On the Control-subroutine Implementation of Subprogram Synthesis

Cheng-tsung Hwang
Dept. of Tourism,
Providence Univ., Taichung,
Taiwan, R.O.C.
Tel: 001-886-4-2511077
Fax: 001-886 4-6328001
e-mail:jthuang@tour.pu.edu.tw

Hsiao-Chien Weng, Yu-Chin Hsu
Dept. of Computer Science
Univ. of California,
Riverside, CA 92521
Tel: 909-787-4406
Fax: 909-787-4643
e-mail: hsu@cs.ucr.edu

Mike Tien-Chien Lee
Fujitsu Labs. of America
3350 Scotts Blvd., Bldg #34
Santa Clara, CA 95054
tel: 408-567-4510
fax: 408-567-4515
email: lee@fla.fujitsu.com

Abstract - In this paper, synthesis of VHDL procedures and functions is studied from the VHDL transformation point of view. Among all the proposed methods, inline expansion and module can be integrated into a VHDL synthesis system by a source-to-source transformation, while a control-subroutine approach requires additional work at the higher level synthesis phases before it can link to a logic synthesis tool. A lot of optimization possibility is explored during the process. We also present various generations of the control-subroutine approach, including the synthesis of recursive programs, a behavioral partitioning methodology that divides the controller into several communicating state machines, and a methodology that mixes the execution of subprograms. Our study shows that the combination of these approaches is flexible to be adapted to various applications in an efficient way.

I. Introduction

As the complexity of a design increases, the size of a design description increases. It becomes necessary to define subprograms within the main program in order to clarify the whole design and provide the opportunity to share some common portion of the program. A subprogram (or subroutine) defines a function using a sequence of declarations and statements in behavior constructions that can be evoked from different locations in a VHDL program [1,2,3,4,5,6]. When two tasks perform the same sequence of operations except on different data sets, a subroutine associated with parameters is usually shared by the two tasks.

Procedures and functions are examples of subprograms in VHDL. Their use in a behavioral description has the advantages of decreasing the description size, enhancing readability, and improving maintainability [7]. In the past, a VHDL program was mainly for design description and simulation. As synthesis tools become more mature, synthesis from behavioral VHDL is becoming a popular issue. Previous research efforts have examined synthesis of descriptions containing subprograms. Their methodologies fall into three philosophies. The first one removes a subroutine by inlining its function into the main program when the subprogram is evolved (inline expansion). The second one allocates dedicated resource for a subroutine (module or macro). The last one compromises them by keeping the control structure of each subroutine while sharing the data path among all subroutines (control subroutine or share-routine).

Among them, inline expansion is the most widely used due to its simplicity of implementation and its flexibility to be adapted to all the applications. However, as the size of the program increase, the routing among the data path components and the control becomes unacceptable. Furthermore, a subroutine usually represents a group of operations that have strong relationship. Data are produced by some operations and then consumed by other operations in the same subroutine. An inline approach is not intended to maintain the property. Therefore, a synthesis method that is able to address the above issues is desired.

The problem of subprogram synthesis is complicated by the presence of parameters, signals and wait statements, the demand of low power consumption, and possibly recursion in nature. We found that the best implementation style of subroutine is different from one to another and the quality of a design cannot be predictable unless lower level design has been completed. A synthesis system that supports various implementation styles, integrates methodologies, and easily links to a lower level design tools, can have the advantage of exploring the design space.

This paper will address the above issues. In Section 2, the various ways of implementing a subroutine is examined from VHDL transformation aspect of view. In section 3, we present the control-subroutine approach in detail. Various generations of the control-subroutine approach and the relationships with other methodologies are discussed in section 4. The experimental result is presented in section 5.

II. Survey of Procedure Implementation Styles

The most common subprogram synthesis method is inline expansion where the subroutine is expanded into the main program whenever it is evolved. In SUGAR[8] and System Architect's Workbench[9], procedure calls are either inlined or implemented as jumps to micro-subroutines in the control stores. Procedures in SUGAR are inlined if they are called only once or if the inlining would not significantly increase the size of the control store. In [10], subroutines are also inlined. However, a fixed pattern, called behavioral template, is defined. During scheduling, they try to match as much such pattern as possible to increase the regularity.

The subprogram is treated as a module in Yorktown Silicon Compiler [11]. In [7], several subprogram synthesis methods, namely, fixed-delay macro, variable-delay macro, procedure inline, and control subroutine are presented and the effects of VHDL signals and wait statements on synthesis methods are studied. Their evaluation criteria is based on the number of control steps, number of registers, and number of functional units needed.

This paper will discuss the subroutine synthesis methods from VHDL transformation point of view with emphasis on the control subroutine approach. To synthesis a program contains subroutines, some tasks of a synthesis system may have to be modified. Among them, inline expansion and process module can

✦ This work is supported by the National Science Council of R.O.C. under contracts No. NSC-84-2215-126-002.

be easily integrated into the synthesis tool by introducing a source to source transformation phase. In the following, we briefly overview these methods using the example in figure 1(a) which is presented in [7]. The implementation of control-subroutine requires additional works at higher level synthesis phases which will be discussed in Section 3 and Section 4.

