CRI 85-36

Synthesis of Optimal Pipeline Clocking Schemes[1]

Nohbyung Park and Alice Parker

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, California 90089-0781

9 January 1985

Abstract

This paper describes a technique for automatic synthesis of clocking schemes for pipelined digital hardware. Two steps in the synthesis process are described: the determination of number and location of pipeline partitions (stages) and the insertion of delays in the pipe in order to achieve minimum throughput latency without resource conflicts. We focus on a single reused resource in the same cycle in this paper, although extension of the solution techniques to cover multiple reused resources is straightforward. We present

1. a method to partition the system into stages subject to the number of stages or the maximum stage time, and
2. an algorithm for delay insertion.

Some examples of pipeline clocking synthesis are given, and the delay insertion algorithm is proven to be optimal.

---

[1]This research was supported by Army Research Office Contract #DAAG29-83-K-0147.

Table of Contents

List of Figures

1. Introduction

The performance of virtually all digital systems is influenced by the choice of clocking scheme. The partitioning of functions into time steps, the number of clock phases, the length of each phase, (i.e. how to pipeline) and the assignment of functions to clock phases are all part of the clocking scheme synthesis task, and each of these choices affects performance.

This paper describes a special class of clocking problem - the control of synchronous pipelines, of which systolic arrays are themselves a special case. The assumption is made that functions have been allocated to pipeline resources and the partial ordering of function execution has been determined. The synthesis task described involves the determination of number and location of pipeline partitions (stages) and the insertion of delays in the pipe in order to achieve minimum throughput latency without resource conflicts. We focus on a single reused resource in the same cycle in this paper, although extension of the solution techniques to cover multiple reused resources is straightforward.

The emphasis of these synthesis techniques is on minimizing the change in the control and data flow of a given partial or complete design. If new operators are added to speed up the execution, both data and control flow must be altered. 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.

The paper begins by defining the problem in Section 2. Section 3 describes the approach to the solution. In Section 4, we present an algorithm for pipeline delay insertion which minimizes fixed latency and avoids resource conflicts. We also prove that the solutions are guaranteed to be optimal.

1.1. Two Sequencing Levels of a Digital System

In digital systems with two-level control structures[2], sequencing is carried out on the

---

[2]Most multi-level systems can be considered to be two-level for the purposes of this synthesis task.

macro and **micro** levels. An execution instance of a machine instruction or a major loop of a finite state machine corresponds to a **macro cycle** and an execution instance of a microinstruction or a state of a finite state machine corresponds to a **micro cycle**. Figure 1-1 shows an example of a two-level microprogrammed CPU.

