OPD: A Toolset for Optimized Pipeline Design

Suresh Krishna

Dept. of EECS, CS Division,

University of California, Berkeley.

ABSTRACT

OPD is a set of four co-ordinated synthesis and analysis tools for the design of optimized VLSI datapath and CPU pipelines. Together, these tools cover a wide range of design tasks, from functional partitioning of the system into pipeline stages through datapath definition and clocking, to the handling of technology-specific constraints.

OPD has tools for stage partitioning, clocking scheme calculation, datapath sequencing, and pipeline initiation scheduling. We describe these tools as well as the optimization algorithms they use. We discuss both probabilistic and heuristic optimization techniques.

We show how it is possible to rapidly design high-quality pipelines by using OPD with existing CAD tools such as logic synthesizers. We show large as well as small examples taken from VLSI chips and discrete logic machines.

February 29, 1988

Acknowledgements

I thank Prof. Carlo H. Séquin, my Advisor, who greatly helped me and guided me in all aspects of my research.

The contribution of several people from the Berkeley CAD Group was essential. I thank Srinivas Devadas and Jeff Burns for many helpful discussions throughout the course of this work.

I thank Kinson Ho, whose comments greatly improved the clarity of this report.

I gratefully acknowledge the financial support provided by the Defense Advanced Research Projects Agency under its DARPA-VLSI Program (contract number N00039-87-C-0182).

Contents

Section I	**Project Overview**
Section II	**Stage Partitioning, Phase Calculation**
II.1	Problem Description
II.2	Optimization Algorithms
II.3	Phase Calculation
Section III	**Phase Assignment**
III.1	Problem Description
III.2	Simulated Annealing
III.3	Heuristic Algorithms
III.4	Constraints Completeness
Section IV	**Reservation Table Scheduling**
Conclusion	
References	
Appendix 1	**SP Input Format, HP21-MX Example**
Appendix 2	**PA Input Format, SPUR-FPU Example**
Appendix 3	**RISC-II Example**
Appendix 4	**Control Synthesis**

Section I - Project Overview

We examine why it is desirable to have a set of tools for Optimal Pipeline Design, and show how the sub-tools of OPD work together and with existing tools such as logic synthesizers. We also compare OPD to similar tools.

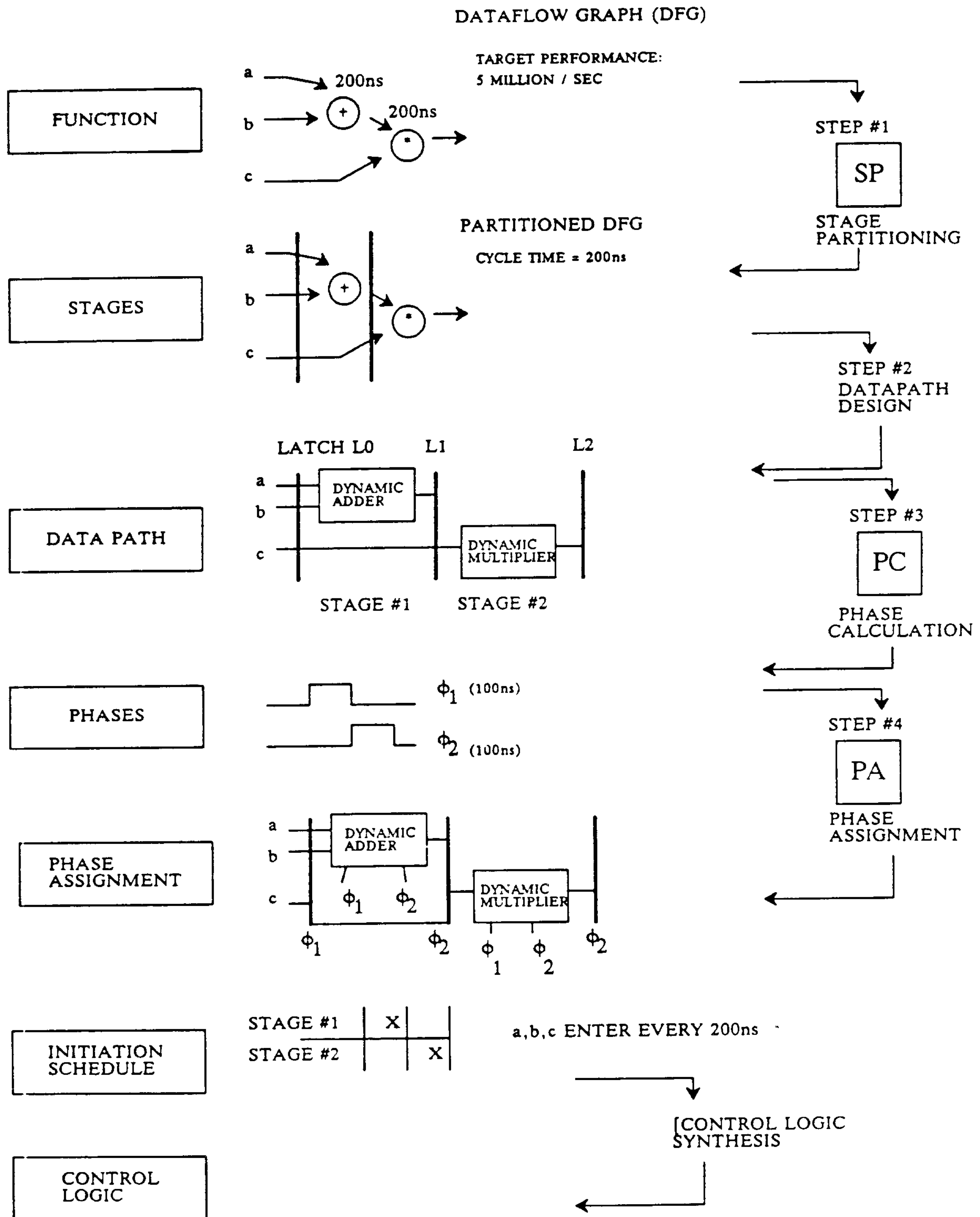
1. Motivation

A pipelined design is often the best way to achieve high performance at a reasonable cost. Pipelining a system dramatically enhances its performance for a relatively modest increase in cost. This is why most digital machines today are pipelined. However, as Fig.1 shows, many complex and inter-dependent steps are required to design a good pipe. OPD provides a tool set to assist the designer in each one of the steps in Fig.1, with the exception of step 2. The objective of OPD is to help designers produce better pipes in less time.

We use two primary quantities to measure the quality of a pipeline design. The first is throughput, or average flow rate; this indicates the processing speed of the system. The second is the complexity of the datapath and of the logic required to control the datapath. It is important to know this, since complex logic implies longer delays and increased chip area. The synthesis tools of OPD aim to maximize throughput and minimize complexity, while the analysis tools rapidly calculate these two quantities.

Fig.1 shows how the system specifications evolve through various levels as the design progresses. Initially, we know the behavioral specification [Blackburn 85] of the system, which defines the system's function. The first step is to split this function into a set of overlapping subfunctions that will define the pipeline stages. The second step is to choose a detailed datapath that will implement the system's functions. This leads to a register-transfer level (RTL) [Snow 78] description of the system. This step, and only this step, is entirely left to the designer. OPD is not a datapath synthesizer; however, it can quickly analyze the speed of a proposed datapath. Of course, this second step depends very much on the target technology. The third step is to determine a clocking scheme to optimally

FIG 1: PIPE DESIGN STEPS



sequence the above set of stages, using the datapath emerging from step 2. This step involves choosing the number of clock phases and their lengths. The fourth step is to determine on which clock cycle and phase each gate in the datapath should be run on. The fifth step is to choose an initiation sequence for the above datapath. This tells us exactly when new data should be allowed to enter the pipeline to maximize processing throughput.

These steps are complex and time-consuming if performed manually. Furthermore, they are highly inter-related, and thus require a great amount of discipline to maintain overall consistency of the design decisions. The first step, stage partitioning, requires the exploration of a large number of alternatives. The third step, clocking scheme determination, depends on the behavioral stage partitioning as well as the detailed datapath implementation. It is hard to keep track of both of these manually. The fourth and fifth steps, which deal with detailed datapaths, are especially complex in the case of MOS-VLSI [Mead 80] [Weste 85] systems, since there are many factors to be considered during these steps. One such factor is the use of complex multiphase clocking schemes ([Weste 85] chap. 5.4). These offer great flexibility, but complicate the designer's task by introducing additional degrees of freedom. Similarly, the use of transparent latches as pipeline staging elements ([Weste 85] ch 5.4 and [Kogge 81] ch 2) makes it possible to effectively trade time across pipe stages - a fast stage can make up for a slow one. However, the designer must keep track of signal delays across these latches. Other factors to be kept in mind are possible constraints that acceptable phase assignments must satisfy. Such constraints can result from external signals, or can arise because part of the phase assignment has already been done, because gates are shared across pipe stages, because of precharging schemes, or because of layout area constraints that limit the number of control lines.

OPD has the ability to take into account all the detailed constraints found at the register-transfer level when assigning clock phases to gates and latches in the datapath. OPD does not stop at the behavioral level, where the system description is usually rough and approximate; it goes from the behavioral level right down to the final datapath blocks.

Moreover, OPD contains features that make it suitable both for VLSI datapath and CPU pipeline design tasks.

2. OPD Description and Usage Scenario

OPD consists of a set of four tools, that cover the design levels from system behavior to datapath and logic synthesis. These tools work primarily in a top-down fashion, where behavioral design tasks are performed before layout related optimization. However, OPD runs fast, which makes it possible to iterate each design step many times. Moreover, each tool within OPD accepts a rich set of constraints that can be used to feed OPD with information gained in previous design cycles. Fig.2 shows how these tools can be used together and with existing tools, such as logic synthesis routines. We now describe each tool briefly. We will describe each tool in detail later; our objective here is to show how the tools work together.

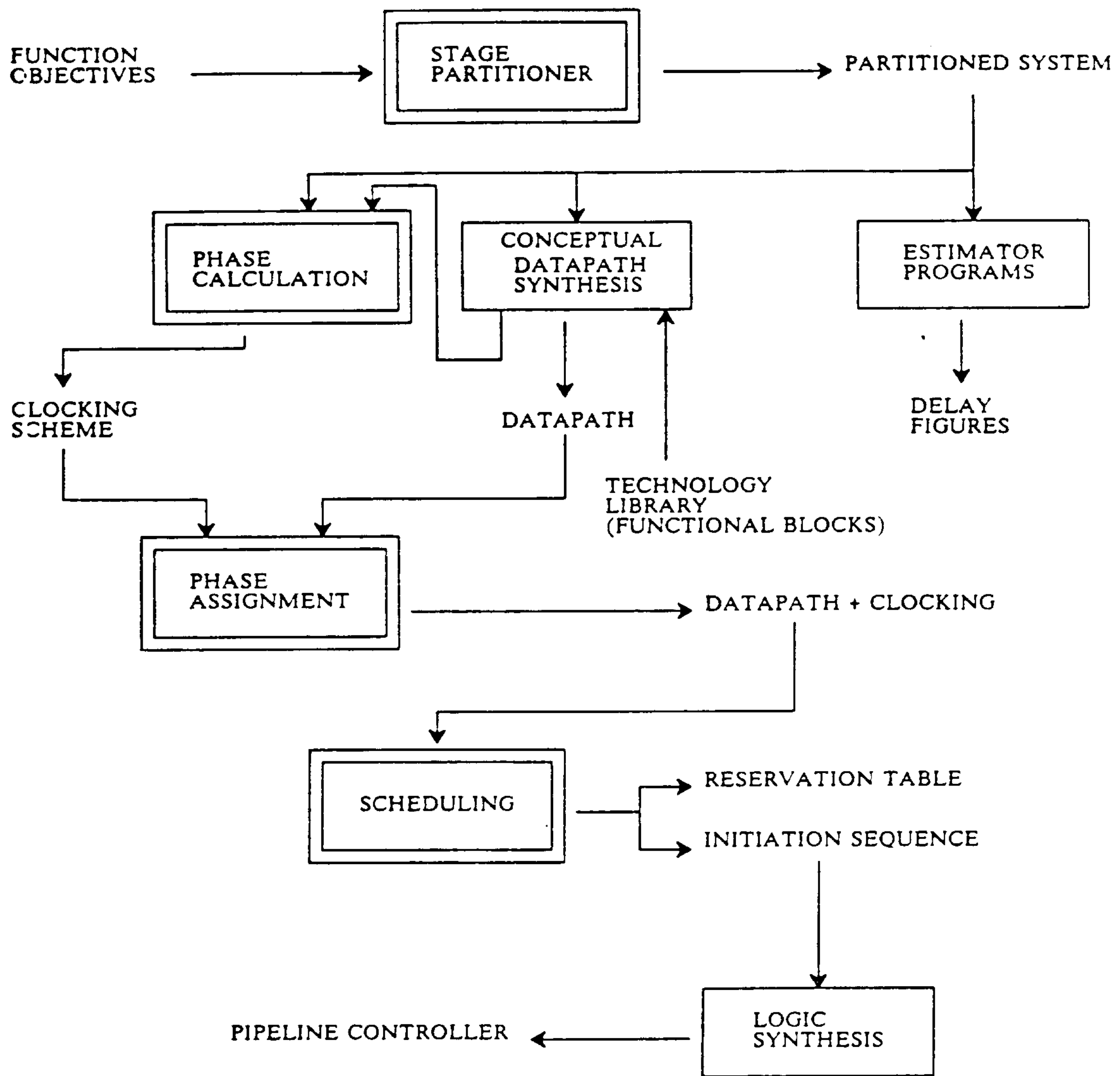
- SP: this is the Stage Partitioning tool. Fig.3 shows an example of what SP does. The input and output files for this example are in Appendix 1.

  The input to SP consists of three items. The first is a set of dataflow graphs [Snow 78] [Park 85] [Parker 85] that describe each one of the functions the system is to execute (in the case of a CPU, each graph might correspond to one instruction). The nodes of these graphs correspond to simple, atomic operations (such as an ALU operation), and each node has an attached delay. Arcs show where each operation gets its arguments from, and where it sends its results to; each arc has an attached bit-width. A probability of occurrence is attached to each dataflow graph that shows how frequently that particular function will be executed. The second input item is a description of shared resources and resource constraints. The third is the target pipeline cycle time. Given these, SP will show where the staging latches should be placed in order to achieve the target stage length and, as a secondary objective, to minimize the number of bits of staging latches.

- PC: this is the Phase Calculation tool. Fig.4 shows an example of what PC does.

  PC takes three input items. The first is the partitioned set of graphs from SP along with

FIG 2: OPD AND OTHER TOOLS



LEGEND

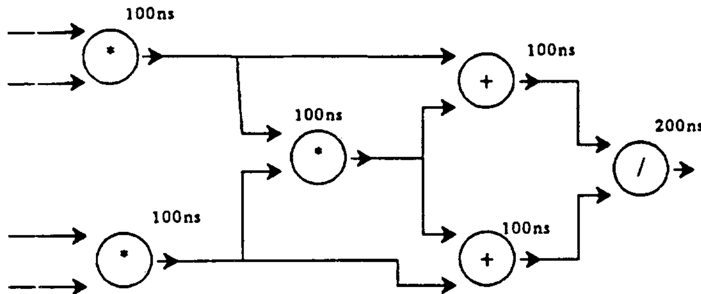
DATAPATH — DESIGN REPRESENTATION

→ — FLOW OF DESIGN INFORMATION

SCHEDULING (double box) — DESIGN TOOL PROVIDED BY OPD

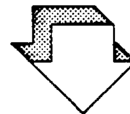
LOGIC SYNTHESIS (single box) — OTHER DESIGN TOOL

FIG 3: AN SP ILLUSTRATION



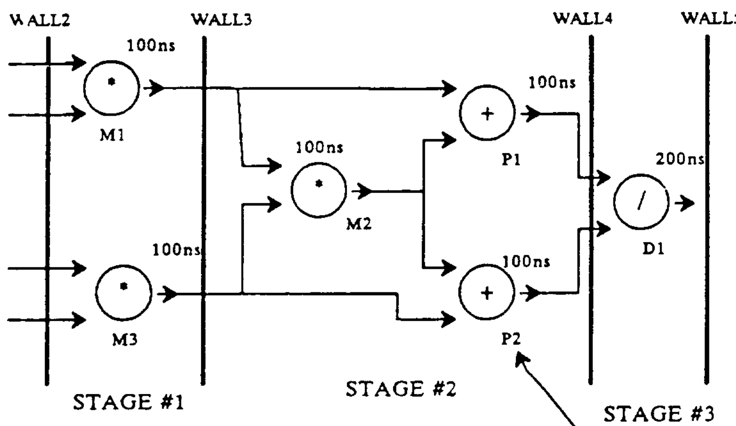
SYSTEM DATAFLOW GRAPH

THROUGHPUT:
ONE DATUM PER 400ns



SP

--NOTE: WALL = STAGE LATCH
WALL NAMES REFER TO FILE IN APPX 1



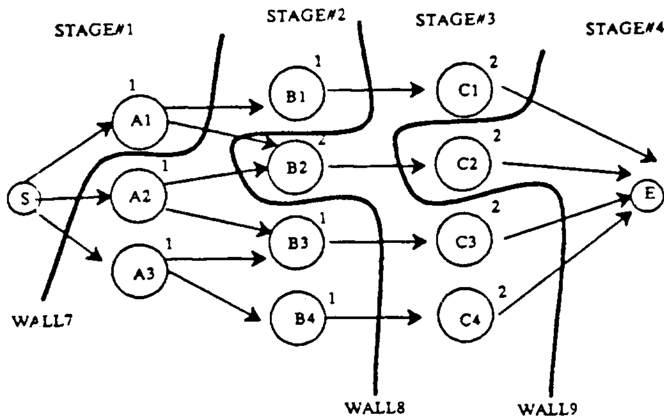
PARTITIONED DATAFLOW GRAPH

THROUGHPUT:
ONE DATUM PER 200ns

--NOTE: NODE NAMES REFER TO FILES IN APPX 1

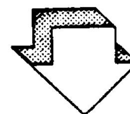
(Actual Files in APPENDIX 1)

FIG 4: A PC ILLUSTRATION

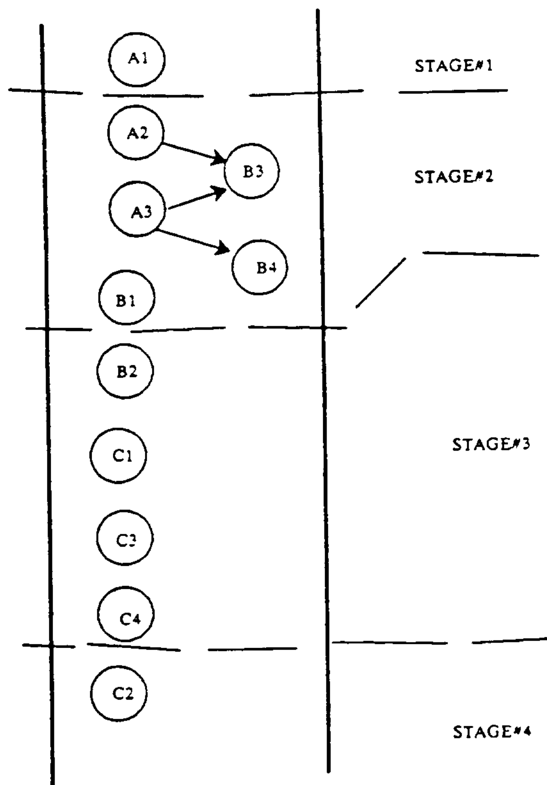


--NOTE: NODE NAMES REFER TO FILES IN APPX 2
NODE DELAYS SAME AS IN APPX 2

--NOTE: A WALL IS A STAGE LATCH
(CF APPX 2)



PC



(ACTUAL FILES IN APPENDIX 2)

the longest stage time (cycle time). The second is the target phase length. The third is a description of the datapath that will implement these dataflow graphs (provided by the designer) along with all the datapath-level constraints to suit the target technology; PC will then split up the cycle into a set of phases whose length will be as close as possible to the target phase length, and such that all the datapath constraints can be met and the longest stage time is minimized. PC therefore breaks up the clock cycle chosen by SP into a set of phases that are well suited to the final datapath. The datapath is described as a DAG; the nodes correspond to hardware operators, and the arcs to signal nets. There is also a set of constraints associated with the datapath. The purpose of these constraints is to capture technological limitations on acceptable phase sets. Such limitations might arise from particular layout or clocking schemes. The datapath specification is described fully in Appendix 2.

- PA: this is the Phase Assignment tool. Fig.5 shows an example of what PA does. PA determines on which phase/cycle each one of the datapath gates and latches should be clocked in order to satisfy the target technology constraints and to maximize pipe throughput.

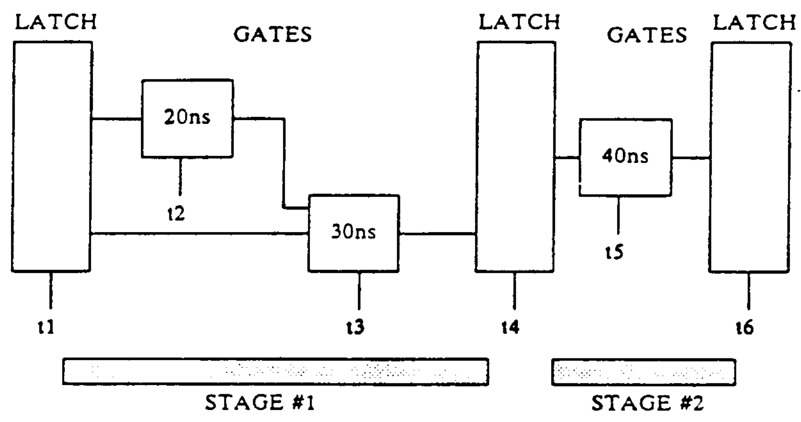
  The input to PA consists of four items. The first is a description of datapath blocks used to implement the system. The second is a set of constraints that the resulting Phase Assignment has to follow in order to be suitable for the target technology. The third and fourth items are the partitioning from SP and phase lengths from PC. PA will then determine on which phase/cycle each one of the datapath blocks should be clocked. PA uses a more precise optimization algorithm than PC. PA produces a reservation table that shows exactly how long each stage will be in the final design.

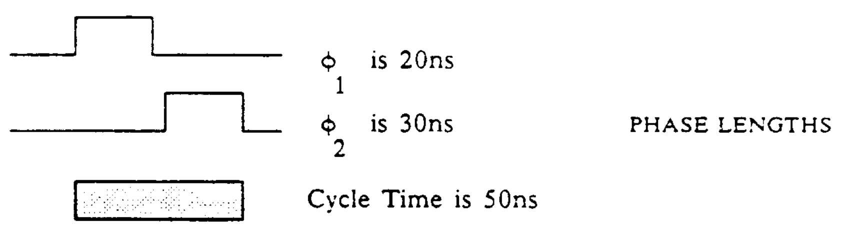
- SCHED: The reservation table scheduler. Fig.6 shows an example of what SCHED does. The input to SCHED is the reservation table produces by PA. SCHED will then determine when new data should be allowed to enter the pipeline in order to maximize the overall processing throughput. The output from SCHED is an optimal initiation

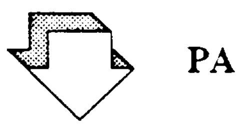
FIG 5: PA ILLUSTRATION

INPUT TO PA

constraint:
(t4, t6 must be same phase)







OUTPUT FROM PA

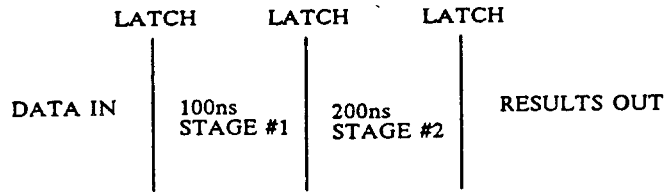
t1 = cycle0.phase2
t2 = c1.p1
t3 = c1.p2
t4 = c2.p1
t5 = c2.p2
t6 = c3.p1

longest stage = 1 cycle = 50ns

Reservation Table =

STAGE #1	X	
STAGE #2		X

FIG 6: SCHED ILLUSTRATION



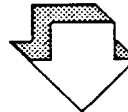
DATAPATH

CYCLE TIME = 100ns

STAGE #1	X		
STAGE #2		X	X

RESERVATION TABLE

FROM PA

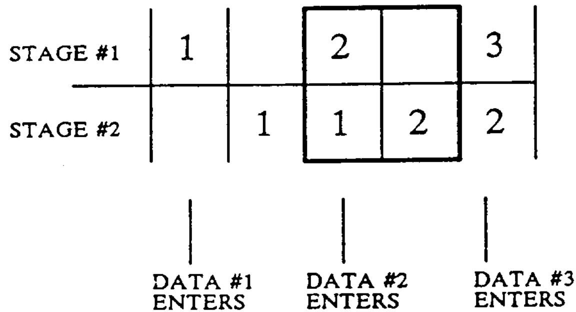


SCHED

DATA ENTERS AT TIMES

T = CYCLE 0, CYCLE 2, CYCLE 4...

INITIATION SCHEDULE



STEADY-STATE

STAGE USAGE PATTERN

sequence [Kogge 81] for the pipe.

The output from SCHED fully defines when each control signal will have to be activated; we can feed this information to logic synthesis routines such as MIS [Brayton 86], ESPRESSO [Brayton 83] or ESPRESSO-ML [Brayton 84] or [Leive 81] to generate the pipeline controller automatically. We will show some examples of how this can be done semi-automatically for two pipeline control strategies. However, we have not built a tool to generate the pipeline controller in a fully automatic fashion. This is because there are many different pipeline control styles.