Inline Expansion:
Inline expansion is the most intuitive and most widely used method for realizing a subprogram. Basically, for each subprogram call, we replace the subprogram call statement by the subroutine body. Parameters are handles in two ways. If they are variables, we have to rename the formal parameters. If they are signals, we have to keep them the same. After transformation, the subprogram is no longer needed and can be removed from the source program. Consequently, we have a single data path and control unit for the whole program. Figure 1(b) shows the program after transformation.

Variable-delay or fixed-delay module:
The main idea behind this method is to use dedicated hardware for a subroutine. There are two different kinds of subroutines, i.e., fixed delay and variable delay. For a variable-delay one, we have to introduce handshaking between the main program and the subroutine. The main program calls the subroutine by setting a start signal. Upon completion, the subprogram informs the main program by setting a finish signal. In this case, the relationship between caller and callee acts like communicating processes as described in figure 1(c). For a subroutine with fixed delay behavior, the handshaking is not required. It is instantiated as if there is a functional-unit designed for it. Figure 1(d) shows the VHDL description of the fixed-delay module. It is declared as a module by using a compiler directive, "-- mebs module".

```
Entity SAMPLE is
      port( I1:in bit_vector(3 downto 0);
            I2: in bit_vector (3 downto 0);
            O1: out bit_vector (3 downto 0);
      );
end;

architecture BEH of SAMPLE is
begin
      process
            procedure SumofSquares(
                  X, Y: in bit_vector(3 downto 0);
                  Z: out bit_vector (3 downto 0) ) is
                  variable XX,YY: bit_vector(3 downto
0);
            begin
                  XX:=X*X;
                  YY:=Y*Y;
                  Z:=XX+YY;
            end SumOfSquares;

         Variable A,B,C,D,E,F:bit_vector(3 downto 0);
      begin
            A:=I1; B:=I2;
            SumofSquares(A, B, C);
            D:=C+1;
            E:=A+1;
            SumOfSquares(D, E, F);
            O1 <= F +3;
      end process;
end BEH;
```

(a)

```
architecture BEH of SAMPLE is
begin
      process

            Variable A, B, C, D, E, F: bit_vector(3 downto
0);
      begin
            A:=I1; B:=I2;
            /* SumofSquares(A, B, C); */
            X1:=A;
            Y1:=B;
            XX1:=X1*X1;
            YY1:=Y1*Y1;
            Z1:=XX1+YY1;
            C:=Z1;
            /* end of SumofSquares */
            D:=C+1;
            E:=A+1;
            /* SumOfSquares(D, E, F); */
            X2:=D;
            Y2:=E;
            XX2:=X2*X2;
            YY2:=Y2*Y2;
            Z2:=XX2+YY2;
            F:=Z2;
            /* End of Sumofsquare */
            O1 <= F +3;
      end process;
end BEH;
```

(b)

```
architecture BEH of SAMPLE is
signal start : bit :='0';
signal finish : bit := '0';
signal X,Y,Z : bit4;
begin

Variable A,B,C,D,E,F: bit_vector(3 downto 0);
      SumOfSquares:
      process
            begin
                  wait until start='1';
                  XX:=X*X;
                  YY:=Y*Y;
                  Z:=XX+YY;
                  finish <= '1';
                  wait until start='0';
                  finish <='0';
            end SumOfSquares;

      begin
            A:=I1; B:=I2;
            X<=A;
            Y<=B;
            start <= '1';
            wait until start='0';
            start <= '0';
            C <= Z;
            D:=C+1;
            E:=A+1;
            X<=D;
            Y<=E;
            start <= '1';
            wait until start='0';
            start <= '0';
            F <= Z;
            O1 <= F +3;
      end process;
end BEH;
```

(c)

```
package body SAMPLE_pkg is
    function SumOfSquares(X,Y: in bit4) return bit4;
end SMAPLE_pkg;

package body SAMPLE_pkg is
      --mebs module
      function SumOfSquares(X,Y:in bit4) return bit4 is
            variable XX,YY: bit4;
      begin
            XX:=X*X;
            YY:=Y*Y;
            return(XX+YY);
      end;
end;

entity SAMPLE is
      port (clock: in bit;
                  I1: in bit4;
                  I2: in bit4;
                  O1:out bit4
                  );
end SAMPLE;

architecture BEH of SAMPLE is
begin
            main:
            process
                  Variable A, B, C, D, E, F: bit4;
            begin
                  A:=I1;
                  B:=I2;
                  C:=SumOfSquares(A, B);
                  D:=C+1;
                  E:=A+1;
                  F:=SumOfSquares(D, E);
                  O1 <= F +3;
            end process;
end BEH;
```

(d)

Figure 1 Source-to-source transformation, (a) the input description, (b) inline-expansion, (c) using a variable delay module, (d) using a fixed delay module.

