

Copyright © 1985, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A DESIGN METHODOLOGY FOR VLSI PROCESSORS

VOLUME I

by

Joan Marie Pendleton

Memorandum No. UCB/ERL M85/88

21 November 1985

A DESIGN METHODOLOGY FOR VLSI PROCESSORS

VOLUME I

by

Joan Marie Pendleton

Memorandum No. UCB/ERL M85/88

21 November 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering  
University of California, Berkeley  
94720

*title page*

Research sponsored, in part, by Defense Advance Research Projects Agency (DoD) Contract No. N00039-83-C-0107 and, in part, by a Fellowship from Eastman Kodak Corporation.

*Back of  
Title Page*

A DESIGN METHODOLOGY FOR VLSI PROCESSORS  
VOLUME I

by  
Joan Marie Pendleton

Memorandum No. UCB/ERL M85/88

21 November 1985

ELECTRONICS RESEARCH LABORATORY  
College of Engineering  
University of California, Berkeley  
94720

Research sponsored, in part, by Defense Advance Research Projects Agency (DoD) Contract No. N00039-83-C-0107 and, in part, by a Fellowship from Eastman Kodak Corporation.


# A Design Methodology for VLSI Processors

Joan Marie Pendleton

## ABSTRACT

A design methodology for VLSI processors has been developed. It is based on five major design levels - microarchitecture, functional block, circuit, interconnect, and process - and the interactions between them. In addition to top-down synthesis, this method formally incorporates the feedback of information from the lower design levels to the higher levels. A preliminary design phase that considers the effects of the lowest levels - circuit, interconnect, and process - on design at the highest level - microarchitecture - is described. After preliminary design, design alternates between synthesis and analysis steps as the designers proceed from the highest level to the lower levels.

SOAR. (Smalltalk on a RISC), a 32 bit microprocessor designed for the efficient execution of compiled Smalltalk provides a case study of this methodology. The chip, implemented in 4 micron, single-level metal NMOS technologies, has a cycle time of 400 ns. Pipelining allows an instruction to start each cycle with the exception of loads and stores. The processor contains 35,700 transistors, is 320x432 mils, dissipates 3 watts, and is assembled in an 84-lead pin grid array package. The methodology that included a large CAD effort provided functioning chips on first silicon.

  
Chairman

## Table of Contents

Preface	1
Chapter 1- Introduction	6
1. VLSI Issues	6
2. Thesis Organization	8
Chapter 2- Present Methodologies	10
1. Design Levels	10
2. Hierarchy	12
3. Design Levels and Hierarchy	14
4. Mead-Conway Style	15
5. CMU-DA System	17
6. Bell Laboratories- Bellmac-32 Techniques	21
7. IBM- Philo VLSI Design System	23
8. Summary	25
9. References	27
Chapter 3- Design Levels	29
1. Microarchitecture Level	30
2. Functional Block Level	34
3. Circuit Level	37
4. Interconnect Level	40
5. Process Level	42
6. Interrelationship Overview	43
6.1. External Inputs	45
6.2. Iteration	50



6.3. Unidirectional Design	52
7. Methodology Implications	53
8. Summary	55
9. References	58
Chapter 4- Design Methodology	59
1. Preliminary	60
2. Synthesis	66
2.1. Microarchitecture Synthesis	71
2.2. Functional Block Synthesis	74
2.3. Circuit Synthesis	75
2.4. Interconnect Synthesis	76
3. Analysis	77
4. Optimization	83
5. Methodology	85
Chapter 5- External Inputs- SOAR Case Study	99
1. Architecture	99
1.1. Data Types	99
1.2. Word Size	101
1.3. Addressing Modes	102
1.4. Register Organization	103
1.5. Instruction Set	108
1.6. Internal Exceptions and Traps	115
2. System Specifications	119
2.1. Memory Requirements	120
2.2. Clocking	120

2.3. External Interrupts and Wait	121
2.4. Fast Shuffle Control	121
2.5. Reset	123
2.6. Loading Characteristics	123
2.7. Size and Power	124
3. Process	124
3.1. Devices and Device Parameters	125
3.2. Layers and Layer Parameters	128
3.3. Design Rules	129
4. References	130
Chapter 6- Preliminary Design- SOAR Case Study	132
1. Preliminary Circuit	135
1.1. Desired Functions	135
1.1.1. ALUs	135
1.1.2. Storage	137
1.1.3. Random Logic	138
1.1.4. Drivers	139
1.1.5. Summary	140
1.2. Circuits Available	142
1.2.1. ALUs	142
1.2.2. Storage	150
1.2.3. Random Logic	154
1.2.4. Drivers	159
1.3. Initial Microarchitecture Design	166
1.4. Further Preliminary Microarchitecture Design	175
1.5. Pipeline	175

1.6. Resource Allocation	183
1.7. Pipeline Exceptions	190
1.8. Preliminary Circuit Summary	198
2. Preliminary Interconnect	201
3. Preliminary Compare	204
4. References	204
Chapter 7– Microarchitecture Design– SOAR Case Study	206
1. Microarchitecture Synthesis	207
1.1. Functional Blocks	209
1.2. Bus Structures	219
1.3. Resource Usage	225
1.4. Complete Functional Block Input and Output Specification	226
1.5. Microarchitecture Verification	228
1.6. Microarchitecture Synthesis Summary	234
2. Microarchitecture Analysis	234
2.1. Speed Analysis	235
2.2. Area and Power Analysis	240
3. References	241
Chapter 8– Functional Block Design– SOAR Case Study	242
1. Functional Block Synthesis	244
1.1. Merging	244
1.2. Splitting	249
1.3. Summary	258
2. Functional Block Analysis	265
2.1. Speed Analysis	265

2.1.1. Method	266
2.1.2. SOAR Speed Analysis	273
2.2. Power Analysis	282
Chapter 9– Circuit and Interconnect Design– SOAR Case Study	284
1. Circuit and Interconnect Synthesis	285
2. Circuit and Interconnect Analysis	286
2.1. ALU	288
2.2. Inserter/Extractor	297
2.3. SWP Comparator	297
2.4. Register File Decoders	298
2.5. Sign Extender	300
2.6. Summary	301
3. Functional Block Analysis	302
4. Microarchitecture vs. Functional Block Analysis	307
4.1. Phase 1 Analysis	307
4.2. Phase 2 Analysis	309
4.3. Phase 3 Analysis	311
4.4. TRAP Analysis	312
4.5. Decode for Phase 3 Analysis	315
4.6. Microarchitecture Analysis	317
5. Optimized Pipeline Analysis	323
6. Split Datapath Analysis	326
7. References	328
Chapter 10– Results– SOAR Case Study	329
1. Methodology	330

<b>2. Processor Results</b>	<b>333</b>
<b>2.1. Test Setup</b>	<b>336</b>
<b>2.2. Functionality</b>	<b>339</b>
<b>2.3. Speed</b>	<b>341</b>
<b>2.3.1. Phase 1</b>	<b>342</b>
<b>2.3.2. Phase 2</b>	<b>343</b>
<b>2.3.3. Phase 3</b>	<b>344</b>
<b>2.4. Process Effects</b>	<b>346</b>
<b>2.5. Summary</b>	<b>351</b>
<b>3. Architecture Results</b>	<b>352</b>
<b>3.1. Overview</b>	<b>353</b>
<b>3.2. Area and Geometry</b>	<b>354</b>
<b>3.3. Complexity</b>	<b>356</b>
<b>3.4. Speed</b>	<b>358</b>
<b>3.5. Summary</b>	<b>359</b>
<b>4. References</b>	<b>361</b>
<b>Appendix A- 4 Micron NMOS Design Rules</b>	<b>362</b>
<b>Appendix B- SOAR SLANG Description</b>	<b>368</b>
<b>Appendix C- Circuit Block Logic Diagrams</b>	<b>471</b>
<b>Appendix D- SOAR Input/Output Timing Specifications</b>	<b>589</b>

## Acknowledgements

First and foremost I wish to thank Professor David A. Hodges for joining this project and promptly providing whatever support was necessary. Due to his continuous interest the SOAR processor became a reality. I would also like to thank Professor Carlo Sequin for reading this dissertation and his useful suggestions. Thanks also go to Professor David A. Patterson for his involvement with the SOAR project. And of course thanks also go to my third reader, Professor Alan Portis of the physics department.

I would also like to acknowledge all the people who worked on the CAD tools that were used in this project. In particular Bob Mayo, Walter Scott, and Professor John Ousterhout for the MAGIC layout editor and their prompt attention to any problems with it. Also Jim Larus and Grace Mah who made automatic layout a reality and saved much time with their PLA generation tools.

Thanks also go to the MOSIS crew for their special attention to this oversized die, prompt replies to all our questions, and for providing working chips to us. I also wish to thank the processing and packaging people at Xerox for fabricating SOAR for us. And special thanks to Kodak for providing my support while I was at Berkeley.

And thanks go to all the people on the SOAR project that made it a fun project to work on. To name just a few - Will Brown, Frank Dunlap, Shing Kong, Chris Marino, and Dain Samples. And to the people who made Berkeley a fun place to be - Mike Arnold, Ricki Blau, Gordon Hamachi, Paul Hansen, Bob Mayo, Rick McGeer, Dave Wallace, and many others.

Thanks also go to the many people who without realizing it, contributed in a variety of ways. To the people at the boathouse and other rowers, among the many Dede and Brian Birch, and Ron and Velma Owen. To Will Brown, Keith Iosso, Frank Dunlap, and Tricia Fordham. And to Peter Eichenberger, Dan

Jablonski, Jim Moody, and Lynne Pollenz. And special thanks to my sisters and brother – Patti, Ann, and Dave – my cousin – Dix Brown – and my parents – Alta and Alvin Pendleton – for the wild and crazy times, proper (not too serious) perspective on the world, some key suggestions, and many interesting discussions about most anything.

## Preface

The purpose of this thesis is to present a methodology for VLSI processor design. However, a methodology is only useful if it proves to be well suited for actual problems and leads to valuable decisions when applied to these problems. The Smalltalk On A RISC - SOAR - project at Berkeley was both a guinea pig and motivating factor behind this methodology. The intent of this methodology is to provide guidelines and insights into VLSI processor design for future use, based on experience from microarchitecture design through to layout. Although the test vehicle for this methodology was an NMOS RISC processor, it is hoped that the methodology is general enough to be used with other technologies and architectures also.

The SOAR project developed from the Berkeley Smalltalk system. Due to its basic nature, Smalltalk proves to be relatively slow on general purpose computers [Unga85]. Therefore, the idea of building a system with special purpose hardware for Smalltalk was proposed. This led to architectural studies of Smalltalk during the fall of 1982 and winter of 1983 [Patt83]. Among other things, the architecture of a special purpose Smalltalk processor was specified by these studies.

The microarchitecture design of the SOAR processor had its origins in previously designed RISC processors [Unga85]. However, it distinguished itself by containing many added features (Figure P.1). Pete Foley provided a solid microarchitectural design for the SOAR datapath. The author of this thesis completed the microarchitectural design. This included among other things a more sophisticated control section than those of previous RISCs, to handle the new features. Key items of this control section were the trapping mechanism and its variety of traps, and a way to handle multicycle instructions within the framework of the RISC pipeline.



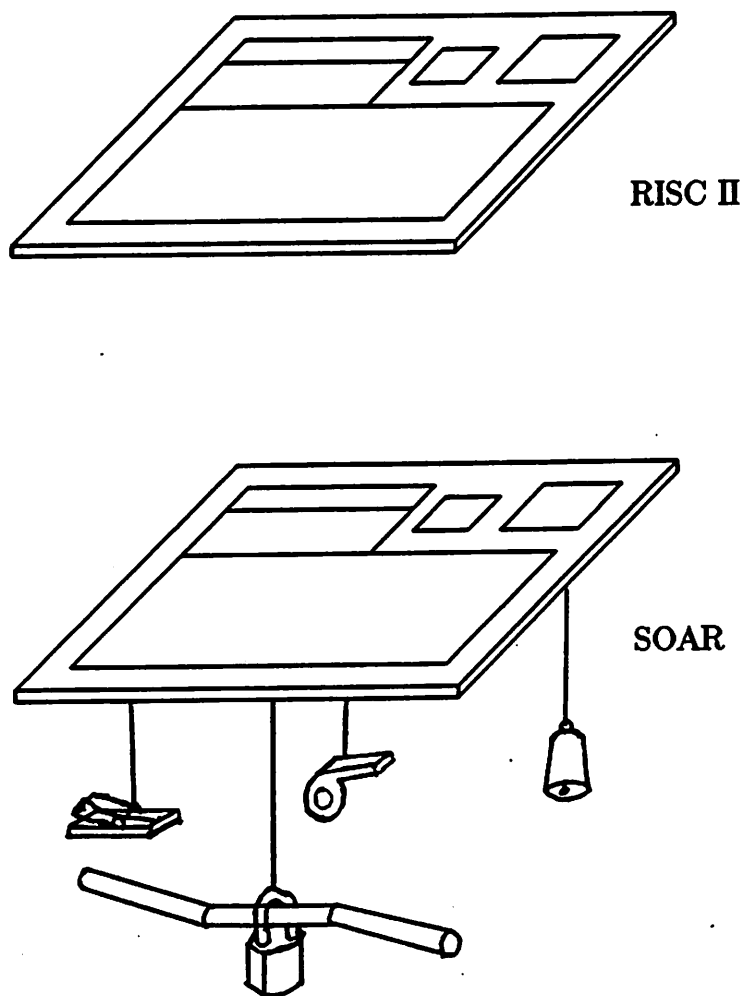


Figure P.1- RISC II and SOAR

Circuit design and layout of an NMOS version of SOAR began with a group of four in the CS292X class – spring 1983. The role of the author in this class was to interface between the microarchitecture design and circuit design, contributing to both. A first cut at the datapath circuit design and layout was completed during this class. The author then continued microarchitecture, circuit, and layout design, along with Shing Kong. In this process the PLA tool – SPLAT – was debugged, diagnostics were written, optimizations for speed and power considerations were introduced, and a first version of an NMOS layout was completed, extracted, and logically verified.

Timing verification led to the discovery of unacceptable, slow critical paths due to the multicycle instructions, in the first version of SOAR. A second version of the microarchitecture was then designed to eliminate this problem. Before complete implementation, analysis according to the author's methodology verified that it would not have the same problem or other unexpected slow paths. Design and layout of this version was completed by September 1984. Logic verification ensued, followed by timing analysis of the extracted layout. This design was fabricated in the winter and spring of 1985. Processor characterization and system development was done in the summer of 1985.

The development of a special purpose Smalltalk system is an ambitious project. It spans the disciplines of both electrical engineering and computer science. The success of a project like this relies on the efforts of individuals with a variety of talents (Figure P.2). Architectural studies culminating with CS292R, provided an efficient Smalltalk architecture [Unga85]. An architectural simulator was written in the early stages of the project [Samp85]. A CMOS version of SOAR was also implemented [Mari85]. Board design occurred concurrently with processor design [Blom83], [Dunl84], [Brow85]. System software in the form of an assembler and compiler was written [Bush85]. CAD tools proved invaluable in a project of this magnitude [Scot85]. And of course no project of this size could succeed without the resources and support of the faculty.

