Clock Optimization for High-Performance Pipelined Design

Hsiao-ping Juan
Daniel D. Gajski
Smita Bakshi

Technical Report #96-01
January 31, 1996

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
(714) 824-7063

hjuan@ics.uci.edu
gajski@ics.uci.edu
sbakshi@ics.uci.edu

Abstract

In order to reduce the design cost of pipelined systems, resources may be shared by operations within and across different pipe stages. In order to maximize resource sharing, a crucial decision is the selection of a clock period, since a bad choice can adversely affect the performance and cost of the design. In this report, we present an algorithm to select a clock period that attempts to minimize design area while satisfying a given throughput constraint. Experimental results on several examples demonstrate the quality of our selection algorithm and the benefit of allowing resource sharing across pipe stages.

Clock Optimization for High-Performance Pipelined Design

Hsiao-ping Juan, Daniel D. Gajski and Smita Bakshi
Department of Information and Computer Science
University of California, Irvine, CA 92717-3425

Abstract

In order to reduce the design cost of pipelined systems, resources may be shared by operations within and across different pipe stages. In order to maximize resource sharing, a crucial decision is the selection of a clock period, since a bad choice can adversely affect the performance and cost of the design. In this report, we present an algorithm to select a clock period that attempts to minimize design area while satisfying a given throughput constraint. Experimental results on several examples demonstrate the quality of our selection algorithm and the benefit of allowing resource sharing across pipe stages.

1 Introduction

In general, high-performance constraints are met by pipelining the design into several concurrently executing stages, such that at any given time each pipe stage operates on a different sample from the stream of incoming samples. Figure 1 illustrates the block diagram of the MPEG decoder algorithm [3], which is a typical example of such a DSP system. The block diagram shows that the design is pipelined or partitioned into several modules that can run one after the other on the same data but at the same time on different data. By partitioning the algorithm into such modules, each module can execute in 4000 ns without violating the throughput constraint of 4000 ns per block. Without this partitioning, each module needs to execute in a much shorter time in order to satisfy the throughput constraint.

There are three possible schemes to implement a pipelined DSP system such as the MPEG algorithm. In the first scheme, as shown in Figure 2(a), the modules are implemented separately. This is the scheme that is currently in practice – the description under development is partitioned into modules by the system designer and then each module, along with its performance and cost constraints, is given to a different design group. In this scheme, different modules can use different clock signals, as long as their delays satisfy the throughput constraint. Note that if different clock signals are used, they need to be synchronized at some point.

Clearly, implementing each module separately would result in a large number of resources, thereby increasing the cost of the design. However, it is possible to reduce this design cost by sharing hardware resources among these modules. For instance, the same functional unit can be utilized to perform several different operations from different pipe stages over different time-steps. Since one resource can now potentially do the job of many, a fewer number of such resources are required, resulting in a lower cost. In Figure 2(b), we show the second implementation scheme, where different modules share components in the data-path. To make resource sharing easier, we assume that a functional unit used by one pipe stage can be used by another pipe stage only after the synchronization point of their clock signals. For example, consider the clock signals clk_i and clk_{i+1} shown in Figure 2(b). The clock period of clk_{i+1} is twice as long as the clock period of clk_i. Obviously, a functional unit that performs an operation in pipe stage $i+1$ cannot be used to perform any operation in pipe stage i within two cycles of clk_i, even though the component delay may be shorter than the clock period of clk_i.

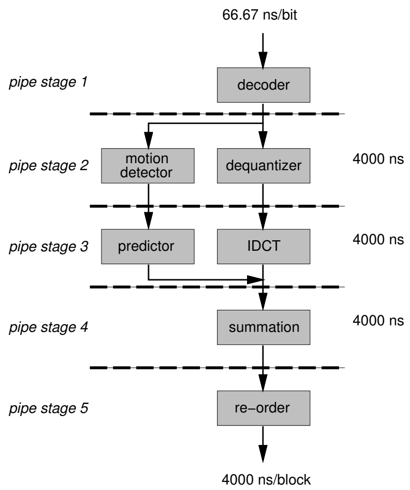

Figure 1: A pipelined MPEG example

The design cost can be further reduced by sharing the control units among the modules, that is, instead of one local control unit for each module, there is a global control unit, as shown in Figure 2(c). This sharing could be done by using the same clock signal for all the modules. Since all pipe stages should have the same delay, and since the clock signals are the same, it is obvious that all pipe stages require the same number of clock cycles. Therefore, we can implement a control unit such that in each state, the

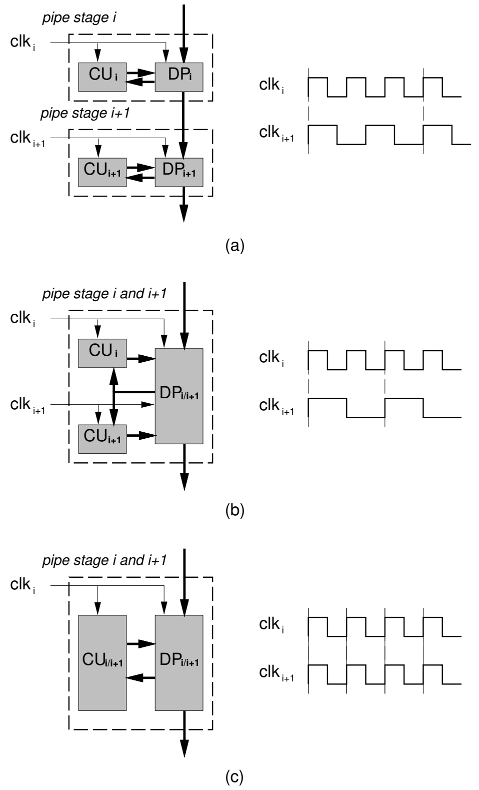

Figure 2: Different implementation schemes

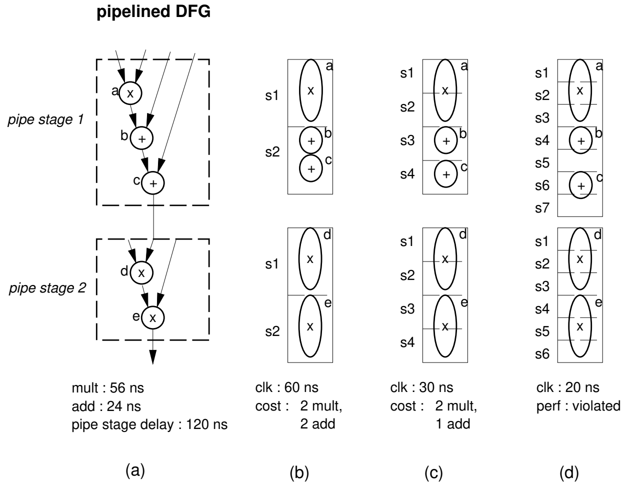

Figure 3: An example illustrating the effect of different clock periods on resource sharing of a pipelined design

operations of different pipe stages but in the same time-step would be executed concurrently. In this report, we will focus on the third implementation scheme since it can achieve the maximum resource sharing and, consequently, minimum design cost.