III. CONTROL-SUBROUTINE APPROACH

Implementing a subroutine as a control-subroutine was first presented in the field of software compiler. The main idea is to associate each subroutine with a dedicated control part, while the operations are executed on the data path allocated for the whole program. In order to transfer control flow among the subprograms, the return address has to be hold in a memory. A device named stack has been designed in most computer systems to simulate the fist-in-last-out behavioral of subroutine call. Whenever a subroutine call happens, the caller pushes the returning address into the stack and sets the new PC to be the first state of the subprogram. At the end of a subprogram, it pops the top address out to the program counter. These control transfer operations, push and pop, are inserted to the caller and the callee at compile time. In this section, implementing a subroutine as a control-subroutine is discussed in detail. We will first describe the target architecture and then present the synthesis flow.

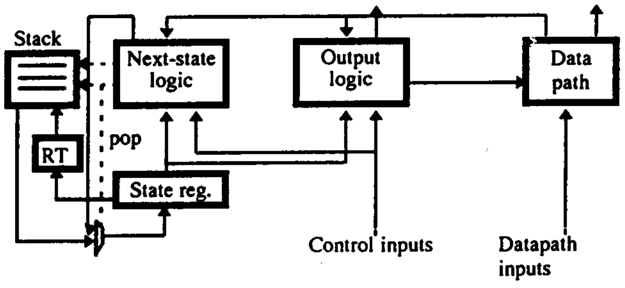


Figure 2. Target architecture: FSMD with a stack.

A. Target Architecture

If a subprogram is going to be synthesized as a control subroutine, a process describing the behavior of a stack has to be created in addition to the original subprograms. Figure 2 shows our target architecture (FSMD with a stack). The stack consists of two components, namely, stack registers and return address generator (RTN). The stack is made of shift registers. It accepts two signals, i.e., push and pop. On push, it pushes the content of each register downward to the register below it and places the return address at the topmost position. On pop, it pops the content of each register upward to the register right above it. At the same time, the value of the topmost register is selected as the new program counter (PC). The return address generator is a combinational logic which takes PC as input and produces a return address to be stored in the stack. The size (depth) of the stack must be large enough to save the sequence of return addresses. However, the exact requirement is in general unknown until run-time if the program is a recursive one. In this case, a designer has to estimate it by simulating the behavior.

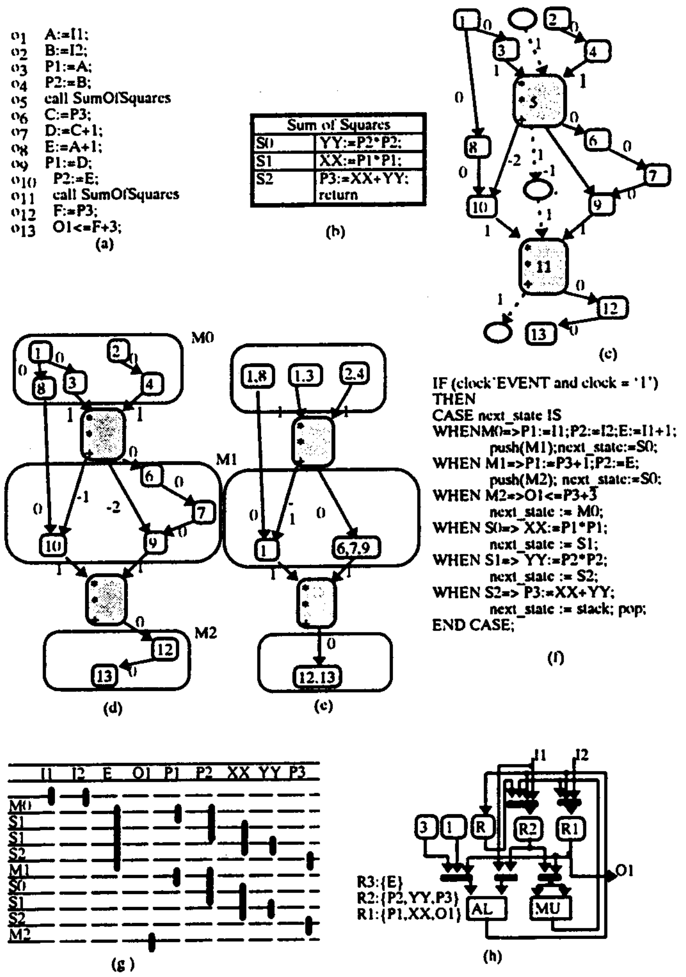
To synthesize a program with control subroutines, some tasks in a synthesis system have to be modified to adopt the new behavioral construction. We will focus on the synthesis of non-recursive programs in this section. The extensions of the method will be discussed in the Section 4. Figure 3 shows the overall algorithm. An example illustrating the synthesis flow is shown in Figure 4. The major steps are explained in detail in the following sections.

```
Synthesize a control subroutine {
      I.    Source transformation
      II.   Schedule subprograms in decreasing ID order
            • Introduce inter-routine dependencies
            • Perform operation scheduling
      III.  Scheduled code optimization
      IV.   Register allocation and Binding
      V.    Functional-units binding and Interconnection generation
}
```

Figure 3. Algorithm for synthesizing a control subroutine.

