High-level Scheduling Model and Control Synthesis for a Broad Range of Design Applications

Chih-Tung Chen and Kayhan Küçükçakar
Unified Design System Laboratory
Motorola, Inc.
E-mail: {chen,kayhan}@adtaz.sps.mot.com

Abstract

This paper presents a versatile scheduling model and an efficient control synthesis methodology which enables architectural (high-level) design/synthesis systems to seamlessly support a broad range of architectural design applications from datapath-dominated digital signal processing (DSP) to micro-processors/ controllers and control-dominated peripherals, utilizing multi-phase clocking schemes, multiple threading, data-dependent delays, pipelining, and combinations of the above. The work presented in this paper is an enabling technology for high-level synthesis to go beyond traditional datapath-dominated DSP applications and to start becoming a viable and cost-effective design methodology for commodity ICs such as micro-processors/ controllers and control-dominated peripherals.

1. Introduction

High-level synthesis (HLS) offers a methodology which promises a significant productivity increase by raising the abstraction level of digital design. A tutorial on HLS methodology and past research can be found in [1] and [2]. After close to two decades of research, commercial HLS tools have been introduced recently [3] and have seen some production use. But, it is not yet known how soon HLS will become a mainstream technology for production designs.

In a cost and benefit study of the HLS methodology conducted in Motorola, several challenges to HLS tools were found such as offering significantly better cycle-time and defect reduction than RTL/logic synthesis while incurring no deterioration of design quality, providing interactive and incremental design process for better user control, and most importantly supporting a broad range of architecture design applications including datapath-dominated designs like DSP, processor-type designs like micro/DSP/ co-processors, and control-dominated designs like peripherals.

The demand of supporting a broad range of design applications is due to the fact that adopting the HLS methodology is a drastic design paradigm shift for designers as well as design organizations. It requires considerable investment from them in training and time to incorporate HLS into the existing design flows and to become mature enough for production design, verification, and optimization throughout the new design flows. Hence, without supporting a broad range of design applications, the return of the investment on a domain-specific HLS methodology becomes sporadic and often unjustified.

To accelerate the industrial adoption of HLS, we have developed *Matisse* [4], a versatile architectural design tool, which answer many challenges described earlier. It is beyond the scope of this paper to describe the differences of Matisse from traditional HLS. This paper addresses two of the important issues; namely, the *scheduling model* and *control synthesis*. Our contribution is that the proposed approaches are versatile and capable of uniformly specifying and implementing static, dynamic, and pipeline scheduling with considerations of clocking and synchronization, which are essential for the successful application of HLS to a broad range of designs in the commodity ICs. Additionally, several unique techniques utilized in our control synthesis are presented, including clocking-driven coarse-grain control partitioning, FSM optimization, and control signal merging via default values. These techniques improve the efficiency of the synthesized controllers and are found to be crucial for commodity IC applications.

In the remainder of this paper, we begin with the related work in Section 2. Our scheduling model and the supported clocking scheme are illustrated in Section 3. Section 4 describes the control synthesis and its unique techniques in detail. In Section 5, we present results obtained from industrial designs including a case study on the effectiveness of FSM optimization and a power-consumption trade-off study of single-clock versus multi-phase clock designs. Finally, we conclude this paper with some future directions in Section 6.

2. Related Work

Static non-pipeline scheduling model and the associated control synthesis have been the subjects of intensive high-level synthesis research in last two decades. Park reported a pipeline scheduling method for synthesis of pipelines from behavioral specifications [5]. The generation of the control path for Park's pipeline scheduling model was subsequently addressed by Weng [6]. Girczyc applied the concept of functional pipelining similar to Park's to pipeline individual loops (loop winding) [7]. Prabhu described a pipeline synthesis system which utilizes a pipelined controller in junction with functional pipelining in the data path for improving the design throughput [8].

In the area of dynamic scheduling, relative scheduling was introduced by Ku to address scheduling with unbounded-delay operations [9]. Additionally, they used a relative control synthesis approach to generate an interconnection of interacting FSMs.

Papachristou et. al proposed a multi-phase clocking architecture model and technique for synthesis of low power RTL implementations [10]. However, the associated control synthesis was not addressed.

To our best knowledge, no prior scheduling model and control synthesis were found to address the issues of static, dynamic, and

0-89791-993-9/97 $10.00 © 1997 IEEE

pipeline scheduling along with single or multi-phase clocking scheme uniformly.

3. Unified Scheduling Model

The scheduling model in this context is regarded as the specification method and representation of the scheduling information and **not** the scheduling algorithms.

In a study of various industrial designs with designers, we created a list of HLS design modeling techniques required to support the majority of designs for datapath-dominated, control-dominated and processor-type applications. As listed in Table 1, they are (1) conditional branching and looping, (2) operation chaining, (3) multi-cycle delays, (4) data-dependent delays, (5) multi-thread execution, (6) pipelining, and (7) multi-phase clocking.

Table 1: Frequent HLS modeling techniques.

Category	1	2	3	4	5	6	7
datapath dominated	X	X	X			X	
control dominated	X	X	X	X	X		X
processor type	X	X	X	X		X	X

In this section, we will present a hierarchical scheduling model for uniformly specifying all the above styles of execution and sequencing. The novelty of our scheduling model is that the specification of clocking and synchronization is no longer global. They are specified locally at each schedulable entity (operation or statement) along with typical attributes like offset and delay.

3.1. Clocking schemes

The supported clocking scheme can include a single clock (master clock), multiple phase clocks, and a mixture of a master clock and one or more phase clocks. As shown in Figure 1, when phase clocks are used, they are required to be synchronous and have the same period called the primary cycle. The primary cycle is further divided into a number of pickets. Each phase clock is restricted to have one active pulse per primary cycle. The duration and active level (high or low) of the active pulse are specified at the picket level. If a master clock is used together with phase clocks, the primary cycle is a multiple of master cycles for the purpose of synchronization.

