Estimation of Schedules for Control/Datapath Pipelining

Nong Fan†
Daniel D. Gajski‡

Technical Report #96-36
August 22, 1996

†Department of Electrical and Computer Engineering
‡Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697
(714) 824-8059

nfan@ics.uci.edu
gajski@ics.uci.edu

Abstract

Scheduling estimation in system level design plays an important role in estimating design metrics of a hardware implementation. Moreover, accurate estimates based on a realistic design model are usually expected. In this report, we present techniques for scheduling estimations based on a control/datapath pipelining architecture. Schedules are computed under both resource and architectural constraints. The resource allocation includes not only functional units but also storage and communication units. Our techniques enable designers to obtain fast and accurate estimates of scheduling.

Contents

List of Figures

1 Introduction

In system design, a system's functionality is usually divided among system components, such that constraints on various design metrics, such as performance, area and power dissipation, are satisfied. In order to determine if these constraints have actually been satisfied, we must be able to obtain metric estimates as rapidly as possible. However, fast estimation of design metrics cannot be achieved by synthesizing a complete implementation and then measuring the design quality metrics. Therefore, estimation of design quality at the system level is needed for at least two reasons. First, it enables the system designer to explore design alternatives by providing quick feedback for any design decision. Second, it enables the designer to evaluate the design quality by comparing the estimates of any design metric with the constraints specified for that metric.

Among the quality metrics commonly used to characterize a design, cost and performance of hardware and software implementations are the most important ones because many high-level decisions are based entirely on these two metrics. Gong *et al.* [5] proposed methods to estimate program size, data memory size and execution time of a software implementation. In this report, we will concentrate on scheduling estimation of a hardware implementation because of its important role in estimating hardware quality metrics. By estimating which operations are performed within each control step, it is easy for us to estimate metrics, such as area, execution time and power dissipation. Particularly, in order to obtain accurate estimates, schedule is computed here based on different target architecture types which may have pipelined or non-pipelined datapath with pipelined or non-pipelined control path. In addition to estimating a specification containing straight line codes, methods of dealing with conditional and loop constructs are also proposed. Since real functional units may have different propagation delays, different number of pipeline stages, and perform several type of operations, the datapath must allow for operation chaining, multicycle and pipelined operations. A given behavioral description is scheduled under resource constraints, where the resource includes not only functional units, but also read/write ports of storage units and communication buses.

The rest of the report is organized as follows. In the next section, we will review previous research in this area. Our design model will be explained in section 3 followed by two scheduling algorithms based on this model. Section 6 presents some experimental results. Finally, section 7 concludes the report and provides future work.

2 Previous Work

Several previous papers [4] have addressed the issue of estimating the number of control steps needed for a behavior's execution at the system level under resource constraints. They can be classified into two categories, i.e, operator-use method and scheduling method.

2.1 Operator-use method

The operator-use method [4] divides all statements in a behavior into a set of basic blocks in such a way that all statements in a basic block can execute concurrently. Let $num(t_i)$ and $clocks(t_i)$ represent the number and delay (in clock cycles) of functional units available to implement operations of type t_i. Then, if there are $occur(t_i)$ occurrences of an operation type t_i in any basic block, then at least $\lceil \frac{occur(t_i)}{num(t_i)} \rceil \times clocks(t_i)$ control steps are needed to execute operations of type t_i. The number of control steps needed for any basic block is equal to the maximum number of control steps needed to perform operations of any type in the basic block.

The operator-use method is illustrated with the example shown in Figure 1(a). Let us assume that two adders, each with delay of one clock cycle, are available in the resource allocation. Because there is a read-after-write dependency between statement (1) and (2), they can not be executed within the same control step. Thus the first basic block contains one statement. Statements (2) to (4) belong to the second basic block because no dependencies exist among them and they can be executed concurrently if enough resources are available. According to the operator-use method, three control steps are needed to execute the behavior as shown in Figure 1(b).

Since the operator-use method operates at a statement-level granularity and ignores the dependencies between the operations within a statement, the operator-use method can provide fairly rapid estimates of the number of control steps required by a behavior.

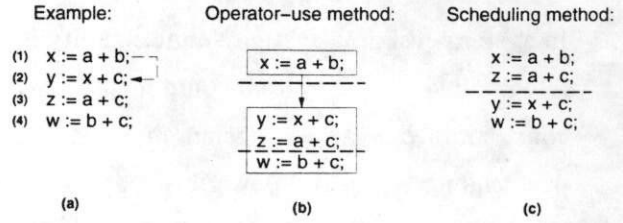


Figure 1: Comparison of operator-use and scheduling methods

2.2 Scheduling method

A scheduling technique may be applied to the behavior in order to determine the number of control steps as well as the scheduling of operations. Given resource constraints, the operations in the behavior are scheduled with a goal of minimizing the total number of control steps. One of the most popular algorithms for scheduling operations under resource constraints is list scheduling. Compared with other scheduling techniques [3], list scheduling is more appropriate for the purpose of design space exploration at system level because of its simplicity and efficiency.