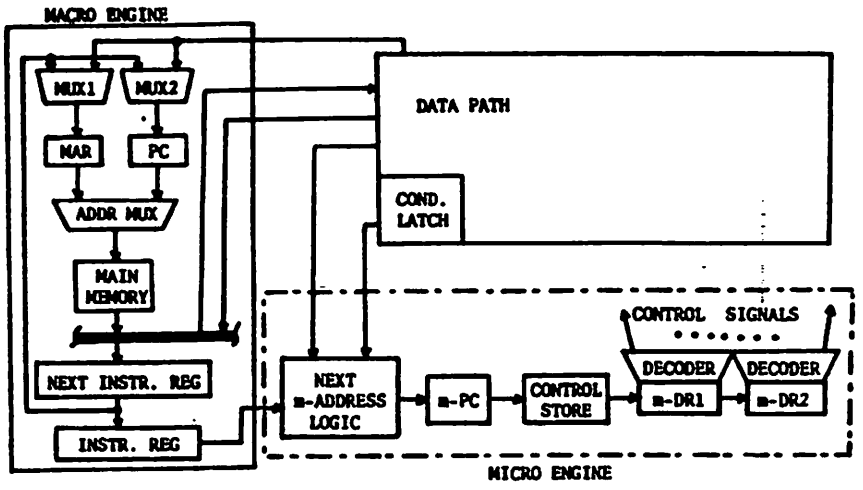


Figure 1-1: Sequencing Engines of a Digital System

Macro cycles consist of sequences of one or more micro cycles. Overlapping macro cycles (i.e. pipelining) is achieved 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 or some micro cycle of the current macro cycle may fetch the next macro cycle task from main memory in advance.* A micro cycle consists of **minor** cycles. Each minor cycle consists of register transfers which read, transform and store data and/or control values from storage elements to storage elements. These storage elements are used to buffer the flow of the values between functional elements, and are called **stage latches**. For the micro engine of Figure 1-1, 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. Overlapping micro cycles is achieved by proper partitioning of micro cycles into sequences of minor cycles, as shown in Figure 2-3. For the purposes of this paper, we can assume that the pipelining we are achieving can be either at the macro or micro level, but the focus will be on micro level overlap.

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, and
3. between the data path stages.

If there is no resource conflict and no branches are executed, the maximum execution speed of a micro engine is determined by the longest interstage propagation delay. If resources are reused, however, delays may have to be inserted between minor cycles in order to avoid resource conflicts. The most common performance measures for pipelined systems are the latency between reuses of the pipe as well as delay through the pipe. Delay insertion gives us a longer pipe, but potentially a higher throughput rate.

1.2. Definition of the Clocking Scheme Synthesis Task

For a given datapath design and a control architecture, the task of clocking scheme synthesis is as follows.

The inputs are (i) a partial datapath and control design with chosen functional units and minimum number of required storage elements[3], (ii) the types of micro cycles (e.g., microinstruction formats or Node-Module-Range bindings [8], which specify the direction and propagation time of data values through functional elements during micro cycles) and (iii) the expected sequences of micro cycles to be executed. The constraint imposed on the task is to minimize the latency through the control/data path pipe. The outputs of clocking scheme synthesis are (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) the insertion of stage delays to avoid resource conflicts.

---

[3]For any data path, the minimum number of storage elements is determined by the maximum number of live values [1] 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.

1.3. Related Work

Related research has been performed by [3], [2], [5], [9], [13], [12], [14], [4], [7], and [10]. However, solutions to this specific problem have not been reported previously. We have previously reported on the general clocking synthesis problem with no resource conflicts [11].

2. The Problem Model

We 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.

Definition 1: The **module propagation delay**(MPD) is the maximum port propagation delay for all possible pairs of input and output ports of a module.

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, called MEGs (Micro-cycle Execution Graphs).

Definition 2: The **MEG** for a set of one or more micro cycles, G(V,E), is a **rooted** directed acyclic data-flow graph where the set of vertices, V, represents the operations on values to perform, and the set of directed edges, E, represents the order of operations (data dependency) during the execution of the micro cycles in the set. Each vertex is weighted with the propagation delay of the module (δ) which are assigned to perform the operation. Each edge is weighted with the bitwidth of the value it carries.

The MEGs are rooted. In general, *for any synchronous sequential circuit or finite state machine, 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 choose a subset of these as the starting point of every cycle. For a microprogrammed micro engine, it can be either the μ-PC or the microinstruction register. In the case of a hardwired sequencer, it can be either the state counter or feedback state-memory.

Figure 2-1 shows the MEGs for non-branch and branch type microinstructions of the