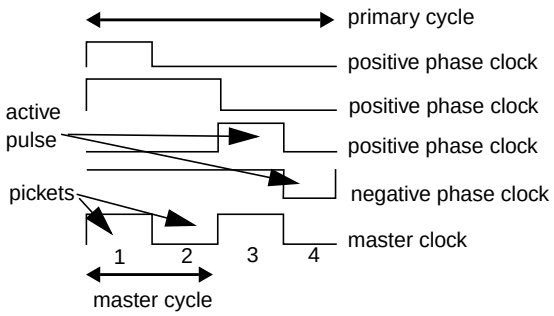


Figure 1. Supported clocking schemes

3.2. Hierarchical scheduling

The procedural HDL description can contain control-flow constructs such as branching and iteration. The control flow specification is modeled through the hierarchy of our scheduling model. The hierarchy is defined by the compound statements such as `if`, `case` and `while`. The generic entity in the hierarchy is called a *statement block* which comprises of a list of statements to be scheduled together according to their data flow and user (serialization) dependencies. Each compound statement in a statement block creates another level of the hierarchy with one or more statement blocks corresponding to the branching bodies or the loop body. Figure 2 shows an example of a behavioral description in Verilog HDL where the branching statement `if (cond)` further links to the two statement blocks of its `then` and `else` bodies at the next level. Note that the offsets of the `then` and `else` blocks shown in Figure 2 are relative to their parent `if` statement.

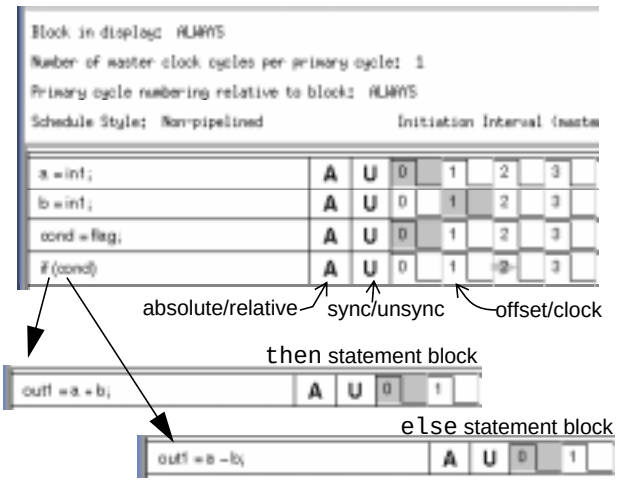


Figure 2. Example of hierarchical scheduling

The semantics of the hierarchical scheduling model is recursively defined as follows. When a statement block is activated, the statements in that statement block execute according their scheduling specifications. The execution of a compound statement involves activating zero or more of the associated statement blocks and is complete when all the activated statement blocks are complete. An activated statement block is complete when the execution of all its statements are complete.

3.3. Intra-block scheduling

Let S be a statement block which comprises of a list of statements. The intra-block scheduling model defines the execution of each statement s_i in S by a tuple of attributes (p_i, y_i, o_i, c_i, d_i); namely reference point p_i, phase synchronization mode y_i, offset o_i, clock waveform c_i, and delay d_i.

Reference point. The p_i attribute defines the reference time $T(p_i)$ to which s_i is scheduled relatively, and is a subset of S such that $\forall s_j \in p_i$, s_i has a dependency on s_j and s_j is a compound statement with a non-deterministic or unbounded delay $D(s_j)$. Formally, $T(p_i)$ can be expressed as:

$$T(p_i) = max(T(S), T(s_j) + D(s_j)\big|_{s_j \in p_i})$$

where $T(S)$ is the activation time of statement block S, $T(s_j)$ is the start time of s_j, and $D(s_j)$ is the duration of s_j.

If s_i doesn't depend on any compound statement within statement block S, i.e. p_i is an empty set, $T(p_i)$ is equal to $T(S)$; in other words, s_i is scheduled with respect to the beginning of S. The latter case is called absolute scheduling in Matisse, whereas the opposite is relative scheduling.

Synchronization mode. The y_i attribute is a boolean flag indicating whether or not to use phase synchronization and is only applicable to multi-phase clock scheduling. When y_i is true, the reference time to which s_i is scheduled is deferred to $T'(p_i)$ which

is the beginning of the next primary cycle after the original reference time $T(p_i)$. This attribute is mostly used to a situation where the statements (events) E scheduled with different phase clocks need to be executed according to the phase order; e.g., a phase-2 event always follows a phase-1 event. Typically, the statements in E have a common reference point p_E. If $T(p_E)$ occurs in a middle of a primary cycle as shown in Figure 3, by setting both $y(s_i)$ and $y(s_j)$ to true, s_i and s_j have a synchronized reference time $T'(p_E)$ after which the order of the clock phases are known statically.

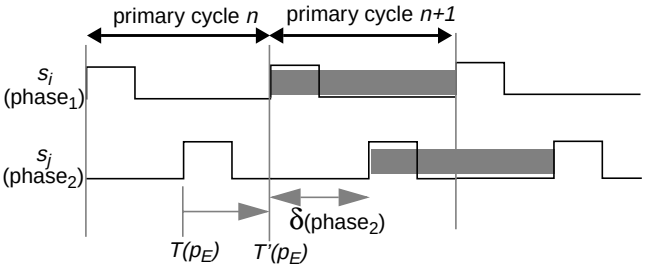


Figure 3. Example of phase synchronization

Offset, clock and delay. The o_i attribute is an integer value $\in Z^{0,+}$ for specifying the offset in cycles of the selected clock c_i after the reference time $T(p_i)$ or $T'(p_i)$. By giving the c_i (clock) attribute locally to each statement (operation), multi-phase clock scheduling and synchronization are enabled in our model. Finally, the d_i (delay) attribute is also an integer value and used for specifying multi-cycle operations

Start time calculation. Table 2 expresses the start time $T(s_i)$ under three scenarios; namely, phase unsynchronized with a zero offset, phase unsynchronized with a non-zero offset, and phase synchronized.

