THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

OPTIMAL INSTRUCTION SCHEDULING FOR A CLASS OF VECTOR PROCESSORS: AN INTEGER PROGRAMMING APPROACH

Siamak Arya

CRL-TR-19-83

APRIL 1983

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

OPTIMAL INSTRUCTION SCHEDULING FOR A CLASS OF VECTOR PROCESSORS: AN INTEGER PROGRAMMING APPROACH

by
Siamak Arya

A Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer, Information, and Control Engineering)
in The University of Michigan
1983

Doctoral Committee:
Professor Donald A. Calahan, Chairperson
Assistant Professor John R. Birge
Assistant Professor Trevor N. Mudge
Professor Norman R. Scott
Associate Professor Toby J. Teorey

ABSTRACT

OPTIMAL INSTRUCTION SCHEDULING FOR A CLASS OF VECTOR PROCESSORS: AN INTEGER PROGRAMMING APPROACH

by

Siamak Arya

An integer programming model that portrays the architectural features of a class of vector and array processors has been developed. This model is used to produce optimal schedules for low-level-instruction codes of such processors. Optimal schedules are produced for both straight codes and instruction loops. The model is extended to optimally reassign registers to instructions in addition to instruction sequencing. The model is further extended to consider processors with multiple identical functional units. A study of the complexity of the model shows that the scheduling time increases exponentially with the number of instructions. Using the model, a number of experiments have been conducted in optimal scheduling of Cray assembly codes.

To

my father, my mother,

Anna, Flooria, and Sohrab

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the members of my committee, my family, and my friends who have provided me with their time, ideas, and support during the course of this research.

I am especially grateful to Professor Donald A. Calahan for providing me with his friendship, guidance, encouragement, and inspiration. He always found the time to helpfully respond to my questions and problems.

I would like to express my appreciation to Professors Birge, Scott, Teorey, and Mudge for the care and wisdom they demonstrated during the supervision of this project. My special gratitude is extended to Professor James C. Bean with whom I have had many inspiring discussions.

My family has played a vital role in encouraging me to pursue a doctorate and in supporting me through out the process. I wish to express my warmest appreciation and love to them for their significant contributions to my progress.

Abundant thanks and love are extended to my friends-- Monica Robinson, Berit Ingersoll, Denise Goulet, Sadegh Moslehi, Khosrow Hadavi, and Tony Da Silva whose friendship and interest in my progress served as a source of

sustenance and enrichment. Special gratitude and love are extended to Monica who contributed indirectly by patiently adjusting her plans to accommodate me and directly by skillfully drawing the graphs for this dissertation.

The personal and intellectual support of each of these individuals has contributed to making this dissertation process an exciting and a pleasant experience.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In recent years, with the increase in available computing power, the size and complexity of scientific simulation problems have increased accordingly. In a large number of such problems, a significant portion of computation time is spent in a relatively small section of the code [AHO]. In these cases, significant efforts to reduce the computation time of such small portions of codes may be worthwhile.

Unfortunately, the optimal control of data flow associated with loops that appear simple in a high level language may be nontrivial for memory-hierarchical functionally-concurrent processors, such as the Floating Point System 164 and the CRAY-1. In many important applications, only a fraction of achievable performance is available from a high level language and one must resort to low-level-language coding to preserve the efficiency of high performance algorithm development. Indeed, timing simulators have been developed for both of these processors [CSIM] [FSIM], encouraging such programming by algorithm developers. Even with such aids, one is often left with the

likelihood that the hand-scheduling of low-level-instruction codes, especially inner loops, is suboptimal. Often one could justify considerable effort in optimal scheduling of only 10-100 low-level instructions.

A. Objective

This research is concerned with the minimization of the computation time of low-level instruction codes for the class of register-to-register vector processors (RRVP's). This class of processors is generally defined as processors a) that utilize pipelining to achieve speedups on vector (array) operations and b) for which these vector elements flow from registers through functional units (FU's) and back to registers. Among commercial processors, the CRAY family is the closest to this model, although other vector and array processors share many of the features of this model.

B. Reduced Computation Time

Sequential execution of instructions results in maximum computation time. In contrast, minimum time is reached when the maximum number of instructions are executed simultaneously throughout the computation. The latter can be achieved by using RRVP's since these processors may provide a number of FU's that can simultaneously operate on different instructions.

Concurrent use of the FU's of RRVP's can be achieved by a careful arrangement of the instructions of these

processors. Such requirements reduce the ability of the average user of such processors to benefit from their full potential. However, that reduction in benefits may be avoided by using optimal instruction scheduling methods.

To efficiently use the FU's of RRVP's, one can resort to low-level languages that provide the ability to directly manipulate the use of the FU's. Experience has shown that the amount of reduction in total computation time makes it worthwhile to code in low-level languages. Then, the challenge is to find a method that is capable of scheduling the instructions of low-level codes so that maximum FU concurrency and minimum computation time are achieved.

C. Limiting the Scope of the Problem

This research has been focused on minimization of total computation times of low-level instruction codes of RRVP's through optimal code scheduling. The reduction in total computation time can be achieved in a number of ways. Such approaches include special considerations during and after the generation phase of the code, [COHA] [NELS]. In this research, it is assumed that a code with a fixed number of instructions is available and its computation time is to be minimized by rearranging the sequence of instructions without changing either the logic or the number of the instructions.

D. A Scheduling Problem

The theory of scheduling deals with the proper sequencing of a set of jobs in the presence of a set of constraints to achieve the objective of the problem. A sequence of jobs is said to be proper if it satisfies the objective and the constraints of the scheduling problem. Two types of constraints can be imposed on a scheduling problem-- 1) precedence constraints and 2) resource constraints. Precedence constraints dictate a specific issue-time order on the pairs of jobs in the sequence, to ensure the correctness of the schedule. Resource constraints impose limits on the number of resources that are available to be used by various jobs. Hence, the theory of scheduling is the subject that deals with proper sequencing of a set of constrained jobs to satisfy a specific objective, using a constrained set of resources.

Scheduling problems can be divided into three classes of problems regarding the constraints imposed on these problems. These three classes are-- 1) scheduling with precedence constraints, 2) scheduling with resource constraints, and 3) scheduling with precedence and resource constraints. The problem of this research falls under the third category due to the limitations on the number of available registers and FU's and sharing of data between instructions.

D.1. A Job-Shop Problem

Optimal scheduling of low-level instructions for RRVP's is a job-shop problem with resource constraints. Each instruction (a job) enters the CPU (job-shop), which contains a specific number of non-identical FU's (machines), and is processed by only one FU before it leaves the shop. It is a special case of a job-shop problem in which each job has only one operation. As in a job-shop problem, a set of precedence constraints must be enforced between the instructions to ensure the proper sequence of executions that produces correct final results. And, since any RRVP has a fixed number of resources that are available to the instructions, the problem is further constrained by resource limitations.

D.2. Solving Job-Shop Problems

In the literature of scheduling, a variety of methods have been introduced to solve job-shop problems. Two main approaches to optimally solve these problems are Active Schedule Generation, [THOM], and Integer Programing (IP) methods, [GREE] [PRIT]. The former approach starts with an empty schedule, considers the next available job, and creates new partial schedules accordingly. This process is repeated for each job to be scheduled. When all of the jobs are considered, a set of active schedules result. Optimal schedules are chosen from the set of active ones. The schedule generation procedure can be either enumerative

or Branch & Bound (B&B) [STIN].

The integer programming approach in solving job-shop problems is a combinatorial one that uses a set of inequalities to represent the problem. A feasible schedule is one that satisfies all of such inequalities. Optimal schedules are selected from the set of feasible ones. The generation of such schedules is often achieved by combined Linear Programming (LP) and B&B techniques.

None of the existing methods can be directly applied to solve the instruction scheduling problem of this research, due principally to the number of various types of constrained resources that must be considered. Nevertheless, Baker's [BAKE] integer programming approach to solve job-shop problems without resource constraints has provided a good starting point.

E. Micro-Code Compaction

The instruction scheduling problem of this research is closely related to the problem of micro-code compaction [TRAN]. Both of these studies attempt to distinguish groups of instructions or micro-operations that can be concurrently or must be sequentially processed. This information is used to maximize the concurrent use of the resources of vector- and micro-processors in order to reduce computation time. The main difference between these problems is the issuing of instructions and micro-operations by vector- and

micro-processors, respectively. An RRVP can issue one instruction at a time, in contrast to a micro-processor that can issue a number of micro-operations at once.

In micro-code compaction, data- and resource-independent micro-operations[1] are grouped into one micro-instruction and are issued simultaneously. In instruction scheduling, no more than one instruction can be issued at any clock-time. And, if two or more instructions compete for a specific issue time, the issue times of all but one of these instructions will be delayed. The issue time delay for some instructions can play a significant role in the arrangement of the rest of the instructions.

The number of papers that have recently been published on micro-code compaction shows a sign of growing interest in this optimization problem. Tokoro et all [TOKO] point out that most of the present compaction methods assume one machine-cycle-time processing time for any micro-operation. They also state that micro-operation execution time in most practical machines requires more than one cycle-time, especially for memory reference operations or operations on pipelined machines.

E.1. One Cycle-Time Restriction

The restriction of one cycle-time execution-time for

---

[1]Instructions and micro-operations are the lowest-level-language commands that can be issued by vector- and micro-processors, respectively.

each operation provides a simplified version for the compaction problem. Since each resource is reserved for only one cycle-time, all of the resources are available to the micro-operations at the beginning of the next cycle. Therefore, it is sufficient to find the operations that have no precedence constraints or have already satisfied this condition, and combine them into a micro-instruction if they do not require the same resource.

E.2. Elimination of the Restriction

To eliminate this restriction, Tokoro et al assume that operations can take longer than one cycle-time from the time of issue to the time of termination of execution. Therefore, to combine micro-operations, these operations should not require a resource which is reserved by a previous operation, in addition to the previous conditions that should be satisfied. The latter condition is stated as a set of "either or" inequalities. If these inequalities are satisfied, that is if the FU required by the operation is not reserved by a previous one, the operations can be combined with the ones that are currently ready to be issued. Otherwise, the issue time of the operation is delayed. This operation, then, can be combined with the succeeding operations when the above conditions are satisfied. This version of micro-code compaction is closest to the problem of this research, since the one cycle time execution time does not hold for RRVP's.

F. Problem Specification

This research views the problem of scheduling low-level instructions for RRVP's as follows.

> The problem is to assign proper issue times to the low-level instructions of RRVP's so that-- 1) total computation time is minimized, 2) precedence constraints are satisfied, and 3) occurrences of resource conflicts are eliminated.

G. The Model

To produce optimal schedules for low-level codes of RRVP's, an integer programming model has been developed. The model of this research is capable of precisely predicting the behavior of such processors and simplifying the complicated job of managing their resources. Resource-management complications arise from the resource-requirement conflicts that occur between the instructions; Chapter II provides a detailed description of these conflicts. The total computation time is dependent on how well these conflicts are resolved. To resolve these conflicts, each instruction is modeled by a set of inequalities that should be satisfied to maintain its relationship with the other instructions while avoiding any conflicts that can increase the computation time. A feasible schedule is one that satisfies all of these inequalities. An optimal schedule is a feasible one with

minimum computation time. Then, the outcome of the model is a set of specific start times at which instructions must be started to realize the minimum computation time.

In addition to straight-code scheduling[2], optimal instruction-loop scheduling is also included in the model of this research. Scheduling of instructions in a loop differs from straight-code scheduling in that the set of instructions of the loop execute repeatedly and the relationships between the instructions of different repetitions must be considered to produce optimal loop schedules. These relationships play a significant role in loop scheduling for processors with more than one pipelined functional units and vector instructions. If none of these resources exist, the execution of each instruction is started after the termination of execution of its preceding one. Therefore, intra-loop relationships between instructions are not important and the attention is focused on reducing loop overhead during code generation [AHO] to reduce computation time.

H. Register Re-assignment

The optimal schedule for the instructions in a code that is produced by applying the model of Chapter II is optimal relative to a specific order of register-

---

[2]Straight-code scheduling refers to scheduling of instructions in a basic-block [AHO], that is a piece of code that can be entered only at the beginning and does not include any branching instructions or entry points.

assignments. Because a limited number of registers are available, each register must be used by a number of instructions. The sharing of registers by instructions can force a delay in the execution of instructions that must wait for specific registers to become available. Such delays that can increase the computation time may be avoided by a change in the register-assignment sequence. Thus, the model of this research is extended to optimally reassign registers to instructions in accordance with optimal instruction sequencing. Hence, the extended model produces optimal-schedules with respect to register assignment and instruction sequencing. The register-reassignment portion of the model is described in Chapter III.

Register reassignment of this research differs from the register allocation problem in that the latter is usually a compile time problem to reduce excessive use of the storage. Register allocation studies are mainly concerned with the use of registers to reduce the transfer of data to and from main memory, [FREI] [DAY] [BEAT], in order to reduce the size of the code and/or the use of the memory to minimize the computation time. Study of the minimum number of required registers to avoid memory transfers [SETH] is also categorized as as register allocation problem. However, the allocation problem of this research is concerned with changing the original assignment of registers to instructions so that concurrent execution of higher number of instructions are made possible.

I. Multiple-Identical FU's

The model is further extended to deal with the case in which duplicate copies of one or more of the FU's are added to the configuration of the processor. In other words, the processor contains m groups of non-identical FU's where each group is composed of p identical FU's. For $p \leq 2$, the extension minimally alters the model. However, for $p > 2$, a more complicated extension is needed. This subject is discussed in Chapter IV.

Scheduling jobs on identical machines has received a great deal of attention in the literature of scheduling. This study is mainly concentrated on the scheduling of independent jobs or dependent ones with identical execution times [BAKE] [SAHN]. The problem of this research is a more general one. It is the study of scheduling jobs on identical machines in the presence of precedence constraints where the execution times of instructions are not identical. Again, the goal is to find optimal schedules.

J. Complexity of the Model

The problem of optimal scheduling of low-level instructions of RRVP's is NP-complete. That is due to the nature of the problem (job-shop) and use of integer programming. This research has shown that for optimal scheduling of these codes, use of integer programming

cannot be avoided. A study of the complexity of the number of variables and inequalities required for modeling such codes has been conducted to help realize the complexity of time and storage requirements for the implementation of the model. Chapter V deals with these subjects.

K. Implementation of The Model

The IP model of this research has been used in optimal scheduling of a number of Cray assembly codes. The scheduling process is composed of three phases: (a) generating the inequalities that model the code, (b) solving the inequalities using an IP, and (c) rearranging the order of instructions according to the optimal sequence. A program that is capable of producing the inequalities of phase (a) has been developed. Using this program and an IP, a number of scheduling experiments have been conducted. The results of the experiments conducted by this research are reported in Chapter VI.

CHAPTER II

THE MODEL

The scheduling method to be introduced is a general one that can be applied to numerous pipelined RRVP's. The method formulates the sharing of the resources of such processors by their instructions. In turn, this formulation provides a model that can produce optimal schedules for low-level instruction codes of these processors.

To model RRVP's, a detailed description of their architectural features is necessary. An RRVP is specified as a vector processor in which each FU receives input from one or more registers and sends its output to a register. In addition, in an RRVP, each instruction uses at least one register. These general descriptions that identify a class of RRVP's do not provide sufficient information for their modeling. For a detailed modeling of these processors, the following specifications of the resources of RRVP's and the restrictions applied to the control of these resources are necessary. These specifications are derived from the CRAY-1 architecture.

A. Architectural Features

The following is a list of resources that are included in the model.

1- 1 instruction issuer to all Functional Units.

2- N register groups. For the convenience of referencing, three special register groups are specified as Address (A), Scalar (S), and Vector (V) registers.

3- P_i identical registers in register group i, for i=1,2,...,N. P_i is a positive integer, and $P_i=P_j$ for some i,j=1,2,...,N is permitted.

4- M pipelined non-identical FU's, including the memory. Each pipeline stage is one clock long; one clock is equivalent to one time unit.

5- A set of access paths connecting FU's to registers.

The following general restrictions apply to the control of these resources.

1- At most one instruction can be issued at any clock period.

2- Instructions are issued in a sequential order. i.e. no instruction can be issued before all of the instructions that precede it in the sequential order of the instructions have been issued.

3- An instruction can be issued only if the registers and the FU required by the instruction have not been reserved by previously issued instructions.

4- At most one register in A register group and one

register in S register group can receive data at any clock period. Each one of these register groups has one update access path.

5- Vector instructions are the only instructions that can reserve the FU's, except for memory. The memory is reserved for any instruction type. The memory reservation period is determined by the type of the memory reference instruction.

6- Result registers are reserved until the result is received.

7- Vector-operand registers are reserved for a known period of time, depending on the vector length.

7- Each FU is dedicated to a specific group of instruction types.

9- Each instruction can be processed by only one specific FU; i.e. two instructions of the same type compete for the only FU available to that instruction type.

10- Execution of an instruction cannot be interrupted after it has been initiated; i.e. when the execution of an instruction is started, it must be fully processed without interruptions.

There are other conditions that control instruction-issue that presently are not considered in the model, such as instruction-buffer-fetch-in-progress, and instruction-not-in-instruction-buffer.

For illustration, the CRAY-1 architecture and

associated assembly language (CAL) [CRAY] will be used throughout the remainder of this dissertation.

B. The Method

The following are the definitions of some expressions that are used through out this dissertation.

Definition 1- Issue-time of an instruction I_i is the time at which all of the resources required by I_i are available and all of the instructions that precede I_i in the sequential order of the instructions have been issued.

This definition specifies a sequential order of issue for the instructions. Issue-time of an instruction I_i can be delayed by the late issue of the preceding instructions or the extension of their reservations of resources.

