Synthesis of[1] Optimal Clocking Schemes for Digital Systems

Technical Report CRI-85-32
(DISC 84-1 - May 4 1984)

Nohbyung Park and Alice C. Parker

[1] This research was supported by Army Research Office Grant DAAG29-80-K-0083 and the International Business Machines Corporation, Contract #S 956501 QLX A B22.

Table of Contents

List of Figures

1 INTRODUCTION

The research discussed here will examine some important problems of synthesis of digital systems. In particular, the focus will be on some specific design decisions which produce a register-transfer hardware implementation of a digital system with near optimal speed under certain design constraints and desired optimization goals. We will also consider optimizing the speed of an already designed system by reconfiguring the interconnections of the hardware modules. The discussion which follows will start with a brief overview of high-performance control styles of digital systems and then focus on the area of speed optimization of digital systems with centralized controllers.

1.1 Speeding Up Digital Systems

Although there are many styles and variations of techniques for high performance digital systems control, they can be classified as following two basic concepts:

- distributed processing under asynchronous distributed control
- overlapped (parallel) processing under centralized control

The former class includes digital systems with multiple autonomous control sequencers such as multi-microprocessor systems, VLSI circuits with multiple autonomous control modules and interfaces (e.g., a UART with separate sequencers for receiver and transmitter). The latter class includes systems or modules with only a single centralized controller. Any system belonging to the first class can be partitioned into subsystems and/or modules each of which can be classified under the latter class, although there are several complex control partitioning problems which must be addressed. The overall speed of such a distributed processing system will be determined as a function of the speed of each partitioned subsystem and/or module. Accordingly, we will focus our discussion on the latter case, overlapped processing under centralized control.

1.2 Two Sequencing Levels of a Digital System

In digital systems with two-level control structures, sequencing is carried out in two levels, the **macro** and **micro** levels. An execution instance of a machine instruction or a major loop of an f.s.m. (**macro task**) corresponds to a **macro cycle** and an execution instance of a microinstruction or a state of an f.s.m (**micro task**) corresponds to a **micro cycle**, which are carried out by a **macro engine** and a **micro engine**, respectively. Most Von Neumann type computer CPU's and simple digital systems have a two-level control structure. In most digital systems whose control structure has more than two levels, we can also find similar levels of sequencing corresponding to the macro and micro levels. For such a system, by properly merging levels of sequencing, we can also partition the sequencing of the system into two levels similar to the macro and micro levels.[1] Figure 1 shows an example of a microprogrammed computer CPU.

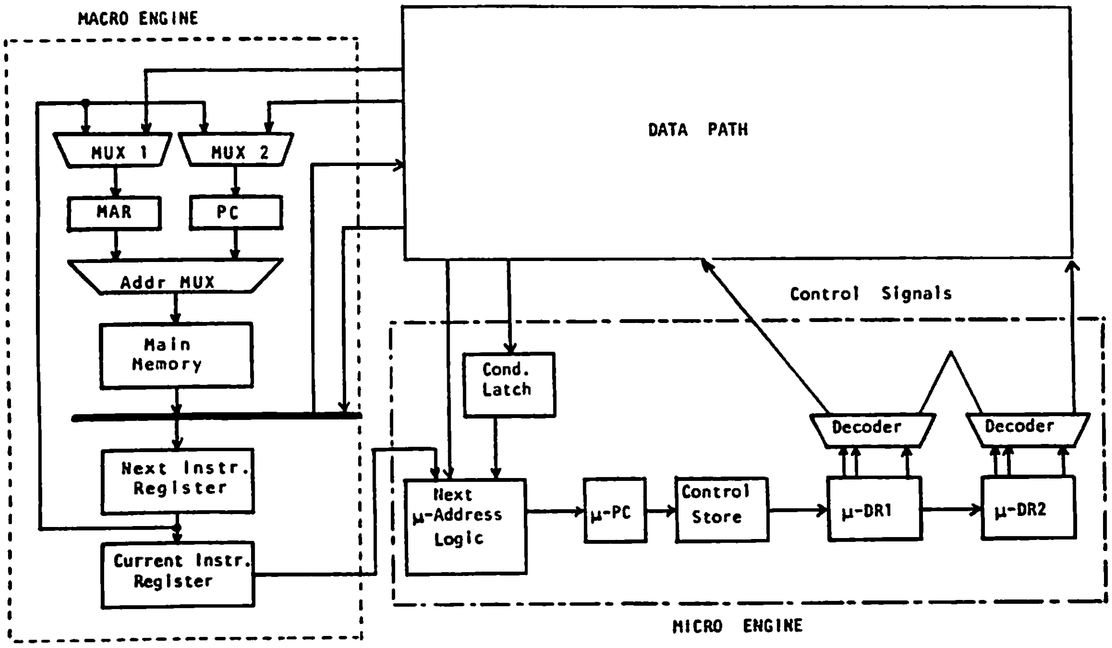


Figure 1: Sequencing Engines of a Digital System

Macro cycles consist of sequences of one or more micro cycles. Overlapping macro cycles are implemented by proper partitioning of macro cycles into sequences of micro cycles. *For example, an operand needed by the current macro cycle could be fetched during some micro cycle of the previous macro cycle and some micro cycle of the current macro cycle may fetch the next macro cycle in advance.*

[1] For a nano-programmed CPU such as the Nanodata QM-1, nanoinstruction cycles can be considered as the micro cycles, and microinstruction cycles and machine instruction cycles can be merged and considered as the macro cycles of our classification.

A micro cycle consists of **minor** cycles. Each minor cycle reads, transforms and stores data and/or control values from storage elements to storage elements which are used to buffer the flow of the values between functional elements. Such buffering storage elements are called **stage latches**. For the micro engine of figure 1, μ-PC, μ-DR1, μ-DR2 and "Cond. Latch" can be considered to be stage latches. In general, any storage element in the system can be a stage latch.

1.3 Overlapped Execution in Micro-level Sequencing

At the macro level, various techniques for speeding up digital systems exist. Examples are instruction look-ahead [25], stack architectures [32] and dataflow machines [13]. However, the ultimate performance of these speed-up techniques will depend very much on good sequencing control schemes at the micro level, since each macro-level task is eventually implemented by micro cycles.

At the micro level, execution overlap is achieved by overlapping the execution of minor cycles of multiple micro cycles. As shown in Figure 2-(a), simple overlap is often used in small computer CPU's. As data path cycles overlap (b and c of Figure 2), overlap within functional units can also be used (e.g., the pipelined multiplier of the IBM 360/91). Possible places where micro-level overlap can be achieved are:

1. between stages of the micro engine

2. between the micro engine and the data path

3. between the data path stages

Figure 2 shows timing examples of micro cycles. Case (b) corresponds to the digital system of Figure 1, where, for a microinstruction i, the minor cycles Ii1, Ii2, Ii3 and Ii4 start by clocking μ-PC, μ-DR1, μ-DR2 and "Cond. Latch", respectively. If there is no conflict in stage usage and no branches are executed, the maximum execution speed of a micro engine is determined by the longest interstage propagation delay (which is the minimal possible clock period) as in static pipelines without loops. Of course, the actual interstage propagation times depend on the number and length of the clock phases.

```
                                    Iij : j-th minor cycle of micro cycle i.

         |      I11       I12
         | |----------|----------|
control  |        (branching)        I21        I22
 flow    |                       |----------|----------|
         |                                      I31        I32
         |                                  |----------|----------|
         +-----------------------------------------------------------> time
            (a) Conventional 2-stage scheme with 2-phase clock


control  |  I11  I12  I13  I14
  flow   | |----|---|----|-----|
         |           (branching)    I21  I22  I23  I24
         |                          |----|---|----|-----|
         |                                 I31  I32  I33  I34
         |                                |----|---|----|-----|
         +-----------------------------------------------------------> time
       (b) 4-stage scheme with 2-stage data path (needs 4-phase clock)


control  |  I11  I12 I13 I14 I15
  flow   | |----|---|----|--|--|
         |          (branching) I21  I22 I23 I24 I25
         |                      |----|---|----|--|--|
         |                            I31 I32 I33 I34 I35
         |                           |----|---|----|--|--|
         +-----------------------------------------------------------> time
       (c) 5-stage scheme with 3-stage data path (needs 5-phase clock)
```

Figure 2: Examples of Micro Cycle Sequencing (Gantt Chart)

As shown in Figure 2, a branch delays fetch of the next micro cycle until the earliest fetch clock phase after the completion of the branch. Accordingly, the time delay due to a branch is a function of the execution time of the branch and the clock period. In other words, overall performance of the micro engine also depends on the sequence of micro cycles to be executed.

Resource conflict and data dependency are other factors reducing the advantage of execution overlap. For example, two microinstructions, I_i and I_{i+1}, are executed consecutively and each has three microoperations (data path cycles), as follows:

	I_i			I_{i+1}	
I_{i1}:	C <-- MDR	NEXT	$I_{i+1,1}$:	C <-- D*2	NEXT
I_{i2}:	A <-- B + C	NEXT	$I_{i+1,2}$:	C <-- C + 2	NEXT
I_{i3}:	A <-- A*2		$I_{i+1,3}$:	E <-- C + D	

$I_{i+1,1}$ has a data-dependency relation with I_{i1} and I_{i2} [11]. It also has a resource conflict (assuming only one multiplier) with I_{i3}. Thus $I_{i+1,1}$ cannot start execution until all the data path cycles of I_i complete. These examples can also be found in pipelines with loops [27]. Forbidden latencies of a pipeline are determined by resource conflicts between tasks and loops are major contributors to them. These problems slow down the execution speed as well as increase the complexity of the control circuitry. Similar problems arise in various levels of most digital designs [36, 23]. The higher the level, the harder the analysis and the higher the control cost. At the micro sequencing level, these problems can be analyzed in a formal way using a graph theoretic, algebraic methodology, which will be proposed by this research proposal.

1.4 Overview of the Research

In this technical report, we consider speeding up digital systems with centralized control at the micro level. The main objective of the research is to achieve maximal performance increase with minimal hardware cost and design effort.

Among the most costly and time consuming tasks at the early stages of data path design are module selection and allocation, which select functional and storage elements and assign functions and values to them. Also, during or after the module selection and allocation phases, control is synthesized, involving the synthesis of either a microprogrammed or a hardwired sequential machine. When near optimal design is required, all these tasks are computationally intractable (we will discuss this in Section 2.1). Furthermore, once all these tasks are completed, any non-trivial change in either control flow or data flow may require almost the same effort as the initial design. Naturally, we can think of the following two fundamental questions:

1. During the module selection and allocation phases of the data path design (assuming a fixed control sequence), how can we efficiently estimate and compare the performance of alternative decisions?

2. For a completed design, how can we increase the performance of the system at minimum hardware cost and minimum design and/or design change time?

The main goal of this research is to develop a methodology which can answer these questions at the micro level of digital systems. Obviously the emphasis of the speed-up techniques to be developed will be on minimizing the change in the control and data flow of a given partial or complete design.

If new operators are to be added to speed up the execution, both the data and control flow must be altered to get the maximum advantage of them. Also additional storage elements are often required in order to store the intermediate results which may exist in parallel. Thus, the task will involve almost the same amount of work as the initial design. For example, adding a new ALU for speed-up often requires rewriting the microprogram as well as changing the value allocated to the operand and/or result registers, which automatically involves changing the interconnections for both the data flow and control flow. In order to avoid such costly and time consuming iterations, we consider adding or reconfiguring only storage elements, which can be done without altering the basic structure of the original control and data flow and thus is considered to be transparent to the data flow and control flow analyzer. Assuming that the control sequence of the micro cycles is fixed, we consider two basic approaches to the problem:

1. For a set of chosen and allocated functional modules (for both the data path and the micro engine), add and connect minimal number of storage elements necessary to achieve a certain level of performance.

2. For a completed design, add certain number of storage elements to the data path and/or micro engine in such a way that the performance increase will be **maximized** by virtue of maximum execution overlap of the micro cycles.

In any case, we try to maximize execution overlap of the micro cycles considering the time overhead due to branches, resource conflicts and data dependency relations. Maximum execution overlap can be achieved by synthesizing an optimal clocking scheme, which involves the following tasks:

- optimal assignment, relocation, addition or deletion of the stage latches
- choice of an optimal clock period and the number and lengths of the clock phases
- optimal clock signal gating and routing.

In carrying out these tasks, we formulate the problem as a graph theoretic problem. Digital circuits are modeled by directed graphs which show the pathways of the data and control flow. By properly weighting the vertices and the directed edges, we can model the execution sequences of the micro cycles as tours on the graphs. Also, the time taken at each segment of the tours can be computed easily. Assigning and/or inserting overlap stage latches can be modeled as finding multiple edge-cut sets. Once the locations of the stage latches are determined, then the optimal clock period and clock sequence can be computed considering the synchronization overhead discussed before.

Chapter 2 gives motivation for the research and discusses previous work. The problem formulation is given in chapter 3. Chapter 4 presents the clocking scheme synthesis results. In Chapter 5, the result of static clocking scheme synthesis will be demonstrated. In the example in Section 5.1, the micro cycle time of a μ-programmed CPU is shown to be sped-up significantly.

2 MOTIVATION AND BACKGROUND

In this chapter, we discuss the general design environment of clocking scheme synthesis. A brief overview of the general digital design problem is followed by the definition of the task, **clocking scheme synthesis**. We also define the *speed* of digital systems, which will be used to evaluate the performance of clocking schemes and ultimately of digital systems. We conclude this chapter by postulating the necessity and importance of a good design methodology for clocking scheme synthesis.

2.1 The General Digital Design Optimization Problem

The general digital design problem is that of producing a hardware implementation of the system which exhibits a required behavior and satisfies any constraints imposed on it. Among the most typical design *constraints* are minimum required speed and maximum allowed cost and power consumption. Common optimization goals are maximizing speed and minimizing cost and power consumption within the constraints. Unfortunately, these optimization tasks often compete with each other. For example, the minimum cost implementation will rarely be the maximum speed implementation. For this reason, *desired design goals* are often used in addition to constraints in order to direct the optimization process towards a certain direction. Whenever there is more than one noninferior design alternative, the one that best meets the desired goals will be chosen. Examples of desired goals are to maximize speed, to maximize speed-to-cost ratio and to maximize speed-to-power consumption ratio.

Use of desired goals makes the design decision process unambiguous and efficient. Then, the synthesis task can be partitioned into subtasks as listed below, which will iterate and proceed towards the direction guided by constraints and desired goals:

1. Choose an appropriate design style (*design style selection*)

2. Choose potentially optimal sets of functional and storage modules which can maximize speed and minimize cost and/or power consumption (*module selection*).

3. Allocate operations and data values to functional and storage elements. Partial interconnection may also be carried out (*resource allocation*).

4. Find an optimal configuration and/or interconnection of modules so as to maximize performance. Detailed control hardware and/or microprogram are also synthesized during this phase (*configuration and interconnection*).

5. For a given design which is non-optimal, find a near-optimal reconfiguration of the design within an allowed cost increase or speed decrease limit (*performance increase or cost reduction by reconfiguration*).

In cases when near optimal solutions are desired, the complexity of these tasks is in decreasing order, since the solutions for the earlier phase problems can only be guaranteed to be optimal after a large number of (in worst case, all possible) solutions for the later phase problems are compared. Unfortunately, finding optimal solutions even for some of the later phase problems is known to be intractable. For example, the resource allocation problem can be modeled as a *job shop scheduling* problem, which is known to be NP-hard [21]. Also, as a subproblem of phase 4, the microcode compaction problem has been proven to be NP-Complete [37]. Many other problems with exponential complexity in various design phases have been identified [6, 38]. Only several problems of the last phase turn out to be polynomial time solvable [14, 30, 31].

In essence, synthesis tasks are carried out by estimating and evaluating cost and speed of feasible hardware implementations of the system at various stages of the design process. Naturally, in order to carry out these tasks efficiently and to get a near optimal design, a good estimation and evaluation strategy is crucial. Especially, in the last two phases, it is desired that the speed estimation and evaluation procedures be able to suggest possible changes in the given design which can increase the speed.