The list scheduling algorithm maintains a priority list which contains a set of ready operations that have all their predecessors already scheduled. At each control step, the algorithm tries to schedule the operations with the highest priority under the resource constraints. Scheduling those operations makes other operations ready and they are inserted into the priority list. The process repeats until all operations in the given behavior have been scheduled. Applying this algorithm to the example in Figure 1(a), two control steps are needed as shown in Figure 1(c).

The operation in statement (3) can be scheduled in step one because it is ready at the first step. For this example, the minimum number of control steps is two.

While the scheduling method allows the designer to obtain more accurate estimates of scheduling, it is computationally more expensive than the operator-use method. The complexities of the two methods are $O(n^2)$ and $O(n)$, respectively, where n is the number of operations in the behavior. However, because the schedule is an important issue in estimating hardware quality metrics, an accurate estimate of scheduling is usually expected.

2.3 Limitations of previous scheduling algorithms

Without considering realistic design issues, even scheduling methods can not produce accurate results. To the best of our knowledge, no scheduling algorithm has been published which takes all the following realistic issues into consideration at the same time:

- In addition to straight line code, behavioral descriptions usually contain both conditional and loop constructs.
- Functional units may have varying delays.
- Operations can be chained within a single control step.
- Operations can take multiple clock cycles to complete its execution.
- Functional units can be pipelined.
- One functional unit can perform several different types of operations.
- One operation can be performed in several different functional units with different delays.
- Resource allocation includes not only functional units, but also storage units and communication units.

By considering all the above issues at the same time, our algorithms provide a more realistic scheduling estimation.

3 Design Model

Before presenting the algorithms, we will show the underlying design models used for our scheduling estimation.

Given the specification of a system for which estimates are to be computed for a hardware implementation, the corresponding design is assumed to be implemented as an finite state machine with a datapath (FSMD) [2], which consists of a datapath and a control unit, as shown in Figure 2. The datapath, consisting of functional units, storage units and communication units, is used to perform computations and data transfers. The control unit contains a state register and two combinatorial blocks computing the next-state and control signals.

This architecture may be defined in terms of a set of functional units, storage units and communication units as follows:

1. Each type of functional units is defined as

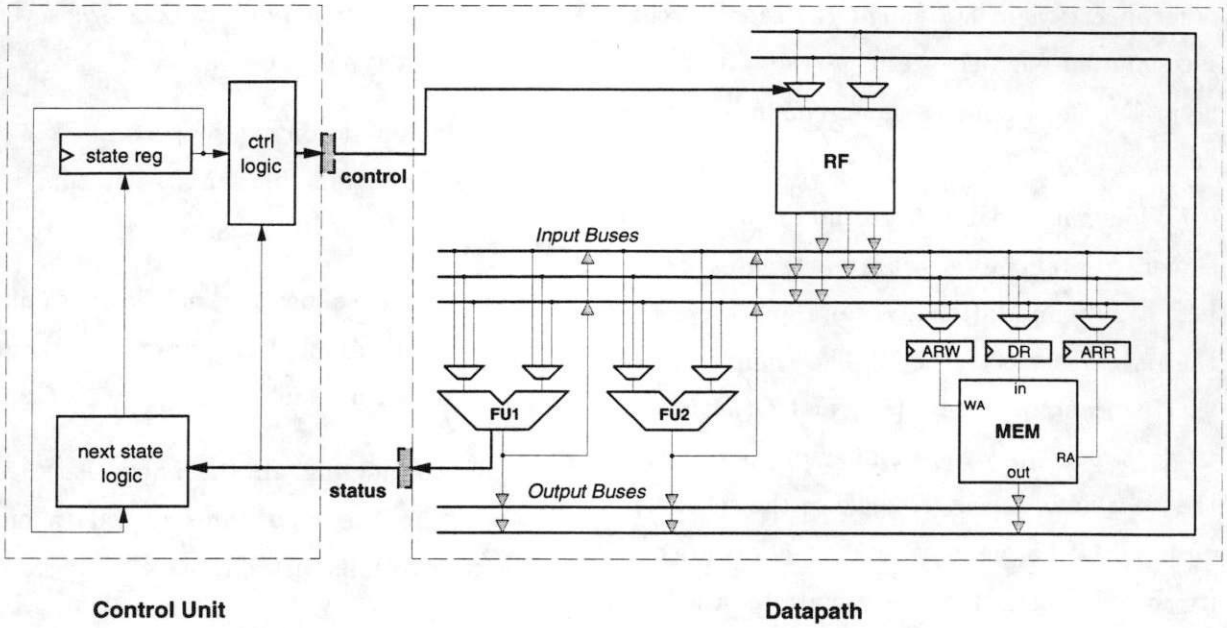


Figure 2: The general FSMD model

a triple, $f_i = (t_i, d_i, O_i)$, in which t_i is the number of available functional units of type f_i, d_i is the propagation delay of f_i, and $O_i = \{o_{i1}, o_{i2}, \cdots, o_{ij}\}$ is the set of operations f_i can perform.