Table 2: Statement start times

y_i	o_i	$T(s_i)$
0	0	$T(p_i)$
0	>0	$T(p_i) - \delta(p_i) + o_i * period(c_i)$
1	X	$T'(p_i) + \delta(c_i) + o_i * period(c_i)$

The first two rows express $T(s_i)$ without phase synchronization ($y_i=0$). In such case, s_i should start ASAP after the reference point $T(p_i)$. When offset o_i is *0*, $T(s_i)$ is same as $T(p_i)$ as shown in the 1st row. The 2nd row expresses $T(s_i)$ when o_i is not *0*. In such case, s_i will be started at the beginning of the cycle o_i after $T(p_i)$. In this equation, $period(c_i)$ is the cycle length of c_i and $\delta(p_i)$ is called the reference point lapse time. The reference point lapse time $\delta(p_i)$ is defined as the delay from the beginning of the first cycle to the reference time $T(p_i)$. It is used due to the cases where $T(p_i)$ is not inline with the clock edges of c_i (especially in multi-phase clocking designs) as shown in Figure 4. Typically, $\delta(p_i)$ is 0; in other words, the entire first cycle is available. A non-zero

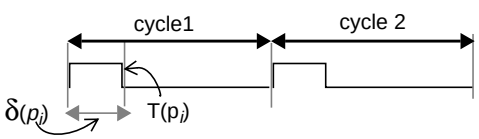


Figure 4. Example of a reference point lapse time

reference point lapse time occurs when statement s_i depends on events which are not synchronized with the clock c_i.

Finally, the 3rd row expresses the start time $T(s_i)$ when phase synchronization is used ($y_i=1$). In such as, $T(s_i)$ becomes the beginning of cycle o_i after $T'(p_i)$, the deferred reference time as shown in Figure 3. Since the clock edges of c_i may not be in line with the primary cycles, a phase lapse time $\delta(c_i)$ is defined as the delay from the beginning of a primary cycle to the beginning of the subsequent clock cycle of c_i, also illustrated in Figure 3.

3.4. Pipeline scheduling

Our scheduling model supports functional and loop pipelining. Functional pipelining [5] is regarded as a form of scheduling under a global *initiation interval*. The initiation interval represents the time interval between two consecutive executions of a scheduled design behavior. Loop pipelining (also called loop folding or winding) is a localized functional pipelining of a loop. The local initiation interval defines the time interval between two consecutive executions of a scheduled loop body. In our model, these two forms of pipelining are modeled the same way since the functional pipelining can be regarded as the loop body of the design process.

In our hierarchical scheduling model, pipeline scheduling can be specified at the top level or to an individual loop. The pipeline execution is implicitly applied to the statement blocks within the scheduling sub-hierarchy derived from the top-level statement block or the loop statement. The following restrictions are also assumed. All the statements within the scheduling sub-hierarchy must use the same clock and absolute scheduling. Additionally, nested loops are not supported in a pipeline.

Pipeline scheduling changes the delay definition of a pipelined loop. Let s_i be a loop statement with n_i iterations and with a loop body LB_i whose latency is $D(LB_i)$. If s_i is not pipelined, $D(s_i)$ is equal to $n_i \times D(LB_i)$. However, if s_i is pipelined with an initiation interval I_i, $D(s_i)$ becomes $(n_i\text{-}1)\times I_i + D(LB_i)$ as shown in Figure 5 where $(n_i\text{-}1)\times I_i$ is the time it takes to initiate the last iteration.

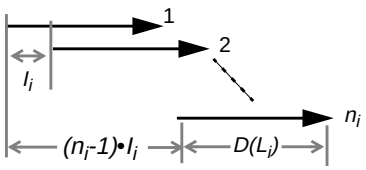


Figure 5. Loop pipelining

Within the pipelined scheduling sub-hierarchy, each statement is associated with an additional pipeline start time. Let s_j be a statement in LB_i, the pipeline start time $T^p(s_j)$ is equal to $(T(s_j)\text{-}T(s_i)) \% I_i$ where % is a modulus operator. During pipeline execution, s_j will be executed periodically at $m \times I_i + T^p(s_j)$ where $m \in Z^{0,+}$ until s_i (the loop) is complete. Note that some statement in LB_i may execute more than n_i times in a pipelined loop s_i. The extra executions occur during pipeline filling and/or flushing.

4. Unified Control Synthesis

Control synthesis for HLS is a process of generating a control unit which will drive the data path as required by the schedule and using the values derived from the resource/interconnect allocation. A tutorial on control synthesis for HLS can be found in [11].

As stated earlier, control synthesis in Matisse is required to support static/dynamic scheduling, single/multi-phase clocking, multi-thread executions, and loop pipelining seamlessly and uniformly. Additional critical requirements are that no extra cycles are introduced for nested branching and loop entry/exit unless

explicitly specified and that conditional blocks or while loops can be skipped with combinational delay (zero cycle skip). To achieve this aggressive goal, we developed a template-driven hierarchical control synthesis with an additional optimization step.

4.1. Hierarchical control model

The multitude of scheduling styles to be supported, compounded with multi-phase clocking schemes, makes the traditional flat and single-FSM control model ineffective. We chose to utilize a hierarchical control model which somewhat resembles to our hierarchical scheduling model. Figure 6 shows that the hierarchical control model is an interconnection of hierarchical control blocks, each of which is responsible for controlling a statement block in our scheduling model. Each control block CB_i has a pair of hand-shaking signals `start`$_i$ and `done`$_i$ for transferring control flow to and from its parent control block.