Definition 2- Start-of-execution-time is the time at which the operand values are transferred to the first stage of the pipelined FU for execution. For vector instructions, this time is the time that the first vector element is transferred for execution.

The start-of-execution-time that succeeds the issue-time is delayed when either the issue-time is postponed or not all of the data items are available. Two types of relationships between these two times can be identified that

are defined as follows.

Definition 3- Synchronous issue-execution is the relationship between issue-time and start-of-execution-time of an instruction in which the time difference between these times is fixed and predetermined.

Definition 4- Asynchronous issue-execution is the relationship between issue-time and start-of-execution-time of an instruction in which the time difference between these times is not fixed and is determined at the time of execution.

From Definition 3, it can be concluded that the start-of-execution-time of an instruction can be determined from its issue-time when synchronous issue-execution is enforced. However, such a conclusion is not true when asynchronous issue-execution is utilized. In the remainder of this dissertation, unless otherwise specified, synchronous issue-execution is assumed. The fixed length of time between issue-time and start-of-execution-time is assumed to be equal to the amount of time required for issuing of the instruction, usually one or two clocks.

Definition 5- Termination-time or termination-of-execution-time is the time at which the last resource reserved by an instruction is released.

Since the result register is usually reserved longer than any other resource, in order to receive the result of the computation, the termination-time is the same as the time at which the result register is freed.

Scheduling of instructions is the act of rearranging the order of instructions in their sequence and determining their issue-times. Given the issue-time of an instruction, the time that its reserved resources will be freed can be determined since the length of time that a resource is reserved is known if the instruction type and vector length are specified. Given a sequence of instructions, the issue- and termination-times of all of the instructions are determined. The total length of the computation is calculated and compared with that of other schedules. Then, the schedule with minimum total computation time is selected as the optimal one.

B.1. Inter-Instructional Relationships

Re-ordering of instructions in order to find an optimal schedule must be done so that it does not change the outcome of the computation. During the computation, the result of a calculation by an instruction I_i is usually used by some other instructions that succeed I_i in the sequential order of the code. If such an order is reversed, the result of the computation may be altered. To preserve the outcome of the computation, any change in the order of

instructions that can alter the correctness[3] of the computation must be avoided.

For optimal scheduling of instruction, a complete understanding of the relationships between the instructions of the code is necessary. The restrictions that are applied to control the resources display a number of possibilities that can delay the execution of an instruction or postpone its issue time. Such delays, in turn, will cause the postponement of the issue and execution of other instructions. Therefore, to optimally schedule a low-level code, inter-instructional relationships that affect their issue-times should be determined.

The following definitions describe the type of inter-instructional relationships that affect the re-ordering of the instructions.

Definition 6- Given two instructions I_i and I_j, where I_j succeeds I_i in the sequential order of instructions, I_j is pairwise dependent on I_i iff at least one of the operand registers of I_j is the result register of I_i or the result register of I_j is one of the registers of I_i.

The pairwise dependency relation, also stated by [TOKO] and [KUCK] in a similar manner, implies that I_j should not be

---

[3]Correctness in this context means that the final result or outcome of the code or computation is identical to the outcome of the original code.

issued until I_i has been issued and the dependency condition has been satisfied. If the issue order is reversed, it may alter the final result of the code. The pairwise dependency between two instructions is displayed by $\xrightarrow{P}$; e.g. if I_2 is pairwise dependent on I_1, then $I_2 \xrightarrow{P} I_1$.

Definition 7- Two instructions I_i and I_j are globally dependent if there exists $I_{k1}, I_{k2}, \ldots, I_{kn}$ such that $I_j \xrightarrow{P} I_{kn} \xrightarrow{P} \ldots \xrightarrow{P} I_{k2} \xrightarrow{P} I_{k1} \xrightarrow{P} I_i$

If two instructions are globally dependent, this dependency enforces a specific issue-time-order on them to preserve the correctness of the code. This relation is displayed by $\xrightarrow{G}$; e.g. if I_2 is globally dependent on I_1, then $I_2 \xrightarrow{G} I_1$.

Two instructions that are neither pairwise nor globally dependent are called pairwise-independent or simply independent and can appear in any relative order with respect to each other. The issue order for two independent instructions is determined by the resource constraints imposed on them. That means that the instruction whose required resources become available prior to the other one's should precede the other instruction in the sequential order of the code. The resource restrictions can occur due to the reservation of resources by preceding instructions or the requirement of one or more resource by both instructions where only one resource of that type is available.

B.2. Dependency Graph

The dependency relation is displayed by a directed graph. Each node of the graph represents an instruction. Each edge specifies the dependency between the two nodes connected by that edge. The direction of each edge is from the predecessor to the successor, and the value on each edge represents the amount of time that the successor must wait after the predecessor has been issued, before it can be issued.

The dependency graph is drawn from the pairwise dependencies between the instructions of the code. The first node of the graph, called the root node, is a null instruction on which any instruction is dependent with a dependency length of zero time units. The instructions in the code are, one by one, added to the graph as new nodes, in their sequential issue order. Each new node is connected to all of its preceding instructions on which it is dependent, starting from its immediate predecessor and ending at the null instruction. Any edge created by a pairwise dependency between two instructions that are already connected by an edge or a chain of edges need not be drawn. This elimination of edges is necessary to reduce the redundant connections between nodes that are already connected by a path. Hence, the root node is the immediate predecessor only to the nodes that do not have any predecessor by pairwise dependency. This node provides a

common time reference to all of the nodes in the graph and prevents the graph from being divided into a number of sub-graphs.

After all of the instructions have been added to the graph, a second null node, called the terminal node, is added to the graph in order to succeed all of the nodes that have no successors. This node represents a null instruction that is dependent on any instruction. The issue-time of the terminal node is the termination-time of the code. Then, the total computation time is the time interval between the issue times of the root node and the terminal one.

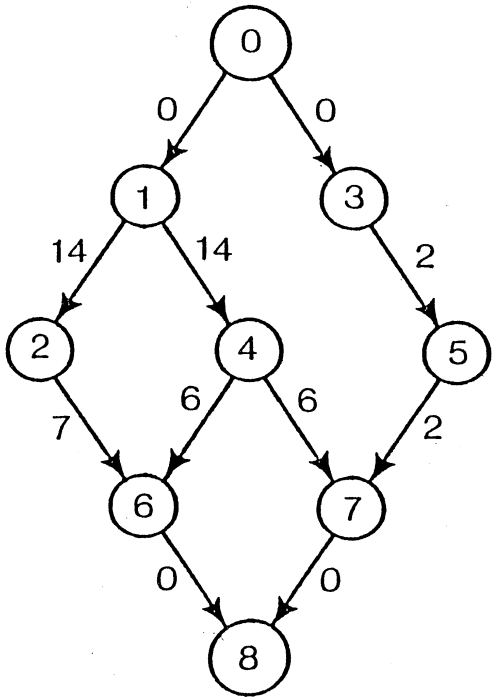
The following relationships can be determined from a dependency graph.

a- Two nodes are called adjacent if they are connected by an edge. Two adjacent nodes are pairwise dependent.

b- A directed path is a set of edges such that for each edge ending at a node, there exists one edge in the same path which starts at that node, except for the nodes that start and finish the path. Each node on the path is connected to only one succeeding node and one preceding node in the same path. Two nodes are globally dependent if there exists a directed path from one node to the other one.

c- Two nodes are called independent if they are not connected by any directed paths.

Fig. 1 shows an example of a dependency graph and



(a)

```
S1    /HS3
S4    S2*FS1
A1    A1+1
S5    S1+FS1
A3    A1+A2
S3    S5+FS4
,A3   S5
```

(b)

Fig. 1- Sample Graph

(a) dependency graph. (b) instructions set for (a).

the set of instructions that it represents. In the sample instruction set of Fig. 1b, $I_2 \xrightarrow{P} I_1$, $I_6 \xrightarrow{P} I_4$, and $I_6 \xrightarrow{P} I_2$. Therefore, in the dependency graph of Fig. 1a, nodes 2 & 1, 6 & 4, and 6 & 2 are adjacent nodes. The numbers on these edges represent the length of their dependency in time units. Also, since $I_6 \xrightarrow{P} I_2 \xrightarrow{P} I_1$, then $I_6 \xrightarrow{G} I_1$. Nodes 1 & 3, 1 & 5, 2 & 3, 2 & 4, 2 & 5, 2 & 7, 3 & 4, 3 & 6, 4 & 5, 6 & 5, and 6 & 7 are independent, because none of the pairs are connected by any directed paths.

In general, the dependency graph is neither a branching tree (only one predecessor per node) nor an assembly one (only one successor per node) ([BAKE]). It is a network structure due to the fact that each instruction is allowed more than one predecessor and/or successor.

B.3. Reflexivity, Symmetry, and Transitivity

To gain more insight into pairwise dependency, global dependency, and independency relations, the following properties of such relations have been investigated.

Reflexivity- Since the execution of an instruction is not delayed by itself, pairwise and global dependencies are not reflexive relations; i.e. no instruction is pairwise or globally dependent on itself. Since no instruction is dependent on itself, it must be independent from itself. Therefore, independency is a reflexive relation.

Symmetry- Because the pairwise and global

dependency relations are defined in one direction and such a direction cannot be reversed, these relationships are not symmetric. In other words, the symmetry property for dependency relations would imply that an instruction can depend on its preceding and succeeding instructions. However, from definitions 6 and 7 these relations are only one directional; an instruction can be dependent on its preceding instructions only. Therefore, dependency relations are not symmetric. Since no direction has been specified for the independency between two instructions, this relation is a symmetric one. It means that the independence of instruction I_i from I_j implies that I_j is also independent from I_i.

Transitivity- If instruction I_j is globally dependent on I_i and I_k is globally dependent on I_j, there exists a directed path from I_i to I_j and a directed path from I_j to I_k. Hence, there exists a directed path from I_i to I_k that includes node I_j. Consequently, I_k is globally dependent on I_i. In other words, global dependency is transitive.

Pairwise dependency relation and independency relation are not transitive. The following two examples prove the claim by introducing examples to the contrary. Using the example of Fig. 1, it can be seen that $I_7 \xrightarrow{P} I_5$ and $I_5 \xrightarrow{P} I_3$, but I_7 is not pairwise dependent on I_3. Also, I_3 is independent from I_4 and I_4 is independent from I_5, but

I_3 is not independent from I_5. Therefore, these relations are not transitive.

C. Elementary Timing Model

The mathematical scheduling model is developed from inter-instructional relationships that imply restrictions on the order and the time of issue of the instructions. The restrictions imposed on the order of instructions are due to the reservation of the registers, FU's, and other resources. The model is then constrained by other architectural features to produce the complete mathematical programming problem.

The elementary model considers only pairwise dependencies between instructions. Pairwise dependency between two instructions enforces a time distance between the issue times of those instructions, in addition to enforcement of a specific issue order. If such issue time restrictions are satisfied, the availability of proper resources at the time of issue of dependent instructions is guaranteed. This model ensures the proper order and distance of the issue times of the instructions to preserve the correctness of the computation.

C.1. Minimum Time Problem

Define the issue distance D_{ji} as the minimum time between issues of $I_j \xrightarrow{P} I_i$. Then, pairwise dependency between instructions I_i and I_j can be formulated as follows.

$$T_j - T_i \geq D_{ji} \qquad (1)$$

where T_j and T_i are issue times of instructions I_j and I_i, respectively. This formulation enforces the issue time of I_i to precede that of I_j by at least D_{ji} time units. Therefore, it preserves the issue order of dependent instructions.

At this stage of the formulation, the formal minimization criterion is the time between the issue of the first instruction and the completion of the last instruction. That is termed the 'minimum time' scheduling problem or minimization of makespan [BAKE]. Since the issue times of instructions are multiples of time units, these times must be specified by integer numbers.

The 'minimum time' problem is a linear programming problem. The problem is to minimize $T_{last} - T_{first}$ subject to a set of inequalities of type (1). This problem can be solved by using linear programming, except that the LP does not ensure integer solutions. However, since D_{ji}'s are integers for all i and j, then the T_j and T_i are guaranteed to be integers[4] as well.

C.2. Lower Bound

The scheduling of instructions can be viewed as mapping of the dependency graph onto the time axis. Since

---

[4]This claim will be justified in Integer Programming section (section H.).

the order and time distance between dependent instructions must be preserved, the minimum amount of time required to issue and execute the instructions is equal to the length of the longest path connecting the root node to the terminal node on the dependency graph. Such a path is called a critical path (CP),[WIES]. Then, the minimum time problem, in which all of the paths can be mapped onto the CP, is the problem of determining the issue times of the instructions such that the total computation time does not exceed the length of the critical path.

The minimum time problem defines a lower bound on the total computation time. Considering the restrictions to be enforced on the instruction issue by the architectural restrictions of the RRVP's, such as single instruction issue per unit time, the issue times of the instructions will be delayed with respect to their issue time in minimum time problem. Such delays cause the length of the computation to be increased. In minimum time problem, a number of instructions can be issued simultaneously. These instructions must be ordered so that the instructions belonging to the critical path are issued as soon as possible and the rest of instructions are arranged so that their execution times overlap the execution times of the instructions on the CP. In general, it is not likely to comply with the architectural restrictions of RRVP's and map all of the paths onto the CP. In other words, the issue of the instructions on the CP must be delayed beyond

their issue times in the minimum time problem due to single issue and resource constraints. Such delays increase the length of the CP and the total computation time. Therefore, the computation time of the minimum time problem is a lower bound for the general problem.

D. Scheduling With Constraints

To determine the total computation time and issue times of the instructions for the general problem, the architectural restrictions and features of the RRVP's must be added to the minimum time problem. The minimum time problem considers only the pairwise dependency relations imposed on the pairs of instructions due to the sharing of the registers. A number of other resource constraints (such as access path conflict, FU conflict, and issue conflict) also exist that should be incorporated in the model. These conflicts will be addressed in the following subsections. In addition to the resource constraints, some architectural features of the RRVP's are also considered. Such features include synchronous and asynchronous chaining and asynchronous issue-execution. These restrictions and features are formulated by a set of inequalities that can be added to those represented by (1) to make up a general model.

D.1. Access Path Conflict Constraints

If both instructions are either A or S type, the

occurrence of access path conflict must be prevented. This conflict occurs when both instructions finish execution at the same time and must deliver the result to two result registers of A or S group at the same clock. Since only one access path is available for result delivery to these register groups, the issue of one of the instructions must be delayed to prevent access path conflict. For two pairwise dependent instructions, this conflict is possible only when the number of clocks that the result register of the preceding instruction is reserved is greater than that of the succeeding instruction plus their issue time distance. Hence, to prevent the access path conflict for two pairwise dependent or two independent instructions, one of the following conditions should hold.

$$\text{or} \quad \begin{matrix} T_i + T_{R_i} - (T_j + T_{R_j}) \geq 1 \\ T_j + T_{R_j} - (T_i + T_{R_i}) \geq 1 \end{matrix} \qquad (2)$$

where T_{R_i} and T_{R_j} are the number of clocks that result registers of instructions I_i and I_j are reserved, respectively. For two independent instructions, any one of these inequalities can be chosen depending on the issue order of those instructions. For two dependent instructions, only one of the inequalities of (2) must be used since the order of these instructions is determined by their dependency and this order cannot be reversed.

D.2. Vector-Operand Conflict Constraints

If both instructions are independent and V type, vector-operand conflict can occur. According to definition 6, two independent instructions can share only their operand registers. For vector instructions , vector-operand registers are reserved and cannot be used by more than one instruction at any time. Therefore, the execution of these instructions cannot be overlapped if they share at least one vector-operand register. However, since the instructions are independent, either one can precede the other one. Hence, this type of conflict is formulated as follows.

$$\text{or} \qquad \begin{matrix} T_j - T_i \geq T_{OP_i} \\ T_i - T_j \geq T_{OP_j} \end{matrix} \qquad (3)$$

where T_{OP_i} and T_{OP_j} are the number of clocks that the operand registers of instructions I_i and I_j are reserved, respectively. Depending on the issue order of instructions I_i and I_j with respect to each other, one of these inequalities must be satisfied and the other one must be ignored.

D.3. FU Conflict Constraints

For independent instructions, FU conflict can occur for either (a) two vector instructions using the same FU, (b) a vector instruction and a scalar instruction using the same FU, or (c) a vector memory reference and any other memory reference. This conflict is similar to vector-

operand conflict, in the sense that the execution of these instructions cannot be overlapped due to sharing of their required FU. Hence, the formulation is the same as (3) with T_{OP_i} and T_{OP_j} replaced with T_{FU_i} and T_{FU_j}, the number of clocks that the FU is reserved for instructions I_i and I_j, respectively. Tokoro [TOKO] also finds it necessary to use similar inequalities to resolve FU conflicts. He uses such inequalities in his micro-code compaction algorithm when he allows the execution time of micro-operations to be longer than unit time.

D.4. Issue Conflict Constraints

Since no more than one instruction can be issued at any clock period, it must be assured that independent instructions will avoid issue conflict; i.e. the instructions should not be scheduled so that their issue is overlapped. Each instruction requires a fixed and predetermined amount of time for issue. Hence, this conflict is identical to FU or vector-operand conflict, and it is formulated similarly by replacing T_{OP_i} and T_{OP_j} in (3) with T_{ISS_i} and T_{ISS_j} that are amounts of time required to issue instructions I_i and I_j, respectively. Issue conflict cannot occur for two independent instructions with FU or vector-operand conflict since the issue-time of one of these instructions must be delayed due to the unavailability of its required resources.