2. Each type of storage units is defined as a quintuple, $s_i = (t_i, d_i, r_i, w_i, rw_i)$, in which t_i is the number of storage units of type s_i, d_i is the access delay of s_i, and r_i, w_i and rw_i are the number of read ports, number of write ports and number of read/write ports of s_i, respectively.

3. Communication units are defined by the number of input buses, *ibus*, and number of output buses, *obus*. Multiplexers are not specified explicitly. A multiplexer is inserted in front of a port if the port has more than one sources connected to it.

Each type of functional units can perform different types of operations and each type of operations can be performed on a different type of functional units. For storage units, currently we only support one register file and one memory module in our architecture with an access delay of one clock cycle for the register file and an arbitrary delay for the memory. Memory read operations are carried out by first setting the read address register, and memory write operations are performed by first setting the write address and data registers. Then, in the next cycle, data can be read from memory or written to it. The input buses are used to connect the read ports of the register file with the input ports of functional units or the write ports of memory module, and the output buses are used to connect the output ports of functional units or the read ports of memory module with the write ports of the register file.

According to our design model, the execu-

tion of a computational operation needs two read ports and one write port of register file to read operands and write results and two input buses and one output bus to move operands from and result back to register file. Memory read needs one read port of the register file to get the address and one read port of the memory to read data from it. One input bus is needed to set the read address register. Memory write needs two read ports of the register file to get the data and the address. Thus two input buses are needed.

If the specified clock period is long and unit delays are short, our design model allows functional unit chaining, that is, it performs two or more operations in a single clock cycle. On the other hand, if the unit delay for a functional unit is long compared to the clock cycle, our model allows both multicycle and pipelined functional units. Designers can select one of them by specifying a non-pipelined or pipelined datapath. If a pipelined datapath is specified, the number of pipeline stages of a functional unit is assumed to be the latency of the unit in terms of the number of clock cycles.

The longest register to register delay, which starts with the state register, through the control logic, the register file, components in the datapath and the next state logic returning the state register, determines the length of the clock cycle in our design model. In order to reduce the longest delay and make the clock run faster, the control path may also be pipelined [11] by inserting a status and/or a control register between control unit and datapath. Therefore, totally six different architecture types are supported by this model:

1. non-pipelined control path and non-pipelined datapath,
2. pipelined control path with status register and non-pipelined datapath,
3. pipelined control path with status and control register and non-pipelined datapath,
4. non-pipelined control path and pipelined datapath,
5. pipelined control path with status register and pipelined datapath, and
6. pipelined control path with status and control register and pipelined datapath.

The architecture type is specified as the architectural constraint in the design model.

4 Problem Definition

Our problem can be defined as follows:

Given is the following:

1. clock period, clk,
2. a behavioral specification in VHDL or SpecCharts,
3. architectural constraint T, and
4. resource allocation which includes:
   - a set of functional units, $\mathbf{F} = \{f_1, f_2, \cdots, f_n\}$,

- a register file, $s_1 = (1, clk, r_1, w_1, rw_1)$, and a memory module, $s_2 = (1, d_2, r_2, w_2, rw_2)$, and
- number of input buses *ibus* and number of output buses *obus*.

The goal is to find the number of control steps needed to execute the specification under the specified architectural and resource constraints.

This problem will be solved in two steps. First we will transform the specification into a CDFG internal representation. Then we will schedule the CDFG along each mutually exclusive path. The algorithm for transforming a behavioral specification into a CDFG is given in [7]. In this report, we will concentrate on the algorithms for scheduling the CDFG based on our design model. The number of control steps will be found by scheduling the specification using the algorithms.

5 Scheduling Algorithms

Our CDFG scheduling algorithm consists of a DFG scheduler and a CDFG scheduler. The DFG scheduler is used to schedule operations in a basic block, while the CDFG scheduler is used to schedule operations across basic blocks.

5.1 DFG scheduler

A data-flow graph (DFG) captures operational activity described by the VHDL assignment statements. By compiling the sequentially arranged assignment statements into a DFG, we can make the hidden parallelism more explicit. Our DFG scheduler assigns operations in the DFG into separate control steps with a goal of minimizing the total number of control steps. It is based on the list scheduling algorithm because of its simplicity and efficiency of scheduling operations under resource constraints. The main feature of the algorithm is that the delay of each functional unit does not need to be the same, each functional unit may perform more than one type of operations, each type of operations can be performed by different types of functional units with different delays.