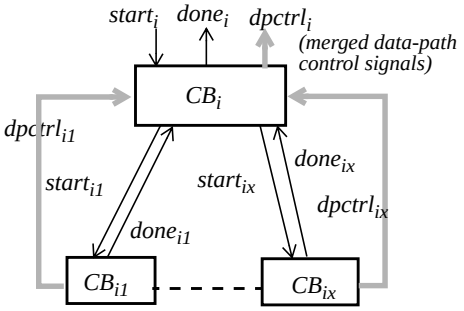


Figure 6. Example of hierarchical control model

4.2. Control block

As shown in Figure 7, a control block CB_i further comprises an interconnection of one or more control elements $\{CE_{i,1},\ldots, CE_{i,n}\}$ and three types of peripheral circuits: *B*, *D*, and *M*.

Coarse-grain FSM partitioning. The responsibility for executing the statements in the statement block of CB_i according to the specified schedule are divided among control elements $\{CE_{i,1},\ldots, CE_{i,n}\}$, where the core of each control element $CE_{i,j}$ is an FSM. A coarse-grain partitioning is performed to divide the statements into groups $\{G_{i,1},\ldots, G_{i,n}\}$ such that each group of statements $G_{i,j}$ has a common reference point $p_{i,j}$, the same phase synchronization mode $y_{i,j}$, and most importantly the same clock $clk_{i,j}$. As a result, the execution of the statements in each group $G_{i,j}$ only have different offsets with respect to $T(p_{i,j})$ or $T'(p_{i,j})$. Furthermore, the different offsets share the same discrete unit, the clock cycle of $clk_{i,j}$. Therefore, it can be seen easily that a single FSM in control element $CE_{i,j}$ is sufficient to control the execution of statements $G_{i,j}$. The structure of a control element will be discussed later in Section 4.3.

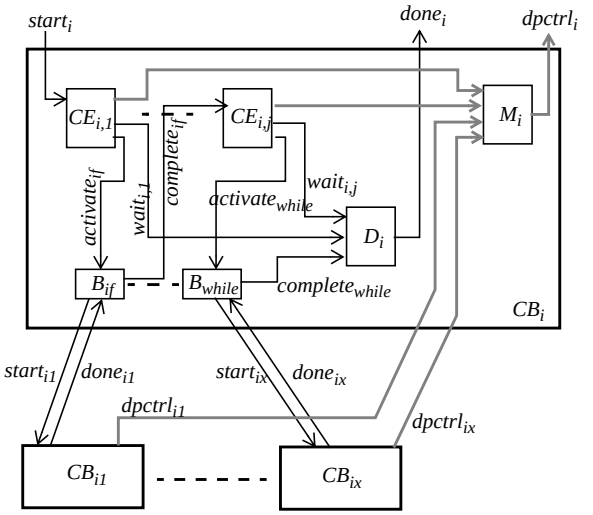


Figure 7. Template of control block

Within control block CB_i, each control element $CE_{i,j}$ receives a list of input signals $P_{i,j}$ required by its reference point $p_{i,j}$. $P_{i,j}$ is either $\{$`start`$_i\}$ if $p_{i,j}$ is empty (absolute scheduling) or $\{$`complete`$_s \mid s \in p_{i,j}\}$. The signal `complete`$_s$ is driven by one of the *B* circuit for signalling the completion of the compound statement *s*. $CE_{i,j}$ begins its execution when each signal in $P_{i,j}$ is raised and subsequently raises its `wait`$_{i,j}$ signal when its execution is finished. The data-path control signals driven by $CE_{i,j}$ are sent to an *M*-type peripheral circuit for merging.

Flow control. *B* circuits are used to implement the control flow of compound statements such as `if`, `case`, and `while`. They are responsible for the `start`/`done` signals between CB_i and its child control blocks and raise the `complete` signals when the execution of the compound statements are finished. For example, let B_c be the *B* circuit for a conditional statement *c* with *K* branches $\{br_1,\ldots, br_K\}$. The input signal of B_c is `activate`$_c$ from a control element $CE_{i,j}$ where $c \in G_{i,j}$. $CE_{i,j}$ raises `activate`$_c$ during the cycle that *c* begins executing. The function of B_c can be expressed as the following logic equations.

$$start_{br_k} = activate_c \cdot cond_{br_k}$$

$$complete_c = \sum_{k=1}^{K} done_{br_k} + activate_c \cdot \prod_{k=1}^{K} \overline{cond_{br_k}}$$

where `cond`$_{br}$ denotes the branching condition of *br*. The first equation implements the branching control by raising the `start` signal of a branch whose condition is true and when *c* begins its execution (`activate`$_c$`=1`). The second equation raises `complete`$_c$ when the selected branch is done or there is no branch to be taken. For a loop statement *l*, the `start`$_{lb}$ signal driven by B_l becomes the iteration control of the loop body *lb*. For example, let *l* be a `while` loop. One way to implement B_l is as follows:

$$start_{lb} = (activate_l + done_{lb}) \cdot cond_l$$

$$complete_l = done_{lb} \cdot \overline{cond_l}$$

Done detection. The *D* circuit D_i in Figure 7 is responsible for raising the `done` signal when the execution of the control block CB_i is finished. Recall from Section 3.2, the completion of CB_i requires the completion of all the statements in its statement block, which is the time that all the control elements $\{CE_{i,1},\ldots, CE_{i,n}\}$ have raised their `wait` signals and all the *B* circuits in CB_i have raised their `complete` signals.