Note that as the amount of resource sharing increases, it becomes more difficult for a human designer to do the implementation without design automation tools. When performing resource sharing, an important decision is the selection of a clock period to schedule the operations into different states. A bad choice of the clock period could adversely affect the performance and cost of the final design. For instance, Figure 3(a) shows a two-stage pipeline with a pipe stage delay (the inverse of throughput) constraint of 120 ns. Figures 3(b), (c) and (d) show scheduling results of the given pipeline using 60, 30 and 20 ns as the clock period, respectively. When the clock period is equal to 60 ns, each pipe stage requires two clock cycles; therefore, each pipe stage has a delay of 120 ns, which satisfies the constraint. Note that the operations d and e can share the same multiplier since they are in different clock cycles. The operations a and d cannot share the same multiplier because they are scheduled in the same clock cycle. Therefore, the minimum cost is two multipliers and two adders. When the clock period is equal to 30 ns, each pipe stage requires four clock cycles and the pipe stage delay constraint is also satisfied. However, as shown in Figure 3(c), the additions b and c are now scheduled in different clock cycles and consequently, they can share the same adder. Thus, the design cost when the clock period is 30 ns is two multipliers and one adder, which is lower than the design cost when the clock period is 60 ns. When the clock period is equal to 20 ns, Figure 3(d) shows that pipe stage 1 requires seven clock cycles, which results in a delay of 140 ns and violates the pipe stage delay constraint. Using this example, we have shown that the selection of a clock period is a non-trivial problem. In this report, we propose a clock estimation algorithm that determines the clock period which satisfies the throughput constraint and requires minimum number of resources.

The rest of the report is organized as follows. In the next section, we discuss previous research done in this area and also explain how we differ from it. We present the problem definition and an overview of our algorithm in Section 3. The important steps in our algorithm are explained in greater detail in Section 4, 5 and 6. Finally, we present experimental results and give conclusions.

2 Previous Work

Several previous papers addressed the issue of clock period estimation for a given data flow graph. For example, there are several clock estimation schemes [7] [6] [4] that use the delay of the slowest component as the estimated clock period. However, using the slowest component delay as the clock period can lead to under-utilized functional units in cases where the components have widely differing delays.

A clock estimation method based on slack minimization is proposed in [5]. The amount of time that the circuit remains idle during a clock cycle is called a slack. This clock estimation method basically evaluates a contiguous range of integer candidate clock periods and the clock period that results in minimum slack would be chosen. However, this estimation method aims to select the clock period that optimizes the performance of the design. In our problem, performance optimization is not the goal; our goal is to minimize the design cost while satisfying the performance constraint.

In [1], a methodology is proposed to estimate the clock period for time-constrained scheduling as well as resource-constrained scheduling. However, this methodology does not consider pipelined designs, while our algorithm aims to select a clock period for pipelined designs.

Finally, the algorithm presented in this report differs from all the algorithms mentioned above in that our algorithm also takes the control unit delay into account. When the number of states is very large, the control unit tends to become very complex and control unit delay contributes significantly to the clock period and cannot be neglected. By considering the control unit delay, our algorithm provides a more realistic estimation than the previously published work.

3 Problem Definition and Algorithm Overview

Our problem can be defined as follows:

> Given (1) a pipeline of n pipe stages $PS_1 \cdots PS_n$, where each pipe stage PS_i is represented by a data flow graph DFG_i, (2) a component library, (3) the pipe stage delay constraint $PSDelay$ and (4) a range of allowable clock periods, represented by [*clkmin, clkmax*], find a clock period *clk* such that, $\forall i$, DFG_i can be scheduled into $\lfloor PSDelay/clk \rfloor$ states of delay $\leq clk$ and the design area is minimized.

The maximum clock period allowed, *clkmax*, is equal to $PSDelay$. Design libraries often specify the maximum clock frequency at which the clock input of a bistate circuit may be driven such that stable transitions of logic levels are maintained. This frequency is used to determine the value of *clkmin* if it is not already specified by the user. The cost of a pipeline is approximated by the total area of datapath components.

The example in Figure 4 illustrates the problem. Given are a 2-stage pipeline, where the pipe stages PS_1 and PS_2 are represented by DFG_1 and DFG_2 respectively, a component library, the range of allowable clock periods [20ns, 200ns], and a pipe stage delay constraint of 200 ns. Our algorithm estimates that choosing 40 ns as the clock period will produce a design with the minimum cost, which is two multipliers and one adder.

Our algorithm selects the clock period in three basic steps.

1. **Pipe stage shape function generation:** The first step in our algorithm is to produce a shape function in terms of clock periods versus the pipe stage delay, individually, for each pipe stage of the description. This shape function can clearly indicate the clocks that can satisfy the pipe stage delay constraint.

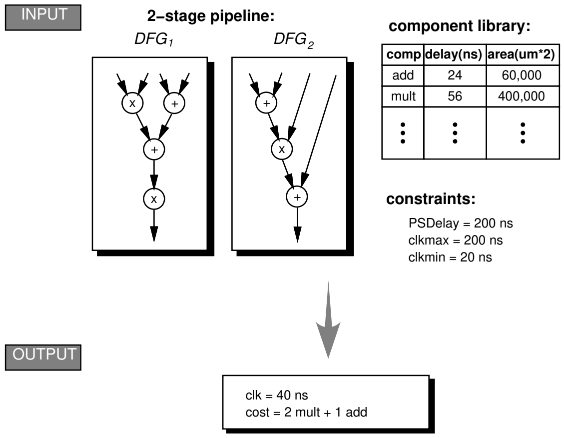

Figure 4: An example illustrating the inputs and outputs of the problem

2. **Clock candidates selection:** Next, given the pipe stage delay constraint, $PSDelay$, and the shape functions of each pipe stage, a set of clock periods that can satisfy the pipe stage delay constraint in all stages can be easily obtained. This set of clock periods is called *clock candidates*.

3. **Resource estimation:** Having obtained the set of clock candidates, the final step in our algorithm is to estimate the amount of resources required by each clock candidate. Then the algorithm would return the clock period that requires the least amount of resources.

Details of each of the steps above will be discussed in the following sections.

4 Pipe Stage Shape Function Generation

In this section, we will discuss how to generate the shape function of one pipe stage. Before we present the algorithm, we will show the underlying design model used for the purpose of clock period calculation.

4.1 Design Model

The design model for clock estimation, as shown in Figure 5, is similar to the design model used in [5]. In this model, the datapath consists of registers, functional units and tri-state drivers. A two-level bus structure is assumed for the interconnection across the registers and functional units. A typical datapath operation involves reading operands from the registers, computing the result in the functional units, and finally writing the result into a destination register. Note that a register could be used to store a temporary value that is used in different states of the same pipe stage or could be used as a pipeline latch between pipe stages. Operation chaining is supported in this model by allowing connections from the output ports of some functional units directly to the input ports of other functional units. Moreover, operations can execute over several clock cycles; that is, multi-cycled operations are possible.

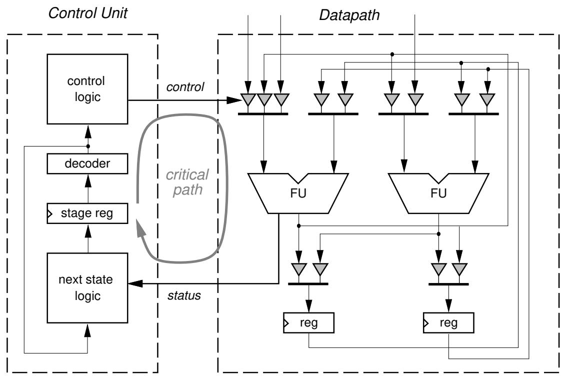

Figure 5: Design model for clock estimation

The control unit consists of the state register, a decoder, the control logic to drive the control lines for the datapath components, and the next-state logic to compute the next state to be stored in the state register. The control unit implements a state machine that sequences a design through a series of states, each of the states represents the set of datapath operations performed concurrently in the same or different pipe stages of the design.

