin
   if (reset) begin
      … // registers initialization
      state <= IDLE;
   end else begin
      case (state)
         IDLE: begin
            if (req) begin
               i = n; x = 0;
               if (i < m) begin
47                req_h <= 1;
48                state <= await_h;
               end
            end
         end
52       await_h: begin
            if (req_h)
53             req_h <= 0;
54          else if (!busy_h) begin
               t = t_h;
56             if (t == -1) begin
58                state <= IDLE;
59             end else begin
60                x = x + t;
                  i = i + 1;
                  if (i < m) begin
63                   req_h <= 1;
64                   state <= await_h;
65                end else begin
66                   state <= IDLE;
67                end
               end
69          end
         end
      endcase
   end
end
assign busy = state != IDLE;
```

(b) *Loop* block

Note: underlined words are inputs; shaded are outputs; bold are registers; italic are constants; otherwise wires.

Fig. 8. Generated Verilog code fragments describing the state transitions for the refined functions in Fig. 6 (a) for Fig. 6 (a); (b) for Fig. 6 (b).

7. EXPERIMENTAL RESULTS

In order to compare with existing C-to-hardware approaches, the proposed approach should be able to automatically generate Verilog code from a given refined C program. The automation can be realized by using a compilation technique based on syntax-directed translations [10] that is a method of translating a set of statements into a sequence of actions or instructions by attaching one such action to each rule of a grammar. The elementary instructions translated in the form of three-address code are then used to generate the corresponding Verilog code.

We compare the performance of the proposed approach with Nios II C2H compiler [8] by implementing several well-known algorithmic benchmarks such as FDCT, 256-point FFT, bubble sorting, and finding primes using prime wheels, on the Altera DE2-70 FPGA development board at a clock frequency of 50 MHz. The netlist compilation for the generated Verilog code is performed by Quartus II Software [11]. C2H compiler is a tool for generating a hardware accelerator module which is functionally identical to the original ANSI C function in a fully automatic way; the subroutine calls in the function are also accelerated. Since the input to the proposed approach is a C program with manually refinements, we also test C2H's performance with these refinements. The indices of the performance evaluation include the runtimes of the circuits, the maximum operating frequency (F_{max}) achieved, the area of logic slices occupied, and the number of Verilog code lines generated. The results are shown in Table 2; we can see that our approach can generate faster circuits with lesser area. Our approach also produces comprehensive code with a reasonable, smaller number of code lines. One penalty of using our approach is the achieved F_{max} that is slightly lower than C2H in benchmarks (a), (c) and (d). The reason is that our approach synthesizes the actions between two consecutive states to be executed in one clock cycle – that would lead to longer critical paths – while C2H generally infers one clock cycle for a C statement. We also observed that C2H has no obvious benefits from the refinements on the given C programs (although in (a) and (d) the area reduces from 13% to 9% and 7% to 6%, respectively, this trend does not appear in (b) and (c); the runtimes without refinements are also shorter than or equal to those with refinements, say 11s in (a), 9s vs. 13s in (b), and 32s vs. 41s in (d)).

Table 2. Performance evaluation of the proposed approach against C2H.

Index / Benchmark	Proposed approach (with refinements on C)				C2H (without/with refinements on C)			
	Run-times(s)	F_{max} (MHz)	Area (%)	Code lines	Run-times(s)	F_{max} (MHz)	Area (%)	Code lines
(a) FDCT in JPEG encoder 256×256 pixels	6	67	8	903	11	68/71	13/9	10,779/7,718
(b) 256-point FFT 10000 iterations	3	85	4	1901	9/13	76/73	6/7	8,408/9,018
(c) Bubble sort 10,000 numbers	13	87	3	268	21	94/96	4	2,736/3,364
(d) Finding primes < 500,000	10	80	5	893	32/41	95/96	7/6	7,427/6,249

Notes: 1. Shaded cells are ones that indicate the performance of our approach is superior to C2H.
2. For each of the experimental cases, we spent no more than half an hour to refine the given C program.

A note on the computational and space complexities of the proposed approach is given as the conclusion of this section. Since the automation of the proposed approach is mainly based on syntax-directed translations that are popularly used in modern compilers such as gcc and javac, these complexities for code translations (C to three-address code and three-address code to Verilog) would not pose critical issues; any modern desktops that are able to run gcc and javac would also be able to run the proposed approach. In contrast, large runtimes and memory usage would be needed if we are searching for an optimized schedule that determines the order of operations execution within an *Atomic* function. For the experimental cases, a simple ASAP schedule is used in each *Atomic* function. Therefore, under these assumptions, the complexities of the proposed approach are low enough.