D.5. Synchronous Chaining Constraints

Synchronous Chaining [CRAY] refers to the immediate transfer of data between pipelines involved in successive vector instructions. Such chaining requires that the second instruction be prepared to issue at the time that the first result emerges from the pipeline. The time at which the first result emerges is called 'chain slot' time [CRAY]. Failure to chain requires that the first vector operation be carried to completion; when the result vector is completely stored in a vector register, the second instruction can issue. Chaining is usually desirable, since both pipelines can be concurrently busied on different parts of a long vector.

Chaining is formulated as follows.

$$\text{or} \qquad \begin{matrix} T_j - T_i \geq T_{R_i} \\ T_j - T_i = T_{CHS_i} \end{matrix} \qquad (4)$$

where T_{CHS_i} is the chain slot time and T_{R_i} is the time to complete the i^{th} instruction, when chaining fails. Inequalities of (4) mean that instruction I_j can be either issued at exactly $T_i+T_{CHS_i}$ or after $T_i+T_{R_i}$. Inequalities of (4) replace (1) for these instructions.

D.6. Asynchronous Chaining Constraints

Asynchronous chaining refers to the transfer of data between pipelines at or after the chain slot time when the

resources are available. This type of chaining allows the chaining of one or more instructions with the current one. This is not a likely possibility for synchronous chaining, because the instruction to be chained must be issued at the chain slot time of more than one instruction. That requires the chain slot times of two or more instructions to occur at the same time. Since asynchronous chaining can be performed at any clock time at or after the chain slot time, the current instruction does not have to be delayed till the preceding instruction is completed if the chain slot is missed. That allows earlier issue of its succeeding instructions. Asynchronous chaining is formulated as follows.

$$T_j - T_i \geq T_{CHS_i} \tag{5}$$

Inequality (5) replaces (1) for these instructions.

E. Asynchronous Issue-Execution

Issuing an instruction requires the availability of the resources needed by that instruction; these resources are the FU, the registers, and the access paths. When all of such resources for the current instruction are available, it will be issued, the resources will be reserved, and the execution of the data element(s) will be started. The length of time that these resources are reserved is determined by the amount of time required for the execution of data element(s) and the type of resources. For scalar

and address instructions, that contain one data element, operand registers are reserved for the duration of issue process. However, the result register of these instructions is reserved throughout the execution. On the other hand, for vector instructions that must process a number of data elements, each one of the required resources is reserved for a number of clocks no less than the vector length. In the previous formulations, it has been assumed that the issue time and the start of execution of data elements are the same, thus only one start time is used in forming the inequalities.

E.1. Flexible Issue

To allow the model and RRVP's a higher degree of flexibility, the synchronous issue-execution of instructions can be replaced by asynchronous issue-execution procedures that do not bind the issue-times and the start-of-execution-times to occur simultaneously. It is still necessary that the resources required by an instruction be available at the time of issue. All of A, S, and V instructions utilize synchronous issue-execution, because the availability of the resources at issue time will cause the start of their execution, except for instructions involved in an asynchronous chain. For instructions in an asynchronous chain, it is assumed that the vector registers that provide the chaining are available to the instruction to be chained, but the data elements do not arrive until

after the chain slot time. Therefore, an instruction that is to be asynchronously chained to one or more previously-issued instructions can be issued when resource requirements are satisfied, but will not start execution till the chained data elements become available. This procedure also affects the length of reservation interval of the resources used by the chaining instruction. It means that such resources are reserved at the time of issue and will be idle until the start of the execution. From this start time, the time at which those resources must be freed is determined.

Asynchronous issue-execution has an important effect on the total computation time. When an instruction that is to be chained is issued before it can be executed, its succeeding instructions are issued earlier than they would be if the issue of the chaining instruction were to be delayed until the chain slot time. Therefore, the total computation time may be reduced by allowing earlier execution of some instructions.

The length of the chain- the time interval starting at the issue time of the instruction starting the chain and ending at the termination-of-execution-time of the last instruction in the chain- is not altered when asynchronous issue-execution and asynchronous chaining are used. The only effect of such asynchronous issue-execution is the earlier issuing of the instructions involved in a chain.

E.2. Formulation Changes

To model asynchronous issue-execution, two independent variables are needed to specify the issue-time and the start-of-execution-time independently. Since the only instructions that are affected are the ones which are involved in a series of asynchronous chains, only the inequalities involving these instructions should be modified in the model.

In the preceding sections, the model has used one time variable to specify each instruction. That time variable is the issue time of the instruction. A new variable E_i is introduced that represents the time that the execution of instruction I_i must be started. For any instruction I_i that is not involved in an asynchronous chain, the execution starts at the time of issue; i.e. $T_i=E_i$. On the other hand, for instructions involved in such a chain, the following changes occur.

1- Change all of the dependency inequalities between the current instruction (I_i) and the succeeding vector instructions (I_j's), except for instructions in the same chain, to the following.

$$T_j - E_i \geq \tau_i \qquad (6)$$

τ_i is the length of their dependency.

2- If I_i is a mixed V and S type instruction, the dependency of the succeeding dependent S-type

instruction is not altered.

$$T_j - T_i \geq T_{ISS_i}$$

3- If I_j is asynchronously chained with I_i, then

$$\begin{aligned} T_j - T_i &\geq T_{ISS_i} \\ E_j - E_i &\geq T_{CHS_i} \end{aligned} \tag{7}$$

4- Since no instruction can be executed before it is issued,

$$E_i - T_i \geq 0$$

5- If I_i is independent from an instruction in the code, say I_j, and there exists an FU or vector-operand conflict between them. Then,

$$T_j - E_i + \theta * y_{ij} \geq \tau_i$$

or

$$T_i - E_j + \theta * (1 - y_{ij}) \geq \tau_j \tag{8}$$

where τ_i and τ_j are the durations of the conflicts depending on which instruction succeeds the other one. And, if the two independent instructions have only issue conflict, then the original issue conflict based only on issue time inequalities apply.

F. Global Dependency

Global dependency between two instructions that are connected by a directed path on their dependency graph is

determined from pairwise dependencies between the instructions of the path. Pairwise dependency between two instructions forces a minimum time-interval between their issue times. Since the global dependency relation can be represented by a combination of a number of pairwise dependency relations, it also enforces a minimum time-interval between the issue times of globally dependent instructions; e.g. $I_k \xrightarrow{G} I_i$ can be represented by $I_k \xrightarrow{P} I_j \xrightarrow{P} I_i$. This minimum time-interval is equal to the longest path connecting two globally dependent instructions where the length of a path is determined by the sum of all edge values of the edges on the path. Since the pairwise dependency ensures the availability of the registers at the time of issue of the dependent instruction, the implicit enforcement of global dependency between two instructions through pairwise dependencies is appropriate, except for some special cases.

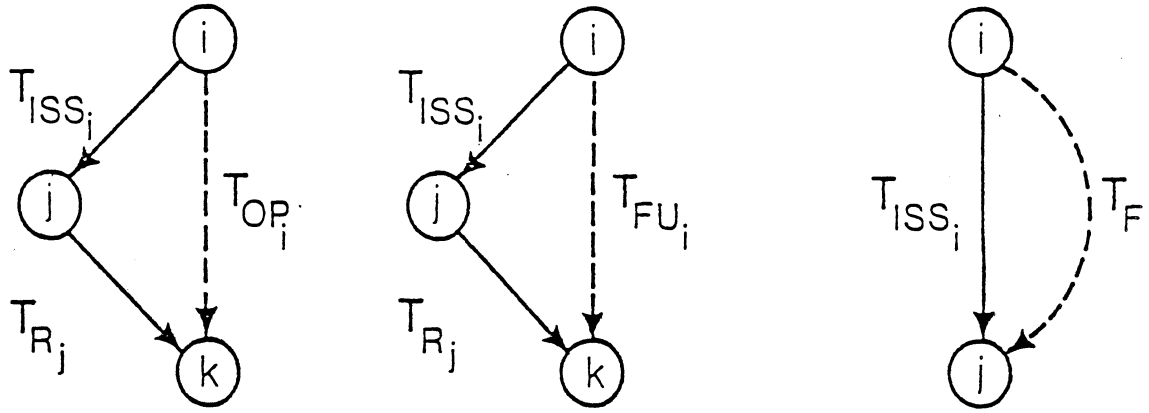
Special cases in which global dependencies between instructions cannot be satisfied through pairwise dependencies are caused by instructions that utilize vector registers plus one A or S register. Suppose two independent vector instructions I_i and I_k with FU or vector-operand conflict use the same A or S register as one of their operand registers; A and S registers are not reserved as operand registers. Now, suppose a scalar or address instruction I_j with the result register which is the same register as the one used by the two V instructions separates

those two V instructions. Obviously, $I_j \xrightarrow{P} I_i$ and $I_k \xrightarrow{P} I_j$. Therefore, I_i and I_k are not considered independent instructions and the FU or vector-operand conflicts are ignored. The dependency between I_j and I_i is T_{ISS_i} clocks since the S or A register is not reserved. And, the dependency between I_k and I_j is T_{R_j}. Therefore, the global dependency between I_i and I_k is $T_{R_j}+T_{ISS_i}$. However, due to FU or vector-operand conflict I_i and I_k should be separated by T_{FU_i} or T_{OP_i}. In cases where T_{FU_i} or T_{OP_i} is larger than $T_{R_j}+T_{ISS_i}$, a dependency inequality is needed to ensure the proper issue time distance between I_i and I_k. Hence, the following inequality is added to the model.

$$T_k - T_i \geq \tau_i \tag{9}$$

τ_i represents T_{OP_i} or T_{FU_i} depending on vector-operand or FU conflict between I_i and I_k, respectively. Fig. 2 (a) and (b) depict examples for vector-operand and FU conflict special cases.

Another case of exception occurs when a vector instruction I_i and a scalar one I_j where $I_j \xrightarrow{P} I_i$ have FU conflict. This case occurs when I_i uses a scalar register as its operand register and I_j uses the same S register as its result register and both of these instructions use the same FU. In these cases the original dependency value of T_{ISS_i} is replaced with T_{FU_i} to ensure the availability of the FU at the time I_j is ready to be issued. Hence, the pairwise dependency between I_i and I_j is displayed by the



i	Vi	Sj+FVk		Vi	,A0,1	Vi	Sj*FVk
j	Sj	/HSm		A0	Aj+Ak	Sj	Sl*FSm
k	Vp	Sj*FVk		Vj	,A0,1		

$T_j - T_i \geq T_{ISS_i}$	$T_j - T_i \geq T_{ISS_i}$	$T_j - T_i \geq T_{ISS_i}$
$T_k - T_j \geq T_{R_j}$	$T_k - T_j \geq T_{R_j}$	$T_j - T_i \geq T_{FU_i}$
$T_k - T_i \geq T_{OP_i}$	$T_k - T_i \geq T_{FU_i}$	
(a)	(b)	(c)

Fig. 2- Cases not Covered by Pairwise Dependency

(a)- special case with vector-operand conflict

(b)- special case with FU conflict

(c)- special case with FU conflict for S and V instructions.

following inequality.

$$T_j - T_i \geq T_{FU_i}$$

Fig. 2 (c) displays an example of such a conflict.

G. Loop Scheduling

In this dissertation, a loop is identified from the following definitions.

Definition 8- A set of instructions that are executed repeatedly and consecutively until a specific condition is satisfied is termed an instruction loop or simply a loop.

Definition 9- Each repetition of the execution of instructions of a loop is termed an iteration.

Usually, a loop is characterized by a set of instructions that are immediately preceded by a loop-label specifying the beginning of the loop and are immediately succeeded by a conditional branching instruction specifying the end of the loop. The instruction that should succeed the branching instruction in the execution sequence of the loop is either the instruction labeled by the loop-label if the termination condition has not been satisfied or the instruction succeeding the loop in the sequence of instructions if otherwise.

G.1. Unfolding the Loop

Optimal scheduling of a loop can be performed by unfolding the loop and scheduling the resulting set of instructions. Unfolding a loop with n iterations is performed by concatenating n copies of the instructions of the loop. Execution of this set of instructions is equivalent to the execution of the loop. Unfolding a loop may reduce its execution time since the time spent in branching does not exist for an unfolded loop. Optimal scheduling of this straight code produces a minimum loop execution time. However, unfolding a loop is not generally feasible because the number of iterations can be very large or, in a large number of cases, this number is not fixed and is determined at the time of execution.

G.2. Inter-Iteration Dependencies

Suppose that it takes T_x time units to issue and execute a fixed set of instructions. Now, the same set of instructions is issued and executed twice; the second iteration immediately follows the first one. The total execution time for the two iterations does not exceed $2*T_x$ time units. The reason is that the first instruction of the second iteration can be issued at any time after the last instruction of the first iteration has been issued and the required resources are available. Therefore, the second iteration can be started before the execution of the first iteration is completed or at the latest at its termination

time; when specific dependencies or resource requirements exist.

To produce optimal loop schedules, it is necessary to consider the effect of the instructions of successive iterations on each other. This necessity arises from the fact that the execution of the instructions of an iteration can be delayed due to pairwise dependencies on the instructions of the previous iteration and unavailability of the resources that are reserved by those instructions. The only exception is the first iteration that is not affected by inter-iteration dependencies, since no previous iterations exist. Therefore, straight code scheduling of the instructions of one iteration of a loop does not produce an optimal loop schedule since it does not consider the inter-iteration dependencies and resource conflicts between the instructions of successive iterations.

Theorem 1- An instruction I_i of iteration j+1 can be directly affected only by I_i of iteration j and the instructions that succeed I_i of iteration j and precede I_i of iteration j+1.

Proof- I_i of iteration j and iteration j+1 are identical instructions. The dependency and resource constraints imposed on them by instructions preceding I_i of iteration j must be resolved before I_i with earlier issue time can be issued. Therefore, these instructions directly affect I_i with earlier issue time, that is I_i of iteration j and not

I_i of iteration j+1. ■

This theorem implies that the identical instructions of different iterations are affected by identical conditions, except for the instructions of the first iteration.

G.3. Execution-Time of an Iteration

For any loop, at least one instruction can be found that determines the length of the execution of each iteration of that loop. Suppose T_{ij} is the time interval between the issue times of instruction I_i at two successive iterations j and j+1, for i=1,2,...,n and j=1,2,...,m-1 where n and m are the number of instructions in the loop and number of iterations of the loop, respectively. The length of the execution of iteration j is $T_{Ij}=\underset{i}{MAX}(T_{ij})$. T_{Ij} is the longest issue distance between two successive issues of instructions. If $T_{ij}=T_{Ij}$, then all of the instructions of the loop (excluding I_i) are issued and executed between two successive issues of I_i. Therefore, the instructions succeeding I_i are bound by its dependencies and must be delayed issue due to the sequential order of issue.

G.4. Steady State

Definition 10- Steady state of a loop is a state in which the relative issue-time distances between instructions do not change from one iteration to the next one.

Theorem 2- After the first iteration, the execution of the loop reaches a steady state.

Proof- According to theorem 1, after the first iteration, the issue time of an instruction in successive iterations is affected by identical set of instructions. Thus, under exactly identical conditions, the relative issue time distance between the instructions of successive iterations is fixed and does not change. Now, assume the contrary. It means that at some iteration the length of some pairwise dependency or resource reservation for at least one instruction differs from such conditions at other iterations. This cannot be true since the length of pairwise dependency and resource reservations are fixed[5] and are not based on iteration numbers. Therefore, due to recurring conditions, the execution of the loop reaches a steady state. ∎

Theorem 3- Steady state iterations have minimum execution times.

Proof- For steady state iterations, $T_{SS}=\underset{i}{MAX}(T_{ik})$ where k is a steady state iteration. If any iteration has a shorter iteration time, say T'_{Ij} for some j, since this is the longest time between the instructions in two successive iterations, all the instructions can be issued and executed during T'_{Ij}. Hence, such a relative issue time distance that

---

[5]It is assumed that the vector length

produces T'_{I_j} can be repeated, and the steady state iteration time will be $T'_{I_j} < T_{SS}$. This contradicts the original assumption. Therefore, the iteration execution time at steady state is the minimum iteration execution time. ■

G.5. Execution-Time of the Loop

Definition 11- An iteration marker is an instruction in the loop for which the time difference between its two successive issues is equal to T_{SS}.

At steady state, at least one iteration marker I_i exists to specify the steady state execution-time T_{SS}. The rest of the instructions can be issued at any time unit during some time range without affecting the iteration time. The issue times of the instructions of the first iteration can be arranged so that the steady state is enforced from the first issue of I_i. Then, the execution time of the loop can be presented by the sum of loop-start-up time, steady state iteration-times, and loop-finish-up time as follows.

$$T_{LOOP} = T_{SL} + (m-1)*T_{SS} + T_{FL} \qquad (10)$$

where T_{SL} is the time interval between the issue of the first loop instruction and the first iteration marker I_i, T_{SS} is the time between two successive issues of I_i, and T_{FL} is the time interval between the last issue of I_i and termination of the loop. T_{SL} and T_{FL}, combined, consider the issue and execution times of the instructions starting

at the loop-label to the first I_i plus the last I_i to the end of the loop. So, $T_{SL}+T_{FL}$ represents the execution time of one iteration. The remainder of execution is presented by the execution of the remainder of iterations; $(m-1)*T_{SS}$. For large values of m, $(m-1)*T_{SS}$ dominates the execution time of the loop. Hence, the minimization of steady state iteration time is the major goal in optimal loop scheduling.

G.6. Two-Copy Loop Scheduling

To enable the model to consider the inter-iteration effects on loop scheduling, a loop is replaced by two of its original copies separated by the branching instruction. Using two copies of the loop allows the inter-instructional dependencies and resource conflicts between two successive iterations to be considered, through modeling the relationships between the instructions of the two copies. Consequently, this loop scheduling method produces loop schedules that are optimal with respect to all of the iterations of the loop.