The clock period is determined by the longest register-to-register delay. Typically, the path through the control logic, as shown in Figure 5, has the largest delay. Consequently, the minimal clock period is equal to or greater than the sum of all the delays associated with the components and the wires in the path. In other words, we can specify the clock period by the following:

$$clk = Tp_{SR} + T_{DEC} + T_{CL} + T_{TR} + T_{FU} + T_{NS} + Ts_{SR} + T_{Wire}$$

where:

Tp_{SR} and Ts_{SR} are the propagation delay and the setup time of the state register, respectively,
T_{DEC} is the delay of the decoder,
T_{CL} is the delay of the control logic,
T_{TR} is the delay of the tri-state driver,
T_{FU} is the delay of the functional units,
T_{NS} is the delay of the next-state logic, and
T_{Wire} is the total delay of the wires in the path.

However, the wire delay cannot be meaningfully estimated without a floorplan, which is not available at this early stage of synthesis. Therefore, in our estimation, the clock period *clk* is approximated using the following equation:

$$clk = T_{DP} + T_{CU}$$
$$T_{DP} = T_{TR} + T_{FU}$$
$$T_{CU} = Tp_{SR} + T_{DEC} + T_{CL} + T_{NS} + Ts_{SR}$$

where T_{DP} is the delay of the datapath, and T_{CU} is the delay of the control unit.

4.2 Shape Function Generation Algorithm

The shape function generation algorithm basically consists of three steps. It first produces a shape function in terms of clock periods versus the minimum number of states by considering only the datapath delay. Then the algorithm estimates the control unit delay and updates the shape function accordingly. Finally, the shape function of clock periods versus the pipe stage delay can be computed by multiplying the clock periods to the corresponding number of states. We will describe the first two steps in the following sections.

4.2.1 Datapath

Given a data flow graph DFG_i of the pipe stage PS_i and the range of clock period allowed, [*clkmin, clkmax*], the goal is to generate the shape function of clock periods versus the minimum number of states that the stage PS_i requires.

Since the clock periods are in the real number domain, clearly, it is infeasible to attempt to go through all possible clock periods and estimate the minimum number of states that the pipe stage requires for each of them. However, the possible numbers of states are in the integer domain. Therefore, instead of computing the minimum number of states required for all possible clock periods, the shape function is generated incrementally by fixing the number of states, and then computing the minimum clock period for the fixed number of states using the procedure *MinClkPeriod* outlined in Figure 6. This process produces one (clock period, number of states) point in the shape function. To obtain the entire shape function, we iteratively increase the number of states, beginning with the smallest possible number, which is $\lceil PSDelay/clkmax \rceil$, and finishing with the largest possible number, which is $\lfloor PSDelay/clkmin \rfloor$. This approach is based on the fact that, assume the algorithm estimates that the shortest possible clock period for scheduling the data flow graph into i states is clk_i and similarly, the shortest clock period for $i+1$ states is clk_{i+1}, then we can conclude that for any clock period clk_j, $clk_{i+1} \leq clk_j \leq clk_i$, the minimum number of states that the data flow graph would be scheduled into by using clk_j is $i+1$.

The procedure *MinClkPeriod* is adapted from the ASAP scheduling algorithm; however, instead of minimizing the number of states given a clock period as in the ASAP algorithm, it minimizes the clock period given the number of states. A brief explanation follows.

Given a data flow graph DFG, the procedure $MinClkPeriod$ first computes the path length for each of the operations in DFG. The path length of an operation is defined as the longest path delay starting from this operation till the output node. Therefore, by definition, the maximum path length, *MaxPathLength*, of all operations in DFG is the critical path length. The next step of the procedure involves determining whether a ready op-

Procedure: MinClkPeriod
Inputs: a data flow graph DFG, the number of states N;
Output: the minimum clock period;
begin Procedure
$Cstep = 1$;
$ComputePathLength(DFG)$;
$MaxPathLength$ = delay of the longest path in DFG;
$MinClk$ = MaxPathLength/N;
$InsertReadyOps(DFG, PList)$;
while ($PList \neq \emptyset$) **do**
if $Cstep = N$ **then**
schedule all the non-scheduled operations;
$MinClk$ = maximum state delay;
$PList = \emptyset$;
else
$op = First(PList)$;
if op is a single-cycled operator **then**
determine chaining or non-chaining;
schedule op and update $MinClk$;
else
determine the number of cycles of op;
schedule op and update $MinClk$;
end if;
$InsertReadyOps(DFG, PList)$;
$Cstep = Cstep + 1$;
end if;
end while;
return $MinClk$;
end Procedure

Figure 6: The procedure to estimate the minimum clock period, given N states

eration can be scheduled. In ASAP scheduling, all ready operations are scheduled as soon as possible, as long as the clock period constraint is not violated. In our procedure, whether a ready operation can be scheduled or not and whether chaining or multi-cycling should be performed depends upon its effect on the clock period.

The variable $MinClk$ is initialized to the optimal clock period $MaxPathLength/N$, where N is the number of states that the DFG would be scheduled into. Then for each operation in the ready list $PList$, we first determine whether it would be a single-cycled operation or multi-cycled operation using the delay of the operation and the current clock period $MinClk$. If the operation delay is less than or equal to $MinClk$, which means it is a single-cycled operation, we then need to decide whether it could be chained with its predecessor in the current state or be deferred to the next state. If the operation delay is larger than $MinClk$, which means that it is a multi-cycled operation, we would require to decide whether to schedule it across $\lfloor$(operation delay)$/MinClk\rfloor$ or $\lceil$(operation delay)$/MinClk\rceil$ states. The scheduling of an operation may increase the clock period and the variable $MinClk$ would be updated if it does. Once an operation is scheduled, some other non-ready operations become ready and would be inserted into the ready list. This process continues and when it reaches the last state, all the non-scheduled operations are scheduled into the last state and the procedure returns the variable $MinClk$, which now stores the longest state delay, that is, the clock period.

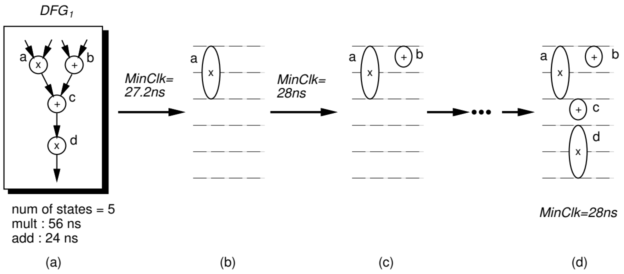

Figure 7: Determining the minimum clock period

Clearly, the result of this algorithm depends upon how it determines chaining and multi-cycling. We now illustrate how chaining and multi-cycling are determined on the example in Figure 7.

Knowing that a multiplication operation takes 56 ns and an addition takes 24 ns, the procedure computes that the maximum path length is 136 ns. Since the data flow graph would be scheduled into five states, the optimal clock period, that is, the current $MinClk$, is 136/5=27.2 ns. In the first iteration of the procedure, it would attempt to schedule the operation a. Knowing that the delay of the operation a is 56 ns and the current clock period is 27.2 ns, a should be a multi-cycled operation and the procedure needs to determine whether to schedule it across $\lfloor 56/27.2\rfloor$=2 states or $\lceil 56/27.2\rceil$=3 states. Let's consider the first case where a is scheduled across two states. This means that average delay of the first two states would be 56/2=28 ns each. Furthermore, if a is finished in two states, its successors c and d could be scheduled across three states, which results in an estimated delay per state of (24+56)/3=26.7 ns. Thus, the maximum state delay in this case, that is, the clock period, would be 28 ns. Consider the second case, where a is scheduled across three states. This gives an average state delay of 18.7 ns for the first three states. However, operations c and d now need to be finished within two states, which gives an estimated delay per state of (24+56)/2=40 ns. That is, the clock period in this case would be 40 ns. Since scheduling the operation a into a two-cycled operation gives an estimation of shorter clock period, the procedure decides to schedule a across the first two states as shown in Figure 7(b).