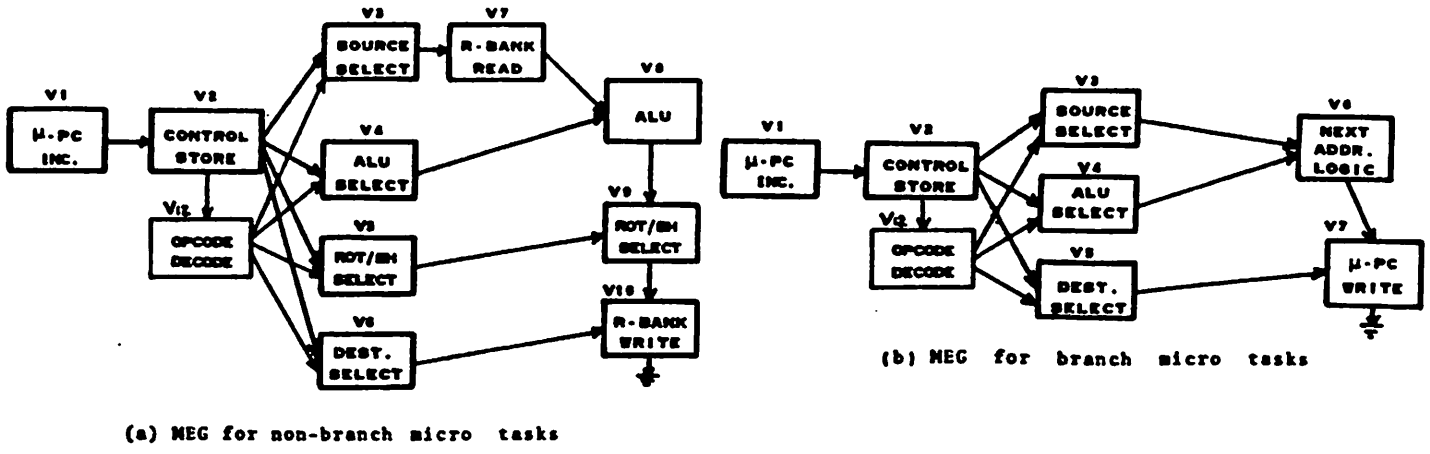


Figure 2-1: Examples of the MEG

HP21MX CPU. They are rooted at the μ-PC. The steps in the execution sequence of the non-branch type microinstructions are

1. Increment PC (v1) and fetch the micro cycle pointed to by the PC (v2).
2. Decode the control fields of the fetched micro cycles (v12, v3, v4, v5, and v6).
3. Fetch the operand from selected register (v7).
4. Perform the selected ALU operation (v8).
5. Perform the selected rotate/shift operation (v9).
6. Store the result in the selected register (v10).

Once the locations and connections for the stage latches are determined, the interstage propagation delays are determined and thus the minimum requirements for clocking and timing of micro cycles are determined. 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.

Figure 2-2 shows examples of the COMs derived from the results of stage partitioning (control step partitioning) of the MEGs of Figure 2-1. The locations of the stage latches are indicated by the edge-cut lines in the MEGs. 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. Then the timing requirements are specified as