B. Source transformation

The first step of dealing with a subroutine in control subroutine implementation is the source transformation for formal parameter renaming and parameter passing. The task is similar to that of inline expansion except that the subroutine body is replaced with a control transfer operation. Every time a caller calls the subroutine, it places the parameters on the variables and then passes the control flow to the subroutine. When the control flow returns, it gets result from the fixed location (variable). The codes after source transformation is shown in Figure 4(a).



```
o1   A:=I1;
o2   B:=I2;
o3   P1:=A;
o4   P2:=B;
o5   call SumOfSquares
o6   C:=P3;
o7   D:=C+1;
o8   E:=A+1;
o9   P1:=D;
o10  P2:=E;
o11  call SumOfSquares
o12  F:=P3;
o13  O1<=F+3;
```
(a)

Sum of Squares	
S0	YY:=P2*P2;
S1	XX:=P1*P1;
S2	P3:=XX+YY; return

(b)

```
IF (clock'EVENT and clock = '1')
THEN
CASE next_state IS
WHENM0=>P1:=I1;P2:=I2;E:=I1+1;
        push(M1);next_state:=S0;
WHEN M1=>P1:=P3+1;P2:=E;
        push(M2); next_state:=S0;
WHEN M2=>O1<=P3+3
        next_state := M0;
WHEN S0=>XX:=P1*P1;
        next_state := S1;
WHEN S1=>YY:=P2*P2;
        next_state := S2;
WHEN S2=>P3:=XX+YY;
        next_state := stack; pop;
END CASE;
```
(f)

Figure 4: Example of control subroutine (a) after source transformation, (b) the scheduling result of Sum Of Squares, (c) dependency graph, (d) schedule result of main program, (e) after code optimization, (f) FSMD description, (g) variables' lifetime table (h) data path.

C. Scheduling

The calling sequence of a program can be represented by a dependency graph, which is a directed graph G(V,E) with V for the set of subroutines and E for the set of directed edges. There is an edge, e, with weight n, emits from v_i to v_j if v_i calls v_j for *n* times. Then, we traverse the graph in a breadth-first order and associate each subprogram with an identity number (ID) in the order it is visited. The subprograms will be scheduled in decreasing ID order. For the example, the subprogram, Sum of Squares, has ID of 2. It is scheduled first and the result is shown in figure 4(b). We then proceed to the main program for scheduling, whose ID is 1.

In order to obtain a more global solution, we allow an operation to be scheduled across subroutine calls. For this purpose, a previously scheduled subprogram is treated as a macro while dealing with its higher level. Obviously, the macro must be executed after its input parameter producers but before its output consumers. Likewise, implicit dependency introduced by signals or global variables as either inputs to or outputs from the macro must be kept in order. All these requirements can be achieved by introducing data dependency edges between macro and regular operations. In the example, an edge of weight 1 from o_3 to o_5 enforces o_5 be executed later than operation o_3 by at least 1 step. The other dependencies, $o_4 \rightarrow o_5$, $o_{10} \rightarrow o_{11}$, $o_9 \rightarrow o_{11}$, are defined similarly as in Figure 4(c). Note that $o_5 \rightarrow o_{10}$ has a weight of -2 which specifies a *use-define* dependency (a new version of P2 can not be generated until its previous version is consumed).

In the example, a distance of 0 is always specified between a data transfer operation and its predecessors or successors because a data transfer can be accomplished by forwarding through a direct link in data path. Besides the data dependency, we also need to specify control dependency. For a caller program, it needs at least 1 state before the subroutine call as the calling state and 1 state after the call as the return state for control transfer. Also, if there are multiple calls in a subprogram, they must be separated by 1 cycle. The control dependency is specified by introducing dummy nodes and dot edges as in figure 4(c).

Based on the control and data flow graph (CDFG), a list scheduling based algorithm is performed to partition the operations into disjoint states. Figure 4(d) shows that scheduled result of the example. In this example, o_8 is scheduled across the subroutine. Also, a lot of operations are performed by chaining because their accumulated propagation delay is within the clock cycle.

D. Scheduled code optimization

In a scheduled CDFG, if a data transfer operation and all of its immediate successors are scheduled in a same state, then we can replace the data transfer operation by propagating the input of the operation to all of its successors and then removing the operation (because it is not used thereafter). As illustrated in figure 5, operations o_2 (B:=I2) and o_4 (P2:=B) in state M0 are replaced with P2:=I2 with the same functionality. Similarly, the set of operations { o_1 A:=I1, o_8 E:=A+1, o_3 P1:=A} is replaced with {E:=I1+1, P1:=I1}. Likewise, if the only successor of an operation is a data transfer operation and they are scheduled to a same state, they can be merged. As an example, in state M1, o_7 D:=C+1and o_9 P1:=D are first replaced with P1:=C+1. It is then merged with o_6 C:=P3, which results in a P1:=P3+1 in the same state. The optimized CDFG is shown in figure 4 (e). In this example, the 11 operations in the main program is reduced to 6 operations. This technique is very important to a control subroutine implementation because a lot of parameter passing operations have been introduced in the source transformation phase. At this time, the result can be described as a VHDL in behavioral state machine level [4] (or finite-state machine with a data path, FSMD). Figure 4(f) shows the kernel of the FSMD description. Notice that it contains 6 states with push and pop signals are inserted at proper states.