2.2 Definition of the Clocking Scheme Synthesis Task

As we have seen so far, the digital design problem is known to be computationally intractable. As one way of reducing complexity, synthesis of digital systems is usually partitioned into data path synthesis followed by control synthesis (this is true for both automated design systems [34, 15, 41] and human designers). In such design procedures, clocking scheme synthesis is constrained by both the data path design and the control design. Clocking scheme synthesis is carried out as one of the last tasks of control synthesis. For a given data path design and a control hardware design, the task of clocking scheme synthesis is as follows:

- choose an optimal clock period
- determine an optimal number and length of the clock phases
- assign clocked control signals to clock phases and route to the data path

However, most of the important parameters determining the execution speed of digital systems are fixed during the data path and control design. Thus, optimality of the design can be guaranteed only if clocking scheme synthesis is done concurrently with both the data path design and other parts of the control design. For example, cheaper data path designs often require more elaborate clocking schemes and therefore a final solution to the data paths cannot be chosen until the clocking cost is examined (and indeed, until the entire cost including control is examined). In this research, we shift the occurrence of clocking scheme synthesis to somewhat earlier phases of the design procedure in order to synthesize near-optimal digital systems. We define the task and goals of clocking scheme synthesis as follows:

INPUT

(i) Partial data path and control design with chosen functional units and minimum required storage elements[2],

(ii) Types of micro cycles (e.g., microinstruction formats or Node-Module-Range bindings [26], which specify the direction and propagation time of data values through functional elements during micro cycles) and

(iii) Expected sequences of micro cycles to be executed.

CONSTRAINTS

(i) Minimum execution speed of the micro engine,

(ii) Maximum number (or total bit width) of storage elements and

[2] For any data path, the minimum number of storage elements is determined by the maximum number of live values [2] at any time. In most cases of computer CPU designs, the registers (e.g., ACC, MAR, and I/O buffer) and the main memory which the machine language programmer can directly access are the minimum set of storage elements. For control hardware, it can be either the μ-PC or the microinstruction register.

(iii) Maximum number of clock phases.

OUTPUT

(i) Assignment, insertion or deletion and interconnection of storage elements necessary to obtain a certain execution speed (or speed to cost ratio),

(ii) Minimum and optimal (not necessarily distinct) clock periods to maximize the execution speed,

(iii) The optimal number and length of clock phases and

(iv) Clock signal routing to stage latches.

We consider three cases of partial data path designs. The first case includes designs which have not been completed yet and need more storage elements to be allocated and connected in order to satisfy the machine behavior. The second case includes designs which have already been completed and only the connections from and to the storage elements are partly or completely undone for the purpose of reconfiguration of the interconnections. The third type includes completed designs as they are. For completed designs, we may need to add or delete storage modules in order to increase performance at minimum cost or to decrease cost at minimum sacrifice of speed. In any case, the objective of the clocking scheme synthesis task is, while satisfying all the design constraints and desired goals, to maximize the execution speed of the system by optimally configuring, adding and/or deleting storage modules optimally and consequently determining an optimal clock period and number of clock phases.

There is no absolute ordering in carrying out these tasks. The result of each task may affect the results of one or more of the others. For example, choice of an optimal clock period and determination of the optimal number of clock phases depends on the result of optimal stage partitioning. Also, the maximum allowed number of clock phases (due to clock generator cost and/or clock signal routing complexity) will affect both the choice of an optimal clock period and optimal stage partitioning. For this reason, *a unified solution methodology is strongly desired in order to examine the attributes of all the design decisions in parallel.*

2.3 Definition of Speed of Digital Systems

As mentioned in the first chapter, the system tasks (processes or programs) consist of sequences of macro tasks, each of which consists of one or more fixed sequences of micro cycles. Therefore, execution times of system tasks can be determined by the execution sequences of their micro cycles and the execution time of those sequences. In this sense, the execution speed of the micro engine can represent the execution speed of the system. There are several ways of defining the execution speed of a micro engine of a digital system for performance evaluation:

1. maximum possible execution speed

2. execution speed for certain micro cycles

3. execution speed for a (weighted) average mixture of micro cycles

The first two parameters are not realistic since they do not encounter an actual mixture of the micro cycles. The third parameter, which is an overall performance measure of the system, can be computed by assuming the average mixture of micro cycles over a long enough time period. The total estimated execution time divided by the number of micro cycles executed will be the average expected execution time of a micro cycle. Appropriate weighting functions may be used to indicate the average occurrence and/or importance of each micro cycle.

2.4 Previous Work

Since the task of high-level (functional level) digital design automation was launched more than a decade ago [10], there has been a vast amount of effort in automating various components of the digital design such as design style selection [40, 29], data path design [22, 39, 18, 20], microprogram synthesis [1, 35, 33], and integrated design automation systems [34, 15, 41, 16]. However, there has not been much work in the area of clocking scheme synthesis except that done for pipeline control, which can be considered to be a special case of general clocking scheme synthesis. As we have discussed before, the clocking scheme synthesis task is important in optimizing the speed of digital systems and must be carried out together with the data path and control design. However, it has been either buried under architectural design [34, 15, 41, 16], or assumed *a priori*, as a part of the control design [5, 24, 33, 3] or data path design [18]. In some cases, clocking scheme synthesis alone is carried out for already completed designs [12, 30, 31]. Among them, we will briefly discuss several which are most closely related to this research.

2.4.1 Related Work in Clocking Scheme Synthesis

Recently, as one of the projects closest to our research, Leiserson [30, 31] proposed a technique which determines a relocation of the registers of a given data path in order to minimize the clock period. The data path is modeled as a directed graph where the vertices represent functional modules and the directed edges represent interconnections. The locations and the number of registers are indicated by the edge-weight. The basic assumption of this technique is that all the hardware modules are performing useful operations at any time and thus all the registers are clocked at the same time by a single clock source (e.g., a systolic array). The technique moves the registers of the original design along the direction of the data flow. If the movement is to be made onto any forked[3] edges, registers are copied to all of the fork-edges in order not to change the original data flow. The optimal relocation of registers is determined as the design in which the longest propagation delay between any two registers is minimized, which minimizes the clock period. The major contribution of this work is that it suggests several formal tools for timing analysis of digital circuits, which are the graph model of the digital circuit and the problem formulation using linear and/or mixed-integer linear programming. There are several shortcomings of this technique to be used in general cases of digital systems speed-up. They are:

- The technique assumes fixed clocking for all the registers at the same time
- The technique assumes fixed data flow for all time
- Control hardware timing is not considered
- Register propagation delays are ignored

[3] In this model, all the forks are AND forks since every hardware module is performing useful operations and thus all the interconnections are carrying useful data values.

Boulaye [5] discusses speeding-up pipelined micro engines by minimizing the time overhead caused by the conditional branches, which is accomplished by clocking the condition latch as early as possible. This approach can also be considered as relocation of registers to reduce the critical path or the critical stage of a pipeline. However, if the propagation delays of both the data path stages and the stages of the micro engine are not considered together, optimal relocation cannot be determined. Also, in any instruction pipeline, any branch causes resynchronization overhead, which also involves the termination and re-initiation of the data path stages.

Andrews [3] considers using a multiphase clocking as one way of reducing the number of microinstruction fetches from a slow microprogram storage. By using a multiphase clock, microinstructions can be horizontally coded and executed serially in several clock phases without having an expensive data path. As he mentions, the performance of this technique depends on the coding efficiency of the horizontal microprogram. If the microinstructions are sparsely coded, then the resource utilization efficiency will be low. Also, after the completion of a microinstruction which has only the microoperations with short execution times, there will be idle time until the fetch cycle of the next microinstruction. This is true for any microprogram (vertical or horizontal) if execution overlap is not used. However, if execution overlap is used, this is not always true since in case of overlapped execution, the execution speed of the micro engine depends much more on the longest microoperation execution time rather than the total execution time span of the microinstructions. Moreover, if execution overlap is properly used, vertically coded microinstructions can be executed as fast as horizontally coded ones. This saves a significant amount of design time and avoids the complexity of horizontal microprogram compaction, which is known to be intractable [21, 37].

As another approach, Berg [4] characterized the timing behavior of a given control and clocking scheme in order to provide a guideline for the synthesis of a fast and correct microprogram. The timing behavior of a controller at the macro level is modeled as a finite state machine. The model allows multiphase execution of micro-instructions. However, the model is focused on modeling the timing of the interactions between main system blocks such as the CPU, main memory, and I/O controller.

Davidson et. al. [12] suggested a formal technique to analyze and determine the reservation table for the sequencing of pipelined data paths with loops. The state of the pipeline is modeled as a finite state machine where each state represents the utilization of the pipeline stages. This reservation table technique is extensively used in general pipeline designs. We believe that this technique can be easily extended and used to analyze a multiphase clocking scheme for general multistage digital systems.

General insight into control architecture for pipelined systems is discussed in depth by Kogge [27] and Ramamoorthy [36]. Basic clock timing requirements for pipelined data paths are analyzed by Cotton [9]. A technique for performance measurement of static pipelines is proposed by Lang [28], which uses a table-driven simulation model.

2.4.2 Other Related Work

The basic concept of execution overlap under a centralized control originates from the look-ahead [25] techniques at the macro (machine instruction) level. Examples of machines which implement macro level execution overlap include the CDC6600, and the IBM 360/91, 195 and 370/165. They assume that instruction fetch, decode, and execute cycles, each consisting of a sequence of micro cycles, take almost equal time, which is the basic assumption of general pipelines. Possible execution overlap is predicted by checking the type and execution status of the current macro task being executed. Typical checking mechanisms use condition flags and/or counters which represent the state of associated resources. Naturally, look-ahead techniques assume flexible execution control mechanisms implemented by the micro level sequencing primitives [24]. However, at the micro level, implementing look-ahead is very costly and difficult since the look-ahead mechanism must be much faster than the micro cycle time in order to achieve execution overlap, which, in most cases, requires hardware level primitives.

Nagle [33] provides a good insight into the general problems of control synthesis at the micro level, although all the problems are not analyzed in depth. The major contribution of this work is microprogram synthesis under given constraints, such as the capacity of the microprogram storage, speed requirements, and the number of control signals that can be activated at the same time. The control flow optimization and control distribution techniques proposed can be used to reduce the number of branches, to shorten conditional branching time and to reduce the number of micro cycles, which is essential to increase the performance of the micro engine.

Cook [8] considered multiphase clocking of PLA's in order to reduce the power consumption of the PLA. A precharge scheme using a multiphase clock is used to compensate the turn-on/off time delay. Although he does not mention it, his PLA partitioning technique may be very useful for multistaging a control store using PLA's. For example, we can partition the AND-plane and OR-plane of a large PLA by inserting a latch to latch the product terms. Then, since it is a sequential machine design, we can overlap the propagation delays of the AND-plane and the OR-plane.

2.5 Motivation

As we have briefly discussed, various techniques for speeding-up digital systems at the macro level (or even higher level such as processes) exist. However, the ultimate performance of these macro-level speed-up techniques depends very much on a good sequencing control scheme at the micro level, since the macro level tasks including those used to implement the speed-up techniques are eventually implemented by certain sequences of micro cycles.

As we have discussed in the previous section, existing techniques for high-speed sequencing of micro cycles are not general in the sense that their models are very restrictive and/or not precise enough to model actual digital circuits and sequencing of the micro cycles. Also, each speed-up technique has been developed rather independently and does not consider various effects of the result of applying the technique to the results of other speed-up techniques or to the results of other optimization tasks such as cost reduction. For these reasons, development of a more general model for clocking scheme synthesis is strongly desired and is proposed here. The model and synthesis techniques must be able to consider precise sequencing characteristics of the micro engines and timing of the hardware as well as the cost of speed-up.

3 THE PROBLEM FORMULATION

This chapter summarizes:

- Problem definition and modeling.
- Extraction of the parameters affecting the performance of a multistage micro level execution overlapping scheme (micro level).

The major problem components of clocking scheme synthesis are based on the discussions in Sections 1.4, 2.2 and 2.5,. Sections 3.1 and 3.2 will discuss modeling the sequencing and timing behavior of micro cycles. The discussion is based on those in Sections 1.2 and 1.3. Sections 3.3 and 3.4 will discuss the sequencing behavior of the overlapped micro cycles as well as the clocking and control requirements for micro cycle sequencing.

3.1 Specifying the Functioning Times of Digital Circuits

The timing behavior of a hardware module can be considered as a function of the timing of external excitations and the functions they specify. Thus, in order to analyze the timing behavior of hardware modules, we must consider the functional behavior of hardware modules.

3.1.1 The Circuit Graph

A physical hardware module which can perform multiple functions can be considered to be multiple **logical modules**, which are defined as follows:

> **Definition 1:** A **logical module** is a set of physical hardware modules which can perform a certain complete function (either functional or archival) without any resource contention with other functions at any time.

Logical modules may be either physically separated or share common physical hardware modules. An adder chip or a pass gate can be considered to be a logical module. *A register chip can be considered as two logical modules, read and write modules, if it can be read and written simultaneously without any conflict in using its control and data lines.* A bidirectional bus must be considered as a separate logical module since it cannot be considered as a part of any one module connected to it. A set of interconnection lines which are always used together to transfer a certain value can also be considered a logical module. In this sense, a logical module can be considered a unit hardware resource whose timing and functional behavior can be unambiguously defined. Also, resource contention between executions of the micro cycles can also be represented in terms of the logical modules. From now on, the term *module* will always imply logical module, unless otherwise specified.

Using the concept of *logical modules*, we can model a digital circuit as a weighted, directed graph (circuit graph), where the vertices of the graph represent modules and the directed edges represent all the possible pathways for both the control and data values between the modules in the circuit. The purpose of the circuit graph is to **connect the control and data path hardware together**.

Definition 2: A **circuit graph**, G = (V,E), is a directed graph where the set of vertices, V, represents modules, and the set of directed edges, E, represents the pathways for data and control values between modules. A directed edge, e(i,j), belongs to E if any output port of module i is connected to any input port of module j. The vertices are weighted with the propagation delays of the modules (δ) and the edges are weighted with the bitwidths of the interconnection lines (σ).

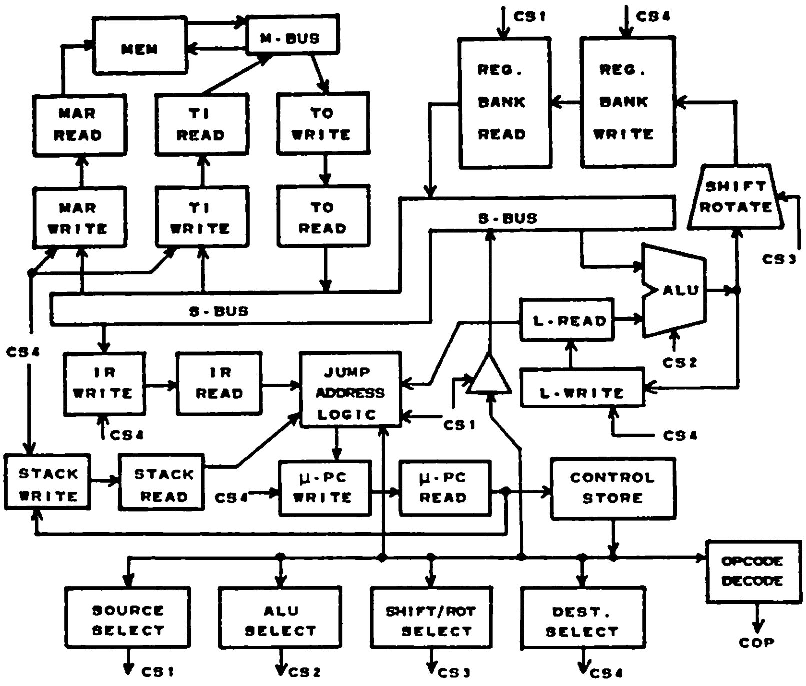


Figure 3: A Circuit Graph of a Microprogrammed Computer CPU

By weighting the vertices with the propagation delays of the modules, the propagation delays of data values and control signals along any path in the circuit can be computed. The vertices for interconnection lines with non-negligible propagation delays must be added. The bitwidths of the data and control flow can be computed easily using the edge weights. In the case of a partially designed system, all the necessary interconnections for the flow of control and data values specified in the data flow graph and timing graph [26] must be added. Figure 3 shows an example of a circuit graph.