In general, it's not necessary for D_i to depend on all the `wait` and `complete` signals because there often exist some precedence relationships among these signals. A simple static timing analysis on the schedule of CB_i's statement block can be performed to eliminate those `wait` and `complete` signals that have precedence of other signals. Consequently, the remaining `wait` and `complete` signals are the ones whose assertions can only be determined dynamically. Often, there is only one remaining signal for D_i to depend on. In such cases, D_i becomes a wire connecting the signal directly to `done`$_i$. However, if there are two or more remaining signals, their assertions may not be simultaneous, D_i may need to memorize some input assertions using latches or flip-flops.

Control signal merging. Since we utilize a distributed multi-FSM control model, the data-path control signals with more than one driver need to be merged before connecting to data-path components. As shown in Figure 7, the control signals from the control elements and the child control blocks are merged hierarchically through an *M* circuit. A typical approach for signal merging is to use data selectors such as multiplexers, which requires generating additional data selection logic.

By exploiting the fact that the HLS designs should be free of resource conflicts and by making use of default control values, we have developed an efficient technique to merge control signals via simple logic; hence, improving area and performance. The basic idea is as follows. If all the sources (FSMs) of a data-path control signal send a pre-defined default value whenever there is no specific (active) control value required to performed the scheduled statements, then the data-path control signal will not be asserted by two or more different active control values at any given time. Otherwise, the design is not free of resource conflicts because the associated data-path component will be asked to perform different functions at the same time. Formally, let *cs* be a 1-bit control signal with multiple sources $\{cs_1,\ldots,cs_M\}$ and a default control value $dval_{cs}$. The merging function can be expressed as follows:

$$cs = \sum_{i=1}^{M} cs_i \qquad \text{if } \mathrm{dval_{cs}} = 0$$

$$cs = \prod_{i=1}^{M} cs_i \qquad \text{if } \mathrm{dval_{cs}} = 1$$

If *cs* is a multi-bit signal, the merging can be done by bit-wise *OR*ing or *AND*ing the source signals depending on the default value of each individual bits. Alternatively, if an *n*-bit control signal does not utilize all the possible 2^n values, one can encode the used values first to reduce the number of bits for merging. The encoded control signal can then be decoded just once after the final merging.

4.3. Control element

As discussed in the previous section, each control element $CE_{i,j}$ controls a group of statements $G_{i,j}$ which have a common reference point $p_{i,j}$, the same phase synchronization mode $y_{i,j}$, and most importantly the same clock $clk_{i,j}$. Figure 8 shows that $CE_{i,j}$ consists of a finite-state machine $FSM_{i,j}$, a *R*-type peripheral circuit $R_{i,j}$, and an optional phase synchronization flip flop.

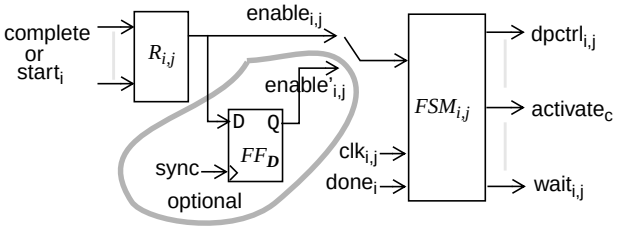


Figure 8. Template of control element

Reference circuit. $R_{i,j}$ is responsible for determining the reference time for $FSM_{i,j}$ according to $p_{i,j}$ and $y_{i,j}$. The circuit behavior of $R_{i,j}$ is similar to the *D*-type peripheral circuit described in Section 4.2. The inputs of $R_{i,j}$ are the `complete` signals of the compound statements in reference point $p_{i,j}$ or `start`$_i$ if $p_{i,j}$ is empty. $R_{i,j}$ raises the `enable`$_{i,j}$ signal to start $FSM_{i,j}$ when all its inputs are asserted. If the number of inputs is greater than one, $R_{i,j}$ contains memory devices such as latches or flip-flops to store their assertions. If $y_{i,j}$ (phase synchronization mode) is true, an additional flip-flop clocked by a `sync` signal as shown in Figure 8 is inserted at the output of $R_{i,j}$ to generate a phase synchronized `enable'`$_{i,j}$. The `sync` signal is typically one of the phase clocks which defines the boundaries of primary cycles; e.g., the phase-1 clock.

Control FSM. $FSM_{i,j}$ is the core of $CE_{i,j}$ which asserts the associated data-path control signals or the `activate` signal of each compound statement in $G_{i,j}$ at the specified offset. $FSM_{i,j}$ is an *L*-state FSM with circular state transitions and clocked with $clk_{i,j}$. *L* is the latency of $G_{i,j}$'s schedule excluding the delays of the compound statements. Figure 9 shows the basic circular state transitions of $FSM_{i,j}$, where the assertions of the data-path control signals or the `activate` signal for each statement *x* in $G_{i,j}$ are assigned to state o_x.

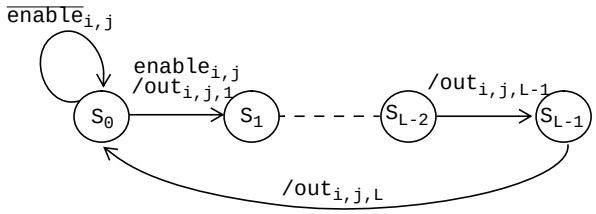


Figure 9. Basic state-transition graph of $FSM_{i,j}$

4.4. Pipeline control block

In our control synthesis, loop pipelining is done in a special pipeline control block which controls the entire loop body. As shown in Figure 10, the external interface of a pipeline control block remains the same as non-pipeline control blocks. Hence, it can co-exist with other control blocks seamlessly. Within a pipeline control block, there are one pipeline control element CE^p_i, one pipeline *D*-type peripheral circuit D^p_i, and zero or more shift registers for storing conditional values.