The next iteration involves the scheduling of the operation b. Note that the clock period $MinClk$ has now been updated to 28 ns. Since the delay of the operation b is less than 28 ns, it is a single-cycled operation and its scheduling does not change the current clock period. The result of this iteration is shown in Figure 7(c). The procedure continues this process for the rest of the operations c and d, and the final result is shown in Figure 7(d) and the minimum clock period for scheduling the data flow graph into five states is 28 ns.

Similarly, we can estimate that the minimum clock periods for scheduling the data flow graph in Figure 7(a) into

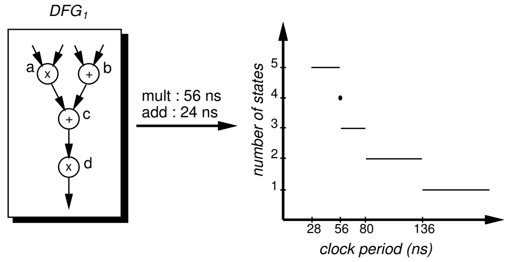

Figure 8: The shape function of clock periods versus number of states for the pipe stage PS_1

one, two, three, or four states are 136 ns, 80 ns, 56 ns and 56 ns respectively. Therefore, we can conclude that for any clock period larger than 136 ns, the minimum number of states that DFG requires is one; for any clock period between 136 and 80 ns, the minimum number of states that DFG requires is two, etc. Figure 8 shows the resultant shape function.

The procedure $MinClkPeriod$ has a $O(n)$ time complexity, where n is the number of operators in the given DFG. Let c denote the possible number of states the given DFG could be scheduled into, that is, $c = \lceil PSDelay/clkmax \rceil - \lfloor PSDelay/clkmin \rfloor$. Then the time complexity of generating a shape function for one pipe stage is $O(cn)$.

4.2.2 Control Unit

The control unit sequences a design through a series of states, where each of the states represents the set of datapath operations performed concurrently in the same or different pipe stages of the design. In general, if a shorter clock period is used to schedule the pipe stages, the number of states per pipe stage would become larger, and consequently, the control unit becomes more complex and the control unit delay is longer. When the number of states is very large, the control unit delay contributes significantly to the clock period and cannot be neglected. In the previous section, an algorithm used to estimate the relation between the clock period and the number of states by considering only the datapath was presented. In this section, we will explain how to estimate the control unit delay and update the shape function accordingly.

Given a clock period clk, and pipe stage delay $PSDelay$, each pipe stage will be scheduled into $N = \lfloor PSDelay/clk \rfloor$ states. The control unit thus needs to implement an N-state state machine, where in state S_i, all the operations scheduled in the i-th state of all pipe stages would be executed.

As illustrated in Figure 5, the control unit consists of a state register, a decoder, the control logic and the next-state logic. The control unit may be implemented as random-logic, a read-only memory(ROM), or a programmable logic array(PLA). In this paper, we will assume a random-logic implementation as shown in Figure 9.

We assume that the present states are encoded as binary

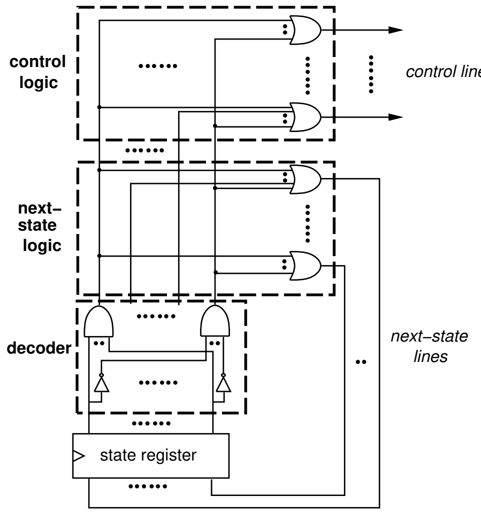

Figure 9: A random-logic implementation of the control unit

values and are stored in the state register. Therefore, given that the total number of states is N, the state register bitwidth will be $B = \lceil \log_2 N \rceil$. Taking the B-bit output of the state register as input, the decoder decodes it into an N-bit output such that each bit corresponds to one state. The decoder consists of B inverters and N AND-gates. Each inverter is used to invert one bit of state register, and the number of inputs to an AND-gate is B.

One OR-gate is required for each control line and next-state line. The size, that is, the number of inputs, of an OR-gate for a control line is identical to the number of states during which the corresponding control line is asserted. For example, a control line of a functional unit will be asserted whenever the functional unit performs an operation bound to it. In the worst case, there exist functional units that are used in every state and consequently, the OR-gates that generate the control lines for these functional units would have N inputs. To determine the size of an OR-gate for a next state line, we assume that each next-state line is "toggled" on the average during half of the states in the design since the state values are binary encoded. Thus, the size of each OR-gate driving a next-state line is assumed to be equal to $N/2$.

Since most component libraries usually provide AND and OR-gates with a limited number of inputs, all the large AND and OR-gates in our model need to be decomposed into a multi-level implementation. The multi-level decomposition aims to produce an implementation with the minimal number of levels. This is guided by the fact that a multi-level implementation of a large AND-gate with I number of inputs using AND-gates with a maximum of M inputs is in the form of an M-ary tree. The height of the tree, which corresponds to the number of levels, is equal to $\lceil \log_M I \rceil$. Let us assume that the component library contains AND and OR-gates with a maximum of M inputs, and let T_{AND}, T_{OR}, and T_{INV} denote the delay of an M-input AND-gate, an M-input OR-gate and an inverter, respectively. The following equations are used to estimate the decoder, the control logic, and the next-state logic delay.

$$\begin{aligned} T_{DEC} &= T_{INV} + \lceil \log_M B \rceil \times T_{AND} \\ &= T_{INV} + \lceil \log_M \log_2 N \rceil \times T_{AND} \\ T_{CL} &= \lceil \log_M N \rceil \times T_{OR} \\ T_{NS} &= \lceil \log_M (N/2) \rceil \times T_{OR} \end{aligned}$$

Having obtained T_{DEC}, T_{CL} and T_{NS} using the equations above, and the propagation delay and the setup time of the state register from the component library, the control unit delay can be computed by the equation given in Section 4.1.

Now let's consider a shape function described in the previous section. Given a point (clk_i, N) in the shape function, we can use the estimation method discussed above to estimate the delay of an N-state control unit. Assume the delay of an N-state control unit is $T_{CU}(N)$, the algorithm would update the point (clk_i, N) to $(clk_i + T_{CU}(N), N)$. Note that given two points (clk_i, N) and $(clk_{i+1}, N+1)$, where $clk_i \leq clk_{i+1}$, it is possible that $clk_i + T_{CU}(N) \geq clk_{i+1} + T_{CU}(N+1)$. In this case, the algorithm would drop the point $(clk_i + T_{CU}(N), N)$.