During the initial design stages, we can profitably use estimator functions [Kurdahi 85] to get approximate delay figures for each node in the behavioral graphs. These estimators use factors such as bit-width and type of operation (integer versus floating point, for instance) to provide delay estimates that can be used for comparative purposes.

It was decided not to include automatic datapath synthesis into OPD. Automatic module selection for datapath design is a very complex task; most datapath synthesis routines [Park 85] [Park 85b] [Parker 85] [Thomas 83] work mainly with the approximate information available in the system's behavioral description. These routines therefore have difficulty taking into account all the constraints that result from target technology details. As a result, most such programs produce technology-independent datapaths that can be very far from optimal when they are mapped into a particular target technology. This does not fit with OPD's "toolbox" approach; we take the "vertical" route; OPD has tools that remain useful down to the layout level.

In summary, OPD is a sequence of tools that perform progressively more detailed optimization tasks on pipelines. The higher-level tool SP has a very simplistic view of the system; the later tools (PA, SCHED) take into account more detailed information. Of course, these various optimization tasks are inter-dependent. For instance, we want to do SP such that when the datapath is finally scheduled using SCHED, the throughput will be maximal.

To handle these inter-dependencies, each tool has a simplified view of what the next tool will do; it uses this simplified view to perform optimization at its own level. For example, SP hopes that, by minimizing the longest stage length, PC will be able to find phases that lead to a faster pipe. In fact, the cost function for SP could be a simplified version of PC. The cost function for PC is actually a simplified version of PA. Of course, SCHED knows the exact, final pipe speed.

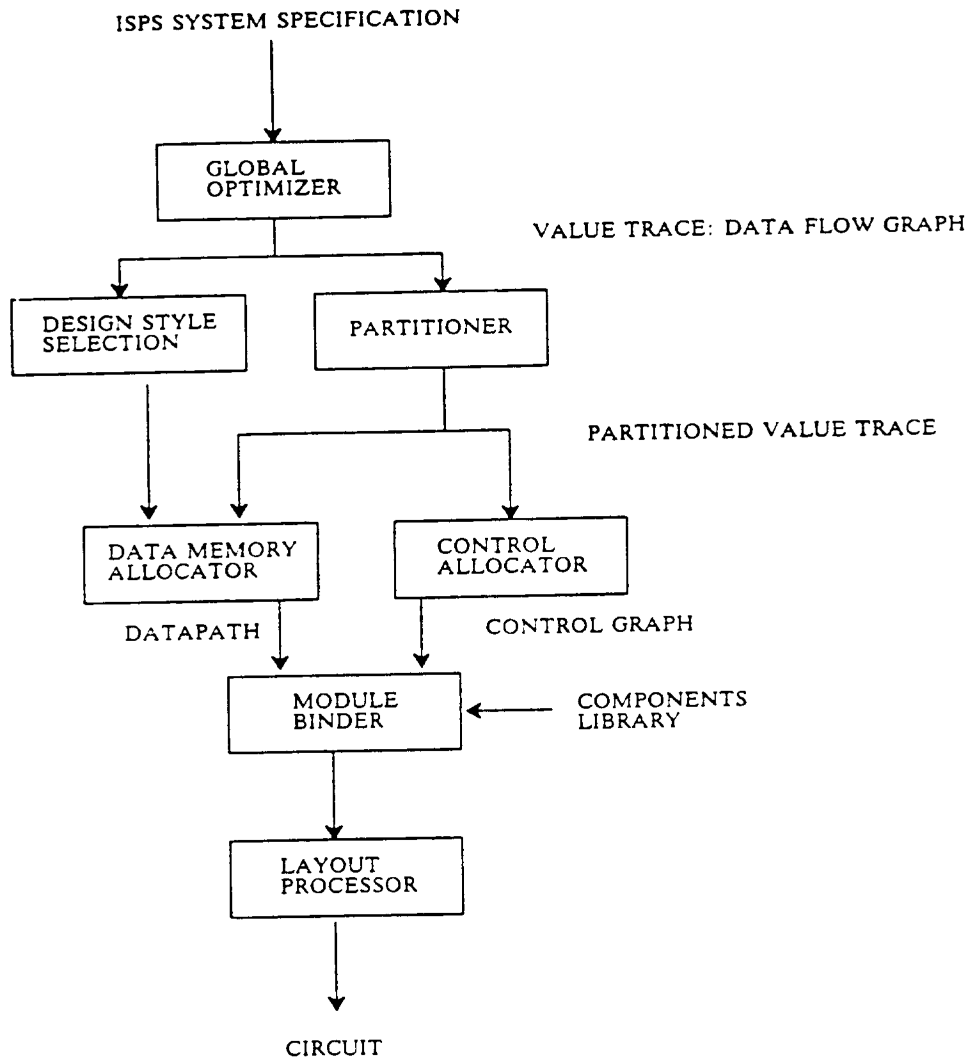
3. OPD Context and Related Tools

We show how OPD relates to and differs from similar tools. We are mainly concerned with datapath synthesis systems and clocking scheme synthesizers.

Datapath synthesizers take a behavioral description of a system, perhaps augmented by performance and cost constraints, and produce a register-transfer level structure that implements this behavior. [Blackburn 85] provides an overview of one such set of tools, the CMU Design Automation System. A datapath synthesizer is also described in [Parker 85]. Fig.7, from [Snow 78], shows the strategy used by the CMU/DA. OPD and the datapath synthesizers are complementary tools. OPD can use a synthesizer to map the partitioned stage graph into a register-transfer level datapath. From then on, OPD could perform phase length calculation and phase assignment using the synthesized datapath. A possible pitfall of this approach is that the synthesis program needs to know quite a lot about the target technology and the pipeline structure to produce a good datapath. The pipelining and datapath synthesis tasks are inter-dependent.

Clocking scheme synthesizers, such as [Park 85] and [Park 85b], decide how to pipeline the datapath and calculate the optimal number of phases per clock cycle, as well as the length of each phase. However, these tools work at a fairly high level of abstraction, where the details of the final datapath are still unknown. As a result, the phases produced by these tools tend to match the structure of the pipeline stages. This is not what we want in the case of a MOS-VLSI pipe. We want the phases to match the interconnection of the gates and the precharging and bussing schemes. Fig.8 and Fig.9 show a phase calculation

FIG 7: CMU / DA OVERVIEW



(TAKEN FROM [SNOW 78])

found in [Park 85]. Each one of the two graphs in Fig.8 represents one instruction that the system, an HP21-MX CPU, is able to execute. Each phase corresponds to one stage latch; phases serve to clock pipe stages. Fig.10 shows what OPD means by a phase; a phase is used to clock gates within a stage, as is commonly done in MOS designs. OPD needs detailed datapath information, in the form of constraints on the possible phase assignments, in order to calculate these MOS-phases.

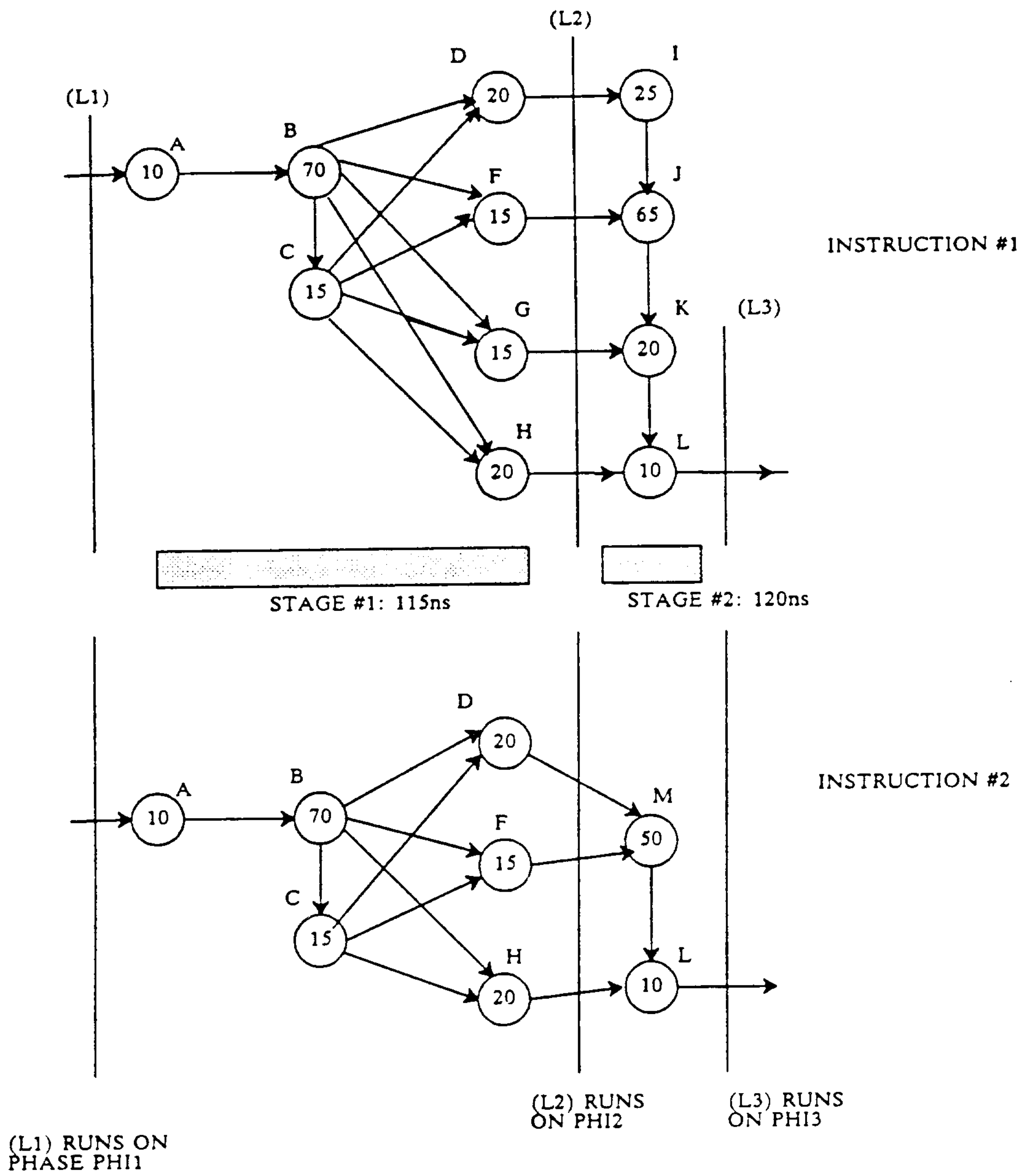
4. Software Organization

The software is organized as a set of independent programs which communicate via shared files. The table below gives some statistics. The more time-critical routines are in the language C [Kernighan 78], while Franz LISP [Franz 86] is used everywhere else.

Function	Language	approx #lines (excluding test code)
Stage Partition and Phase Calculation	Franz LISP	2000
Preprocessing for Phase Assignment	Franz LISP	1500
Phase Assignment Heuristic & Annealing	C	3000
Reserv. table Scheduler	C	700

The next chapter describes the Stage Partitioning and Phase Calculation tools.

FIG 8: HP21-MX CPU DATA FLOW GRAPH

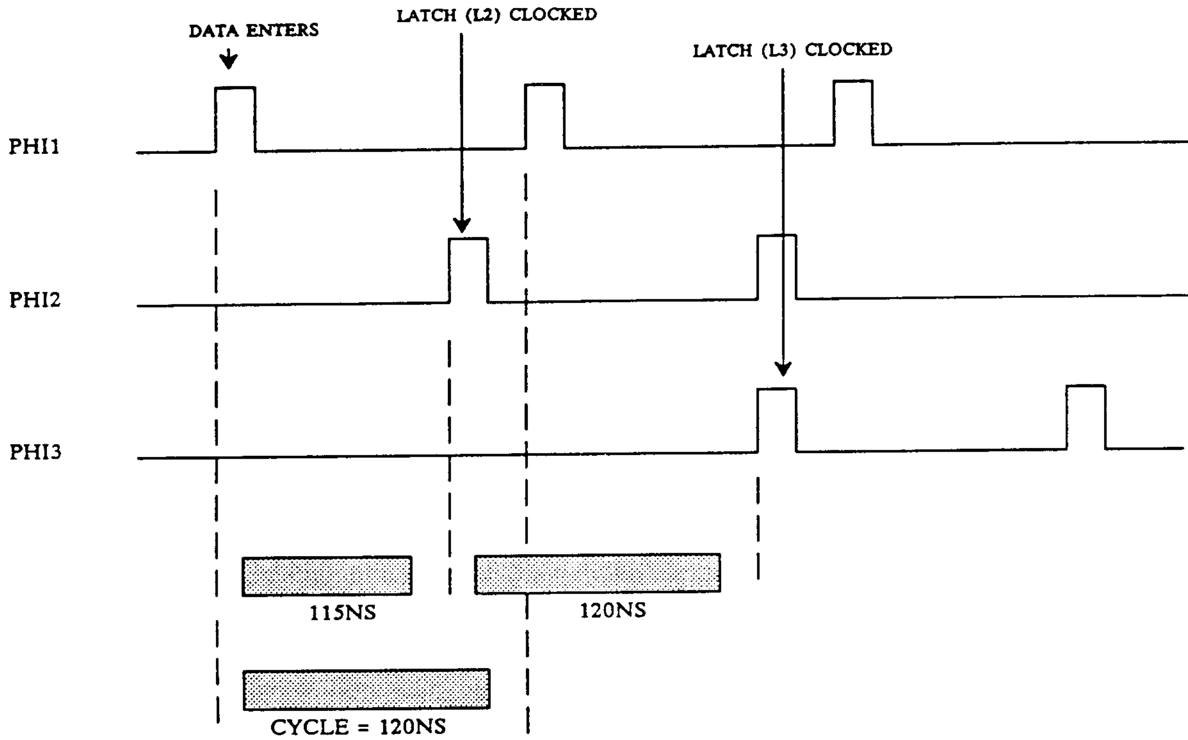


HP21-MX GRAPHS FROM [PARK 85]

ACTUAL FILES IN APPENDIX 1

NOTE: THE NUMBERS ARE NODE DELAYS

FIG 9: HP21-MX PHASES ACCORDING TO [PARK 85]



PHI1: CLOCKS DATA INTO (L1)

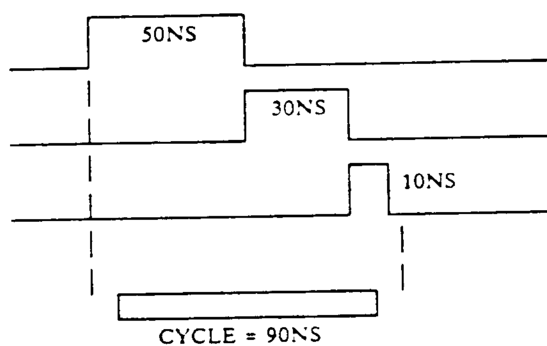
PHI2: CLOCKS DATA AT THE END OF 1ST STAGE

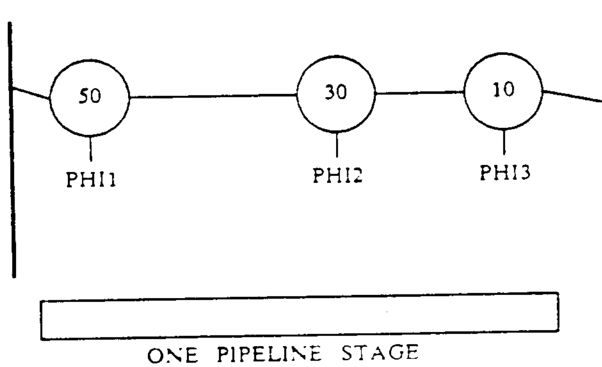
PHI3: CLOCKS DATA AT THE END OF 2ND STAGE

PHI(I) TO PHI(I+1) IS THE LENGTH OF STAGE #(I)

PHASES SERVE TO SEQUENCE STAGE LATCHES IN [PARK 85]

FIG 10: MOS PHASES IN OPD





MOS PHASES IN OPD ARE USED TO SEQUENCE GATES -- NOT STAGE LATCHES

Section II - Stage Partitioning and Phase Calculation

In this chapter, we define the Optimal Stage Partitioning problem (SP), as well as the Optimal Phase Calculation (PC) problem. We use the Microcode Compaction Problem [Fisher 81] to show that SP is NP-complete. We describe the heuristic algorithms used to perform the SP task, and finally show how the same algorithms can be used to perform PC. We also present other possible approaches for solving SP.

1. Problem Specification for SP

1.1. Input

The input to SP consists of a set of dataflow graphs ("traces") with an attached probability of occurrence, plus a set of resource constraints and a target stage length. Each graph represents one of the operations or instructions that the system is to execute. The attached probability represents how often that particular operation is expected to occur. For instance, in a CPU, there might be one graph for load/store instructions, another for arithmetic, and another for branches. The attached probabilities are the relative dynamic instruction frequencies. Specifically, the input consists of the following elements.

$N_i (i = 1,..,NN)$ — A set of NN dataflow nodes. A node corresponds to an operation; a node can optionally have an attached resource-type RT_j used to specify resource constraints. For instance, in a CPU, the dataflow node corresponding to arithmetic instructions would have ALU as an associated resource type. This models the fact that arithmetic instructions require the ALU (or one of the ALUs if there are many).

$RT_j (j = 1,..,NRT)$ — A set of resource-types; each resource-type has an associated number called the "limit". A resource-type corresponds to a class of hardware units; the "limit" is the number of available units of that type. For instance, we may have a resource-type "ADDER" with a "limit" of 2. This means that not more than 2 "ADD" dataflow nodes can be simultaneously scheduled.

$T_k (k = 1,..,NT)$ — A list of "traces". Each trace is a directed acyclic dataflow graph on a subset of the above set of nodes. Each arc of each trace has an associated bit-width. Each trace corresponds to one of the operations that the system will have to execute.

$PT_k (k = 1,..,NT)$ — A probability figure attached to each trace. PT_k is the probability, or relative frequency, with which the system will be expected to execute the operation corresponding to T_k.

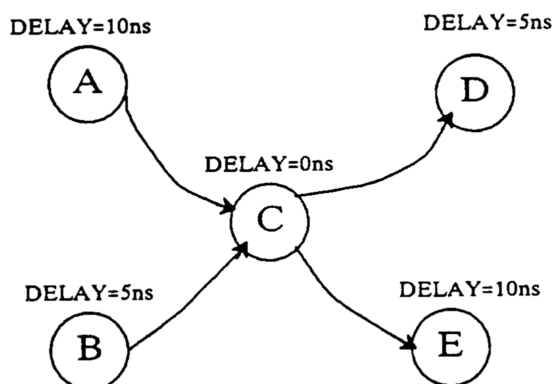
$TARG$ — The target stage length

Traditional Data Flow Graph input descriptions need to have a way to describe that certain operations are mutually exclusive in the system. Fig.10.bis shows the problem; if we did not specify that nodes A and E are never simultaneously active, the stage partitioner would (falsely) deduce that the critical path in the system is A-C-E. This path can never occur. Our system description avoids this problem altogether since we only specify those dataflow graphs that corresponds to actual, possible instructions. For the example of Fig.10.bis, we would describe the system by giving two dataflow graphs, each one corresponding to a real operation. The first graph would be A-C-D, the second B-C-E. OPD will trace these graphs independently, and avoid the false path A-C-E altogether. We therefore do not need a special dataflow node to describe mutual exclusivity in OPD.

1.2. Optimization Objective

The objective is to partition the given set of nodes into consecutive pipeline stages such that the length of each stage is minimal and less than $TARG$ and, secondarily, so as to minimize the sum of the bit-widths of the arcs that are cut by stage latches. This

FIG 10.BIS: MUTUAL EXCLUSION



OPERATIONS A AND E
ARE MUTUALLY EXCLUSIVE

THE PATH | A ➞ C ➞ E | (20ns)

IS A "FALSE" CRITICAL PATH - IT CAN NOT OCCUR

| A ➞ C ➞ D | AND | B ➞ C ➞ E |

ARE THE ACTUAL CRITICAL PATHS.

THEY ARE 15ns LONG.

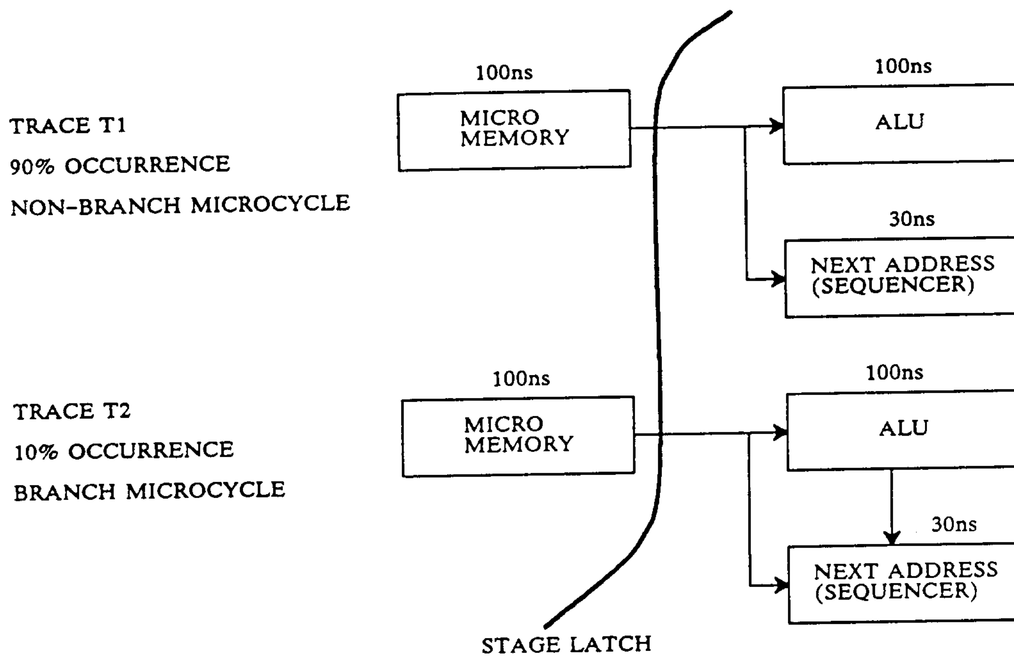
minimizes the number of stage latch bits. In fact, SP first fixes the number of stages, then moves the stage latches around (described below).

The cost function we use is equal to the length of the longest stage plus a small factor times the total number of stage latch bits. The length of each stage is determined by scheduling the nodes that are in that stage so as to respect the precedence constraints specified by the "traces" and so as to respect the resource limit constraints specified by the RT_k.

We also take into account resource sharing between nodes in different stages. For instance, assume two stages S1 and S2 both have ADD operations; S1 has $N1$ ADDs, while S2 has $N2$ ADDs. Furthermore, assume there are only *NADDER* "ADDERs" available in the system. We would then set the stage lengths for both S1 and S2 to be greater than or equal to $TMAX = \frac{(N1+N2)}{NADDER}$. This is because *TMAX* is the time it will take to get data through both these stages; the possibility to overlap S1 and S2 will be reduced due to this resource sharing. Our calculation does assume, however, that some overlap between S1 and S2 will occur so as to keep the shared resources busy all the time. In other words, we take the best-case impact of resource sharing on the stage length term: we assume that the shared resources will always be kept busy.

The above cost function makes the slowest trace determine the cycle time. For instance, in a CPU, this means that the slowest instruction determines the machinés cycle time. Another possible choice is to calculate the longest stage time for each "tracé'. We could then use the weighted average of these per-trace longest times as the çost. Fig.11 shows the difference between these two cost functions. In order to benefit from the possibility of having different cycle times for each trace, we need a controller which can dynamically choose the proper cycle time according to the operation the system is executing. Examples of such machines are the PDP11/34 and the PDP11/40. These machines had micro-engines that could choose between two or three possible micro-cycle times ([SBN 82] ch. 34) in a dynamic fashion, according to the particular micro-instruction being executed.

FIG 11: WORST-CASE VERSUS PER-TRACE CYCLE TIMES



SINGLE CYCLE TIME FOR ALL MICRO-INSTRUCTIONS: CYCLE = 130ns

VARIABLE CYCLE TIME: CYCLE(T1) = 100ns
CYCLE(T2) = 130ns
AVERAGE = 0.9*100 + 0.1*130
AVERAGE CYCLE TIME = 103ns << 130ns !!

THE VARIABLE CYCLE TIME LEADS TO INCREASED PROCESSING THROUGHPUT

However, OPD uses the simpler worst-case stage length cost.

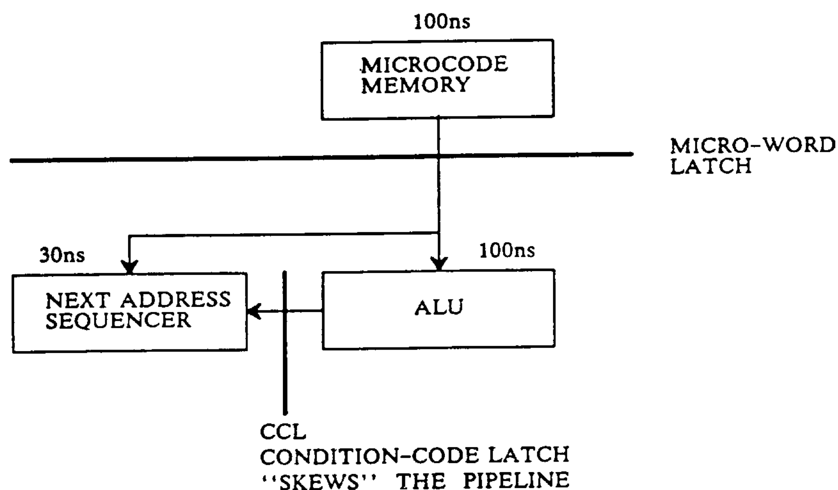
The output from SP is therefore a list of stage latches that completely partition each one of the input traces into disjoint stages. Each latch therefore consists of a cut-set of arcs for each input trace. Appendix 1 describes the input and output format used by SP, and shows two examples: a small test example, and the HP21-MX CPU graph taken from [Park 85].