<u>Berkeley Smalltalk</u>	<u>NMOS SOAR</u>	<u>CAD Support</u>
Peter Lee Dave Ungar	Artie Chang Mike Klein Shing Kong Joan Pendleton Mike Remillard	Gordon Hamachi Ken Keller Jim Larus Grace Mah Bob Mayo Peter Moore John Ousterhout Joan Pendleton Dierdre Ryan Dain Samples Walter Scott George Taylor
<u>Architectural Studies</u>	<u>CMOS SOAR</u>	
John Blakken Ricki Blau Wayne Citrin Bruce D'Ambrosio Pete Foley Carl Ponder Richard Probst Harry Rubin Stuart Sechrest Dave Ungar Dave Wallace	B. K. Bose Mark Hofmann Grace Mah Chris Marino Peter Moore Dave Wallace John Zapisek	<u>Faculty</u>
	<u>Board Design</u>	Paul Hilfinger Dave Hodges Richard Newton John Ousterhout Dave Patterson
<u>Software</u>	Rich Blomseth Will Brown Frank Dunlap Joan Pendleton	
Ricki Blau Bill Bush Pete Foley Paul Hilfinger Dain Samples Dave Ungar		

Figure P.2- SOAR Design Groups

## 1. References

[Blom83] Blomseth, R.; Davis, H.; 'The Orion Project- A Home for SOAR', (Unpublished) Proceedings of CS292R- Smalltalk on a RISC Architectural Investigations, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., April 1983.

[Brow85] Brown, E. W.; 'A Virtual Memory CPU Board with a Large Cache', M.S. Thesis, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., Jan. 1985.

[Bush85] Bush, W.; 'Smalltalk 80 to SOAR Code', (draft) Computer Science Division, EECS Dept., University of California, Berkeley, Ca., 1985.

[Dunl84] Dunlap, F.; 'How To Make It Work', M.S. Thesis, EECS Dept., University of California, Berkeley, Ca., 1984.

[Mari85] Marino, C.; 'CMOS SOAR', M.S. Thesis, EECS Dept., University of California, Berkeley, Ca., 1985.

[Patt83] Patterson, D. (editor); 'Proceedings of CS292R- Smalltalk on a RISC Architectural Investigations' (Unpublished), Computer Science Division, EECS

Dept., University of California, Berkeley, Ca., April 1983.

[Samp85] Samples, A. D.; 'Software for SOARing on a SUN' (draft), Computer Science Division, EECS Dept., University of California, Berkeley, Ca., February, 1985.

[Scot85] Scott, W.; Hamachi, G.; Ousterhout, J.; Mayo, R.; '1985 VLSI Tools: More Works by the Original Artists', T.R. UCB/CSD/85/225, University of California, Berkeley, Ca., February 1985.

[Unga85] Ungar, D.; 'The Design and Evaluation of a High Performance Smalltalk System', Ph.D. Thesis, University of California, Berkeley, Ca., August 1985.

## **Chapter 1**

### **Introduction**

A method is an orderly way to arrive at a solution to a problem that has been posed. From Webster's dictionary:

**method:** An orderly procedure or process; regular way or manner of doing anything; hence a set form or procedure as in investigation or instruction.

Problems can range from very simple to extremely complex. For a simple problem, the solution may be readily apparent and no methodology is needed. As problems become more complicated solutions are not obvious and a procedure of some type is needed to arrive at a solution. Procedures can range from haphazard to highly organized. The disadvantages of haphazard procedures are many. A solution may never be reached. If one is reached, it may take a long time and not be very optimal. Organized procedures or methodologies ensure that a solution is reached, or reveal why it can not be reached if the problem is impossible. A good methodology provides a direct route to the solution and addresses all aspects of the problem. In this way a solution may be arrived at more quickly and its quality or optimality is higher than with a haphazard procedure.

#### **1. VLSI Issues**

In the past decade integrated circuit technology has evolved so that it is now feasible to put on the order of one million transistors on a chip. As the number of available transistors has increased, the amount of circuitry has likewise increased. Complex systems that in the past were composed of many individual chips, are now being built on a single or a small number of chips. Thus, a

complex system may now be spread over only one or a few chips. This leads to greatly increased complexity within a single chip. The problem addressed by an individual chip has become much more complicated. Managing this complexity is a key issue that any VLSI methodology must face.

A second issue in VLSI design involves the time needed to realize a complicated chip such as a processor. The time necessary to take a VLSI design from concept to reality increases prohibitively with the increase in circuitry if new design methods are not developed. A good methodology identifies the most time consuming parts of design and tries to find faster ways of doing this work.

Once a solution to a problem is proposed, it must somehow be checked to verify its correctness. Increased complexity means many more opportunities for malfunction. First, all functions of the VLSI processor must be identified. This alone can be a major task. Once identified, provisions must be made for verifying the solutions at various stages in the design. Unlike board design, one can not cut a trace and rewire it if a bug exists in the prototype. Instead new masks must be generated and the processor is then refabricated. This can take a few months. Once the processor has been fabricated, problems that were unforeseen during the design phase can arise. Locating these errors in a complex processor can also be difficult and time consuming. Thus, debugging time is also an issue.

A fourth issue that must be addressed is that of optimality. Complex problems, such as VLSI design, usually have more than one solution. Different solutions address the many aspects of the problem with varying amounts of success. No solution is necessarily perfect. Furthermore, many factors influence the quality of the resulting characteristics. The way in which individual factors influence the final results should be understood so that tradeoffs may be made during the design process. These tradeoffs are made with the priorities of the final processor's characteristics in mind. In this way a solution that best answers all

aspects of the problem may be chosen. This may be considered an optimal (non-theoretical) solution.

Thus, four key issues in VLSI processor design are:

1. Complexity
2. Time investment
3. Correctness
4. Optimality

This does not mean that there are no other issues. It simply means that these are four important issues. A good VLSI design methodology addresses all four of these issues.

## **2. Thesis Organization**

Many VLSI design methodologies exist. They all address the four key issues with varying degrees of emphasis. It is hard to categorize them because no hard and fast lines seem to exist. New methodologies evolve from the old, resulting in a spectrum of design styles. Chapter 2 discusses examples of present VLSI design methodologies. Examples were chosen on the basis of their popularity and to exemplify various possible design styles.

All VLSI design methodologies are structured around various design levels. Design is done at all levels in the course of realizing a VLSI processor. Many motivations exist behind the choices of design levels. Chapter 2 introduces some present design level schemes as today's methodologies are described. In Chapter 3 the design levels of the methodology proposed in this thesis are discussed. Interrelationships between the levels are described and reveal another source of the complexity in VLSI design. Implications for the methodology based on an understanding of level interrelationships, are identified.

Chapter 4 discusses the methodology in great detail. The methodology is based on the design levels of Chapter 3 and their interrelationships. Although the nature of the problem is complex, the methodology breaks the large problem into a series of smaller, more straightforward problems. The ordering of these smaller problems is suggested by their natural order in the design process and with the goal of efficient use of design time.

The value of any methodology lies in its suitability to practical problems. In light of this, Chapters 5 through 10 take the Smalltalk On A Risc - SOAR - processor through the methodology. Portions of this methodology were used during SOAR design and lead to significant improvements that were subsequently put into the processor. Completion and further application of the methodology to SOAR revealed additional improvements after submittal for fabrication. The complete methodology also reinforced many design decisions on the realized chip.



## **Chapter 2**

### **Present Methodologies**

Presently, many differing design styles exist. These styles have developed from previous styles with the designers' backgrounds having a significant impact. All VLSI design styles address the four key issues outlined in Chapter 1 with varying degrees of emphasis.

#### **Key VLSI Issues**

1. Complexity
2. Time investment
3. Correctness
4. Optimality

#### **1. Design Levels**

To realize a VLSI processor a description of the high level behavior or architecture of the processor, is transformed into a collection of geometric shapes made of various materials: silicon, silicon dioxide, aluminum, etc. When viewed from the outside world, this collection of geometries and the high level behavioral description function in the same way. Although their functions are equivalent, the forms of these two systems are entirely different. Thus, equivalent systems may be represented in more than one way. These different representations distinguish the levels of a design.

As a processor goes from an architectural description and set of system specifications to a small piece of silicon, it passes through many other representations and their corresponding design levels. Design is the process of transforming a problem from one representation to another until a final representation provides a solution to the original problem. Each intermediate

transformation is a solution to one or more aspects of the problem posed by the previous representation level. The choice of levels reflects the designers' concerns and design strategy. Thus, the levels a design passes through can be determined by the methodology. The design levels may also influence a methodology. For example, more emphasis and consequently greater optimization, is usually placed on explicitly defined levels than on levels that are only implied.

Design at any given level can range from very coarse to extremely detailed. For example, if a logic level is called for, the initial outline might only specify the logic blocks. Detailed logic design might specify the processor in terms of inverters, switches, NAND, and NOR functions. VLSI processor design begins with a rough idea for a solution at a level just below the outsider's behavioral description and is completed when a detailed representation in a material, such as silicon, is reached (Figure 2.1). Many different paths, corresponding to the many different design methodologies, may be taken from the starting point to the end point. Each path may graphically represent a methodology. Some of the possible paths would be very inefficient and therefore would not be used. Other efficient paths are in widespread use.

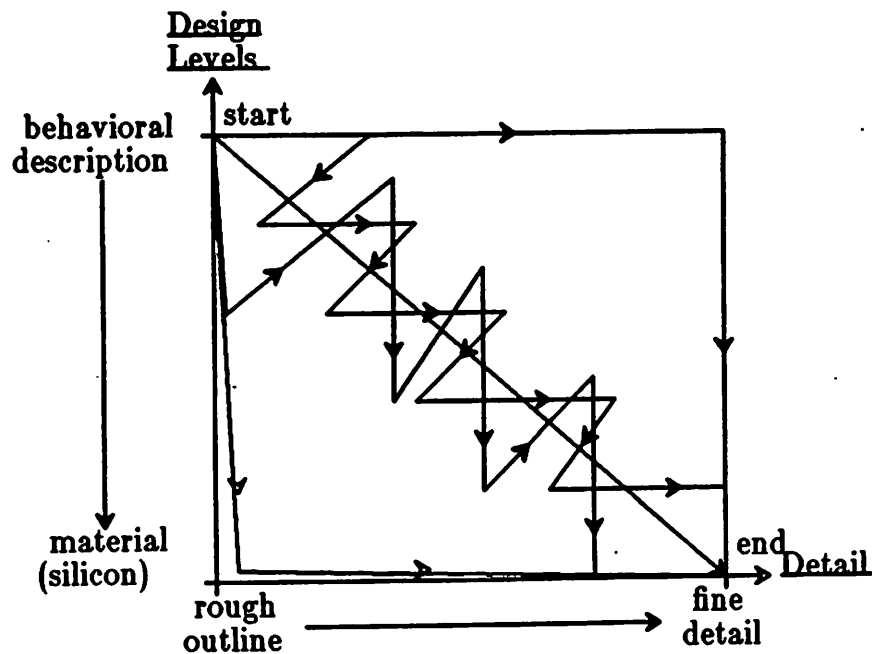


Figure 2.1- A Few of the Many Possible Design Paths

## 2. Hierarchy

Hierarchy is an important concept in all VLSI design methodologies. Hierarchical structures are stratified. Levels are formed in an attempt to isolate the activities of any given level, from the activities of other levels. Isolation is desirable because it simplifies decisions, by eliminating considerations due to outside levels, at any given level. However, total isolation is also impossible since all levels contribute to the total structure.

Hierarchies have proven to be very useful in complex designs. The concept of isolating the pieces of a problem through the use of levels allows the overall complex problem to be broken into a set of simpler problems. Each of these simpler problems can be worked on separately and is further subdivided. Subdivision continues until the lowest level of the hierarchy is reached. This is known as top down design.

Each level is composed of one or more entities. In a hierarchical structure each entity is made up of subentities. Subentities are the entities of the next lower strata of the hierarchy. For example, an ALU may be made up of 32 bit slices. All entities are specified in two ways. First, the entity is viewed as a black box with inputs and outputs. The behavior of these inputs and outputs is specified by the entity that the given entity is a subentity of. This represents the connection between levels. Then to realize the entity, the internal components – subentities – and their interconnections are specified. This internal specification must result in the required behavior as viewed from the inputs and outputs. Except for the inputs and outputs, this design is done in isolation from the other levels. Specification of the internal components of the entity leads to the input/output specifications of the subentities.

In VLSI processor design the highest hierarchical level is the entire processor chip (Figure 2.2). The first part of its behavior is the specification of the behavior at the pins. The processor may then be divided into a datapath and control section. The datapath can then be specified in terms of ALUs, registers, shifters, and buses. An ALU may then contain bitslices. Each bitslice is made up of logic gates and wires. The logic gates are composed of transistors which are formed by geometries of the various fabrication materials. The majority, but not all, of the design considerations at any one of these levels, are independent of the other levels.

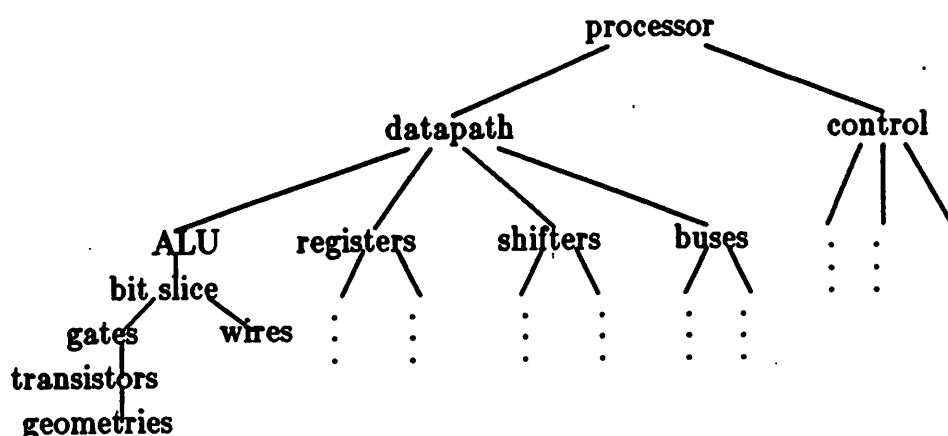


Figure 2.2- Hierarchy Example in a VLSI Processor

### 3. Design Levels and Hierarchy

Both design levels and hierarchies are useful when solving a complex problem. Therefore, the next question becomes: How do the design levels relate to the levels of a hierarchy. Hierarchies can exist within a design level. In this type of hierarchy all hierarchical levels have the same representation. The levels of the hierarchy are distinguished by the amount of detail that they contain. This is typically called a structural hierarchy [LaPo83]. The coarse outline of a processor in a given representation corresponds to the high levels of the structural hierarchy. The structural hierarchy is then traversed to the lowest levels as detail in the same representation form, is added to the processor components.

A second possibility is that the design levels may correspond to one or more levels of a different hierarchy. As this type of hierarchy is traversed the representation form of a design changes. This is considered an abstraction hierarchy. For example, if the abstraction hierarchy contains a logic level, logic diagrams may form the basis of this logic level. Another possibility is that the logic level may correspond to a sublevel within a circuit design level. Depending on the methodology, hierarchies exist within design levels and across the design

levels.