After the shape function of clock periods versus the number of states is updated, we can obtain the shape function of clock periods versus the pipe stage delay by multiplying the clock periods to the corresponding number of states. Figure 10(a) and (b) illustrate the final shape functions of pipe stage 1 and 2 in the example shown in Figure 4.

5 Clock Candidates Selection

Having obtained the shape functions for each pipe stage of an n-stage pipeline, the next step in our algorithm is to determine the set of clock periods, called clock candidates, that can produce schedules satisfying the $PSDelay$ constraint in every stage. This is done by first determining, for each pipe stage PS_i, the set of clock periods, $ClkSet_i$, that can satisfy the pipe stage delay constraint. Then by intersecting all $ClkSet_i$, $i = 1, 2, \cdots, n$, the set of clock periods that can satisfy the $PSDelay$ constraint for all pipe stages can be generated.

Figures 10(a) and (b) show the shape functions of the two-stage pipeline shown in Figure 4. Given that the $PSDelay$ constraint is 200 ns, the shaded regions in Figure 10(a) and (b) indicate the range of clock periods that can produce schedules whose delay is less than or equal to $PSDelay$. By intersecting these two sets of clock periods, a set of clock period, shown as the shaded regions in Figure 10(c), that can satisfy $PSDelay$ in both pipe stages is obtained.

Note that the set of clock candidates is still in the real number domain. We minimize the number of clock candidates by considering only the integer divisions of the pipe stage delay constraint $PSDelay$. The reason is the following. Let's consider two different clock periods clk_i and clk_j,

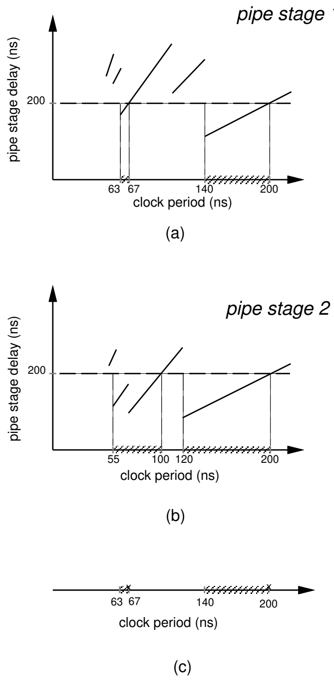

Figure 10: Determining the set of clock candidates

both can satisfy the $PSDelay$ constraint. Moreover, using either of them, the pipe stages would be scheduled into N states. Assume that $clk_i = PSDelay/N$. Since clk_j also satisfies the $PSDelay$ constraint, we can conclude that $clk_j \leq PSDelay/N$ and consequently, $clk_j < clk_i$ since $clk_i \neq clk_j$. Given a DFG and a fixed number of states, we know that the longer clock period would require equal or less resources than the shorter clock period. Since the goal is to select the clock period that requires the minimum number of resources, only clk_i needs to be considered. Thus, from the set of clock periods obtained above, only those which are integer divisions of $PSDelay$ are selected as clock candidates. For instance, the set of clock candidates selected for the example shown in Figure 10 is $\{67, 200\}$.

6 Resource Estimation

Having obtained the set of clock candidates, the next step is to estimate the amount of resources required for each clock candidate. The resource estimation algorithm is outlined in Figure 11.

The inputs to this algorithm consist of the data flow graphs $DFG_1, \cdots, DFG_n$, each of which represents one pipe stage, the clock period clk, and the number of states N. Note that the clock period clk here should be a clock period that does not include the control unit delay. This can be done by subtracting the control unit delay from the clock candidates. The output of the algorithm is the minimum number of resources required.

The underlying concept of the resource estimation algorithm is that, if there are n operations that need to be finished within s states, and a component used to perform

Algorithm: resource estimation
Inputs: clock period clk, number of states N, and data flow graphs $DFG_1, \cdots, DFG_n$;
Outputs: the minimum number of components for each type of operation;
begin Algorithm
  call ASAP;
  call ALAP;
  for each operation type t **do**
    partition N states into a set of disjoint operation distribution intervals I;
    $MinComp(t) = 0$;
    for each $I_i \in I$ **do**
      $NumOp$ = the number of operations of type t in I_i;
      $NumOpCycle = \lceil$(delay of component type t)$/clk\ \rceil$;
      $NumComp = \lceil \frac{NumOp \times NumOpCycle}{\text{the length of interval } I_i} \rceil$;
      if ($NumComp > MinComp(t)$) **then**
        $MinComp(t) = NumComp$;
      end if;
    end for;
  end for;
end Algorithm

Figure 11: The algorithm to estimate resources

an operation requires at least c clock cycles to finish the execution before it could be used again to execute another operation, then clearly, the minimum number of components required is equal to $\lceil (n \times c)/s\ \rceil$.

An example to illustrate the algorithm is shown in Figure 12(a). We know that all pipe stages are executed concurrently and in order to consider resource sharing across the stages, the algorithm needs to consider the operations in all the stages at the same time. In order to demonstrate this, we put the $DFGs$ from two pipe stages side by side in Figure 12. Note that these pipe stages are executed in parallel but on different input samples.

Given the clock period and the number of states, the first step of the algorithm is to compute the ASAP and ALAP values of each operation. Let $ASAP_i$ and $ALAP_i$ denote the ASAP and ALAP value of operation o_i respectively, the time frame of o_i is defined as $(ALAP_i - ASAP_i + cycle(o_i))$, where $cycle(o_i)$ represents the number of clock cycles required to finish the operation o_i. Consider the example shown in Figure 12(a). Assume that the clock period is 30 ns and each of the $DFGs$ will be scheduled into 5 states. Figure 12(b) shows the time frames of all the operations.

Having computed the time frames of all the operations, the algorithm estimates the minimum number of required components for each operation type separately. For example, Figure 12(c) shows the time frames of all the multiplications, and Figure 12(d) shows the time frames of all the additions. The next step in the algorithm is to partition states into a set of disjoint *operation distribution intervals* such that there are no overlapping time frames between two consecutive intervals. For example, in Figure 12(c), there is no way of partitioning those five states into intervals such that there are no overlapping time frames of the multiplication operations; therefore, there is only one operation distribution interval, {s1,s5}, for multiplication operations, where s1 is the starting state and s5 is the ending state of the interval. On the other hand, as shown in Figure 12(d), there are three operation distribution intervals for additions: interval I_1={s1,s2}, interval I_2={s3,s3} and interval I_3={s4,s5}. After the operation intervals are obtained, we can estimate the required number of components for each interval separately, and the maximum number of required components over all intervals is the minimum number of components needed to perform all the operations.