3.1.2 Specifying the Propagation Delays of Modules and Circuits

The micro or minor cycle time can be computed by summing up the propagation delays of the control and data flow through the modules along the execution paths. We consider two types of propagation delays which are defined as follows:

> **Definition 3:** The **port propagation delay**(PPD) of a pair of input and output ports of a module is the maximum time taken for any change of input values to possibly change any output value.

For an adder, carry-in and operand ports are input ports, and sum and carry-out ports are output ports. For a read module of a storage element, the read control input is also an input port. For a bus, each set of lines outputting data on the bus is an input port and each set of lines receiving data from the bus is an output port.

> **Definition 4:** The **module propagation delay**(MPD) of a module is the maximum port propagation delay for all possible pairs of input and output ports of the module (maximum port propagation delay).

In order to compute precise execution times of the micro cycles and the critical paths of the circuits, the PPDs are preferred for the vertex-weighting of the circuit graphs. However, if the PPDs are used, more complex variations of the original circuit graph or complicated graph theoretic algorithms are required. In the models that we are going to discuss in the following sections, we will use MPDs.

3.2 Modeling Sequencing Behavior of Micro Cycles

In order to analyze the sequencing and timing behavior of the micro cycles efficiently, we introduce two directed graphs, the MEG (Micro cycle Execution Graph) and the COM (Chain Of Minor cycles). They are based on the circuit graph and model the pattern of resource usage and timing of the micro cycles.

3.2.1 The MEG - the Micro-cycle Execution Graph

In order to model the pattern of resource usage and the execution time of all the types of micro cycles, we construct edge-weighted, vertex-weighted digraphs, the MEG's (Micro-cycle Execution Graphs). The MEG's are subgraphs of the circuit graph.

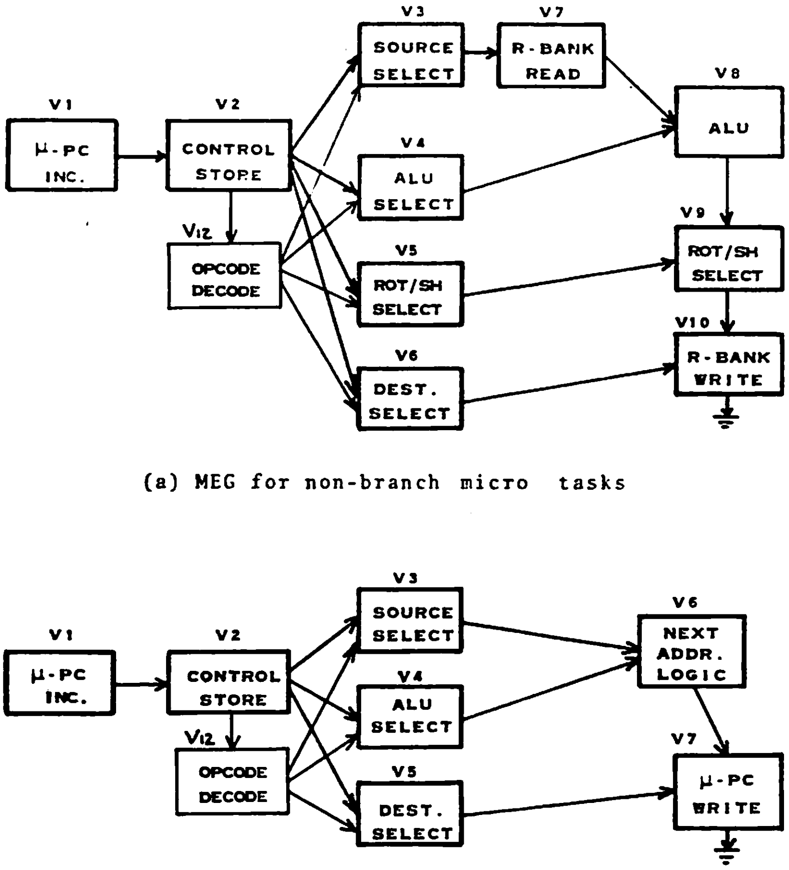
> **Definition 5:** For a given circuit graph, $\mathcal{G}(\mathcal{V},\mathcal{E})$, the **MEG** for a set of one or more micro cycles, G(V,E), is a **rooted** subgraph of $\mathcal{G}$ where the set of vertices, V $\subseteq$ $\mathcal{V}$, and the set of directed edges, E $\subseteq$ $\mathcal{E}$, represent only the modules and interconnections **activated by** and **necessary to** the execution of the micro cycles in the set.[4] The vertices and edges are weighted in the same way as in the circuit graphs. In addition to the bitwidth-weight, the edges are also weighted with the number of visits to the edges during a micro cycle (λ).

The MEG's are **rooted** by a **common root**. In general, *for any synchronous sequential circuit or finite state machine [17], there must be memory and/or delay elements in order to prevent state-change races and/or to control the time intervals between state changes.* Among the memory or delay elements, we can choose a subset of them as the starting point of every cycle. For a microprogrammed micro engine, it can be either the μ-PC or the microinstruction register. In case of a hardwired sequencer, it can be either the state counter or feedback state-memory.

Figure 4 shows two MEG's derived from the circuit graph of Figure 3, using the MPD. Both are rooted at the μ-PC. Figure 4-(a) is the MEG for non-branch type microinstructions. The execution sequence of this type of microinstruction is:

1. Increment PC (v1).

[4] Read modules whose outputs are always enabled and write modules which are not written during the execution of the micro tasks in the set are not included, although the values contained in them may be needed by the micro cycles.



(a) MEG for non-branch micro tasks

(b) MEG for branch micro tasks

Figure 4: Examples of the MEG's of the Circuit Graph of Fig. 3

2. Fetch the micro cycle pointed to by the PC (v2).

3. Decode the control fields of the fetched micro cycles: opcode (v12), operand register address (v3), ALU function code (v4), rotate/shift function code (v5), and result register address (v6).

4. Fetch operand from selected register (v7).

5. Perform selected ALU operation (v8).

6. Perform selected rotate/shift operation (v9).

7. Store the result in the selected register (v10).

An example of the execution sequence of a branch micro cycle corresponding to the MEG of Figure 4-(b) is as follows:

1. Increment PC (v1).

2. Fetch the branch micro cycle pointed to by the PC (v2).

3. Decode the control fields of the fetched micro cycle: opcode (v12), condition select (v3), branch address modification (v4), and branch type (v5).

4. Select a test condition and select the full jump address (v6).

5. Load the PC with jump address. At the same time, if it is a CALL to a subroutine, save the current PC contents in the stack (v7).

Assuming that there are no nested cycles, the MEG can show the sequence of activation of the modules during the micro cycles by means of the visit-weight of the edges, λ.[5] Also by weighting the vertices with propagation delays, the MEG can also represent critical execution paths and execution times of micro cycles. The MEG's can be used to determine the locations for the stage latchs which either have to be added or already exist. The locations and connections of the stage latches determine the interstage propagation delays between the stage latches. Also, by weighting the edges with the bitwidths of the corresponding interconnection lines (σ), the bitwidths of the added stage latches can be computed.

3.2.2 The COM - the Chain of Minor Cycles

Once the locations and connections for the stage latches are determined, the interstage propagation delays are also determined and thus the minimum requirements of the clocking and timing for the micro cycles are determined. This basic timing requirements are modeled by one or more line graphs - more precisely, chains (COM: Chain Of Minor cycles) - which show the minimum required execution time of minor cycles as well as the minimum required clock period.

[5] In case when there are nested cycles which are visited more than twice, then a vector of vertex indices which represent the sequence of visits to the modules may be associated with each MEG.

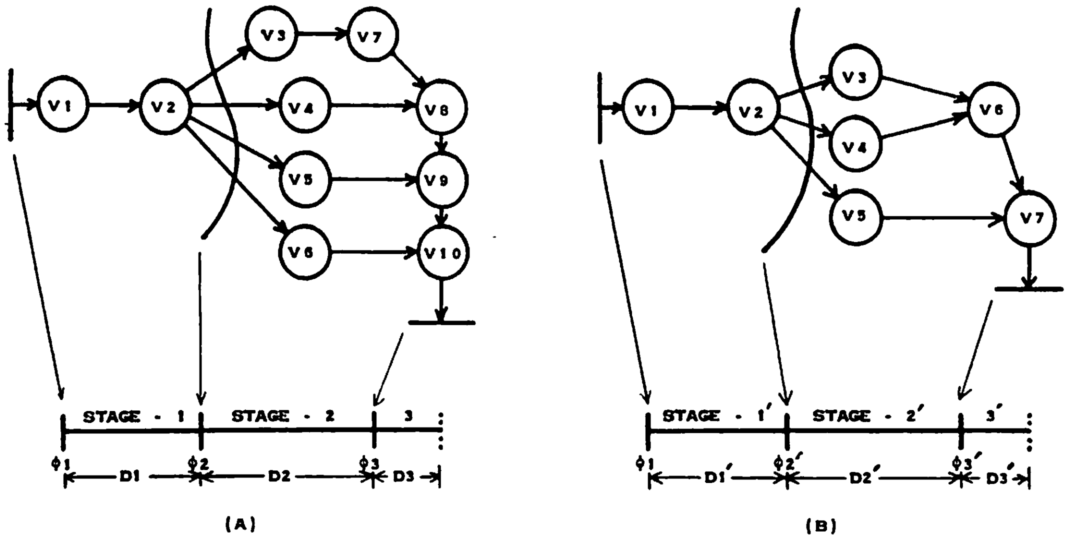


Figure 5: COM's Derived from the Results of Stage Partitioning

Figure 5 shows examples of the COM's derived from the results of stage partitioning of the MEG's of Figure 4. The locations of the stage latches are indicated by the edge-cut lines in the MEG's. In case (A), the stage latches are the μ-PC, an added latch next to the control store and the register bank of Figure 3. Let ϕ_i be the clock phase used to clock the i-th stage latch, L_i, and Di be the phase difference between ϕ_i and ϕ_{i+1}. Also let $\delta(i)$ be the weight of vertex v_i, and $\delta(3) + \delta(7) \geq \delta(j)$, for j = 4, 5 and 6 (COM (A)). Then the timing requirements are specified as:

$$D1 \geq \delta(1) + \delta(2) + \delta(12) + D_{SS}(L_2)$$

$$D2 \geq D_{SP}(L_2) + \delta(3) + \delta(7) + \delta(8) + \delta(9) + \delta(10)$$

where D_{SS} and D_{SP} are the set-up and storage propagation delays of storage elements defined by Hafer [19]. Note that Dsp(L1) and Dss(L3) is already included in the MEG as v_1 and v_{10} (v_7 in COM (B)), respectively.

At the end of the chain, D3 must be added in order to consider the completion time of all the effects of an execution of a micro cycle, although it is not explicitly specified in the MEG. It is necessary to analyze the effect (data dependency, resource contention, etc.) of a micro cycle on its successor. For example, if the next micro cycle reads the result of the current micro cycle which will be stored in the third stage latch (e.g., v_{10} of MEG (A) or v_7 of MEG (B)), then the next micro cycle can only read the correct value after the buffer has been clocked and the stored values propagated to its outputs.

The COM's can be used to determine a major clock period and the number and length (phase lag) of the clock phases which clock the stage latches and execute the micro cycles. Resource conflicts between the minor cycles can also be represented by attaching the module names used by the minor cycles to corresponding edges of the chain.

3.3 Sequencing Behavior of Overlapped Micro Cycles

In this section, we analyze the sequencing behavior of the overlapped micro cycles according to their timing, pattern of resource usage, and interactions (data dependencies and resource conflicts) between them.

Maximum Initiation Rate of the Micro Cycles

The maximum initiation rate of micro cycles is defined as the maximum possible number of initiations of micro cycles during some unit time period when there are neither branch micro cycles nor resource/data conflicts between micro cycles.

Figure 6 shows examples of micro cycle sequencing with different sequences of clock phases. We assume a static clocking sequence. (a) does not use any overlapping, hence there is only one stage-latch and one stage. The micro cycle times of (b) through (d) are longer than that of (a) due to the propagation delays of the stage latches. In any case, the maximum initiation rate of the micro cycles is the same as the clock rate, t_{cy}. The clock period must be longer than the longest interstage propagation delay in order to ensure that no two micro cycles occupy the same stage. Figure 6-(b) uses the shortest clock period possible, which is 2.

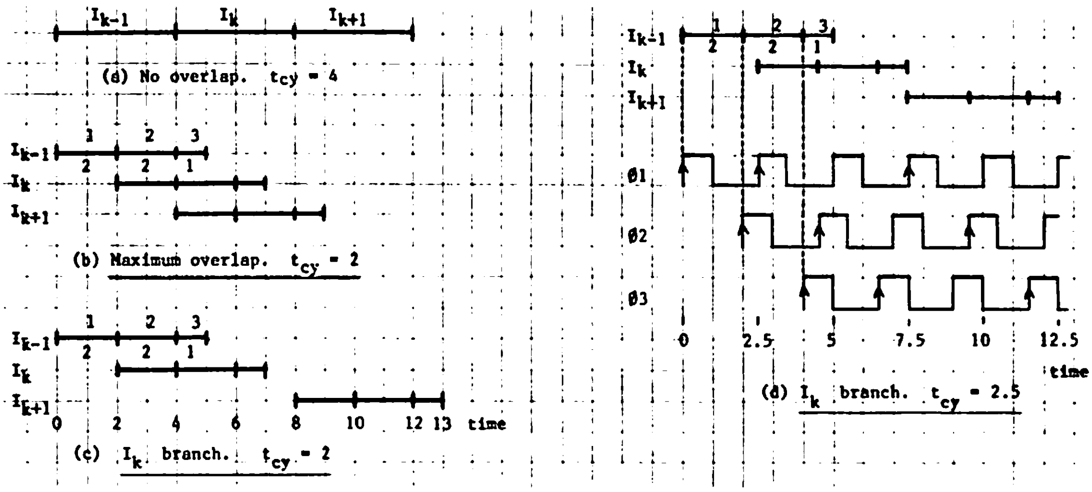


Figure 6: Examples of Micro Cycle Sequencing and Clocking

Micro Branching

A branch micro cycle delays the fetch of the next micro cycle until the earliest fetch clock cycle after the completion of the branching. Due to this **resynchronization overhead**, the shortest clock period does not guarantee the fastest overall initiation rate. *Even increasing the clock period may result a faster execution if it can reduce the resynchronization overhead.* As shown in Figure 6, the total execution time of (d) is shorter than that of (c) in executing I_{k-1} through I_{k+1}. Thus, the overall initiation rate will also depend on the frequency of the branch micro cycles. Therefore, determination of the optimal clock period should consider the resynchronization overhead due to branches.

Resource and/or Data Contention Between Micro Cycles

Resource and data contentions between micro cycles are other causes of resynchronization overhead. If there is any data or resource contention between any two micro cycles, fetching the later micro cycle must be delayed until its initiation does not cause any contention with its predecessor. The delay time is dependent on both the clock period and the pattern of data and/or resource contention between the micro cycles. These cases are shown in Figure 7. In case (a), if I_{k+1} is initiated as the dotted cycle (I'_{k+1}), then there will be a resource conflict or data dependency violation between the minor cycles using resources R_{14} and R_{23}. Case (b) does not have any resynchronization overhead. This shows that the resynchronization overhead can also be reduced by choosing a proper clock period.