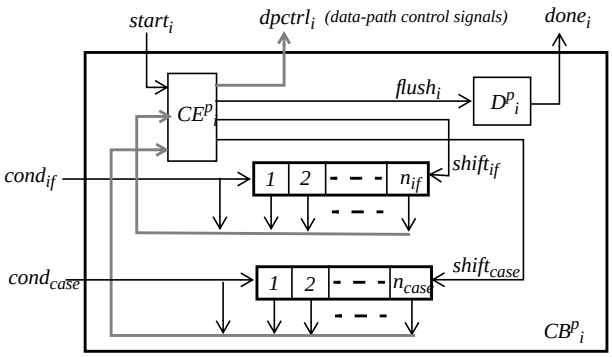


Figure 10. Template of pipelined control block

Pipeline control element. CE^p_i is an FSM with I^c_i states, where I^c_i is the discrete initiation interval. The assertions of each statement *s* within the pipeline scheduling sub-hierarchy is assigned to state $\lfloor T^p(s)/period(clk_i) \rfloor$ where $T^p(s)$ is the pipeline start time as defined in Section 3.4. As with non-pipeline control elements, the state transitions of CE^p_i are also circular.

Conditional assertions. The assertions are conditional if the statement *s* within the pipeline scheduling sub-hierarchy is enclosed by one or more conditional statements. The condition for the assertions is the conjuction of all the branching conditions of the enclosing conditional statements. Due to the nature of pipe-

line execution, if the latency of a conditional statement is longer than initiation interval, multiple instances of a conditional statement will be executed concurrently. However, throughout the execution of each instance, the conditional value for conditional assertions should always be the one at the time the instance is initiated. Hence, we utilize shift registers to store the history of the conditional values.

Let $sreg_c$ be the shift register for a conditional statement c with latency $D(c)$. The number of c instances, n_c, executing concurrently is $\lceil D(c)/I \rceil$. The length of $sreg_c$ is also n_c. $CE^p{}_i$ raises signal `shift`$_c$ at the cycle $\lfloor T^p(c)/period(clk_i) \rfloor$ in which a new c instance is initiated. Raising `shift`$_c$ cause $sreg_c$ to shift once and store a new value of $cond_c$. For each statement s within c, the source of $cond_c$ which s should use for its conditional assertions depends on *it's* offset with respect to the beginning of c. For example, Figure 11 shows the execution of three instances and each of which is divided into three stages of length I. At the beginning of c_1, a new $cond_{c,1}$ is stored in $sreg_c[1]$ and used during its 1st stage execution. By the time c_2 is initiated, c_1 is executing the 2nd stage and $cond_{c,1}$ is now shifted to $sreg_c[2]$. Hence, $sreg_c[2]$ is the source of $cond_c$ used by the 2nd stage execution. Formally, the source of $cond_c$ for each statement s within a conditional statement c can be expressed as $sreg_c[\lceil (T(s)\text{-}T(c))/I \rceil]$.

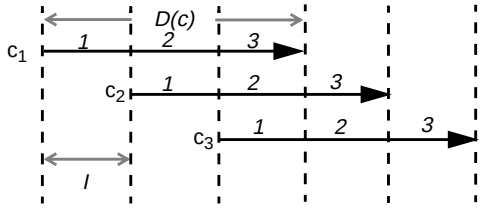


Figure 11. Pipeline instances

Pipeline done. The `done`$_i$ signal driven by the $D^p{}_i$ circuit also becomes more involved than the non-pipeline counterparts. Recall that the fundamental part of our pipeline execution (loop pipelining) is to initiate a new iteration of loop body every initiation interval (see Figure 5) until the loop exit condition is met. However, the pipeline execution is not yet complete at that moment due to the fact that there are still $(\lceil D(L_i))/I \rceil - 1)$ iterations yet to be finished. The process of completing the remaining active iterations is typically called pipeline flushing. In our pipeline control model, $CE^p{}_i$ raises signal `flush`$_i$ at the first pipeline state when `start`$_i$ is dropped and continues the execution of the remaining iterations. $D^p{}_i$ is a delay circuit, such as a counter, which raises `done`$_i$ after a pre-determined number of cycles for the remaining execution, which is $\lceil (D(L_i)\text{-}I)/period(clk_i) \rceil$ cycles.

4.5. Optimization

Although our hierarchical control model with multiple distributed FSMs is extremely flexible and capable of supporting all the its objectives uniformly, we also found that there is room for optimization; in particular, the portions of the design which use static scheduling and a single clock. We developed two types of optimization techniques; namely, logic collapsing and FSM collapsing. The primary goal of both techniques is to collapse a sub-hierarchy of control blocks if possible into a single entity. Thus, both the state registers and the logic of merging data-path control signals can be reduced.

Logic collapsing is a bottom-up technique which recursively collapses stateless control elements, control blocks, and *B*-type circuits (see Section 4.2). Recall from Section 4.3 that each control element $CE_{i,j}$ contains an L-state FSM. $CE_{i,j}$ becomes stateless when L is 1, meaning that the FSM degenerates into a