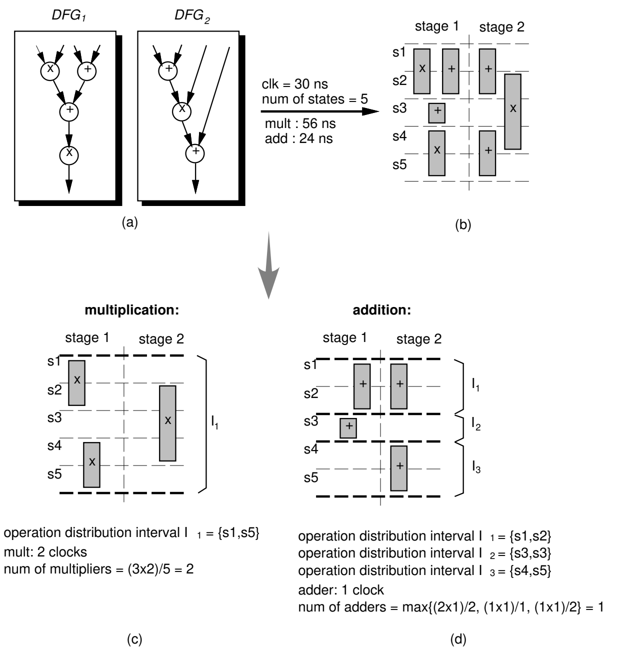

Figure 12: An example illustrating the algorithm of resource estimation

The required number of components for one distribution interval is estimated by applying the concept introduced at the beginning of this section. In the algorithm outlined in Figure 11, the variable $NumOp$ computes the number of operations that need to be finished within the given interval. For example, consider the multiplication operations shown in Figure 12(c). There are three operations that need to be finished in interval I_1. The variable $NumOpCycle$ represents the number of clock cycles required to finished one execution of the operations. Since the delay of the multiplier is given as 56 ns and the clock period is 30 ns, clearly it would need at least 2 clocks to finish one multiplication. The number of required components, denoted by the variable $NumComp$, can then be computed by $\lceil (NumOp \times NumOpCycle)/(\textit{the length of the interval}) \rceil$. For example, the minimum number of multipliers required is $\lceil (3 \times 2)/5 \rceil$, that is, at least two multipliers are needed. Similarly, knowing that an addition operation needs one clock to execute, we can estimate the required number of adders for the three distribution intervals of addition operations individually. As shown in Figure 12(d), the required numbers of adders for intervals I_1, I_2 and I_3 of additions are $\lceil (2 \times 1)/2 \rceil$, $\lceil (1 \times 1)/1 \rceil$ and $\lceil (1 \times 1)/2 \rceil$ respectively, resulting in the estimation of one adder.

The resource estimation algorithm has a time complexity of $O(n \log n)$, where n is the total number of operations in all the given $DFGs$. A brief explanation follows. The first step in our algorithm is to partition states into a set of disjoint operation distribution intervals. This can done by first sorting the list of all operations using their ASAP values as the main key and their ALAP values as the secondary key. Then the first operation distribution interval is initialized to be the time frame of the first operation in the list. Then if the time frame of the next operation intersects with the current operation distribution interval, then the operation distribution interval is updated to be the union of the current operation distribution interval and the time frame of the next operation. If there is no intersection between the current operation distribution interval and the time frame of the next operation, then one operation distribution interval is found and this process starts again from the next operation. Clearly, this process has a $O(n \log n)$ time complexity due to the sorting algorithm. After all the operation distribution intervals are found, the next step in our algorithm is to compute the number of components required in each interval. Since in this step, each operation would be count exactly once, this step has a linear complexity. Thus, we can conclude that the resource estimation algorithm has a complexity of $O(n \log n)$.

7 Experiments

In this section, we present results of three experiments with the clock estimation algorithm which we have implemented using C on a SUN SPARC 5 station. In the first experiment, we demonstrate the quality of our algorithm by comparing the selected clock against the "best" clock obtainable using force-directed scheduling. The second experiment studies the impact of resource sharing across different pipe stages on the cost of a design, and finally, the third experiment demonstrates the effect of considering control unit delay on the clock selection.

For all experiments we have used the the VLSI Technology Inc. VDP370 1.0 micron Datapath Element Library [8] to obtain the area and delays of the functional units. The datapath elements used are shown in Figure 13.

component	delay(ns)	area(x1000 um*2)
adder	11.2	54
subtractor	15.5	60
multiplier	32.0	320

Figure 13: Datapath component library

7.1 Experiment 1: Quality of Results

As discussed in Section 2, there are no existing clock selection algorithms for pipelined designs; furthermore, the existing clock selection algorithms do not take control unit delay into account. Thus, in order to demonstrate the quality of our algorithm, we have been unable to compare our results with related research in clock selection; instead, we have utilized force-directed scheduling, which is a well known time-constrained scheduling algorithm.

This experiment is conducted on four examples that are typically implemented as pipelined designs: the AR lattice filter (AR) [4], the linear phase b-spline interpolated filter (BSpline) [5], the elliptical filter (EF) and the HAL benchmark. For each of the examples, we first generate a number of input descriptions by manually pipelining the specification into a different number of stages. For example, we pipeline the elliptical filter into 2, 3, and 4 stages, where the delay of the pipe stages in each pipeline is as equal as possible. We then place different pipe stage delay constraints on each of the pipelined descriptions, and for a given pipe stage delay constraint we obtain the estimated and the "best" clock period as follows:

- The best clock period is obtained by executing the force-directed scheduling algorithm for a number of clock periods, each corresponding to a different number of states. The clock period that gives the minimal area design is then the best period. For instance, for the 4-stage elliptical filter design with a pipe stage delay constraint of 150 ns, we run force-directed scheduling using 10 different clocks (150, 75, 50, ..., 15) corresponding to (1, 2, 3, ..., 10) states per pipe stage, and we find that the best clock, that is, the clock that results in minimal area, is 16.66 ns (corresponding to 9 states).
- The estimated clock period is obtained by executing our clock-selection algorithm.

Examples	# of stages	PSDelay (ns)	FDS		ours		res. diff. (%)
			clk(ns)	resources	clk(ns)	resources	
AR	2	150	16.6	4A,5M	16.6	4A,5M	0
	3	100	16.6	6A,8M	16.6	6A,8M	0
BSpline	2	150	21.4	2A,2M	18.75	2A,2M	0
	3	100	33.3	3A,2M	12.5	2A,3M	33.2
EF	2	300	27.2	3A,2M	33.3	4A,2M	6.7
	3	200	33.3	5A,2M	22.2	5A,3M	35.2
	4	150	16.6	5A,2M	16.6	5A,2M	0
HAL	2	150	50	1A,1S,2M	50	1A,1S,2M	0

A: adder, S: subtractor, M: multiplier

Figure 14: Comparing the best and the estimated clock period for four benchmarks: AR, BSpline, EF, and HAL

The results of comparing the best and the estimated clock period for the four examples mentioned above are shown in Figure 14. The first three columns give the example name, the number of pipe stages and the $PSDelay$ constraint used for each pipelined description. The next two columns give the best clock and the corresponding resources obtained by executing the force-directed scheduling algorithm for different clocks. The next two columns give the estimated clock and the corresponding resources obtained using our algorithm. Finally, the last column gives the percentage of difference in design area, which is approximated by the sum of the areas of all the components, obtained by the force-directed scheduling algorithm and our clock-selection algorithm. As can be seen from the results, the estimated clock period was identical to the one obtained with FDS in most cases; however, in three cases our algorithm estimated a clock period that resulted in the use of either one more multiplier or one more adder than that obtained with FDS. In two of the three cases, where the clock period selected by our algorithm requires one more multiplier than that required with FDS, the percentage of difference is larger than 30%. We observe that this is due to the fact that the multiplier we use in this experiment is about five times larger than the adder.