M0	o_2 (B:=I2) o_4 (P2:=B)	P2:=I2	

	o_1 A:=I1
	o_8 E:=A+1
	o_3 P1:=A
		E:=I1+1,
		P1:=I1	

M1	o_6 C:=P3
	o_7 D:=C+1
	o_9 P1:=D
		C:=P3
		P1:=C+1
			P1:=P3+1

M2	o_{12} (F:=P3) o_{13} (O1<=F+3)	O1:=P3+3	

Figure 5. Scheduled code optimization.

E. Register Binding

In the control subroutine implementation of a subprogram, the value of a variable defined in main program must be retained during the period of subroutine execution. Instead of allocating dedicated registers for each subroutine as in [7], our method tries to share registers among all subprograms. For this purpose, a variable lifetime table is created as shown in figure 4(g). The table is global in the sense that it includes all the variables and spreads across all the execution cycles. Based on the lifetime table, a storage compatible graph is constructed, G(V,E), where V is the set of variables, E is the set of edges. There is an edge between two variables v_i and v_j if v_i and v_j do not overlap their lifetimes. The register binding problem is then solved by a clique partitioning algorithm[13]. In this example, the 9 variables and 2 constants are partitioned into 5 cliques {E}, {I1, P1, XX, O1}, {I2, P2, YY, P3}, {c1}, {c3} and a register (or ROM) is allocated for each clique. In comparison with [7] where 14 registers is used for the same example, we use only 5 registers, which is a significant improvement.

F. Functional-Unit and Interconnection Binding

The most important advantage of control subroutine implementation is that for each operation in a subroutine, we perform functional-unit binding just once. Recall that in an inline approach, if a subroutine is called n times, every operation of it has to appear *n* times in the main program. In the synthesis process, every instance of an operation may be scheduled in any order for maximizing performance, so every instance of a variable may be bound to a different register to maximize register sharing and every instance of an operation may be bound to a different functional unit. Consequently, a lot of wires connecting the various registers and functional units have to be introduced in order to execute the instances of the operations. Therefore, the data path becomes more complicated, and the clock cycle time becomes longer. Furthermore, the controller can become bigger because more control signals must be generated. Sometimes, it becomes the bottleneck of a design because random logic is much more difficult to synthesize.

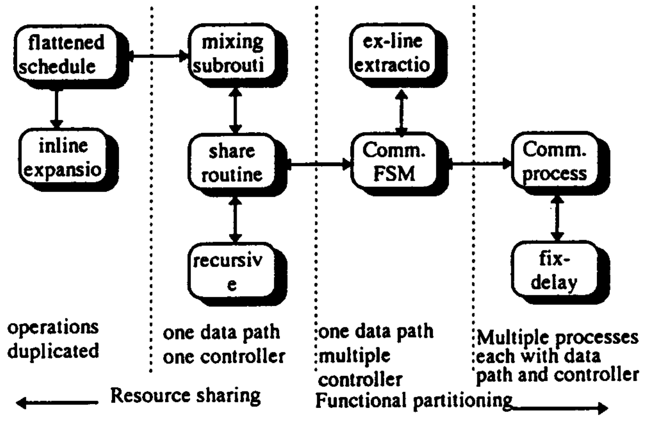


Figure 6. The relationships among subroutine synthesis methods.

IV. Variations of the Control Subroutine

In this section, we will present three enhancements of the control subroutine approach. In Section 4.1, the synthesis of a recursive program is discussed. Section 4.2 presents a behavioral partitioning methodology which divides the controller into several communicating state machines. In Section 4.3, the methodology of mixing the execution of subprograms is studied. These features either increase the range of applications or improve the quality of a design. We also relate the proposed methods with well known approaches. The result of these methods vary from a signal data path and controller (inline) to multiple processes each with data path and controller (module/macro) as illustrated in figure 6.