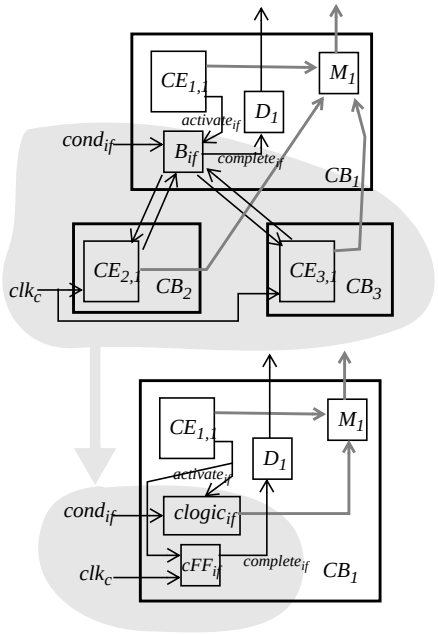


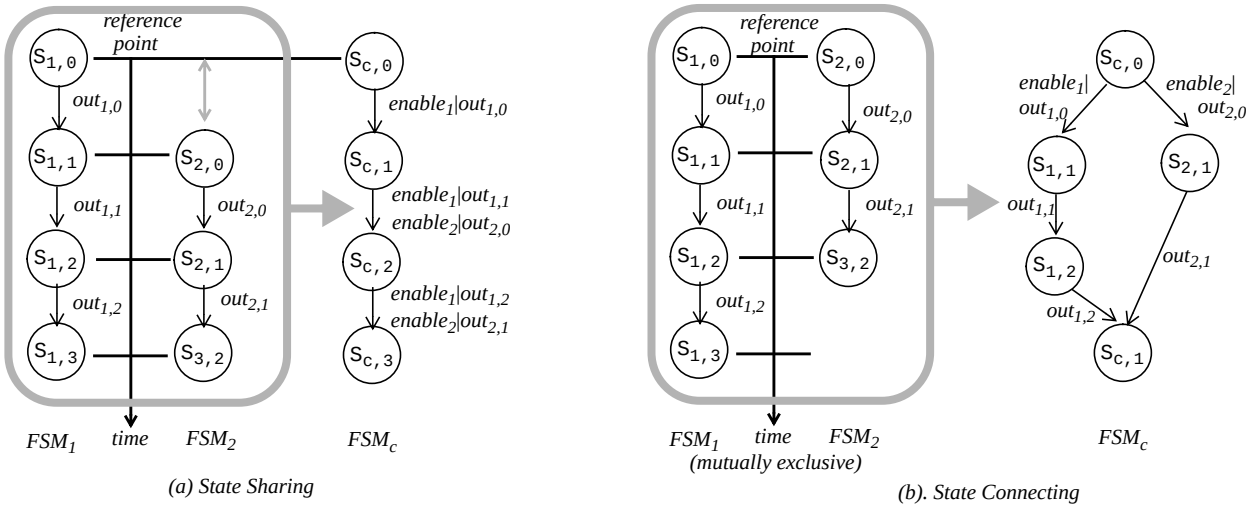
Figure 12. Example of logic collapsing

combinational circuit. A logic collapsible entity denotes an entity whose behavior degenerates into logic functions. Two logic collapsible entities are mutually collapsible if they have the same starting cycle and clocking. Figure 12 shows an example of logic collapsing an `if` statement where both `then` and `else` parts are mutually collapsible.

FSM collapsing is also a bottom-up technique which recursively collapses control elements, control blocks, and B-type circuits which are FSM collapsible. A FSM collapsible entity denotes an entity whose behavior can be implemented by a single-thread FSM. Therefore, control elements, by default, are FSM collapsible. Two FSM collapsible entities are mutually collapsible if they share the same clocking and there exists a common reference point among them. By sharing the same clocking and a common reference point, the execution of two single-thread FSMs are in lock steps; hence, they can be statically merged into one. There are two common ways of merging single-thread FSMs; namely, state sharing and state connecting. State sharing is applicable to two or more single-thread FSMs whose executions can share common starting and ending points. An example of state sharing is shown in Figure 13(a). The result of state sharing is a new single-thread FSM whose state number is equal to the one with more states and the outputs become conditional by the respective FSM `enable` signals. State connecting, on the other hand, is suitable for two or more *mutually exclusive* single-thread FSMs whose executions can share common starting and ending points. Figure 13(b) shows an example of two mutually exclusive single-thread FSMs. State connecting creates two new starting and ending states $S_{c,0}$ and $S_{c,1}$ to replace the ones used by original FSMs. The original mutually exclusive execution is achieved through two conditional state transitions emitting from $S_{c,0}$.

5. Experiments

In this section, we first report the range of design applications that the work described in this paper was used. Following that, a case study of FSM optimization is presented. Finally, the support



(a) State Sharing *(b). State Connecting*

Figure 13. Example of FSM collapsing

of multi-phase clocking schemes is demonstrated with an experiment of single vs. multi-phase clocking on power consumption.

5.1. Coverage of design applications

The scheduling model and control synthesis presented in this paper are implemented in an industrial HLS system, *Matisse*, and has been successfully used in a broad range of industrial designs for evaluation or production. The designs included hard-wired DSPs, an encryption/decryption engine, a bus-to-bus interface, a control-dominated peripheral, and a microcontroller. Table 3 lists some notable designs and their underlying HLS modeling techniques (refer to Table 1 for numbering scheme). So far in all of these projects, the area and performance of the automatically generated controllers were found or estimated by the designers to be comparable to traditional RTL design techniques.

Table 3: Designs and modeling coverage

Design	1	2	3	4	5	6	7
ADPCM predictor	X	X	X			X	
Encryption/decryption	X	X			X		
Bus-to-bus interface	X	X		X	X		X
Vector I/O buffer	X	X		X	X		
FIR filter	X	X	X			X	
8-bit microcontroller	X	X	X	X			

5.2. A case study of FSM optimization

To study the effectiveness of our logic and FSM collapsing techniques, we conducted a case study on 3 designs in Table 3. For each design, we generated a full hierarchical controller and an optimized controller using both logic and FSM collapsing. The generated controllers were synthesized using Synopsys Design Compiler using the same set up, including the library and the performance constraints.

It can be seen from Table 4 that an optimized controller can achieve up to 48% area reduction over an unoptimized one. We also found that the amount of area improvement is design-specific and proportional to the room allowed for optimization.