To make the explanation of the algorithms easy, we assume that $\mathbf{F}$ is arranged in the ascending order of d_i, i.e., $d_i \leq d_j$, if $i \leq j$.

Let $\mathbf{O} = \{o_1, o_2, \cdots, o_m\}$ be the set of all types of operations that can be performed by $\mathbf{F}$, then we have

$$\mathbf{O} = O_1 \cup O_2 \cup \cdots \cup O_n$$

Let $d(o_i)$ be the minimum delay among all types of functional units which can perform operation o_i, i.e.,

$$d(o_i) = \min_{1 \leq j \leq n} \{d_j | o_i \in O_j\}$$

The total number of functional units, $t(o_i)$, which can perform operation type o_i, is defined as:

$$t(o_i) = \sum_{j=1}^{n} t_j \delta_j$$

where

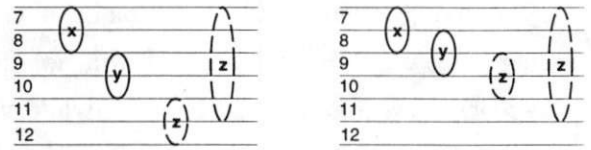
$$\delta_j = \begin{cases} 1 & \text{if } o_i \in O_j \\ 0 & \text{otherwise} \end{cases}$$

Our DFG scheduler inserts ready operations, which have all their predecessors scheduled, into

a priority list with a three level key. The first level is the operation's mobility which is calculated as the difference between the latest and the earliest start time of the operation. The smaller the difference, the higher the priority because a smaller value of mobility indicates a higher urgency for scheduling an operation. If two operations have the same mobility values, the second level key is used such that the one with a smaller $t(o_i)$ is given a higher priority because the scheduler will run out of alternative functional units sooner. In case two operations have the same values for the first two level keys, the third level key is used which is the number of immediate successor operations: an operation with more immediate successors is scheduled earlier because it makes more of these operations ready.

At each control step, the DFG scheduler always tries to use the fastest functional unit to perform the ready operations. If the fastest functional unit is not available, a decision should be made on whether to use slower ones. Because the functional units can be pipelined or non-pipelined, which is specified as the architectural constraint T, different decisions will be made.

In case the fastest functional unit has already been assigned to other operations, whether or not to use a slower functional unit is determined as follows. Let us consider non-pipelined functional units first. As shown in Figure 3(a), suppose that at current control step 7 there are three additions x, y and z are ready to schedule with their priorities from high to low and there are two adders available in **F** with delay of 2 and 5 clock cycles, respectively. If the three additions are executed only with the faster adder, it will take six clock cycles to finish them. However, if we use the fast adder to perform x and the slower adder to perform z at step 7, then five cycles are needed to finish them. In case of pipelined functional units as shown in Figure 3(b), four cycles are needed if all of them are executed using the faster adder while five cycles are needed if z is executed with the slower one at step 7.



(a) non–pipelined functonal units (b) pipelined functional units

Figure 3: Example of binding checking

In general, the conditions under which the operation x should be assigned to a slower functional unit with delay d_i, $d_i \geq d(Type(x))$, such that the operation can be finished in the earliest possible state are given as the following two rules:

Rule 1 *For a non-pipelined functional unit f_i, the conditions are*

$$d_i \leq Mobility(x) + d(Type(x)) \quad \textbf{or}$$

$$w \geq \lfloor \frac{d_i - d'_1}{d'_1} \rfloor + \lfloor \frac{d_i - d'_2}{d'_2} \rfloor + \cdots + \lfloor \frac{d_i - d'_i}{d'_i} \rfloor.$$

Rule 2 *For a pipelined functional unit f_i, the conditions are*

$$d_i \leq Mobility(x) + d(Type(x)) \quad \textbf{or}$$

$$w \geq (d_i - d'_1) + (d_i - d'_2) + \cdots + (d_i - d'_i).$$

In the above rules, $d(Type(x))$ is the delay of the fastest functional unit able to perform operation x, and w is the number of operations which have higher priority than x and ready to be scheduled in the current control step but are not scheduled because they do not satisfy either condition, like operation y in Figure 3 at step 7. d'_j's, $1 \leq j \leq i$, are the delays of those functional units which have smaller delays than f_i and have already been assigned to execute other operations at the current step.

Satisfying the first part of the condition means using a slower unit to execute x will not delay the worst case completion time of x, while satisfying the second part means that it is better to use the slower one because there are too many operations waiting for the faster units.