1.3. Skewed Stages

SP also has the capability of inserting stage-latches that do not form cut-sets for all the input graphs. This feature is optional, since inserting such latches will lead to a "skewed" pipeline. In such a pipeline, the total number of clock cycle required for a data item to travel through all the stages depends on the data item - it may require one or more extra cycles if it hits the non-cutset latches. Fig.12 shows a "skewed" pipeline example inspired from a very common micro-engine design. Micro-engines (and RISC CPUs) are often skewed pipelines; that is, some data takes longer than others to filter through the engine. For instance, a branch based on condition codes will take two micro-cycles to get executed; a normal ALU operation will take only one. Any system that supports a delayed branch is, in effect, a skewed pipeline.

Skewed pipes are useful when we want to have a small number of stages (to make it easier to handle data dependencies for example) and at the same time keep each stage very short. The penalty we pay is that "rare' operations will execute with an extra delay (like branches). In the case of Fig.12, we do not want to have more than two stages (one pipe latch) because of data dependencies. The un-skewed pipe has a cycle time of 130ns; the skewed version will execute instructions in an "average' of 110ns (the usual case executes in 100ns; branches require 200ns but only happen 10% of the time). Of course, if conditional branches were more common, the skewed pipe would be the wrong choice. It is better to a have a single-cycle conditional branch if there are many of them, even if this means increasing the cycle time slightly.

FIG 12: A SKEWED PIPELINE



ASSUME 10% CONDITIONAL BRANCHES

NO CCL: CYCLE = 130ns; 1-CYCLE BRANCH

WITH CCL:: CYCLE = 100ns

2-CYCLE BRANCH (200ns)

AVERAGE INSTRUCTION EXECUTION TIME:

0.9*100+0.1*200 = 110ns

THIS IS SHORTER THAN 130ns

FOR ANOTHER WAY TO SEQUENCE THIS DATAPATH,

USING A SHORT CYCLE TIME (33ns),

PLEASE REFER TO TEXT

Another possibility is to sequence this skewed pipeline with a very short cycle time. Each instruction can then use the minimum number of cycles required for its execution. For the example of Fig.12, we could have a 33ns cycle time. Non-branch instructions execute in 3 cycles, while branches require four. In effect, by using a short and well chosen cycle time, we have finer control over the timing of data flow through the pipeline. In Fig.12, a cycle of 33ns was chosen because the node delays are 100ns and 30ns. A value of 33ns represents some form of greatest common divisor.

OPD does not calculate such short "GCD" cycle lengths automatically. However, once the designer has chosen a suitable cycle time, this can be fed to OPD. OPD can then break the cycle up into phases (if required), determine how many cycles each instruction will require, and produce the reservation table for this pipeline. Finally, OPD can calculate an optimal initiation sequence for this table.

The last task - calculating an optimal initiation sequence - can become quite complex with short cycle times. This is because the reservation table will spread over a great many cycles, and will have a rather irregular pattern. Short cycle times will therefore usually require more involved pipeline control logic. OPD helps the designer specify and synthesize this logic.

1.4. Related Algorithms and Problems

We show that the Microcode compaction problem (MC) [Fisher 81] is a subtask of SP. The objective of MC is to pack a sequence of elementary micro-operations into horizontal micro-instructions so as to minimize the total execution time of the sequence by exploiting the parallelism available in the micro-engine.

In order to calculate the length of a particular stage under resource limitations, SP has to pack the operations (nodes) in that stage so as to minimize the total stage time while satisfying the resource constraints. In order to calculate the length of a particular pipe stage, SP therefore has to perform MC for the operations within that stage. MC is there-

fore a subtask of SP. [Fisher 81] shows that MC is NP-complete; he does this by showing that a constrained multiprocessor scheduling problem, known to be NP-complete [Coffman 76], is reducible to MC. It then follows that MC, and therefore SP, are NP-complete. We now describe the algorithms used for SP.

2. Optimization Algorithms for SP

SP is done in three steps; the first step, **opt0**, creates a starting-point stage partitioning; the next two steps, **opt1** and **opt2**, iteratively optimize this partitioning to reduce the maximal stage length and the number of stage latch bits.

Opt0 relies on the delays of the nodes in each stage to estimate the stage length. **Opt1** and **opt2** use a more precise cost function that actually schedules the nodes in each stage to calculate the stagés length.

2.1. Cost Function

The cost function is made up of three terms: the length of the longest pipe stage, an extra term (called the *share-term*) to account for resource sharing between stages, and a small factor times the number of stage latch bits.

In order to find the length of each stage in the presence of shared resources, it is necessary to schedule the operations of that stage while satisfying the resource limits. We use forward-urgency scheduling [Park 85] for this. Whenever two independent operations compete for a shared resource, the operation that is farther from the end of the stage gets the resource first. The distance of a node from the end of the stage is the length of the longest path from that node to the output stage latch. We know how far each operation is from the end of the stage by calculating the delay from that operation's node to the stage latch. Fig.13 shows the **forward-urgency-sched** procedure along with an example.

2.2. Opt0 Seed Algorithm

Opt0 takes a target stage length as argument, and packs blocks into successive stages that

FIG 13: FORWARD-URGENCY SCHEDULING ALGORITHM

PROCEDURE FORWARD-URGENCY-SCHEDULE;

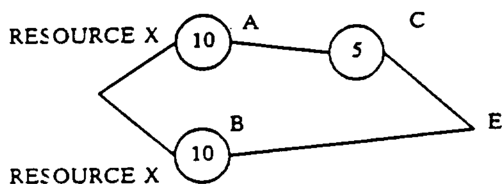
THE INPUT IS A SET OF NODES ALL LOCATED IN THE SAME STAGE;

STEP #1: CALCULATE THE FORWARD URGENCY OF EACH NODE
(THIS IS THE DISTANCE OF THE NODE TO THE STAGE END-LATCH)

STEP #2: SELECT NODES IN TH STAGE BY ORDER OF DECREASING URGENCY;
FOR EACH SELECTED NODE, SCHEDULE IT TO RUN AT THE EARLIEST TIME
COMPATIBLE WITH THE PRECEDENCE CONSTRAINTS
AND WHEN REQUIRED RESOURCES ARE AVAILABLE;

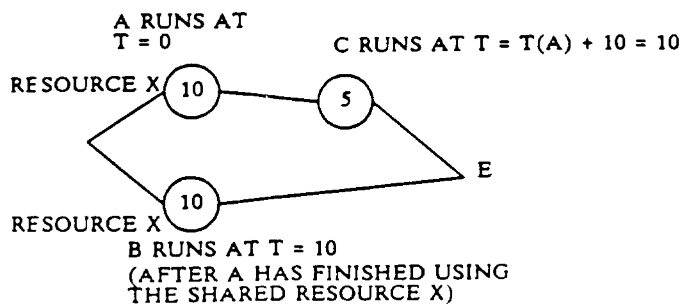
END;

EXAMPLE: THE GRAPH IS:



ASSUME
A AND B SHARE SOME RESOURCE X

A IS SCHEDULED FIRST SINCE IT IS FARTHEST AWAY FROM E.



are each shorter than the target length. **Opt0** iteratively proceeds from the graph's root nodes (those with no fanin) towards the graph's tail nodes (those with no fanout), creating a new stage whenever the target length is reached. Fig.14 shows how **opt0** works.

2.3. Opt1 Optimization

Opt1 takes an existing stage partition and attempts to iteratively improve it; it does so by trying to move single nodes from one stage to the next or to the previous stage. After trying to move every boundary node, the best resulting partition is picked. Fig.15 shows an example of **opt1**.

2.4. Opt2 Optimization

Opt2 also takes an existing stage partition which it iteratively improves; it does so by an algorithm based on Kernighan and Lin's bipartition exchange algorithm [Kernighan 70]. **Opt2** will successively move every operation node N_1 through N_k, located at a stage boundary, to the next stage. it will then pick the best point $k=k0$ such that the stage length reached by moving N_1 through N_{k0} is minimal with respect to k. **Opt2** then repeats this procedure, moving boundary nodes to the previous stage this time. Fig.16 shows an example of **opt2**. [Devadas 87] shows another application of the Kernighan and Lin algorithm to synthesis problems.

The reason we have three algorithms is as follows. **Opt0** finds a starting configuration. **Opt1** performs optimization with respect to local motion of nodes across stage boundaries. As such, **opt1** typically gets stuck in configurations that correspond to local minima for the cost function. On the other hand, **opt2** attempts more drastic changes to the pipeline structure. **Opt2** has the potential to get the search out of the kind of local minima that block **opt1**.

Therefore, we would normally call **opt0** first. We would then iteratively call **opt1** until a local minimum is found. At this point, we call **opt2** to escape from the local minimum, and, hopefully, find a more promising configuration. We iterate **opt1** on the new

FIG 14: OPT0 OPTIMIZATION ALGORITHM

PROCEDURE OPT0;

INPUT: TARG (TARGET STAGE LENGTH) + G (GRAPH TO PARTITION)

OPT0 PARTITIONS G INTO A SET OF CONSECUTIVE STAGES;
THE LENGTH OF EACH STAGE IS <= TARG.

OPT0 STARTS OUT AT THE ROOT NODES (THOSE WITH NO FANIN) AND WORKS ITS WAY TO THE TAIL NODES (THOSE WITH NO FANOUT). ALONG THE WAY, OPT0 CREATES NEW STAGES AS FOLLOWS.

OPT0 STARTS OUT WITH ONE (IMAGINARY) STAGE LATCH LOCATED JUST BEFORE THE ROOT NODES.

GIVEN A STAGE LATCH, OPT0 CREATES A NEW STAGE AS FOLLOWS.
FIRST, OPT0 WILL TRACE ALL THE PATHS FROM THE GIVEN LATCH TOWARDS THE TAIL NODES. OPT0 WILL GO AS FAR AS POSSIBLE FROM THE GIVEN LATCH, SUCH THAT THE DISTANCE ALONG EACH PATH TO THIS LATCH IS <= TARG.
SECOND, OPT0 WILL CREATE A NEW STAGE LATCH WHICH CUTS THE GRAPH AT THE EXTREMETIES OF THE TRACED PATHS.

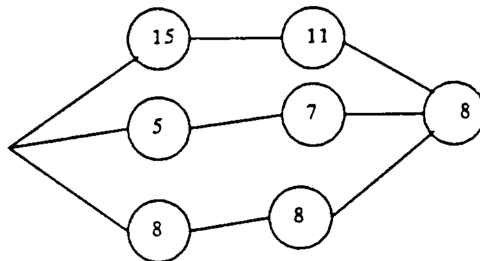
THE NODES BETWEEN THE GIVEN AND THE NEW STAGE LATCHES FORM A NEW MAXIMAL STAGE WHOSE LENGTH IS <= TARG.

OPT0 THEN REPEATS THIS PROCEDURE, STARTING FROM THE NEW STAGE LATCH.

OPT0 STOPS WHEN THE TAIL NODES HAVE BEEN REACHED.

END;

EXAMPLE:



NOTE: NUMBERS ARE DELAYS
ALL PARALLEL PATHS ARE ACTIVE

OPT0 WITH TARG = 15 GIVES:

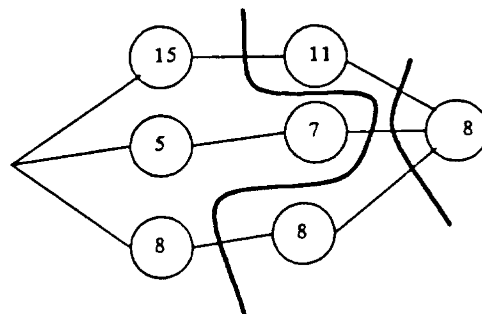
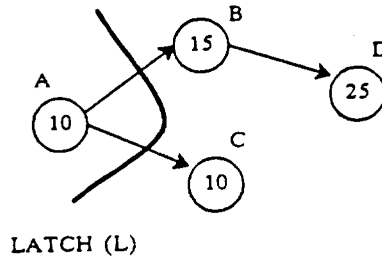


FIG 15: OPT1 EXAMPLE

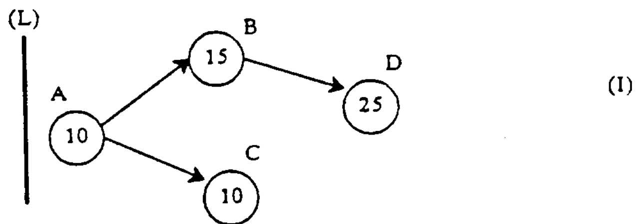
ORIGINAL PARTITION
(O.P.)
CYCLE TIME = 40



LATCH (L)

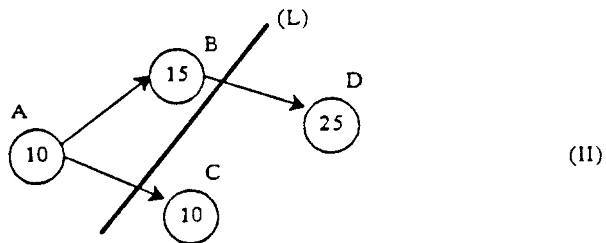
OPT1 WILL TRY MOVING A,B,C TO THE OTHER SIDE OF (L)
OPT1 STARTS FROM O.P. FOR EACH MOVE:

MOVE A =>
CYCLE TIME = 50



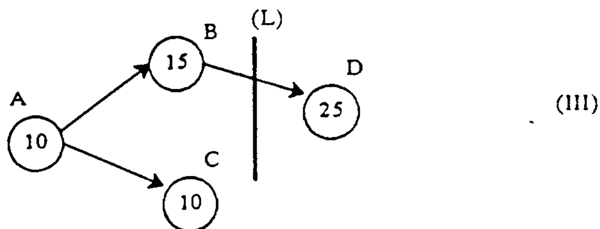
(I)

MOVE B =>
CYCLE TIME = 35



(II)

MOVE C =>
CYCLE TIME = 25

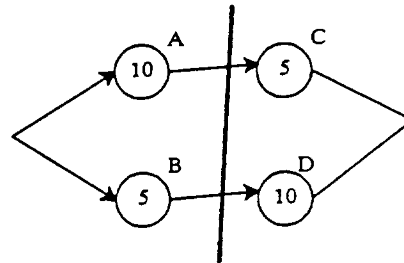


(III)

OPT1 WILL THEREFORE PICK (II), AFTER MOVING B.
OPT1 WILL ITERATE THESE MOVES ONCE MORE, BUT WILL FIND NO IMPROVEMENT IN THE CYCLE TIME ((II) IS OPTIMAL). OPT1 WILL THEN STOP.

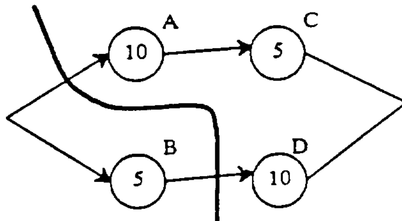
FIG 16: OPT2 EXAMPLE

ORIGINAL PARTITIONING
CYCLE TIME = 10

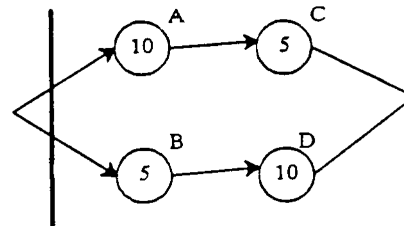


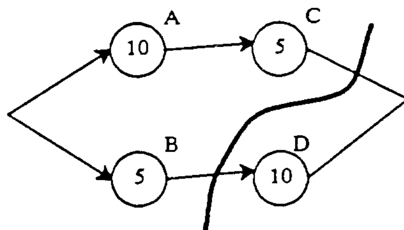
OPT2 WILL TRY THE FOLLOWING CONFIGURATIONS AND PICK THE BEST:

CYCLE TIME = 15

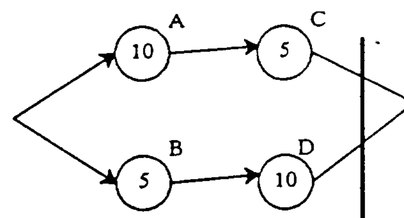


CYCLE TIME = 15





CYCLE TIME = 15



CYCLE TIME = 15

IN THIS EXAMPLE, THE ORIGINAL CONFIGURATION YIELDS THE BEST CYCLE TIME.

configuration until a new local minimum is found, call **opt2** on it, and so on. We stop when consecutive passes of **opt2** followed by iterations of **opt1** yield no improvement in the cost function. Appendix 1 shows how **opt1** and **opt2** are used in sequence. We see that **opt2** helps the search escape from **opt1**'s local minima.

2.5. Skewed Pipe Generation

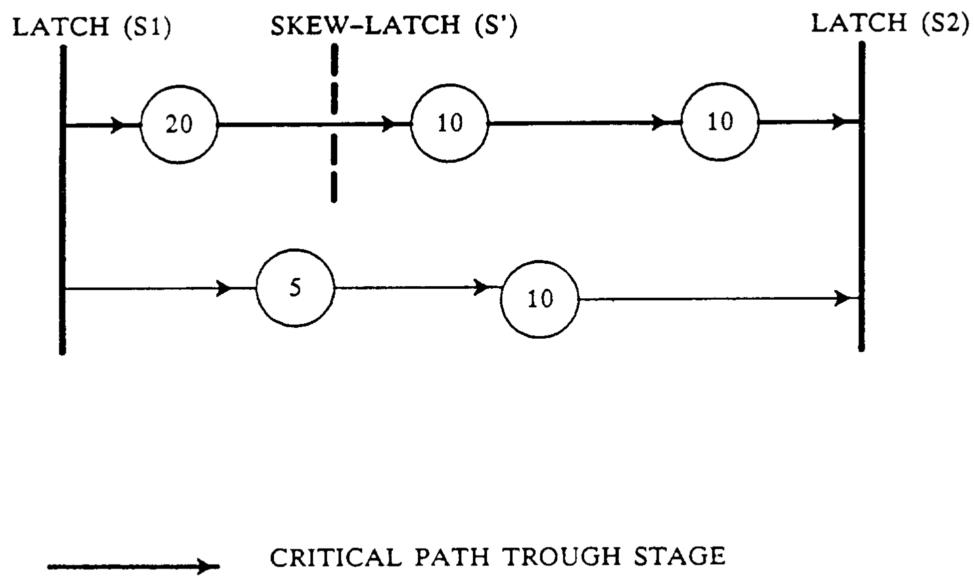
Skewed pipes are generated in two steps: we first pick a (regular) stage from latch S1 to latch S2 that is a good candidate for "skewing"; this is typically the longest stage. Next, we "skew" this stage by inserting a latch S' about half-way on the longest path of this stage. As Fig.17 shows, we then have a skewed stage such that the lengths of the three new component skewed stages (S1 to S') and (S' to S2) and (S1 to S2 with S') are all strictly less than that of the original stage (S1 to S2). As we mentioned, this is optional and especially useful to avoid long but rare operations (like branches) from slowing down frequent ones (like loads/stores). We now describe how the algorithms designed for SP can be used to determine a set of phase lengths for sequencing the actual datapath.

3. The Phase Calculation Problem

3.1. Problem Specification For PC

The objective of PC is to break up the cycle into a set of non-overlapping phases such that these phases can be efficiently used to sequence the datapath gates and latches. The input to PC consists of the following items.

FIG 17: SKEWED PIPE EXAMPLE



CRITICAL PATH TROUGH STAGE

WITHOUT SKEW-LATCH (S'): CYCLE TIME = 40

WITH (S'): CYCLE TIME = 20

... SEE ALSO: FIG.12

$S_i(i=1,..,NST)$ — The stage latches that correspond to the best partition found by SP

MINIPHIS — The minimum number of phases in a cycle

TARGPHI — The target length for each phase

DPATH — A description of the datapath used to implement the system along with the datapath's associated constraints. The format for DPATH is that required by the Phase Assignment step PA. The task of finding a suitable datapath is left up to the designer. OPD does not perform datapath synthesis.

PC then calculates a set of at least *MINIPHIS* phases such that the length of each phase is close to *TARGPHI*, and such that when these phases are used to "sequencé' the above datapath, the length of the longest pipe stage in minimized. The Phase Assignment routine, PA, decides how a set of phases is used to sequence the given datapath.

3.2. Using SP To Solve PC

We notice that PC and SP are very similar; the objective of SP is to break up a dataflow graph into stages; that of PC is to break up the cycle time, which corresponds to the length of one pipe stage, into phases. The cost function for SP is the length of the longest pipe stage; that of PC is the length of the longest stage after PA has calculated how to best use the given phase lengths to sequence the actual datapath.

Because of this similarity, we re-use SP to solve PC. To map PC to SP, we first "fold" our dataflow graph. This step involves breaking the dataflow graph into the separate pipe stages found by SP; these stages are then placed in parallel to form the folded graph. The folded graph therefore has only one pipe stage, and the length of that stage is equal to the cycle time found by SP.

We now run SP on PC, with a target stage length (an input of SP) equal to the target phase length we want. SP will then break up the folded graph into stages, where the length of each stage is close to our target phase length. By calling **opt0**, **opt1** and **opt2** on the folded graph, SP will generate a range of possible phase lengths. The cost function we use to evaluate a phase length sequence is the datapath's pipe throughput, as calculated by

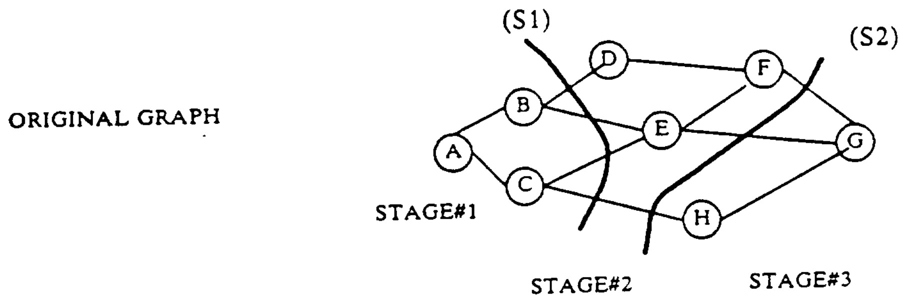
the Phase Assignment routine after it actually sequences the datapath. This procedure requires that the datapath be re-sequenced by PA every time SP generates a new set of phase lengths. Fig.18 shows the procedure.

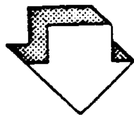
In effect, what we have done is to use the structure of the dataflow graph to generate a good set of phase length sequences via SP on the folded graph. We try out every sequence of phase lengths on the actual datapath via PA and pick the phase lengths that lead to the best results.

The software has routines to fold the dataflow graph and automatically run PA; Appendix 3 shows PC at work on a CPU example derived from the RISC-II [Katevenis 83]. As we see from this example, it is not easy to find a good set of phase lengths solely from the dataflow or even the datapath graphs. One reason for this is that only certain phase length patterns may be suitable. For instance, the designer might want to have equal length phases. This is indeed the case with the RISC-II [Katevenis 83] [Sherburne 84]. Appendix 3 shows that, within this constraint, the phase length chosen by the original designers (120ns) is very close to the optimum. However, the sample runs in Appendix 3 show clearly that when phases are used to sequence detailed MOS gates, there is no simple relationship between stage length and phase lengths. For example, slow gates may run over many phases, while many fast static gates could fit into the same phase. The relationship between dataflow stages and phases is therefore very tenuous.

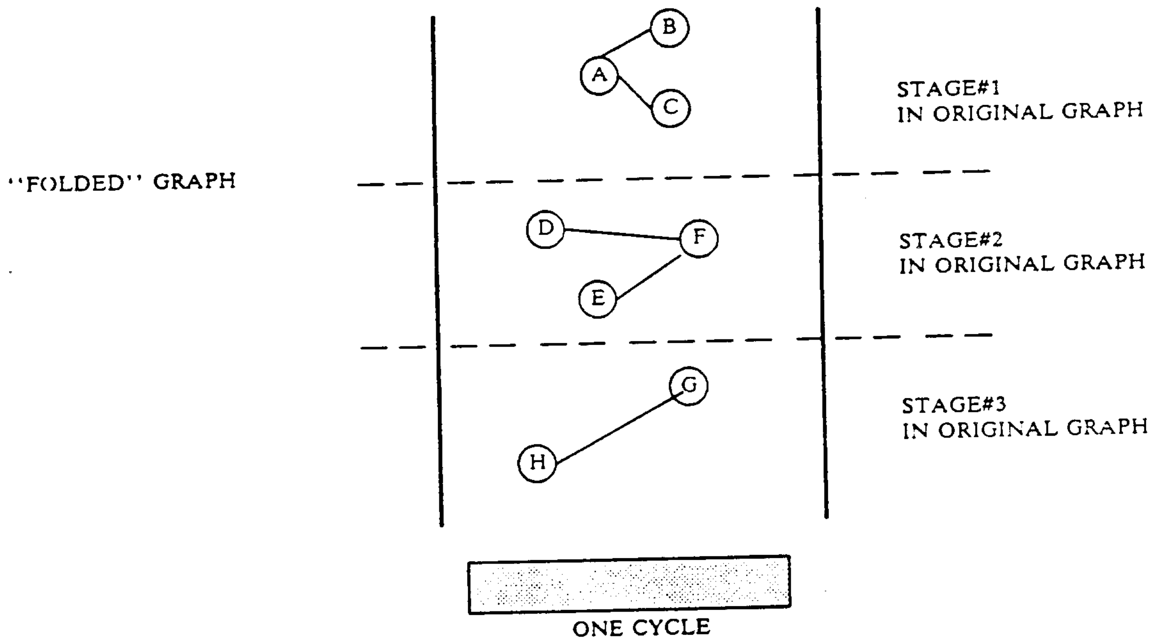
Once the dataflow graph has been partitioned, and the phase lengths have been calculated, we can use the Phase Assignment tool (PA) once again to determine on which phase/cycle each gate of the datapath should be run. When PA was called from PC, we set control parameters for PA so that PA would run fast and produce an approximate result as a guide to the higher level PC task. This time, we can call PA to do a full-precision job. The next chapter describes how this is done.