Top down design leads to both types of hierarchical structures. Each level explicitly specifies the characteristics of the next lower level. Using these characteristics the next lower level is designed. A drawback of this is that there is no obvious way to consider influences of lower levels when designing at higher levels, or influences from other branches of the hierarchy. Tradeoffs may be overlooked and optimality can suffer. For this reason most VLSI design methodologies are not purely hierarchical. Most recognize the need for feedback from the lower levels to the higher levels. Different methodologies place varying amounts of emphasis on these non-hierarchical paths and implement them differently, as will be seen in the next sections.

#### **4. Mead-Conway Style**

The Mead-Conway design style was one of the first formalized design styles [Mead80], [Trim81], [Joha81]. It is still in widespread use today as an instructional tool. It also forms the basis of more sophisticated design methodologies. The Mead-Conway style emphasizes a top down hierarchical approach. Designs are composed of composition and leaf cells (Figure 2.3) Leaf cells make up the lowest level of the structural hierarchy. They contain circuit components and wires, but no instances of other cells. Composition cells occupy all but the lowest hierarchical level. They are composed of lower level cells and the interconnects between these lower cells.

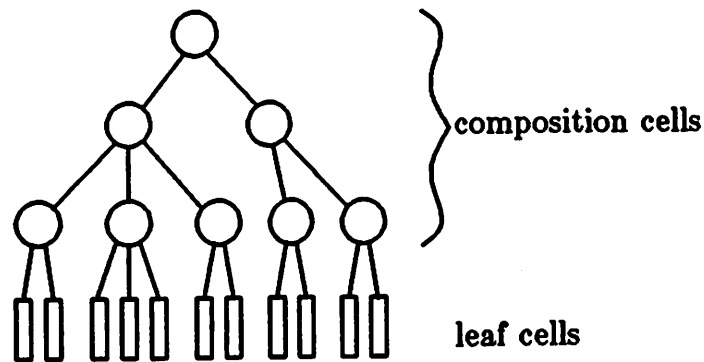


Figure 2.3- Mead-Conway Structural Hierarchy

Figure 2.4 shows the design levels of this style. The formalized methodology concentrates on proceeding through these levels in a top down manner as indicated by the solid arrows. Design begins with a behavioral description. At the architectural level, the composition cells at the top of the structural hierarchy, that are needed to implement this behavior, are identified and assigned to areas of the floorplan. Wiring is considered and the critical paths of the chip are estimated. Subsequent levels fill in the details of all cells according to the requirements passed down from the higher levels. At the cell estimation level inputs and outputs of the blocks are specified and their areas estimated. To do this the large composition cells of the floorplan are decomposed into intermediate composition cells and leaf cells. The cell detailing level calls for layout of the leaf cells. During chip integration, cells are connected together starting at the bottom of the structural hierarchy and moving towards the top until all interconnections have been made. The design is then ready for fabrication. Thus, this design style emphasizes increased refinement of the design as the levels are traversed to the cell detailing level.

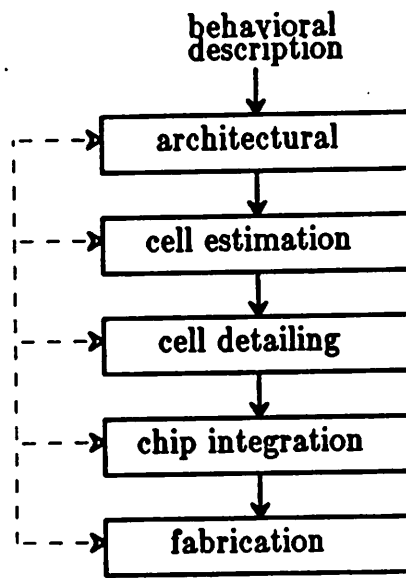


Figure 2.4- Mead-Conway Design Levels

The Mead-Conway style recognizes the need for feedback of information from the lower levels to the higher design levels as indicated by the dashed lines of Figure 2.4. It calls for iteration between the levels if pieces of the design are not compatible. However, it has not developed formalized guidelines for this process as it has done for the top down procedure.

## 5. CMU-DA System

Research work at Carnegie Mellon University has developed a system that automates processor design from the behavioral description to the layout- the CMU-DA System [Hafe78], [Snow78], [Park78], [Dire81], [Thom83], [Kowa83], [Hitc83], [Tsen83], [Walk83]. It differs from other systems by automating the highest levels of design- the microarchitecture. Traditionally, microarchitecture design has been left to humans. A methodology is needed as the basis of any design automation system, if processor design is to be fully automated. The CMU design methodology, as reflected in the design automation system, is reviewed here.



Figure 2.5 is a block diagram of the design automation system. Each block accepts input in an appropriate form for that block, and generates output of a form that can be used by the next piece of the system. Except for the optimizer, the system blocks transform the design from one representation to another. They therefore correspond to the design levels of the methodology.

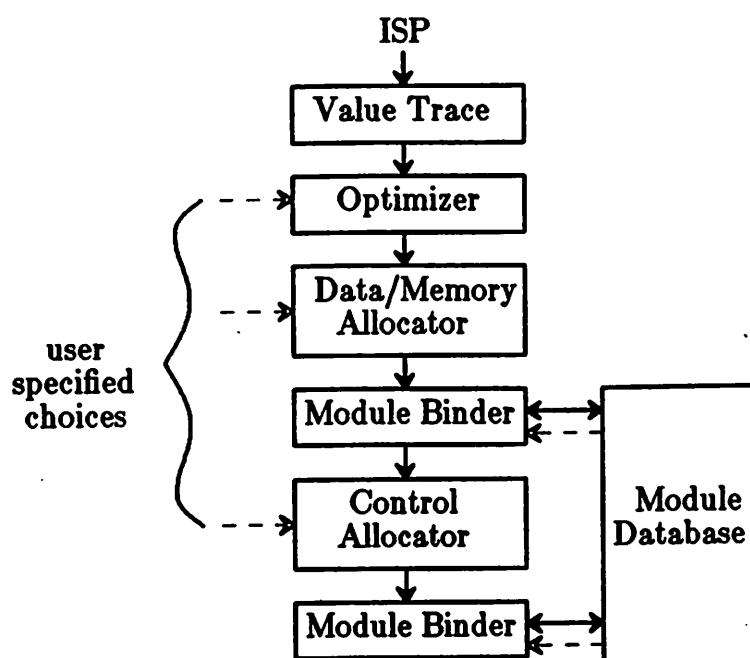


Figure 2.5- CMU-DA System

The architecture and system specifications of the processor are originally specified in the ISP hardware description language. Due to the structure of this language, the ISP description of the processor will imply certain structures in the microarchitecture. For example, temporary registers can be specified in an ISP description. This implies the same temporaries in the microarchitecture, although there might be microarchitectures that do not need the specified temporaries. The first block in the CMU-DA system converts the ISP description to a data flow representation— the value trace. This translation removes artifacts of the ISP

language such as temporaries, data dependencies, and implied control sequences. The value trace is a more general representation than the ISP description. The value trace may then be optimized to improve the speed or cost of the processor and eliminate inefficiency. The optimizer output is an optimized value trace.

The allocators generate register transfer structures. The datapath structure is generated by the data/memory allocator. The control allocator generates a register transfer representation of the control section. These register transfer structures specify the physical blocks – modules – and their interconnections.

The module database contains a library of circuits, their layouts and characteristics such as speed, size, power, and cost. The module binders select circuits from this database according to the requirements passed down by the allocators.

The blocks of the CMU-DA system may be grouped into design levels according to their output representations (Table 2.1). The ISP description and module database provide input data to the design system.

Design Level	CMU-DA Block	Output
1	Value Trace Optimizer	Value Trace
2	Data/Memory Allocator Control Allocator	Register Transfer Structure
3	Module Binder	Layout
External Inputs	ISP Module Database	

Table 2.1- CMU-DA Design Levels

This methodology provides for analysis and optimization within each level. Inputs for optimization come from requirements of higher levels or from human intervention. This does not rule out feedback from lower levels. However, there are no formal guidelines for it. Feedback from any of the lower levels can be carried out through human interaction with the system.

At the highest level, the value trace may be optimized for speed or cost improvements and to avoid inefficiency. The optimizer works with the existing value trace and optimizes according to the users specifications. The user may look at the results and try another optimizing criteria if the results are not satisfactory.

At the allocator level the user may choose the style for the datapath and control sections. Control may be microcode or PLA based. Styles for the datapath are distributed (highly parallel), bus oriented, and pipelined. Again

tradeoffs are made by trying the options and choosing the best.

Higher levels pass a register transfer description and desired module characteristics to the module binder. Optimization occurs at the module binder level as circuits are chosen from the module database on the basis of their required speed, size, power, and cost as well as their function.

In Figure 2.5 the dashed lines indicate inputs for optimizations. These inputs are all external – either user generated or from the module database. As just described, the user can specify optimizing criteria to the optimizer and design styles to the allocators. The module database provides information on the speed, size, power, and cost of each module so that the module binder can choose the best module. Optimization inputs for a given level do not formally come from any lower design levels. Optimization occurs within a given level in response to requirements from higher levels, external inputs, and choices by the user.

## **6. Bell Laboratories– Bellmac–32 Techniques**

The design methods used by Bell Laboratories in their Bellmac–32 processor project were distinguished in two significant ways [Murp81], [Kang82], [Murp83]:

1. Simultaneous design at all levels
2. Explicit feedback from lower design levels to higher levels during design

This project also explicitly defined the structural hierarchy of the processor (Figure 2.6). The highest level of this hierarchy was the chip level, with approximately 100,000 transistors. The chip was divided into macroblocks of about 10,000 transistors each. The macroblocks were then subdivided into 1,000 transistor superblocks. At the lowest level of the structural hierarchy were the blocks with about 100 transistors each.

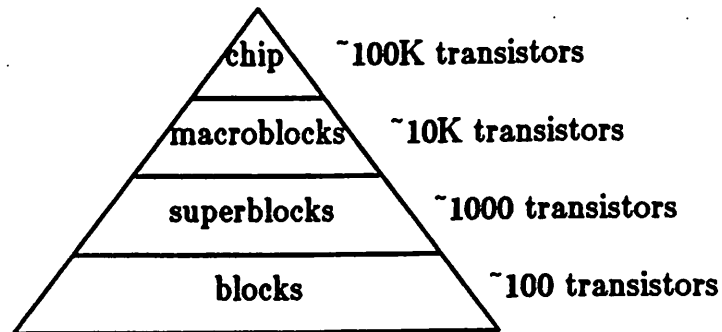


Figure 2.6- Bellmac-32 Structural Hierarchy

Design levels of this project were the system, architecture, logic, and layout levels. Unlike the Mead-Conway and CMU-DA design styles that emphasized topdown approaches, all design levels were worked on simultaneously in the Bellmac-32 project. The result of this was that information from the lower design levels was available early in the design process. Decisions at the higher levels were made with information available from the lower design levels.

Figure 2.7 illustrates the design process that included timing considerations. Design at the architecture level defined the superblocks, their function, and input/output specifications. It also generated a floorplan and the netlist for suberblock interconnections. The logic level continued with the design of the superblocks. Meanwhile the inter-superblock routing was laid out on the floorplan. Inter-superblock routing capacitances were calculated from this layout and used in a first cut at the timing analysis of the processor. This provided feedback to the system designers. Concurrently, logic, layout, and timing analysis of the blocks was also carried out.

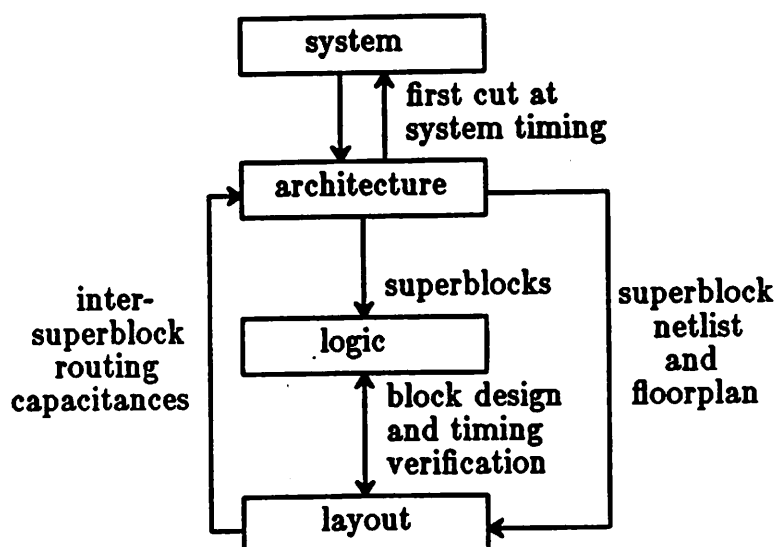


Figure 2.7- Bellmac-32 Design Process with Timing Considerations

As can be seen in Figure 2.7, the arrows of the design process not only point from the higher levels to the lower levels. They also point from the lower levels to the higher. The Bellmac-32 design process sets explicit guidelines for feedback of lower level information into design decisions at the higher levels. This feedback from the lower design levels to the higher ones, can also be done for other characteristics such as area and power.

## 7. IBM- Philo VLSI Design System

IBM's Philo design system is used in their master image designs [Donz82], [Ahdo83]. Master image designs emphasize both regularity and flexibility. Chip size for these designs is fixed. The chip is organized into a regular array of cell locations. Power buses and other routing are also fixed. Circuits are composed of cells and macros. The cells occupy one or two adjacent cell locations. A macro is any circuit that requires more than 2 cell locations. PLAs and RAMs are two types of macros. Layout complexity is reduced by this regularity.

The Philo design system provides for logic design and layout. System behavior and architecture are outside the realm of this tool set. Layout is accomplished in two steps – placement and routing. Figure 2.8 shows the design levels and their interactions. Design proceeds in a top down manner with timing simulation at each level. This timing simulation uses information from the lowest level – the wiring level.

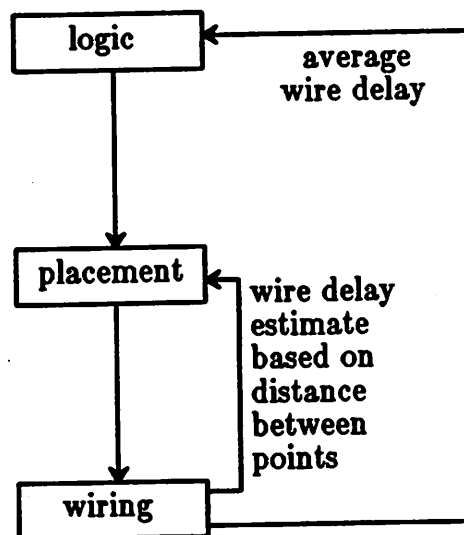


Figure 2.8- Philo Design Process with Timing Considerations

Design starts at the logic level. Accurate estimates of the wire lengths and loads are not available so an average delay is assumed. This average delay is based on characterization of the master image process. Using this rough estimate gross timing errors can be avoided.

After the logic has been verified, the cells and macros are placed onto the grid. After placement, the lengths of the wires may be estimated from the distances between their terminations. A more accurate estimate of routing delays than the logic level estimate, is now available based on this wire length estimate. Timing simulation at the placement level uses this refined wire delay estimate.