8. SUMMARY

We proposed an approach to specify hardware designs and expose parallel algorithms starting from C language. We proposed four built-in HFSM templates that follow a communication protocol and can be further edited for the desired behavior. Each HFSM can be mapped to a specific hardware block with the signals and state transitions defined. We presented one possible way to automatically generate the Verilog code describing the set of derived HFSMs. With considerations of the message passing and shared memory mechanisms used in hardware, we proposed a guideline on the refinement of a C program such that each refined C function can be mapped to a specific HFSM that in turn represents a hardware block. A set of refined C functions can be extended to support a cycle-approximated simulation by adding an argument to each function. Finally, compared with Nios II C2H compiler [8] that is an automatic behavioral synthesis tool, our approach can produce faster, smaller circuits with a smaller number of Verilog code lines.

REFERENCES

1. S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Design & Test of Computers*, Vol. 23, 2006, pp. 375-386.
2. T. Grötker *et al.*, *System Design with SystemC*, Kluwer Academic Publishers, 2002.
3. K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: An ASIC and system vendor perspective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 19, 2000, pp. 1507-1522.
4. *Handel-C Language Reference Manual*, RM-1003-4.0, Celoxica, 2003.
5. T. Grötker *et al.*, *System Design with SystemC*, Kluwer Academic Publishers, 2002.
6. D. D. Gajski *et al.*, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.
7. Catapult C, http://www.mentor.com/.
8. *Nios II C-to-Hardware Acceleration Compiler*, http://www.altera.com/.
9. D. Shin, A. Gerstlauer, R. Domer, and D. D. Gajski, "An interactive design environment for C-based high-level synthesis of RTL processors," *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 16, 2008, pp. 466-475.

10. A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles*, *Techniques*, *& Tools*, 2nd ed., Addison-Wesley, Boston, 2007.
11. Quartus II Software, http://www.altera.com/.
12. "Accelerating Nios II systems with the C2H compiler tutorial," http://www.altera.com /literature/tt/tt_nios2_c2h_accelerating_tutorial.pdf.
13. S. Baranov, *Logic Synthesis for Control Automata*, Kluwer Academic Publishers, 1994.
14. M. Fowler *et al.*, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999.
15. A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18, 1999, pp. 742-760.
16. D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, Vol. 8, 1987, pp. 231-274.
17. M. von der Beeck, "A comparison of statecharts variants," in *Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems*, LNCS 863, 1994, pp. 128-148.
18. P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *Proceedings of the 24th ACM/IEEE Conference on Design Automation*, 1987, pp. 195-202.
19. A. M. Sllame and V. Drabek, "An efficient list-based scheduling algorithm for high-level synthesis," in *Proceedings of the Euromicro Symposium on Digital System Design*, 2002, pp. 316-323.
20. Y.-H. Wu, C.-J. Yu, and S.-D. Wang, "Heuristic algorithm for the resource-constrained scheduling problem during high-level synthesis," *IET Computers & Digital Techniques*, Vol. 3, 2009, pp. 43-51.
21. C.-J. Yu, Y.-H. Wu, and S.-D. Wang, "An in-place search algorithm for the resource constrained scheduling problem during high level synthesis," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 15, 2010, pp. 1-13.

Cheng-Juei Yu (余承叡) was born in Taiwan in 1983. He received the B.S. degree in Electrical Engineering from National Taiwan University, Taipei, Taiwan, in 2006. He is currently a Ph.D. student in the Department of Electrical Engineering, National Taiwan University. His research interests include digital circuit designs, high-level synthesis, and search algorithms.

Yi-Hsin Wu (吳怡欣) was born in Taiwan in 1984. She received the B.S. degree in Electrical Engineering from National Taiwan University, Taipei, Taiwan, in 2007. She is currently a Ph.D. student in the Department of Electrical Engineering, National Taiwan University. Her research interests include digital circuit designs, high-level synthesis, and search algorithms

Sheng-De Wang (王勝德) received the B.S. degree from National Tsing Hua University, Hsinchu, Taiwan, in 1980, and the M.S. and the Ph.D. degrees in Electrical Engineering from National Taiwan University, Taipei, Taiwan, in 1982 and 1986, respectively. From 2001 to 2003, He has been served as the Department Chair of Department of Electrical Engineering, National Chi Nan University, Puli, Taiwan for the 2-year appointment. His research interests include parallel and distributed computing, embedded systems, hardware software co-design, and intelligent systems.