Let us assume the mobility values for operations x, y and z are 0, 1 and 2, respectively. Figure 3(a) shows the case of non-pipelined functional units. At step 7, operation x is considered for scheduling first because it has the highest priority. The fastest adder with delay two in the allocation is assigned to x. The delay of the second fastest adder for y is 5 and no operations are waiting for the first adder. The conditions in **Rule 1** becomes $5 \leq 1+2$ or $0 \geq \lfloor (5-2)/2 \rfloor$. Because neither of them are satisfied, y will not be scheduled to step 7 and it will wait for a faster adder than the current one. Then w becomes 1. For z, the conditions are $5 \leq 2+2$ or $1 \geq \lfloor (5-2)/2 \rfloor$. It is assigned to this adder because the latter is met. Thus, x and z get scheduled at step 7 if the functional units are not pipelined.

In case of pipelined functional units, the conditions for y and z are $5 \leq 1+2$ or $0 \geq (5-2)$ and $5 \leq 2+2$ or $1 \geq (5-2)$, respectively. None of them is met, so only x gets scheduled at step 7.

```
Algorithm: DFG Scheduler
  input: a DFG, clk, T, and resource allocation;
  output: schedule;
begin
  Cstep = 0;
  while (there are operations to schedule) do
    Cstep = Cstep + 1;
    Reset(Wait);
    Compute_Mobility(DFG);
    Insert_Ready_Op(DFG, PList);
    while (PList ≠ φ) do
      Op = First(PList);
      Funit = Find_Avail_Funit(F, Op);
      if (Binding_Check(Op, Funit, T, Wait) = Ok)
      then
        if (Check_Ports_Buses(s1, s2, ibus, obus) = Ok)
        then
          Update_Schedule(S, Cstep, Op, Funit);
        end if;
      else
        Wait[Type(Op)]++;
      end if;
    end while;
  end while;
end.
```

Figure 4: General DFG scheduling algorithm

The DFG scheduler is described in Figure 4. For each type of operations, the element of the array $Wait$ stores the number of operations which are ready to be scheduled but are not scheduled because waiting for faster ones may speed up their completion time. $Cstep$ represents the current control step into which operations are being scheduled. The procedure $Compute\_Mobility$ computes the mobility values of the unscheduled operations in the DFG by using the ASAP and ALAP algorithms. The procedure $Insert\_Ready\_Op$

inserts the ready operations in the DFG into the priority list *PList* using as a three level key the mobility, number of functional units which can perform the operation and the number of immediate successor operations. Given a list, the function *First* returns the first element of the list and removes it from the list. The function *Find_Avail_Funit* scans the available functional unit in **F** in the ascending order of delay and tries to find the fastest one which is able to perform the type of operation *Op*. The function *Binding_Check* decides whether or not the scheduler should use the available fastest functional unit *Funit* to perform operation *Op* based on the above two rules. The function *Check_Ports_Buses* checks if there are enough ports and buses to fetch operands and write results. The procedure *Update_Schedule* modifies the schedule S by assigning the specified operation to the current control step.

Compute_Mobility calculates the earliest and the latest start time of each unscheduled operation in the DFG by using $d(o_i)$ as the delay of the operation if its type is o_i. During each control step, the mobility values of unscheduled operations need to be recomputed only if the scheduler could not use the fastest functional units to perform any operations in the previous step.

5.2 CDFG scheduler

A behavioral specification usually contains control constructs, like conditional branches and loops. To schedule the operations specified in it, the input textual description is first transformed into a control dataflow graph (CDFG) internal representation, where control constructs are mapped to control-flow nodes and assignments within basic blocks are mapped to data-flow nodes [7]. Figure 5 shows a behavioral description and its corresponding CDFG. The goal of our CDFG scheduler is to find the minimum number of control steps needed to execute the operations in a CDFG under the given resource and architectural constraints.

The architecture type specified as the architectural constraint plays an important role in scheduling condition evaluations and mutually exclusive operations [11]. Mutually exclusive operations can be scheduled either into different control steps or into the same steps. The result of condition evaluation may be available to the control unit in the same state as it is generated or in the next state after it is generated. Also the control signals may be available to the datapath in the same state as they are generated or in the next state after they are generated. These will be illustrated as follows by using the example shown in Figure 5.

Figure 5 shows a specification in a high level language and its corresponding CDFG representation. The specification includes one condition evaluation operation, $x > n$ and two mutually exclusive operations, y/n and $y + x$. Suppose that the resource allocation includes one adder, one divider and one comparator with delays of one, two and one clock cycle, respectively. We will consider the impact of different kinds of control pipelined architecture types on the schedul-

ing method.