Design then moves to the wiring level. All interconnects are now routed and the actual length of each wire becomes known. Wire delays can now be accurately estimated from the actual wire lengths. Timing simulation is done once again with even greater accuracy now. Thus, the Philo design system has provided a formal method for feedback of delay information, from the lowest design level - wiring - to the highest level of the design system - the logic level.

## **8. Summary**

These are four design styles that portray many of the concepts and approaches that are characteristic of VLSI design methodologies. The four key issues of VLSI design are addressed by each of them. All of them rely heavily on computer aided design - CAD - tools to address these issues. Many CAD tools exist to aid in the synthesis, verification, and analysis of VLSI designs. Synthesis tools include such things as layout editors, placement tools, and routers to name a few. Logic verifiers, timing verifiers, and design rule checkers are just a few of the verification tools that are available. Examples of analysis tools are timing simulators and tools for power estimation. CAD tools can do much of the tedious, time consuming work of processor design. Thus, the issue of the time investment needed to realize a VLSI processor is directly answered by the development of CAD tools.

The issue of correctness can be answered in more than one way. Correctness must be verified at all levels and between the levels. One approach is to develop CAD tools that can check the correctness of these complex chips. For example, design rule checkers check the layout level for design rule violations. The layout may be checked against the logic by extracting the layout and then using a transistor level logic simulator on the extracted layout. A second approach is to develop tools that synthesize a lower level from a higher level description. Rules



for correctness are built into the synthesizer. In this way the lower level representation is correct by definition (assuming a correct higher level description) if it can be synthesized.

These four design styles all depend on hierarchies. As previously discussed, hierarchy is very important in dealing with the complexity issue of VLSI design. It provides an organized way of viewing the complex problem. The efficiency of many CAD tools is also based on hierarchical organization.

Perhaps the least understood of the key issues is optimality. The four examples of design styles emphasize optimality to varying degrees. They ranged from mentioning iteration as a way to optimize, to providing analysis tools that could be used to calculate and improve optimality at specified places in the design process. Analysis tools help evaluate the optimality of a design but an understanding of relationships between the many aspects of a design is needed to be able to optimize efficiently. The optimizer of the CMU-DA system is an example of a CAD tool that has incorporated some understanding of these relationships. Another approach to optimality is to cut design time, through the use of CAD tools. With a shorter design cycle multiple alternatives can be tried and evaluated. The best one is then chosen and fabricated.

The experience of participating in all levels of the SOAR processor design provided valuable insight into the many interrelationships between design levels. This thesis outlines the interrelationships between the various design levels. These interrelationships are numerous and have many causes. This is another source of the complexity in VLSI design. The effects of tradeoffs may be predicted from an understanding of these interrelationships. Many of these interrelationships are influences of lower design levels on the higher levels. This thesis presents guidelines for incorporating these bottom up influences into the design process. This can lead to increased optimality in a shorter design time.

## 9. References

- [Ahdo83] Ahdoot, K.; Alvarodiaz, R.; Crawley, L.; 'IBM FSD VLSI Chip Design Methodology', Proceedings of the 20th Design Automation Conference, Miami Beach, Fla., June 1983.
- [Dire81] Director, S.; Parker, A.; Siewiorek, D.; Thomas, D.; 'A Design Methodology and Computer Aids for Digital VLSI Systems', IEEE Trans. on Circuits and Systems, V.28, N.7, July 1981.
- [Donz82] Donze, R.; Jenkins, M.; Sanders, J.; Sporzynski, G.; 'Philo- A VLSI Design System', Proceedings of the 19th Design Automation Conference, Las Vegas, June 1982.
- [Hafe78] Hafer, L.; Parker, A.; 'Register-Transfer Level Digital Design Automation: The Allocation Process', Proceedings of the 15th Design Automation Conference, 1978.
- [Hitc83] Hitchcock, C.; Thomas, D.; 'A Method of Automatic Datapath Synthesis', Proceedings of the 20th Design Automation Conference, Miami Beach, Fla., June 1983.
- [Joha81] Johannsen, D. L.; 'Silicon Compilation', PhD. Thesis, Dept. of Computer Science, T.R. 4530, California Institute of Technology, Pasadena, Ca., 1981.
- [Kang82] Kang, S.; Krambeck, R.; Law, H.; Lopez, A.; 'Gate Matrix Layout of Random Control Logic in a 32bit CMOS CPU Chip Adaptable to Evolving Logic Design', Proceedings of the 19th Design Automation Conference, Las Vegas, June 1982.
- [Kowa83] Kowalski, T.; Thomas, D.; 'The VLSI Design Automation Assistant: Prototype System', Proceedings of the 20th Design Automation Conference, Miami Beach, Fla., June 1983.
- [LaPo83] LaPotin, D.; Nassif, S.; Rajan, J.; Bushnell, M.; Nestor, J.; 'DIF: A Framework for VLSI Multi-Level Representation', T.R. CMUCAD-83-20, Carnegie Mellon University, Pittsburgh, Pa., Nov. 1983.
- [Mead80] Mead, C.; Conway, L.; 'Introduction to VLSI Systems', Addison/-Wesley Publishing Co., Reading, Ma., 1980.
- [Murp81] Murphy, B.; Thomas, L.; Molinelli, J.; Edwards, R.; 'A CMOS 32-bit Single Chip Microprocessor', International Solid State Circuits Conference Digest, V; Feb. 1981.
- [Murp83] Murphy, B. T.; 'Microcomputers: Trends, Technologies, and Design Strategies', IEEE Journal of Solid State Circuits, V.18, N.3, June 1983.
- [Park79] Parker, A.; Thomas, D.; Siewiorek, D.; 'The CMU Design Automation System: An Example of Automated Data Path Design', Proceedings of the 16th Design Automation Conference, 1979.
- [Snow78] Snow, E.; Siewiorek, D.; Thomas, D.; 'A Technology - Relative Computer Aided Design System: Abstract Representations, Transformations, and

Design Tradeoffs', Proceedings of the 15th Design Automation Conference, 1978.

[Thom83] Thomas, D.; Nestor, J.; 'Defining and Implementing a Multilevel Design Representation with Simulation Applications', IEEE Trans. on Computer Aided Design, V.2, N.3, July 1983.

[Trim81] Trimmerger, S.; Rowson, J.; Lang, C.; Gray, J.; 'A Structured Design Methodology and Associated Software Tools', IEEE Trans. on Circuits and Systems, V.28, N.7, July 1981.

[Tsen83] Tseng, C.; Siewiorek, D.; 'Facet: A Procedure for the Automated Synthesis of Digital Systems', Proceedings of the 20th Design Automation Conference, Miami Beach, Fla., June 1983.

[Walk83] Walker, R.; Thomas, D.; 'Behavioral Level Transformation in the CMU-DA System', Proceedings of the 20th Design Automation Conference, Miami Beach, Fla., June 1983.

## **Chapter 3**

### **Design Levels**

The concept of design levels and hierarchy was reviewed in Chapter 2. Two types of hierarchies were defined – the abstraction and structural hierarchies. Both types of hierarchy are composed of levels. Design levels as defined in this thesis, correspond to the levels of the abstraction hierarchy. They are distinguished by:

1. The type of problem addressed
2. Processor representation
3. The way that processor characteristics are analyzed

Levels of a structural hierarchy are defined by the amount of detail that they contain. Structural hierarchies exist within the design levels.

Levels are defined so that the initial complex design problem is broken into many smaller, simpler problems. Ideally, each smaller problem would be entirely self contained. Solutions to other problems would not influence this isolated problem. In practice this is not possible. To complete the design, solutions to all of the small problems are needed. These solutions affect each other and tradeoffs must be made. Although the smaller problems can not be entirely self contained, a judicious choice of levels can minimize the influences of other levels on a given level. This does not mean that tradeoffs are ignored. It simply means that the levels are organized so that the majority of considerations, decisions, and tradeoffs occur within a level – not across levels.

Levels of an abstraction hierarchy have different representations for the same structure. For example, at the higher levels a processor may be represented by a behavioral description written in an algorithmic language. At lower levels the same processor might be represented by logic diagrams or transistor schematics. Each representation provides a unique view of the overall problem that leads to a

solution to some aspect of the original problem.

A consequence of these different representations is that processor characteristics are analyzed differently at the different levels. For example, the speed of a processor may be characterized by the number of instructions needed to run a benchmark at the higher levels. At a lower level it might be characterized by the number of gate delays in a given circuit block.

This thesis proposes five major design levels that a VLSI processor design can be partitioned into. Each level requires certain types of information as inputs to it. Design strategies are then formulated and decisions are made based on these inputs. This results in outputs from the given level. These outputs may be used as inputs to other levels. Level inputs may also be predetermined by factors outside the scope of the processor design. Design at any level satisfies the inputs for that level and results in outputs that affect other levels. The five design levels are:

Microarchitecture

Functional Block

Circuit

Interconnect

Process

## **1. Microarchitecture Level**

The highest level of processor design is the microarchitecture level. One set of external inputs to the microarchitecture is the architecture (Figure 3.1). This includes data types, word size, addressing modes, the instruction set, register organization, operations to be performed, and internally generated exceptions and trapping conditions. The system that the processor is to be a part of also places

restrictions and requirements on the processor. Interfaces between the processor and other parts of the system determine processor behavior. System requirements include such things as timing specifications at the pins, bus protocols, coprocessor protocols, memory and I/O configurations, external interrupts, package, power budget, and test methods.

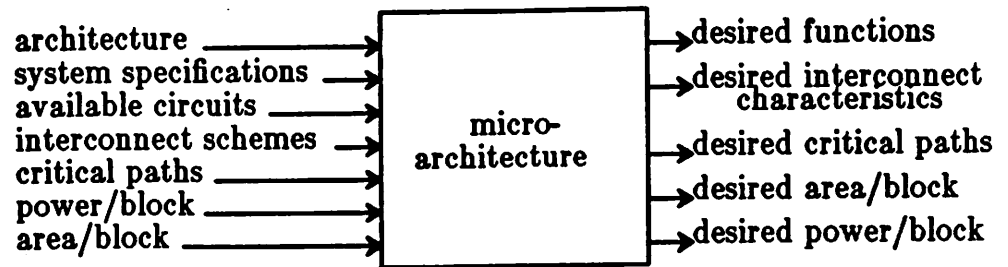


Figure 3.1- Microarchitecture Level

Another set of inputs to the microarchitecture level are the circuits that are available for use in the design. Various characteristics, such as speed, area, and power, of these circuits are important to the microarchitecture design. The available circuits and their characteristics are matched to the architecture and system requirements through the proposed microarchitecture.

Possible interconnect schemes are another input to the design at the microarchitecture level. This includes the number and configuration of data busses, distribution of control lines, and communication between various subsystems within the processor. Interconnect schemes are usually expressed in a floorplan of the processor.

The critical paths that exist on a processor are also important to the microarchitecture. Once a microarchitecture is proposed, the critical paths can be estimated from an analysis of design at the lower levels: process, circuit, interconnect, and functional block levels. The proposed microarchitecture can

then be analyzed, using the critical paths, to see if it will meet the system specification inputs. This can result in further design of the microarchitecture or a viable compromise of the system specifications. Similarly, the power and area of each functional block is also an input to the microarchitecture level. These may also be determined by analysis at the lower levels. They can then be totaled to arrive at an estimate of the processor size and power. These estimates are then compared with the system specifications to determine if further optimization is needed.

Given these inputs, design alternatives for the microarchitecture are proposed and analyzed. Design decisions concerning high level implementation issues are made. The choice of the microarchitecture should not be visible to the programmer or system designer. There are two major problems to be solved during microarchitecture design. First, the functions that the processor must perform are identified. The instruction set and system specifications determine the basic functions of the processor. Secondly, the coordination - timing - of the blocks that perform these functions, must be specified. Good timing leads to correct and efficient execution of instructions. Examples of design at the microarchitecture level are parallelism, pipeline design, the use of microcode, resources needed and their allocation, and interrupt and trapping mechanisms.

Given a proposed microarchitecture, an important question is: How good is it? Typical characteristics that are analyzed to answer this question, are speed, power, and area. As previously discussed, the yardstick for measuring any characteristic depends on the design level being analyzed. At the architecture and system level, the number of instructions needed to complete a job using a given instruction set, is a measure of the speed of the architecture. The number of clock cycles needed for each instruction type is also a measure of the system and architecture speed if system factors influence this number. For example, if a

single instruction is to do eight loads, the memory speed will influence the number of cycles needed to complete this instruction. The number of clock cycles/instruction can also be a measure of the speed of the microarchitecture. For example, in a microcoded machine each instruction is typically implemented with a series of single cycle microinstructions. The number of microinstructions needed for a given instruction determines the number of cycles required for that instruction. Cycles are typically split into phases and processor activities are assigned to each phase during microarchitecture design. Therefore, a second measure of microarchitecture speed is the number of clock phases in each cycle. The total cycle time is the sum of the individual phase lengths. This leads to a third criteria for processor speed evaluation at the microarchitecture level: phase lengths. At the microarchitecture level, the processor activities assigned to each phase determine the phase length and are therefore the third measure of speed.

Power evaluation at the architecture and system level, is done by determining the power consumption of each chip in the system. Microarchitecture design divides a chip into functional blocks. The power budget of each functional block is the criteria for power analysis at the microarchitecture level.

Similarly, the total processor area is used for the area measurement at the system level. At the microarchitecture level, the area of each functional block is important. Further design at the functional block level will divide this area into circuit and interconnect areas.

One set of outputs of the microarchitecture design are the desired functions and their characteristics. The proposed microarchitecture necessitates various functions that must be provided for by the lower levels. The characteristics of these functions are also important to the microarchitecture. For example the functions' speeds directly affect the timing.



Desired interconnect characteristics are another set of outputs from the microarchitecture level. Examples of this are interconnect speed and the number and types of levels available. Interconnect speeds partially determine signal speeds which are important to the processor timing. The number of interconnect levels affects the interconnect scheme.

The desired critical paths are also outputs of the microarchitecture design. These evolve from the system specifications. The system specifications place timing requirements on the processor. For a proposed microarchitecture, each signal path will have to operate at a speed that will allow the system specifications to be met. These speeds are the desired critical paths. When doing design at lower levels these critical paths must be taken into consideration. Similarly, the microarchitecture design may specify the sizes and power budgets of the individual blocks. This is done so that the total power and area of the processor matches the system requirements.

## **2. Functional Block Level**

Design at the functional block level involves mapping the functional blocks specified during microarchitecture design, into the circuit blocks that will be used in the processor layout. As this is done the function of each circuit block and their interconnections are determined. Examples of datapath blocks are ALUs, counters, shifters, latches, various registers, sign extenders, and comparators. Typical control blocks are latches, counters, comparators, PLAs, ROMs, and random logic blocks.

Microarchitecture design assigned processor activities to the clock phases. Speed analysis at the functional block level determines the time needed to complete each activity and identifies the major components of this delay. The delay of the output signals of an activity is a measure of the time needed for that

activity to complete. These signals – critical paths – typically propagate through circuit blocks and across interconnects. For example, an instruction decode may start with the propagation of the opcode to a decode PLA. The decode PLA then determines control line values. These values then travel across other interconnects to the places where they are used. Decode delay is the sum of these two interconnect propagation delays and the PLA delay. Thus, the circuit block and interconnect delays determine the speed of each activity.