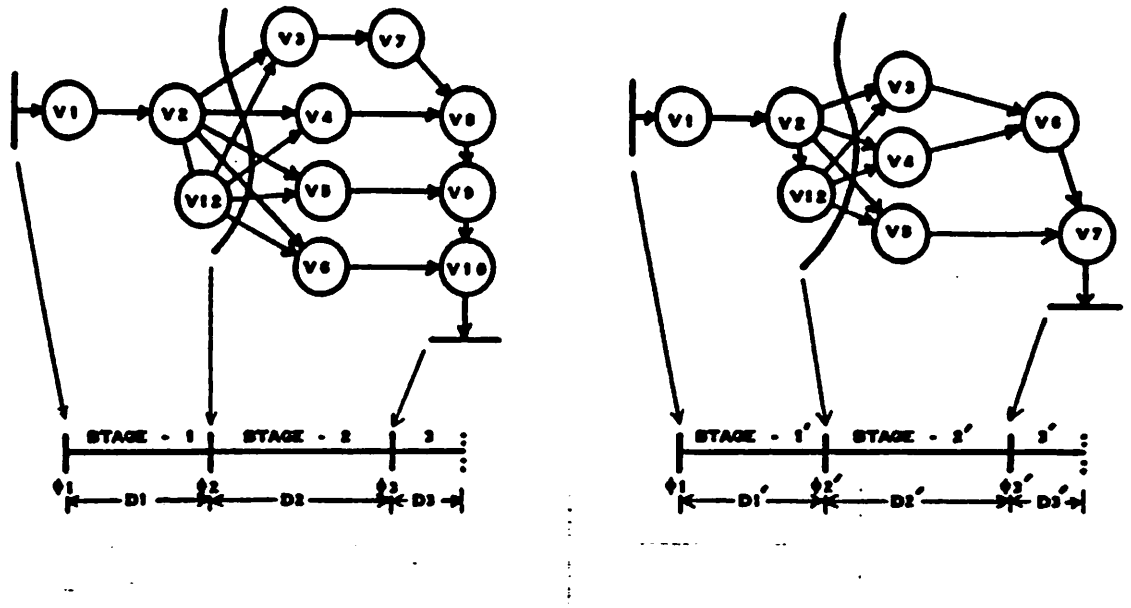


Figure 2-2: The COM Derived from the Results of Stage Partitioning

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

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

where D_{SS} and D_{SP} are the set-up and propagation delays of storage elements defined by Hafer [6]. Note that Dsp(L1) and Dss(L3) are included in the MEG as v_1 and v_{10}. At the end of the chain, D3 must be added in order to evaluate the completion time of all the effects of an execution of a micro cycle. The next micro cycle can only read the result of the current micro cycle after the buffer has been clocked and the stored values propagated to its outputs. The COMs can be used to determine a major clock period as well as the number and length of the clock phases.

Branching must also be considered. A branch micro cycle delays fetch of the next micro cycle until the earliest fetch clock cycle after completion of the branching. Thus, the overall initiation rate will also depend on the frequency of branch micro cycles.

If there is data or resource contention between any two micro cycles, 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.

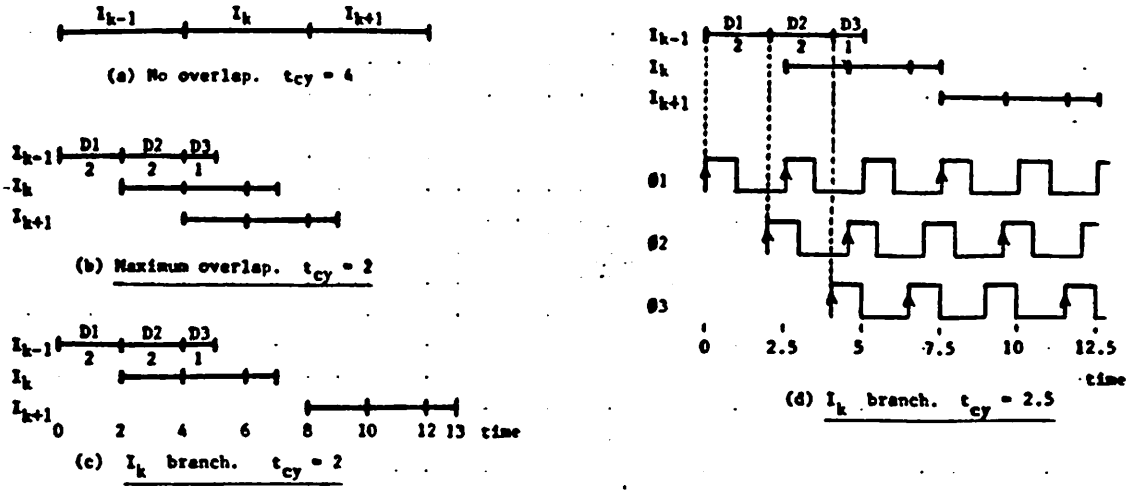


Figure 2-3: Examples of Micro Cycle Sequencing and Clocking

3. The Solution Technique

The pipeline synthesis problem involves determination of the number and location of pipeline stages, along with the insertion of delays in order to avoid resource conflicts. In order to perform these tasks, we rely on the previous results of static clocking scheme synthesis for sequencing of *loop-free* micro cycles. By *loop free* we mean that each micro cycle uses the same module no more than once. These techniques were used to speed up the execution time of an HP-21MX computer and produced about a 60% performance improvement over the original machine. Detailed proofs of the lemmas and theorems used to obtain these results are found in [11].

3.1. Optimal Stage Partitioning

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 and, furthermore, to choose an optimal number of stages, we need

1. a method to partition the system into k stages while maximizing execution speed,
2. a method for performance comparison of a k-stage scheme to a single stage scheme with given statistics regarding the execution sequence(s), and
3. a technique for cost analysis (including speed/cost tradeoff) of a multistage system compared to a single stage system.

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 the MEG(s) in such a way that the actual number of partitioned stages is less than k and the longest interstage propagation is minimized).
2. Determine the optimal number of stages, k_{opt}, which maximizes the execution speed.