The steady state iteration execution time T_{SS} varies according to the order of instructions. This value is fixed for a specific order and if such an order is changed T_{SS} will change. Also, since T_{SS} is the time between the issue times of identical instructions of successive iterations and those instructions have a fixed relative issue distance from each other, the following equality can be set between the

instructions of the two copies of the loop.

$$T_j - T_i = T_{SS} \tag{11}$$

where T_i and T_j are issue times of the same (arbitrary) instruction in successive iterations. The minimization of T_{SS} is then the optimal loop scheduling criterion.

G.7. Floating Jump

The location of the conditional loop JUMP may be fixed or floating. Each steady state iteration starts with the iteration marker instruction, ends with the instruction preceding the marker at the following iteration, and includes a branching instruction (JUMP). The branching instruction can be left fixed at its original place at the end of the loop or it can be floated down into the second copy. The former approach will allow the relative reordering of instructions of only one copy of the loop. On the other hand, the latter approach will allow the branching instruction and the loop label to move in the code and find the best order of instructions to minimize T_{SS}. The floating loop method also makes it possible to place the JUMP at a place where the overhead generated by its execution is minimized; i.e. the issue delay of the instructions succeeding the JUMP due to its execution is minimized.

Fig. 3 illustrates a loop, its two copy substitute, and floating branching instruction or floating JUMP. The large brackets show the floating range of the loop-limits.

Only one set of loop instructions can be placed between the loop label and the branching instruction. When the JUMP is floating, the instructions succeeding the JUMP are brought into the loop and their identical counterpart is taken out of the loop by moving it above the loop-label. This action is equivalent to moving the label and JUMP downwards into the second copy. The JUMP can be floated by forcing it to be dependent on its preceding instructions and independent from its succeeding ones. When T_{SS} is enforced between two successive issues of an instruction, the JUMP-label combination can float into the second copy with the guarantee that only one set of instructions are placed between them.

H. Integer Programming

The mathematical model that has been introduced is composed of three types of inequalities: (a) dependency, (b) independency, and (c) chaining. The dependency inequalities are already in proper IP format. However, the two remaining types of inequalities, (b) and (c), must be converted to IP-acceptable inequality format.

The independency inequalities shown in (2) and (3) are called "either or" inequalities and can be converted to IP ones as follows [MURT] [BAKE].

$$\begin{array}{l} T_i - T_j \geq a_j \\ \text{or} \\ T_j - T_i \geq a_i \end{array} \quad \text{===>} \quad \begin{array}{l} \theta * y_{ij} + T_i - T_j \geq a_j \\ \\ \theta * (1 - y_{ij}) + T_j - T_i \geq a_i \end{array} \qquad (12)$$

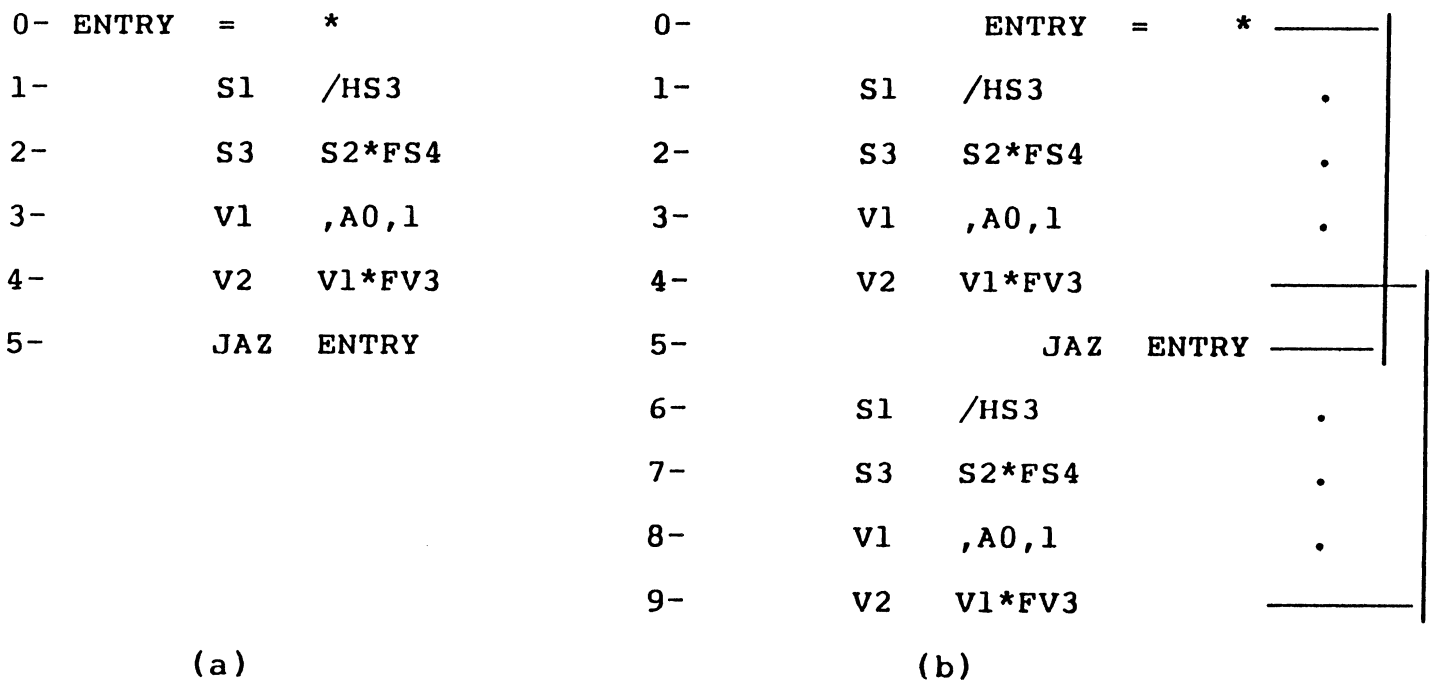


Fig. 3- Sample Loop with Floating Jump

(a) sample loop. (b) 2 copies of (a) with floating JUMP.

where y_{ij} can only take one of the values 0 or 1, and θ is a large positive integer. The inequalities of (12) can be presented in the following simplified form.

$$a_j \leq \theta * y_{ij} + T_i - T_j \leq \theta - a_i \qquad (13)$$

To check the correctness of this conversion, y_{ij} must be set to 0 and 1 and the resulting inequalities should produce the original ones. Thus, the appropriate inequality is chosen by selecting the proper value for y_{ij}.

The chaining inequalities of (4) also can be represented by such inequalities.

$$\text{or} \quad \begin{matrix} T_j - T_i \geq a \\ T_j - T_i = b \end{matrix} \quad ===> \quad \begin{matrix} \theta*(1-y_{ij}) + T_j - T_i \geq b \\ \theta*(y_{ij}-1) + T_j - T_i \leq b \\ \theta*y_{ij} + T_j - T_i \geq a \end{matrix} \qquad (14)$$

The inequalities of (14) can also be presented as follows.

$$\theta*(1 - y_{ij}) + T_j - T_i \geq b$$

$$\text{and} \qquad (15)$$

$$a \leq \theta*y_{ij} + T_j - T_i \leq \theta + b$$

Again, setting y_{ij} to 0 or 1 will result in the original inequality or equality, respectively.

The T_i and T_j are assured to be integers if the y_{ij}'s equal 0 or 1. To justify this claim, let the values of y_{ij}'s be known. Then, inequalities of (13) become type (1) inequalities. This change converts the problem to a

minimization problem subject to $Ax \leq b$ where x and b are solution and right-hand side vectors, respectively, and A is the coefficient matrix in which the elements of each row are all zeros, except for one +1 and one -1; i.e. the minimum time problem. Based on corollary 12.6 and theorem 2.7 of [LAWL], A^T and A are totally unimodular. Therefore, any basis [MURT] of A, say B, has an integer inverse B^{-1}. This means that any feasible solution of the problem $(B^{-1}b)$ results in integer variables. Consequently, an optimal solution that is also a feasible one must be integer. Thus, only the y_{ij} need be forced to integers.

CHAPTER III

REGISTER RE-ASSIGNMENT

Optimal instruction scheduling is directly affected by the choice of register assignments. Registers assigned to instructions dictate the pairwise and global dependencies between them. The number of dependent instructions, whose concurrent execution is prohibited, is highly dependent on the choice of register assignment that sets a lower bound on the total computation time. However, if the register assignment is changed so that the number of dependent instructions is decreased, and consequently the number of independent instructions is increased, the lower bound on the computation time may be reduced.

A. Modified Objective

The current definition of optimal instruction schedules for RRVP's that is based only on the order of instructions must be revised to include the effect of register assignment. The model of Chapter II is capable of producing optimal schedules with respect only to instruction sequences. It is clear that if a number of registers are reassigned , a previously optimal schedule may no longer be optimal. Therefore, define an optimal schedule

as one in which changes in the sequence of instructions and/or register assignments cannot result in shorter computation times.

B. Result Register Reassignment

The register reassignment problem is the problem of reassigning result registers where the operand registers are accordingly adjusted. An operand register that is expected to contain a specific value can be assigned only the result register that has received the proper value. Thus, after a result register is reassigned, to preserve the integrity of the computation, all of the matching operand registers of the succeeding instructions also must be similarly reassigned until the next matching result register is encountered. Hence, the register reassignment problem is equivalent to the result register reassignment problem.

C. Complete and Partial Assignments

Definition 12- Complete reassignment of registers refers to the reassignment of the result registers of all of the instructions in the code.

Definition 13- Partial reassignment of registers refers to the reassignment of the result registers of some, but not all, of the instructions.

The result register reassignment need not be

complete. A complete reassignment of registers allows the model to select the best assignment for all of the instructions. However, in some cases a partial assignment in which some of the result registers are fixed‘ might be unavoidable. Therefore, for the sake of generality, in the following discussion complete and partial reassignment problems are both formulated.

Call-by-value is a good example of the case for which fixed registers cannot be avoided. When a sub-program is called by value, the information from the calling routine is transferred to the called one using a specific set of registers. To ensure the proper transfer of information and to preserve the integrity of the computation, those specific registers must not be reassigned in the called routine. Therefore, in such occasions, a complete reassignment is not possible.

D. Register Selection

An important criterion in register reassignment problem, as in instruction sequencing, is the preservation of the correctness of the computation. This means that a register reassignment that changes the outcome of the computation is not valid.

The following theorems specify the criteria for register assignment invalidity.

‘The register is not to be reassigned.

Definition 14- A register-use-range (RUR) for a register is a portion of the code which starts at an instruction that uses the register as its result register and ends at the last instruction that uses it as operand register, before it is reused as result register.

This definition specifies a range of instructions in which the contents of a register is not altered. Usually, each register has a number of RUR's through out the code. Register-use-range of register k which is the result register of instruction I_i is displayed by RUR_{ki}

Theorem 4- Assignment of register k as result register of instruction I_i (RR_{ki}) is invalid if I_i is in the register-use-range of RR_{kh}.

Proof- If $I_i \varepsilon RUR_{kh}$, there exists an instruction $I_j \varepsilon RUR_{kh}$ such that $I_j > I_i$[7] and register k is an operand register of I_j (OR_{kj}). Originally $[RR_{kh}]=[OR_{kj}]$, where [x] means contents of x. If register k is assigned as the result register of I_i and $I_h < I_i < I_j$, the following will be true.

$$[OR_{kj}]=[RR_{ki}]\neq[RR_{kh}]$$

This assignment alters the logic and the outcome of the

---

[7]The relationships 'I_i succeeds I_j' and 'I_i precedes I_j' are displayed by $I_i > I_j$ and $I_i < I_j$, respectively.

code; thus it is invalid. ■

Theorem 5- Assignment of register k to instruction I_i is invalid if the original RUR of the result register of I_i includes I_j that has a fixed RR_{kj}.

Proof- Suppose RR_{kj} is fixed and $I_j \varepsilon RUR_{mi}$; register m is presently assigned to the result register of I_i. There exists an $I_p \varepsilon RUR_{mi}$ such that $[OR_{mp}]=[RR_{mi}]$ and $I_j<I_p$. If RR_{mi} is reassigned as RR_{ki}, then

$$[OR_{kp}]=[RR_{kj}]\neq[RR_{ki}]$$

Therefore the computation is altered and the assignment is invalid. ■

Definition 15- A register whose assignment as the result register of instruction I_i is valid is called a candidate register of instruction I_i.

Definition 16- The collection of candidate registers for I_i is called the candidate set (CS) of the result register of instruction I_i.

The candidate set of I_i is displayed by $CS_{RR_{?i}}$ where ? shows that the result register is to be reassigned and is not currently fixed.

The problem of reassignment of registers is the process of assigning a register to an instruction from its CS, so that the assignment is valid and reduces the

computation time. For each instruction a set of candidate registers can be selected from the complete set of registers so that a) the type of the candidate register matches the type of the register to be reassigned and b) the assignment of the register is valid.

E. Changing the Original Model

Register-reassignment changes the inter-instructional dependency and independency relations, thus altering their respective inequalities. The dependency (independency) relations between an instruction and its predecessors[8] either are unaffected by register reassignment or are converted to independency (dependency) relations. To easily reflect this change of relations, all of the necessary inequalities of an instruction that display the assignment of a register should be included in or eliminated from the model if the register is or is not assigned, respectively. This change is incorporated in the original model (the model of Chapter II) by assigning a 0-1 variable to each register in the candidate set of each instruction. The value of each of these variables determines if its respective register is or is not assigned. Consequently, these variables can be used to include the desired dependency and independency inequalities in the

---

[8]It is sufficient for an instruction to consider only the ones that precede it for dependency and independency relations. This is necessary to avoid redundancy.

model by forcing the undesired inequalities to be unconditionally satisfied.

F. Changing the Dependency Inequalities

To display the effect of register assignment on dependency relations, the pairwise dependency relation is partitioned into two relations, as follows.

Definition 17- Given two instructions I_i and I_j, where I_j succeeds I_i in the sequential order of instructions, I_j is data dependent on I_i iff at least one of the operand registers of I_j is the same as the result register of I_i, and I_j is in the RUR of I_i.

This definition explicitly specifies the instruction whose dependency relations are independent of the register assignment. The dependency is due to instruction I_j that must use the result of the calculations performed by I_i. Therefore, regardless of the register assignment, I_j must be held issue until I_i provides the required data. This relationship is displayed by $\xrightarrow{D}$.

Definition 18- Given two instructions I_i and I_j, where I_j succeeds I_i in the sequential order of instructions, I_j is register dependent on I_i iff the result register of I_j is the same as one of the registers of I_i.

From this definition it is clear that the register dependency between two instructions is dependent on the register assignment. This relationship is displayed by $\xrightarrow{R}$.

The data and register dependency inequalities should be substituted for pairwise dependency inequalities of (1). If two instructions I_i and I_j are data dependent, instruction I_j cannot be issued until the required data becomes available. Therefore,

$$T_j - T_i \geq t_i \tag{16}$$

where t_i is the duration of data dependency between I_i and I_j. This dependency is due to the availability of the result registers of I_i. Hence, $t_i = T_{R_i}$. If this dependency between I_i and I_j exists, no other relationships can affect the issue time distance between I_i and I_j. Therefore, if data dependency between two instructions exists, that is the only relationship that needs be considered.

Suppose that (16) does not apply and the result register of instruction I_j is to be reassigned. A group of 0-1 variables are needed to specify the register that is chosen from the candidate set to replace the original result register. Define r_{jk} with $k \in CS_{RR_?j}$ to be 1 if candidate register k is chosen for the assignment and 0 otherwise.

$$r_{jk} = \begin{cases} 1 & \text{if } RR_{kj} \\ 0 & \text{otherwise} \end{cases}$$

Since only one register is assigned as the result register of instruction I_j, one and only one of r_{jk}'s can be 1 and the rest should be zeros. This constraint can be formulated as follows.

$$\sum_{k \varepsilon CS_{RR_{?j}}} r_{jk} = 1 \qquad (17)$$

Now, to show how the dependency inequalities should be altered to reflect the register dependency relation, the following cases must be considered. In the following cases, the result register of instruction I_j is reassigned and its relation with instruction I_i is under consideration.

F.1. Case 1

In this case, the dependency relationship between I_j and I_i is evaluated based on the operand registers of I_i and the result register of I_j. Since all of the operand registers of I_i behave similarly with respect to this relationship, the effect of only one of the operand registers is discussed.

Because only result registers are reassigned, an operand register is equivalent to the result register of the instruction on which it is data dependent. It means that to determine whether register k is assigned as an operand register of I_i, r_{hk} must be examined where I_h is the instruction that supplies $OR_{?i}$. Similarly, referring to $CS_{OR_{?i}}$ is the same as referring to $CS_{RR_{?h}}$ $(=CS_{OR_{?i}})$.