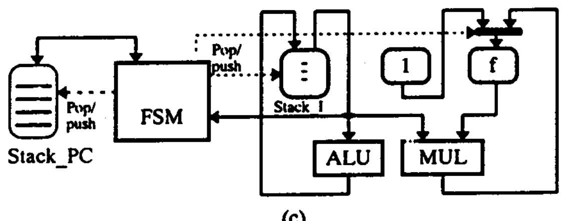
A. Recursive Programs

Control subroutine is the only method that can handle recursive call. If the program is a recursive one, then we have to stack the instances of each variable in addition to the states of the controller. There are two kinds of recursion: direct or indirect. A direct recursion has a loop of length 1 in the call sequence graph. An indirect one consists of cycles longer than 1. Figure 7 gives an example of a direct recursive program described in behavioral (figure 7(a)). The example computes the factorial of a given input I. It is translated to a FSMD in figure 7(b). Figure 7(c) shows the architecture design that we proposed. In this example, variable I is live across the state boundary of the recursive call (S1). Therefore, a stack, instead of a register, is allocated to save its previous versions. An instance of the variable is pushed into the stack when control flow of the program reaches the loop at the end of S1. In general, if the program contains a cycle of length *L* and the iterations of the cycle is *I*, then the depth of each data stack is *I* and the depth of program stack is *L*I*. A designer should try to describe a design with a non-recursive one to reduce the hardware cost. For example, the same function can be described as a for-loop as illustrated in Figure 7(d).

```
Factorial(I)
{ if I=1 return (1)
  else return(I*factorial(I-1))
}
```
(a)

```
f:=1;
for (p=I to 1, --p)
  f = f*p ;
```
(d)

```
case next_state is
when S0 => if (I=1) {
               next_state := stack_PC;
               pop_PC; pop_I;}
             else next_state:=S1;
when S1=> push_PC(S2),push_I;
             I=I-1,next_state :=S0
when S2=> f:=f*I;
             next_state:=stack_PC;
             pop_PC; pop_I;
end case;
```
(b)



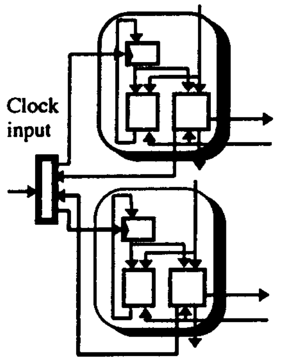
(c)

Figure 7. Compute factorial, (a) a recursive description, (b) FSMD description with push and pop inserted, (c) the architecture, (d) a non-recursive description.

B. Behavioral Partitioning of the Controller

Given a scheduled CDFG as in figure 4.(e), the realization of the controller has two scenarios. The first leaves the flexibility to a logic synthesizer and will produce a mixed controller; while the others force them to be disjointed. In this subsection, the control of each subroutine is synthesized individually as a FSM that has a state register, next-state logic and output-logic of its own (figure 8(a) and 8(b)). A special hardware, named arbitrator, is used to control over the execution of all FSMs and is used to select appropriate control signals for the data path. The arbitrator supervises the execution of FSMs by controlling their clock inputs. The privilege is granted clockwise. At any cycle, only one state machine gets clock pulse. It then proceeds to the next state and has the right of control over the data path. Figure 8 (c) shows a snapshot of the given example. At the initial state (cycle 0), both the main and the subroutine are in their last states (M2 and S2). When a program is just started, the arbitrator (GR) is set to M which grants the main program starting execution in the first cycle. By the end of the first cycle, the man sets the GR as S. In the next cycle, the subroutine awakes up, it proceeds to S0 and continues in the success cycles. By the end of cycle 4, it sets GR as M which resumes the main. At the next cycle, the subprogram halts at S2 while the main program proceeds to M1.

The synthesis flow of the method is identical to that of control subroutine. The data path will be the same. However, a controller consists of smaller ones has a lot of advantages. First of all, each FSM takes a smaller state inputs and takes only related inputs from data path or outside word. All the signals that do not appear in this subprogram is treated as don't cares. This would greatly reduce the complexity of each state machine. Second, since the delay time of the controller is the maximum of each FSM, it has a much smaller propagation delay as compare to the global FSM approach. This would become more significant for a control dominated circuits. Finally, the method actually implements a low power controller because there is no circuit switching for those unauthorized FSM, and hence consumes no dynamic power dissipation. The above facts also encourage of dividing a large program into smaller pieces, each forms a subprogram. This is similar to the idea of ex-line in [14].



(a)

main			Sum of Squares		
PS	NS	NB	PS	NS	NB
M0	M1	S	S0	S1	S
M1	M2	S	S1	S2	S
M2	M0	M	S2	S0	M

(b)

cycle	GR	main			Sum Of Squares		
		PS	NS	NB	PS	NS	NB
0	M	M2	M0	M	S2	S0	M
1	M	M0	M1	S	S2	S0	M
2	S	M0	M1	S	S0	S1	S
3	S	M0	M1	S	S1	S2	S
4	S	M0	M1	S	S2	S0	M
5	M	M1	M2	M	S2	S0	M

(c)

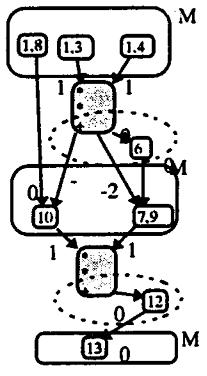
Figure 8. Behavioral partitioning of the controller, (a) arbitrator and state machines, (b) state table for main and Sum of squares, (c) snapshot of execution. GR: status of the arbitrator, PS: present state, NS: next state, NB: next block.

Another close related approach is the communicating processes for variable-delay subroutine in which a dedicated hardware is allocated for each procedure. The proposed method differs from the other in the following ways, (1) a single data path is shared among all subroutines (reduce area), (2) being embedded onto a same data path, parameters can be passing directly through forwarding, while they have to be transmitted explicitly between processes (reduce state overhead), (3) control transfer is completed by a centralized arbitrator instead of by processes handshaking (more efficient). Nevertheless, if a process performs a fixed delay behavioral, the drawbacks raised in (2) and (3) can be eliminated.