The power consumption of each functional block is the sum of the power dissipated in its circuit blocks and interconnects. Circuit blocks consume both d.c. and a.c. power. The interconnects require a.c. power according to their capacitive load and clock rate. Thus, power dissipation at the functional block level, is the sum of the power consumed by the individual circuit blocks and interconnects.

Functional block area is also the sum of the areas of the circuit blocks and interconnects. Therefore, circuit block and interconnect areas are of concern for area analysis at the functional block level.

The first inputs to the functional block design level are the desired functions of the processor (Figure 3.2). These desired functions are specified at the design level above the functional block level – the microarchitecture level. Design at the functional block level assigns these functions to the circuit blocks.

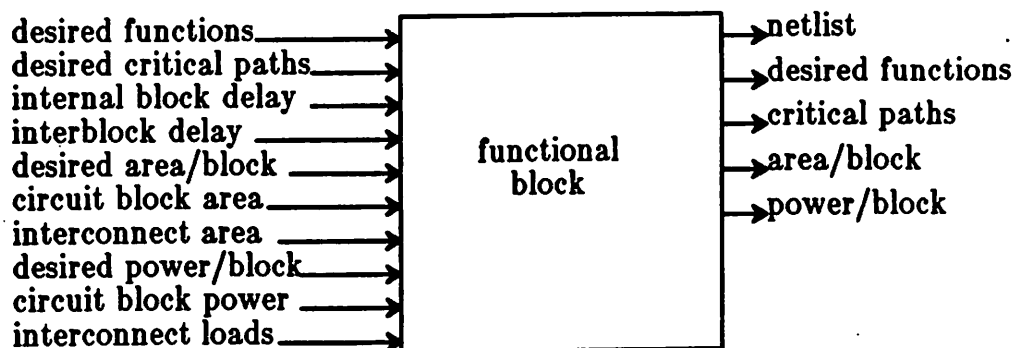


Figure 3.2- Functional Block Level

Another set of inputs to the functional block level are the desired critical paths of the processor. When doing timing analysis, these desired critical paths are compared to the predicted critical paths of the proposed functional block organization. The predicted critical paths are arrived at by considering the internal delays of each block and the delays between blocks, the interblock delays. Therefore, to do this type of analysis of the functional block design, the internal block delays and interblock delays must be inputs to the functional block design level.

Similarly, the desired areas and power budgets of the functional blocks are inputs to the functional block level. These are compared to the predicted areas and power budgets. To predict the area, the circuit block sizes and interconnect areas are needed. These are therefore inputs to the functional block level. A power estimate for the processor requires the power dissipation of the individual circuit blocks and interconnect loads to be an input to the functional block level. The circuit blocks consume both d.c. and a.c. power while the interconnects require a.c. power according to their loads and the clock rate.

One output of the functional block design is a netlist. Knowing the functions that are in each block and how they relate to each other, leads to a complete

netlist for the processor. Typically, the netlist will contain some nets that may limit processor speed. These critical nets are identified in the complete netlist.

Another set of outputs are the desired functions of the circuits in each circuit block. These desired functions and their characteristics are passed to the circuit design level for realization.

Analysis of the functional block design results in estimated critical paths for the proposed design. These critical path outputs are compared with the desired critical path inputs. Based on these comparisons, optimization and redesign of the functional block scheme is carried out. Analysis also provides estimates of the area and power consumption of the functional blocks. These estimates are compared with the desired areas and power budgets. Redesign is then done if necessary.

### **3. Circuit Level**

Circuit level design results in a layout realization of the circuit blocks. This includes the logic design of each block. The logic design is then transformed into a gate level design. From the gate level description a transistor level design is specified. The transistor level schematics are the basis for the actual layouts of the functional blocks. The logic, gate, transistor, and layout representations are sublevels of the circuit level in this methodology. It is not uncommon for these sublevels to be distinct design levels in other methodologies [Thom83].

Speed analysis at the circuit level determines the speed of each circuit block. This may be done in a series of steps according to the sublevels. First, the circuit block speed may be expressed in terms of a number of logic block delays. Each logic block delay is the sum of the gate delays within that logic block. Gate delays are determined by the speeds of the transistors that make up the gate and the loads that they must drive. Finally, layout parasitics can be included in the

gate delay analysis.

The concern of power analysis at the circuit level is the power consumption of each circuit block. This may also be analyzed according to the sublevels. Power dissipation of the logic blocks and gates can be determined. At the transistor sublevel, the transistor sizes determine power consumption.

Similarly, area analysis at the circuit level determines the area of each circuit block. Area may be evaluated at each of the sublevels also. The circuit block area can be estimated from the number of logic blocks that it contains. The size of each logic block is determined by its gate count. Gate size is determined by the number of transistors in each gate. Transistor sizes are found by consulting the layout.

Inputs to the circuit level come from all other levels (Figure 3.3). The process level specifies the available devices, layers, and their parameters. These will partially determine the types of circuits that are used and their speeds. The process also specifies the design rules for the circuit design. These are used by the layout sublevel and influence the area and geometry of the circuits.

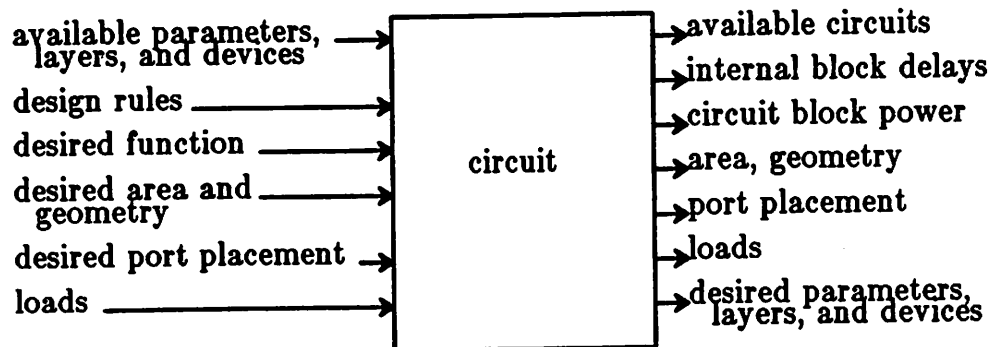


Figure 3.3- Circuit Level

Another important set of inputs to the circuit level are the desired functions of the circuits and their characteristics. Custom circuit design proceeds according to the desired circuit functions. Cell libraries can be built for popular functions. Circuit design attempts to satisfy the desired function and its characteristics for each circuit block. If this can not be done compromises are made at other levels.

Other inputs to the circuit design are the desired areas and geometries of each circuit block. The interconnect scheme leaves empty areas that the circuit blocks must fit into. The order of interconnect wires will also affect the placement of the terminals for each circuit block. Loading due to interconnects and other circuit blocks at their terminations, is another input to circuit design. Circuits must be designed to drive these loads at the speeds as specified by the desired function input.

Outputs from the circuit level are used by all other levels also. Circuit design will result in a list of circuits that are available for use in the processor and their characteristics. A knowledge of these is important to the higher design levels.

The internal delays of each circuit block is another output of circuit design. These delays are used in critical path analysis at the functional block level. The power consumption output is used to predict the total power required by the processor.

The area and geometry of circuit blocks is another output of circuit design. The circuit block areas are used by the functional block level to estimate the processor size. Interconnects must be routed around the circuit blocks and therefore use the area and geometry output of circuit design. The port placement on each circuit block is another output of circuit design used by the interconnect design.

Another set of outputs are the loads of the input ports of each circuit block. Loads must be taken into consideration when designing other circuit blocks and their interconnects.

A final output is a set of desired parameters, layers, and devices. To meet certain aspects, such as speed or power, of the desired function input, various process parameters, layers, and devices are important.

#### **4. Interconnect Level**

Communication between the circuit blocks of a processor is accomplished through the interconnects of the processor. Interconnects connect the circuit block inputs and outputs according to the scheme that was proposed by the functional block design level. Design at the interconnect level results in a layout realization of all required interconnects.

Speed analysis at the interconnect level is concerned with interconnect delays. Power analysis estimates the a.c. power required to drive the interconnect load at the given clock rate. Interconnect areas are also determined by analysis at the interconnect level.

The interconnect level also has inputs from all other levels (Figure 3.4). One important characteristic of the interconnects is the area that they occupy. The circuit blocks have certain sizes and geometries, and the interconnects must fit around them. Thus, one set of inputs to interconnect design are the sizes and geometries of the circuit blocks. The design rules are another input to the interconnect level. The dimensions specified for interconnect levels by the design rules, are a major factor in determining the area occupied by the interconnects. The available layers for the interconnects are another set of inputs to this level that will affect their area.

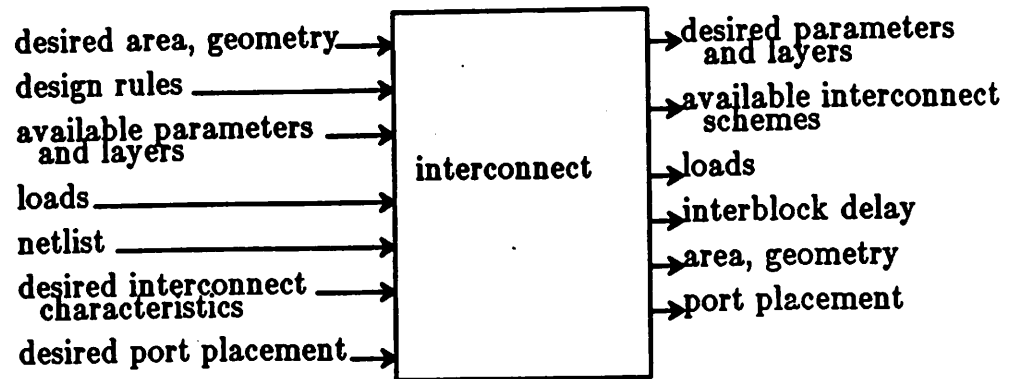


Figure 3.4- Interconnect Level

Speed is another important characteristic of the interconnections. Their speed is affected by such things as the resistance and capacitance of the interconnect layers. Therefore, the process parameters must be an input to the interconnect design. Another input that will affect the speed of the connection, is the load at the termination of the connection. This load partially limits the maximum speed of the interconnect.

The netlist for the processor is another input to the interconnect level. The netlist specifies all terminals that are to be connected together. Another set of inputs are the desired characteristics of the interconnects. Design at the higher levels may benefit if various characteristics are available. The placement of the ports to the circuit blocks is another input to the interconnect design. Ideally, the order of the ports should match the order of the interconnect lines.

Outputs of the interconnect level are used by all other levels. From a knowledge of the higher design levels, the interconnect level may generate an output of desired process parameters and layers. The interconnect design level also generates a set of available interconnect schemes for use by the higher levels. It does this from inputs that were generated by lower levels. Examples of things that affect interconnect schemes are double level metal and silicides.



Another set of outputs are the loads of the interconnects. The circuit blocks must have the capability to drive the interconnect loads at the desired speeds. Interconnect loading also contributes to the a.c. power consumption of the processor.

The interblock delay is another output. It is used to analyze the speed of a proposed functional block design.

The area and geometry are other outputs of the interconnect level. Interconnects can occupy a significant area. This must be taken into consideration when estimating the chip size. The interconnect area and geometry output is also used when doing layout of the circuit blocks in order to leave a minimum of wasted space.

Port placement is another output. A compromise between the optimum port placement for the circuit design and the interconnect design must be arrived at.

## **5. Process Level**

The fifth and lowest major level is the process level. Design at the process level consists of using the available processing techniques to produce devices, parameters and layers that are desired by the higher levels. The available processing techniques are inputs to the process level (Figure 3.5). They are determined by technology - not by any of the other design levels. Other inputs are the desired devices, parameters, and layers. These are requested by the higher design levels and the process level attempts to satisfy these requests.

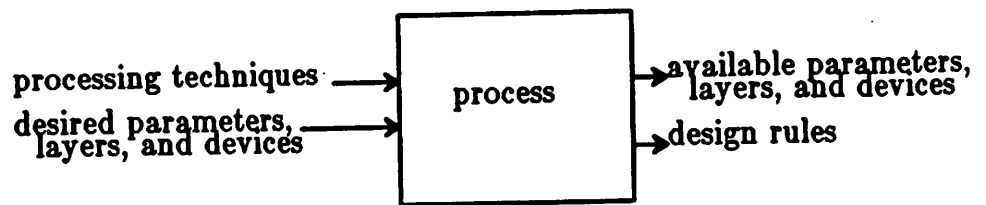


Figure 3.5- Process Level

Process design determines the available devices and their parameters. Devices of an MOS technology may include both NMOS and PMOS enhancement and depletion transistors. Examples of device parameters are the threshold voltages and transconductances of all available transistor types. Process design also specifies the available layers and their parameters. Another important output is the design rule set for the process. This specifies the minimum feature sizes for all layers.

At the process level, process parameters that affect speed are the focus of speed analysis. Examples of these for an MOS process are transconductances, oxide thickness, diffusion capacitances, and resistivities. Process parameters also determine power dissipation. Threshold voltages, transconductances, and capacitances are examples of parameters that influence power dissipation for an NMOS process. Area analysis concentrates on the design rules, at the process level.

## 6. Interrelationship Overview

Many tradeoffs have to be considered during the design of a VLSI processor. This can be seen from an examination of the inputs and outputs for each level. A few inputs – processor architecture, system specifications, and processing techniques are not usually dependent on design activities of the five levels. The definition and specification of these inputs is primarily independent of the

processor design. Compromises in these inputs, might arise due to processor design but they are basically determined outside of the scope of the processor design.

The vast majority of the inputs to the various design levels, can be outputs of other levels. This means that design at any given level depends on the design at other levels. This is the source of the many tradeoffs and complexities that exist in VLSI design. Figure 3.6 shows this interdependence of levels according to the previously described inputs and outputs for each level. Some levels – circuit and interconnect levels – have inputs from all other levels and outputs that affect all other levels. Even the level with the fewest relationships to other levels, the process level, has inputs from two of the other four levels, and outputs that affect two other levels. One definition of an ideal design, is a design that is optimized based on all possible level interrelationships and tradeoffs.