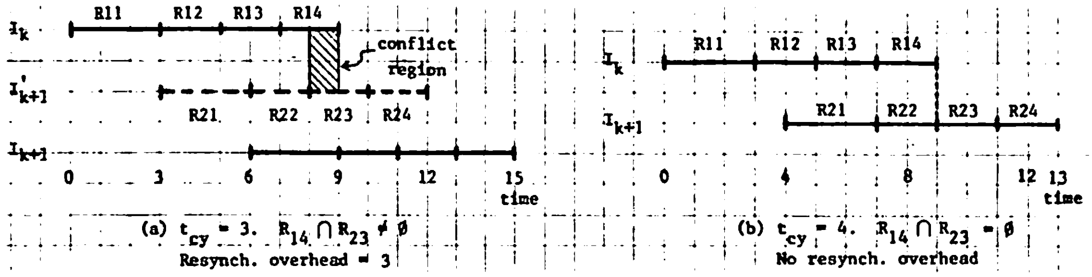


Figure 7: Resynchronization Overhead due to Data/resource Contention

Resynchronization Overhead vs. the Number of Clock Phases

The time required for resynchronization may depends on the length of the clock phases even if the same clock period is used. Increasing the intervals between the clock phases may reduce the number of distinct clock phases without increasing the clock period. Figure 8(b) shows a clocking sequence which is exactly the same as the COM, which requires three distinct clock phases. Figure 8(c) has only two distinct clock phases with the same initiation rate as (b) regardless of the resynchronization overhead. However, in (d), the branching overhead is longer than that of (b) or (c) by one clock period (4 time units) and thus the overall initiation rate is lower. Although there might be some difficulties in gating and routing a single phase clock to multiple stage latches selectively, reducing the number of clock phases may reduce the physical routing problem significantly. However, if the longest clock phase interval is increased, it will always result in a slower maximum initiation rate since the minimum possible clock period must be longer than the longest interstage propagation delay.

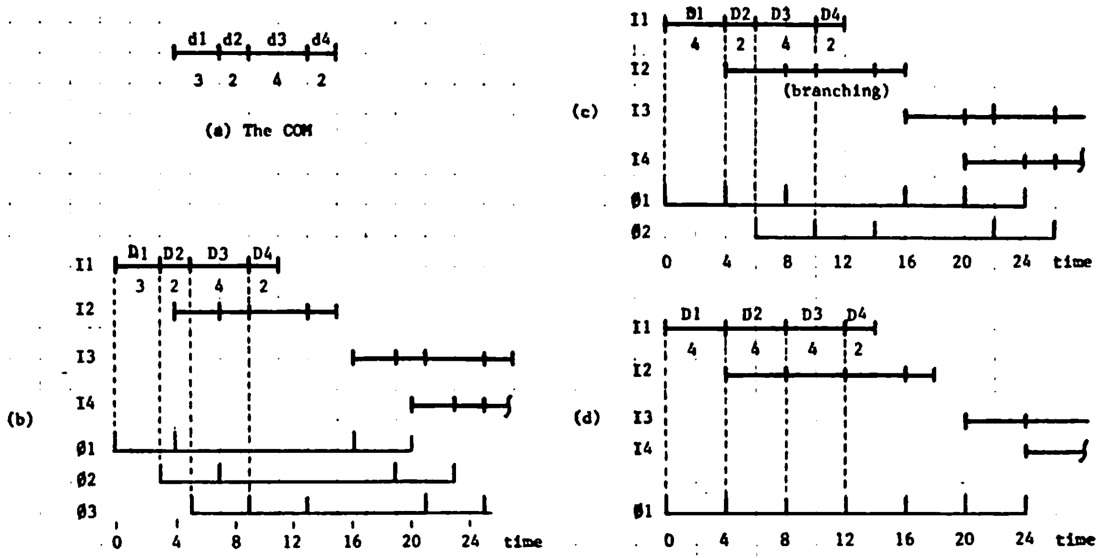


Figure 8: The Number of Clock Phases vs. Resynchronization Overhead

3.4 Clocking Requirements of Overlapped Micro Cycles

In the worst case, we may have as many distinct COM's as the number of MEG's, which also requires as many distinct clocking sequences for optimal design. Especially in the cases where execution overlap is extensively used, all the different clocking sequences may have to be overlapped and thus as many separate clock generators are required. A very complex initiation and termination control mechanism for the clock sequences is required in order to prevent conflicts in the usage of both the hardware resources and the data values between micro cycles using different clocking sequences. In actual designs, this is not realistic and seldom happens because of the cost and control complexity of the clock generator(s) and clock signal gating and routing. In actual designs, a single clocking sequence (fixed number and sequence of clock phases) is usually used with proper gating and routing of the clock phases to the stage latches. In addition, wait cycles to extend certain clock phase(s) are often used for micro cycles with exceptionally long minor cycles (e.g., I/O and main memory micro cycles). In this report, we will focus on synthesizing clocking schemes with a single clocking sequence while allowing all the variations as discussed. Two examples of such clocking sequences for the COMs of Figure 5 are in Figure 9.

```
φ1            φ2,φ2'     φ3' φ3      φ1            φ2,φ2'      φ3,φ3'
|-------------|----------|--|----|   |-------------|------------|----|

|<---- t1 ---->|<-- t2 -->|t3| t4 |  |<---- t1 ---->|<--- t2 --->| t3 |

   t1 = max{D1, D1'}                         t1 = max{D1, D1'}
   t2 = D2'   t3 = D2 - D2'                  t2 = max{D2, D2'}
   t4 = D3 (t3 + t4 ≥ D3')                   t3 = max{D3, D3'}

  (a) Dynamic clocking (4-phase)        (b) Static clocking (3-phase)
```

Figure 9: Examples of Single-chain Clocking Sequences

The dynamic clocking sequence, (a), is determined by the overlap of all the COM's. Clock phases are gated and routed selectively according to the type of micro cycle. In the static sequencing case, all the micro cycles are executed by a single common clock sequence, a scheme which has been the most widely used in general purpose computer CPU's and simple synchronous digital controllers. Each type of clock sequencing has its own advantages and disadvantages. Dynamic clocking sequences require more clock phases and thus more expensive clock generators. However, by making the length of each clock phase as short as possible, they may reduce the resynchronization overhead. In other words, the overlapped time between minor cycles with resource contention can be minimized. However, in any case, the longest interstage propagation delay is not changed and hence the maximum initiation rate of the micro cycles will be the same.

4 LOOP-FREE CLOCKING SCHEME SYNTHESIS RESULTS

This chapter contains discussions and results of static clocking scheme synthesis for the sequencing of *loop-free* micro cycles. By "loop free" we mean that each micro cycle uses the same (logical) module no more than once and thus there is *no cycle in any MEG.*

We analyze optimal stage assignment and choice of optimal clock period. Also, determination of an optimal number of clock phases and their length is also analyzed. These analyses are carried out under two different goals: (i) to find an absolute optimal solution and (ii) to find an optimal solution with respect to certain constraints. Simple and efficient algorithms to determine optimal positions of the stage latches and optimal number of clock phases are developed. We believe that we can easily extend these results to analyze data path cycle with loops and, furthermore, to analyze more general cases of system timing styles. In each section, we summarize the results only. Detailed proofs of lemmas and theorems are attached as an appendix.

4.1 Definition of Variables

[6]

δ_i The module propagation delay of module i.

L_{ij} The j-th inter-stage latch of the i-th COM (or MEG).

In the single static clocking case, L_j is the set of L_{ij}'s for all i

C_{ij} The control/data path stage in between L_{ij} and $L_{i,j+1}$

d_j $\mathbf{\max_i}\{D_{SP}(L_{ij}) + D_{FP}(C_{ij}) + D_{SS}(L_{i,j+1})\}$. The maximum interstage propagation delay of the j-th stages.

d_{max} $\max\{d_1, d_2, \dots, d_m\}$ where m is the number of stages[7].

[6] The reader is urged to skip this section and refer back to it while reading this chapter.

[7] m is the number of stages of the MEG with the largest number of stages. We call such a system an **m-stage system**.

d_S $d_S = d_1 + d_2 + \dots + d_m$

I_i A micro cycle as an instance of an execution of a micro task (e.g., an execution of a microinstruction)

$S_n^{n_b}$, n_b A sequence of micro cycles of length n, $(I_1.I_2. \dots .I_{n-1}.I_n)$, where there are n_b branches in $(I_1.I_2. \dots .I_{n-1})$

n_d $= (n - n_b - 1)$. The number of non-branch micro cycles in $(I_1.I_2. \dots .I_{n-1})$. Note that the last micro cycle, I_n, is excluded from n_d even if it is a non-branch micro cycle.

ϕ_j Clock phase j. Used to latch L_{ij} for all i.

$\phi_j(k)$ Time when clock phase j latches L_{ij} to execute a micro cycle I_k.

D_i The actual interstage time of the i-th stage determined as:

$$D_i = \phi_{i+1}(j) - \phi_i(j) \geq d_i,\ 1 \leq i < m, \text{ and } D_m \geq d_m$$

Ψ_D A chosen multiphase clocking scheme for an m-stage system

$$\Psi_D = (D_1, D_2, \dots, D_m)$$

D_{max} $\max\{D_1, D_2, \dots, D_m\}$

D_S $D_1 + D_2 + \dots + D_m$

E_i The actual execution time span of type i micro cycle over D_S.

t_{cy} Clock period. $t_{cy} \geq D_{max}$

$\mathcal{T}_{t_{cy}}^{\Psi_D}(S_n^{n_b})$ Execution time for an execution sequence, $S_n^{n_b}$, with Ψ_D and t_{cy} (from $\phi_1(1)$ to $\phi_1(n+1)$)

$\mathcal{T}_t(x)$ $\mathcal{T}$ as a function of t_{cy} (x) with fixed Ψ and S

$\mathcal{T}_S(y)$ $\mathcal{T}$ as a function of S (y) with fixed t_{cy} and Ψ

4.2 Execution Speed Analysis

Determination of the Minimum Clock Period

The minimum clock period for a multiple stage system is determined by the interstage propagation delays. In order to ensure correct sequencing of micro cycles, the minimum clock period should be longer than the longest interstage propagation delay [9, 36, 7].

Lemma 1 : For an m-stage system, $\mathcal{M}$, with the stage times $(D_1, D_2, \dots , D_m)$, $\min(t_{cy}) = D_{max}$. (Refer to Figure 2, 7, 8, or 9) □

The proof is found in the Appendix.

Execution Time of an Execution Sequence

For an execution sequence of micro cycles, the execution time is defined as the *time from the fetch clock phase for the first micro cycle in the sequence to the earliest fetch clock phase after the completion of the last micro cycle in the sequence.* For an execution sequence of n micro cycles, if there are no branch micro cycles in it and there is no resynchronization overhead, then the execution time, $\mathcal{T}$, is computed as the sum of the following:

1. $(n\text{-}1)\cdot t_{cy}$ for the first (n-1) micro cycles which are initiated every t_{cy} period.
2. $\left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot t_{cy}$, which is the execution time of the last micro cycle

$$\mathcal{T} = (n - 1 + \left\lceil \frac{D_S}{t_{cy}} \right\rceil)\cdot t_{cy} \qquad (4\text{-}1)$$

Slow-Down Due To Branching

Any branch cycle delays the fetch of the next micro cycle until it completes branching. The difference between the fetch time of next micro cycle *after a branch cycle* and *after a non-branch cycle* is defined as **branching overhead**.

Lemma 2 : (Refer Figure 7) Let $\mathcal{M}$ be an m-stage machine with a multiphase clocking scheme $\Psi_D = (D_1, D_2, \dots , D_m)$ and clock period t_{cy}. For any two execution sequences,

S_1 and S_2, let S_2 be the same as S_1 with some non-branch cycle, I_j, $1 \leq j < n$, replaced with a branch cycle, I_j'. Then

$$\mathcal{T}_S(S_2) - \mathcal{T}_S(S_1) = t_{cy} \cdot \{ \left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1 \}$$

□

```
       |<---------- Ds --------->|
I1  :  |--------|---------|------|

I2' :            |--------|---------|------|
                  (when I1 is not a branch)
I2  :                                |--------|---------|------|
                                        (when I1 is a branch)
       |<--tcy-->|<----- 2 tcy ----->|        |         |
```

If there are n_b branch micro cycles, then the total branch overhead is

$$n_b \cdot \{ \left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1 \} \cdot t_{cy} \qquad (4\text{-}2)$$

Thus, the difference in execution times for two sequences of micro cycles is a function of the cycle time and the total interstage times. In actual systems, there may be several types of branch micro cycles with different execution times (no more than 4 types in most cases of micro-sequencers). Typical types of branch micro cycles which may have different execution times are conditional branch, unconditional branch, decode branch and "sense and skip". In such a case, we can compute the branching overhead for each type of branch micro cycle. For example, let E_i be the execution time span of type-i branch micro cycles over the sequencing chain. Then the branching overhead of type-i branch micro cycles is $\{ \left\lceil \frac{E_i}{t_{cy}} \right\rceil - 1 \} \cdot t_{cy}$.

Execution Time of an Execution Sequence with Branches

The execution time of an execution sequence of n micro cycles with n_b branch micro cycles can be calculated as the sum of the execution time of n non-branch executions and the branching overhead for n_b branches.

Theorem 3 : On an m-stage system $\mathcal{M}$ with a multiphase clocking scheme $\Psi = (D_1, D_2, \ldots, D_m)$ and clock period t_{cy}, if there is no resynchronization overhead, an execution sequence $S_n^{n_b}$ is executed in:

$$\mathcal{T} = \{n_d + \left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot (n_b + 1)\ \} \cdot t_{cy} \qquad \square$$

Theorem 3 shows the relationship between the execution speed and the number of branches, the clock period and the length of the clock phases. The proof of this theorem is found in the Appendix.

Modification for Micro Tasks with Different Execution Times

As mentioned before, the system may have several types of branch micro cycles with different execution time spans. Suppose that there are j different types of branch micro cycles. Let n_{bi}, $1 \leq i \leq j$, be the number of type i branch micro cycles with execution time span E_i out of n_b. Then, we can replace Equation (4-2) with

$$\sum_{i=1}^{j} n_{bi} \cdot \{ \left\lceil \frac{E_i}{t_{cy}} \right\rceil - 1\ \} \cdot t_{cy} \tag{4-3}$$

Also non-branch micro cycles may have different execution time spans. Assuming that we know the execution time span of each type of micro cycle and the execution sequence of micro cycles, we can also generalize Equation (4-1). In order to generalize Equation (4-1), we only need to consider cases where the execution of I_l, for some l, $l < n$, completes later than I_n. Since any branch micro cycle, I_i, $1 \leq i < n$, must complete execution before I_{i+1} starts, we can exclude branch micro cycles from this special case computation. Therefore we only need to consider such I_l's that there is no branch micro cycle in between I_{l-1} and I_n. Then we can replace Equation (4-1) with

$$(n-1) \cdot t_{cy} + \max_{l} \{ \left\lceil \frac{E_l}{t_{cy}} \right\rceil - (n-l)\ \} \cdot t_{cy} \tag{4-4}$$

where $1 \leq l \leq n$, and there is no branch micro cycle in between I_{l-1} and I_n.

Using Equations (4-3) and (4-4), we can fully generalize all the previous analyses to dynamic clocking analysis, where the micro cycles may have different execution time spans. However, as we can see by Equations (4-3) and (4-4), dynamic clocking analysis can simply be considered as a special case of static clocking analysis. Exactly the same approach and methods can be used for both analyses by simply adjusting several variables and/or constants as is done in Equations (4-3) and (4-4). For this reason, we will focus on static clocking analysis.

4.3 Maximum Execution Speed Analysis

Execution speed of a multi-stage system is a function of:

1. Interstage propagation delay d_i's.

2. Clock Period t_{cy}

3. Clocking scheme $\Psi = (D_i\text{'s})$

4. Given execution sequence, $S_n^{n_b}$.

In this section, we analyze the effects of these execution speed parameters.