The second problem is a superset of the first one. 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. We have developed an optimal stage partitioning algorithm (called KPART) which runs in polynomial time ($O(n^2 log n)$) with respect to the number of nodes in the MEG. This algorithm 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. On the other hand, given a desired number of stages k_{opt}, we might like to determine the minimum longest stage time L_{max}. Another algorithm called OPART 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. These algorithms have been programmed in FRANZ LISP and run on the VAX/750 under Berkeley UNIX 4.2.

3.2. Delay Insertion

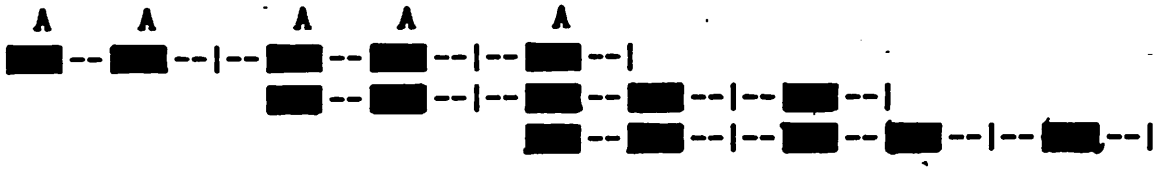
In this section, we discuss insertion of delay steps (minor cycles) in order to increase the overall performance of a pipeline. A delay step can be implemented by delaying the clocking of a stage latch by one clock cycle and/or inserting an intermediate buffering latch so that the stage latch clocking is delayed. We assume that fixed microcycle time (fixed latency) is used. In fact, since we can achieve the minimal possible average latency by using a fixed latency scheme, there is no reason to use a variable latency scheme.

The delay insertion step begins with the partitioned MEG, as shown in Figure 3-1 (a). Stages which use the shared resource A are shaded. An interval vector which contains the number of stages between the shared resource is constructed as shown in (b). Suppose that the minimal possible fixed latency is 5 (refer to Lemma 3 in the next section). Pipelining the MEG with the minimum possible fixed latency of 5 will produce resource conflicts. Delays must be inserted into the pipe in order to avoid such conflicts.

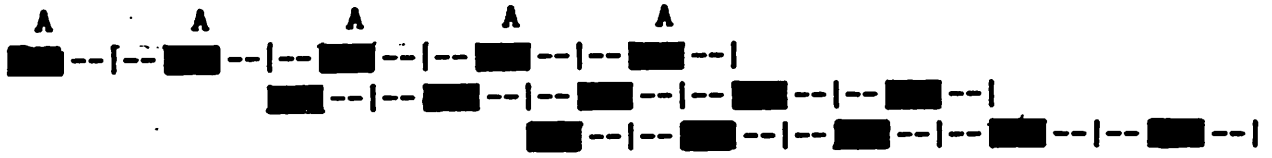
(a) A MEG

I = (2 3 2 3)

(b) The interval vector for (a)



(c) Pipelining the original MEG with fixed latency 5



(d) The result of the delay insertion (I = (3 3 3 3))

Figure 3-1: An Example of Delay Insertion

The delay insertion algorithm, which we will discuss in detail in the next section, initially produces a number of quick solutions. Two solutions are produced by simply incrementing the interval vector until a solution is found; the algorithm increments the intervals from left-to-right to produce one solution, and right-to-left for the other. The two solution interval vectors for this example are [2 4 2 6] and [3 3 3 3]. One or more solutions are found by incrementing all the intervals until they become equal (modulo n, n=5 for this example). Proofs of correctness are found in the next section.

Potential solutions are determined in the following manner: If any contiguous intervals

sum to 0 modulo n, where n is the chosen fixed latency, then there are resource conflicts and the solution must be discarded. A proof of this assumption is also found in the next section. These solutions are used as upper bounds on the search for the optimal solution, since a more optimal solution would be found with lower increments on the intervals. The interval vector is incremented from right-to-left. Whenever the increment on a digit becomes n-1, the digit is reset to its original value and the next digit to the left is incremented. Also, whenever the sum of the increments reaches the upper bound, the digit just incremented is reset to the original value and the next digit to the left is incremented. Then the new interval vector is examined. Although the complexity of this search in time is $O[n\text{-}1^n]$ in the worst case, the bounded procedure converges very quickly. When n=10, which is quite a long pipe, the Franz LISP procedure converges in a few minutes of CPU time on a VAX 11/750.