Table 4: Results of FSM Optimization

Design	Area Reduction
8-bit microcontroller	48%
FIR Filter	37%
ADPCM predictor	18%

5.3. Effects of clocking on power

Since multi-phase clocking schemes are supported in Matisse and Matisse also provides an environment for architectural power optimization [12], we set out to experiment a power saving technique using multi-phase non-overlapping clocks proposed by Papachristou et al [10]. The basic idea is as following. Given a schedule, instead of using traditional resource and interconnect allocation using a single clock with a frequency *f*, their technique utilizes *n* non-overlapping clocks with frequency *f/n* and a datapath architecture with *n* disjoint modules, each running on a distinct clock. Additionally, two special allocation methods were proposed to map each node in the data-flow graph (DFG) into a specific module according to the operation's time step in a modulus fashion. For example, nodes scheduled at time step *t* are mapped to a module running on CLK_k where *k* is *t mod n*. The authors argued that their approach can achieve lower power consumption than traditional HLS designs due to a "capacitance reduction" effect and the reduced frequency. Power savings up to 50% were reported in their paper.

We implemented various designs of the example described in their paper. Due to the limited space of this paper, please refer to [10] for the detailed description of the example and design schematics. Each design, including the controller, synthesized into gates using Synopsys Design Compiler and power was estimated by an accurate gate-level power estimation tool [13]. Table 5 summarizes the experimental results

Circuit1 is a single-clock design with two add/subtract ALUs and used by the authors for comparison with *circuit2* which is a

two-clock design. With a set of random input test vectors, *circuit2* indeed was found to achieve 39% power reduction over *circuit1*. Since circuit1 and circuit2 have different resource allocations and clocking schemes, it was not apparent what the individual effects were on power savings. We found that all architectures generated by the multi-phase clock technique can still be clocked by a single clock with clock gating at registers. Hence, we implemented *circuit3* which has the same architecture as *circuit2* but uses a single clock. As shown in Table 5, by comparing *circuit2* and *circuit3*, the power savings by the multi-phase clock technique alone becomes 23%. Finally, we implemented another two-clock design, *circuit4*, which was given by the authors using their integrated allocation method (Figure 7 in [10]). It was also found to consume less power than single-clock designs, but not better than *circuit2*. Note that the controllers synthesized by our tool for *circuit2* and *circuit4* consist of two FSMs driven by two specified non-overlapping clocks respectively.

Table 5: Effects of clocking on power

Design	Clocking/Arch.	Power	Area	Func. Units
Circuit1	1 clock	1.00	1000	2(+/-)
Circuit2	2 clocks	0.61	1111	2(+), 1(-)
Circuit3	1 clock, same architecture as circuit2	0.77	1115	2(+), 1(-)
Circuit4	2 clocks, integrated allocation version	0.68	1388	2(+), 1(-)

6. Conclusion

We have presented a novel scheduling model and an efficient control synthesis which seamlessly and uniformly support static/dynamic scheduling, single/multi-phase clocking, multi-thread execution, and loop pipelining. Our contribution is an enabling technology for HLS to go beyond traditional datapath-dominated DSP applications and to start becoming a viable and cost-effective design methodology for commodity ICs such as micro-processors/controllers and control-dominated peripherals.

The presented scheduling model and control synthesis have been implemented in an industrial HLS system and used to successfully design many industrial designs for evaluation or production, including digital filters, an encryption/decryption engine, a bus-to-bus interface, a high-speed vector I/O buffer, and a microcontroller. We also showed the effectiveness of our optimization techniques which make our hierarchical control synthesis as efficient as traditional single flat FSM control synthesis for static scheduling while capable of supporting dynamic scheduling and multi-phase clocking schemes. Finally, we conducted a study on a power-saving technique using multi-phase clocking as proposed in [10].

We believe that there are still other opportunities for further optimizing our hierarchical control model. Furthermore, microcoded controllers along with hardwired FSMs should be supported due to the increasing need of architecture reuse and programmable designs.

7. Acknowledgments

We would like to thank Thomas E. Tkacik and Rajesh Gupta for their stimulating discussions during the development of this work and Jie Gong for her extensive review of the paper.

8. References

[1] M. C. McFarland, A. C. Parker and R. Camposano, "The High-Level Synthesis of Digital Systems", *In Proceedings of IEEE*, 78(2):301-318, February 1990.

[2] R. A. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, 1991.

[3] *Behavioral Compiler*, Synopsys, Inc.

[4] *Matisse User Manual*, Motorola, Inc., 1995.

[5] N. Park. and A. C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications", *IEEE Transaction on Computer-Aided Design.* 7(3): 356-370, March 1988.

[6] J.-P. Weng and A. C. Parker, "CSG: control path synthesis in the ADAM system", *In Proc. 6th International Workshop on High-Level Synthesis*, pp. 52-64, 1992.

[7] E. Girczyc, "Loop Winding - A Data Flow Approach to Functional Pipelining", *In Proceedings of International Symposium on Circuits and Systems*, pp. 382-385, 1987.

[8] U. Prabhu and B. M. Pangrle, "Superpipelined control and data path synthesis", *In Proceedings of Design Automation Conference*, pp. 638-43, June, 1992.

[9] D. Ku and G. De Micheli, *High-Level Synthesis of ASICs under Timing and Synchronization Constraints*, Kluwer Academic Publishers, 1992.

[10] C. Papachristou, M. Spinning, and M. Nourani, "An effective power management scheme for RTL design based on multiple clocks", *In Proceedings of Design Automation Conference*, pp. 337-342, June, 1996.

[11] G. De Micheli, *Synthesis And Optimization of Digital Circuits*, pp. 166-178, McGraw-Hill, 1994.

[12] C.-T. Chen and K. Kucukcakar, "An Architectural Power Optimization Case Study using High-level Synthesis", To appear in *Proceedings of International Conference on Computer Design*, October, 1997.

[13] B. George, G. Yeap, M. G. Wloka, S. C. Tyler, and D. Gossain, "Power Analysis for Semi-Custom Design", *In Proceedings of IEEE Custom Integrated Circuits Conference*, pp. 249--252, May, 1994.