Determination of an Optimal Clock Period

Lemma 4 : Let $\mathcal{M}$ be an m-stage digital system with $\Psi = (D_1, D_2, \ldots, D_m)$ fixed. On $\mathcal{M}$, for any execution sequence $S_n^{n_b}$,

$\mathcal{T}_t(\frac{D_S}{k}) \leq \mathcal{T}_t(\frac{D_S}{k'})$ for any integer k, $\left\lceil \frac{D_S}{D_{max}} \right\rceil > k \geq 1$, and real k', $0 < k' \leq k$. □

The proof of this lemma is found in the Appendix. With Lemma 4, we can see that the execution time function in terms of the clock period is not linear and reducing the clock period does not always reduce the execution time. However, we can determine an optimal clock period of an m-stage system with fixed a clocking scheme with the following lemma.

Lemma 5 : Let $\mathcal{M}$ be an m-stage digital system with $\Psi = (D_1, D_2, \ldots, D_m)$ fixed. On $\mathcal{M}$, for any execution sequence $S_n^{n_b}$,

$$\min(\mathcal{T}_t) = \min\{\mathcal{T}_t(D_{max}),\ \mathcal{T}_t(\tfrac{D_S}{p-1})\ \},\ \text{where } p = \left\lceil \frac{D_S}{D_{max}} \right\rceil$$

□

Using Lemma 5, we can determine an optimal clock period by evaluating the execution time of given execution sequence(s) only for two clock periods. In practice, execution sequences may be nondeterministic due to nondeterministic conditional branches (e.g., conditional branches on some external conditions and exception handling). However, if we can obtain statistics regarding the average length and composition of the execution sequence(s), then by using Lemma 5, we can easily estimate an optimal clock period.

Theorem 6 : Let $\mathcal{M}$ be an m-stage digital system with $\Psi = (D_1, D_2, \dots, D_m)$ fixed. On $\mathcal{M}$, for any execution sequence $S_n^{n_b}$,

$$\min(\mathcal{T}_t) = \begin{cases} \mathcal{T}_t(D_{max}) & \text{if } (n_b + 1)\cdot(D_{max} - l) < n_d \cdot \frac{l}{p-1} \\ \mathcal{T}_t(D_{max}) = \mathcal{T}_t(\frac{D_S}{p-1}) & \text{if } (n_b + 1)\cdot(D_{max} - l) = n_d \cdot \frac{l}{p-1} \\ \mathcal{T}_t(\frac{D_S}{p-1}) & \text{if } (n_b + 1)\cdot(D_{max} - l) > n_d \cdot \frac{l}{p-1} \end{cases}$$

$$\text{where } p = \left\lceil \frac{D_S}{D_{max}} \right\rceil \text{ and } l = D_S - (p - 1)\cdot D_{max}$$

□

According to Lemma 5 and Theorem 6, we see that the shortest possible clock period, D_{max}, may not be optimal. Next, we will prove that any clocking scheme other than the original interstage propagation delays with $D_{max} > d_{max}$ (accordingly $D_S > d_S$) will always result slower execution speed for the same execution sequence of micro cycles.

Figure 10 shows an example of the relationship between the execution time of an execution sequence and the chosen clock period. As shown in Figure 10, if there is any branch micro cycle in the execution sequence, then execution time $\mathcal{T}$ is a discontinuous function of clock period t_{cy}. The slope of each straight-line is determined by the number of branches (n_b). If there is no branch cycle in the execution sequence at all, then $\mathcal{T}$ becomes a straight line as shown with a broken line.

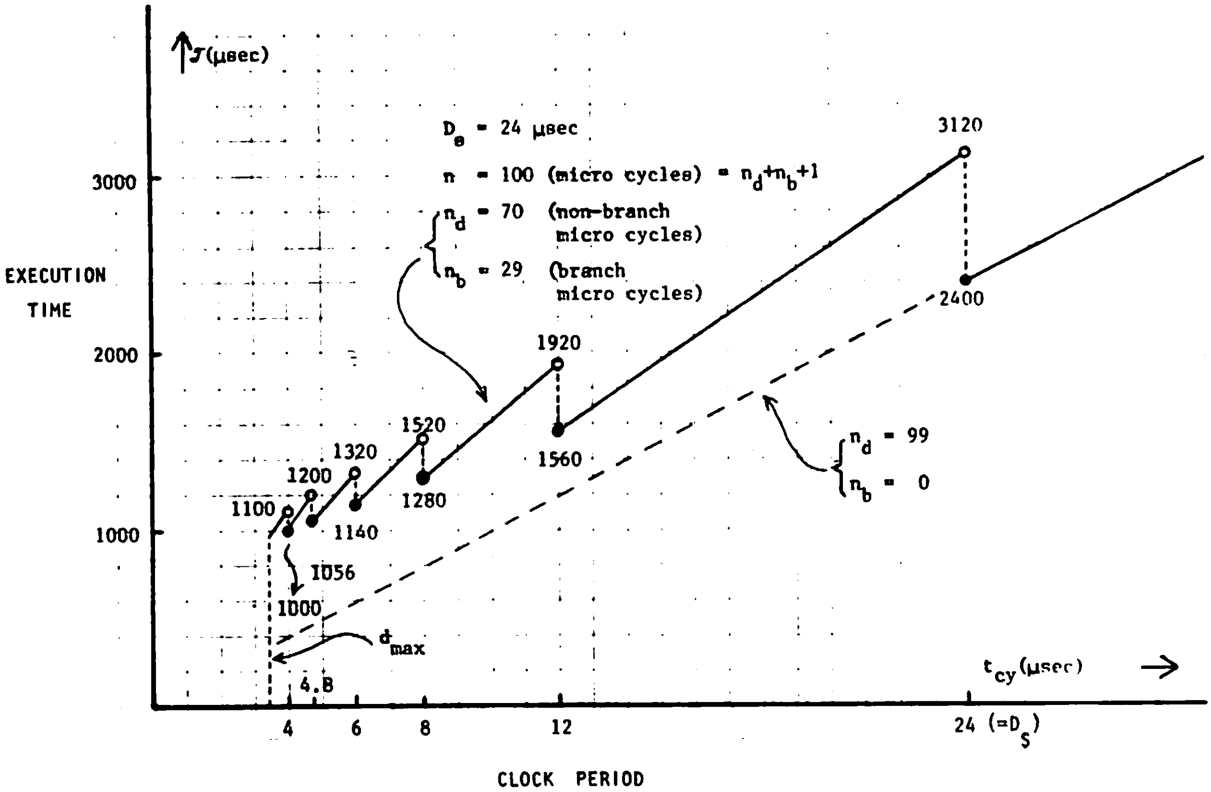


Figure 10: Execution Time vs. Clock Period

Theorem 7 : For an m-stage system, $\mathcal{M}$, with interstage propagation delays $(d_1, d_2, \dots, d_m)$, let $\Psi_d = (d_1, d_2, \dots, d_m)$ (same as the original Ψ_d) and $\Psi_D = (D_1, D_2, \dots, D_m)$, where $D_i \geq d_i$, $1 \leq i \leq m$, be two different clocking schemes.

If $D_{max} > d_{max}$ (then also $D_S > d_S$), then $\min(\mathcal{T}_t^{\Psi_D}) > \min(\mathcal{T}_t^{\Psi_d})$ □

Theorem 7 shows that, even if the optimal clock period resulting in the fastest execution speed is chosen to be longer than the longest interstage propagation delay, increasing the longest stage time (D_{max}) will always result in a slower maximum execution speed. In other words, in Figure 10, if the longest stage time is increased, then the length of the clocking sequence (D_S) is also increased and the execution time curves are shifted upward and to the right.

4.4 Optimal Stage Partitioning

In the previous section, we analyzed the performance of multi stage systems. As we have discussed, if the execution sequence of micro cycles is given (fixed stage partitioning), the execution speed of the system is determined by the clocking scheme. However, the optimality of the clocking scheme is a function of the stage partitioning since the interstage propagation delays determine the minimum requirements of the clocking scheme. To determine whether to use a multistage scheme or not and, furthermore, to choose an optimal number of stages, we need the following:

1. A good stage partitioning method to partition the system into certain number, k, of stages while maximizing execution speed

2. A method for performance comparison of a k-stage scheme to a single stage scheme with given system specifications and statistics regarding the execution sequence(s)

3. A technique for cost analysis (including speed/cost tradeoff) of a multistage system compared to a single stage system.

In the first paragraph, we discuss stage partitioning and performance comparison. Cost analysis and speed/cost tradeoff analysis will be discussed in the following paragraph.

4.4.1 Optimal Stage Partitioning

The stage partitioning problem consists of two subproblems:

1. Given the number of stages, k, get an optimal k-partition of the system to maximize execution speed (i.e., partition each MEG in such a way that the number of partitioned stages is less than k and the longest interstage propagation is minimized).

2. Determine the optimal number of stages, k, which maximizes the execution speed.

The second problem is a superset of the first problem. After determining an optimal partitioning of the system for all possible cases of k, we need to compare the performance of a single stage scheme to a multistage scheme for certain k's. For this reason, we need an efficient algorithm which can determine an optimal k-stage partitioning of a given design, given the desired number of stages. In this paragraph, we develop an optimal stage partitioning algorithm which runs in polynomial time to the number of partitionable points (called intervals) of the design. We first introduce a useful procedure and, based on it, we design the main algorithm.

The following procedure, KPART, determines the minimal number of partitions, k, necessary when the maximum length of stage time is limited to L_{max}. Time delays due to the stage latches are also considered. Let δ_{max} be the longest module propagation delay. If $L_{max} = \delta_{max} + Dss + Dsp$, then k found by KPART is the minimum number of partitions to minimize the length of the longest partition of a given system. The procedure also determines the locations for the stage latches, though there may be some other partitions which have the same L_{max} and k. The algorithm also computes the actual minimum clock period after the stage partitioning. The partitioning procedure will be demonstrated in the next paragraph using an example.

ALGORITHM **KPART(G, Lmax, cutset[N], d[N], dmax, Dss, Dsp, K);**

```
{*       G  input circuit graph                                    *}
{*    Lmax  maximum limit of clock period $\geq$ max{$\delta_i$}            *}
{* cutset  sets of edges on which stage latches are to be added *}
{*       d  stage propagation delay                                *}
{*    dmax  computed minimum clock period $\leq$ Lmax                    *}
{*     Dss  Set up time for stage latches                          *}
{*     Dsp  Storage propagation time of stage latches              *}
{*       K  The number of partitions determined                    *}
```

variable

```
    H    : Set of the starting vertices of the current partition
   SF    : Set of the current searching fronts
   EH    : Set of the edge candidates to have stage latches
  TEMP   : Set of the vertices to be added to current H
   NH    : Set. Starting vertices for the next partition
  w(i)   : Delay time from the previous partition to vertex i
           including $\delta_i$
 OE(i)   : Set of all the edges coming out of vertex i
 IE(i)   : Set of all the edges going into vertex i
mark(i)  : Boolean variable. True if vertex i is already checked
```

```
begin{*KPART*}

for every MEG do

   NH := {root(s)};   EH := { };   K := 1;   dmax := 0;
   w(i) := $\delta_i$ for every i $\epsilon$ NH;

   repeat {* until empty(NH) *}

      {* get starting points of a new partition *}
      K := K + 1;   TEMP := NH;   NH := {};     {* init H and NH *}

      {* initialize the propagation delays *}
      if (k > 1) then
         for every i $\epsilon$ TEMP do w(i) := $\delta_i$ + Dsp;

      repeat {* until empty(TEMP) *}

         H := TEMP;    TEMP := {};

         {* remove all indirect vertices *}
         for every i $\epsilon$ H do H := H - descendents(i);

         {* get searching fronts *}
         SF := {};
         for every i $\epsilon$ H do
            SF := SF + children(i);
            EH := EH + OE(i);     {* candidate loc. for stage latch *}
         for every j $\epsilon$ SF do SF := SF - descendents(j);
         for every j $\epsilon$ SF do mark(j) := false;

         for every i $\epsilon$ H do
            for every j $\epsilon$ children(i) do
               if j $\epsilon$ SF then
                  if w(i) + $\delta_j$ + Dss > Lmax
                     then
                        cutset(k) := IE(j);  {* cut the edges   *}
                        EH := EH - IE(j);    {* and remove them *}

                        {* update stage propagation delays *}
                        d(K) := w(i) + Dss;
                        if d(K) > dmax then dmax := d(K);

                        {* if j is not a leaf,  put it in NH    *}
```

```
                if j not a leaf then NH := NH + j;

                {* if put in TEMP previously, remove it *}
                if mark(j) then TEMP := TEMP - j;

              else if j not a leaf then
                {* move searching head *}
                if not mark(j)
                  then
                    mark(j) := true;
                    TEMP := TEMP + j;
                    w(j) := w(i) + $\delta_j$;
                    EH := EH - IE(j) + OE(j); {* update edges *}

                  {* if j is already in TEMP but with shorter  *}
                  {* delay, then update it with this longer one*}
                  else if w(j) < w(i) + $\delta_j$ then
                    w(j) := w(i) + $\delta_j$;

    until empty(TEMP)

    {* Put all the edges still remaining in the cutset.   *}
    {* These edges go into vertices beyond the current SF *}
    cutset(k) := cutset(k) + EH;

  until empty(NH);

end{*KPART*}
```

This algorithm has been programmed in FRANZ LISP and currently runs on the VAX/750 under UNIX 4.1-2.

Lemma 8 : The number of partitions, k, derived by procedure *KPART* is minimal. □

The proof of this lemma is found in the Appendix.

Run Time Analysis

The algorithm traverses each edge of the MEGs only once. For each traverse, it performs comparisons and additions a constant number of times. Therefore, the total computation time is O(|E|), where |E| is the sum of the number of the edges in all the MEGs. In actual designs, the fan-in/fan-out limits the number of interconnections to/from each component, which can be considered as a constant. Therefore the time complexity of this algorithm is O(|V|), where |V| is the sum of the number of the vertices in all the MEGs. (In the case of infinite fan-in/fan-out designs, the time complexity will be $O(|V|^2)$.)

On the other hand, given a desired number of stages, K, we might like to determine the minimum longest stage time, L_{max}. The following algorithm, OPART, calls a procedure which enumerates all the possible stage times of given MEGs. It uses a *Mergesort* procedure and binary search followed by a call to KPART to check the feasibility of the choice of L_{max}. The binary search continues until the minimum feasible L_{max} is found to determine an optimal K-partition of a system with a given K.

Algorithm *OPART*(K; var d_{max}; var p[K]);

```
{*   K     ....  Input. Desired number of partitions          *}
{*  $d_{max}$ .... Output. Length of the longest partition        *}
{*  p(i)   ....  Output. Locations for the i-th stage latches *}

{* This algorithm uses a binary search technique to determine *}
{* the shortest interstage propagation delay which partitions *}
{* the system into the given number of stages, K.             *}

    begin
       {*enumerate all the possible interstage propagation delays*}
       findintervals(l[1..N]);

       {*sort the propagation delays in non-decreasing order*}
       Mergesort(l[1..N],s[1..N]);


       startpoint := $\lceil N/2 \rceil$

       lastpoint  := N
       level      := 1
       k := m;                               {*initialize k*}

       {* binary search among all the possible time intervals *}
       while startpoint $\neq$ lastpoint do
```

```
        begin

            {* update status *}
            level := level+1

            step := $\lceil$N/(2**level)$\rceil$;


            {* compute the number of stages with given time interval *}
            KPART(MEG, s[startpoint], p[K], d[K], dmax, Dss, Dsp, $k$);

            {* determine the direction of next search *}
            if $k$ > K then startpoint := startpoint + step

                    {* even if $k$ = K, continue search to *}
                    {* find the minimum dmax.              *}
                    else  if $k$ = K then lastpoint := startpoint;
                          startpoint := startpoint - step;


        end
    end
```

Run Time Analysis : The main loop is iterated $O(log(m))$ times (binary search), where m is the total number of modules in the MEGs. There are $O(m^2)$ elements in *s* making *findintervals* $O(m^2)$ (the enumeration of distances between all the possible pairs of nodes in the MEGs). The inner loop inside the main loop takes $O(m)$ steps at each iteration. Thus, the time complexity of the main loop is $O(mlogm)$. The *MERGESORT* for $O(m^2)$ elements takes $O(m^2logm)$ time steps. Therefore, the actual runtime of this algorithm, $O(m^2logm)$, is determined by that of the *Mergesort*.