4. Insertion of Delay Steps

4.1. Definition of the Problem

Patel [12] has shown how the throughput of a pipeline can be improved by proper insertion of delays. In this section, we formalize the problem of delay insertion, and suggest a practical technique which finds an optimal delay insertion configuration for the original pipeline. By "optimal delay insertion configuration", we mean that the "fixed latency" is minimized and the number of delays to be inserted in order to achieve the minimal fixed latency is minimized.

Lemma 3: (Shar's Lemma [14]) For any statically configured pipeline executing some reservation table, the minimum average latency (MAL) is always greater than or equal to the maximum number of marks in any single row of the reservation table. (For the proof, see [9] pp.83)

□

Assuming that buffering latches for the stage latches can be added by inserting delays, we can always make the fixed latency equal to the maximum number of marks in any single row. We will construct the proof and the solution technique to this proposition step by step.

Definition 4: The position vector of a stage, P_s, is defined as a vector of

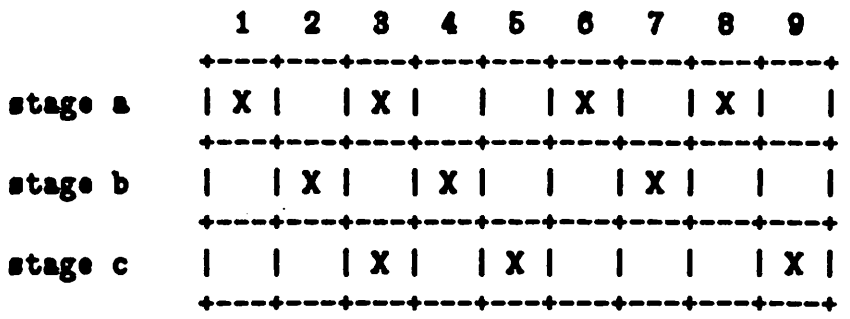


Figure 4-1: An Example Reservation Table

column indices which are marked in the corresponding row, s, in the reservation table. The **interval vector** of a stage, I_s, is defined as a vector of the distances between the adjacent marks in the corresponding row, s, in the reservation table.

For example, the position and interval vectors for the reservation table in Figure 4-1 are $P_a = (1\ 3\ 6\ 8)$, $P_b = (2\ 4\ 7)$, $P_c = (3\ 5\ 9)$, $I_a = (2\ 3\ 2)$, $I_b = (2\ 3)$, and $I_c = (2\ 4)$.

Lemma 5: Let $P_s = (p_1\ p_2\ \ldots\ p_n)$ be the position vector for some stage, s. Also let N be the chosen fixed latency which is greater than or equal to n. If and only if $(p_j - p_i)$ is not divisible by N for every pair of i and j, $1 \leq i \leq j \leq n$, there will be no conflict in using stage s.

□

Proof: The time steps when stage s is used are

$p_i, p_i + N, p_i + 2N, \ldots\ldots, p_i + kN, \ldots$ (due to p_i) and

$p_j, p_j + N, p_j + 2N, \ldots\ldots, p_j + k'N, \ldots$ (due to p_j).

For any k and k', $(p_j + k'N) - (p_i + kN) = (k' - k)N + (p_j - p_i)$. Therefore if and only if $(p_j - p_i)$ is not divisible by N, then $(p_j + k'N) - (p_i + kN)$ cannot be zero for any k and k', and hence, $(p_j + k'N)$ and $(p_i + kN)$ cannot occur during the same time slot. Thus, if this is true for every possible pair of i and j, $1 \leq i \leq j \leq n$, then there will be no conflict in using stage s.

Corollary 6: Let $I_s = (i_1\ i_2\ \ldots\ i_{n-1})$ be the interval vector for some stage, s. Then if and only if $\sum_{j=u}^{v} i_j$ is not divisible by N (N > n-1) for every pair of u and v, $1 \leq u \leq v \leq n-1$, then there will be no conflict in using the stage s.