A register dependency relation between I_i and I_j is established when $RR_{?j}=OR_{?i}$. In other words, if $k\varepsilon CS_{RR_{?j}}/\backslash CS_{RR_{?h}}$ and $r_{jk}=r_{hk}=1$, then $I_j \xrightarrow{R} I_i$. The following inequalities formulate this dependency relation.

$$T_j-T_i+\gamma*(2-r_{jk}-r_{hk})\geq\tau_i \quad \forall\ k\varepsilon CS_{RR_{?j}}/\backslash CS_{RR_{?h}} \tag{18}$$

γ is a large positive number and τ_i is the number of time units that $OR_{?i}$ is reserved. If $CS_{RR_{?j}}/\backslash CS_{RR_{?h}}=\phi$, inequalities (18) do not apply to this case. Also, if the operand register of I_i is fixed, say register k, then $CS_{RR_{?h}}=CS_{OR_{?i}}=\{k\}$. Thus, $r_{hk}=1$ and (18) is simplified to the following single inequality.

$$T_j-T_i+\gamma*(1-r_{jk})\geq\tau_i \tag{19}$$

If more than one operand is fixed, say k and m, (19) can be extended to the following.

$$T_j-T_i+\gamma*(1-r_{jk}-r_{jm})\geq\tau_i$$

F.2. Case 2

This case deals with preventing the occurrence of theorems 1 and 2. If $RR_{?i}$ and $RR_{?j}$ are both reassigned and $I_j\varepsilon RUR_{RR_{?i}}$, then $RR_{?j}\neq RR_{?i}$. This simply means the following.

$$r_{jk}+r_{ik}\leq 1 \quad \forall\ k\varepsilon CS_{RR_{?j}}/\backslash CS_{RR_{?i}} \tag{20}$$

This inequality enforces the selection of register k for at most one of the instructions I_i or I_j and not both

simultaneously.

Obviously, if there are no common registers in the candidate sets, inequalities (20) do not apply to this case. Inequalities (20) prevent the occurrence of Theorem 4. Theorem 5 can occur if $RR_{?i}$ is being reassigned, $RR_{?j}$ is fixed (say k), $k \varepsilon CS_{RR_{?i}}$, and $I_j \varepsilon RUR_{RR_{?i}}$. This occurrence can also be prevented by using (20).

$$r_{ik} + r_{jk} \leq 1$$

F.3. Example

The following example demonstrates the formulation of the dependency relation between two instructions in the presence of register reassignment. It is required to formulate the relationship between I_i and I_j when $RR_{?j}$ is reassigned in the following code.

h	$S_?$	.....	$CS_{RR}=\{2,7\}$
i	$S_?$	$S_{RR_{?h}}+S_6$	$CS_{RR}=\{1,3,4\}$ & $CS_{OR}=\{2,7\}\setminus\{6\}$
j	$S_?$	.....	$CS_{RR}=\{2,3,6\}$
m	..	$S_{RR_{?i}}$+..	

First, only one of the registers 2, 3, or 6 can be assigned to $RR_{?j}$. Using equality (17), the following results.

$$\sum_{k \varepsilon CS_{RR_{?j}}} r_{jk} = 1 \quad ====> \quad r_{j2}+r_{j3}+r_{j6}=1$$

Since the second operand register of I_i is fixed, $I_j \xrightarrow{R} I_i$

if $RR_{?j}$=6. This is a Case 1 formulation, as follows.

$$T_j-T_i+\gamma*(1-r_{j6})\geq\tau_i$$

Also, $I_j \xrightarrow{R} I_i$ if $RR_{?j}$ is reassigned the same register as the first operand register of I_i. To determine that possibility, the CS of these registers must be compared.

$$CS_{RR_{?j}} /\backslash CS_{RR_{?h}}=\{2\}$$

So, $I_j \xrightarrow{R} I_i$ if $RR_{?j}=OR_{?i}=RR_{?h}=2$. This also is a Case 1 formulation that follows.

$$T_j-T_i+\gamma*(2-r_{j2}-r_{h2})\geq\tau_i$$

Since $OR_{?m}=RR_{?i}$ and $I_i<I_j<I_m$, RR(?j) cannot be assigned the same register as $RR_{?i}$. That means no registers can be assigned as result registers of both of the instructions I_i and I_j since $I_i \varepsilon RUR_{RR_{?i}}$. Because

$$CS_{RR_{?i}} /\backslash CS_{RR_{?j}}=\{3\}$$

assignment of register 3 to both I_i and I_j must be avoided. This is a Case 2 formulation, as follows.

$$r_{j3}+r_{i3}\leq 1$$

Therefore, the formulation of the dependency relationship between I_i and I_j where $RR_{?j}$ is reassigned requires three inequalities and one equality.

G. Changing the Independency Inequalities

Since the independency relation between two instructions is affected by the register reassignment, the independency relation formulations of Chapter II must be modified. To determine the independency of two instructions, it is sufficient to compare the result register of the succeeding instruction with the registers of the preceding instruction. Suppose the result register of instruction I_j is presently being reassigned, and instruction I_i precedes I_j and the registers of I_i have been reassigned as follows.

$$RR_{?i},\ OR_{?i}=RR_{?h},\ \text{and another}\ OR_{?i}=RR_{?g}$$

If

$$(CS_{RR_{?j}} /\backslash CS_{RR_{?i}}) \backslash/ (CS_{RR_{?j}} /\backslash CS_{RR_{?h}}) \backslash/ (CS_{RR_{?j}} /\backslash CS_{RR_{?g}}) = \phi \quad (21)$$

then, I_j and I_i are unconditionally independent and original independency inequalities remain unchanged. A simple example for unconditional independency occurs when I_j and I_i are instructions of different types, and thus their CS's do not have any registers in common.

If (21) does not hold, then there exist a number of common registers that can affect the independency relation between I_i and I_j. Suppose one register, register k, is shared between the CS's of I_h and I_j. The effect of register k can be represented by the following inequalities.

$$\begin{cases} T_i - T_j + \theta * y_{ij} + \gamma * r_{jk} \geq \tau_j \\ T_i - T_j + \theta * y_{ij} + \gamma * r_{hk} \geq \tau_j \\ T_j - T_i + \theta * (1 - y_{ij}) + \gamma * r_{jk} \geq \tau_i \\ T_j - T_i + \theta * (1 - y_{ij}) + \gamma * r_{hk} \geq \tau_i \end{cases} \tag{22}$$

τ_i and τ_j represent the original right hand sides of the independency inequalities. These inequalities are needed because I_i and I_j are independent, except when $r_{jk}=r_{hk}=1$. Hence, in cases when both r_{jk} and r_{hk} are 0 or only one of them is 1, the independency inequalities exist and must be satisfied. Only when both instructions choose the same register k, $I_j \xrightarrow{R} I_i$ and all four inequalities are automatically satisfied.

If two registers k and m are shared the following is used.

$$\begin{cases} T_i - T_j + \theta * y_{ij} + \gamma * (r_{jk} + r_{hm}) \geq \tau_j \\ T_i - T_j + \theta * y_{ij} + \gamma * (r_{hk} + r_{jm}) \geq \tau_j \\ T_j - T_i + \theta * (1 - y_{ij}) + \gamma * (r_{jk} + r_{hm}) \geq \tau_i \\ T_j - T_i + \theta * (1 - y_{ij}) + \gamma * (r_{hk} + r_{jm}) \geq \tau_i \end{cases} \tag{23}$$

Again, the original independency inequalities will remain to be satisfied if neither register k nor register m are shared between I_j and I_i. On the other hand, if one of those registers is shared ($r_{jk}=r_{hk}=1$ or $r_{jm}=r_{hm}=1$), all of the inequalities are satisfied and instructions are not independent.

Beyond two registers, each added possible register

will double the number of the inequalities, one set of inequalities with $\gamma*(...+r_{j?})$ and the other set with $\gamma*(...+r_{h?})$. Hence, if either (both) $r_{j?}$ or (and) $r_{h?}$ is (are) zero(s), at least one set of inequalities remains to be satisfied by other register assignments. And, if both $r_{j?}$ and $r_{h?}$ are 1's, then all of the inequalities are satisfied and $I_j \xrightarrow{R} I_i$.

For simplicity of presentation the registers introduced in the inequalities (22) and (23) are the ones that result from the middle intersection operation in (21). In addition to these registers, registers that result from the other intersection operations in (21) should also be included in the formulation. To consider those registers, such as r_{gm}, replace r_{hm} in (23) with $(r_{gm}+r_{hm})$ since selection of register m by either instruction will similarly affect the dependency relation. Now, it is possible for a register to be considered more than once if it belongs to more than one of the intersections in (21). That, however, is quite proper since such a register can be independently assigned to either one, but not both, of instructions I_h or I_m.

These inequalities can be simplified when some or all of the registers of I_i are fixed. In this case, the requirement is to avoid the fixed register if independency relation between I_i and I_j is desired. That can be done only when $RR_{?j}$ is not assigned any of the fixed $OR_{?i}$'s or

the $RR_{?i}$. Suppose that $RR_{?i}$ and $OR_{?i}$'s are fixed at registers k, m, and p respectively. Then,

$$\begin{aligned} T_i - T_j + \theta * y_{ij} + \gamma * (r_{jk} + r_{jm} + r_{jp}) \geq \tau_j \\ T_j - T_i + \theta * (1 - y_{ij}) + \gamma * (r_{jk} + r_{jm} + r_{jp}) \geq \tau_i \end{aligned} \qquad (24)$$

These inequalities specify that if I_j selects any one of the registers that are selected as result and operand registers of I_i, $I_j \xrightarrow{R} I_i$. Then, the inequalities are satisfied since no independency relationships exist.

An optimal register assignment is found by selecting 0 or 1 values for r_{ik}, for all i and k, such that all of the inequalities are satisfied and minimum computation time is resulted. The assignment of r_{ij}'s, that is equivalent to reassignment of registers to instructions, forces a group of the inequalities to be satisfied. The remainder of inequalities form a model that represents the code with the new register assignment. Or, it can be viewed as a dependency graph in which the paths connecting the instructions change with a change in the register assignment. A graph or a set of inequalities that can produce minimum computation time for the code represents the optimal instruction sequence and register assignments.

H. Branching Complications

When a conditional branch is encountered, the registers used before and after the branch should be carefully matched to preserve the correctness of the

computation. The cases displayed in Fig. 4 show two possible complications that may arise.

Conditional branching instructions affect the order of issue of the instructions. If the branching condition is satisfied, the execution of the branching instruction either eliminates the execution of a number of its succeeding instructions (Case 1 in Fig. 4) or forces the repeated execution of a number of its preceding instructions (Case 2 in Fig. 4). On the other hand, if that condition is not satisfied, the branching instruction has no effect on the order of instruction-issue. In any case, inter-instructional relationships between instructions that are affected by the branching condition are also dependent on that condition.

Consider Case 1 in Fig. 4 in which register A1 is RR of I_i and I_j and OR of I_{j+1}. For this case, $I_{j+1} \xrightarrow{D} I_i$ or $I_{j+1} \xrightarrow{D} I_j$ if the branching condition for JAP is or is not satisfied, respectively. Since an operand register is reassigned the same register as the result register that supplies its data, the operand register of instruction I_{j+1} must be the same register as the result register of instructions I_i and I_j. Therefore, $RR_{?i}=OR_{?(j+1)}=RR_{?j}$. To enforce the assignment of the same register for instructions I_i, I_j, and I_{j+1}, identical 0-1 variables must be used for the reassignment of the result registers of these instructions. More specifically, if instruction I_i uses $r_{i?}$, then instruction I_j should use $r_{i?}$ or I_j can use $r_{j?}$

```
Case 1 :        i           A1      A3+A2
                            JAP     NEXT
                              .
                              .
                              .
                j           A1      A4+A5
                    NEXT    S1      ,A1
```

```
Case 2 :        i           A1      A3+A2
                    NEXT    S1      ,A1
                              .
                              .
                              .
                j           A1      A4+A5
                            JAP     NEXT
```

Fig. 4- Branching Complications in Register Reassignment

and $r_{j?}=r_{i?}$ should be included in the formulation. If the reassignment of registers is partial and one of the result registers of instructions I_i or I_j is not reassigned, the other one must be kept fixed and cannot be reassigned. The reassignment restrictions mentioned for Case 1 also apply to Case 2.

CHAPTER IV

MULTIPLE IDENTICAL FUNCTIONAL UNITS

In previous chapters, it has been assumed that a processor contains M non-identical FU's, where M is a fixed number for a vector processor. It means that an instruction can be processed by one and only one FU that is of the correct type. To increase the generality of the model, it is extended to permit the use of identical copies of the original FU's. Therefore, since each instruction may have access to more than one FU of the proper type, the number of concurrently-executing independent-instructions may increase. In turn, such an increase will reduce the total computation time. This chapter is devoted to the description of such an extension.

A. Multiple-Identical FU's

Addition of multiple copies of FU's to the architecture of the RRVP's mainly affects the resolution of the FU conflicts. This addition becomes useful only when two or more instructions compete for the same type of FU. In this case, when extra copies of the FU are available, some of these instructions may be concurrently processed.

Concurrent execution of instructions is possible when the instructions are independent and require the same multiple-copy FU. Since the independency relation is not transitive, assignment of concurrent execution of more than two instructions requires a careful examination of their relationships. Therefore, to identify groups of instructions that can be concurrently processed, the following definitions for group-independency relationships are necessary.

Definition 19- Two instructions that are independent and require the use of the same type of FU are called independent and functionally identical (IFI) instructions.

Definition 20- A set of mutually independent instructions is a set of instructions in which any two instructions are pairwise-independent.

Definition 21- A set of mutually independent and functionally identical (MIFI) instructions is a mutually independent set of instructions in which all of the instructions require the use of the same type of FU.

It is necessary to model the multiple-identical FU problem when the number of MIFI instructions is larger than the number of available copies of their type of FU. If this condition is not satisfied, all of the instructions of a set

of MIFI instructions can be concurrently processed on separate copies of their proper type FU. In this case, FU conflict does not exist and should be eliminated from the original formulation. On the other hand, when the above condition does not hold and the FU conflicts exist, the formulation of that conflict is necessary. The remainder of this chapter is concerned with the case in which FU conflicts occur.

The solution of the multiple-copy-FU problem is rather simple when no more than two copies of each FU exist, compared with the case where the number of multiple copies of FU's exceeds two, as is true for a number of multi-machine scheduling problems [COFF]. Due to this distinction, the rest of this chapter is divided into two subsections describing solutions for a two-copy case and a general case. The general case considers any predetermined number of copies of FU's. Even though the two-copy case is included in the general case, a less complicated formulation of this case is presented.

B. The Two-Copy Case

To properly utilize the extra copy of each FU, the model is extended so that it can determine whether or not two IFI instructions should be concurrently processed. Two IFI instructions have the potential to be concurrently processed only if those instructions are assigned to two different copies of an FU. Comparing the copies to which the

instructions are assigned will reveal the concurrency or the sequentiality of their execution. If the IFI instructions are assigned to two different copies for concurrent execution, FU conflict need not be considered. Otherwise, their FU conflict must be formulated and included in the model.

B.1. Discriminating Between Copies

To discriminate between two copies of an FU, each instruction (I_i) is assigned a unique variable (f_i) that is defined as follows.

$$f_i = \begin{cases} 0 & \text{if instruction } I_i \text{ is assigned to copy number 1} \\ 1 & \text{if instruction } I_i \text{ is assigned to copy number 2} \end{cases}$$

Specification of the type of FU in f_i has been ignored since the type of an instruction determines the type of FU that it can utilize. Also, when some, but not all, of the FU's have two copies, the f_i associated with the instructions that have access to only one copy of the proper type FU should be eliminated. In other words, if only one copy of an FU exists, the instructions to be processed on that FU have an $f_i=0$ implicitly associated with them, and the explicit assignment of f_i is not necessary.

B.2. Independency Inequalities

Since only the number of copies of FU's has changed, that change affects only the FU conflict inequalities. Some

of the previously sequentially-processed independent instructions can now be processed concurrently. To include this change in the model and allow the model to optimally assign the copies of these FU's to competing instructions, the inequalities representing FU conflict should be replaced with the following inequalities.

B.1.a. Case 1

Both IFI instructions I_i and I_j are assigned to execute on the same copy. Such an assignment can be determined by examining f_i and f_j. If $f_i=f_j$, FU conflict exists. On the other hand, $f_i \neq f_j$ means that the instructions can execute concurrently and FU conflict cannot occur. Hence, the following inequalities that represent this case should be substituted for original conflict inequalities.

$$\begin{aligned} T_i - T_j + \theta * y_{ij} + \gamma * (f_i + f_j) &\geq T_{FU_j} \\ T_j - T_i + \theta * (1 - y_{ij}) + \gamma * (f_i + f_j) &\geq T_{FU_i} \end{aligned} \tag{25}$$

and,

$$\begin{aligned} T_i - T_j + \theta * y_{ij} + \gamma * (2 - f_i - f_j) &\geq T_{FU_j} \\ T_j - T_i + \theta * (1 - y_{ij}) + \gamma * (2 - f_i - f_j) &\geq T_{FU_i} \end{aligned} \tag{26}$$

The inequalities of (25) or (26) remain to be satisfied when f_i and f_j are both either 0 or 1, respectively. In these cases, the FU conflict exists and the instructions must be sequentially processed. And, if they are assigned to two different copies (only one of f_i or f_j is 1) those

inequalities will be automatically satisfied.

B.1.b. Case 2

The IFI instructions I_i and I_j execute on different copies of their FU. In this case, the only concern is the issue conflict. Since these instructions must not be issued at the same time, the issue conflict inequalities of Chapter II should be unconditionally added. These inequalities are restated in the following.

$$\begin{aligned} T_i - T_j + \theta * y_{ij} &\geq T_{ISS_j} \\ T_j - T_i + \theta * (1 - y_{ij}) &\geq T_{ISS_i} \end{aligned} \tag{27}$$

On the other hand, the inequalities of (27) should be activated only when inequalities of (25) and (26) are not active and deactivated otherwise.

$$T_i - T_j + \theta * y_{ij} + \gamma * (1 - f_i + f_j) \geq T_{ISS_j}$$

$$T_j - T_i + \theta * (1 - y_{ij}) + \gamma * (1 - f_i + f_j) \geq T_{ISS_i}$$

and,

$$T_i - T_j + \theta * y_{ij} + \gamma * (1 + f_i - f_j) \geq T_{ISS_j}$$

$$T_j - T_i + \theta * (1 - y_{ij}) + \gamma * (1 + f_i - f_j) \geq T_{ISS_i}$$

The above formulation enforces the issue conflict resolution when $f_i \neq f_j$ and is automatically satisfied when $f_i = f_j$.