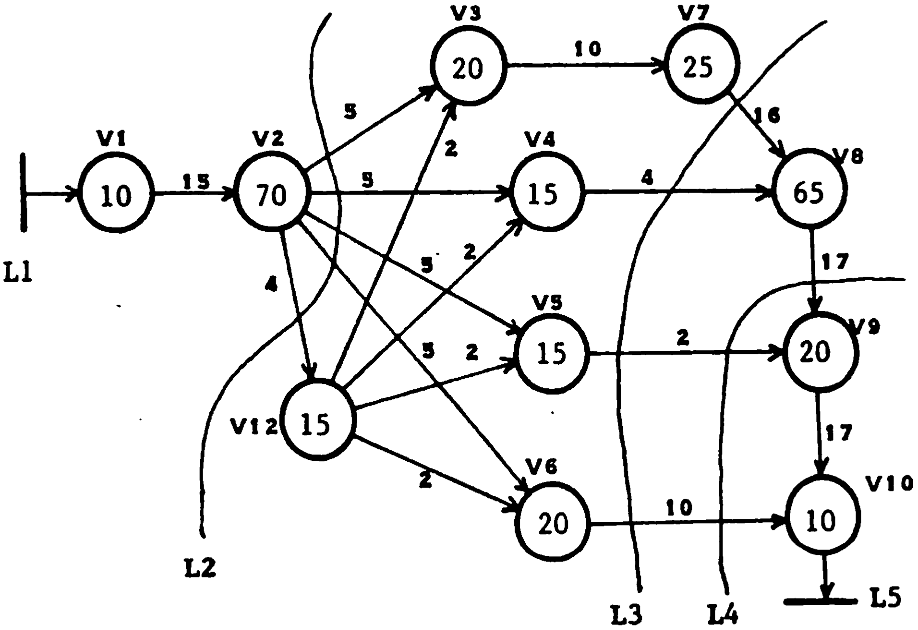
Lemma 9 : d_{max} computed by the algorithm, *OPART*, is minimal.

Proof : Proof is obvious by the construction of the algorithm and its procedures. s[i]'s are the only possible cases of the length of any partition and the algorithm chooses the minimal possible length from s. Therefore, d_{max} is minimal.

In the next chapter, we will demonstrate how the stage partitioning algorithm works with several examples.

4.4.2 An Example Stage Partitioning

Figure 11 shows the weighted MEG for the non-branch group microinstructions of the HP-21MX CPU. The execution sequence of the microinstructions is already shown in Paragraph 3.2.1. The edge- and vertex-weights are computed directly from the actual circuit diagram. Using this example, we will trace some important steps of the partitioning algorithm KPART.



[L1]	=	15 (bits)
D1	=	85 (nsec)
[L2]	=	24 (bits)
D2	=	75
[L3]	=	32
D3	=	80
[L4]	=	29
D4	=	40
[L5]	=	16
d_{max}	=	85 (nsec)
D_s	=	290 (nsec)

Figure 11: An Example of a Weighted MEG of Figure 4

ALGORITHM KPART(G, Lmax=85, cutset, d, dmax, Dss=5, Dsp=10, K);

1. Initially, cutset(1) = $e_{0,1}$ (**the first stage latch L1**), H = $\{v_1\}$, SF = $\{v_2\}$, and EH = $\{e_{1,2}\}$.

2. v_2 can be included in the first partition since $\delta_1 + \delta_2 + \text{Dss} \leq \text{Lmax}$. Thus H is updated and new SF is computed.

   a. v_2 is put in TEMP and moved to H. TEMP is cleared.

   b. SF gets $\{v_3, v_4, v_5, v_6, v_{12}\}$

   c. EH becomes $\{e_{2,3}, e_{2,4}, e_{2,5}, e_{2,6}, e_{2,12}\}$

d. Vertices v_3, v_4, v_5, and v_6 are removed from SF since they are descendents of v_{12}.

3. v_{12} in SF cannot be included in the first partition since ($\delta_1 + \delta_2 + \delta_{12}$ + Dss) exceeds Lmax.

   a. EH = EH - IE(v_{12}). $e_{2,12}$ is removed from EH and put in cutset(2).

   b. v_{12} is put in NH to become a head for the second-stage.

   c. d(1) and dmax are updated with ($\delta_1 + \delta_2$ + Dss) = 85.

4. TEMP is empty. Thus, all the edges in EH are also put in cutset(2). The locations for the second stage latches are $e_{2,3}$, $e_{2,4}$, $e_{2,5}$, $e_{2,6}$, and $e_{2,12}$ {**the second stage latches(L2)**}. d(1) = 85.

5. v_{12} is moved from NH to H and new SF and EH are computed.

   a. w(12) = δ_{12} + Dsp = 25, SF = {v_3, v_4, v_5, v_6}.

   b. EH = {$e_{2,3}$, $e_{2,4}$, $e_{2,5}$, $e_{2,6}$} + {$e_{12,3}$, $e_{12,4}$, $e_{12,5}$, $e_{12,6}$}.

6. All the current searching-front vertices in SF can be included in the second stage. Thus the TEMP is updated to contain v_3, v_4, v_5, and v_6. The corresponding updating procedures during the initialization pass of the inner "repeat" loop are:

   a. H = {v_3, v_4, v_5, v_6}, w(3) = w(6) = 45, w(4) = w(5) = 40.

   b. EH = {$e_{3,7}$, $e_{4,8}$, $e_{5,9}$, $e_{6,10}$}.

   c. SF = {v_7, v_8, v_9. v_{10}} - **descendents**(v_7) = v_7.

7. v_7 can be included in the second stage and thus v_7 becomes the next searching head.

   a. H={v_7} (w(7) = 70), EH={$e_{7,8}$, $e_{4,8}$, $e_{5,9}$, $e_{6,10}$}, SF = {v_8}.

8. Including v_8 violates the maximum stage propagation delay (w(7) + δ_8 + Dss = 140).

   a. NH = {v_8}, cutset(3) = {$e_{7,8}$, $e_{4,8}$, $e_{5,9}$, $e_{6,10}$} {**the third stage latches(L3)**}. d(2) = 75.

   b. d(2) = w(7) + Dss = 75, EH = EH - IE(v_8) = {$e_{5,9}$, $e_{6,10}$}.

9. H = $\{v_8\}$, $w(8) = \delta(8) + Dsp = 75$, SF = $\{v_9\}$.

   EH = EH + $\{e_{8,9}\} = \{e_{5,9}, e_{6,10}, e_{8,9}\}$.

10. $w(8) + \delta_9 + Dss = 100 > Lmax$. Thus, d(3) = 80 and

    cutset(4) = EH = $\{e_{5,9}, e_{6,10}, e_{8,9}\}$ **{the fourth stage latches(L4)}**.

    NH = $\{v_9\}$, w(9) = 30, SF = $\{v_{10}\}$.

    EH = EH - IE(v_9) + OE(v_9) = $\{e_{6,10}, e_{9,10}\}$.

11. The remaining vertices, v_9 and v_{10} becomes the fourth stage and are terminated by **the fifth stage latch(L5)**. d(4) = 40.

Finally, d(5) is determined by the storage propagation delay of L5. The result of the stage partitioning is shown in Figure 11. The corresponding COM is shown below.

```
      d(1)      d(2)      d(3)      d(4)  d(5)              dmax = 85 nsec.
  |---------|---------|---------|---------|--|              dS   = 290 nsec.
      85        75        80        40    10   (nsec.)
```

4.4.3 Performance Comparison - k-stage vs. Single Stage

As the number of branch executions increases, the efficiency of a multistage system gets worse due to the additional delay through the interstage latches. Also, if the longest interstage propagation delay (D_{max}) is too long, the performance of a multistage may not be as good as a single stage system since the amount of overlapped execution time may be very small. Using the execution time equations developed in Sections 4.2 and 4.3, we must compare the average expected execution speeds of all the possible configurations of the system. That is, we must compare:

$\mathcal{T} = \{n_d + \left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot (n_b + 1)\} \cdot t_{cy}$ of multistage configurations and

$\mathcal{T} = n \cdot t'_{cy}$ of a non-overlapped configuration.

where t'_{cy} is the critical path propagation delay of the MEGs.

In addition, we must consider the cost increase. Multistage implementation of a system requires some additional hardware such as interstage latches and a multiphase clock generator. Routing the multiphase clock signals may cause problems in the same way that power routing does.

5 EXAMPLES ILLUSTRATING STATIC CLOCKING SCHEME SYNTHESIS

In this chapter, we demonstrate the results of the static clocking scheme synthesis discussed in the previous chapter. We choose two examples, a microprogrammed CPU, HP-21MX, and a systolic array. The first example, HP-21MX, shows how the proposed technique can be used to complete a partial design. The second example, a systolic array, shows how an already existing system can be sped-up by virtue of execution overlap without changing any data or control flow.

5.1 A Microprogrammed CPU

The circuit graph of the HP-21MX CPU is shown in Figure 3. The corresponding MEGs are shown in Figure 4. Three different results of stage partitionings are shown in Figure 12. (a) is the original 3-stage configuration used. (b) and (c) show the optimal 3-stage and 4-stage partitionings determined by the algorithm OPART. As mentioned before, the algorithm OPART requires an interval enumeration procedure in order to partition the MEGs for every possible length of the interstage propagation delays. Currently, we do not have an efficient interval-enumeration algorithm. Instead, we enumerate the intervals which are possible from the root including the longest module propagation delay, which takes O(|V|) steps. They are (80, 95, 110, 115, 140, 165, 175, 205, 225, 235). Since the partitioning algorithm KPART computes the actual stage propagation delays, these intervals are accurate enough to be used by OPART, the optimal stage partitioning algorithm.

We assume that Dss is 5 nsec. and Dsp is 10 nsec. The 3-stage partition (b) is obtained when L_{max} = 130 nsec. including 15 nsec. total for Dss and Dsp of the stage latches. The 4-stage partition (c) is obtained when L_{max} = 110 also including 15 nsec. total for Dss and Dsp.

The timing values determined by the stage partitionings are listed below. The lengths of the clock phases have a certain safety margin, as shown.

partition	(a)	(b)	(c)	
d_{max}	165	130	105	nsec.
D_{max} (t_{cy})	175	140	110	nsec.
D_1	175	130	105	nsec.
D_2	175	140	85	nsec.
D_3	10	10	110	nsec.
D_4	-	-	10	nsec.
D_S	360	280	310	nsec.
$\|L_i\|$	24	26	60	bits

The corresponding clocking sequences are shown below.

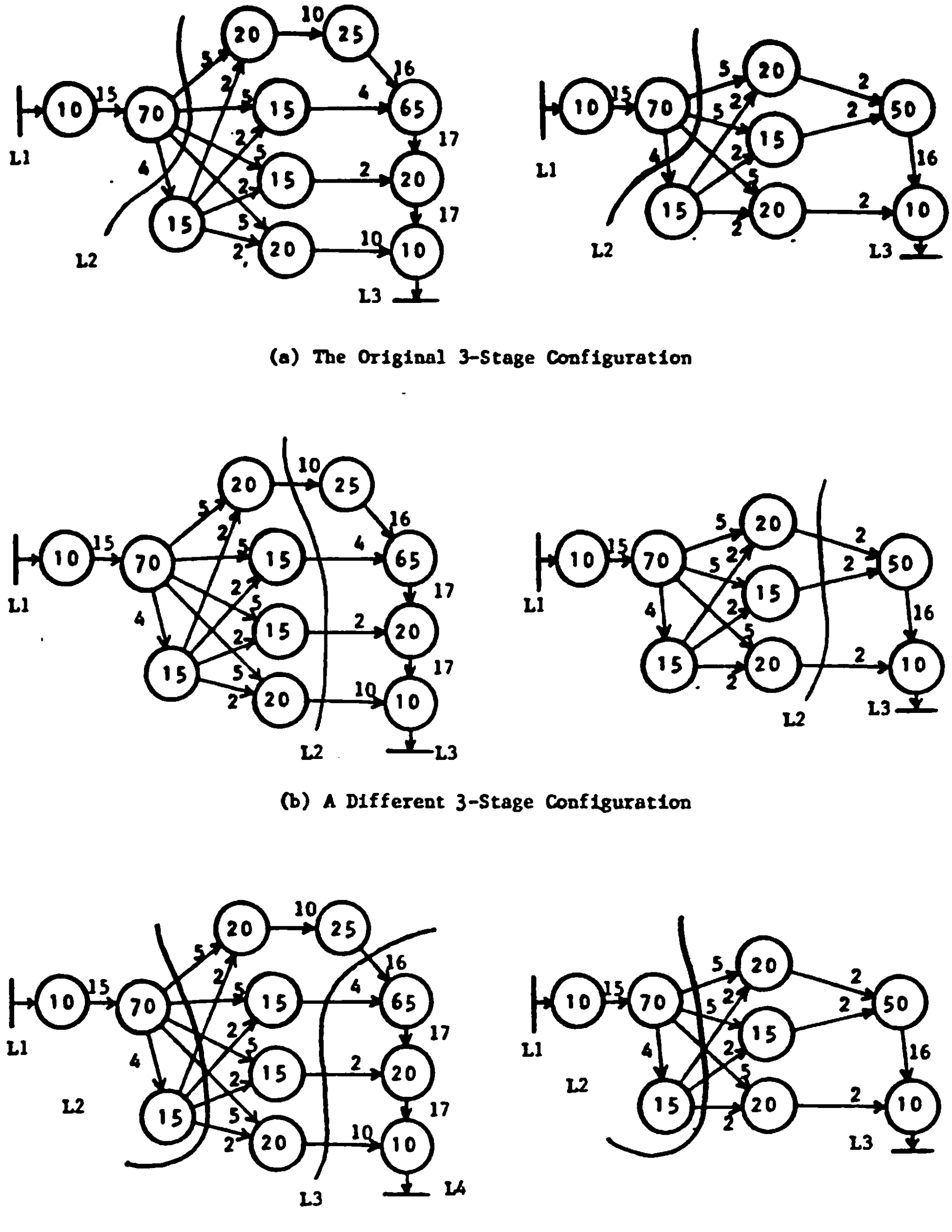
```
          (a)                          (b)                          (c)
   D1          D2     D3        D1      D2     D3        D1      D2       D3      D4
|--------|--------|--|      |-------|-------|--|      |------|--------|--------|--|
  175         175    10        130      140    10        105      85      110     10
```

For configurations (a) and (b), there is no resynchronization overhead. For configuration (c), there may be data contention between two minor cycles, the "store result (D4)" of a micro cycle and the "read operand (D2)" of the next micro cycle, which requires delay of the next micro cycle fetch for one clock period. The branching overhead of the configurations (a) and (c)[8] is two clock periods. For configuration (b), the branching overhead is only one clock period since $\left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1 = 1$ (Lemma 2).

We first compare the original design (a) and our 3-stage partitioning (b). As shown in

[8] 1. Refer to Figure 12 and Equation (4-3) for the calculation of the branching overheads.

2. For all the configurations, we assume that the lengths of the clock phases are fixed and no wait clock periods are added.



(a) The Original 3-Stage Configuration

(b) A Different 3-Stage Configuration

(c) A 4-Stage Configuration

Figure 12: Stage Partitioning of the HP-21MX CPU

the circuit graph of Figure 3, the second stage latch of the original design is the microinstruction buffer, which is usually determined in *ad hoc* fashion and most widely used in microprogrammed controller designs. However, as shown in Table 5.1, we increase the performance of the system significantly by moving the location of the second latch. The cost increase is only 2 latch bits.