□

Proof: For any pair of u and v, $1 \leq u \leq v \leq n-1$,

$$\sum_{j=u}^{v} i_j = i_u + i_{u+1} + \ldots + i_v$$
$$= (p_{u+1} - p_u) + (p_{u+2} - p_{u+1}) + \ldots + (p_{v+1} - p_v)$$
$$= p_{v+1} - p_u$$

Therefore, the rest of the proof is the same as that of Lemma 5.

According to Lemmas 3 and 5, and Corollary 6, the problem of optimal delay insertion for a single stage conflict can be stated as follows.

Problem Statement: For a given interval vector, $I_s = (i_1\ i_2\ i_3\ \ldots\ i_{n-1})$, where n is the minimal possible fixed latency according to Lemma 3, find any or all increment vectors, $K = (k_1\ k_2\ k_3\ \ldots\ k_{n-1})$, such that

$\left(\sum_{j=u}^{v} (i_j + k_j) \right) / n \neq 0$, for any u and v, $1 \leq u \leq v \leq n-1$, while minimizing $\sum_{j=1}^{n-1} k_j$.

The worst-case complexity of finding all possible such increment vectors is $O((n-1)^n)$, since each element of K, k_j, can have any value in between 0 and n-1 (modulo n arithmetic) and there are n-1 elements in K.

4.2. An Optimal Delay Insertion Algorithm

In this section, we discuss a branch-and-bound algorithm which finds all possible increment vectors K which satisfy the conditions stated in the problem statement in the previous section (the single resource conflict case). In spite of the exponential complexity of the problem, this algorithm runs fast enough to be used for realistic size and complexity pipelines (e.g., the maximum number of marks in a single row of a reservation table is less than 15).

Algorithm DINS

{The original interval vector: $I = (i_1\ i_2\ \ldots\ i_{n-1})$}
{Increment vector $K = (k_1\ k_2\ \ldots\ k_{n-1})$}

Phase 1: *Compute Approximate Upper Bound On* Σk_i

; Find three quick solutions and get the minimal one.

(A) a: Set $m = 2$
b: Increment m-th element (modulo n) of K until the subsequence $((i_1+k_1)\ (i_2+k_2)\ \ldots\ (i_m+k_m))$ does not have any sub-interval the sum of which is divisible by n.
c: Increment m. If $m < n$, then GOTO Step b.
d: UB1 <-- Σk_i

(B) Repeat Step A in reverse order, i.e., from $m = n-2$ to 1.
UB2 <-- Σk_i

(C) a: UB3 = infinite
b: Do for $m = 1$ to n-1
1: Check if m and n have a common divisor larger than 1. If yes, GOTO Step 4.
2: Increment k_i for every i until $\{ (i_i + k_i) \bmod n \} = m$.
3: If $\Sigma k_i <$ UB3, then UB3 <-- Σk_i
4: Continue.

(D) Set UB = min (UB1 UB2 UB3)

Phase 2: *Branch-and-Bound Search for the Solution(s)*

; Increment the increment vector and check the new configuration.
; Whenever a better solution is found, update the upper bound, UB.

(E) Set the increment vector to zero ($K = (0\ 0\ \ldots\ 0)$).

(F) (Increment K vector)
a: Set $d = n-1$
b: Increment k_d modulo n. If no carry, then GOTO Step G.
c: If d=1, then report the results and STOP.
d: Increment d and go to Step b.

(G) (Check if Σk_i exceeds UB)
  a: Set d = n-1
  b: If $\Sigma k_i > UB$,
      THEN Set $k_d = 0$ and decrement d.
      ELSE GOTO Step H.
  c: If d = 0,
      THEN report the results and STOP.
      ELSE GOTO Step b.

(H) (Check the sequence I + K)
  a: If I + K has any subsequence the
     sum of which is divisible by n,
     then GOTO Step F.
  b: (I+K does not have any subsequence the sum
     of which is divisible by n)
     If $\Sigma k_i < UB$
      THEN set results to {(I+K)}.
        $UB = \Sigma k_i$
     If $\Sigma k_i = UB$
      THEN add {(I+K)} to results.
  c: GOTO Step F.