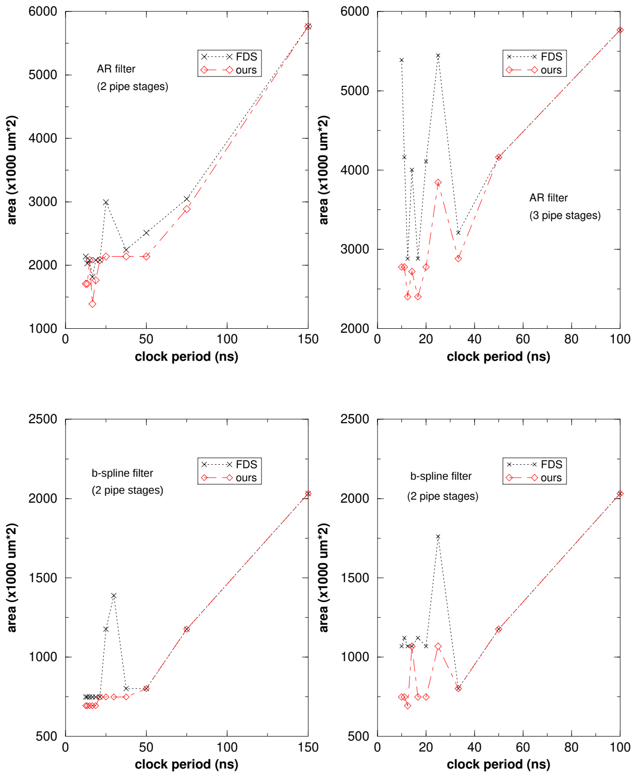

Figure 15: Comparing our resource estimates against the results of the FDS algorithm for the AR and BSpline examples

This discrepancy between the estimated and the best clock period may be explained by considering the fidelity of our resource estimation method, which essentially gives a lower bound on the number of resources. It is important to note that the correct selection of the clock depends more on the fidelity rather than on the accuracy of the resource estimation. Roughly speaking, fidelity refers to how closely the shape of the estimation curve resembles that of the actual curve, while accuracy refers to how closely the two curves follow each other. We would like to point out that our fidelity measure is drawn against FDS, which itself is a heuristic and does not guarantee an optimal solution. Since we do not have an optimal solution, we cannot make any conclusions on our algorithm's fidelity in general, but merely on its fidelity with respect to a known heuristic, which in our case is FDS.

In order to illustrate the role of fidelity of our resource estimates, in Figures 15 and 16, we have compared the results of our resource estimates with the resources obtained by the force-directed scheduling algorithm for all examples and *PSDelay* constraints shown in Figure 14. Note that the AR 2 and 3-stage pipelines as well as the bspline 2-stage design have high fidelity. Hence, our clock selection algorithm selected the best clock - in spite of the fact that the accuracy of the estimation (especially for the 3-stage AR design) was low in some cases. However, for the 3-stage BSpline example, the fidelity of our resource estimation when compared with FDS, between clock periods of 10 to 20 ns is low; hence, our clock selection algorithm picks a clock of 12.5 ns rather than 33.3 ns. Similarly, for the 2 and 3 stage elliptical filter designs, the fidelity between clock periods of 10 to 20 ns is low and thus the best clock period is not selected. The fidelity and accuracy for the 4-stage elliptical filter design is excellent while the 2-stage HAL example also has good fidelity, thus best clock periods are selected for both cases.

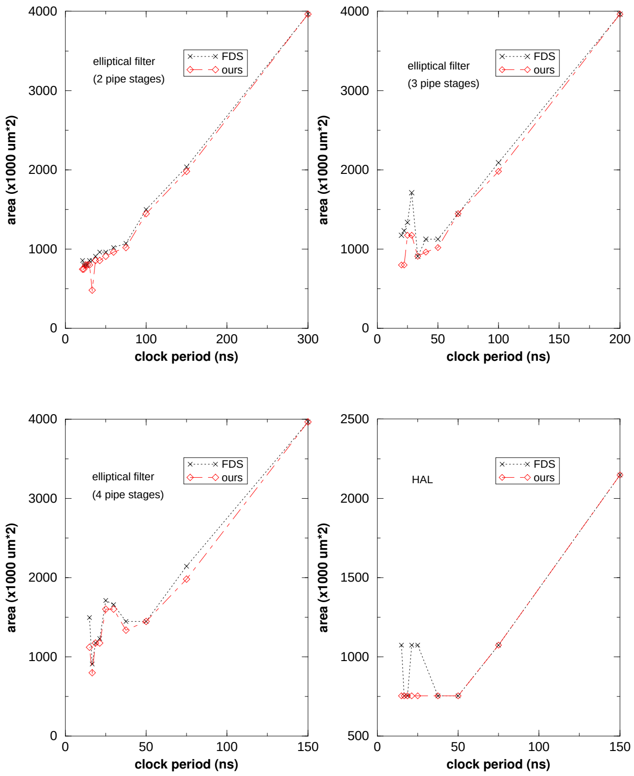

Figure 16: Comparing our resource estimates against the results of the FDS algorithm for the EF and HAL examples

As an aside, we would like to point out that the fidelity is low only for low clock periods, since the extent of resource sharing increases at lower clock periods, and thus increases the probability of an erroneous resource estimate.

From the results it may appear that the FDS approach is superior than our approach; however, we would like to point out that in the case of the elliptical filter example, whereas it took approximately 1 second to estimate and select the clock period for a given pipe stage delay constraint using our algorithm, it took more than 17 minutes to obtain the best clock period using the FDS algorithm since it had to be iterated over approximately fifteen different clock periods.

Thus, in conclusion, our algorithm selects a clock period that uses minimal area resources (or close to the minimal) within less than one second.

7.2 Experiment 2: Resource Sharing

Resource sharing, or scheduling, is a well-known technique that is utilized to reduce the area of a design, by allowing a resource to be shared by two or more operations. In this section, we study the impact of sharing resources in a pipelined design, where a resource can not only be shared by operations within the same pipe stage, but also by operations across different pipe stages.

This experiment is conducted on the same examples that were used in the previous section: AR, BSpline, EF, and the HAL benchmark. Once again, we generate input descriptions by manually pipelining the examples into a different number of pipe stages, and for each of the pipelines we use a number of different $PSDelay$ constraints. For each description and constraint, we then compare the minimum number of resources obtained by implementing all the pipe stages individually (that is, by dis-allowing resource-sharing across different pipe stages) to that obtained by implementing all the pipe stages together and thus allowing resource sharing across different pipe stages. This difference is illustrated in Figures 2(a) and (c).

The minimum cost of a design without resource sharing is computed by first obtaining the best clock period and the minimum number of resources required for each pipe stage separately using force-directed scheduling, and then summing up the resources of all the pipe stages. To compute the minimum number of resources required with sharing, we first select a clock period by applying our algorithm to the pipelined descriptions and then generate the minimum number of resources required using force-directed scheduling.

The results on the four examples are shown in Figure 17. The first three columns give the name of the example, the number of pipe stages and the $PSDelay$ constraint used for each description. When resource sharing across different pipe stages is not allowed, the results show the clock period, the number of states and the minimum number of resources required for each pipe stage. Note that in this case, different pipe stages can have different clock periods. When resource sharing is considered, all the pipe stages have the same clock period.

In all the cases, the results indicate that resource sharing within and across different pipe stages reduces the design area from anywhere between 3.6 and 43.3 %. This shows a substantial reduction in area when resource-sharing across different pipe stages is allowed and gives an indication of the effectiveness of our algorithm.

7.3 Experiment 3: Control Unit Delay

This experiment is conducted for the AR filter and the elliptical filter benchmarks. Note that in this experiment, the descriptions of the benchmarks are not pipelined.