The performance comparison of the three configurations is summarized in Table 5.1. For each configuration, the execution times for 100 micro cycles are computed with different numbers of branch cycles and resynchronizations. As shown in the table, the 4-stage configuration shows the best performance in general. In the worst cases when more than half of the micro cycles either branch or need resynchronization, the performance of the 4-stage configuration, (c), is worse than that of (b). However, such cases are unusual. In such cases, we can re-compute the optimal clock period and corresponding execution time using Lemma 5 to determine whether to use the multistage scheme or not.

# of	# of	Execution time (μ-sec)			Normalized initiation rate		
branches(n_b)	resynch	(a)	(b)	(c)	(a)	(b)	(c)
0	0	17.5	14.0	11.0	$\boxed{1.00}$	1.25	1.59
10	0	21.0	15.4	13.2	0.83	1.14	1.33
20	0	24.5	16.8	15.4	0.71	1.04	1.14
40	0	31.5	19.6	19.8	0.56	0.89	0.88
0	50	17.5	14.0	16.5	1.00	1.25	1.06
10	40	21.0	15.4	17.6	0.83	1.14	0.99
20	30	24.5	16.8	18.7	0.71	1.04	0.94
40	10	31.5	19.6	20.9	0.56	0.89	0.84

(n=100)

Table 5.1 Comparison of execution time and initiation rate of three different configurations of the HP-21MX CPU.

(a) The original 3-stage configuration (Fig.12-a)

(b) The reconfigured 3-stage configuration (Fig.12-b)

(c) The 4-stage configuration (Fig.12-c)

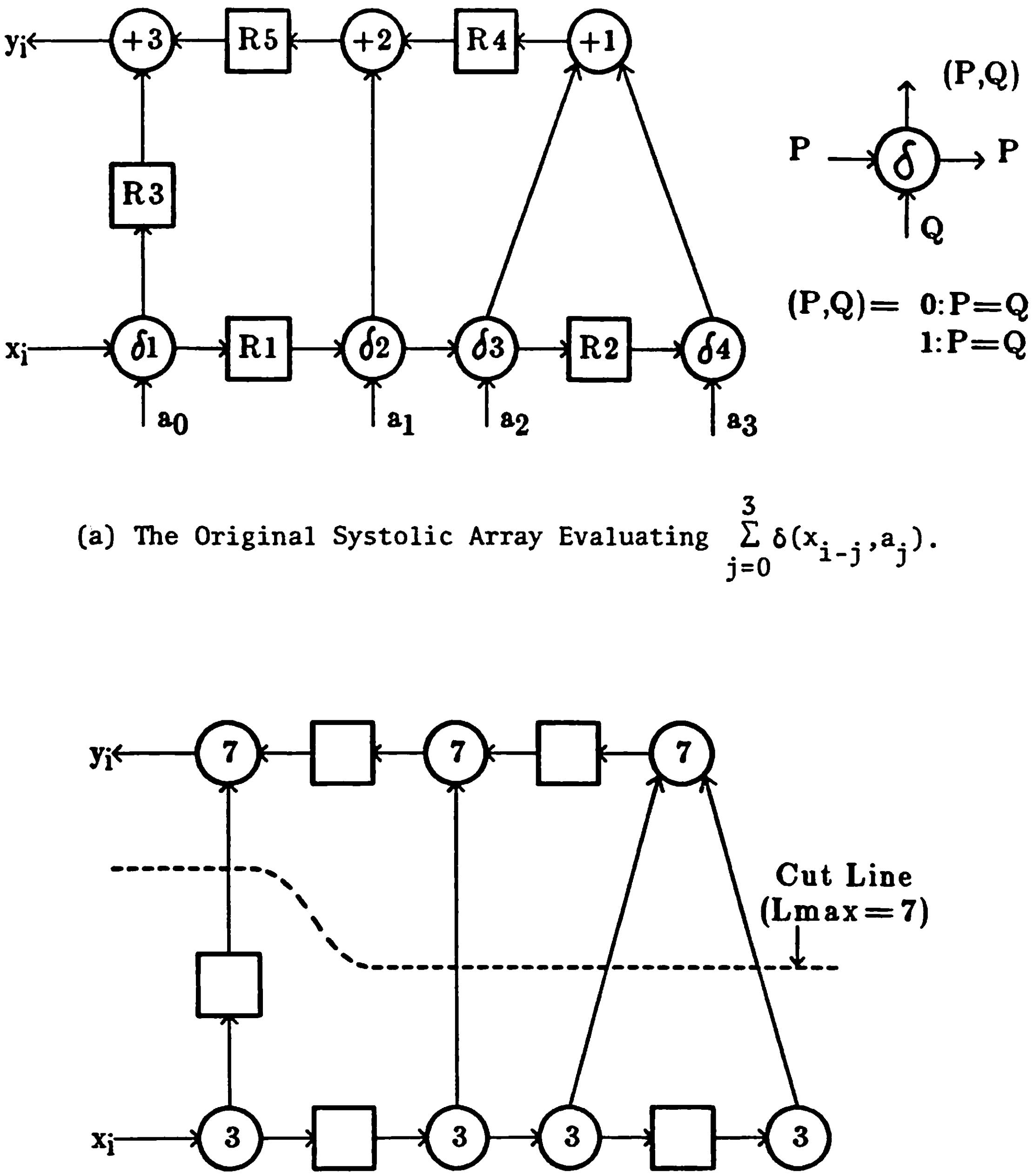
5.2 A Systolic Array

In this example, we show how an already designed systolic array can be sped-up without changing the original data and control flow.

The original systolic array design is taken from [31] and shown in Figure 13-(a), which continuously evaluates the function $y_i = \sum_{j=0}^{3} \delta(x_{i-j}, a_j)$. In the original design, the propagation delays of the registers are assumed to be negligible and we make the same assumption here. The clock period is 13, which is determined by the critical path δ_2 --> δ_3 --> "$+_1$". Each y_i is calculated by clocking all the registers R_1 through R_5 at the same time.

Figure 13-(b) shows the corresponding MEG. The MEG is rooted at the external input port x_i since each micro cycle reads in the input port and all the constants, a_0 through a_3, are always enabled and remain the same. The desired interstage propagation delay is chosen to be the same as the longest module propagation delay, which is 7. As shown in (b), four latches are to be inserted, in between R_3 and "$+_3$", δ_2 and "$+_2$", δ_3 and "$+_1$", and δ_4 and "$+_1$" as the result of the stage partitioning. The resulting COM after the stage partitioning is shown in (c). ϕ_1 clocks the original registers R_1 through R_3, ϕ_2 clocks the added stage latches and ϕ_3 clocks the original registers R_4 and R_5. The resulting clock period is 7 (D2), which is almost twice as fast as the original design.

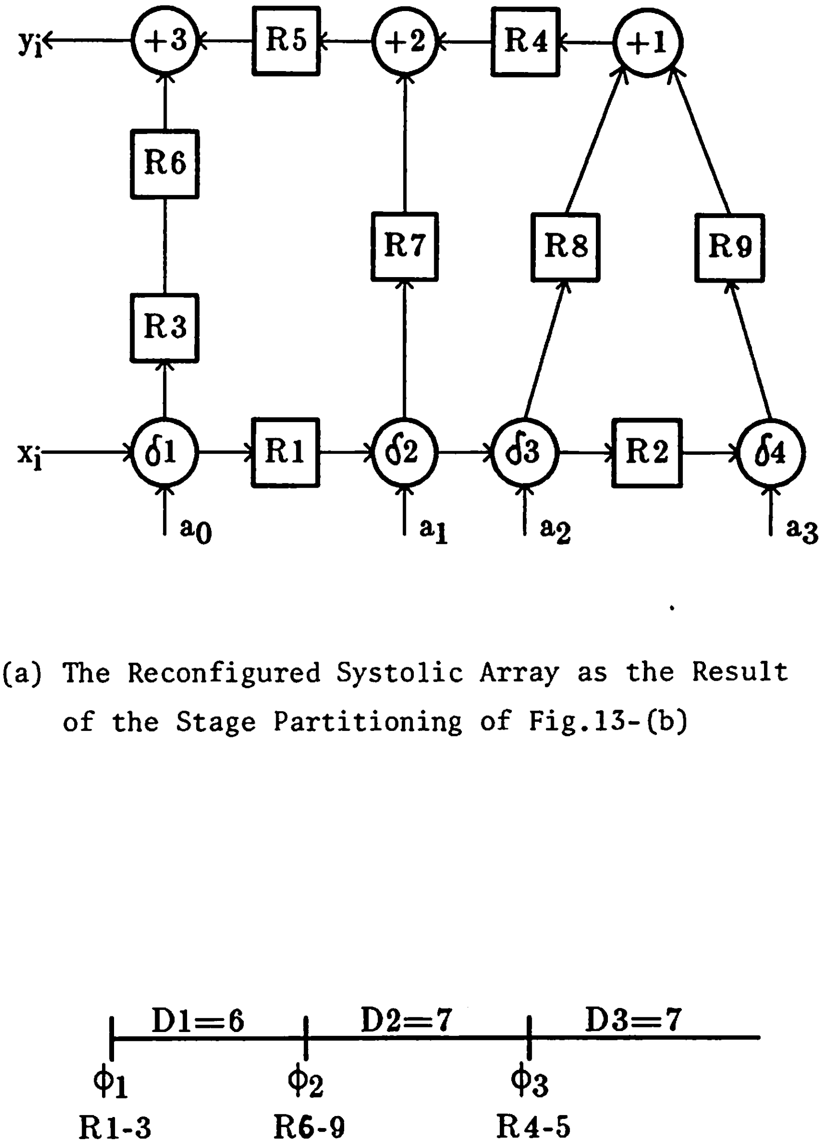
The systolic array continuously evaluates the same function every cycle and there are neither branch nor resynchronization overheads. Accordingly, the throughput rate is inversely proportional to the clock period. Therefore, the throughput rate is increased by (13-7)/7 = 85.7 (%). This throughput rate increase is achieved at the cost of the four added overlap stage latches.



(a) The Original Systolic Array Evaluating $\sum_{j=0}^{3} \delta(x_{i-j}, a_j)$.

(b) The Corresponding MEG and Stage Partitioning

Figure 13: Stage Partitioning of a Systolic Array



(a) The Reconfigured Systolic Array as the Result of the Stage Partitioning of Fig.13-(b)

(b) The Clocking Scheme for the Reconfigured Systolic Array

Figure 14: Stage Partitioning Result of the Systolic Array of Fig. 13

APPENDIX

1 Proofs of Lemma 1 through Lemma 8

Proof :(Lemma 1) To ensure that (i) each stage can have enough time to execute a given subtask and (ii) there is no collision between micro-cycles at any stage, the following three conditions must always be true:

1. from (i), $\phi_{i+1}(j) \geq \phi_i(j) + D_i$, $\forall$ i,j, $1 \leq i < m$, (1)

2. from (ii), $\phi_i(j+1) \geq \phi_{i+1}(j)$, $\forall$ i,j, $1 \leq i < m$, and (2)

3. also from (ii), $\phi_m(j+1) \geq \phi_m(j) + D_m$. (3)

By the definition of $\phi_i(j)$'s, $\phi_i(j+1) = \phi_i(j) + t_{cy}$, $\forall$ i,j, $1 \leq i \leq m$ (4)

By applying conditions (1) and (2) to Equation (4), we get:

$\phi_i(j+1) = \phi_i(j) + t_{cy} \geq \phi_i(j) + D_i$, $\forall$ i,j, $1 \leq i < m$

Thus, $t_{cy} \geq D_i$, $\forall$ i, $1 \leq i < m$ (5)

Also, by applying condition (3) to Equation (4), we get $\phi_m(j) + t_{cy} \geq \phi_m(j) + D_m$.

Thus $t_{cy} \geq D_m$ (6)

By (5) and (6), $t_{cy} \geq D_i$, $\forall$ i, $1 \leq i \leq m$. Therefore $\min(t_{cy}) = D_{max}$. (Q.E.D.)

Proof :(Lemma 2) The only execution interval affected by the replacement is between $\phi_1(j)$ and $\phi_1(j+1)$. Let T1 and T2 be $\phi_1(j+1)$'s before and after the replacement, respectively. Then,

$$T1 = \phi_1(j) + t_{cy} \tag{7}$$

Since I'_j is a branch, $T2 \geq \phi_m(j) + D_m = \phi_1(j) + D_S$ (8)

From Equation (8), we get: $T2 = \phi_1(j) + \left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot t_{cy}$ (9)

Therefore, (branching overhead) $= T2 - T1 = \{\left\lceil \frac{D_S}{t_{cy}} \right\rceil -1\}\cdot t_{cy}$. (Q.E.D.)

Proof : (Theorem 3)

1. Every non-branch micro cycle except the last one is fetched and executed at the clock rate, t_{cy}.
2. The last micro cycle execution takes $\left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot t_{cy}$
3. By 1 and 2, an execution sequence of length n with all non-branch executions is executed in time $\{\left\lceil \frac{D_S}{t_{cy}} \right\rceil + (n-1)\}\cdot t_{cy}$ (10)
4. By Lemma 2, time overhead caused by replacing n_b non-branch executions with branch executions is $n_b\cdot\{\left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1\}\cdot t_{cy}$ (11)
5. Replacing the last micro cycle, I_n, with a branch micro cycle does not change the execution time, since there is no overlapped execution afterwards anyway.

Therefore, the execution time = (10) + (11), or

$$\mathcal{T} = \{\left\lceil \frac{D_S}{t_{cy}} \right\rceil + (n-1)\}\cdot t_{cy} + n_b\cdot\{\left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1\}\cdot t_{cy} = \{n_d + \left\lceil \frac{D_S}{t_{cy}} \right\rceil\cdot(n_b+1)\}\cdot t_{cy} \qquad \text{(Q.E.D.)}$$

Proof : (Lemma 4) By Theorem 3,

$$\mathcal{T}_t(\frac{D_S}{k'}) - \mathcal{T}_t(\frac{D_S}{k}) = n_d \cdot (\frac{D_S}{k'} - \frac{D_S}{k}) + (n_b + 1) \cdot \{\lceil k' \rceil \cdot \frac{D_S}{k'} - D_S\} \tag{12}$$

By evaluating the range of each components in Equation (12), we get:

1. $n_b \leq (n\text{-}1)$, $n_d = n - 1 - n_b \geq 0$
2. $n_b \geq 0$, $n_b + 1 \geq 1$
3. $0 < k' \leq k$, $\frac{D_S}{k'} - \frac{D_S}{k} \geq 0$
4. $\lceil k' \rceil / k' \geq 1$, $\lceil k' \rceil \cdot \frac{D_S}{k'} - D_S \geq 0$

By Equation (12) and from 1 to 4, $\mathcal{T}_t(\frac{D_S}{k'}) - \mathcal{T}_t(\frac{D_S}{k}) \geq 0$.

Therefore, $\mathcal{T}_t(\frac{D_S}{k}) \leq \mathcal{T}_t(\frac{D_S}{k'})$. (Q.E.D.)

Proof : (Lemma 5) $t_{cy} \geq D_{max}$ and, by the definition of p, $\frac{D_S}{p} \leq D_{max} \leq \frac{D_S}{p-1}$. Accordingly we can partition the range of t_{cy} into $D_{max} \leq t_{cy} < \frac{D_S}{p-1}$ and $t_{cy} \geq \frac{D_S}{p-1}$. Then

$$\min(\mathcal{T}_t) = \min[\ \min\{\mathcal{T}_t(t_{cy}) | D_{max} \leq t_{cy} < \frac{D_S}{p-1}\},\ \min\{\mathcal{T}_t(t_{cy}) | t_{cy} \geq \frac{D_S}{p-1}\}\ ]$$

(i) By Lemma 4, $\min\{\mathcal{T}_t(t_{cy}) | t_{cy} \geq \frac{D_S}{p-1}\} = \mathcal{T}_t(\frac{D_S}{p-1})$

(ii) $\frac{D_S}{p} \leq D_{max} < \frac{D_S}{p-1}$. From Theorem 3,

$$\mathcal{T}_t(t_{cy}) = n_d \cdot t_{cy} + p \cdot (n_b + 1) \cdot t_{cy},\ \frac{D_S}{p} \leq t_{cy} < \frac{D_S}{p-1} \tag{13}$$

Equation (13) is a linearly increasing function with the slope $(n_d + (n_b + 1) \cdot p) > 0$.