The 0-1 variable assigned to each instruction specifies the copy of the proper type FU on which it should

execute. Assignment of 0 or 1 to these variables determines the concurrency of execution of instructions with respect to each other. The optimal assignment of values to these 0-1 variables occurs when a high degree of concurrency results in minimum computation time.

C. The General Case

The objectives of the general case are the same as the ones of the two-copy case. It is required to sequentially process the instructions that use the same copy of an FU, and enforce the issue conflict on the concurrent ones. Obviously, the changes, again, only affect the FU conflict inequalities.

The method used in the two-copy case cannot be simply applied to the general case. When the number of copies of the FU's exceed two, a number of 0-1 variables are required to represent the copy of the FU on which an instruction must execute. The elaborate formulations required for comparison of these copy numbers causes the rejection of use of such a method for the general case. Therefore, a different method will be introduced for the formulation of this case. This general case method can also be applied to the two-copy case, but due to its higher degree of complexity such a use is not recommended.

In the general case, to reduce the complications of the problem, the specific copy to which an instruction is

assigned will not be determined. However, it will be determined whether two IFI instructions should be processed by one copy or by different ones. Therefore, a 0-1 variable f_{ij} is defined to display the sequentiality or concurrency of two such instructions I_i and I_j, as follows.

$$f_{ij}= \begin{cases} 0 & \text{if } I_i \text{ and } I_j \text{ execute on the same copy} \\ 1 & \text{otherwise} \end{cases}$$

f_{ij}'s are defined only for two IFI instructions that require the use of an FU with multiple copies.

Depending on the value of f_{ij} either the FU conflict inequalities or issue conflict inequalities should hold for two IFI instructions. All of these inequalities can be compressed into the following form:

$$\begin{aligned} T_i - T_j + \theta * y_{ij} &\geq T_{FU_j} - f_{ij} * (T_{FU_j} - T_{ISS_j}) \\ T_j - T_i + \theta * (1 - y_{ij}) &\geq T_{FU_i} - f_{ij} * (T_{FU_i} - T_{ISS_i}) \end{aligned} \qquad (28)$$

Inequalities of (28) represent either FU conflict or issue conflict when f_{ij} is set to 0 (sequential) or 1 (concurrent), respectively. Therefore, determination of f_{ij} is equivalent to the specification of concurrent or sequential execution of instructions, and vise versa.

C.1. Limited Concurrency

Neither the definition of f_{ij} nor the above inequalities limit the number of concurrent operations.

Since the number of copies of an FU is limited, say m, the maximum number of MIFI instructions that can concurrently execute cannot exceed m. To enforce this limit, all groups of m+1 MIFI instructions are forced to contain at least two instructions that are scheduled to execute sequentially on one of the copies of the FU. Therefore, the following inequality must be satisfied for any group of m+1 MIFI instructions.

$$\sum_{h=1}^{m} \sum_{k=h}^{m} f_{i_h i_{k+1}} \leq m*(m+1)/2 - 1 \qquad (29)$$

$i_1, i_2, \ldots, i_{m+1}$ represent any m+1 MIFI instructions. This inequality represents the relationship between the m+1 instructions that are limited to m processors. The first instruction is concurrent with all others; hence $\sum_{k=1}^{m} f_{i_1 i_{k+1}} = m$. The second instruction has m-1 concurrent instructions to consider, and so on. Finally, the m^{th} instruction is sequentially processed with the $(m+1)^{st}$; $f_{i_m i_{m+1}} = 0$. Thus, the result is the following sum.

$$m + (m-1) + (m-2) + \ldots + 2 + 0 = m*(m + 1)/2 - 1.$$

The inequality (29) is only needed when the number of copies of the FU that are concurrently required by the MIFI instructions is smaller than the number of such instructions. Otherwise, the instructions could all be processed concurrently and f_{ij}'s would no longer be required. Also, since instructions cannot be issued at the

same time, the maximum limit on m can be determined given maximum vector length and issue times. For example, if it takes T_{ISS} clocks to issue any instruction and that instruction reserves its FU for only VL clocks, the maximum number of concurrent instructions, given sufficient number of copies of the required FU, is VL/T_{ISS}. This also shows that if the number of copies is larger than VL/T_{ISS}, the extra copies will never be used.

C.2. One-Copy-Assignment Restriction

The 0-1 variables f_{ij}'s are assigned to monitor the concurrency of the instructions. However, these variables do not specify the specific copy of the FU on which an instruction should be processed. These assignments do not prevent an instruction from incorrectly executing on more than one copy of an FU. This incorrect condition is displayed in the following example.

$$f_{ij} = 0 \qquad f_{jk} = 0 \qquad f_{ik}=1$$

Obviously, if I_i & I_j and I_j & I_k should be sequentially processed, I_i & I_k cannot execute concurrently. In the above formulations, no measures have been taken to prevent the occurrence of such conditions. Therefore, the following inequalities are introduced to guarantee the correctness of these assignments. These inequalities should be applied to all groups of 3 MIFI instructions.

$$f_{ij} + f_{jk} + f_{ik} \neq 1 \qquad (30)$$

i, j, and k represent any 3 MIFI instructions. The general multiple copy FU case is modeled by combining inequalities of (28), (29), and (30). Equation (30) can be displayed as follows.

$$\text{or} \qquad \begin{aligned} f_{ij} + f_{jk} + f_{ik} &= 0 \\ f_{ij} + f_{jk} + f_{ik} &\geq 2 \end{aligned} \qquad (31)$$

Conversion of inequalities (31) to IP inequalities is similar to the conversion of inequalities (4).

C.3. Two-Copy Version

As an example, the general case model is applied to the two-copy case problem. Inequalities (28) and (30) should be included with no changes. Inequality (29), adjusted for m=2, appears as follows.

$$f_{ij} + f_{jk} + f_{ik} \leq 2$$

Combining the adjusted version of inequality (29) with inequalities of (31) results in the following equations.

$$\text{or} \qquad \begin{aligned} f_{ij} + f_{jk} + f_{ik} &= 0 \\ f_{ij} + f_{jk} + f_{ik} &= 2 \end{aligned}$$

These inequalities display that either all of the 3 MIFI instructions are sequentially processed or two out of three of them may be concurrently processed.

It can be seen that the general method requires at least as many variables as the two-copy case does. For n MIFI instructions, n f_i's, n!/((n-2)!2!) f_{ij}'s are needed. As n increases, the advantage of the original two-copy method over the adjusted general method becomes more apparent.

D. Global Dependency and Multiple Copies of FU

Two of the cases that require explicit specification of the length of the global dependency between two instructions are the ones that involve FU conflicts (Chapter II). These instructions can execute either on the same FU where the T_{FU} distance must be enforced or on different FU's where the original dependency should be satisfied.

Suppose two functionally identical instructions I_i and I_j are globally dependent, i.e. $I_j \xrightarrow{G} I_i$, such that their FU conflict is not satisfied by their implicit global dependency. For the two-copy case, the following inequalities should be satisfied to resolve the conflict.

$$T_j - T_i + \gamma*(f_i+f_j) \geq T_{FU_i}$$

$$T_j - T_i + \gamma*(2-f_i-f_j) \geq T_{FU_i}$$

For the general case, the following must be satisfied.

$$T_j - T_i + \gamma*f_{ij} \geq T_{FU_i}$$

These inequalities replace the original global dependency

inequalities. The original pairwise dependency inequalities must still be satisfied.

CHAPTER V

COMPLEXITY OF THE MODEL

To evaluate the use and effectiveness of the model to determine its practical aspects, some performance and complexity analyses must be performed. The factors that distinguish between two different codes are the number of instructions and the types of relationships between those instructions. Use of the latter factor requires the individual analysis of each code. Thus, the number of instructions is the only factor that can be used in a general-case quantitative analysis. However, this number does not provide sufficient information to distinguish between different codes. Therefore, formulation of exact performance measures for codes cannot be established based on quantitative information and other measures of performance must be used.

Even though the number of instructions in a code does not reveal any information about the code itself, it can be used to determine lower and upper bounds on the number of variables and inequalities that are needed to model the codes with that number of instructions. These bounds can specify a general complexity range for the problem and help determine the class of problems to which the problem of this research belongs. Since the number of

variables and inequalities have a direct relationship with the scheduling time and storage requirements of the problem, these bounds specify the type of processor that can be used for this scheduling problem.

A. NP-Completeness of The Model

The problem of optimal scheduling of low-level instruction codes for RRVP's using the IP model is NP-complete due to NP-completeness of job-shop [COFF] and integer programming [PAPA] problems. NP-completeness implies that a polynomial-time algorithm that can solve the problem does not exist. Only 0-1 variables affect the exponential order of the scheduling time since, as shown in Chapter II, the issue time variables need not be forced to integers, and, due to unimodularity of the constraint matrix for minimum-time problems, issue-time variables are ensured to have integer solutions.

A.1. IP Methods

One of the methods that is used in solving IP problems is the enumerative method. This method considers both values of 0 and 1 for each 0-1 variable in the model to produce all possible schedules, and selects the optimal one from among those schedules. Hence, it must consider 2^m cases for a model with m 0-1 variables. Consequently, enumerative method has exponential execution time with respect to the number of variables.

A more common approach in solving IP problems is Branch and Bound (B&B) [BAKE] [STIN]. Even though B&B techniques are usually faster than the enumerative ones, their execution time is highly dependent on the bounding criteria and may still turn out to be nearly exponential. The new developments in implementation of B&B methods, such as reducing the number of 0-1 variables with minimum effect on the optimal solution [SWEE] or replacing inequalities with equivalent ones that create extra restrictions and adding penalties to force faster convergence of the optimal solution [GUIG], may soon reduce the scheduling time for large problems to tolerable levels.

A.2. Integer Solutions

If the model could be solved as an LP problem, the complexity of scheduling of instructions would be reduced. For minimum-time problem, the variables need not be forced to integers and any feasible solution is guaranteed to be integer. If such a condition is proved for the general problem, the IP model will become an LP one that has been shown to have complexity of polynomial order [HACI]. To investigate such a possibility, a similar approach to the one used for minimum-time problem will be used.

Definition 22- The constraint matrix A is a matrix in which the rows represent the constraints and the columns represent the variables of the problem.

The known criteria for unimodularity of matrices, [LAWL], do not apply to the general case constraint matrix A, because it includes some large values θ that are coefficients of 0-1 variables. The sufficient conditions that are shown to determine the unimodularity of a matrix are based on matrices which are composed of elements with values limited to one of -1, 0, or 1. However, since these conditions are sufficient ones, it is still possible that the inverse of any basis of A, say B, is an integer matrix B^{-1}. If such an inverse exists, the problem of $Ax \geq b$ has an integer solution $x=B^{-1}b$ (b is an integer vector).

A square integer matrix B has an integer inverse, $B^{-1}=[B_{ij}]^T/detB$ [NERI], if the determinant of B (detB) is +1 or -1; where B_{ij} is the cofactor of b_{ij} that is the i^{th} row and j^{th} column element of $B=[b_{ij}]$, and Y^T represents the transpose of matrix Y. The cofactors are all integers since the matrix B is an integer matrix. Therefore, if detB is +1 or -1, B^{-1} and the solution will be integers.

Matrix B that is a basis of A is a square non-singular sub-matrix of A, [MURT]. For the minimum-time problem, the columns represent the issue time variables. The 0-1 variables that participate in the basic vector, [MURT], are added to the columns to produce the basis matrix for the general problem. This implies that

a- each row of B is composed of one 1 and one -1, representing the issue-times involved in each

inequality, with the rest of the elements being zeros except for the rows that contain 0-1 variables. Such rows contain an extra non-zero element with value of θ or $-\theta$.

b- each 0-1 variable that is contained in the basic vector occupies a column of B such that all of the elements of that column are zeros except for the θ element.

Condition b is essential because a basis must be a non-singular matrix and cannot include inequalities that can be produced by a linear combination of other inequalities.

From the description of B, the easiest way to calculate detB ($=\sum_j b_{ij}B_ij$ [NERI]) is by using the columns that contain only one non-zero element θ.

$$|\text{detB}| = \theta * B_{ij}$$

where B_{ij} is the cofactor of the i^{th} row j^{th} column element that is θ. $|x|$ represents the absolute value of x. This process can be continued until all of 0-1 variable columns are considered. Suppose there are n 0-1 variables in B.

$$|\text{detB}| = \theta^n * |\text{detB}_\theta|$$

where B_θ is the B matrix with all of its rows and columns containing 0-1 variables eliminated. Then, B_θ is a matrix in which each row is composed of zeros except for one 1 and one -1; it is the minimum-time-problem portion of the general problem. Thus, B_θ is unimodular and $\text{detB}_\theta = +1$ or -1; $\text{detB}_\theta \neq 0$, otherwise it would imply detB=0 or B is a

singular matrix that contradicts the assumption that B is a basis for A. Therefore, $|detB|=\theta^n$.

Since the determinant of B is not +1 or -1, the inverse of B cannot be guaranteed to be an integer matrix. Each cofactor B_{ij} has an absolute value of θ^n or θ^{n-1} for $b_{ij}\neq\theta$ or $=\theta$, respectively. It is because each B_{ij} contains as many θ elements as B except for B_{ij} which is the result of elimination of a row and a column containing a θ. Therefore, $B^{-1}=[B_{ij}]^T/detB$ is a matrix whose non-zero elements are 1, -1, $1/\theta$, or $-1/\theta$. Hence, B^{-1} is not an integer matrix.

Since the inverse of the bases of the constraints matrix are not guaranteed to be integers, the problem remains an IP problem. The size and complexity of this IP problem is highly dependent on the number of variables and inequalities that comprise the model. The determination of such numbers is achieved by studying the complexity of the model.

B. Complexity of The Model

Complexity of the model is determined from the number of variables and inequalities that are generated in order to model the instruction codes. Given the dependency graph of a code, pairs of dependent and independent instructions can be identified. Each dependent or independent pair of instructions contributes a predetermined

number of variables and inequalities to the model. Therefore, the total number of required variables and inequalities can be determined by using the dependency graph. However, for quantitative analysis, only the number of instructions of the code is known from which the inter-instructional relationships cannot be determined. Consequently, the exact number of variables and inequalities for classes of codes with a fixed number of instructions can not be determined.

Even though the number of instructions cannot specify the degree of dependency or independency of the code, it can be used to determine the best (lower) and worst (upper) cases (bounds) that bound the codes with specific number of instructions.

B.1. Bounds for the Original Model

Given a code with n instructions and its respective dependency graph, the number of variables required to model the code can be easily determined. The types of variables that are required by the model are 1) one variable per instruction to determine its issue-time, 2) one 0-1 variable per conflict per pair of independent instructions to determine their order of issue, and 3) one 0-1 variable per pair of instructions with chaining potential to determine if the chaining should or should not occur. Therefore, to determine the number of variables (NV), the number of variables for independent instructions (NI), and the number

of variables for the instructions with chaining potential (NC) must be determined. Having determined NI and NC from the code and its dependency graph, NV can be formulated as follows.

$$NV = NI + NC + n \qquad (32)$$

B.1.a. Bounds on the Number of Variables

Theorem 6- The lower bound on the number of variables is equal to the number of instruction.

Proof- The lower bound can be easily determined from equation (32). Since NI and NC are dependent on the existence of independency and chaining relations, they can vary from one code to another. Then, the lower bound occurs when NI=NC=0. Therefore, NV=n is the lower bound for the number of variables. ∎

A code that represents this lower bound is a totally dependent one such as the one displayed in Fig. 5a.

Lemma 1-The upper bound on the number of variables is equal to MAX(NI)+n'.

Proof- The upper bound on the number of variables can be determined from equality (32) when NI+NC is maximum. Since the representation of an independency relation requires at least as many 0-1 variables as required to represent

---

'MAX(NI) represents the maximum value for NI.

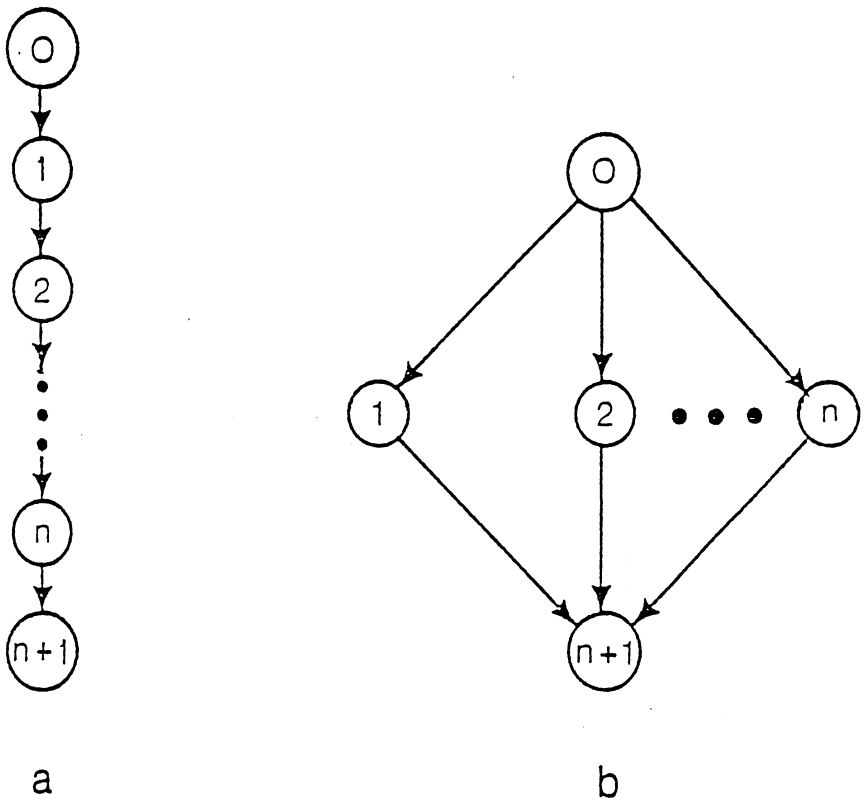


Fig. 5- Totally Dependent and Independent Graphs

(a) a totally dependent code

(b) a mutually independent set of instructions

Nodes 0 and n+1 are the root and terminal nodes

chaining and these two relationships can not be simultaneously applied to a pair of instructions, independency can be substituted for chaining. Therefore, the maximum number of variables are required when the code contains a maximum number of independent instructions. Hence, the upper bound on the number of variables is MAX(NI)+n. ■

An exact number for NI cannot be determined except for the extreme case in which all of the instructions are mutually-independent. This case produces a maximum number of independent pairs of instructions. Fig. 5b displays the graph for a set of mutually-independent instructions.