4.3. Analysis of the Algorithm

The algorithm DINS, described in the previous section, basically checks the feasibility of every possible configuration of the increment vector K, and compares the results. Therefore the algorithm guarantees completeness (finds all possible solutions) and correctness (every solution has the minimal number of delays). The only steps of the algorithm which must be proven correct are those which find upper bounds (Phase 1) on the sum of the increment vector and those which skip some configurations of the increment vector.

In Phase 1 of the algorithm, the only substep that does not check whether the quick solution satisfies Corollary 6 is Step C. Any solution increment vector K produced by Step C can be defined as: $((i_j + k_j) \bmod n) = l$ for some integer l (Step C.b.3), where l and n do not have any common divisor in between 2 and l inclusively (Step C.b.2). Lemma 7 proves that the solution increment vectors produced as such satisfy Corollary 6 and are therefore correct.

Lemma 7: Let $I = (i_1\ i_2\ \ldots\ i_{n-1})$ be a sequence of non-negative integers. Suppose that $(i_1 \bmod n) = (i_2 \bmod n) = (i_{n-1} \bmod n) = l$, for some integer, $1 \leq l \leq n-1$. If and only if l and n do not have any common divisor in between 2 and l inclusively, then $(\sum_{j=u}^{v} i_j) \bmod n \neq 0$, for any pair of u and v, $1 \leq u \leq v \leq n-1$.

□

Proof:

(1) First prove that $(\sum_{j=u}^{v} i_j) \bmod n \neq 0$ only if l and n do not have a common divisor. The proof is by contradiction. Suppose that l and n have a common divisor, m, $1 < m \leq l$. Then we can rewrite l and n in terms of m as $l = c_l m$ and $n = c_n m$ where c_l and c_n are positive integers. Then

$$(\sum_{j=u}^{v} i_j) / n = (v - u + 1)l / n = (v - u + 1)c_l m / c_n m \qquad (1)$$

Since $1 < m$, $c_n \leq n/2$. Thus, we can always choose some u and v such that $(v - u + 1) = c_n$ and hence $(v - u + 1)c_l m / c_n m = c_l$,

and consequently $(\sum_{j=u}^{v} i_j) \bmod n = 0$.

(2) (Proof by contradiction) Now assume the summation mod n is equal to zero. Let $I' = (l\ l\ \ldots\ l)$, $|I'| = n-1$. Then

$$(\sum_{j=u}^{v} i_j) = xl,\ 1 \leq x \leq n-1 \qquad (2)$$

Suppose that xl is divisible by n. Then $xl = yn$ for some positive integer y, and can be rewritten as $l/n = y/x$. From Equation (2), we know that $1 < x \leq n-1 < n$ and therefore $0 < l/n < 1$. Hence, for $l/n = y/x$ to be true, there must be some integer common divisor, m, such that $n = mx$ and $l = my$, and $m > 1$.

□

In Step G, the algorithm may skip more than one configuration of K. However, it is obvious that the sum of any skipped configuration of K exceeds the current upper bound, and hence there is no need to check it.

The initial upper bound derived from the three types of quick solutions by Step 1

serves as the initial bound on the increment vector. As the algorithm finds new solutions, the upper bound becomes tighter (lower) and the chances for branch pruning increase.

References

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

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

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

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

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

[6] 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.

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

[8] 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.

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

[10] Nagle, A.
Automatic Design of Sequencers for The Control of Digital Hardware.
PhD thesis, Carnegie-Mellon University, October, 1980.

[11] Park, N. and Parker, A.
Synthesis of Optimal Clocking Schemes for Digital Systems.
Technical Report DISC/84-1, Dept. of EE-Systems, University of Southern California, May, 1984.

[12] Patel, J. H. and Davidson, E. S.
Improving the Throughput of a Pipeline by Insertion of Delays.
In *IEEE/ACM 3rd Ann. Symp. Computer Arch.*, pages 159-163. 1976.

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

[14] Shar, L. E.
Design and Scheduling of Statically Configured Pipelines.
Technical Report Digital Systems, Lab Report SU-SEL-72-042, Stanford University, September, 1972.