FIG 18: PHASE LENGTH CALCULATION





FOLD (AUTOMATIC)



BY RUNNING SP ON THE FOLDED GRAPH, THE CYCLE IS BROKEN UP INTO A SET OF PHASES.

EACH SUCH PARTITION IS USED TO SEQUENCE THE ACTUAL DATAPATH VIA PA.

PA THEN TELLS US HOW FAST THE PIPE WILL RUN WITH THIS SET OF PHASES; WE PICK THE BEST SET OF PHASES GENERATED BY SP.

Section III - Phase Assignment

PA is the next tool in the OPD set, and performs Phase Assignment on datapath gates. In this chapter, we define the Optimal Phase Assignment (PA) problem. We use the Microcode Compaction problem [Fisher 81] to show how to prove that PA is NP-complete. We then examine related problems and algorithms, and discuss and evaluate two approaches for solving PA. The first one is probabilistic and uses Simulated Annealing (SA). We discuss how we applied SA for solving PA, and use an analogy with physics to discuss some of the tradeoffs involved. We cover the annealing cost function, the move generation and the choice of the annealing control parameters. The second approach is heuristic, and leads to a fast algorithm for solving PA.

The main purpose of the Simulated Annealing algorithm is to produce high-quality reference solutions for a few examples. We use these solutions to demonstrate the effectiveness of the much faster heuristic algorithm. The long run-time of SA is therefore not a problem in our case.

We also show that the constraint set chosen for PA is powerful enough to capture most usual design situations.

1. Problem Specification

1.1. Input

The input to the PA optimization problem consists of a register-transfer level description of the circuit, plus a description of the clock and the pipeline stages. The input consists of the following elements.

B_i $(i=1,..,N)$	A set of N interconnected blocks (gates and latches). A block is simply an operator; it can be as simple as a logic gate, or as complex as a whole processor.
TY_i $(i=1,..,N)$	The type of each block (static logic gate, dynamic, domino, nora, latch)
D_i $(i=1,..,N)$	The propagation delay of each block expressed in the same units used to specify the lengths of each clock phase.
DAG	A Directed Acyclic Graph that expresses the connectivity among blocks. We show later how common causes for cyclic connectivity graphs can be worked around.
L_j $(j=1,..,P)$	The length of each clock phase. There are P phases. These lengths are calculated by PC.
$(ST1_k, ST2_k)$ $(k=1,..,S)$	For each one of the S pipeline stages, $ST1_k$ is the input latch to the stage, while $ST2_k$ is the output latch. The stages could, for instance, be those calculated by SP. Since stages are consecutive, $ST1_{k+1}$ is the same latch as $ST2_k$.
Constraints	There are 5 types of constraints. It is possible to force a block to run on a particular phase, to make a set of blocks run on the same phase, to force a set of blocks to run on disjoint phases, or on different starting phases, and to make pipeline stages run at non-overlapping times (so as to allow resource sharing, for example).

Appendix 2 describes the input and the output format for PA. Appendix 2 also shows two examples, a small test case and a datapath example derived from the "fraction datapath" of the SPUR [Patterson 87] [Hill 85] Floating Point Unit [Adams 86].

1.2. Optimization Objective

PA does not modify the datapath or the stage partitioning since those tasks have already been performed earlier during the design by the designer and SP/PC. The objective is to assign a clock phase to each block so as to maximize the throughput of the pipeline subject to all given constraints. In fact, the program assigns a fire time F_i $(i=1,..,N)$ to each block B_i. The fire time is the absolute time when the block can start evaluating. For a static block, this is the time when the inputs are ready. The precedence constraints imply that:

For any blocks B_i and B_j such that there is an arc from B_i to B_j in the connectivity

DAG, we should have $F_i + D_i \leq F_j$. This ensures that the inputs to B_j will be stable when it uses them.

The objective of maximizing the pipeline flow rate is replaced by the simpler aim of minimizing the longest stage length. This objective was chosen because the time spent in the slowest stage is often what limits the overall throughput. The simplified objective can be expressed as:

Minimize the Max of the difference between the fire time of the output latch minus the fire time of the input latch for each pipe stage, over all the pipe stages. or:

$$\text{Minimize } \underset{k=1}{\overset{k=S}{Max}}\left[F_{ST2(k)} - F_{ST1(k)}\right]$$

1.3. Related Optimization Problems and Algorithms

The Micro-code compaction (MC) problem [Fisher 81] can be reduced to PA.

In order to solve MC using PA, we map each micro-op to a block whose delay is equal to the time taken to execute that micro-op. We use the connectivity DAG to specify the precedence constraints among micro-ops. We can describe resource conflicts among micro-ops by introducing a constraint that specifies the blocks that should not be simultaneously used. For example, if the micro-engine has only one adder, we would create a constraint to specify that two ADD blocks should not be used at the same time. Lastly, we define a single pipeline stage that corresponds to the total execution time. This mapping of MC to PA can be done in polynomial time.

An optimal solution to the corresponding PA problem will produce an optimal packing of the micro-ops into horizontal micro-instructions. MC is therefore reducible to PA. As we mentioned earlier, MC is NP-complete. It therefore follows that MC and PA are both NP-complete.

However, it is difficult to use micro-code compaction algorithms directly for PA. Most such algorithms optimize in a fairly local way, such as [Landskov 80], performing code

motion mainly within basic blocks. Those that apply global reorganization [Fisher 81] rely on the structure of typical micro-programs in order to achieve fast algorithms. This makes them less applicable to PA.

PA is also analogous to a one-dimensional constrained **multi-layer** compaction problem. To see this, we map each logic block B_i to a line of length D_i, the delay of the block. The absolute x-coordinate of the line in the layout X_i corresponds to the fire time F_i of the block. We also create as many layers as there are pipeline stages, and map all the blocks in one stage to the corresponding layer. The objective of minimizing the max stage length then maps to one of minimizing the layout size, subject to the ordering relation specified by the connectivity DAG. Our constraints map into layout constraints that are somewhat bizarre; for instance, firing a block on the fixed phase Φ_1 maps into placing the line such that its x-coordinate is an integer multiple of some quantity (the cycle time). Moreover, some of the constraints in PA are global - while layout compaction is a local process. This is why it is difficult to apply standard layout compaction algorithms such as [Hsueh 81] [Weste 81] to solve PA.

A typical circuit might have a hundred blocks, with a total cycle time of 500 units (for ex. 500ns). There might be about 50 or 100 constraints; this gives us a search space with one hundred dimensions. Of course, the constraints reduce this number.

The large search space is the main reason why integer programming techniques, such as those used by [Leiserson 83] for retiming, were not chosen for PA. Dynamic Programming can not be applied since the *Principle of Optimality*, as described in [Horowitz 84] p. 199, is not satisfied by our problem.

2. Simulated Annealing Based Phase Assignment

2.1. The Generic Algorithm

Simulated Annealing (SA) [Kirkpatrick 83] is a general search technique suitable for solving constrained multi-variate optimization problems. In the case where there are no

constraints, the objective is to minimize a given function $F(x_1,..,x_N)$, where each variable x_i belongs to a domain $D_i, i=1,N$ of possible values.

SA works by establishing a physical system that is analogous to the problem. Such a system is described by giving its state vector $\vec{x}=(x_1,...,x_N)$ and its energy function $E(\vec{x})$. We can think of this system as being a crystal, for instance. It is possible to find a state of minimal energy for a physical system by using the (physical) process of annealing. First, the crystal is melted by heating it up; the energy E is then high, the state $\vec{x}$ is random. Then, the crystal is progressively cooled (annealed); the temperature T is reduced by small steps, waiting for thermal equilibrium at each temperature. Finally, when T is low enough, the crystal settles into a stable state of minimal energy.

If we consider our $\vec{x}$ to be the state vector of a physical system whose energy function is $E = F(\vec{x})$, then we can minimize F by simulating the physical annealing of the crystal. Of course, we also need to introduce a parameter T, that is the analogue of physical temperature.

This simulation is performed via the Metropolis algorithm [Metropolis 53], which was used in the early days of scientific computing to simulate many-bodied physical systems in equilibrium at a given temperature. This method determines the expected values of variables of the system at a given temperature T by generating a set of states $S_1, S_2, \cdots$ that are representative of the system's behavior at T. The variables are then averaged over this set of states. The states are generated one by one. A new state S' is generated from the previous state S by randomly changing one of the state variables and calculating the change in energy ΔE that would result; the new state is then entered into the set of representative states with a probability equal to the Boltzmann factor $e^{-\frac{\Delta E}{kT}}$, or 1 if $\Delta E \leq 0$. The first state to enter the set can be random. Fig.19 outlines the SA algorithm and the Metropolis technique. To cool the system, T is multiplied by the cooling factor α, usually chosen such that $0.8 \leq \alpha < 1$.

FIG 19: GENERAL SA ALGORITHM

```
procedure SA;

    /* for each temperature */
    while outer-loop-criterion not satisfied, do

        /* do the Metropolis simulation */

        while inner-loop-criterion not satisfied, do

            generate move at random;
            evaluate ΔE for move;

            if ΔE<0 accept move;
            else if e^(-ΔE/T) > R0_1 accept move;
            /* R0_1 denotes a random number between 0 and 1 */
            else reject move;

            if move was accepted, update configuration to new state;

        end while;

        /* end of Metropolis simulation */

        /* update Temperature Temp to cool system; */
        T = T*α;

    end while;
end SA;
```

SA is therefore characterized by three procedures:

- The *Move Generator*, which decides which variable should be randomly changed, and to what value, in order to generate S' from S;
- The *Inner Loop Criterion*, which decides when enough representative states have been generated at a given temperature to ensure that the Monte-Carlo Metropolis simulation truly reflects the physical state of the system in thermal equilibrium at T;
- The *Outer Loop Criterion*, which decides when the system has been cooled to a sufficiently low temperature.

In the physical process of annealing, it is necessary to cool the substance *very* slowly to produce a proper crystal. If the cooling is too rapid (quenching), the substance may enter a metastable state that corresponds to a local minimum of the energy. This local minimum can be quite far from the global minimum. The number of variables required to describe a physical system may be of the order of Avogadro's number, or $10^{23} = 2^{77}$. Moreover, the number of states generated per temperature for each variable can be estimated by taking the ratio of the time needed by the crystal to attain equilibrium at a fixed temperature divided by the characteristic vibration frequency of the crystal lattice. This ratio may be hours or days divided by pico-seconds, or roughly $10^{15} = 2^{50}$.

It follows that, in practice, the SA algorithm can not perform a true simulation of the physical process of annealing. Even with the fastest CPU's available, it is very difficult to generate more than a few hundred states per variable per temperature. One reason why physical crystals can go through so many states in a fairly short time (hours) is that the crystal is a highly parallel analog computer, with 10^{23} or so atoms working together, and the clock cycle is very fast (picoseconds).

Practical SA implementations can make up for the small number of moves by generating them in a smart fashion, so as to rapidly go through a set of representative states. In particular, it is pointless to generate moves that will systematically be refused. We want

to generate moves that result in a value of ΔE that is negative or not too large compared to T. This ensures that the probability of accepting the new state, given by $\min\left[1, e^{-\frac{\Delta E}{T}}\right]$ is not minuscule. Using such techniques, SA provides a robust search method that is capable of getting out of local minima by accepting uphill moves with a probability given by the Boltzmann factor.

To handle a particular constraint in SA, we add a corresponding term to the objective function F to be minimized. This term C is a *penalty function*, and takes on large positive values when there exists a constraint violation. At high temperatures, SA will therefore explore configurations that do not meet some of the constraints, since $\frac{\Delta E}{T}$ will be small even if ΔE is large. This leads to a value of $e^{-\frac{\Delta E}{T}}$ that is large enough to accept the state, even if there is a constraint violation. However, as T decreases, the system will refuse moves that increase C, and settle in a state that minimizes $F+C$, thereby avoiding constraint violations. The other approach is to generate the moves so as not to cause constraint violations.

We now discuss how we have used SA to solve the PA problem. We cover the choice of the energy function, the move generation, and the annealing control.

2.2. Energy Function and State

In our use of SA for PA, the state of the system is defined by the vector $\vec{FT} = (F_1, .., F_N)$ of the fire times of each block. We take the penalty function approach to ensure that the given constraints are met by the final solution.

2.2.1. Basic Terms

The Energy function $F(\vec{FT})$ has three terms:

$$F(F\vec{T}) = SL + ST + PEN$$

SL	The slack cost, proportional to the sum of the squared signal slacks. The slack on a signal that connects the output of one block *A* to the input of another block *B* is the amount by which we could delay the output of *A* and still have the value ready before *B* fires.
ST	The stage length term, proportional to the sum of the squared length of each pipe stage.
PEN	The penalty function, to ensure that constraints are met. We have different penalty terms for each type of constraint; in general, the penalty function is equal to a constant plus the square of the amount by which the constraint is violated. For instance, for a non-overlap constraint, the cost would be a constant plus the square of the overlap time between the concerned blocks. If the corresponding constraint is met, the penalty is zero.

The terms are therefore calculated as follows:

SL	$sl_scale * sum\ slacks^2$
ST	$st_scale * sum\ (stage\ lengths)^2$
PEN	$pen_offset + pen_scale * (amount\ of\ violation)^2$ (only if the corresponding constraint is broken). Note: we have one *PEN* term for each (broken) constraint; we sum all these terms into the cost function.

sl_scale, st_scale, pen_scale are scaling factors. pen_offset is chosen to ensure that all constraint violations will be removed at $T = 0$.

2.2.2. Balancing Cost Function Terms

The choice of the offset and of the scale factors for the various terms in the cost function heavily influences the optimality of the final solution.

If one of the energy terms is much larger than the others, then that term will dominate the annealing process at high temperatures. For instance, if we have a very large stage length term, the SA algorithm will concentrate on minimizing the stage length at high T, while it will try minimizing *SL* and *PEN* only at lower temperatures. In effect, if the cost function terms are unbalanced, the minimization objectives are separated according to temperature. The algorithm will therefore not minimize all the criteria simultaneously; this generally leads to sub-optimal solutions.

We tried a very large ST term, and found that, at high T, SA decided to crush the stage lengths down, even at the expense of breaking certain constraints. As T decreases, it might turn out impossible to remove the constraint violations. On the contrary, if the terms are well balanced, SA will simultaneously optimize for all the criteria. This requires slower cooling, but produces better results.

In order to choose well balanced factors, we reason on the physical system that SA simulates. Since the energy functions are of the form x^2, this system consists of a set of blocks interconnected by perfect springs. The scale factors correspond to the stiffness of the springs. Fig.20 shows the equivalent system. We see that the stage term springs connect very distant blocks, while the slack term springs tie up neighboring blocks. Similarly, the penalty function terms are short range, since they tie a block to the nearest slot that does not violate the constraint. We want the forces exerted by the springs to be reasonably balanced, so as to avoid crushing the system.

From this analogy, it is immediately clear that the penalty offset pen_offset should be larger than the amount by which the slack term could be reduced by allowing constraint violations:

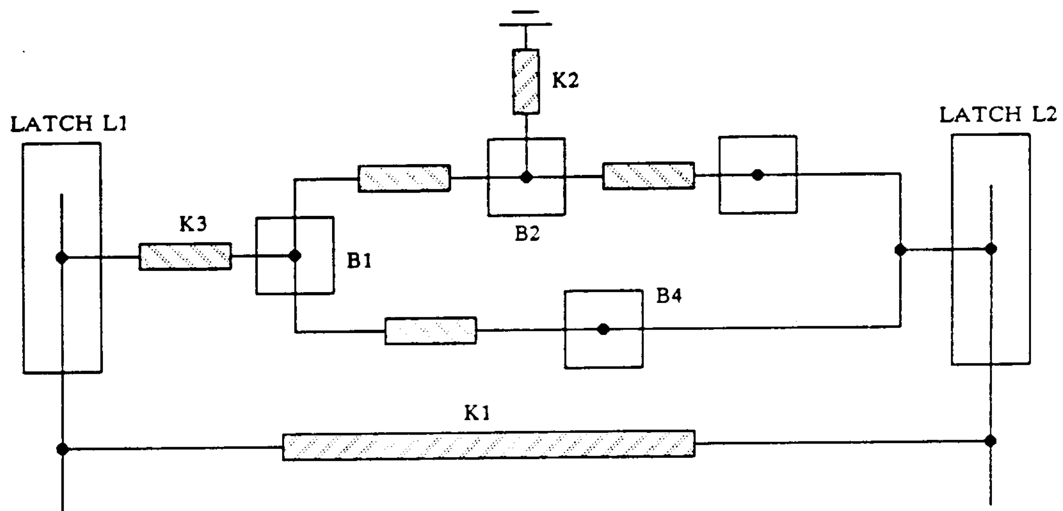
$$pen_offset > slack_{ij}(constrained) - slack_{ij}(unconstrained)$$

for any single constraint violation. This ensures that those springs will not crush the blocks and make them break constraints.

A set of slack term springs in series is connected to a single stage term spring. Typically, there as many slack springs in series as blocks per stage, bl_st. Since bl_st springs in series have a combined stiffness equal to $\frac{1}{bl_st}$ times that of a single spring, we need to weaken the stage term scale factor. This ensures that the stage term will not crush the blocks and cause precedence violations. Using this model and experimentation, we arrived as the following energy function factors. These factors do not account for blocks in parallel; this omission can be overcome by more conservative annealing parameters, which leads to longer run times. However, this was not a problem in our case since we used SA only to

FIG 20: PHYSICAL EQUIVALENT OF PA

NOTES: Bi ARE THE BLOCKS
Ki ARE THE SPRING STIFFNESSES
Li ARE THE LATCHES



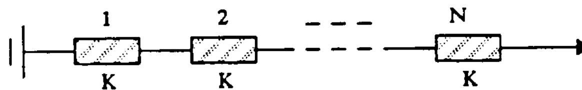
STAGE = FROM LATCH (L1) TO LATCH (L2)

K1 = STAGE SPRING

K2 = LOCAL CONSTRAINT FOR BLOCK B2

K3 = PRECEDENCE CONSTRAINT

SERIES SPRING NOTE:



OVERALL K = $\frac{1}{N} \cdot K$

produce reference solutions.

Parm	Value	Why
sl_scale	1	Reference Factor
st_scale	$\frac{1}{\sqrt{bl_st}}$	avoid crushing blocks
pen_offset	(dynamic)	should exceed the reduction in the slack and stage cost functions that is achieved if we can gain one clock cycle by breaking constraints.
pen_scale	$\sqrt{bl_st}$	the stiffness of these springs should be *bl_st* times greater than that of the stage length term springs to avoid precedence violations.

2.3. Move Generation

We describe three flavors of move generation, from random to smart, heuristic move generation.

For random move generation, we select the blocks sequentially, and displace each block by generating a new, random fire time. The move generation is fast, but this results in a less effective generation of useful moves than if exchange moves are used, as in the PLA-folder **genie** [Devadas 86]. PA is more similar to placement problems [Sechen 85] than it is to permutation problems such as the Traveling Salesman Problem [Kirkpatrick 83]. While annealing can rapidly solve the Traveling Salesman Problem with thousands of cities [Kirkpatrick 83], it takes much more CPU time to solve thousand-block placement problems.

The Range Limiter mechanism helps generate more useful moves. At low temperatures, only local moves that do not perturb the system too much will be accepted. We therefore generate new fire times that are within a specified range of the old ones. As the cooling progresses, we reduce the extent of this range. In our SA, the range is reduced so as to ensure that at least 5% or 10% of the generated moves are accepted.

We experimented with a move generator that never broke precedence constraints. However, we found that this lead to sub-optimal solutions, or to no solution at all in some cases. SA tends to get stuck in illegal or sub-optimal solutions from which it can not

escape if precedence constraints are always satisfied.

We also tried another approach to respecting the precedence constraints. We re-schedule a large part of the network each time a (random) move that breaks precedence is generated. We found that SA had trouble gauging the effect of any one move. This is because every move is followed by a heuristic procedure that may re-organize large parts of the circuit before generating the next move. We also experimented with a move generator that always satisfies a larger set of constraints (all local constraints). But the different constraints interact, and such an approach prevents SA from considering them all.

A better way might be to anneal only on the global (scheduling, non-overlap) constraints and use heuristic algorithms to re-assign phases each time one of the scheduling constraints is changed. However, since our main purpose in coding SA was to generate reference solutions to validate faster heuristic algorithms for PA, the long run times did not appear to be a problem.

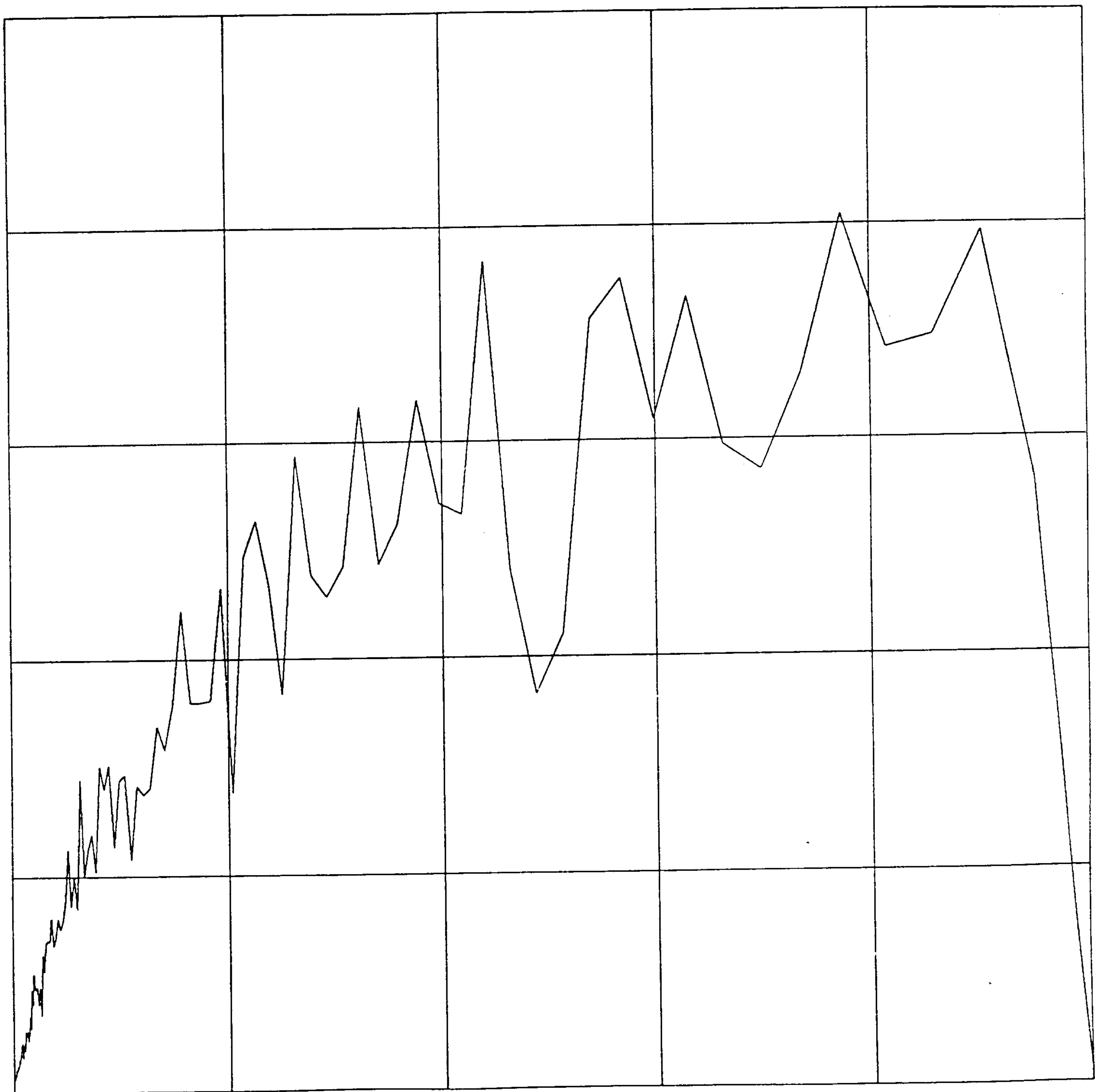
2.4. Annealing Control

Annealing control refers to the inner loop criterion, the outer loop criterion and the choice of parameters.

The inner loop criterion decides when a sufficient number of states have been generated at a given temperature. We have shown that it is not practical to generate as many states as in the physical annealing. In practice, we generate a fixed number $NS(T)$ of states per block per temperature. $NS(T)$ is on the order of 100 to 300. This implies that we do not test, and therefore do not wait, for thermal equilibrium. In fact, the curve in Fig.21 shows clearly that we do not come anywhere close to equilibrium. The correct energy function - had we waited at each T - is a monotonically decreasing function of T. The fact that the energy can sometimes go up as T goes down shows that the Monte-Carlo Metropolis simulation is not exact. This scheme is similar to that used by the placement package **TimberWolf** [Sechen 85] and the PLA-folder **Genie** [Devadas 86].