Lemma 2-MAX(NI) for a set of n mutually-independent instructions is equal to n^2-n.

Proof- A set of n mutually-independent instructions have n*(n-1)/2 pairs of independency relations. The first instruction to be considered must establish an issue time order with n-1 other instructions. The next instruction has n-2 instructions to consider, and so on. Since each pair of independent instructions that are both either S or A type can have access-path and issue conflicts at the same time, each independency relation needs at most 2 0-1 variables; one for each conflict. Therefore, a maximum of 2*[(n-1)+(n-2)+...+1]=n*(n-1) 0-1 variables are needed. ■

Lemma 3-The number of mutually-independent instructions

cannot exceed $\sum_{i=1}^{N} P_i$.

Proof- There exist P_i identical registers in any of the i=1,2,...,N register groups. Each instruction uses at least one result-register. To be mutually independent from other instructions, its registers must not be used by any other instructions. Therefore, the number of mutually-independent instructions cannot exceed $Q=\sum_{i=1}^{N} P_i$. ■

In practice, an average instruction possesses more than one register; it requires a result and one or more operand registers. However, since the intention is to find the maximum number of mutually independent instructions, the use of the extreme case in which each instruction requires only one register is appropriate.

Theorem 7- The upper bound on the number of variables has a polynomial complexity with a degree of at most 2.

Proof- According to Lemma 1, the upper bound for NV is MAX(NI)+n. From Lemma 2 and Lemma 3, it is concluded that $MAX(NI)=n^2-n$ for $n\leq Q$ where all of the instructions are mutually independent and NC=0. Therefore, the upper bound is equal to n^2. For n>Q, a number of dependency relations replace the independency ones. This number cannot be determined since it is highly dependent on the type of the code. However, since the code does not contain a set of mutually-independent instructions, $NV<n^2$ for the latter case. Therefore, n^2 is an upper bound on NV for any n. ■

B.1.b. Bounds on the Number of Inequalities

As the number of 0-1 variables, the number of inequalities that model a code is highly dependent on the number of independency relations and the characteristics of the code. A dependency relation is represented by either an inequality of type (1) or the set of three inequalities of type (14). An independency relation is represented by a set of two inequalities of type (12). In the case of scalar or address instructions, it is possible to use two sets of these inequalities if access-path conflict must be settled in addition to other conflicts. Therefore, the number of inequalities is a multiple of the number of dependency and independency relations.

Theorem 8- The lower bound on the number of inequalities is equal to the number of instructions.

Proof- Since the representation of a dependency relation requires one inequality where as the representation of an independency relation requires two or more inequalities, a minimum number of inequalities is generated for a totally dependent code in which each instruction is dependent only on its preceding instruction (Fig. 5a). In this case, each instruction is represented by only one dependency inequality. Therefore, the lower bound for the problem is equal to the number of instructions. ∎

Theorem 9- The upper bound on the number of inequalities is

equal to 2*n*(n-1).

Proof- An independency relation requires at least twice as many inequalities as a dependency relation does. Thus, the upper bound is reached when the code contains a set of mutually-independent instructions. Since each 0-1 variable is used in two inequalities, the upper bound on the number of inequalities is twice the upper bound on the number of 0-1 variables, that is, $2*n^2-2*n$. ∎

B.2. Bounds for Register Reassignment Model

To determine the effect of the register reassignment on the order of complexity of the model, the increase in the number of variables and inequalities, due to reassignment, must be determined. The number of added variables is determined from the number of registers available for reassignment to each instruction. The determination of the number of inequalities requires some knowledge about the number of those registers and the relationship changes caused by their reassignment. Therefore, an important piece of needed information is the number of candidate registers in the CS of each result register.

B.2.a. Bounds on the Number of Variables

The number of registers in the CS of a result register determines the number of 0-1 variables that are required to represent the reassignment of that register.

Each element in the CS of a register is represented by a 0-1 variable that specifies the status of the assignment of that element. Therefore,

$$NRRV = \sum_{i=1}^{n} |CS_{RR_{?i}}| \qquad (33)$$

where NRRV is the number of variables required for register reassignment and $|X|$ represents the number of elements in X (cardinality of X).

Specification of an explicit formula to calculate NRRV requires some knowledge about the specifics of the code. It is clear that if a number of fixed result registers exist, some registers in the CS's of instructions can be explicitly eliminated due to their assignment invalidity caused by the inclusion of a fixed register in their RUR (Theorem 5). Thus, to determine $|CS|$ for any instruction, some information on the number of fixed registers and their effect on reassignment of others is necessary. On the other hand, if all of the registers are reassigned, $|CS_{RR_{?i}}|$ is equal to the number of registers with the same type as I_i. In this case, the invalidity of a reassignment cannot be explicitly determined since the registers are not explicitly assigned. However, since the number of registers that are available to different types of instructions varies depending on the register type, the calculation of NRRV is still dependent on the number of instructions of each type that are used in the code.

Even though a general formulation of NRRV requires more knowledge about the code than just the number of instructions in it, a lower bound and an upper bound for NRRV can be determined based on that number. Such bounds can be determined by calculating NRRV when minimum and maximum number of registers are reassigned. These two extreme cases occur when the number of pairs of dependent and independent instructions in the code are maximized, respectively.

Theorem 10-The lower bound on the number of variables is zero.

Proof- The lower bound for this problem occurs for the code in which each instruction is data dependent on its preceding instruction. In this case register reassignment will not alter the order of instructions and cannot force any dependency relation to independency one. Therefore, the reassignment should be ignored, and no 0-1 variables are needed. ∎

Theorem 11-The upper bound on the number of variables is equal to $n*\underset{i}{MAX}(P_i)$.

Proof- The upper bound is equal to the maximum number of 0-1 variables that are needed to specify the assignment of registers to instructions. Since each candidate register requires a 0-1 variable, a complete reassignment of all registers requires the maximum number of 0-1 variables (one

variable per each register of each CS). For an exact formulation of this number, the number of instructions of each instruction type should be found by inspection of the code. That is because each register type has a number of registers available for use by the proper type instructions and the number of such registers varies from one register type to another. Suppose the number of instructions of type j is NIT_j where j=1,2,...,N. The number of variables needed to model the reassignment of the result register of I_i of type j is equal to the cardinality of its CS which is equal to P_j. Therefore,

$$NRRV = \sum_{k=1}^{N} P_k * NIT_k \qquad (34)$$

A code with n instructions may contain a number of different types of registers. The upper bound occurs when CS of each instruction is assigned $MAXR=\underset{i}{MAX}(P_i)$ registers where MAXR represents the maximum of P_i's for i=1,2,...,N. Therefore, the upper bound for NRRV is n*MAXR. ∎

The upper bound on the number of variables required to model a code using register reassignment is equal to the upper bound without reassignment plus the number of variables required by the reassignment method. Reassignment of registers produces sets of independency relations between instructions. These sets differ for different register assignments. It is likely that a number of 0-1 variables are required to monitor the new independency relations, in

addition to the ones already in use by the original model. This must be the case for an effective register reassignment. However, the upper bound on the number of variables for the original model already considers the maximum number of required variables to model the code. Therefore, that upper bound is increased only by $n*MAX_i(P_i)$ when the register reassignment is considered.

B.2.b. Bounds on the Number of Inequalities

Theorem 12-The lower bound on the number of inequalities is equal to zero.

Proof- The lower bound on the number of inequalities, as for NRRV, occurs for a totally data-dependent code for which register assignment has no effect on the order of instructions. Hence, no inequalities are needed to model the register reassignment. ∎

Lemma 4-The upper bound on the number of inequalities is equal to the sum of upper bounds on the number of inequalities needed to represent register-dependency and independency relations plus n.

Proof- The reassignment of registers changes the relationships between instructions such that two instructions can be register-dependent or independent based on the assignment, except for data-dependent instructions. That is because two register-dependent instructions that share one or more registers will become independent when one

of the instructions reassigns its registers such that no registers are shared between those instructions. Similarly, two independent instructions may become register-dependent when their register assignment is altered.

Since the register-dependency and independency relations between two instructions cannot occur simultaneously, the upper bound on the number of register-dependency (independency) inequalities is not affected by independency (register-dependency) relations. On the other hand, data dependency relation between two instructions eliminates their need for register reassignment formulation. Also, each CS has a number of registers from which only one can be assigned. To guarantee this restriction, n equations of type (17), one per instruction, must be used. Therefore, the upper bound for the number of inequalities occurs when no data dependency relations exist and is equal to the sum of bounds on the number of register-dependency and independency inequalities plus n. ∎

To preserve the generality of the discussion, suppose that each instruction is composed of one result register and x operand registers.

Lemma 5-An upper bound on the number of register-dependency inequalities is equal to MAXR*x*n*(n-1)/2.

Proof- The register-dependency relation must be considered between each instruction and the instructions that precede

it in the sequential order of the code. The dependency between I_i and I_j is guaranteed if $RR_{?j}$ is the same as any of the operand registers of I_i. The inequalities that display this possibility are inequalities of (18). Each register in $CS_{RR_{?j}}$ is checked against the ones in $CS_{OR_{?i}}$'s. Therefore, $|CS_{RR(?i)}|*x$ inequalities are required per pair of instructions. this number is maximized when all registers have MAXR candidate registers. When all of the instructions preceding each instruction are considered, this number sums up to MAXR*x*n*(n-1)/2. ■

Lemma 6-An upper bound on the number of independency inequalities is equal to $2^{MAXR}*n*(n-1)/2$

Proof- The independency relations are considered between each instruction and any other instruction. There are a maximum of n*(n-1)/2 possible combinations of pairs of instructions that must use inequalities of (23). The maximum number of inequalities that are needed to represent each pair of instructions is equal to 2^{MAXR}. Consequently, the maximum number of inequalities needed to model independency is $2^{MAXR}*n*(n-1)/2$. ■

Theorem 13-An upper bound on the number of inequalities is equal to $n*(n-1)*(MAXR*x+2^{MAXR})+n$.

Proof- According to Lemma 4, the upper bound is equal to the sum of the results of Lemma 5 and Lemma 6 plus n. Therefore, the upper bound is equal to the following.

$$MAXR*x*n*(n-1)/2+2^{MAXR}*n*(n-1)/2+n=n*(n-1)*(MAXR*x+2^{MAXR})+n$$

that is a polynomial order of degree two, $O(n^2)$. ■

B.3. Bounds for the Multiple-Identical FU's

The multiple FU assignment is considered for two-copy and general cases. Since the special case when two copies of each FU are available has been formulated differently from the general case, their associated complexities are different.

Multiple copy FU affects independency relations, only. The availability of extra copies of FU's does not affect the dependency relations since the pair of dependent instructions cannot be concurrently executed. Since the number of instructions that compete for a specific FU varies from code to code, the exact formulation of the number of required variables and inequalities is not possible. Therefore, the upper and the lower bounds will, again, be determined to display the complexity of the problem.

B.3.a. Bounds on the Number of Variables

Theorem 14-The lower bound on the number of variables for either the general case or the two-copy case is equal to zero.

Proof- The lower bound occurs when the code does not contain any independent instructions or the independent

instructions do not use the same FU's. Therefore, no extra variables are needed to control the concurrent execution of the instructions. ■

Theorem 15-For the two-copy case, the upper bound on the number of variables is equal to n.

Proof- The upper bound occurs when every instruction must compete for a copy of its needed FU. That is because each instruction is assigned one 0-1 variable to specify the copy of FU on which it must execute. Such a variable is not needed if no other jobs should be concurrently executed on the same FU. Therefore, the upper bound is equal to n variables, one per instruction. ■

Theorem 16-For the general case, the upper bound on the number of variables is equal to n*(n-1)/2.

Proof- Since each pair of independent instructions that compete for an FU are assigned one 0-1 variable to reflect their concurrency, the maximum number of variables are needed when a maximum number of mutually-independent instructions have FU conflicts. That occurs for a code with mutually-independent instructions that compete for the copies of one FU. In this case n*(n-1)/2 variables govern the concurrent or sequential execution of the instructions.■

The upper bound on the number of variables required for the general case is not dependent on the number of copies, since the number of copies does not affect the

independency between the instructions. However, if the number of copies is larger than the number of mutually independent instructions, all of the jobs can run concurrently and the number of required variables is zero.

Comparing the complexity of the general case with that of the two-copy case shows that the number of variables required for the formulation of the former case with two copies per FU (n*(n-1)/2) is noticeably larger than the number of variables needed for the latter one (n).

B.3.b. Bounds on the Number of Inequalities

Theorem 17-The lower bound on the number of inequalities for either the general or the two-copy case is zero.

Proof- Same as the proof for Theorem 14.

Theorem 18-For the two-copy case, the upper bound on the number of inequalities is equal to n^2-n.

Proof- The upper bound on the number of inequalities occurs when a maximum number of pairs of independent instructions exist. Each pair of independent instructions that use the same FU are represented by three pairs of inequalities. Two pairs of these inequalities have already been considered by the upper bound on the number of inequalities of the original model. Thus, one pair of inequalities per pair of independent instructions is the contribution of the two-copy case to the complexity of the problem. Then, the upper

bound occurs when all of the instructions in the code are mutually independent and use the same FU. Therefore, the upper bound for these inequalities is $6*n*(n-1)/2$ and the upper bound increase for the model due to availability of two copies of each FU is $2*n*(n-1)/2=n^2-n$. ∎

Theorem 19-For the general case, the upper bound on the number of inequalities is equal to $\binom{n}{m+1}+\binom{n}{3}$.

Proof- The upper bound occurs when the maximum number of independent instructions exist. Comparing inequalities of (28) with the ones of (3) shows that the effect of multiple copies is the addition of an extra condition to each pair of inequalities. Hence, the number of inequalities that represent independent instructions is not altered. However, a number of inequalities are added to control the use of the multiple copies.

The inequality of (29) must be present for each group of m+1 mutually independent instructions using the same FU, where m is the number of copies of the FU. Hence, a maximum number of $\binom{n}{m+1}=n!/(m+1)!(n-m-1)!$ inequalities are needed when all of the instructions are mutually independent and use the same FU. Also, using similar reasoning, inequality of (30) is used a maximum of $\binom{n}{3}=n!/(n-3)!3!$ times. Therefore, an upper bound on the number of inequalities is equal to $\binom{n}{m+1}+\binom{n}{3}$, that is, a polynomial of degree m+1, $O(n^{m+1})$. ∎

B.4. Summary of Complexities

A summary of the complexity of the bounds on the number of variables and inequalities required to model the code are displayed in Fig. 6.

		Complexity of Variables		Complexity of Inequalities	
		UB	LB	UB	LB
Original Model		$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$
Register Reassignment		$O(n)$	0	$O(n^2)$	0
Multiple Identical FU	Two-copy Case	$O(n)$	0	$O(n^2)$	0
	General Case	$O(n^2)$	0	$O(n^{m+1})$	0

Fig. 6- Complexity of Bounds on Variables and Inequalities

UB represents the upper bound
LB represents the lower bound
n is the number of instructions
m is the number of copies of each FU

CHAPTER VI

IMPLEMENTATION OF THE MODEL

The model of Chapter II has been used in scheduling a number of Cray Assembly Language (CAL) codes to be processed on a CRAY-1S machine. A program has been developed that receives the CAL code and produces the inequalities that represent the code. Then, these inequalities are supplied to an IP package that solves those set of inequalities and produces optimal issue-times for the instructions. The code is then rearranged to match the optimal schedule.

A. Pre-Processor

The Pre-Processor is a Fortran program that is capable of generating the inequalities that represent a CAL code with the architectural features of CRAY-1S. It accepts all of the CAL instructions of CRAY-1S and produces two different outputs that contain 1) information about the code and 2) information for the IP. The first output will be explained later. The second output represents the set of inequalities that model the code. This output is prepared with the proper formatting to be used by the IP package. The types of outputs that are generated by the Pre-Processor depend on the requirements of the user.

The Pre-Processor provides the user with a number of options that can cause a number of different types of schedules to be produced for a code. These options are best expressed as follows.

1- The user may request the code to be modeled as a straight code or the loop modeling of Chapter II with floating JUMP to be used for all of the inner loops in the code. The default is straight code.

2- The user may select the vector length that is used for register and FU reservations related to vector instructions. The default is 64; maximum vector length allowed for CRAY-1S vector instructions.

3- The user may select to have the formal output generated or ignored. The default is to ignore.