Thus, $\min\{\mathcal{T}_t(t_{cy}) | D_{max} \leq t_{cy} < \frac{D_S}{p-1}\} = \mathcal{T}_t(D_{max})$. Therefore, according to (i) and (ii),

$\min(\mathcal{T}_t) = \min\{\mathcal{T}_t(D_{max}),\ \mathcal{T}_t(\frac{D_S}{p-1})\}$, where $p = \left\lceil \frac{D_S}{D_{max}} \right\rceil$ (Q.E.D.)

Proof : (Theorem 6) We prove the theorem by comparing $\mathcal{T}_t(D_{max})$ and $\mathcal{T}_t(\frac{D_S}{p\text{-}1})$. From Theorem 3, we know that:

$$\mathcal{T}_t(D_{max}) = n_d \cdot D_{max} + (n_b + 1) \cdot p \cdot D_{max} \text{ and}$$

$$\mathcal{T}_t(\frac{D_S}{p\text{-}1}) = n_d \cdot \frac{D_S}{p\text{-}1} + (n_b + 1) \cdot D_S$$

$$= n_d \cdot (D_{max} + \frac{l}{p\text{-}1}) + (n_b + 1) \cdot \{(p\text{-}1) \cdot D_{max} + l\}$$

Thus, $\mathcal{T}_t(D_{max}) - \mathcal{T}_t(\frac{D_S}{p\text{-}1}) = n_d \cdot (- \frac{l}{p\text{-}1}) + (n_b + 1) \cdot (D_{max} - l)$ (14)

Therefore, from Equation (14),

1. if $(n_b + 1) \cdot (D_{max} - l) < n_d \cdot \frac{l}{p\text{-}1}$, then $\mathcal{T}_t(D_{max}) - \mathcal{T}_t(\frac{D_S}{p\text{-}1}) < 0$ and therefore, $\mathcal{T}_t(D_{max}) < \mathcal{T}_t(\frac{D_S}{p\text{-}1})$
2. if $(n_b + 1) \cdot (D_{max} - l) = n_d \cdot \{l/(p - 1)\}$, then $\mathcal{T}_t(D_{max}) = \mathcal{T}_t(\frac{D_S}{p\text{-}1})$
3. if otherwise, $\mathcal{T}_t(D_{max}) \geq \mathcal{T}_t(\frac{D_S}{p\text{-}1})$.

(Q.E.D.)

Proof : (Theorem 7) By Theorem 3 and Lemma 4, we know that

$$\overset{\Psi}{\mathcal{T}}{}_t^{D}(t_{cy}) = n_d \cdot t_{cy} + (n_b + 1) \cdot \left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot t_{cy},\ t_{cy} \geq D_{max} \tag{15}$$

$$\overset{\Psi}{\mathcal{T}}{}_t^{d}(t_{cy}) = n_d \cdot t_{cy} + (n_b + 1) \cdot \left\lceil \frac{d_S}{t_{cy}} \right\rceil \cdot t_{cy},\ t_{cy} \geq d_{max} \tag{16}$$

$$\min(\overset{\Psi}{\mathcal{T}}{}_t^{D}) = \min\{ \overset{\Psi}{\mathcal{T}}{}_t^{D}(D_{max}),\ \overset{\Psi}{\mathcal{T}}{}_t^{D}(\frac{D_S}{p\text{-}1}) \},\ \text{where } p = \left\lceil \frac{D_S}{D_{max}} \right\rceil \tag{17}$$

$$\min(\overset{\Psi}{\mathcal{T}}{}_t^{d}) = \min\{ \overset{\Psi}{\mathcal{T}}{}_t^{d}(d_{max}),\ \overset{\Psi}{\mathcal{T}}{}_t^{d}(\frac{d_S}{q\text{-}1}) \},\ \text{where } q = \left\lceil \frac{d_S}{d_{max}} \right\rceil \tag{18}$$

Then, from Equations (15) and (17),

$$\begin{aligned}
\overset{\Psi}{\mathcal{T}}{}_t^{D}(D_{max}) &= n_d \cdot D_{max} + (n_b + 1) \cdot \left\lceil \frac{D_S}{D_{max}} \right\rceil \cdot D_{max} \\
&\geq n_d \cdot D_{max} + (n_b + 1) \cdot \left\lceil \frac{d_S}{D_{max}} \right\rceil \cdot D_{max} \\
&\geq \overset{\Psi}{\mathcal{T}}{}_t^{d}(D_{max}) \qquad (19)
\end{aligned}$$

$$\begin{aligned}
\overset{\Psi}{\mathcal{T}}{}_t^{D}\left(\frac{D_S}{p\text{-}1}\right) &= n_d \cdot \frac{D_S}{p\text{-}1} + (n_b + 1) \cdot D_S \\
&\geq n_d \cdot \frac{D_S}{p\text{-}1} + (n_b + 1) \cdot \left\lceil \frac{d_S}{D_S} \cdot (p\text{-}1) \right\rceil \cdot \frac{D_S}{p\text{-}1} \\
&\geq \overset{\Psi}{\mathcal{T}}{}_t^{d}\left(\frac{D_S}{p\text{-}1}\right) \qquad (20)
\end{aligned}$$

CASE 1: If $\overset{\Psi}{\mathcal{T}}{}_t^{d}(d_{max}) < \overset{\Psi}{\mathcal{T}}{}_t^{d}\left(\frac{d_S}{q\text{-}1}\right)$, then, obviously from Equations (19) and (20) and by Lemma 5,

From Equation (19), $\overset{\Psi}{\mathcal{T}}{}_t^{D}(D_{max}) \geq \overset{\Psi}{\mathcal{T}}{}_t^{d}(D_{max}) > \overset{\Psi}{\mathcal{T}}{}_t^{d}(d_{max})$ and

from Equation (20), $\overset{\Psi}{\mathcal{T}}{}_t^{D}\left(\frac{D_S}{p\text{-}1}\right) > \overset{\Psi}{\mathcal{T}}{}_t^{d}\left(\frac{D_S}{p\text{-}1}\right) > \overset{\Psi}{\mathcal{T}}{}_t^{d}(d_{max})$

Thus, $\min(\overset{\Psi}{\mathcal{T}}{}_t^{D}) > \min(\overset{\Psi}{\mathcal{T}}{}_t^{d})$

CASE 2: If $\overset{\Psi}{\mathcal{T}}{}_t^{d}(d_{max}) \geq \overset{\Psi}{\mathcal{T}}{}_t^{d}\left(\frac{d_S}{q\text{-}1}\right)$, then from Equations (15) and (16),

$$\begin{aligned}
\overset{\Psi}{\mathcal{T}}{}_t^{D}\left(\frac{d_S}{q\text{-}1}\right) &= n_d \cdot \frac{d_S}{q\text{-}1} + (n_b + 1) \cdot \left\lceil \frac{D_S}{d_S} \cdot (q\text{-}1) \right\rceil \cdot \frac{d_S}{q\text{-}1} \\
&\geq n_d \cdot \frac{d_S}{q\text{-}1} + (n_b + 1) \cdot q \cdot \frac{d_S}{q\text{-}1} \\
&> n_d \cdot \frac{d_S}{q\text{-}1} + (n_b + 1) \cdot d_S \\
&> \overset{\Psi}{\mathcal{T}}{}_t^{d}\left(\frac{d_S}{q\text{-}1}\right)
\end{aligned}$$

Therefore, if $D_{max} > d_{max}$, $\min(\overset{\Psi}{\mathcal{T}}{}_t^{D}) > \min(\overset{\Psi}{\mathcal{T}}{}_t^{d})$ is always true. (Q.E.D.)

Proof : (Lemma 8)

```
|      P(1)      |    P(2)                 P(K-2)|      P(K-1)      | P(K) |
|------|--|-----|-----|--|----/ /----|------|-----|--|--------|---|--|
| u1          v1 | u2                      v(k-2)|u(k-1)     v(k-1)|uk  vk|
                                          =u(k-2)
```

As depicted above, let ui and vi be the left-most and right-most intervals (boundaries) of i-th partition (for any MEG). By Lemma 1, the minimal length of the longest partition is L_{max}. In order to have a smaller k, at least the length of one of the partitions must be increased and the boundaries be changed. Thus, at least one pair of vi and u(i+1) will be in the same partition, say Pi (either Pi or P(i+1)). Then,

1. If we move u_{i+1} into P_i, then ui must not remain in Pi in order not to increase the maximal length of the partition, L_{max}. Also, for the same reason, ui cannot be absorbed into P(i-1) without partitioning P(i-1).

2. Also, for the same reason, going the opposite direction, Pi can only contain, at most, up to interval v(i+1).

Thus the number of stages remains, at least, the same. By repeating the adjustment according to the rules 1 and 2 until u1 and vk are reached, we can see that the number of partitions cannot be decreased. Therefore, k is minimal.

(Q.E.D.)

References

[1] Agerwala, T.
Microprogram Optimization: A Survey.
IEEE Transactions on Computers C-25(10):962-973, October, 1976.

[2] Aho, A. and Ullman, J.
Principles of Complier Design.
Addison-Wesley, Massachusets, 1977.

[3] Andrews M.
Principles of Firmware Engineering in Microprogram Control.
Computer Science Press, 1980.

[4] Berg, H. K.
A Model of Timing Characteristics in Computer Control.
Euromicro 5, July, 1979.

[5] Boulaye, G. G.
Microprogramming.
John Wiley & Sons, New York, N.Y., 1971.

[6] Breuer,M. (ed.).
Digital System Design Automation, vol.I: Theory and Techniques.
Prentice-Hall, Englewood Cliffs, N.J., 1972.

[7] Chen, T. C.
Introduction to Computer Architecture.
SRA, Chicago, 1975, chapter 9. Overlap and Pipeline Processing.

[8] Cook, P., Chung, H. and Stanley, S.
A Study in the Use of PLA-Based Macros.
Solid-State Circuits SC-14:833-840, October, 1979.

[9] Cotton, L. W.
Circuit Implementation of High-Speed Pipeline Systems.
In *Proceedings of FJCC*, pages 489-504. AFIPS, 1965.

[10] Darringer, J.
The Description, Simulation and Automatic Implementation of Digital Computer Processors.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, May, 1969.

[11] Dasgupta, S. and Tartar, J.
The identification of maximal parallelism in straight-line microprograms.
IEEE Transactions on Computers C-25(10):986-992, October, 1976.

[12] Davidson, E. et. al.
Effective Control for Pipelined Computers.
In *COMPCON Digest*, pages 181-184. 1975.

[13] Dennis, J. B. and Misunas, D.P.
A Preliminary Data Flow Architecture for a Basic Data Flow Processor.
In *2nd Symposium on Computer Architecture*, pages 126-132. 1975.

[14] Dervos, D. and Parker, A. C.
A Technique for Automatically Producing Optimized Digital Designs.
In *Proceedings of the Mediterranean Electrotechnical Conference,Athens*, pages B2.04. IEEE, May, 1983.

[15] Estrin, G.
A methodology for design of digital systems - supported by SARA at the age of one.
In *Proceedings of National Computer Conference*, pages 313-324. NCC, 1978.

[16] Foulk P. W. and O'Callaghan J.
AIDs - an integrated design system for digital hardware.
In *IEE Proceeding Vol.127, No.2.* IEE, March, 1980.

[17] Friedman, A.. Menon, P.
Theory & Design of Switching Circuits.
Computer Science Press, Woodland Hills, California, 1975.

[18] Hafer, L.
Automated Data-Memory Synthesis : A Formal Model for the Specification, Analysis and Design of Register-Transfer Level Digital Logic.
PhD thesis, Dept of Electrical Engineering, Carnegie Mellon University, Pittsburgh, Pa., May, 1981.

[19] Hafer, L., and Parker, A.
A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic.
IEEE Transactions on Computer-Aided Design CAD-2(1), January, 1983.

[20] Hitchcock, C.Y.
Automated Synthesis of Data Paths.
Master's thesis, Carnegie-Mellon University, 1983.

[21] Horowitz, E. and Sahni, S.
Fundamentals of Computer Algorithms.
Computer Science Press, 1978.

[22] Irani,K., McClain,G.
Optimal Design of Central Processor Data Paths.
Technical Report 58, Systems Engineering Laboratory, University of Michigan, Ann Arbor, Michigan, May, 1972.

[23] Kartashev, S. P., Kartashev, S. I. and Vick, C. R.
Designing and Programming Modern Computers and Systems.
Prentice-Hall, 1982, chapter 1. Historic Progress in Architectures for Computers and Systems.

[24] Katzan, H.
Computer Organization and the System/370.
Van Norstrand Reinhold, 1971.

[25] Keller, R.
Look-Ahead Processors.
Computing Surveys (7), December, 1975.

[26] Knapp, D. and Parker, A.
A Data Structure for VLSI Synthesis and Verification.
Technical Report, Digital Integrated Systems Center, Dept. of EE-Systems, University of Southern California, October, 1983.

[27] Kogge, P. M.
The Architecture of Pipelined Computers.
McGraw-Hill, New York, N.Y., 1981.

[28] Lang, D. E., Agerwala, T. K., Chandy, K. M.
A Modeling approach and design tool for pipelined central processors.
In *Proceedings of the 6th Annual Symposium on Computer Architecture.* IEEE Computer Society, April, 1979.

[29] Lawson,G.
Design Style Selector, An Automated Computer Program Implementation.
Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., August, 1978.

[30] Leiserson,C.E., Rose,F.M. and Saxe,J.B.
Digital circuit optimization.
Technical Report, Dept. of Electrical Engineering and Computer Science, M.I.T., 1982.

[31] Leiserson, C. E., Rose, F. M. and Saxe, J. B.
Optimizing synchronous circuitry by retiming.
In *Proceedings of Third Caltech Conference on VLSI*, pages 23-36. Computer Science Press, 1983.

[32] McKeeman, W. M.
Introduction to Computer Architecture.
SRA, Chicago, 1975, chapter 7. Stack Computers.

[33] Nagle, A.
Automated Design of Digital-System Control Sequencers from Register-Transfer Specifications.
PhD thesis, Carnegie-Mellon University, 1980.

[34] Parker, A.C., et al.
The CMU Design Automation System .
In *Design Automation Conference Proceedings No. 16.* ACM SIGDA, IEEE Tech. Comm. on Design Automation, June, 1979.

[35] Patterson,D.
STRUM: Structured Microprogram Development System for Correct Firmware.
IEEE Transactions on Computers c-25(10):974-985, October, 1976.

[36] Ramamoorthy, C. V. and Li H. F.
Pipeline Architecture.
ACM Computing Surveys 9(1):61-102, March, 1977.

[37] Robertson, E.
Microcode Bit Optimization is NP-Complete.
IEEE Transactions on Computers C-28(4):316-319, April, 1979.

[38] Sastry, S. and Parker, A.
The Complexity of Two-Dimensional Compaction of VLSI Layouts.
In *Proceedings of the 1982 IEEE International Conference on Circuits and Computers*, pages 402-406. IEEE, September, 1982.

[39] Snow, E.
Automation of Module Set Independent Register Transfer Level Design.
PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April, 1978.

[40] Thomas,D.
The Design and Analysis of an Automated Design Style Selector.
PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April, 1977.

[41] Zimmermann,G.
The MIMOLA Design System: A Computer Aided Digital Processor Design Method.
In *Proceedings of the 16th Design Automation Conference*, pages 53-58. ACM SIGDA, IEEE Computer Society - DATC, June, 1979.