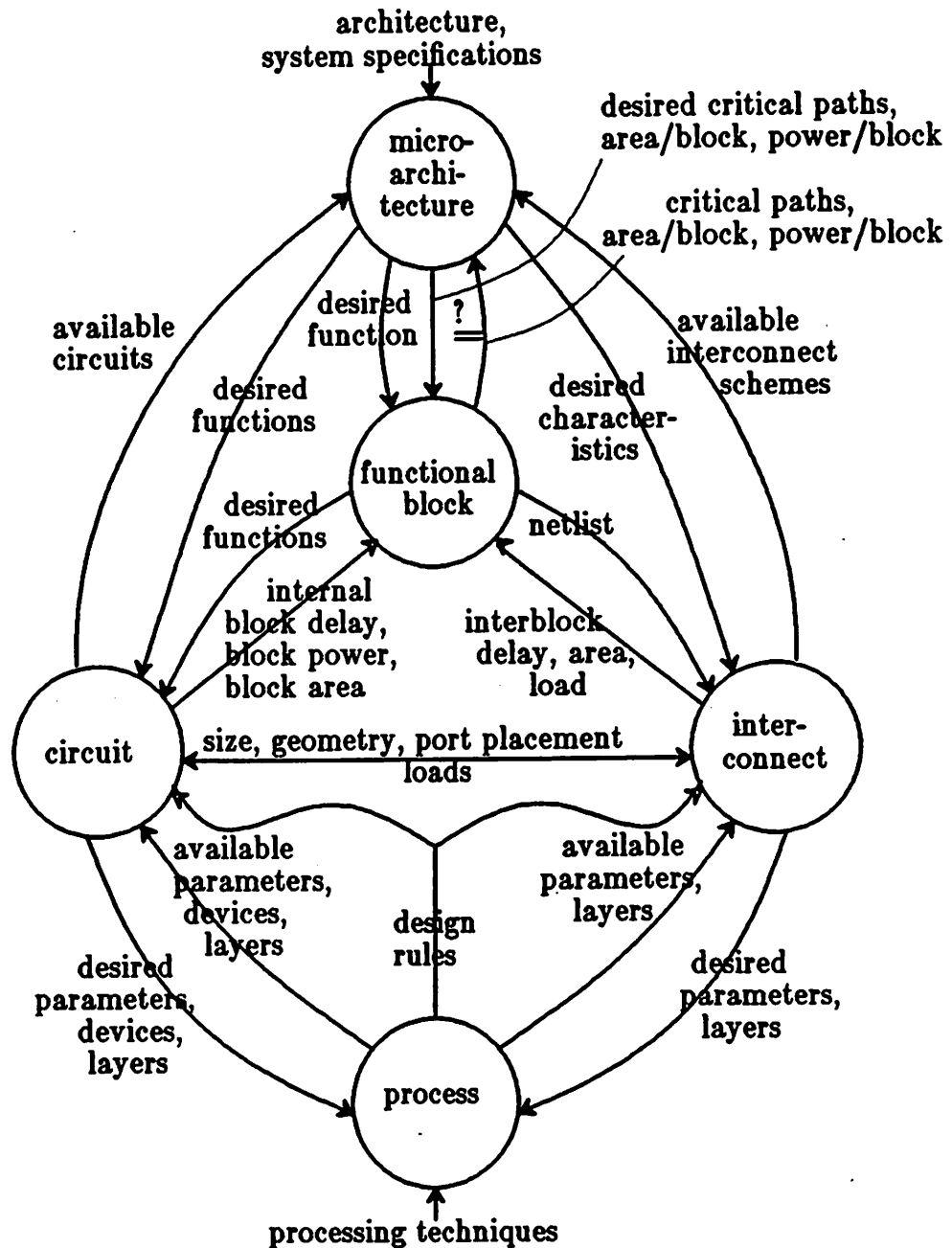


Figure 3.6- Overall Flow Diagram

### 6.1. External Inputs

Design is easiest when outputs of a design level are based only on external fixed inputs to that level. When this is the case, design is done for the given set of inputs or range of inputs. Values for these inputs which depend on the design

itself, do not need to be considered. Using an analogy to a function  $y=f(x)$ , it is much easier to determine  $y$  for one specific value or range of values for  $x$ , than for  $x$  which depend, in some complicated way, on the given inputs and possibly  $y$  itself (Figure 3.7). However, this simple dependence of outputs on external inputs, is not the situation for any of the outputs in a full custom design. The only levels where this might be the case because they do have external inputs, are the microarchitecture and process levels (Figure 3.8). However, these levels also have inputs from other design levels. All outputs from these two levels are functions of both the external inputs and the inputs from other design levels. Thus, the situation for these levels, is analogous to  $y=f(x,z)$ , where  $x$  is the set of given external inputs and  $z$  is the set of internal inputs that relate to  $x$  in some complicated way through the other design levels.

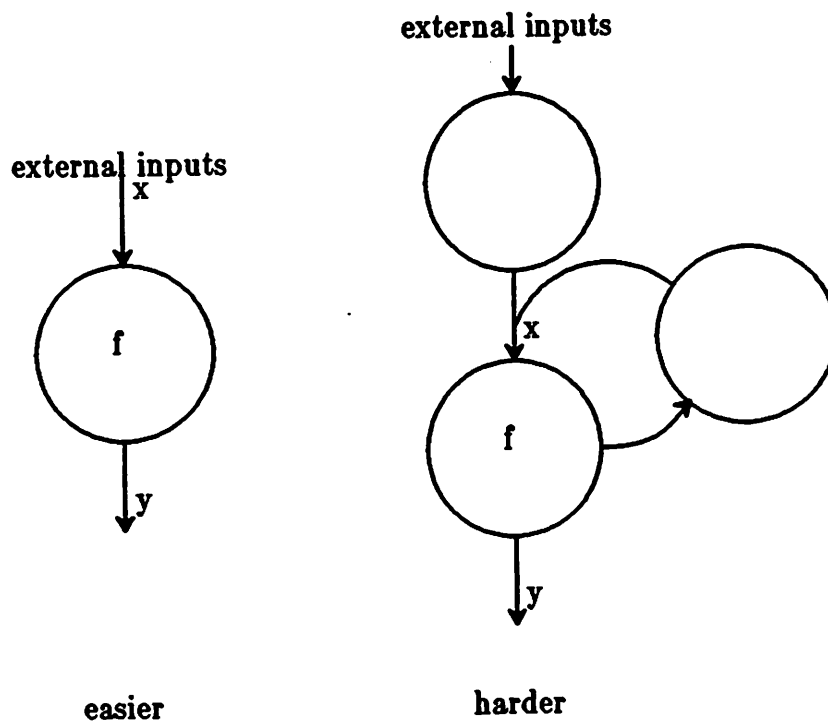


Figure 3.7-  $y=f(x)$

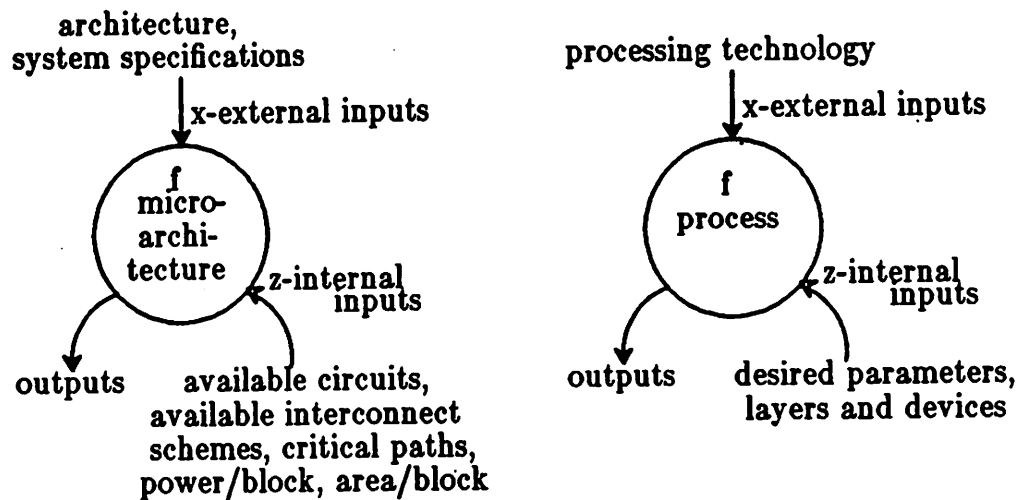


Figure 3.8-  $y=f(x,z)$

Thus, it can be seen that if more inputs to the various design levels are externally specified, instead of being dependent upon other design levels, fewer options and tradeoffs will exist and design will be easier. However, performance of the final design might suffer since there is less room for optimization. So this is not the final goal; only a way to understand how to make design easier.

The first set of inputs that may realistically be reclassified as external, are the outputs of the process level. In many design situations the process design is totally independent of circuit design (Figure 3.9). The process level outputs - design rules and available parameters, devices, and layers, that are inputs to the circuit and interconnect levels, are now externally defined inputs. Any outputs from other levels that were inputs to the process level are now irrelevant. These were the desired parameters, devices, and layers. There are now fewer tradeoffs and options to be considered in the processor design. The circuit and interconnect levels now have externally defined inputs. However, both the circuit and interconnect levels still have many internally generated inputs. The outputs of these two levels - available circuits, circuit block areas, interconnect area, loads, circuit block power, circuit block delays, interblock delays, and available

interconnect schemes, do not depend solely on the external inputs. The internally generated inputs influence all of these outputs to some extent. Thus, design is still difficult because there are no outputs that depend solely on external inputs.

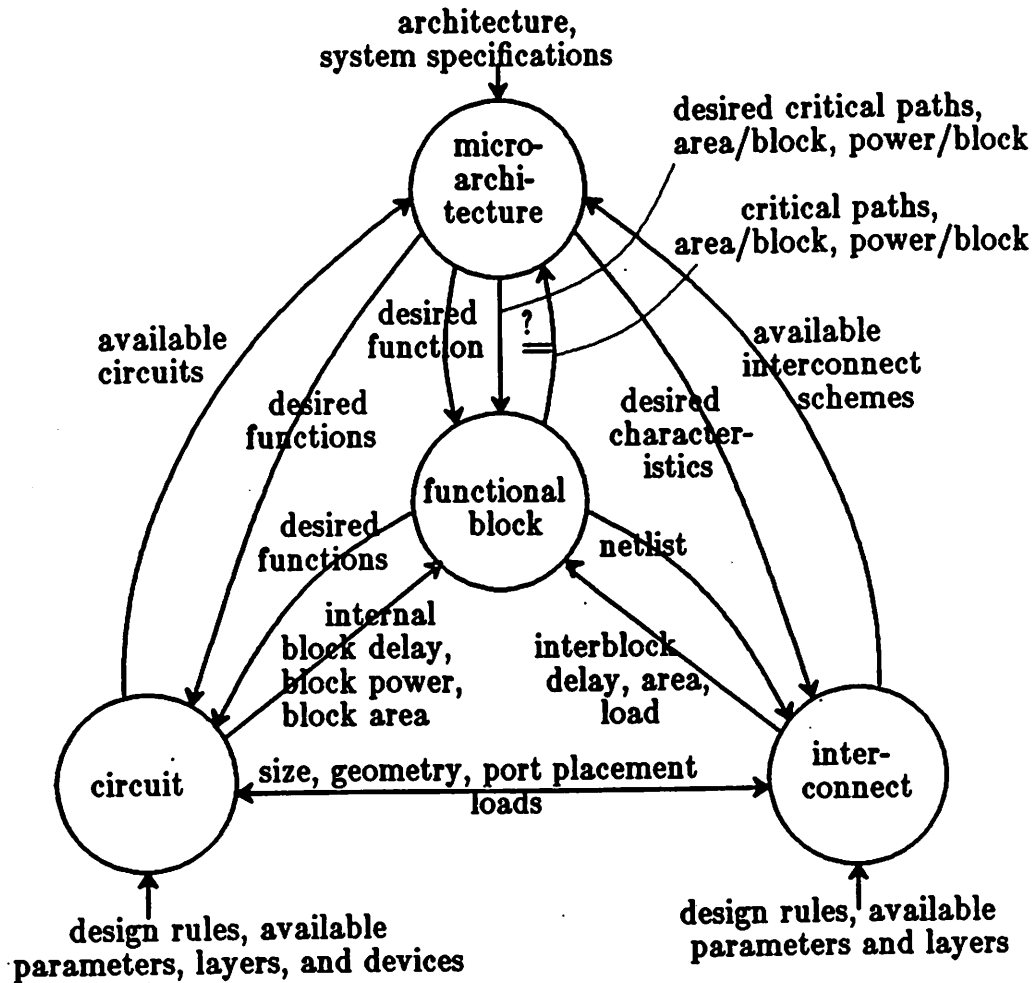


Figure 3.9- Flow Diagram- Predetermined Process

The standard cell design approach is popular in many design situations. Standard cell designs are based on a library of circuit cells that have been previously designed and characterized. The amount of circuitry within each cell depends upon the library under consideration. Typical cells are ALU bit slices, register cells, latch cells, counter cells, and various types of drivers. Circuit blocks are things such as ALUs, counters, shifters, registers, latches, PLAs, and ROMs.

They are each typically made of several cells from a cell library. Thus, circuit block design exists in the standard cell approach but the available circuits are now an external input (Figure 3.10). The possible ways in which to realize the desired function blocks are limited by the variety of cells in the library. Quite often there might be just one way that is clearly better than all other ways, to realize a desired block. Previously, the size, geometries, and port placement for the circuit blocks, were both inputs and outputs of the circuit design. Compromise between the interconnect and circuit design determined these characteristics. With standard cells, the size, geometries, and port placement are determined primarily by the circuits. They are fixed according to the standard cells. Therefore, interconnect design is constrained. Size, geometry, and port placement are no longer internal inputs to the circuit design level. Loads due to other circuit blocks also become external inputs, since they are determined by input loads of the standard cells. Thus, in standard cell design, design at the circuit level exists but can be far less involved than in a fully custom design. The circuit level design is left with two inputs – desired function and loads, which depend on other levels. All other inputs are now defined outside the scope of processor design for the circuit level. The microarchitecture level has also been simplified. It only has one input, available interconnect schemes, that is an output of another level. Limiting the number of inputs that depend on other levels of a design is analogous to reducing the number of inputs,  $z$ , which are complicated functions of the given inputs,  $x$ , and possibly the output,  $y$ , in the example  $y=f(x,z)$ .