FIG 21: ENERGY (COST) VS. TEMPERATURE ANNEALING - FPU RUN

ENERGY*10^6
VERSUS
TEMPERATURE*10^6



0 2 4 6 8 10

We ran an experiment in which we waited for the average energy to stabilize at each temperature before cooling. This is somewhat analogous to achieving thermal equilibrium. We found that $NS(T)$ increased roughly by a factor of 10. $E(T)$ also became a monotonically decreasing function. However, the quality of the results did not improve enough to justify a ten-fold increase in computing time; we therefore use the non-equilibrium scheme.

In order to improve the quality of the final solution, we increase $NS(T)$ as T decreases. It is sufficient to generate a small number of states at high temperatures since the system is in a random configuration anyway and since SA is only performing an approximate exploration. However, as the system is cooled, it becomes useful to generate more states in order to evaluate the energy with greater precision. In our implementation, we divide $NS(T)$ by the cooling factor α when the fraction of accepted moves drops suddenly. Such a situation occurs when the energy is rapidly changing, corresponding to a phase transition in the system. It is then desirable to generate more states to observe this transition better.

The outer loop criterion is simple. If the energy has not changed appreciably over the last three temperatures, we stop the annealing process. This criterion has proven useful in **TimberWolf** [Sechen 85].

In practice, we found that $NS(T)$ should be greater than 150 generated states (accepted or refused) per block per temperature. Also, the offset term *pen_offset* for the penalty function should be just large enough, but not more. The best results are obtained with $\alpha \geq 0.90$ to ensure slow cooling. We also found that both $NS(T)$ and T have to be much higher if the problem involves global constraints. Other annealing-based programs also have trouble with global constraints; for instance, placement packages using SA are not well-suited to regular, highly-constrained datapath placement. For such problems, the choice of annealing parameters is very critical. For the SPUR FPU and an example derived from the RISC-II [Katevenis 83] [Sherburne 84], run times are about 0.5 to 1.5 hours of VAX8800 CPU. Our main aim in coding SA is to verify the solutions produced by

heuristic algorithms described next. In both cases, SA did not find a significantly better result than the heuristic algorithms. Fig.22a and Fig.22b show how the move generation range and the number of moves generated vary with temperature. There is a noticeable phase transition around $T=10^5$, as evidenced by a rapid decrease in the range of those moves that the annealing generates. The program adjusts this range in an attempt to ensure that at least 5% or 10% of the moves generated are accepted. The rapid decrease of this range takes place because the program is responding to a rapid decrease of the system's randomness. The system is suddenly "solidifying".

2.5. Coding Issues

In order to ensure reasonable run times, we calculate the energy functions in an incremental fashion. Every time a move is generated, we re-calculate only those terms that change in the energy function. To achieve this, we use data structures that immediately tell us which constraints and which terms depend on a particular block's fire time. Also, much of the annealing code is inline - a compiler that could automatically expand functions inline would be very helpful here. It is worth avoiding function calls in the critical loop since this loop may be executed several million times.

In order to debug the annealing code, we compare the energy as calculated incrementally with the energy re-calculated at every accepted move. We found the direct calculation of the energy to increase compute time by about a factor of 10. Overall, our annealing code was more difficult to debug than usual because of the need for very high speed, its exotic data structures, and a rather long critical loop.

3. Heuristic Algorithms for Phase Assignment

We describe heuristic algorithms that provide good solutions rapidly, within seconds for typical problems. Such algorithms make it possible to use OPD interactively. These algorithms are built upon GPA, a Greedy Phase Assignment algorithm.

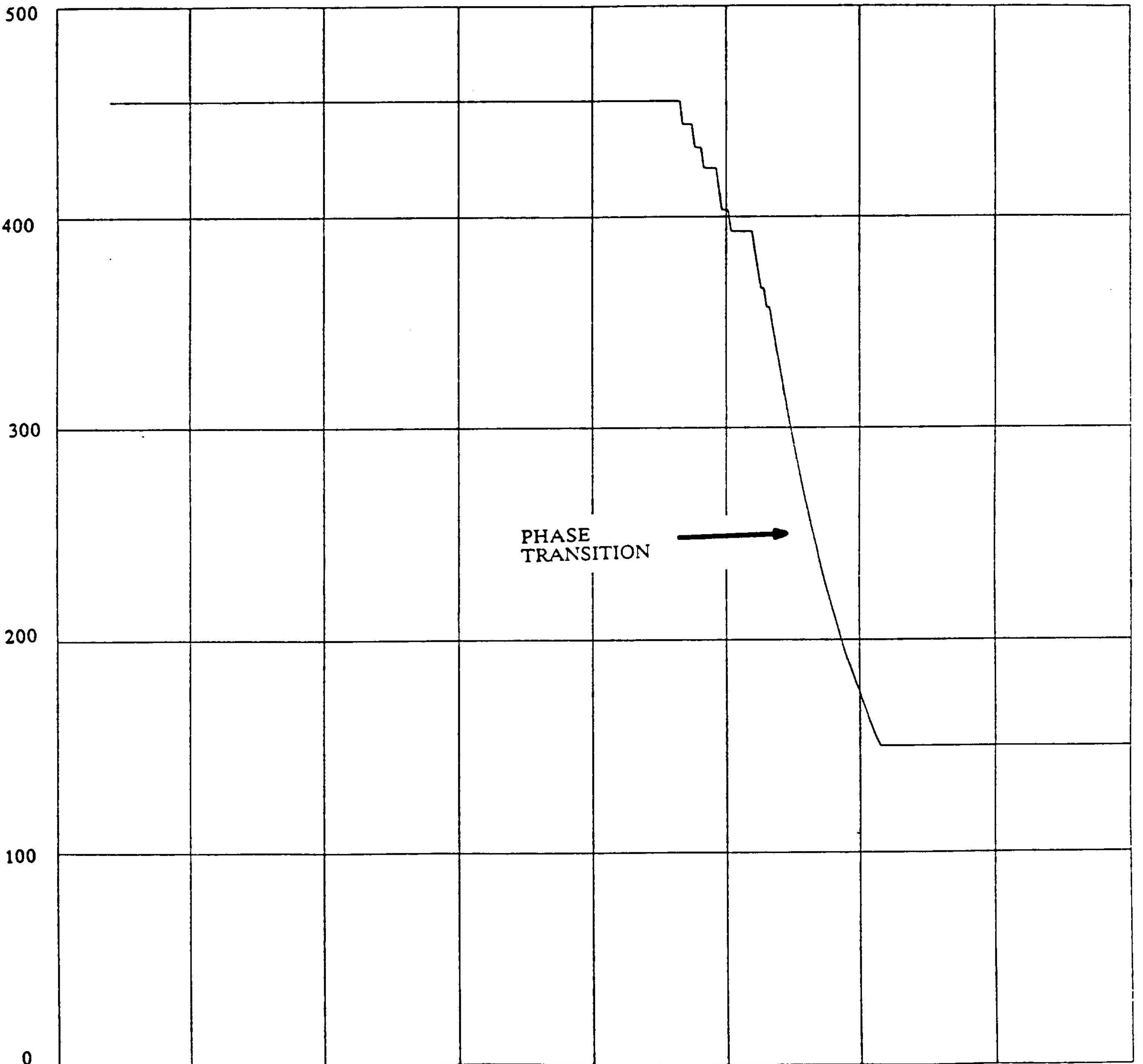
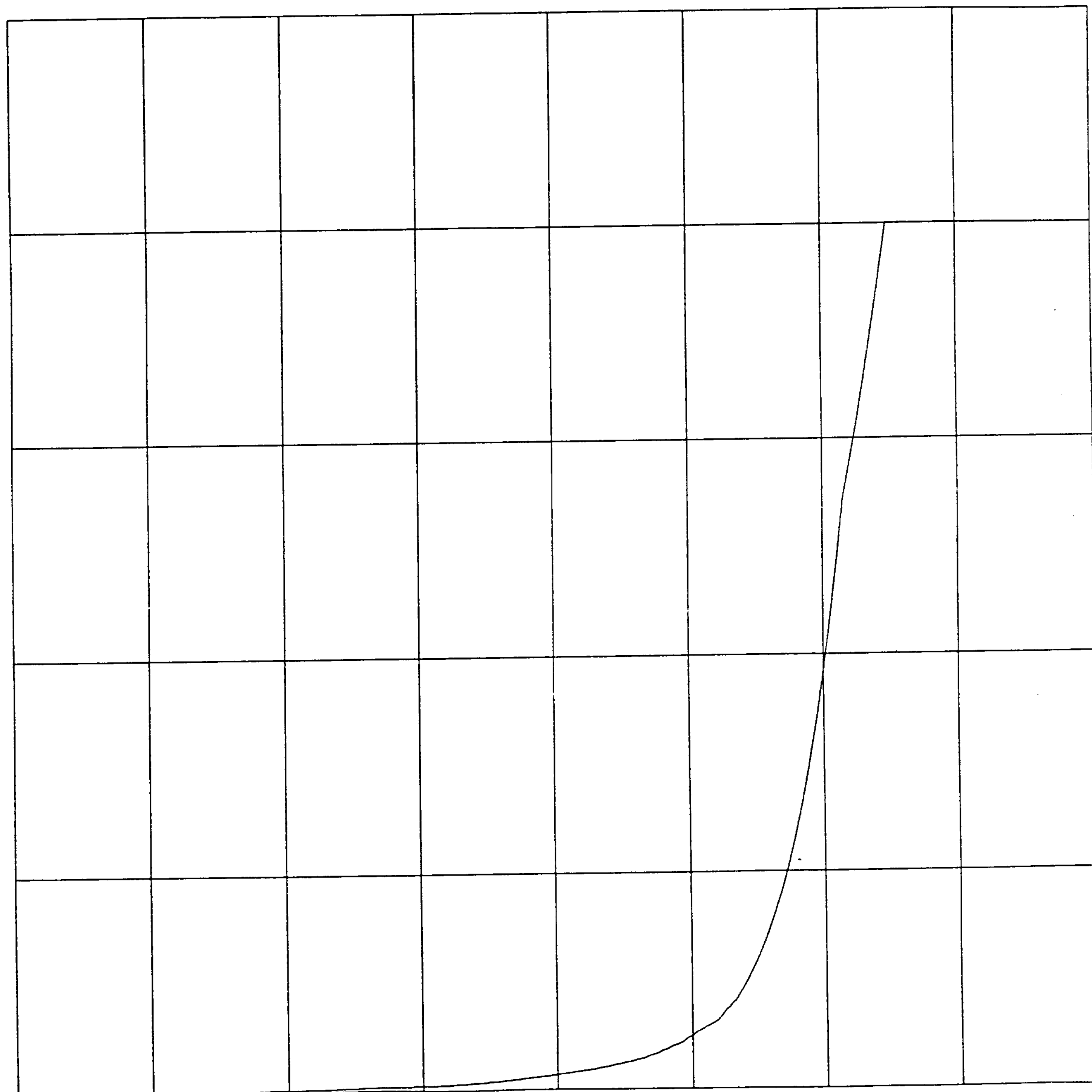

NUMBER OF MOVES PER BLOCK AT EACH TEMPERATURE
VERSUS
LOG (TEMPERATURE)
500
400
300
200
100
0
PHASE
TRANSITION
FIG 22(a): #MOVES PER BLOCK VS LOG(TEMP)
0.1 -log x- 1e+07 0 -y- 500
T=0.1
T=1
T=10
T=100
T=10^3
T=10^4
T=10^5
T=10^6
T=10^7

FIG 22(b)

RANGE OF ANNEALING (MAX RANDOM DISPLACEMENT OF A BLOCK'S FIRE TIME)
VERSUS
LOG (TEMPERATURE)



3.1. GPA: Greedy Phase Assignment

The basic GPA performs either a breadth-first or a depth-first trace of the network. Each block is assigned the earliest fire time that meets all the constraints involving it. Constraints that refer to blocks that are not yet scheduled are ignored. GPA using depth-first search guarantees a solution if there exists one; this is because any constraint can be met by delaying a set of blocks sufficiently.

3.2. HA: Heuristic Algorithm

There are two reasons why GPA often produces sub-optimal solutions. GPA may schedule a block to run too early, which can force other blocks to be delayed in order to meet the constraints. This leaves dead slack in the network and decreases throughput. Also, global constraints such as specifying that two stages A and B must not overlap in time can have two valid solutions. Either all the blocks in A can run before any block of B, or the reverse. GPA, by its nature, does not consider any such exchange moves between independent blocks.

Fig.23 outlines HA. HA visits each block B in the network level by level, using depth-first ordering. For each B, HA tries *NTRIALS* consecutive possible fire times. For each possible fire time, HA calls GPA to re-schedule the part of the network that topologically follows the block. The fire time for B that produced the best schedule is selected.

HA therefore re-schedules, on the average, half the network for each block. The run time grows as the square of the size of the network. HA is capable of delaying a block's fire time to find a better schedule, and can consider the interchange of independent blocks by delaying one of them by a sufficient number of phases. HA attempts to pack stages that should not overlap in as compact a fashion as possible; however, should this packing fail, HA switches to an algorithm that guarantees a solution but might decrease throughput. This algorithm delays all the blocks of one of the stages to ensure non-overlap.

The mechanism that HA uses to maintain a global view of the network is similar to

Figure 23: Heuristic Phase Assignment Algorithm HA

```
{ Picks best fire time for a given block B. Tries NTRIALS different times }
procedure ONE-HA (integer NTRIALS, block B)
   let T = earliest fire time for B consistent with constraints;
   for I from 1 to NTRIALS do

      unschedule all blocks topologically following B;

      { Try scheduling B after T, by the Greedy scheduler }
      { This move checks if a better solution results by delaying B }
      call GPA (block = B, min-fire-time = T);

      { re-schedule that part of the network which topologically follows B }
      schedule all other blocks on the same DFS level as B
      and following B by calling GPA on them;

      if this is a solution that meets all constraints, record it.

      let T = next sensible fire time to try for B;

   end for;
   Pick the best found solution: this gives the fire time for B;
end ONE-HA;
{ Note: the run time of ONA-HA is proprotional to Network-size * NTRIALS }


{ Finds a good assignment for the whole network }
procedure HA (NTRIALS);
   for each block B of the network in topological DFS order,
   call ONE-HA (B, NTRIALS);
   { Note: this progressively fixes the fire time of each block.
     The network is in effect "zone-refined".
   }

   If no solution found, switch to a guaranteed algorithm.
end HA;
{ Note: the run time of HA is proprotional to (Network-size^2) * NTRIALS }
```

the Zone Refining (ZR) process, as applied to compaction problems [Shin 86]. Unlike ZR, HA has to re-process the whole part of the network that follows the point of change every time a zone gets changed since constraints can be global. Run times are usually of the order of a few seconds.

Fig.24 shows how the quality of the solution improves with *NTRIALS*.

3.3. Other Heuristics Considered

In order to handle global scheduling constraints, we considered using **urgency-scheduling** [Park 85]. This method orders blocks according to their distance to the end of the network (forward-urgency) or from the start of the network (backward-urgency). The distance of a block to the end of the network is the length of the longest path from that block to a terminal block.

Urgency-scheduling consists of scheduling the blocks with the highest urgency first. This method was not chosen because the local constraints interact with the global ones; and urgency-scheduling has no mechanism to explore these interactions. [Park 85] describes applications of urgency-scheduling to synthesis problems that do not involve as many constraints as ours.

Another approach, used by A. Parker in the datapath synthesizer MAHA [Parker 85], is to schedule the blocks according to their **freedom** [Nagle 81]. Loosely, the freedom of a block is the difference between the earliest time at which it can be scheduled and the latest. Freedom-based scheduling was not chosen for similar reasons.

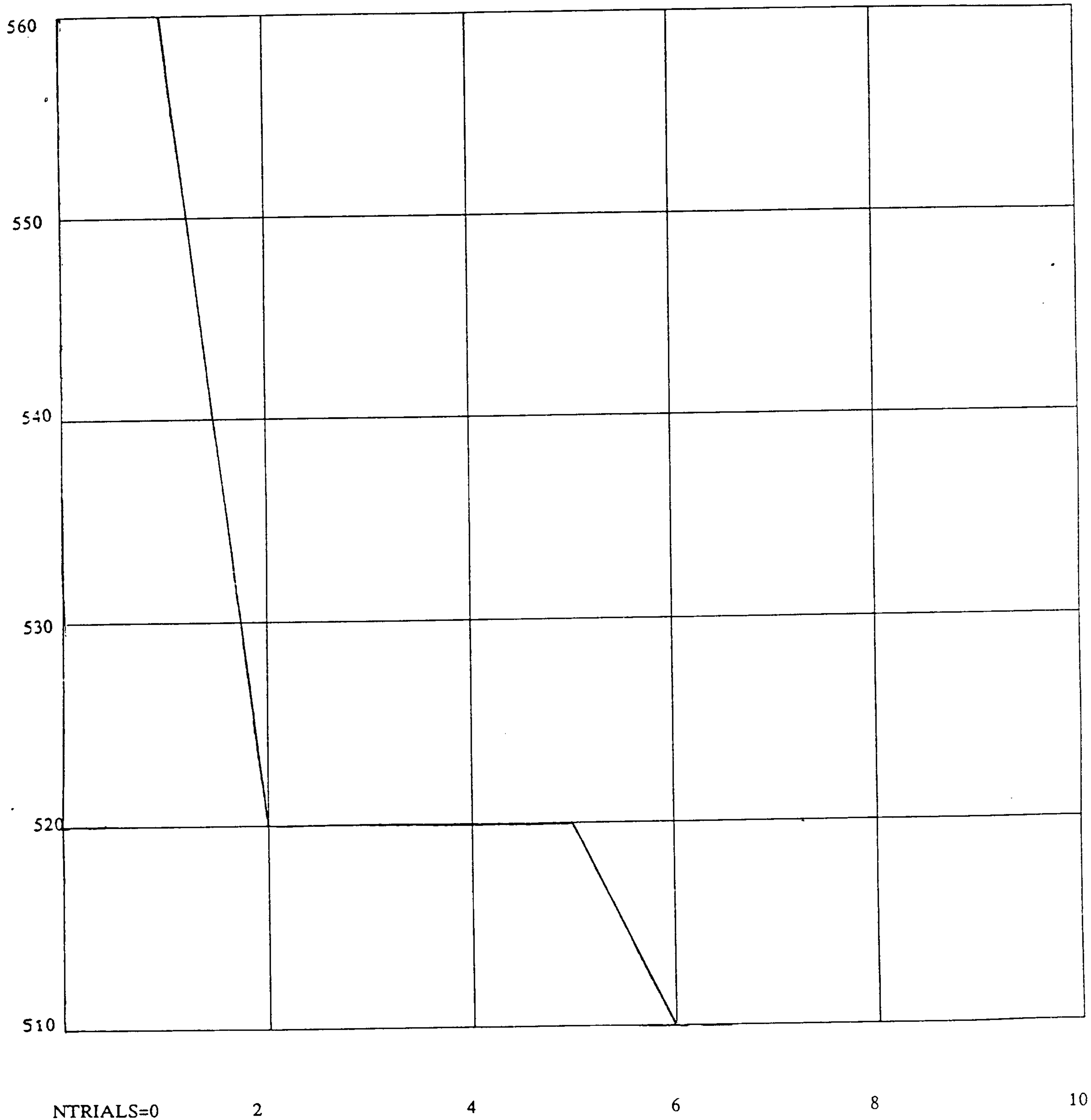
Local operations to "shaké' the network appear insufficient, since these cannot explore global interchanges while respecting the constraints.

4. Constraint Set Completeness

The constraint set is designed to capture most of the common hardware and scheduling constraints, so as to adapt to any technology and any design style.

FIG 24

SOLUTION QUALITY VS. NTRIALS (HEURISTIC PHASE ASSIGNMENT)



- Shared Resources: busses, register files, ALUs. Fig.25 shows an example. Shared resources are modeled as follows. Let *RESC* be a shared resource. We create blocks $RESC_1,...,RESC_N$ for every use of this resource. We introduce an extra constraint to force all these instances of a use of *RESC* to fire on the same phase, but at non-overlapping times. This constraint reflects the fact that two instances of *RESC* should not occur at the same time, but will however occur at the same phase (but different cycles) since they share the same hardware clock line. The constraints can also be used to describe datapaths with fixed loops. These loops correspond to a block being re-used. Such loops can be unrolled.

- Precharge schemes: fixed-phase precharge (the same phase is used for precharging all the gates), per-gate variable precharge phase, and intermediate schemes can be described. Chained dynamic and domino or nora gates can be handled too [Weste 85]. To handle chained dynamic gates, we create constraints to specify that a gate should not run on the same phases as a gate that it fans out to. To handle precharging, we first create a 'precharge' block for each precharged gate. This block fires when the associated gate is being precharged. We then specify that a gate G and its "precharge' gate P should not run on the same phase. This reflects the fact that a gate should not be precharged while it is being used. Moreover, if all the gates should be precharged on a common precharge phase, we simply specify that all these 'precharge' blocks should run off a common (and perhaps fixed) phase.

- Delays that depend on the input/output pair: such gates can be handled by adding dummy "delay" blocks.

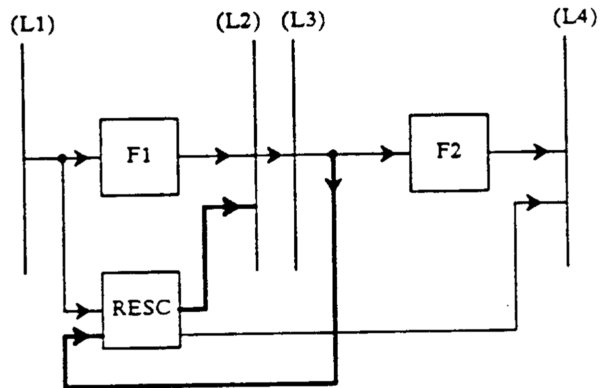
The output from PA therefore specifies exactly when each gate and each latch will be clocked; this output precisely defines the length of each pipeline stage. PA can generate a reservation table in the file format required by SCHED. SCHED can then calculate an optimal initiation sequence [Kogge 81] for the pipeline.

FIG 25: MODELLING SHARED RESOURCES FOR PA

L1,L2 L3,L4 LATCHES
F1,F2 LOGIC BLOCKS
RESC THE SHARED RESOURCE

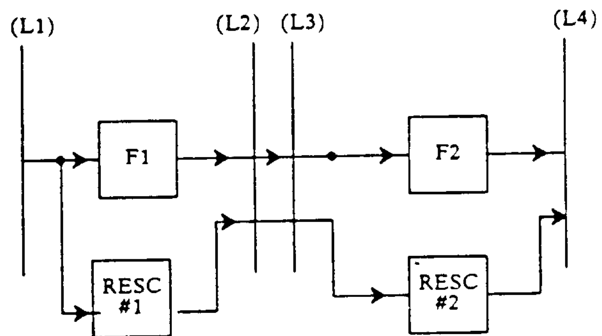
RESC IS SHARED BY STAGES (L1,L2) AND (L3,L4)

DATAPATH LOOP RESULTING FROM RESOURCE SHARING



A CORRECT WAY TO MODEL RESOURCE SHARING FOR PA
BY UNROLLING THE DATAPATH LOOP

RESC IS DUPLICATED; EACH COPY CORRESPONDS TO ONE USE OF THE ORIGINAL RESC.
CONSTRAINT: RESC#1 AND RESC#2 MUST RUN ON THE SAME CLOCK PHASE, BUT DURING DIFFERENT CYCLES.



Section IV - Reservation Table Scheduling

This section describes a scheduling procedure from [Kogge 81] that determines an optimal initiation sequence for a pipeline. The throughput of a pipe does not depend exclusively on the length of the longest stage; it depends on the exact pattern of stage lengths, that is, how many clock cycles each stage requires. After PA, we know this exact pattern. We can therefore now determine exactly when new data should be allowed to enter the pipe so as to maximize the average processing rate.

It can be quite difficult to find a good initiation schedule by hand. This is because the pattern of stage lengths found by PA can be irregular, with stages requiring different numbers of cycles to complete. With such complex patterns, it is not easy to manually determine when new data should be allowed to enter the pipe in order to maximize the processing throughput. PA tells us how long each stage is; however, PA does not produce the optimal new data entry times.

1. Scheduler Overview

The scheduler takes a static reservation table as input. It calculates an initiation sequence that maximizes the throughput. The optimal initiation sequence can turn out to be fairly irregular; it need not be cyclic with respect to the compute time of the reservation table.

1.1. Static Reservation Table

The initiation sequence specifies when new data should enter the pipe. The input to the procedure is a reservation table, which specifies which pipeline stages are used by a particular computation, for how long and in what sequence. Two examples from [Kogge 81] are shown in Fig.26.

Time

Stage	0	1	2	3	4	5	6
1			A		A		
2		A		A		A	
3	A		A				A

Compute time = 7

(a)

Time

Stage	0	1	2	3	4	5	6	7
1	B	B					B	B
2			B		B			
3				B		B		

Compute time = 8

(b)

Figure 26 Sample reservation tables

1.2. Reservation Table Generation

An exact reservation table, that takes into account all the hardware constraints, can be generated by PA after the Phase Assignment step has been done. Given this reservation table from PA, the scheduler will try to avoid collisions in order to maximize the pipe throughput. The performance is based on the **MAII** defined in the next section.

1.3. Latency

A key parameter in determining the performance of a pipeline is the *initiation interval*, or number of time units separating two initiations. An initiation takes place when a new datum enters the pipe. Since the goal of a pipe is to maximize throughput, the prime measure of actual system performance is the initiation rate, or average number of initia-

tions per clock unit. The average initiation interval is the reciprocal of the initiation rate, or the average number of time units between two initiations. We are looking for the MAII or the minimum average initiation interval.