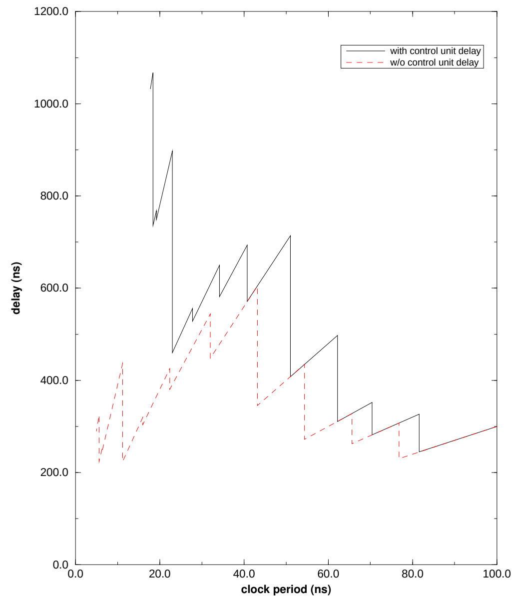

Figure 18: The clock period vs delay shape functions of the elliptical filter example, generated with and without the control unit delay estimation

Figure 18 shows the result of the elliptical filter benchmark. There are two shape functions of clock periods versus total delay. The shape function in solid line is obtained by our shape function generation algorithm with the control unit delay estimation, while the shape function in dashed line is generated by our algorithm, but assuming the control unit delay is zero. Similarly, Figure 19 shows the result obtained for the AR filter example.

From the results, we observe that if the control unit delay is not taken into account, the delay is smaller when the clock period becomes very short. However, when the control unit delay is considered, the delay is actually longer when the clock period becomes shorter. The reason is, when the clock period is very short, the number of states becomes large and consequently, the control unit is more complicated and results in longer delay and longer clock period.

Now take the shape functions of the elliptical filter as an example and let us assume that the delay constraint is 500 ns. Without considering the control unit delay, a clock period as small as 6 ns may be selected as the clock period. However, if the control unit delay is taken into

Examples	# of stages	PSDelay (ns)	without res. sharing acr. pipe stg.					res. sharing acr. pipe stg.			imprv. (%)
			stage	clk(ns)	# of states	resources per stage	resources total	clk(ns)	# of states	resources	
AR	2	150	stage 1	18.75	8	2A,4M	**6A,7M**	16.6	9	**4A,5M**	**29.2**
			stage 2	18.75	8	4A,3M					
	3	100	stage 1	12.5	8	2A,2M	**8A**,8M	16.6	6	**6A**,8M	**3.6**
			stage 2	50	2	2A,2M					
			stage 3	12.5	8	4A,4M					
BSpline	2	150	stage 1	50	3	2A,1M	**3A**,2M	21.4	7	**2A**,2M	**6.7**
			stage 2	18.75	8	1A,1M					
	3	100	stage 1	50	2	2A,1M	**4A**,3M	12.5	8	**2A**,3M	**9.2**
			stage 2	20	5	1A,1M					
			stage 3	50	2	1A,1M					
EF	2	300	stage 1	25	12	2A,1M	**5A**,2M	33.3	9	**4A**,2M	**5.9**
			stage 2	37.5	8	3A,1M					
	3	200	stage 1	20	10	2A,1M	**7A**,3M	22.2	9	**5A**,3M	**8.1**
			stage 2	50	4	2A,1M					
			stage 3	25	8	3A,1M					
	4	150	stage 1	30	5	1A,0M	**6A**,4M	16.6	9	**5A**,2M	**43.3**
			stage 2	18.75	8	2A,1M					
			stage 3	50	3	1A,1M					
			stage 4	18.75	8	2A,2M					
HAL	2	150	stage 1	37.5	4	1A,0S,1M	**2A**,1S,2M	50	3	**1A**,1S,2M	**6.7**
			stage 2	75	2	1A,1S,1M					

A: adder; S: subtractor; M: multiplier

Figure 17: The effects of resource sharing on four benchmarks AR, BSpline, EF, and HAL

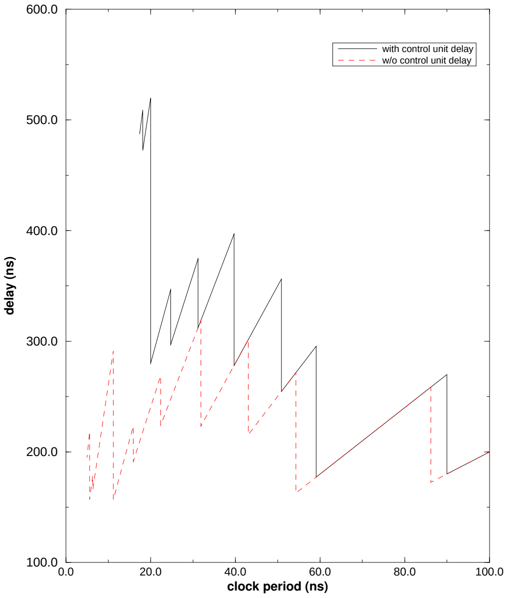

Figure 19: The clock period vs delay shape functions of the AR filter example, generated with and without the control unit delay estimation

account, clearly, a clock period of 6 ns does not exist because the control unit delay alone would be larger than 6 ns. Furthermore, we observe that the difference between delays obtained with and without considering the control unit delay can be as large as 720 ns. Therefore, we conclude that the control unit delay contributes significantly in the clock period and neglecting the control unit delay may result in a bad choice of the clock period.

8 Conclusions and Future Work

In summary, we have presented a clock selection algorithm that, given a pipelined behavioral description and a throughput constraint, selects the clock period that leads to the minimal-area design.

We tested our clock-selection algorithm on several examples and the results show that, in most cases, our algorithm selects a clock period that uses minimal area resources within less than one second.

We conducted an experiment to demonstrate the impact of resource sharing across pipe stages on the cost of a pipelined design. From the results, we note that, for all the benchmarks, the cheapest designs obtained for a given throughput are those with resource sharing.

The experimental results also show that the control unit delay contributes significantly to the clock period, and thus plays an important role in clock period calculation. Most importantly, neglecting the control unit delay may result in a bad choice of the clock period.

We plan to extend our model to incorporate wire delays. Currently, we are working on a clock selection algorithm that allows multiple clock signals.

9 Acknowledgements

This work was partially supported by the Semiconducter Research Corporation grant #94-DJ-146, and we gratefully acknowledge their support. We would also like to thank Viraphol Chaiyakul for his help and suggestions in defining the problem and developing the algorithm.

10 References

[1] S. Chaudhuri, S. A. Blythe, and R. A. Walker, "An Exact Methodology for Scheduling in a 3D Design Space," in *Proceedings of the 8th International Symposium on System Synthesis,* 1995.

[2] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[3] D. Le. Gall, "MPEG: A Video Compression Standard for Multimedia Applications," in *Communications of the ACM 34,* 1994.

[4] R. Jain, A. C. Parker, and N. Park, "Module Selection for Pipelined Synthesis," in *Proceedings of the 25th ACM/IEEE Design Automation Conference,* 1988.

[5] S. Narayan, and D. D. Gajski, "System Clock Estimation based on Clock Slack Minimization," in *Proceedings of the European Design Automation Conference,* 1992.

[6] N. Park, and A. C. Parker, "Synthesis of Optimal Clocking Schemes," in *Proceedings of the 22nd ACM/IEEE Design Automation Conference,* 1985.

[7] A. C. Parker, T. Pizzaro, and M. Mlinar, "MAHA: A Program for Datapath Synthesis," in *Proceedings of the 23th ACM/IEEE Design Automation Conference,* 1986.

[8] VLSI Technology Inc., *VDP370 1.0 Micron CMOS Datapath Cell Library,* 1991.