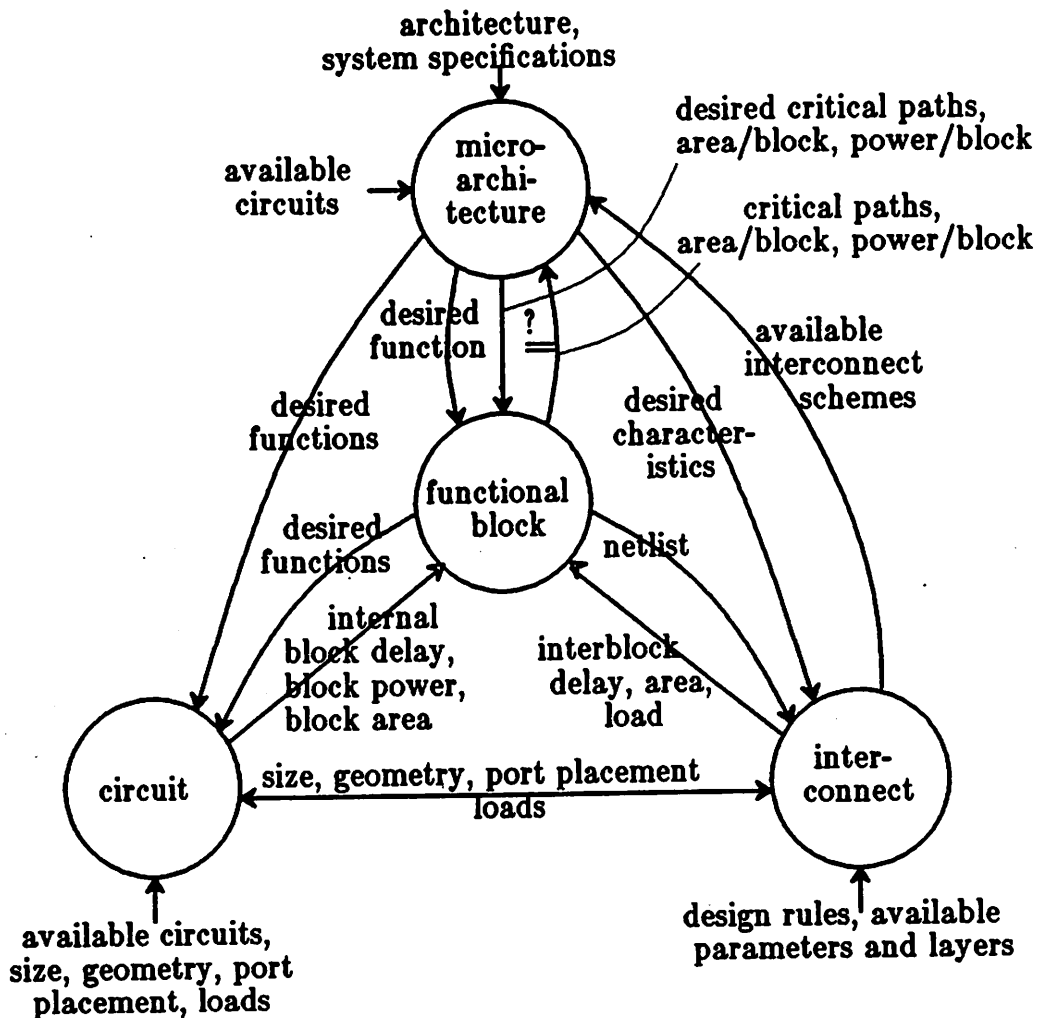


Figure 3.10- Flow Diagram- Standard Cell Design

## 6.2. Iteration

Although with the standard cell approach, the internal inputs have been limited, especially for the circuit and microarchitecture levels, there are still situations in which design at a given level depends in some way on design that has already been done at that level. For example, design at the functional block level depends on interblock delays which are an output of interconnect design. But interconnect design depends on functional block design through the netlist. This is analogous to  $y=f(x,z)$  but  $x=g(y,w)$  (Figure 3.11). These situations can be

found by identifying closed loops in the design flow diagram. When closed loops exist in the design flow, iteration is necessary to arrive at an optimal design. By eliminating process design from the processor design cycle, and using standard cells, design options and therefore, performance has been limited. Even with these limitations, iteration is still needed as shown by the remaining closed loops. Custom design was characterized by fewer fixed inputs and more closed loops.

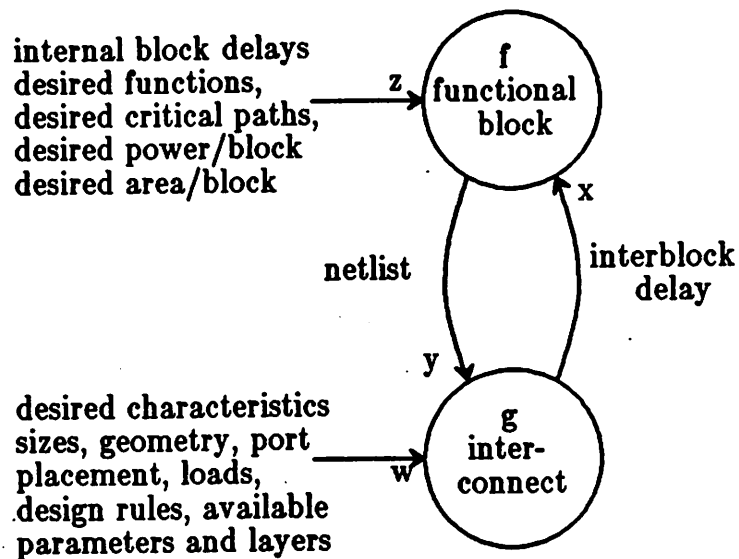


Figure 3.11- Iteration

The indirect dependency of design decisions at a given level, on decisions at that same level, makes optimal design difficult. These dependencies make iteration necessary. Iteration takes time which is one symptom of the difficulty. Any design method whose goal is an optimal design must have a way of dealing with the closed loops in the design flow diagram.

### 6.3. Unidirectional Design

So far it has been shown that design is easiest when the outputs of the level under consideration, are determined only by fixed external inputs. However, VLSI processor design does not fit this model. The design flow diagram for processor design is characterized by closed loops that involve two or more levels. This problem is much harder to solve than the one in which all inputs are external. A third possibility exists in which levels depend on each other but closed loops do not exist in the design flow diagram. This is analogous to  $y=f(x)$  and  $x=g(z)$  but  $z$  has no dependence on  $y$  (Figure 3.12a).  $Z$  might be an external input. If it is not itself an external input, it can be traced back to one without revisiting any of the previously used functions (Figure 3.12b). If no closed loops exist, design at any given level will not need to be redone due to input changes that were the result of previous design at that same level. Thus, no iteration is necessary. Design proceeds in one direction and is always a function of inputs from levels that are closer to the external inputs. This unidirectional design is more difficult than design with external inputs only, but easier than design that involves iteration.

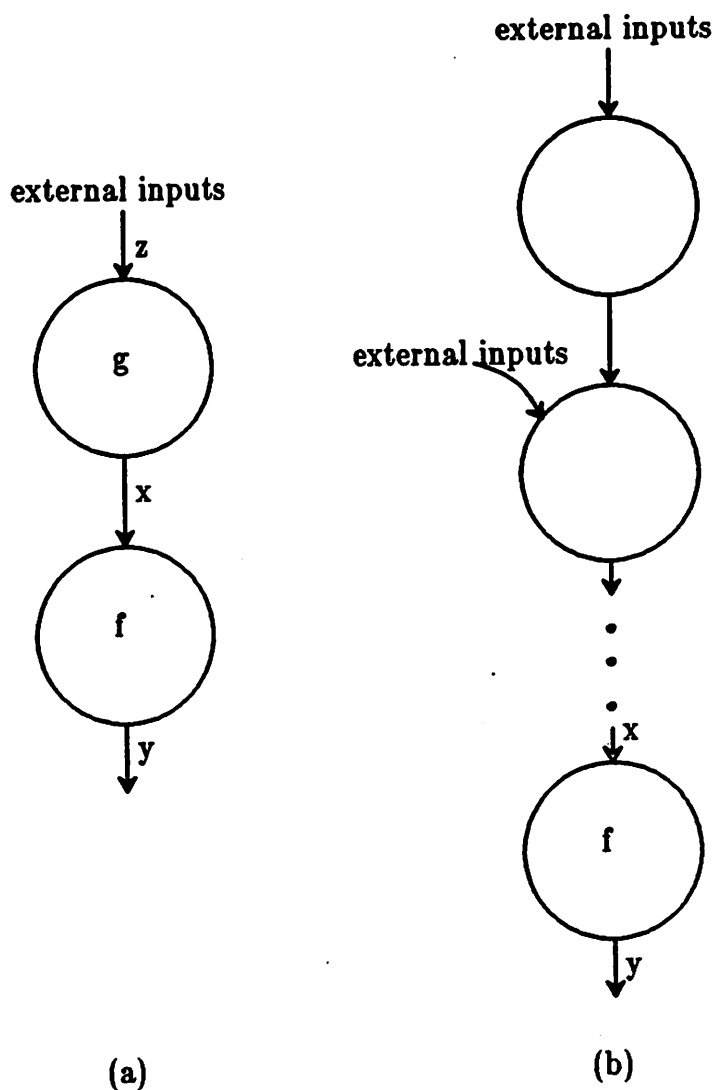


Figure 3.12- Unidirectional Design

## 7. Methodology Implications

Typical design flow diagrams for VLSI processors (Figures 3.6, 3.9, 3.10) all contain many closed loops that involve two or more levels. This is a source of complexity and makes optimal design difficult and time consuming. A good design methodology simplifies the design process while still considering all the relationships and tradeoffs between the design levels.

The flow diagram may be simplified by breaking it up into many simpler flow diagrams. Each simpler flow diagram must be considered at least once in order to complete an optimal design. If all the simpler flow diagrams are superimposed, the result must be the original complex diagram. If any connections between the levels are missing in the superimposed version, then the corresponding relationships and tradeoffs between the levels will not be considered in the design. Depending on the importance of these ignored relationships, some degree of optimality will be sacrificed.

Ideally, the flow diagram would be split into many diagrams, each having only one design level and all inputs would be external. As previously discussed, this is the easiest type of design. However, this type of split will artificially isolate the design levels far too much. Some type of split must be found that does not force the design process into a framework which is so artificial, that optimal design is obscure. More difficult than design with external inputs only, but easier than iterative design, is unidirectional design. Whenever possible the simpler flow diagrams are restricted to have external inputs and unidirectional design only. There will still be some situations where this restraint is still too artificial. When these situations arise simple loops may be added to the flow diagram. Simple loops are closed loops that involve the fewest levels possible – ideally only two. They also are as isolated from other parts of the flow diagram as possible.

As the overall flow diagram is split into many diagrams, care must be taken to ensure that the splits are in some way natural splits. One logical split is the split between phases of a design. Typical phases are synthesis, analysis, and optimization. During the synthesis phase, one or many solutions to the design problem are proposed. Analysis consists of determining various characteristics, such as speed, power, and area, for the proposed solutions. Using the results of analysis and possibly further analysis, fine tuning and elimination of bottlenecks is

accomplished. This is optimization. Each phase has its own flow diagram which includes the important relationships and tradeoffs to be considered, for that phase. The flow diagrams for this type of split, represent the natural phases of design, and are also simpler, as previously described, than the original diagram. Design becomes more manageable due to the splitting of the original problem into many simpler problems, and remains realistic by choosing each simpler problem to correspond to a natural design phase. Thus, time, in the form of design phases is used to organize an originally complex problem into many simpler problems.

Once the flow diagram has been broken into many simpler diagrams and each simpler diagram represents a design phase, the diagrams must be considered in some logical sequence. Information that is needed for a given diagram must be determined by a previous diagram. For example an analysis diagram must follow a synthesis diagram. A solution must first be proposed or synthesized, before it can be analyzed. Thus, the sequence in which the flow diagrams are considered is determined by the natural order of the design phases.

## **8. Summary**

VLSI processor design can be broken into five major levels – microarchitecture, functional block, circuit, interconnect, and process levels. These design levels use various inputs, that may be either defined outside the scope of processor design – external inputs, or generated by other levels – internal inputs. These levels are distinguished by the problems that they address and the way that processor characteristics are analyzed (Tables 3.1 and 3.2). Processor representations also vary according to the level. Some levels can have more than one type of representation or way to analyze the characteristics. This leads to the definition of sublevels.

<b>Design Level</b>	<b>Problems Addressed</b>
<b>Microarchitecture</b>	<b>Identify functional blocks</b> <b>Timing</b>
<b>Functional block</b>	<b>Map functional blocks to circuit blocks</b> <b>Define interconnects</b>
<b>Circuit</b>	<b>Circuit block layout</b>
<b>Interconnect</b>	<b>Interconnect layout</b>
<b>Process</b>	<b>Process development</b>

**Table 3.1- Problems Addressed by the Design Levels**

Design Level	Speed Criteria
Microarchitecture	Cycles/instruction Phases/cycle Activities/phase
Functional block	Delay/activity
Circuit	Delay/circuit block
Interconnect	Delay/interconnect
Process	Process parameters influencing speed

Table 3.2- Speed Analysis According to Design Level

Many relationships and tradeoffs exist between the design levels, as shown by the numerous internal inputs. This is an important source of the complexity in VLSI design. Consideration of these design levels and their interrelationships can be shown graphically in an overall flow diagram for processor design. This total flow diagram has many complex loops between the levels, which indicate that iteration is necessary for optimal design. Simultaneous consideration of all possible tradeoffs is difficult. Therefore, the total flow diagram is split into many simpler flow diagrams, each representing a design phase. The complex loops are spread over the various flow diagrams by splitting the paths that they are composed of, between the simpler diagrams. Thus, iteration is minimized within any given phase. The total required iteration is now spread out in time, over the different design phases. Iteration is accomplished as the designer progresses through the phases, represented by the simpler flow diagrams. From a practical



viewpoint, design is considered to be optimal when no changes are made as the designer moves through a specified number of phases. This design methodology is a way to organize the inherent iteration due to complex relationships, in processor design. The iteration and consideration of tradeoffs is organized into some logical sequence so that an optimal design can be achieved. Overlooking and ignoring possible tradeoffs is eliminated because all diagrams, when superimposed, must result in the original complete flow diagram.

## **9. References**

[Thom83] Thomas, D. E.; Nestor, J. A.; 'Defining and Implementing a Multilevel Design Representation with Simulation Applications', *IEEE Transactions on Computer Aided Design*, V.2, N.3, July 1983.

## Chapter 4

### Design Methodology

Design level interrelationships and tradeoffs for VLSI processor design, can be shown graphically by an overall flow diagram. In order to make the design problem more manageable, this overall flow diagram is broken into simpler diagrams. Each simpler diagram should be associated with a natural design phase so that design remains realistic. Three main design phases for VLSI processor design can be identified as:

Preliminary

Synthesis

Analysis

Just as the design levels have sublevels, the design phases can have subphases. Optimization is possibly another phase. However, optimization can be shown to be composed of sequential synthesis and analysis steps. For this reason it is not included in the list of primary phases.

The preliminary phase is the first phase of design. It assumes that the architecture, system specifications, and processing techniques are known. The goal of this phase is to explore possible options for circuits and interconnect schemes. These options are based on the requirements of the architecture and system specifications. They are limited by processing technology.

The second major design phase is synthesis. The goal of synthesis is to arrive at a detailed solution to the design problem. Synthesis steps all contribute to the transformation of a high level problem into a detailed, low level solution. Verification is one important subphase of synthesis. A proposed solution is shown to correctly answer a problem through verification.

Analysis is a third major phase. It consists of examining various aspects of the performance of proposed solutions. Based on performance predictions and any

bottlenecks that are observed during analysis, further synthesis might be done.

## **1. Preliminary**

At the start of the preliminary phase, the only things known about the processor design are the external inputs. These are the processing techniques, system specifications, and architecture, for a fully custom design. In many design situations the process is fixed. Therefore, in these situations the available devices, layers, and process parameters are fixed external inputs. For standard cell design the circuits available are also external inputs.

The flow diagram for the preliminary phase of a full custom design is shown in Figure 4.1. It contains loops which means that iteration is possible. There are two halves to the flow diagram— the microarchitecture, circuit, process side and the microarchitecture, interconnect, process side. If all possible loops are considered in this diagram design things can get complicated. However, the two sides may be treated independently (Figure 4.2). Solutions for all levels involved, can be arrived at for each side and then the two solutions can be compared, for the process and microarchitecture levels. If contradictions between the two sets of solutions for these levels exist, compromises must be worked out.