C. Mixing the operations of subprograms

If any operation has been scheduled concurrently with the macro during the scheduling of a CDFG, the FSMD of a previously scheduled subroutine should be modified to include these new operations. As an example, figure 9 (b) shows the FSMD of the scheduled CDFG in figure 9(a) where o_6 and o_{12} have been merged to S2 with a case construction. Now, the FSMD will have to look at the stack to decide which conditional operations should be executed. The presence of conditional execution operations doesn't increase the complexity of data path. However, the control becomes more complicated because it has to take both state-register and stack as inputs. To achieve the best performance while keeping a low overhead, a post-optimization algorithm is used to reduce the number of operations scheduled concurrently with the subroutine body.

Given the result in figure 9(a), we may interpret it in another way as shown in figure 9(c) where the scheduled CDFG is kept flatten instead of merged into a control-subroutine. A flatten result has many advantages: (1) we do not have to execute the operations by conditions, (2) the control transfer states can be omitted, (3) the resulting data path could stay as simple as that of control-subroutine since we could bind every instances of an operations to the same resource. In comparison, a control-subroutine takes less states, but its circuit is more complicated in each state; on the other hand, the flatten one takes more states but is simpler in each state. The merits of each other won't be clear until gate level design is completed. This method is similar to the concept of behavioral template in [10].



(a)

```
IF (clock'EVENT and clock='1')
THEN
CASE next_state IS
WHEN M0=> P1:=I1;P2:=I2;E:=I1+1;
        push; next_state:=S0;
WHEN M1=> P1:=C+1;P2:=E;
        push; next_state:=S0;
WHEN M2=>O1<=F+3
        next_state := M0;
WHEN S0=> XX:=P1*P1;
        next_state := S1;
WHEN S1=> YY:=P2*P2;
        next_state := S2;
WHEN S2=> P3:=XX+YY;
        CASE stack IS
        WHEN M1 => C:=P3;
        WHEN M2 => F:=P3;
        WHEN others =>
        END CASE;
        pop; next_state := stack;
END CASE;
```

(b)

```
IF (clock'EVENT and clock = '1')
THEN
CASE next_state IS
WHEN M0=>
                P1:=I1;P2:=I2;E:=I1+
                1;
                next_state:=S00;
WHEN S00=> XX:=P1*P1;
                next_state := S01;
WHEN S01=> YY:=P2*P2;
                next_state := S02;
WHEN S02=> P3:=XX+YY; C:=P3;
                next_state:=M1;
WHEN M1=> P1:=C+1;P2:=E;
                next_state:=S10;
WHEN S10=> XX:=P1*P1;
                next_state := S11;
WHEN S11=> YY:=P2*P2;
                next_state := S12;
WHEN S12=> P3:=XX+YY; F:=P3;
                next_state:=M1;
WHEN M2=>O1<=F+3
                next_state := M0;
END CASE;
```

(c)

Figure 9. Mixing the operations of subprograms, (a) a scheduled CDFG, (b) the FSMD description, (c) an alternative design description.

D. Overall strategy of the control subroutine approach

To adapt the various generations presented above, the scheduling phase of a control-subroutine approach described in figure 3 is modified. As shown in figure 10, we start with a scheduling that takes data dependency only. If the result can not be realized as a control-subroutine, a flatten code is reported as one of the optimal solution (F1). Then, the control dependency is included and mixed operations are minimized (C1). Finally, we restrict any operation to be scheduled with the subroutine, which produces a pure control-subroutine (C2).

```
Schedule the subroutine with data dependency only
if (control transfer states do not exist)
        {report Flatten-code F1;
        reschedule the subroutine with control dependency
included; }
reduce mixed operations;
report Control-subroutine C1;
if (any regular operation mixed with subroutine)
        { re-schedule with no mixed operation allowed;
        report pure Control-subroutine C2;}
}
Data path generation and controller synthesis (F1,C1,C2).
```

Figure 10. The overall strategy of the control subroutine approach.

The strategy above will produce at most three solutions, namely, (1) flatten schedule, (2) control-subroutine with mixed operations, and (3) control-subroutine without mixed operations, for further synthesis. Obviously, as more constraints are enforced on a schedule, it will costs more states, but its control circuit could be simpler. Thus, each of them is treated as a local optimal solution. The behavioral partitioning technique in Section 4.2 can apply to cases (2) and (3) for controller optimization.

V. EXPERIMENTS

The proposed methods are integrated into a behavioral synthesis system called MEBS [12]. The MEBS contains a serial of synthesis tools. As design enters at the algorithmic level, the scheduling tools will convert it into behavioral state machine level, then the allocation tools convert it into register transfer level, and finally the logic synthesis tools convert it into a gate level design. We use MEBS directive to specify the implementation method. For the inline expansion and process module approach, a source to source transformation converts the program into processes at the same level. The control-subroutine approach is realized according to the algorithm in figure 10 and figure 3. Two examples are chosen for demonstrating the performance of each method.