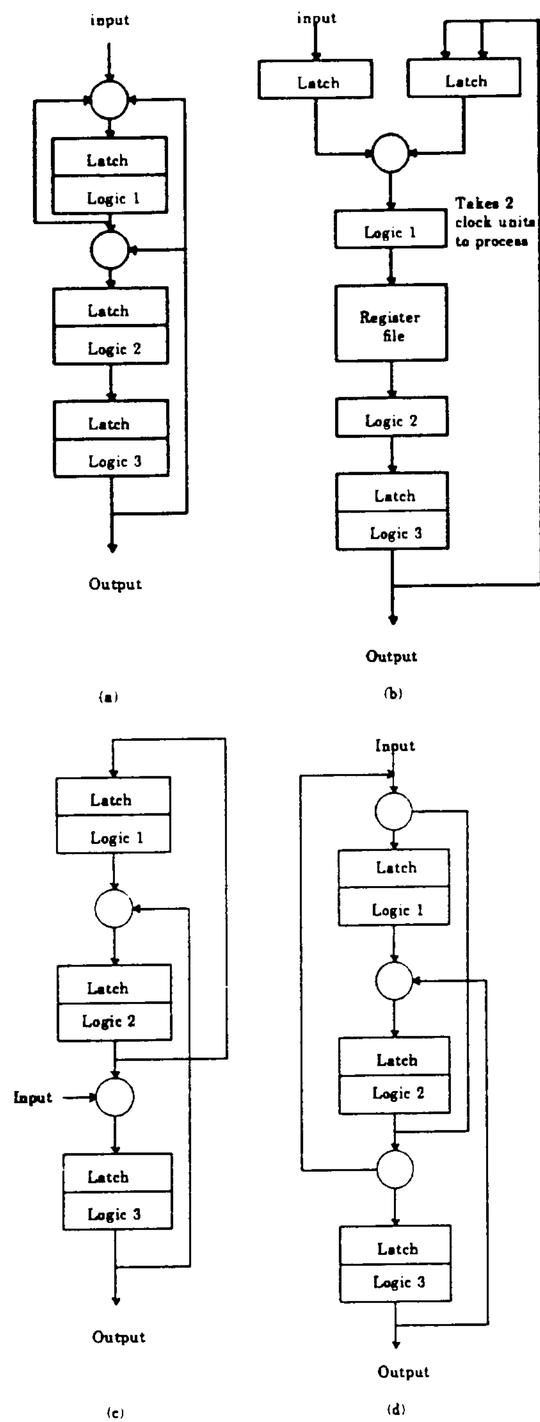


Fig. 27 sample pipeline
(a) and (b) for reservation table B
(c) and (d) for reservation table A

2. Scheduling Algorithm

2.1. Strategy

A greedy strategy is not optimal since, if a datum is allowed to enter the pipe too early, this may block further initiations and actually increase the MAII. The algorithm we use is exhaustive and finds the best solution. The algorithm also runs fast by using various techniques to effectively prune the search tree.

A bound on MAII can be determined from the reservation table alone according to the following lemma:

2.1.1. Lemma (Shar 1972)

For any statically configured pipeline executing some reservation table, the MAII is always greater than or equal to the maximum number of marks in any single row of the reservation table. On the other hand, the MAII is bounded above by the number of 1's in the initial collision vector defined in next section.

Let the number of marks in the ith row of the reservation table be $N(i)$.

$$max(N(i)) \le MAII \le Number\ of\ 1's\ in\ initial\ collision\ vector$$

In Fig 26, reservation table A has 3 marks in rows 2 and 3, and consequently its MAII is at least 3. Likewise the MAII of reservation table B is bounded below by 4. This lower bound gives the scheduler a quick estimate of the maximum performance possible for a given reservation tabie. Also, any scheduling giving a MAII larger than the upper bound can be discarded. However, there is no guarantee that the actual MAII equals the lower bound. The reason is that small latencies may cause collisions and therefore can not be used in an initiation sequence.

3. Data Structures

3.1. State Diagram

The state diagram [Kogge 81] is a technique to rapidly determine if, at a given time, a new initiation in the pipeline will conflict with any previous initiations. At each time

unit, the current pipeline configuration corresponds to one of the states. The arcs from one state to the next indicates what new state the pipeline might be in at the next time unit. All possible initiation sequences corresponds to paths in such diagrams. By analyzing all such paths, particularly the ones that form closed loops, those with minimum average initiation interval can be identified. Fig.28 shows a state diagram. Each box represents a state and contains a collision vector. The collision vector, described next, shows in compact form when new initiations can be made without resource conflicts from that pipeline state. The arcs in Fig.28 show how the pipe can change states by making initiations. Each arc is labeled with the number of clock cycles required by the pipe to make a transition from the arc's source state to its sink state.

3.2. Collision Vector

The particular information encoded into each state is termed a *collision vector*. This vector is a d-bit binary sequence, where d is the compute time of the reservation table. The d bits are labeled 0 to $d-1$ from left to right, with a 0 in position i indicating that a new initiation i time units from now will not conflict with any currently uncompleted initiations. A 1 indicates that a collision will occur, and therefore an initiation at that time must be avoided. The collision vector for each time period takes into account whether or not a new initiation was made in that period.

The collision vector for the initial state has a special name *initial collision vector*. Since it corresponds to the time unit when the pipeline is first started, it is a representation of what latencies are permissible between just two initiations, one at time 0 and one at time i.

3.3. Algorithm For Finding The Initial Collision Vector

3.3.1. Conceptual

```
for i = 0 to d-1 do {
    make one copy of the reservation table;
    shift the copy to the right i times;
    OR-in an unshifted copy of the same table;
    If there are two marks in the same stage-time entry anywhere
      bit i of the initial collision vector = 1
    else
      bit i of the initial collision vector = 0; }
```

In all cases bit 0 of this collision vector is 1 because overlaying a reservation table on itself causes a collision everywhere there is a mark. Likewise bit positions d and beyond are always 0 because the shifted and nonshifted tables never overlap.

3.3.2. Actual Implementation

An alternative approach is to construct the *forbidden initiation interval set*. The number i is a member of this set when, in at least one row of the reservation table there are two marks separated by i columns. Analysis of the marks in each row quickly identifies the members of this set. We notice that 0 is always in the set. Finally, for all i in the set the corresponding bit of the initial collision vector is 1. All other bits are 0.

```
for i = 1 to number of stage do /* for each stage */
  for j = 1 to compute-time do /* for each time */
    if (table[i][j] is marked)
      for k = j+1 to compute-time do /* find the forbidden set */
        if (table[i][k] is marked)
          insert_into_forbidden_set(k-j);
```

3.4. State Diagram

Once the initial state of the pipeline at time 0 is available, the equivalent states for all future times may be computed.

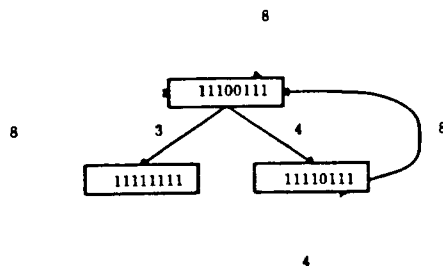
The state diagram shows the relationship between states at consecutive times. Because the problem itself is NP-complete, the number of states generated may be large. Since the states where no initiation occurred carry no information, they are deleted to produce the modified state diagram.

3.5. Modified State Diagram

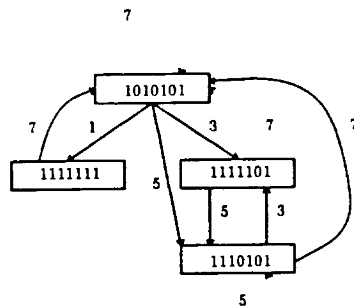
Such a diagram is similar to the original diagram but includes only those states resulting from new initiations. Two states in this new diagram are connected by an arc if and only if they were connected by some series of arcs in the original diagram. The number attached to an arc is the interval between two initiations. Figure 28a lists the modified diagram for reservation table B.

3.5.1. Algorithm For Generating The Modified State Diagram.

```
Put initial state with initial collision vector in unprocessed list;
while (Get_From_Unprocessed_List(state) != EMPTY) do {
  for each k such that the kth bit of the collision vector is 0 {
    New_State = Left_Shift(current_state.collision_vector) k times;
    New_State = New_State OR Initial collision vector;
    If New_State already exists
      Insert arc from current_state to existing state with value k
    else if New_State == current_state
      Insert arc to current_state itself with value k;
    else
      Connect New_State to state with an arc of value k;
      Enter New_State to unprocessed list; }
   Include an arc with value d from each state back to the initial state;
   Insert current_state into processed_list; }
```



(a) reservation table B



(b) reservation table A

Fig. 28 Modified State Diagram

3.6. Efficient Search

An initiation schedule is a cyclic sequence of initiations. Each initiation corresponds to an arc in the modified state diagram; an initiation sequence corresponds to a cycle in the state diagram. The initiation sequence must contain the initial state. An optimal schedule is an initiation sequence such that the average time separating consecutive initiations in the cycle is minimal.

Simple cycles are an important class in which each state appears no more than once. Figure 28b shows the modified state diagram for reservation table A. The initiation interval cycle (3,7,5,7) is not simple while (3,5,7) is. The utility of simple cycles comes from the following lemma

3.6.1. Lemma 2 (Shar 1972)

In any modified state diagram if there is a cycle with an average initiation interval L, there is at least one simple cycle with average initiation interval no greater than L.

3.6.2. Reducing the Search Time

The above lemma allows the scheduling algorithm to limit its search for optimal cycles to simple cycles because it guarantees that no nonsimple cycle can have a lower average initiation interval than one of the simple ones. This makes an exhaustive search feasible.

4. Examples and Results

4.1. Simple Pipeline Machines

The scheduler ran in a fraction of a second on most small examples (less than 10 stages, and less than 10 units compute time). Most of the time is spent searching the modified state diagram. The complexity of the state diagram depends on the possibility of new initiations. This is directly proportional to

Compute Time – Number of Elements in the Forbidden set. Therefore the less dense a reservation table is, the more complicated the modified state diagram is going to be.

Since the problem is NP complete, we may run into trouble with a complicated reservation table that generates many states. In practice, this seldom happens because new data is typically introduced frequently into the pipe and instructions usually complete within 15 to 20 cycles. If an instruction requires more than 10 cycles, usually it is an iterative finite state machine which implies the reservation table is very dense. Also, there are usually fewer than 15 stages. Therefore in real applications, the above problem seldom occurs.

4.2. More Complicated Machines

An artificial reservation table with compute time 15 cycles and 5 pipeline stages with the worst case pattern is used to test the scheduler.

```
% time scheduler -t 15 -s 5 < T5
The reservation table being optimized:
C0000C0000C0000
0C0000C0000C00C
00C0000C0000C00
000C0000C0000C0
0000C0000C0000C

The forbidden initiation
interval set contains time slot:
0 3 5 8 10 13

The range of MAII is : 4 <= MAII <= 6

The optimal sequence is as follows :
( 2 4 12 )
The order of states are:
110101001010010
110101101011010
111111111110010
with a MAII of 6.

23.9u 1.5s 0:46 55% 17+4102k 2+1io 0pf+0w  -- VAX 8800 times
```

This example causes 2^{15} states to be generated and corresponds to a worst-case pattern. It is very unlikely to find a hardware configuration that would result in such a table.

5. Control Synthesis Examples

Appendix 4 shows two control synthesis examples, using time-stationary and data-stationary pipeline control schemes [Kogge 81].

Conclusion

This report describes OPD, a set of co-ordinated tools whose objective is to help designers produce better pipeline structures in a shorter time. OPD spans a wide range of design levels, starting at the behavioral level. Because OPD has the ability to capture most technology-specific constraints, it remains useful right down to the final datapath and scheduling design steps.

Good pipeline design requires the solution of many complex inter-related optimization problems. We have developed, coded and evaluated various heuristic and probabilistic algorithms for solving these NP-complete optimization problems within OPD. Inter-dependencies are handled during one optimization step by using a simplified model of the related optimization steps.

We also illustrate how OPD works in conjunction with existing CAD tools and with the designer. The Design Methodology and the Optimization routines were verified using several small and large examples.

References

[Adams 86] G.D. Adams, "Functional Specification and Simulation of a Floating Point Co-Processor for SPUR", *Report No. UCB/CSD 86/311*, Computer Science Division, U.C. Berkeley, August 1986.

[Barbacci 79] M.R. Barbacci, "Instruction Set Processor Specifications for Simulation, Evaluation and Synthesis", *Proc. 17th Design Automation Conference*, 1979.

[Blackburn 85] R.L. Blackburn, D.E. Thomas, "Linking the Behavioral and Structural Domains of Representation in a Synthesis System", *Proc. 23rd Design Automation Conference*, 1985.

[Brayton 83] R.K. Brayton *et al*, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishing, 1983.

[Brayton 84] R.K. Brayton, C.T. McMullen, "Synthesis and Optimization of Multi-Level Logic", *Proc. ICCD 84*, pp 23-30, 1984.

[Brayton 86] R.K. Brayton *et al*, "Multiple-level logic optimization system" *Proc. IEEE Int. Conf. on CAD (ICCAD)*, Santa Clara, 1986.

[Coffman 76] E.G. Coffman Jr., ed., "Computer and Job Shop Scheduling Theory", Wiley 1976.

[DeMicheli 85] G. DeMicheli *et al*, "Optimal State Assignment of Finite State Machines", *IEEE Trans. CAD, July 1985.*

[Devadas 86] S. Devadas, A.R. Newton, "GENIE: A Generalized Array Optimizer for VLSI Synthesis", *Proc. 24th Design Automation Conference*, 1986.

[Devadas 87] S. Devadas, A.R. Newton, "Data Path Synthesis From Behavioral Descriptions: An Algorithmic Approach", *ISCAS*, 1987.

[Devadas 87b] S. Devadas *et al*, "MUSTANG: State Assignment of Finite State Machines targeted towards optimal multi-level logic implementation", *Proc. ICCAD 1987.*

[Fisher 81] J. Fisher,
"Trace Scheduling: A Technique for Global Microcode Compaction",
IEEE Transactions on Computers, v. C-30 No. 7, July 1981.

[Franz 86] Franz LISP Reference Manual, Opus 42
Franz Inc. of Berkeley, 1986.

[Hill 85] M.D. Hill *et al*,
"SPUR: A VLSI Multiprocessor Workstation",
Report No. UCB/CSD 86/273,
Computer Science Division, U.C. Berkeley, Dec. 1985, pp 13 and 21-24.

[Hockney 81] R.W. Hockney and C.R. Jesshope,
Parallel Computers,
Adam Hilger Ltd, Bristol, 1981.

[Horowitz 84] E. Horowitz, S. Sahni,
"Fundamentals of Computer Algorithms",
Computer Science Press, 1984.

[Hsueh 81] "Symbolic Layout Compaction",
in P. Antognetti at at eds.,
Computer Design Aids for VLSI Circuits,
pp. 175-241, Sijthoff and Noordhoff, 1981.

[Katevenis 83] M.G. Katevenis,
"Reduced Instruction Set Computer Architectures for VLSI",
PhD Dissertation, Report No. UCB/CSD-83/141,
Dept. of EECS, Computer Science Division, University of California, Berkeley.

[Kernighan 70] B.W. Kernighan, S. Lin,
"An efficient heuristic procedure for Partitioning graphs",
The Bell Syst. Tech. journal 49:2, pp. 291-307.

[Kernighan 78] B.W. Kernighan, D.M. Ritchie,
"The C Programming Language",
Prentice-Hall Software Series, 1978.

[Kirkpatrick 83] Kirkpatrick, D. Gelatt, M. Vecchi,
"Optimization by Simulated Annealing",
Science, pp 671-680, 1983.

[Kogge 81] P.M. Kogge,
"The Architecture of Pipelined Computers",
Prentice-Hall, 1982.

[Krishna 86] S. Krishna, T. Hu,
"Pipeline Scheduling for MOS-VLSI datapath pipes",
EE244 class report, Fall 1986, Department of EECS, University of

California, Berkeley.

[Kurdahi 85] F. Kurdahi, A.C. Parker,
"Area Estimation for VLSI Integrated Circuits",
Technical Report CRI-85-05, EE-Systems Dept., University of Southern California, 1985.

[Landskov 80] D. Landskov, S. Davidson, B. Shriver, P.W. Mallet,
"Local Microcode compaction techniques",
Comput. Surv., v. 12, pp. 261-294, Sept. 1980.

[Leiserson 83] C.E. Leiserson, F.M. Rose, J.B. Saxe,
"Optimizing Synchronous circuitry by retiming",
Proc. third Caltech conf. on VLSI, pp. 23-36.

[Leive 81] G.W. Leive, D.E. Thomas,
"A Technology Relative Logic Synthesis and Module Selection System",
Proc. 19th Design Automation Conference, 1981.

[Mead 80] C. Mead, L. Conway,
"Introduction to VLSI systems",
Addison-Wesley, 1980.

[Metropolis 53] N. Metropolis *et al*,
J. Chem. Physics 21. 1087, 1953.

[Nagle 81] A.W. Nagle, A.C. Parker,
"Algorithms for Multiple-Criterion Design of Microprogrammed control Hardware",
Proc. 19th Design Automation Conf., 1981.

[Park 85] N. Park,
"Synthesis of High-Speed Digital Systems",
PhD Dissertation, Dept. of Electrical Engineering, University of Southern California, September 1985.

[Park 85b] N. Park, A.C. Parker,
"Sehwa: A Program for Synthesis of Pipelines",
Proc. 23rd Design Automation Conference, 1985.

[Parker 85] A.C. Parker,
"MAHA: A Program for Datapath Synthesis",
Proc. 23rd Design Automation Conference, 1985.

[Patterson 87] D.A. Patterson, P.I.,
"A SPURious Progress Report: February 1, 1987.",
U.C. Berkeley CSD Report.

[SBN 82] D.P. Siewiorek, C.G. Bell, A. Newell,
"Computer Structures: Principles and Examples",

Chap. 39 "Implementation and Performance Evaluation of the PDP-11 Family",
McGraw Hill, 1982.

[Sechen 85] C. Sechen, A. Sangiovanni-Vincentelli,
"The TimberWolf Placement and Routing Package",
IEEE J. of Solid State Circuits, vol. SC-20 no 2, pp 510-522, April 1985.

[Sherburne 84] R.W. Sherburne Jr.,
"Processor Design Tradeoffs in VLSI",
PhD. Dissertation, Report No. UCB/CSD-84/173,
Dept. of EECS, Computer Science Division, University of California, Berkeley.

[Shin 86] H. Shin, C.H. Séquin,
"Two-Dimensional Compaction by Zone Refining",
Proc. 23rd Design Automation Conference, 1986.

[Snow 78] E.A. Snow, D.P. Siewiorek, D.E. Thomas,
"A Technology-Relative Computer-Aided Design System: Abstract Representation, Transformations and Design Tradeoffs",
Proc. 16th Design Automation Conference, 1978.

[Thomas 83] D.E. Thomas *et al*,
"Automatic Data Path Synthesis",
IEEE Computer, Dec 1983.

[Weste 81] N. Weste,
"Virtual Grid Symbolic Layout", *Proc. 18th Design Automation Conference*,
pp. 225-233, 1981.

[Weste 85] N. Weste, K. Eshraghian
"CMOS VLSI Design",
Addison-Wesley, 1985.

Appendix 1 - Input Format for SP

We describe the ASCII input and output format for the stage-partitioning tool SP. We show the files for a small graph shown in Fig.3 and for the HP21-MX CPU taken from [Park 85] shown in Fig.8. The main purpose of this ASCII format is to facilitate the debugging of SP. The format is therefore easy to parse; it is not meant to serve as a refined input language.

1. Input Format for SP

The input to SP consists of LISP expressions of the form (**keyword** arguments ...). Each statement is implemented as a LISP function; the arguments can therefore be any LISP expression. There are four possible statements described below.

- The **nodes** statement

  This defines the set of operations (nodes) of the dataflow "traces" that the system can execute. The format is:

  (**nodes**

  '(name-1 delay-1 [resource-type-1])

  '(name-2 delay-2 [resource-type-2])

  ...

)

  The quotes protect the arguments from evaluation by the LISP interpreter. The resource-type is optional. This statement declares the name of each node and associates a delay and an [optional] resource-type with each node. The resource-type is used to describe resource sharing; the number of nodes with a given resource-type that are simultaneously active must not exceed the number of resources of that type available to the system (this number is specified by the **resource-limits** statement, below).

- The **resource-limits** statement

  This sets a limit on the number of resources of a particular type available to the system.

The format is:

(**resource-limits**

'(resource-type-1 limit-1)

...

)

Where limit-i is the number of units of type resource-type-i available to the system.

- The **paths** statement

serves to describe one dataflow "tracé'. The format is:

(**paths**

'trace-name

trace-probability-of-occurrence

'(dst-1 sce-1-1 sce-1-2 ... sce-1-N1)

'(dst-2 sce-2-1 ... sce-2-N2)

...

)

This statements means that there are arcs from sce-1-2 to dst-1, from sce-1-1 to dst-1, and so on. In short, the fanin arcs of dst-i are the sce-i-1, sce-i-2 , through sce-i-Ni.

- The **bit-widths** statement

optionally declares the bit-width of the arcs connecting various nodes. This statement only applies to 2-point nets that connect the output from one block to the input of one other block. The format is:

(**bit-widths**

number-1 'node-source-1 'node-sink-1

number-2 'node-source-2 'node-sink-2

...

)

where number-i is the bitwidth of the arc joining node-source-i to node-sink-i.

- The **net-widths** statement

  optionally declares the bit-width of arcs connecting various nodes. This statement applies to multi-point nets that connect the outputs from one or more blocks to the inputs of one or more blocks. The format is:

  ```
  (net-widths
     number-1 '((ns-1.1 nd-1.1) (ns-1.2 nd-1.2) ... (ns-1.N nd-1.N))
     number-2 '((ns-2.1 nd-2.1) ... (ns-2.N2 nd2.N2))
     ...
  )
  ```

  where number-J is the bitwidth of net #J, which is defined as joining the outputs of nodes ns-J.1 through ns-J.NJ to the inputs of nodes nd-J.1 through nd-J.NJ. This statement simultaneously defines the multi-point nets and their bitwidth.

2. Examples

We now show the input files to SP and the output from SP for two examples. The first one is for a small graph shown in Fig.3; the second one describes the HP21-MX CPU and is taken from [Park 85], shown in Fig.8.

The purpose of these examples is to give the reader a feel for the number and type of interactions involved in using OPD.