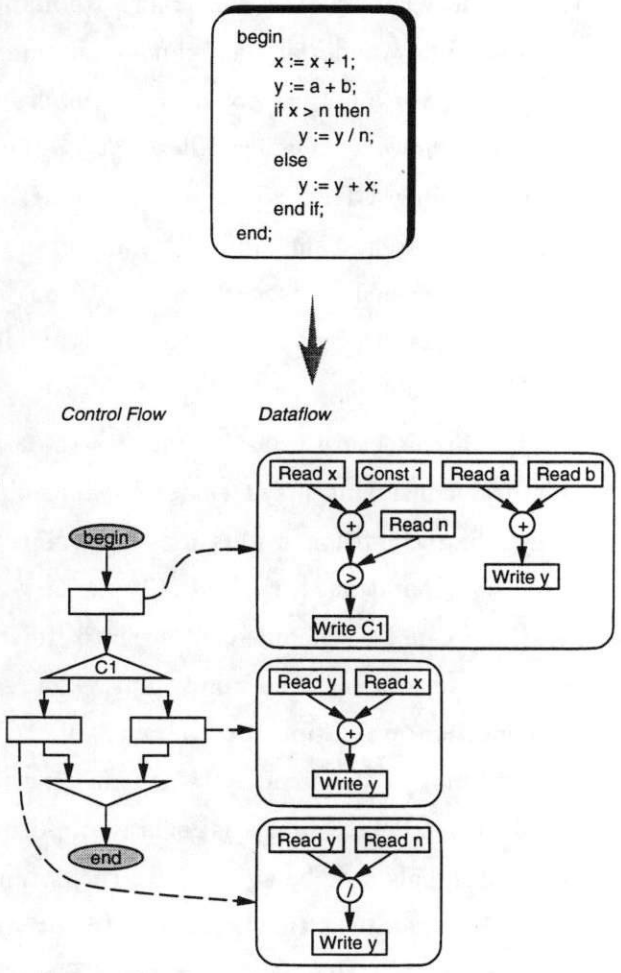


Figure 5: A sample specification and its CDFG representation

First let us consider the case of non-control pipelined architecture type without status and control registers. As shown in the state action table of Figure 6(a), the condition evaluation operation is computed in S_1 and the result is immediately available to the controller. Without the status register, the architecture provides no place to store the result of the condition evaluation for future use. Therefore, it is impossible for mutually exclusive operations to share the same state in order to reduce the number of control states. The execution of the mutually exclusive operations must be scheduled into different states, like S_2 and S_4, respectively. The result of condition evaluation must be tested in the same state as the condition evaluation operation, i.e., S_1, to determine whether the next state is S_2 or S_4. At the subsequent clock rising edge, the selected next state is stored in the state register and control signals are generated which are immediately available to the datapath.

Rule 3 *For non-control pipelined architectures, mutually exclusive operations should be scheduled into different control steps and the next state may be determined immediately after the condition evaluation.*

We now consider the control pipelined architecture with only status registers. The condition evaluation is scheduled in S_1 and the result is stored in the status register. Thus, the result is not available to the controller for testing until next state S_2. At state S_2, based on the result of the condition, the decision can be made on which of the mutually exclusive operations should be carried out and the control signals generated at S_2 are immediately available to the datapath.

Rule 4 *For control pipelined architectures with only status register, mutually exclusive operations may be scheduled into the same control steps and their execution should begin one state later than the condition evaluation.*

Finally, we come to the control pipelined architecture with both status and control registers.

Because of the control register, the control signals generated in S_3 cannot reach the datapath until S_4. Therefor, a NOP operation is scheduled in S_3. Also, the result of condition evaluation stored in the status register is used in two states after it is generated in S_2. It is first used in S_3 to generate the appropriate control signals to be used in S_4. Then it is used in S_4 to decide what the next state is.

Present state	*Next state*		*Datapath actions*	
	condition	*state*	*condition*	*operations*
S0		S1		x := x + 1;
S1	C1 = True C1 = False	S2 S4		C1 := x > n; y := a + b;
S2		S3		(y := y / n);
S3		-		y := y / n;
S4		-		y := y + x;

(a) Non Control Pipelined Architecture

Present state	*Next state*		*Datapath actions*	
	condition	*state*	*condition*	*operations*
S0		S1		x := x + 1;
S1		S2		C1 := x > n; y := a + b;
S2	C1 = True C1 = False	S3 -	C1 = True C1 = False	(y := y / n); y := y + x;
S3		-		y := y / n;

(b) Status Pipelined Architecture

Present state	*Next state*		*Datapath actions*	
	condition	*state*	*condition*	*operations*
S0		S1		NOP
S1		S2		x := x + 1;
S2		S3		C1 := x > n; y := a + b;
S3		S4		NOP
S4	C1 = True C1 = False	S5 -	C1 = True C1 = False	(y := y / n); y := y + x;
S5		-		y := y / n;

(c) Control Status Pipelined Architecture

Figure 6: State action tables for different architecture types

Rule 5 *For control pipelined architectures with both status and control registers, mutually exclusive operations may be scheduled into the same control step and their execution should begin two states later than the condition evaluation.*

Our CDFG scheduling algorithm takes as the inputs the CDFG, the resource and architectural constraints and the start state from which the operations are to be scheduled. It recursively traverses the nodes in the CDFG along mutually exclusive paths. It schedules operations contained in the DFG node using our DFG scheduling algorithm. At the FORK node, it decides when to schedule mutually exclusive operations based on the architectural constraint. The mutually exclusive paths get merged at the JOIN nodes. The number of control steps for non-control pipelined architecture is the sum of the steps of each node along all paths, while the number of control steps for control pipelined architectures is the length of the longest path of the CDFG because the mutually exclusive operations can share the states under these architectures.