The first output is generated by default and it provides useful information about the code and the model representing it. The following information are produced by this output.

a- A list of instructions of the code and their respective instruction numbers, operation codes, and registers.

b- A dependency table that displays the pairwise dependency of each instruction on its preceding instructions and the length of dependency.

c- A list of instructions that precede an instruction and are independent from it, for each instruction of the code.

d- A list of dependency and independency inequalities and loop equalities (equality (11)).

The above information is used in producing the dependency graph for the code and determining the number of inequalities and variables that are required to model the code.

B. Integer Programming Package

The formal output of the Pre-Processor is supplied to an IBM-owned integer programming package, called MPSX/MIP/370. The package uses a two-phase process to solve any given integer programming problem. In phase one, the problem is processed as a Linear Programming (LP) problem by MPSX/370. The second phase of the process uses the MIP/370, which is a mixed IP to generate integer solutions. The package provides the options to limit the processing time and to start the first phase with a user-provided starting solution for faster convergence of the optimal solution.

Using an LP to solve the problem is equivalent to solving a problem in which all of the conflicts are relaxed, the variables are assumed to be real numbers, and no attempt is made to force an integer solution. So, only the minimum-time variables result in integer values, as are shown in Chapters II and V. In the optimal LP solution, the 0-1 variables are limited between 0 and 1. Such a limit must exist since a negative or a larger than 1 value for a 0-1 variable will cause the issue-time difference between

instructions to become very large. Because this large value is also reflected by the objective value, such a solution cannot be optimal. Therefore, since the LP solution does not result in integer optimal solutions, it does not have any significance except for providing the IP with a starting minimum solution.

The MIP phase starts with the optimal LP solution and using a B&B method produces integer optimal solutions. MIP forces the variables to integers such that the resulting integer solutions are feasible. When such solutions are found, it displays each one that has a lower objective value than the one last displayed. When no solution can be found with a smaller objective value than the previous ones, MIP stops with the latest displayed solution as the optimal solution. Unfortunately, since no one specific test can determine if a feasible solution is optimal, the MIP spends a large portion of its execution time proving the optimality of the latest displayed solution by checking other possible solutions and rejecting them due to their larger objective value.

C. Scheduling Experiments

Full access to the Pre-Processor and a limited access to the IP package provided the opportunity to conduct a few experiments in optimal scheduling of CAL codes. Due to time requirements of the MIP and limited amount of access time available, only a few fully processed experiments have

been performed. These experiments resulted in production of optimal schedules with relatively significant computation time reductions.

The first experiment is a scalar code with 12 instructions from a linear algebra code for LU factorization of a 2x2 matrix. The timing results for scheduled and unscheduled codes (Experiment #1 in Fig. 7) are not significantly different, since a high degree of dependency exists between instructions and rearrangement is limited. Experiment #2 involves the simultaneous factorization of 2x2 matrices. This finds application in the solution of (two) coupled partial differential equations, where different grid points are to be simultaneously eliminated but data storage or short vector length does not favor a vector mode solution. Two copies of the unscheduled code of Experiment #1 are concatenated, but the codes use different scalar registers and so are independent. The intention of scheduling is to interlace the independent instruction sequences. The results of Fig. 7 show that the timing is more than halved using the MIP scheduling.

Another important application in scientific codes is the scheduling of assembly-coded library routines. In Experiment #3, a 64-bit vector integer multiply, obtained from Cray Research, Inc., was rescheduled for a vector length of 64. The speedup was only 9%, since some care had already been taken with the hand-coded version.

The CPU time (on AMDAHL 470 V/6) spent on optimal scheduling of the codes of these experiments are displayed in Fig. 7. The first experiment with the fewest number of instructions and a relatively high degree of dependency requires a minimal amount of CPU time. In fact, the LP phase produced integer results and the use of MIP phase was not necessary. The second and third experiments with higher numbers of instructions and degrees of independency require the use of the MIP phase. The scheduling times displayed in Fig. 7 reflect the amount of time that was spent in finding the optimal solution. However, the MIP continued, in order to verify the optimality of the latest reported solution. For the second and third experiments, the execution was intentionally interrupted at 1 minute and 0.5 minute of CPU times, respectively. These relatively small experiments display that a high price may be paid to achieve optimality.

D. Complexity Experiments

To examine the complexity of some arbitrary problems, a number of codes have been processed by the Pre-Processor. The results of such experiments are displayed in Fig. 8. The number of variables and inequalities that are required to model the codes have been determined from the first output of the Pre-Processor. The upper and lower bounds on NV and NI have been calculated from the formulations of Chapter V.

Experiment #	# of CAL Instructions	Unscheduled Time (C1)	Scheduled Time (C1)	Reduction Ratio (%)	Scheduling Time (min)
1	12	87	68	22	0.01
2	24	165	74	55	0.58**
3	36	876*	798	9	0.37***

Fig. 7- Timing results for sample experiments.

(timing results are in clocks)

(* hand scheduled code).
(** interrupted at 1.0 minute)
(*** interrupted at 0.5 minute)

n	NV	UB NV	NI	UB NI
12*	17	144	24	264
15	31	225	55	420
24*	197	576	376	1104
26	62	676	179	1300
32	398	1024	800	1984
36*	238	1296	485	2520
61	980	3721	2025	7320

Fig. 8- Complexity information for CAL codes

(* these codes are the ones of Fig. 7)

n represents the number of instructions
NV represents the number of variables
NI represents the number of inequalities
UB represents the upper bound;
either on NV or on NI

E. Non-Linear Programing (NLP)

Due to the NP-completeness of the IP model, efforts to find a method that can produce optimal schedules faster than the IP is worthwhile. Non-linear programming techniques were thought to provide such properties. However, some experiments with the locally available NLP package called MINOS produced unsatisfactory results. MINOS is an NLP package capable of solving problems with linear constraints and linear and/or non-linear objective function.

The IP model can be solved by an NLP if the 0-1 variables are forced to their limits. One way of forcing a 0-1 variable y to 0 or 1, is to include y in the objective function that should be minimized as $\phi*y*(1-y)$. Having assigned a large positive integer to ϕ, the minimum objective can be achieved when y is either 0 or 1 which eliminates the above term from the objective function. However, NLP is a local optimum finder and the experiments proved that to be the case. Therefore, because finding optimal solutions is the goal of this research, local optimums are not acceptable and the use of NLP is rejected.

F. Conclusion

The following conclusions can be reached from the experiments of the previous sections.

1- The experiments of Fig. 8 display that the number of variables and inequalities used for modeling are

considerably smaller than their respective upper bounds. This is due to the presence of precedence constraints that have been eliminated in order to calculate the upper bounds.

2- Comparing the information on the codes with 15, 24, and 26 instructions in Fig. 8, it can be concluded that the number of instructions in the code does not reflect any information about intra-code relationships that are reflected in NV. Such a conclusion can be justified by observing that the code with 26 instructions uses a higher number of variables and inequalities than either the code with smaller number of instructions (n=15) or the one with higher number of instructions (n=26). A similar observation can be made by comparing the codes with 26, 32, and 36 instructions.

3- The experiments of Fig. 7 show that the percentage of reduction in computation time of the code with 24 instructions is higher than that of the code with 12 instructions or the one with 34 instructions. In other words, maximum improvement is achieved in scheduling the code that contains the highest proportion of independent instructions (Fig. 8). This is not a surprise, since independent instructions provide a higher flexibility in reordering of the code.

4- The experiments of Fig. 7 and Fig. 8 display that the code with higher proportion of independent instructions requires a larger amount of CPU time for scheduling.

Among the above conclusions, the second one must always hold since a contrary conclusion can be rejected using the above example.

CHAPTER VII

CONCLUSION

An integer programming model has been developed that considers the architectural features of RRVP's and is capable of producing optimal schedules for low-level codes of these processors. Optimal schedules are produced by using integer programming techniques to resolve the resource sharing conflicts (Chapter II). These conflicts can cause a hold issue on some instructions and prolong the total computation time. The effect of hold issue on the instructions can be magnified if, due to some conflicts, the issue is held for an instruction in an instruction-loop. The original model produces exact times at which the instructions should be issued to achieve minimum computation time. The model is developed for RRVP's with a limited number of non-identical FU's with unequal processing times, single instruction issue, and a limited number of access paths. Optimal scheduling of instruction loops based on the effect of instructions of one iteration on the instructions of the next iteration is also included in the model.

The model has been extended to optimally reassign registers in addition to optimal instruction sequencing

(Chapter III). This extension eliminates the influence of the register dependencies between instructions that can force unnecessary dependencies and prevent instruction order reversals necessary in reducing computation time. The extended model produces optimal schedules with respect to instruction sequencing and register assignment.

The model is further extended to permit the use of multiple copies of FU's (Chapter IV). This extension eliminates the restriction of availability of only one FU of each type that, in turn, allows the model to adjust itself to a wider range of problems. The two extensions can separately or jointly be added to the model.

Chapter V has been focused on determining the complexity of the time required to schedule a code (complexity of the model), and the complexity of the number of required variables and inequalities. It has been shown that the problem is NP-Complete and the use of integer programming cannot be avoided. The size and complexity of this IP problem, however, is highly dependent on the number of variables and inequalities that are used by the model to represent the code. The exact number of these variables and inequalities cannot be determined without extensive knowledge about the code. Therefore, upper and lower bounds for these numbers have been determined to provide some insight into the complexity of this problem.

Chapter VI reports the results of a number of

experiments that have been conducted using the model of Chapter II. The experiments conducted using an integer programming package resulted in optimal schedules. The reduction in total computation time and the amount of CPU time used for scheduling a number of Cray assembly codes have been reported. These experiments show that a higher degree of independency in the code may result in higher computation time. Due to the presence of inter-instructional dependencies, the experiments on the complexity of the model resulted in a considerably smaller numbers for variables and inequalities than their upper bounds suggest. And, the use of non-linear programming in optimal scheduling was rejected due to generation of schedules that are locally optimal and do not guarantee the minimization of the total computation time.

A. Applications of The Model

The IP model that is developed for scheduling low-level codes of RRVP's is a general one that can be partially or wholly applied to other problems. The model is composed of a number of constraints and conflicts that are represented by groups of inequalities. These constraints and conflicts do not individually represent any specific RRVP, and their effect is to prevent two instructions or jobs from being simultaneously started, finished, or processed. Thus, any job-shop problem that is composed of such conflicts and constraints can be modeled by a

combination of appropriate inequalities. For example, consider the class of memory-to-memory vector processors for which the flow of vector elements is from memory to FU's and back to memory. These vector processors can use the formulations of the model if they regard the memory as a special type vector register. Also, the extensions of the model can be independently applied to other problems.

B. Reducing Scheduling Time

Simple implementation considerations can effectively reduce scheduling time without affecting the optimality of the solution. One such consideration is the elimination of some of the redundant optimal solutions. For any code, there exist a number of optimal register assignments. For example, consider the instructions at the top of the dependency graph. Since these instructions do not have any predecessors, for some of them, the assignment of any one of the registers in their CS's will result in the same optimal total computation time. Explicit assignment of randomly selected registers from their CS's will eliminate the need to consider a number of redundant solutions. A modified version of this procedure may be applied to some of their succeeding instructions in the graph. Also, similar procedures can be used in assigning multiple copies of the FU's. i.e. explicitly assign specific copies of the proper type FU to one or more of the MIFI instructions to reduce the number of possible ways that these instructions can use

the available copies. Such pre-assignment of a few instructions forces the selection of a specific optimal solution among a number of such solutions that may reduce the time required to schedule the code.

C. Research Suggestions

The efforts of this research have resulted in development and study of a model that can produce optimal schedules for low-level codes of RRVP's. To help direct interested researchers to the study of related subjects, a few research suggestions in continuation of this one are to follow.

C.1. Experiments with the Extensions

The experiments in optimal scheduling of CAL codes for CRAY-1S using the Pre-Processor and the MPSX/MIP are limited to the original model and do not include its two extensions of Chapters III and IV. It is of interest to extend the Pre-Processor to include these extensions of the model in order to produce optimal schedules that are not relative to register assignment or limited to one FU of each kind.

Even though the complexity of the model (as described in Chapter V) is increased when its extensions are included, it does not automatically imply that such additions increase the IP time required for scheduling. That is due to possibility of addition of extra restrictions that

can force a faster convergence of the optimal solution. Therefore, it is also of interest to realize the practical effects of such extensions in scheduling time of CAL codes.

C.2. Specialized IP

Looking at the problem from IP point of view, it seems that due to the special structure of the model, it might be possible to produce a specialized IP that can considerably reduce the scheduling time. An IP package that is designed to solve a large variety of problems must choose generalized criteria that can be applied to any problem. However, the special structure of the model of Chapter II (at most three variables per inequality or row that creates a highly sparse constraints matrix) suggests that development of a special IP that is specifically tailored for this model might have a considerable impact in reducing scheduling time. Therefore, development of such an IP package is suggested.

C.3. Heuristic Scheduling

Since the goal of this research has been optimal scheduling of instructions, no attention has been paid to non-optimal methods of instruction scheduling. The high cost of optimal scheduling due to the NP-Complete nature of the problem provides some incentive for the study of heuristic solutions.

REFERENCES

REFERENCES

[AHO] Alfred V. Aho, Jeffrey D. Ullman, "**Principles of Compiler Design**," Addison-Wesley Publication Co., 1977

[BAKE] K. R. Baker, "**Introduction to Sequencing and Scheduling**," Wiley, N.Y., 1974

[BEAT] J. C. Beaty, "**Register Assignment Algorithm for Generation of Highly Optimized Object Code**," IBM Journal of Research and Development, Vol. 18, NO. 1, Jan. 1974

[COFF] E. G. Coffman et al., "**Computer and Job-Shop Scheduling Theory**," Wiley, N.Y., 1976

[COHA] William L. Cohagan, "**Vector Optimization for the ASC**," Proceedings of the 7th Annual Princeton Conference on Information Sciences and Systems,1973

[CRAY] **CRAY-1S Hardware Reference Manual**, CRAY-1 Computer Systems Pub. No. HR-0808, Cray Research Inc., June 1980

[CSIM] D. A. Orbits, "**A CRAY-1 Simulator**," Report # 118, Systems Engineering Laboratory, University of Michigan, Sep. 1978

[DAY] W. H. E. Day, "**Compiler Assignment of Data Items**," IBM Systems Journal, Vol. 9, No. 4, 1970

[FREI] R. A. Freiburghouse, "Register Allocation Via Usage Counts)," Communications of ACM, Vol. 17, No. 11, Nov. 1974

[FSIM] **APSIM/APDBUG Reference Manual**, FPS Publication No. 860-7364-003, Floating Point Systems Inc., Beaverton, Oregon, 1979

[GREE] H. H. Greenberg, "**A Branch and Bound Solution to the General Scheduling Problem**," Operations Research, Vol. 16, No. 2, March-April 1968

[GUIG] Monique Guignard, Kurt Spielberg, "**Logical Reduction Methods in Zero-One Programming**," Operations Research, Vol. 29, No. 1, Jan.-Feb. 1981

[HACI] L. G. Hacijan, "**A Polynomial Algorithm in Linear Programming**," Soviet Mathematics Doklady, Vol. 20, No. 1, 1979

[KUCK] D. J. Kuck, "**The Structure of Computers and Computations**," Vol. 1, Wiley, N.Y., 1978

[LAWL] E. L. Lawler, "**Combinatorial Optimization Networks and Matroids**," Holt, Rinehart and Winston Publication, 1976

[MURT] K. Murty, "**Linear and Combinatorial Programing**,"

Wiley, N.Y., 1976

[NELS] Harry L. Nelson, "**Timing Codes on the CRAY-1: Principles and Applications**," Lawrence Livermore Laboratory, Computer Documentation UCID-30179, Version 2, May 1981

[NERI] Evar D. Nering, "Linear Algebra and Matrix Theory," John Wiley & Sons, second edition, 1970

[PAPA] Christos H. Papadimitriou, "**On the Complexity of Integer Programming**," Journal of ACM, Vol. 28, NO. 4, Oct. 1981

[PRIT] A. A. B. Pritsker, L. J. Watters, and P. M. Wolfe, "**Multiproject Scheduling with Limited Resources: a Zero-One Programming Approach**," Management Sci., Vol. 16, No. 1, Sep. 1969

[SAHN] Sartaj K. Sahni, "**Algorithms for Scheduling Independent Tasks**," Journal of ACM, Vol. 23, No.1, Jan. 1976

[SETH] Ravi Sethi, "**Complete Register Allocation Problems**," S.I.A.M. Journal on Computing, Vol. 4, No. 3, Sep. 1975

[STIN] J. P. Stinson, E. W. Davis, and B. M. Khumawala, "**Multiple Resource-Constrained Scheduling Using Branch and Bound**," AIIE Trans., Vol. 10, No. 3, Sep. 1978

[SWEE] D. J. Sweeney, R. A. Murphy, "**Branch and Bound Methods for Multi-Item Scheduling,**" Operations Research, Vol. 29, No. 5, Sep.-Oct. 81.

[THOM] B. G. and G. L. Thompson, "**Algorithms for Solving Production Scheduling Problems,**" Operations Research, Vol. 8, No. 4, July 1960

[TOKO] M. Tokoro, E. Tamura, and T. Takizuka, "**Optimization of Microprograms,**" IEEE Trans. on Comp., Vol. C-30, No. 7, July 1981

[TRAN] **IEEE Transactions on Computers**, Vol. C-30, No. 7, July 1981

[WIES] Jerome D. Wiest and Ferdinand K. Levy, "**A Management Guide to PERT/CPM,**" Prentice-hall, N. J., 1977