Contd Appendix 1 - Example from Fig.3

```
Script started on Wed Feb 24 22:22:07 1988
% cat nfig3.l
;
; This example is the one illustrated in Figure 3.
; It also appears in Appendix 1.
;

(nodes
  '(m1 100) '(m2 100) '(m3 100) '(p1 100) '(p2 100) '(d1 200)
  )
(paths
  't1 1.0 '(m2 m1 m3) '(p1 m1 m2) '(p2 m3 m2) '(d1 p1 p2)
  )
(make-sorder)
(bit-widths 1 'm2 'p2 1 'm2 'p1 1 'p2 'd1 1 'p1 'd1)
(net-widths 1 '((m3 m2) (m3 p2)) 1 '((m1 m2) (m1 p1)))

% lisp
Franz Lisp, Opus 43.1 [sun-20.5]
(C) Copyright 1985,1986,1987 Franz Inc., Alameda Ca.
=> (load 'nfig3)
;; Loading file "nfig3.l"
gen-node: WARN - node S delay=0
gen-node: WARN - node E delay=0
t
=> (setq partition (opt0 200))
-- NOTE: 200 is the target stage length.
(wall2 wall3 wall4 wall5)
=> (opt1 partition)
curcost = 202
202
=> (mapc 'print-wall partition)
-- NOTE: wall2 is a stage latch; the format is:
-- NOTE: arc-name source-node sink-node
WALL wall2:
        arc19.sorder    S->m3
        arc17.sorder    S->m1
WALL wall3:
        arc8.sorder     m1->p1
        arc12.sorder    m3->p2
        arc13.sorder    m3->m2
        arc9.sorder     m1->m2
WALL wall4:
        arc14.sorder    p1->d1
        arc15.sorder    p2->d1
WALL wall5:
        arc21.sorder    d1->E
(wall2 wall3 wall4 wall5)
=> (print-part partition)

Stage wall2->wall3
        (m3 m1)
```

```
Stage wall3->wall4
      (p1 p2 m2)

Stage wall4->wall5
      (d1)
((wall2 wall3) (wall3 wall4) (wall4 wall5))
=> (exit)
% ^D
script done on Wed Feb 24 22:24:02 1988
```

Contd Appendix 1 - HP21-MX CPU Example (Fig.8)

```
Script started on Mon Nov 23 10:44:03 1987
%
%
% cat hp21mx.l
-- NOTE: This is the HP21-MX Example. See Fig.8.

;
; dataflow graph for the HP-21MX CPU
; See also Fig.8 please.
; From N. Park's Thesis, p. 82
; T1 corresponds to non-branch microcycles;
; T2 describes branches.
;

(nodes
 '(S 0)
 '(A 10)
 '(B 70)
 '(C 15)
 '(D 20)
 '(F 15)
 '(G 15)
 '(H 20)
 '(I 25)
 '(J 65)
 '(K 20)
 '(L 10)
 '(M 50)
 '(E 0)
 )

(paths
 'T1        ; non-branch microcycles
 0.1
 '(A S)
 '(B A)
 '(C B)
 '(D B C)
 '(F B C)
 '(G B C)
 '(H B C)
 '(I D)
 '(J F I)
 '(K G J)
 '(L H K)
 '(E L)
 )

(paths
 'T2        ; branch microcycles
 0.9
 '(A S)
```

```
'(B A)
'(C B)
'(D B C)
'(F B C)
'(H B C)
'(M F D)
'(L M H)
'(E L)
)


%
%
%
% lisp
Franz Lisp, Opus 43.1 [sun-20.5]
(C) Copyright 1985,1986,1987 Franz Inc., Alameda Ca.
=> (load 'hp21mx]
;; Loading file "hp21mx.l"
gen-node: WARN - node S delay=0
gen-node: WARN - node E delay=0
t
-- NOTE        The next statement sets up a seed partition The "wall" (stage-latch)
-- NOTE        W is created such that its fanin nodes are B and C.

=> (walls '(W B C]
((W B C))
=> (trace cost]
-- NOTE: we trace the cost function to show you how often it is called during optimization.

(cost)
=> (optl (list W]
-- NOTE: this optimizes the seed partition above
1 <Enter> cost ((wall2))
1 <EXIT>  cost  155
1 <Enter> cost ((wall2))
1 <EXIT>  cost  225
1 <Enter> cost ((wall2))
1 <EXIT>  cost  155
1 <Enter> cost ((wall2))
1 <EXIT>  cost  140
curcost = 140
1 <Enter> cost ((wall2))
1 <EXIT>  cost  155
1 <Enter> cost ((wall2))
1 <EXIT>  cost  155
1 <Enter> cost ((wall2))
1 <EXIT>  cost  155
1 <Enter> cost ((wall2))
1 <EXIT>  cost  155
140
=> (optl (list W]
1 <Enter> cost ((wall2))
1 <EXIT>  cost  140
1 <Enter> cost ((wall2))
```

```
1 <EXIT> cost 155
1 <Enter> cost ((wall2))
1 <EXIT> cost 120
curcost = 120
1 <Enter> cost ((wall2))
1 <EXIT> cost 140
1 <Enter> cost ((wall2))
1 <EXIT> cost 140
1 <Enter> cost ((wall2))
1 <EXIT> cost 140
120
-- NOTE: optimized max stage length = 120ns.
=> (print-wall W]
WALL wall2:
      arc37     B->F
      arc36     B->G
      arc35     B->H
      arc42     C->F
      arc41     C->G
      arc40     C->H
      arc52     D->M
      arc44     D->I

-- NOTE: Although this is not the same Partition as in [Park 85], it is
-- NOTE: just as good (same cycle time). Fig.8 shows the partition from [Park 85].

(arc37 arc36 arc35 arc42 arc41 arc40 arc52 arc44)
=> (setq part-IV (opt0 130]
-- NOTE: we are going to calculate
(wall87 wall88 wall89)
-- NOTE:  a four-stage partition.
=> (opt1 part-IV]
      .
      .
curcost = 115
115
-- NOTE: a rough 4-stage partition (cycle = 115ns)

=> (opt1 part-IV]
      .
      .
curcost = 115
115
-- NOTE: Opt1 has now gotten stuck in a local minimum
-- NOTE:  with a cycle time of 95ns for a 4-stage
-- NOTE:  partition.
-- NOTE: so we are going to run Opt2 to escape from this local minimum.

=> (opt2 part-IV]
      .
      .
      .
110
-- NOTE: found a better 4-stage partition (cycle=110ns)
=> (opt1 part-IV]
```

```
      .
      .
      .
95
-- NOTE: optl has a still better 4-stage partition.
=> (optl part-IV]
      .
      .
      .
95

=> (mapc 'print-wall part-IV]
WALL wall87:
      arc33    S->A
WALL wall88:
      arc41    C->G
      arc36    B->G
      arc40    C->H
      arc35    B->H
      arc42    C->F
      arc37    B->F
      arc43    C->D
      arc38    B->D
WALL wall89:
      arc53    F->M
      arc52    D->M
      arc46    G->K
      arc47    H->L
      arc48    I->J
      arc45    F->J
(wall87 wall88 wall89)
=>
Exiting...
%
script done on Mon Nov 23 11:06:35 1987
```

Appendix 2 - PA input format, examples

1. Tool Input Description

The PA input format is meant to be an easily parsed description syntax. Each statement is a LISP expression and is implemented by a Franz LISP function; the arguments can therefore be arbitrary LISP expressions.

The input format consists of five items: a clock specification, a description of the blocks and latches, a description of the connectivity among blocks, a set of constraints, and a description of which latches are actually used for pipeline staging (MOS designs sometimes use latches to temporarily hold results; these latches do not define stage boundaries).

1.1. Clock Specification

A clock cycle is assumed to consist of a repeating set of phases. The length of each phase is specified via the **phases** keyword:

(**phases** length-of-phase-1 ... length-of-phase-N)

Where the cycle has N phases. The time units used have no intrinsic meaning to the tool. However, the iterative optimization algorithms use a basic increment equal to one time unit for exploring alternatives. With a finer unit, we can therefore obtain greater precision at the expense of increased run times.

1.2. Block Description

For each block, we specify three items: a name, the block's type, and its propagation delay:

(**blocks** '(block-name1 block-type1 block-delay1) ...)

Known types are STATIC, DYNAMIC, LATCH. A STATIC block corresponds to combinational logic or external blocks introducing delay. Such a block does not need to run on any particular phase; it starts computing outputs as soon as the input data is stable. A DYNAMIC block corresponds to dynamic or nora or domino logic gates. The phase on which such gates run must open only after all the input data has settled. In order to

distinguish true dynamic gates from domino and nora, we give the tool an extra constraint to make sure that no chained dynamic gates will be assigned to the same clock phase. A LATCH may or may not be used for staging. The difference between a LATCH and other gates is that the phase on which LATCHes run can open before the input data has settled; however, it must close only after the data is valid.

The delay associated with a block corresponds to a worst case all-pairs input to output propagation delay. Delays that depend on the particular input/output pair can be handled by mapping each block to a set of parallel "dummy" blocks, with one "dummy" per input/output pair.

The tool makes a few assumptions about the blocks. By default, gates are assumed to be able to freeze their output; this usually means there has to be an enable line or some logic on the clock. It is possible to set up constraints to model dynamic gates that loose output when precharged.

1.3. Connectivity Description

This specifies the path that a datum follows through the gates and latches in order to get the computation done. The **path** keyword is used:

(**path** '(destl scel.1 scel.2 ... scel.N1) ...)

Here, destl and scel.x are block names. The construct means that the output from blocks scel.1 through scel.N1 feed block destl. In other words, the path keyword is a convenient way to specify the connectivity graph of the datapath. This graph must be a DAG. There can be parallel paths, but there must be no loops. Parallel paths are scheduled for worst-case. It is possible to deal with shared resources and fixed loops by unrolling the loops and adding extra constraints.

1.4. Constraints Specification

The general input syntax is:

(**constraints** '(type block1 ... blockN) ...)

The tool can handle five different constraints:

- EQPHI: Equal Phase constraint.

  The syntax is (EQPHI block1 ... blockN). This constraint specifies that the given blocks should be run on the same clock phase. The blocks need not necessarily run at the same time (they can be off by an integer number of cycles). This constraint is typically used when gates share a common clock line, or when we want to force all the dynamic gates to be precharged on the same phase(s).

- FPHI: Fixed Phase constraint.

  Syntax (FPHI block-name phase-number). Specifies that the given block should be run on the given phase. This is useful for precharge schemes, as well as for handling external constraints, such as inputs arriving on a know phase. Phases are numbered starting with 0.

- NEQPHI: Non_equal phases constraint.

  Syntax (NEQPHI block1 ... blockN). Specifies that the phases assigned to the given blocks should be pairwise distinct.

- NVPHI: No_overlap "work-phases" constraint.

  Syntax (NVPHI block1 ... blockN). Specifies that the given blocks should run off a set of phases that are pairwise disjoint. This constraint is primarily used to describe resource sharing within a stage. This is also useful to make sure that there are no open paths in the stage latches, or to handle chained dynamic gates (which should not run off common phases).

- NVT: No_overlap "work-times" constraint.

  Syntax (NVT block1 .. blockN). Specifies that the work times of the given blocks should be pairwise disjoint. This constraint is used to describe resources that are shared across pipe stages. Each stage locks the resource for the duration of that stage. The work time

of a block is defined as the time during which the result from that block is needed to ensure correct operation of the pipeline. The work time of a block therefore begins when the stage latch that holds its input data opens. The work time ends when the output from the block gets effectively latched at the end of the stage. This constraint describes resource sharing between stages and can also describe unrolled loops. A shared resource RESC is a block that is used twice, generally in different pipeline stages. As Fig.25 shows, we can not model such a situation with only one copy of block RESC, since this would be interpreted to mean that RESC must wait for both stages to complete, and this would cause a loop in the connectivity graph. We need to duplicate RESC into two blocks RESC1 and RESC2, one duplicate per utilization. Since RESC1 and RESC2 are in fact implemented by the same physical gates, they can not be used simultaneously. This is what the constraint (NVT RESC1 RESC2) captures. We can then be sure that the phase assignment and schedule produced will be implementable with only one physical resource RESC.

Four of the five constraints are therefore concerned with phase assignment; the fifth, NVT, is a scheduling constraint.

1.5. Stage Latches Description

We simply input a list of latch pairs that are to be used for staging. The syntax is:

(**stages** '(stage1-latch-start stage1-latch-end) ...)

We now show the input to PA and the output from PA for two examples. The first is a small example which also illustrates PC, the phase calculation step; the second describes the section of the SPUR [Patterson 87] [Hill 85] Floating Point Unit [Adams 86] datapath which manipulates the fractional part of floating data. The SPUR FPU has about 40 blocks. We describe the FPU fraction datapath as it is used to perform the ADD instruction. Since the FPU is a datapath chip with few opportunities for scheduling optimizations, the resulting timing is very close to that used by the designers. There is a layer of logic

between the clocks and the latch controls in the FPU. It is therefore potentially possible to change the phase assignment dynamically, to adjust to the particular instruction or instruction mix being executed.

Contd Appendix 2 - Tutorial example from Fig.4

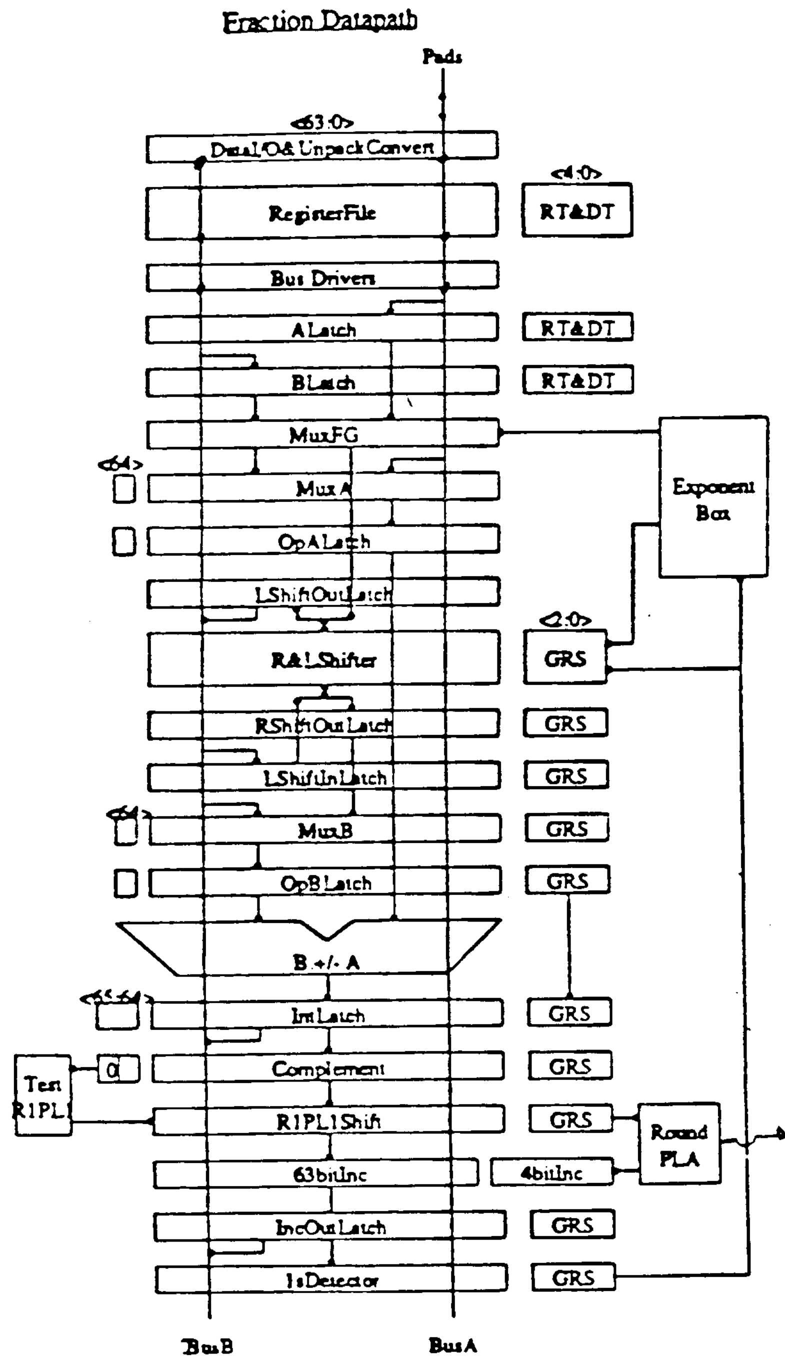
```
Script started on Sun Nov 22 19:50:41 1987
% cat nfig4.l
;
; Please refer to Fig.4
; This example illustrates Both SP and PA/PC
; on the system show in Fig.4
;
(nodes
 '(S 0)
 '(A1 1 rl)
 '(A2 1 rl)
 '(A3 1)
 '(B1 1)
 '(B2 2)
 '(B3 1)
 '(B4 1)
 '(C1 2)
 '(C2 2)
 '(C3 2)
 '(C4 2)
 '(E 0)
 )
(resource-limits
 '(rl 1)
 )
(paths
 'T1
  1.0
 '(A1 S)
 '(A2 S)
 '(A3 S)
 '(B1 A1)
 '(B2 A1 A2)
 '(B3 A2 A3)
 '(B4 A3)
 '(C1 B1)
 '(C2 B2)
 '(C3 B3)
 '(C4 B4)
 '(E C1 C2 C3 C4)
 )

(setq part (opt0 2))

% lisp
Franz Lisp, Opus 43.1 [sun-20.5]
(C) Copyright 1985,1986,1987 Franz Inc., Alameda Ca.
=> (load 'nfig4)
;; Loading file "nfig4.l"
gen-node: WARN - node S delay=0
gen-node: WARN - node E delay=0
t
```

```
=> (setq part (opt0 2))
(wall7 wall8 wall9 wall10)
=> (opt1 part)
curcost = 2
2
-- NOTE: Best SP Partition cycle time = 2
=> (mapc 'print-wall part)
WALL wall7:
        arc18    S->A2
        arc17    S->A3
        arc21    A1->B1
        arc20    A1->B2
WALL wall8:
        arc23    A2->B2
        arc20    A1->B2
        arc26    B1->C1
        arc28    B3->C3
        arc29    B4->C4
WALL wall9:
        arc27    B2->C2
        arc30    C1->E
        arc32    C3->E
        arc33    C4->E
WALL wall10:
        arc33    C4->E
        arc32    C3->E
        arc30    C1->E
        arc31    C2->E

(wall7 wall8 wall9 wall10)

-- NOTE: We now run PC to break up this cycle=2units into phases.
-- NOTE: the following step produces the FOLDED graph as shown in Fig.4

=> (go-phase-calc part]
data path file name ?
-- NOTE: the datapath is provided by the designer.
nfig4.dp.l
(setq datapath-file 'nfig4.dp.l)
mini #phases ?
-- NOTE: We give one as minimal number of phases.
1
temp file for phase parms ?
nfig4.phases-temp.l
-- NOTE: this file holds the FOLDED graph (see Fig.4)

% cat nfig4.phases-temp.l

(setq datapath-file 'nfig4.dp.l)
(setq GLphase-calc 't)
(setq PCminiphis '1)
(nodes
        '(S 0)
        '(A1 1 r1)
        '(A2 1 r1)
```

```
      '(A3 1)
      '(B1 1)
      '(B2 2)
      '(B3 1)
      '(B4 1)
      '(C1 2)
      '(C2 2)
      '(C3 2)
      '(C4 2)
      '(E 0)
      )

(resource-limits
      '(r1 1)
      )


(paths
      'pg
      1.0
      '(A1 S)
      '(A2 S)
      '(A3 S)
      '(B1 S)
      '(B2 S)
      '(B3 A3 A2)
      '(B4 A3)
      '(C1 S)
      '(C2 S)
      '(C3 S)
      '(C4 S)
      '(E C4 C3 C2 C1 B4 B3 B2 B1 A1)
      )


-- NOTE: we are now going to run SP on the FOLDED graph
-- NOTE:  to determine the phase lengths. See Fig.4.

% lisp
Franz Lisp, Opus 43.1 [sun-20.5]
(C) Copyright 1985,1986,1987 Franz Inc., Alameda Ca.
=> (load 'nfig4.phases-temp)
;; Loading file "nfig4.phases-temp.l"
gen-node: WARN - node S delay=0
gen-node: WARN - node E delay=0
t
=> (setq phase-part (opt0 2))
(wall2 wall3)
=> (opt1 phase-part)

-- NOTE: SP on the FOLDED graph calls PA to actually schedule the
-- NOTE: datapath with the phase length sequence under evaluation.

phase-calc-cost: (exec.refine nfig4.dp.l 2)
```

```
I> PAchk [clock.l]: CPU time for preprocessing: 0.69 seconds
;; Loading file "result.refine.l"
phase-calc-cost 7
7
-- NOTE: this is the result from PA.
        .
        .
        .

-- NOTE: The best split found is with just one phase.

=>
Exiting...
%
script done on Sun Nov 22 20:02:20 1987
```

APPENDIX 2
FIG A2.1: SPUR FPU "FRACTION DATAPATH"



Contd Appendix 2 - FPU Example

We now show the SPUR FPU example. What follows is the input file that describes the FPU "Fraction" Datapath, used to produce the energy curve of Fig.21.

```
;
; This version has an extra latch at the output of the 1s detector
; to make the pipeline function correctly.
;

; (flags debug_poke debug_greedy)

(phases 35 35 35 35)

(blocks

  '(pads:0 LATCH 0)
      ; input pad
  '(pads:1 STATIC 0)
      ; output pad
      ; no NVT constraint here: they are different pins.

  '(RegDecoder DYNAMIC 9)

  '(DataI/O&UnpackConvert:0 STATIC 48)
      ; copy for input data unpacking
  '(DataI/O&UnpackConvert:1 STATIC 48)
      ; for packing output result

  '(RegisterFile:0 LATCH 15)
      ; copy for writing out data
  '(RegisterFile:1 LATCH 15)
      ; for reading back result

  '(BusDriver:0 STATIC 9)
      ; to load in data
  '(BusDriver:1 STATIC 9)
      ;  write back result

  '(BusA:0 STATIC 0)

  '(BusB:0 STATIC 0)
      ; copy fed by input bus driver
  '(BusB:1 STATIC 0)
      ; copy fed by incrementer output latch
  '(BusB:2 STATIC 0)
      ; copy used for writing back result

  '(ALatch LATCH 5)
  '(BLatch LATCH 5)
  '(MuxFG STATIC 12)

  '(ExponentBox:0 STATIC 120)
```

```
'(ExponentBox:1 STATIC 10)
    ; ; fed by onés detector.
    ; Shorter delay since already set up.

'(MuxA STATIC 12)
'(OpALatch STATIC 5)

'(R&LShifter:0 DYNAMIC 16)
    ; Right Shift for aligning fractional part
'(R&LShifter:1 DYNAMIC 16)
    ; Left Shift for normalizing fractional part

'(RShiftOutLatch LATCH 5)
'(MuxB STATIC 12)
'(OpBLatch LATCH 5)
'(Adder STATIC 37)
'(IntLatch LATCH 20)
'(Complement STATIC 5)
'(TestR1PL1 STATIC 10)
'(R1PL1Shift STATIC 12)
'(RoundPLA STATIC 10)
'(63bitInc STATIC 37)
'(IncOutLatch LATCH 20)
'(1sDetector DYNAMIC 20)
'(DetectorLatch LATCH 5)
'(LShiftInLatch LATCH 5)
'(LShiftOutLatch LATCH 20)
)

(path
'(RegDecoder pads:0)
'(DataI/O&UnpackConvert:0 pads:0)
'(RegisterFile:0 RegDecoder DataI/O&UnpackConvert:0)
'(BusDriver:0 RegisterFile:0)
'(BusA:0 BusDriver:0)
'(BusB:0 BusDriver:0)
'(ALatch BusA:0)
'(BLatch BusB:0)
'(MuxFG ALatch BLatch ExponentBox:0)
'(ExponentBox:0 pads:0)
'(MuxA MuxFG BusA:0)
'(OpALatch MuxA)
'(R&LShifter:0 MuxFG ExponentBox:0)
'(RShiftOutLatch R&LShifter:0)
'(MuxB BusB:0 RShiftOutLatch)
'(OpBLatch MuxB)
'(Adder OpALatch OpBLatch)

;
; This concludes the section of the pipeline that feeds the adder.
; The following part gets the intermediate result and loads it back into
; the register file or the pads.

'(IntLatch Adder)
'(Complement IntLatch)
```

```
'(TestR1PL1 Complement)
'(R1PL1Shift TestR1PL1 Complement)
'(RoundPLA R1PL1Shift)
'(63bitInc RoundPLA R1PL1Shift)
'(IncOutLatch 63bitInc)
'(BusB:1 IncOutLatch)
'(1sDetector BusB:1)
'(DetectorLatch 1sDetector)
'(ExponentBox:1 DetectorLatch)
'(LShiftInLatch BusB:1)
'(R&LShifter:1 LShiftInLatch DetectorLatch ExponentBox:1)
'(LShiftOutLatch R&LShifter:1)
'(BusB:2 LShiftOutLatch)
'(BusDriver:1 BusB:2)
'(RegisterFile:1 BusDriver:1 RegDecoder)
'(DataI/O&UnpackConvert:1 RegisterFile:1)
'(pads:1 DataI/O&UnpackConvert:1)
)

(constraints
 ; make sure dynamic gates fire only after input has stabilized.
 '(NVPHI RegDecoder pads:0)
 '(NVPHI R&LShifter:0 ExponentBox:0)
 '(NVPHI R&LShifter:0 MuxFG)
 '(NVPHI 1sDetector BusB:1)
 '(NVPHI R&LShifter:1 ExponentBox:1)
 '(NVPHI R&LShifter:1 LShiftInLatch)

 '(NVT BusB:0 BusB:1)
 '(NVT BusB:0 BusB:2)
 '(NVT BusB:1 BusB:2)

 '(NVT R&LShifter:0 R&LShifter:1)
 '(NVT BusDriver:0 BusDriver:1)
 '(NVT RegisterFile:0 RegisterFile:1)
 '(NVT DataI/O&UnpackConvert:0 DataI/O&UnpackConvert:1)
 )

(stages
 '(pads:0 ALatch)
 '(pads:0 BLatch)
 '(ALatch OpALatch)
 '(BLatch OpBLatch)
 '(OpALatch IntLatch)
 '(OpBLatch IntLatch)
 '(IntLatch IncOutLatch)
 '(IncOutLatch LShiftOutLatch)
 '(LShiftOutLatch pads:1)
 )
```

We now show the output of the Phase Assignment routine PA on the SPUR FPU.

CYCLE # 0 1 2 3

PHASE # 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

INPUT LATCH CLOSES

REG FILE FIRES

A-LATCH & B-LATCH CLOSE

R&L-SHIFTER FIRES

OPA-LATCH, OPB-LATCH, RSHIFTOUT-LATCH CLOSE

ADDER FIRES

INT-LATCH CLOSES

COMPLEMENTER FIRES

63-BIT INC FIRES

INCOUT-LATCH CLOSES

1SDETECTOR FIRES

DETECTOR-LATCH CLOSES

R&L-SHIFTER FIRES

LSHIFTOUT-LATCH CLOSES

The raw text output from PA for the FPU "Fraction" Datapath follows.