Figure 7 describes our recursive CDFG scheduler. The function *DFG_Scheduler* is similar to our DFG scheduler discussed before. The difference is that it starts scheduling operations from the specified control step instead of from state one and it returns the last control step of the schedule along that particular path so that the schedule of the succeeding CDFG node can be accumulated with it. At each FORK node, the result of the condition evaluation is used to select

```
Algorithm: CDFG Scheduler
  input: root, next_step, T, and resource allocation;
  output: schedule;
begin
  if (root is DFG_NODE) then
    next_step = DFG_Scheduler(root, next_step);
  else if (root is FORK_NODE) then
    next_step = Find_Cond_Evaluate_State(S);
    if (T is status pipelined) then
      next_step = next_step + 1;
    else if (T is status+control pipelined) then
      next_step = next_step + 2;
    end if;
  else if (root is JOIN_NODE) then
    if (all predecessors scheduled) then
      next_step = Longest_Path(S) + 1;
    else
      return;
    end if;
  end if;

  for each successor of root do
    CDFG_Scheduler(child of root, next_state);
  end for;
end.
```

Figure 7: CDFG scheduling algorithm

which mutually exclusive path should be taken. The function *Find_Cond_Evaluate_State* finds the control state during which the condition evaluation is scheduled. At each JOIN node, the function *Longest_Path* returns the number of control states along the longest path from the beginning node of the CDFG to the JOIN node.

If the current CDFG node contains a DFG, the CDFG scheduler uses our DFG scheduler to schedule the operations in the DFG starting from the specified control state. After scheduling it, the CDFG scheduler continues scheduling the succeeding CDFG nodes. If the current node is a FORK node, it first finds when the condition evaluation is performed, and then it schedules operations along each mutually exclusive paths separately. The starting state is determined according to the architectural constraint. If the architecture type is non-control pipelined, the mutually exclusive operations can be scheduled right after the condition evaluation. If the architecture type is control pipelined with only status register, the mutually exclusive operations can only be scheduled one state later then the condition evaluation. If the architecture type is control pipelined with both status and control registers, the operations can only be scheduled two states later than the condition evaluation. If the current node is JOIN node, its successor nodes will not be traversed until all its predecessors have been traversed.

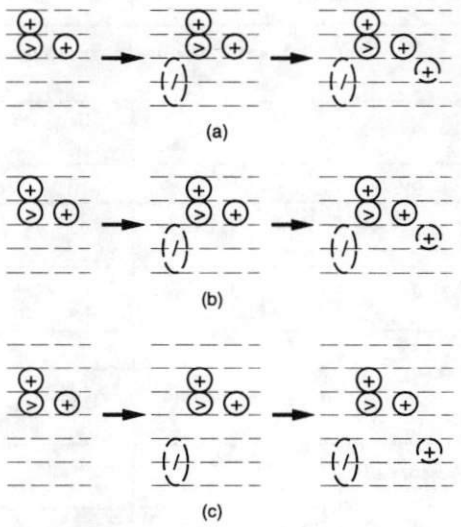


Figure 8: Generated schedules for different architecture types: (a) non-pipelined control, (b) pipelined control with status register, (c) pipelined control with status and control registers.

Figure 8 depicts the scheduling process for different architecture types. The nodes in the CDFG are traversed from up to bottom and from left to right if branches occur. For each architecture type, the figures show from left to right the

intermediate results after travsersing and scheduling each CDFG node.

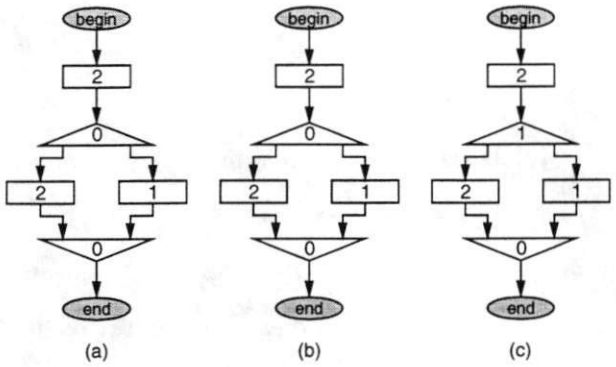


Figure 9: Number of control steps needed to execute each CDFG node

Figure 9 shows the number of control steps needed to execute each CDFG node. The number of control steps for non-control pipelined architecture is obtained by summing the number of control steps for each CDFG node as shown in Figure 9(a). The number of control steps for both pipelined architectures is easily obtained by checking the length of the schedule in Figure 8(b) and (c), respectively. In all the cases, the numbers are the same as the ones obtained by manually scheduling the CDFG with state action tables, which are 5, 4, and 6, respectively.