A. Result on the illustrating example

TABLES 1 RESULT ON SUM OF SQUARES

	+	*	states in (main,sub)	execution cycles	Reg	links
Fixed-delay module	2	1	(7,2)	7	7	-
Inline expansion	1	1	7	7	6	-
Control subroutine	1	1	(5,3)	11	9	-
Fixed-delay module	2	1	(8,3)	8	(4,2)	(9,7)
Inline expansion	1	1	7	7	5	14
Flatten	1	1	7	7	5	14
Control-subroutine	1	1	(3,3)	9	5	15

This example is proposed in [7] and is used as an illustrating example in this paper. In each experiment, a minimum resource is allocated. The table is consisted of 2 parts. The first part shows the results of [7], while the second part shows our results. Some observations are explained below.

1. In each methodology, our implementation achieves better result than [7] except the case using fixed-delay module, this is due to they use a module of delay 2, while we use a slower (but cheaper) module of delay 3.
2. According to figure 10, our method produces 2 optimal solutions, the flatten one takes 7 cycles, while the control-subroutine one takes 9 cycles. Further synthesis shows that the result of the flatten approach achieves better design with little control overhead (7 states v.s. 6 states).

B. Results on the modified example

```
XX:=X*X;
XX:=XX+1;
Y:=Y+1;
YY:=Y*Y;
Z:=XX+YY;
```

Fig. 11. The modified subroutine.

In this example, the subroutine, Sum Of Squares, is replaced with a more rich one (figure 11). Table 2 summarizes the results. Again, our method produces 2 optimal results, flatten schedule and pure control-subroutine. Although they take the same cycles (9) to perform, the control-subroutine needs only 6 states in the controller. The difference will increase as the number of subroutine calls increases.

TABLES 2 RESULT ON MODIFIED EXAMPLE.

	+	*	states in (main,sub)	execution cycles	Reg	links
Fixed-delay module	2	1	(9,3)	9	(4,2)	(10,7)
Inline expansion	1	1	9	9	5	16
Flatten	1	1	9	9	5	15
Control-subroutine	1	1	(3,3)	9	5	15

The above examples show that inline (flatten) based approach is good for a small case, while the control-subroutine approach is able to control the complexity of data path and control path with acceptable overhead (less than 2 cycles). Synthesis using fixed-delay module always produce a design with more objects (functional units, registers, and links); however, it may have the advantage of locality.

VI. CONCLUSION

In this paper, the control-subroutine implementation of a subprogram as well as its variation are extensively studied. A lot of optimization possibility is explored during the process. Our study shows that choosing a procedure implementation style has significant impact on design quality. Unfortunately, there is no simple way to decide which approach will produce a better design. Therefore, we proposed a methodology which produces several solutions to be evaluated by a low level synthesis tool. And finally a best result is selected for implementation. Being able to support various implementation styles, integrate methodologies, and easily link to a lower level design tools, the proposed methodology is flexible enough to adapt various applications.

REFERENCES

[1] P. Gutberlet, W. Rosenstiel, "Specification of Interface Components for Synchronous Data Paths", Proceedings of the International Workshop on High Level Synthesis, 1993.

[2] V. Nagasamy, N. Berry, C. Dangelo, "Specification, Planning and Synthesis in a VHDL design environment", IEEE Design \& Test of Computers, Jun. 1992.

[3] J. W. Davidson, A.M. Holler, "Subprogram Inlining: a Study of its Effects on Program Execution Time", IEEE Transactions on Software Engineering, Feb. 1992.

[4] Y.C. Hsu, F.S. Tsai, T.Y. Liu and S.Z. Lin, "VHDL Modeling for Digital Design Synthesis", Kluwer Academic Publishers, 1995.

[5] "IEEE Standard VHDL Language Reference Manual", IEEE Std 1076-1987, Mar. 1988.

[6] R. Camposano, L.F. Saunders, R.M. Tabet, "VHDL as Input for High-Level Synthesis" IEEE Design & Test of Computers, Mar. 1991.

[7] L. Ramachandran, S. Narayan, F. Vahid and D. Gajski, "Synthesis of Functions and Procedures in Behavioral VHDL", Proceedings of the EuroDAC/EuroVHDL, Hamburg, 1993.

[8] R. Walker and D.E. Thomas, "Behavioral transformation for algorithmic level IC design", IEEE Transactions on Computer-Aided Design, Oct. 1989.

[9] R. Camposano and J. Eijndhoven, "Partitioning a Design in Structural Synthesis", Proceedings of the ICCD, 1987.

[10] Tai Ly, David Knapp, Ron Miller, Don MacMillen, "Scheduling using Behavioral Templates", in Proc. of 32th DAC, June 1992.

[11] J. Rajan and D.E. Thomas, "Synthesis by Delayed Binding of Decisions", Proceedings of the DAC, 1985.

[12] Y. C. Hsu, T. Y. Liu, F. S. Tsai, S. Z. Lin, and C. Yu, "Digital Design From Concept to Prototype in Hours", Proceedings of IEEE Asia-Pacific Conference on Circuits and Systems, Dec., 1994.

[13] C.J. Tseng and D.P. Siewiorek, "Automated Synthesis of Data Paths on Digital Systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. CAD-5, no.3, pp.379-395, July 1986.

[14] Frank Vahid, "Procedure exlining: a transformation for improved system and behavior synthesis", 8th international symposium on system synthesis, pp. 84-89, 1995.