Block________Fire Time

The Fire Time is in the form: cycle#.phase#.offset time

pads:0________-1.3.35
RegDecoder________0.0.0
DataI/O&UnpackConvert:0________0.0.0
ExponentBox:0________0.0.0
RegisterFile:0________0.1.35

BusDriver:0________0.2.15
BusB:0________0.2.24
BusA:0________0.2.24
ALatch________0.2.24
BLatch________0.2.24
MuxFG________0.3.15
MuxA________0.3.27
R&LShifter:0________1.0.0
OpALatch________1.0.4
RShiftOutLatch________1.0.16
MuxB________1.0.21
OpBLatch________1.0.33
Adder________1.1.3
IntLatch________1.2.5
Complement________1.2.25
TestR1PL1________1.2.30
R1PL1Shift________1.3.5
RoundPLA________1.3.17
|63bitInc|________1.3.27
IncOutLatch________2.0.29
BusB:1________2.1.14
|1sDetector|________2.2.0
LShiftInLatch________2.1.14
DetectorLatch________2.2.20
ExponentBox:1________2.2.25
R&LShifter:1________2.3.0
LShiftOutLatch________2.3.16
BusB:2________3.0.1
BusDriver:1________3.0.1
RegisterFile:1________3.0.10
DataI/O&UnpackConvert:1________3.0.25
pads:1________3.2.3

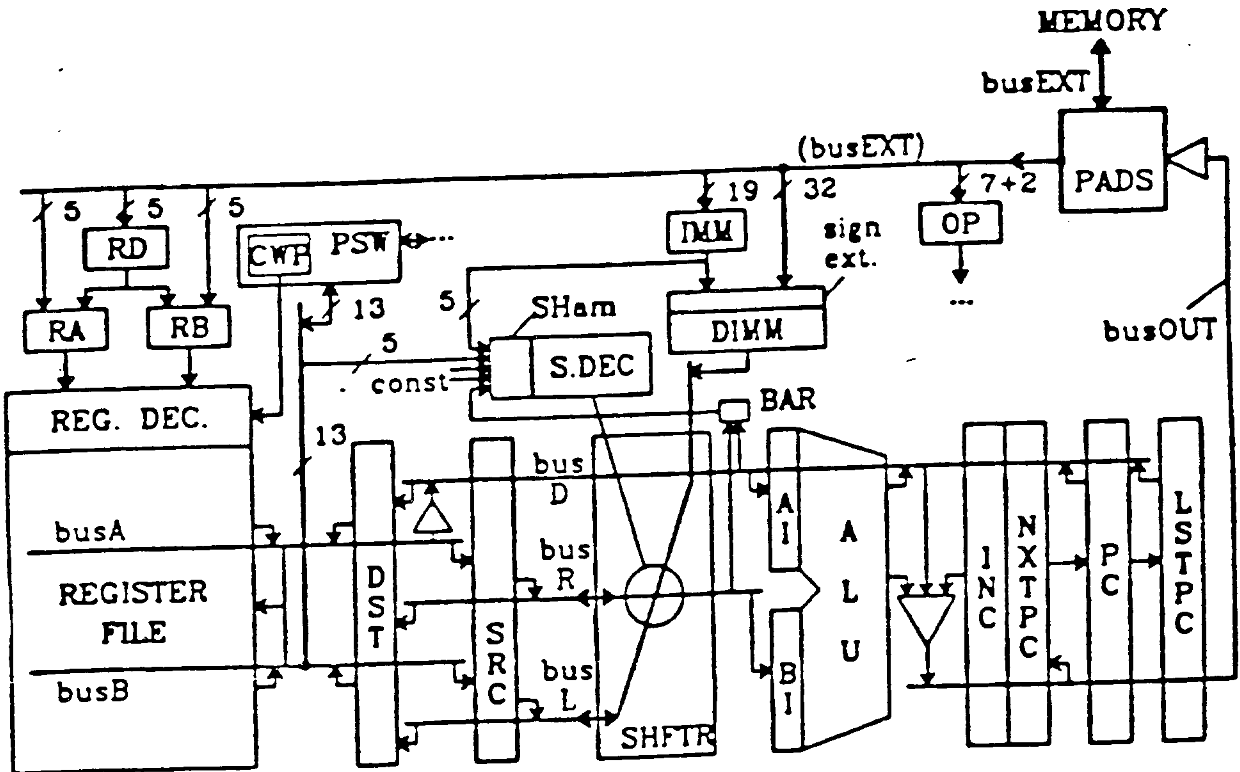
Appendix 3 - Stage Partitioning and Phase Length Calculation for the RISC-II

We first run SP on a dataflow graph based on the RISC-II CPU [Katevenis 83]. Next, we run PC with this partitioning and a RISC-II based datapath. Finally, assuming we want four phases of equal length, we calculate the best phase length to sequence the RISC-II datapath by trying out a range of values.

This example illustrates how the pipe stage lengths are related to the phase lengths. In fact, we will see that this relationship is very tenuous. Even though we may be able to find a good stage partitioning, and to do a good phase assignment with designer-chosen phase lengths, it remains difficult to calculate these phase lengths from the dataflow graph and the datapath.

The RISC-II example has about 30 blocks. In this case, using the designer's phase length choice, PA assigned phases to datapath blocks in a way that is very close to what the designers did, the only difference being in the reference - which phase is really the *first* one ? This comes as no surprise, because the designers of the RISC-II chose the phase lengths so as to satisfy the timing dependencies of the chip in an optimal fashion (see [Katevenis 83] chap. 4.2 pp 102-112). By trying out different lengths for the phases, we were able to confirm that the chosen length for each one of the four phases - 120 nanoseconds - is very close to the optimal length. The pipe throughput decreased for other phase lengths. In fact, we found the optimum to be at 110ns, but the small difference - less than 10% - can be accounted for by uncertainty on delay figures for the blocks.

APPENDIX 3: RISC-II DATAPATH
[KATEVENIS 83]



Contd Appendix 3 - RISC-II Example

```
Script started on Mon Nov 23 11:29:43 198
%
%
% cat r21-dfg.l
;
; Example inspired by the RISC-II Micro-architecture
; (cf. M. Katevenis' Thesis).
;
; This version has only one datapath for all instructions,
; and therefore
; covers the worst-case paths.
;
; This is the DFG corresponding to the actual datapath in r21.l
;

(nodes
 '(S 1) '(E 1)
 ;
 ; First I-Fetch
 ;
 '(nxtpc        1)
     ; PC for instr access
 '(dmux_0       1 DMUX)
     ; latching multiplexor for memory address
 '(pads_0       70)
     ; delay to pins
 '(mem_0        300 MEM)
     ; Memory behaves like a dynamic_ fixed work phases
 '(ir           1)
     ; Instruction Register - abstraction for RD&IMM&OP
 ;
 ; Register File components
 ;
 '(rr-dec       90)
     ; Register Decode - number of reg accessed for read
 '(rw-dec_0          90)
     ; Register Decode - number of reg accessed for write
 '(rw-dec_1          90)
     ; Register Decode - number of reg accessed for write
 '(rr           160)
     ; Reg Read+precharge
 '(rw_0         140 RW)
     ; Reg Write - r/r instr
 '(rw_1         140 RW)
     ; Reg Write - load
 ;
 ; ALU and surrounding latches
 ;
 '(alu               170 ALU)
 '(alu-dum_0    170 ALU)
 '(alu-dum_1    170 ALU)
 '(ai           1)
```

```
'(bi            1)
'(dst_0         1 DST)
    ; DST used to hold ALU output
'(dst_1         1 DST)
    ; DST used to hold data from memory to write in RF
'(src               1)
    ; to hold reg file output
;
; Dynamic Shifter
;
'(shift_0       40 SHIFT)
    ; Shifter used for a shift instruction
'(shift_1       40 SHIFT)
    ; Shifter used to route immediates
'(shift_2       40 SHIFT)
    ; Shifter used to align data from memory
;
; LOAD Instruction
;
'(dmux_1        1 DMUX)
'(pads_1        70)
    ; delay to pins
'(mem_1         300 MEM)
'(dimm          1)
    ; to store data from memory
    ; see shift_2, dst_1
;
; STORE Instruction
;
)
(resource-limits
'(MEM 1)
'(ALU 1)
'(DST 1)
'(DMUX 1)
'(RW 1)
'(RFP 1)
'(SHIFT 1)
)

(paths
'T1
1.0
;
; First Ifetch
;
'(dmux_0 nxtpc) '(pads_0 dmux_0)
'(mem_0 pads_0) '(ir mem_0)
;
; Reg File to ALU
;
'(rr-dec ir)
'(rr rr-dec)
    ; Read involved regs - address for LD/ST
'(src rr)
```

```
     ; temp latch
'(ai rr)
'(bi rr shift_1)
'(alu ai bi)
'(shift_1 ir )
     ; immediates routing
;
; LOAD instruction
;
'(dmux_1 dst_0)
'(pads_1 dmux_1)
'(mem_1 pads_1)
'(dimm mem_1)
'(shift_2 dimm)
'(dst_1 shift_2)
;
; STORE instruction
;
'(dst_0 alu shift_0)
;
; SHIFT instruction
;
'(shift_0 ir  src)
;
; Dest Write into RegFile - r/r instr
;
'(rw-dec_0 ir)
'(rw_0 dst_0  alu-dum_0)
'(alu-dum_0 rw-dec_0)
;
; Dest write into regfile - load instr
;
'(rw-dec_1 ir)
'(rw_1 alu-dum_1 dst_1)
'(alu-dum_1 rw-dec_1)
)

%
%
```

-- NOTE: we are now going to run SP on the RISC-II Data Flow Graph.

```
% lisp
Franz Lisp, Opus 43.1 [sun-20.5]
(C) Copyright 1985,1986,1987 Franz Inc., Alameda Ca.
=> (load 'r2l-dfg]
;; Loading file "r2l-dfg.l"
t
=> (setq part (opt0 450]
(wall2 wall3 wall4 wall5 wall6)
=> (opt1 part]
curcost = 671
curcost = 591
591
=> (opt1 part]
```

```
curcost = 501
curcost = 423
423
=> (opt1 part]
curcost = 422
422
=> (opt2 part]
curc = 421
421
=> (opt2 part]
421
=> (opt1 part)
421
-- NOTE: best partition. Cycle time = 421ns.
=> (mapc 'print-wall part]

WALL wall2:
        arc63    S->nxtpc
WALL wall3:
        arc37    ir->shift_0
        arc35    ir->rw-dec_1
        arc36    ir->rw-dec_0
        arc39    ir->rr-dec
        arc38    ir->shift_1
WALL wall4:
        arc35    ir->rw-dec_1
        arc36    ir->rw-dec_0
        arc55    shift_0->dst_0
        arc46    alu->dst_0
WALL wall5:
        arc65    rw_0->E
        arc53    dst_1->rw_1
        arc42    rw-dec_1->alu-dum_1
WALL wall6:
        arc65    rw_0->E
        arc67    rw_1->E
(wall2 wall3 wall4 wall5 wall6)

-- NOTE: we are going to run PC/PA now

=> (go-phase-calc part]
data path file name ?
r21.l
(setq datapath-file 'r21.l)
mini #phases ?
4
temp file for phase parms ?
r21.phase-calc-temp.l
%
%
%
% cat r21.phase-calc-temp.l
(setq datapath-file 'r21.l)
(setq GLphase-calc 't)
(setq PCminiphis '4)
```

```
(nodes
	'(S 1)
	'(E 1)
	'(nxtpc 1)
	'(dmux_0 1 DMUX)
	'(pads_0 70)
	'(mem_0 300 MEM)
	'(ir 1)
	'(rr-dec 90)
	'(rw-dec_0 90)
	'(rw-dec_1 90)
	'(rr 160)
	'(rw_0 140 RW)
	'(rw_1 140 RW)
	'(alu 170 ALU)
	'(alu-dum_0 170 ALU)
	'(alu-dum_1 170 ALU)
	'(ai 1)
	'(bi 1)
	'(dst_0 1 DST)
	'(dst_1 1 DST)
	'(src 1)
	'(shift_0 40 SHIFT)
	'(shift_1 40 SHIFT)
	'(shift_2 40 SHIFT)
	'(dmux_1 1 DMUX)
	'(pads_1 70)
	'(mem_1 300 MEM)
	'(dimm 1)
	)

(resource-limits
	'(DMUX 1)
	'(MEM 1)
	'(RW 1)
	'(ALU 1)
	'(DST 1)
	'(SHIFT 1)
	'(RFP 1)
	)


(paths
	'pg
	1.0
	'(E shift_0 dst_1 alu rw_1 rw_0 rw-dec_1 ir)
	'(nxtpc S)
	'(dmux_0 nxtpc)
	'(pads_0 dmux_0)
	'(mem_0 pads_0)
	'(ir mem_0)
	'(rr-dec S)
	'(rw-dec_0 S)
	'(rw-dec_1 S)
	'(rr rr-dec)
```

```
      '(rw_0 dst_0 alu-dum_0)
      '(rw_1 alu-dum_1)
      '(alu ai bi)
      '(alu-dum_0 rw-dec_0)
      '(alu-dum_1 S)
      '(ai rr)
      '(bi rr shift_1)
      '(dst_0 S)
      '(dst_1 shift_2)
      '(src rr)
      '(shift_0 src)
      '(shift_1 S)
      '(shift_2 dimm)
      '(dmux_1 dst_0)
      '(pads_1 dmux_1)
      '(mem_1 pads_1)
      '(dimm mem_1)
      )


%
%
%
% lisp
Franz Lisp, Opus 43.1 [sun-20.5]
(C) Copyright 1985,1986,1987 Franz Inc., Alameda Ca.
=> (load 'r2l.phase-calc-temp]
;; Loading file "r2l.phase-calc-temp.l"
t
=> (setq phase-part (opt0 120]
(wall2 wall3)
=> (opt1 phase-part]

-- NOTE: we have four phases of 90ns each, here.

phase-calc-cost: (exec.refine r2l.l 90 90 90 90)
I> PAchk [clock.l]: CPU time for preprocessing: 3.26666536 seconds
;; Loading file "result.refine.l"
phase-calc-cost 510

      .
      .
      .

=>
Exiting...
% lisp
Franz Lisp, Opus 43.1 [sun-20.5]
(C) Copyright 1985,1986,1987 Franz Inc., Alameda Ca.
=> (load 'r2l.phase-calc-temp]
;; Loading file "r2l.phase-calc-temp.l"
t
=> (setq part2 (opt0 200]
(wall2 wall3 wall4 wall5)
```

```
=> (opt12 part2)

-- NOTE: four phases of 170ns each.
-- NOTE: predictably, this is not a good solution.

phase-calc-cost: (exec.refine r21.1 170 170 170 170)
I> PAchk [clock.l]: CPU time for preprocessing: 3.48333194 seconds
;; Loading file "result.refine.l"
phase-calc-cost 710

        .
        .
        .

-- NOTE: phases of different lengths - not acceptable for implementation.

phase-calc-cost: (exec.refine r21.1 340 161 170 223)
I> PAchk [clock.l]: CPU time for preprocessing: 3.54999858 seconds
;; Loading file "result.refine.l"
phase-calc-cost 523
curcost = 523

        .
        .
        .

Exiting...

%
```

Appendix 4 - Control Synthesis examples

We show how pipeline control logic can be synthesized in two different styles using PA and SCHED for a small example. Our objective here is to detail the steps that lead to the definition of the control machine.

The example consists of MOS dynamic gates, which require precharging. The blocks P1 and P2 represent the precharging of gates G1 and G2, respectively. The blocks L0, L1 and L2 are the stage latches. We start with the PA step; the cycle time is 100ns, with a two-phase clock at 50ns per phase. Fig.a4.1 is a diagram of the example; the files are in Fig.a4.4.

The following pages show the result from PA; the PA step is trivial on this small example; our purpose is to describe control synthesis techniques here. We also show the output from SCHED. The resulting initiation sequence corresponds to a loop of minimal length in the pipés state graph (cf. [Kogge 81] and this report, chap. 4). We show this state graph for the example.

This pipe can use either time-stationary or data-stationary control [Kogge 81]. In Data-Stationary control, the control signals flow through the pipeline along with the data they control. The control logic is therefore trivial. We simply ship control signals to enable the gates and latches every time an initiation is made. Fig.a4.2 shows the control logic for the initiation sequence produced by SCHED. The tradeoff here is that data-stationary control requires extra latches to correctly time the flow of control signals through the pipe.

The second control style is time-stationary control. In this scheme, all the gate control and latch control signals are generated centrally. The controller therefore has to know where there is valid data in the pipe at every instant. Using this information, the controller generates the proper signals to time the flow of data through the pipe. Each exact pattern of valid data in the pipeline corresponds to one state in the pipés state diagram [Kogge 81].

FIG A4.1: A TUTORIAL EXAMPLE

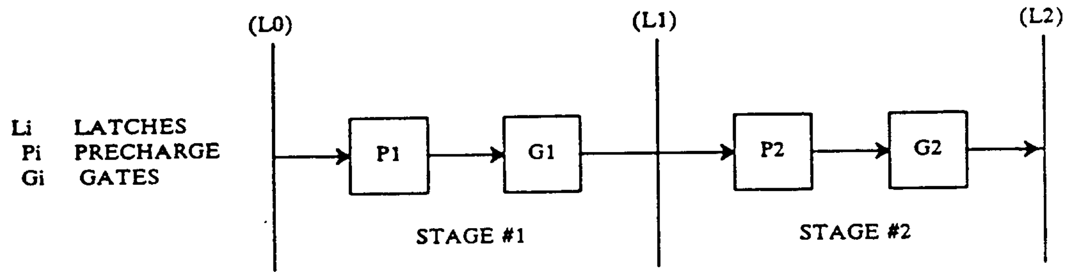
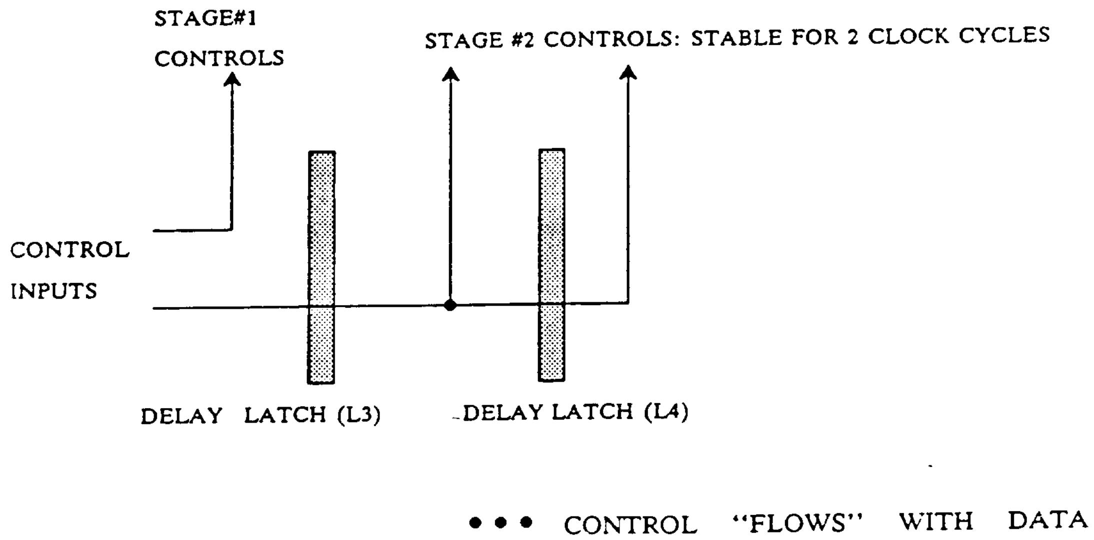
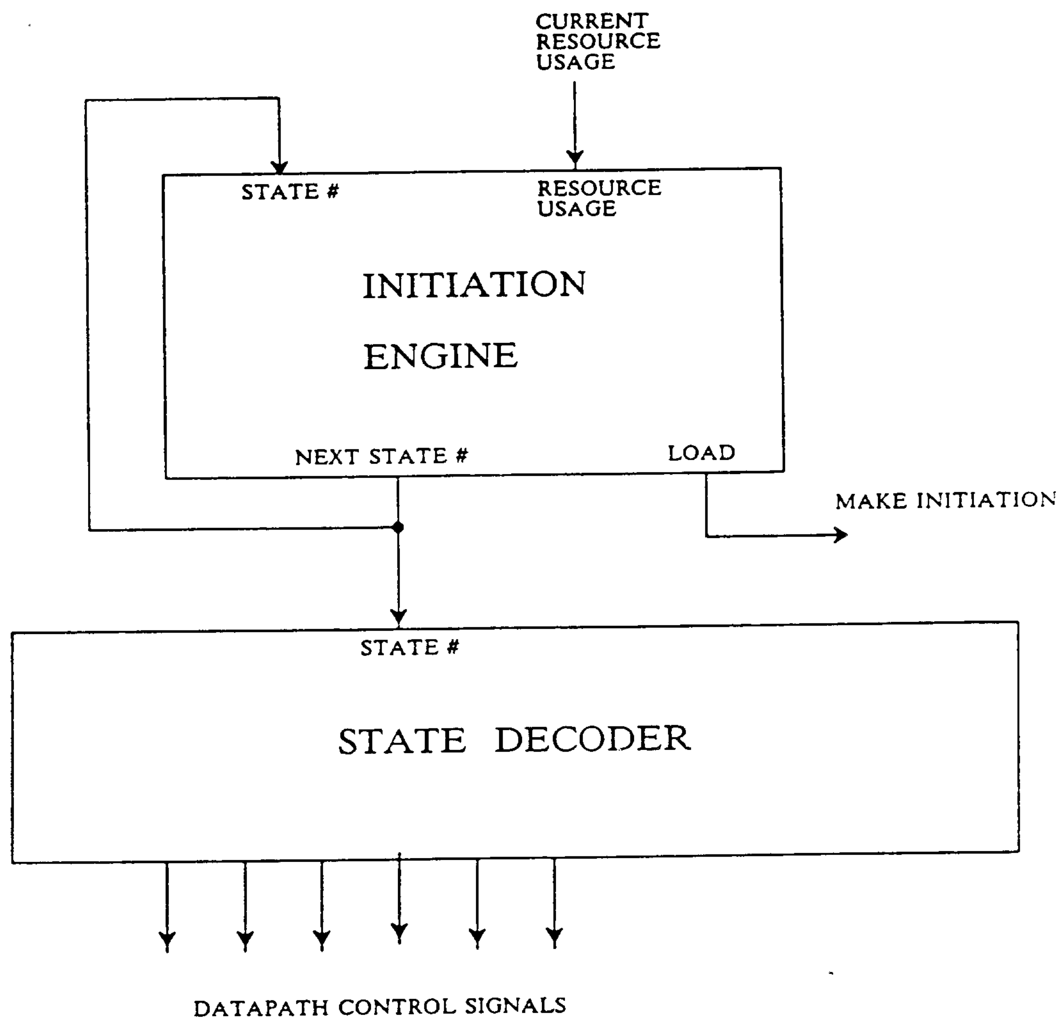


FIG A4.2: DATA-STATIONARY CONTROL



• • • CONTROL "FLOWS" WITH DATA

FIG A4.3: TIME-STATIONARY CONTROL



We can describe the controller as a two-level machine, shown on Fig.a4.3. The top-level machine is the *initiation engine*. The next machine is the *state decoder*. The initiation engine decides when new data should enter the pipe. Its output consists of two signals: one signal indicates that a new datum should enter the pipe (LOAD), and the other is the number of the new state the pipeline configuration will be entering (these are the states from the pipe's state diagram). The state decoder machine takes a pipeline state number as input. This state number is sufficient to fully specify where valid data is located in the pipe. The output of the state decoder is all the control signals required to enable the various gates and latches of the pipeline.

In the case of static [Kogge 81] pipes, the initiation engine can be a simple FSM. This FSM can be generated automatically as follows. We note that the transition diagram of this FSM is exactly the state diagram of the pipeline. We then feed the pipe's state diagram to an FSM synthesis tool such as KISS [DeMicheli 85] or MUSTANG [Devadas 87] to generate the initiation engine.

For dynamic pipes that support multiple initiation patterns, this engine could be a fairly complex microcoded machine that decides when to accept new data based on current resource usage and scheduling priorities. For instance, the Burroughs BSP Supercomputer [Hockney 81] has a very complex control engine that dynamically schedules instructions onto its vector pipeline to maximize throughput. The BSP can be viewed as a pipeline that executes instructions.

The state decoder can be viewed as a combinational logic function. Each state number corresponds to a pattern of valid data present in the pipe. The output from OPD shows what this pattern looks like, as in Fig.a4.4. This pattern is sufficient to specify the state decoder. We generate "enable" signals for a particular stage during a particular clock cycle if there is a number in the entry that corresponds to that stage/cycle pair in the valid data pattern calculated by OPD. Fig.a4.5 shows this in more detail; it also shows how we can then generate a FSM or PLA to perform the state decoding.

FIG A4.4

```
;
; Small Pipe example for control synthesis description
; The Pipe is:
;     L0 -- P1 - G1 -- L1 -- P2 - G2 -- L2
;
; where the Li are stage latches, the Pi are precharge blocks
; and the Gi are dynamic gates.
;
(phases 50 50)

(blocks
  '(L0 LATCH 0)
  '(P1 DYNAMIC 50)
  '(G1 DYNAMIC 50)
  '(L1 LATCH 0)
  '(P2 DYNAMIC 50)
  '(G2 DYNAMIC 100)
  '(L2 LATCH 0)
  )

(path
  '(P1 L0) '(G1 P1) '(L1 G1) '(P2 L1) '(G2 P2) '(L2 G2)
  )

(constraints
  '(EQPHI L0 L1 L2)
  '(NEQPHI G1 P1)
  '(NEQPHI G2 P2)
  )

(stages
  '(L0 L1)
  '(L1 L2)
  )
```

Output from PA:

```
        Block________Fire cycle.phase.offset

         L0________-1.1.50
         P1________0.0.0
         G1________0.1.0
         L1________0.1.50
         P2________1.0.0
         G2________1.1.0
         L2________2.1.0
```

Output from SCHED, the Reservation Table Scheduler:

```
    restab unit:   70
    scheduler:     ~/work/sched.hu/all -t 3 -s 2

    The reservation table being optimized:
    X--
```

-XX

The forbidden latency set contains time slot:
0 1

The range of MAL is : 2 <= MAL <= 2

The optimal sequence is as follows :
OPTSEQ 1 (2)
The order of states are:
110
OPTMAL 2

Reservation Table at each State:

State0
2-3
122
From this, we extract the steady-state reservation table:
(please refer to the text)

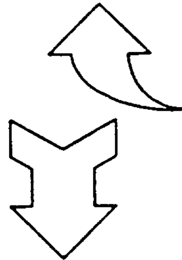
```
-----
|X|-|
-----
|X|X|
-----
```

This corresponds to the first two cycles of the table above.

FIG A4.5: TIME-STATIONARY CONTROL

X,O,*: SUCCESSIVE INITIATIONS.

CYCLE NO.	1	2	3	4	5	6	7
STAGE #1	X		O		*		
STAGE #2		X	X	O	O	*	*



CYCLE NO.	1	2
STAGE #1	O	
STAGE #2	X	O

STATE DECODER SPECIFICATION

CYCLE 1 => ENABLE STAGE#1 AND STAGE#2
CYCLE 2 => DISABLE STAGE#2, ENABLE STAGE#2
THERE IS ONLY ONE PIPE STATE HERE
THE PIPE STATE REPEATS ITSELF EVERY TWO CLOCKS

THE SPEC IS FED TO A LOGIC SYNTHESIZER