6 Experimental Results

We have incorporated our architecture based scheduling method into our system design tool, SpecSyn [4]. To compare our method with previous operator-use method, we assume the infinite number of ports and infinite number of buses in our design model and use the architecture type of pipelined control path with status register and non-pipelined datapath. The allocation of datapath components is shown in Figure 10. The delays of the components are obtained from the VDP 100 datapath library [14] as examples. Let us assume that the clock period is 60ns.

Component	Delay(ns)	Num
MUL	163	1
DIV	163	1
ADD	49	2
SUB	56	1
COMP	23	1
ABS	56	1

Figure 10: Resource allocation with delays

Benchmark	Manual design	Operator_use method	error1	Our algorithm	error2
hal	21	23	**9.5**	21	**0.0**
median	20	12	**-40.0**	21	**5.0**
dct	17	12	**-29.4**	17	**0.0**
beamformer	212	139	**-34.4**	207	**-2.4**
volume	56	66	**17.9**	53	**-5.4**
jacobian	418	339	**-18.9**	413	**-1.2**
kalman	23	43	**87.0**	22	**-4.3**
elliptical2	30	45	**50.0**	30	**0.0**

$$\textbf{error1} = \frac{\text{Operator\_use} - \text{Manual}}{\text{Manual}} \times 100\%$$

$$\textbf{error2} = \frac{\text{Our} - \text{Manual}}{\text{Manual}} \times 100\%$$

Figure 11: Number of states for different benchmarks for manual design, operator-use and our algorithms

We have tested 8 examples written in SpecCharts to compare the results obtained with different methods. For each example, we first manually schedule it by building the state action table and then use the result as our reference. Some examples may includes several behaviors and they are scheduled separately. The results obtained by using operator-use method and our scheduling method are compared with the man-

ually scheduling results to see how accurate they are. As shown in Figure 11, the errors for operator-use method can range from -40 percent to 87 percent, while the errors for our scheduling method only range from -5 to 5 percent. Particularly, 5 out of 8 benchmarks have errors less than 2.5 percent. The results for control intensive applications, like volume, are not relatively accurate because of wait statements in the specifications.

7 Conclusion and Future Work

In this report, we have presented our design model which supports six different target architecture types. Based on this model, we have presented two scheduling algorithms to schedule operations in a DFG and/or CDFG. A set of experiments have been conducted to show how accurate our schedule is.

Our future work includes obtaining more accurate results on datapath area estimation based on our scheduling estimation. The datapath area consists of the area of functional units, storage units and communication units.

8 Acknowledgments

This work was partially supported by the grant from Toshiba Corporation, and we gratefully acknowledge their support. We would also like to thank Smita Bakshi and H-P Juan for their help in defining the problem.

9 References

[1] R. Camposano, "Path-Based Scheduling for Synthesis," in *IEEE Trans. CAD, Vol.10, No.1,* Jan. 1991.

[2] D. D. Gajski, *Principles of Digital Design,* Prentice Hall, 1997.

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

[4] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems,* Prentice Hall, 1994.

[5] J. Gong, D. Gajski and S. Narayan, "Software Estimation from Executable Specifications," in *Journal of Computer and Software Engineering,* 1994.

[6] R. Jain, "MOSP: Module Selection for Pipelines Designs with Multi-Cycle Operations," in *Proc. of the IEEE Conference on Computer Aided Design,* 1990.

[7] A. Orailoglu and D. Gajski, "Flow Graph Representation," in *Proc. of the 23rd Design Automation Conference,* 1986.

[8] B. Pangrle and D. Gajski, "State Synthesis and Connectivity Binding for Microarchitecture Compilation," in *Proc. of the IEEE Conf. on Computer Aided Design,* 1986.

[9] U. Prabhu and B.M. Pangrle, "Super-pipelined Control and Data Path Synthesis," in *DAC Proceedings,* June'92.

[10] T. Kim, J.W.S. Liu and C.L. Liu, "A Scheduling Algorithm for Conditional Resource Sharing," in *Proc. ICCAD,* 1991.

[11] L. Ramachandran and D. Gajski, "Architectural Tradeoffs in Synthesis of Pipelined Controls," in *Proc. of the European Design Automation Conference (EuroDAC),* 1993.

[12] K. Wakabayashi and T. Yoshimura, "A Resource Sharing Control Synthesis Method for Conditional Branches," in *Proc. ICCAD,* 1989.

[13] W. Wolf, "Hardware-Software Co-Design of Embedded Systems," in *Proc. of the IEEE, Vol. 82, No. 7,* 1994.

[14] VDP100 1.5 Micron CMOS Datapath Cell Library, 1988.