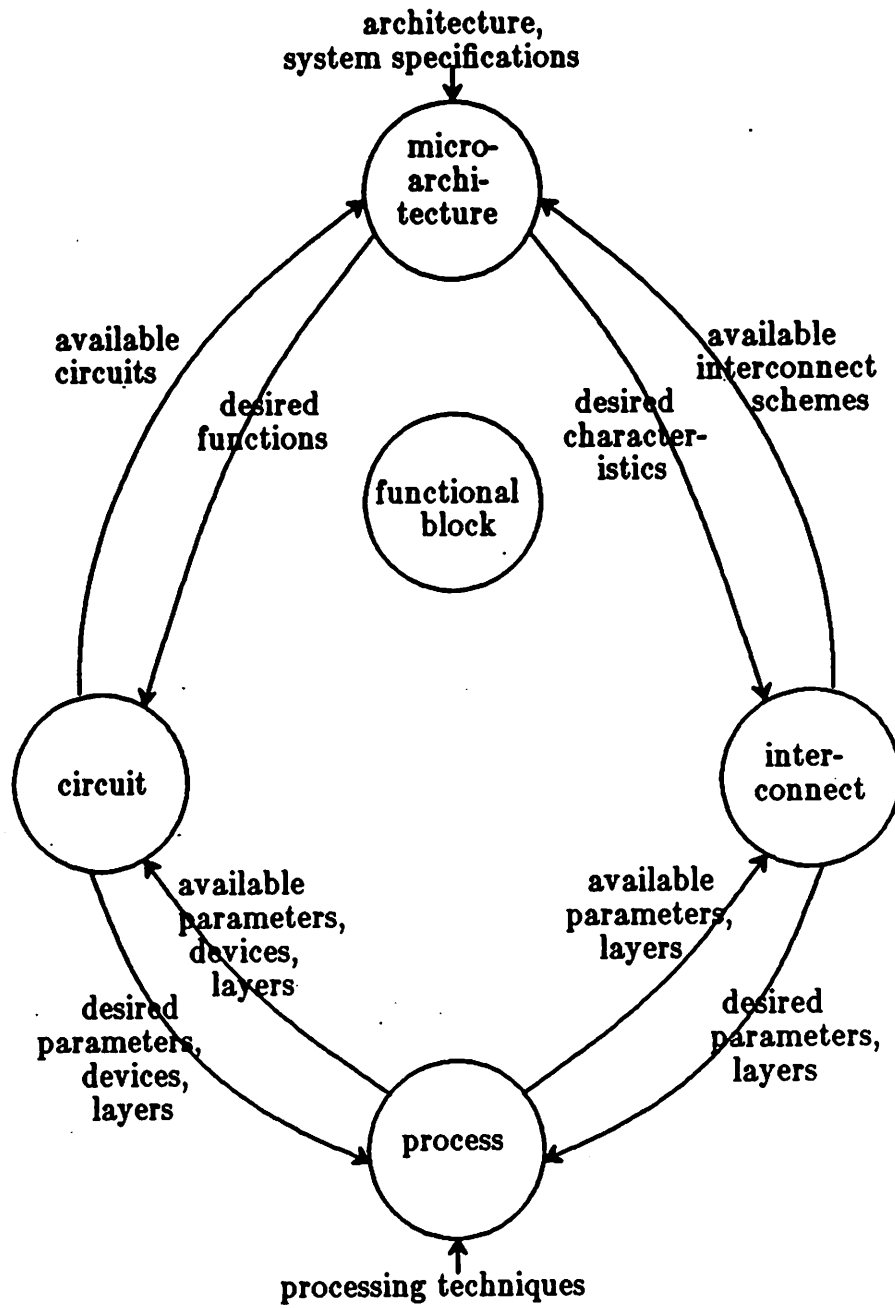


Figure 4.1- Preliminary Phase- Full Custom Design

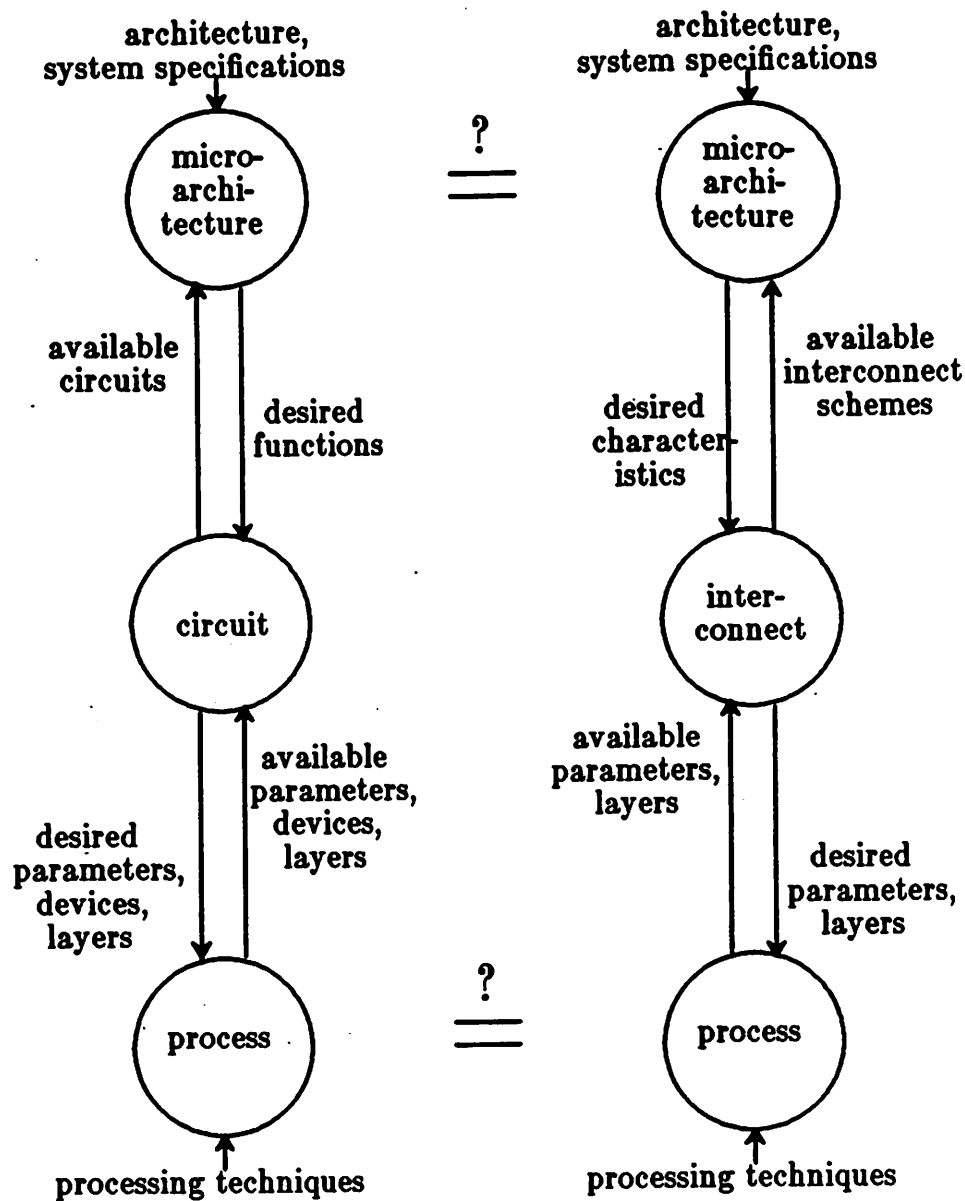


Figure 4.2- Preliminary Phase Simplification

When the process is fixed the flow diagram becomes much simpler (Figure 4.3). There are still two sides, but each side is now only a simple loop involving only two levels. Standard cell design would further simplify the preliminary phase by removing all circuit design from this phase (Figure 4.4). The circuit loop is replaced by the external input of the available circuits, and only the interconnect loop remains.

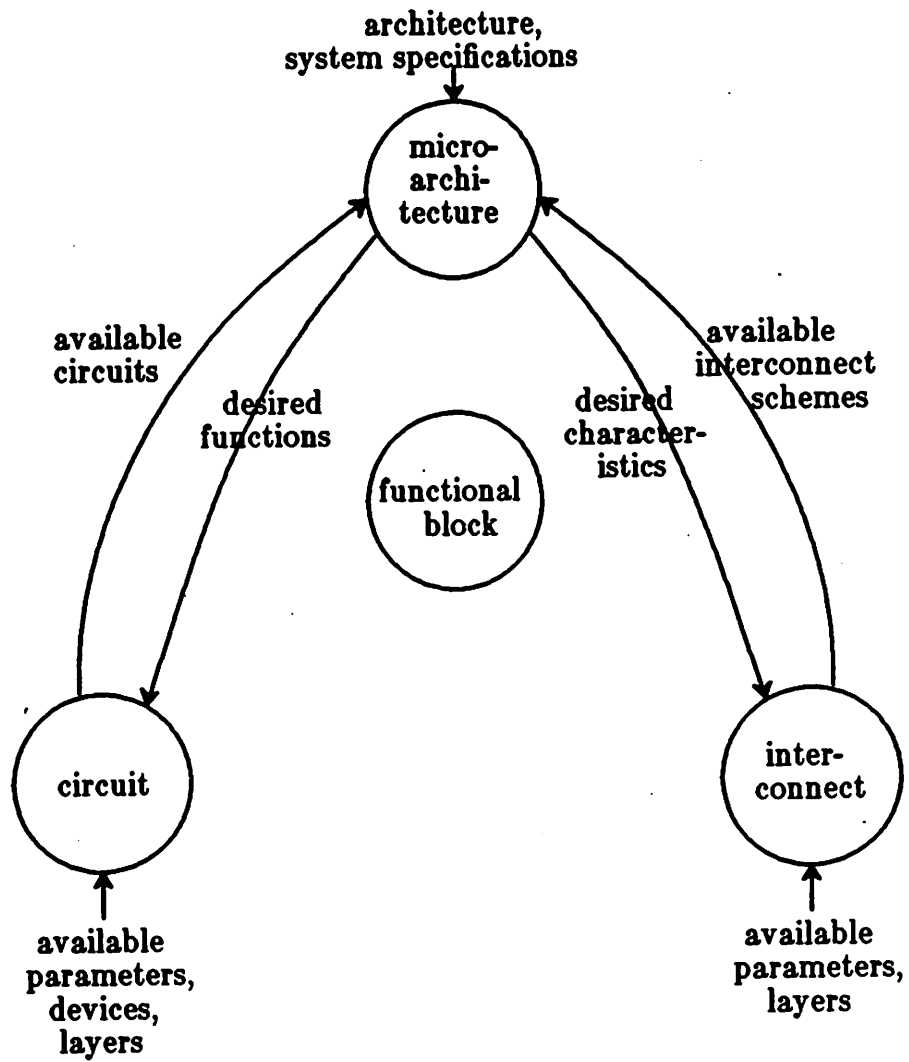


Figure 4.3- Preliminary Phase- Fixed Process

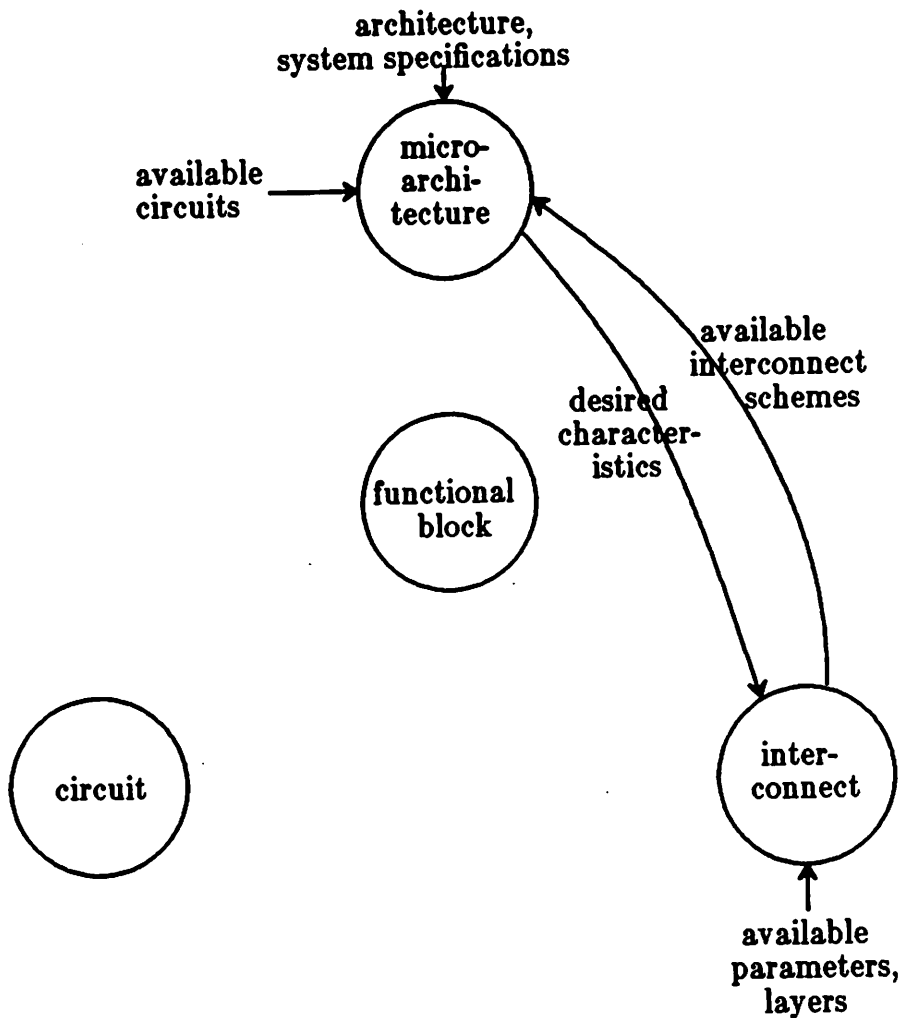


Figure 4.4- Preliminary Phase- Standard Cell Design

During the preliminary phase, the architecture and system specifications are examined to determine various functions that the circuits must provide. This is done most efficiently if various microarchitectures can be considered to determine how well they will satisfy the architecture and system requirements. A list of circuit functions needed by the most promising microarchitectures is developed. Preliminary circuit design can then be carried out so that the necessary circuits and their characterization are available to the microarchitecture design. This is represented by the available circuits input to the microarchitecture level (Figure 4.1). Examination of the architecture alone reveals the operations that must be

performed by the ALUs. System specifications will place requirements on such things as pad driver circuitry. High level microarchitecture considerations influence circuit function also. For example pipeline design specifies which tasks are to be done in parallel. The amount of parallelism dictates the hardware that will be needed, such as the number and types of ALUs or register ports. Timing of the various pipeline stages places speed restrictions on the circuits. If microcode is to be used, ROMs must be designed. These are all examples of ways in which the microarchitecture influences circuit design in the preliminary phase. Graphically this is shown in Figures 4.1, 4.2, and 4.3 by the desired functions arrow, between the microarchitecture and circuit levels.

If the process is open to modification, the circuit designer can request various parameters and devices that will make his job easier, from the process designer. This might include such things as substrate bipolar transistors, diodes, various threshold voltages, or buried contacts, in an MOS process. With these requests in mind, the process designer uses the processing techniques and equipment that are available, and informs the circuit designer of the parameters and devices to be used. Graphically this is shown by the loop between the process and circuit levels (Figure 4.1).

Using the process parameters and devices that have been provided by the process level, design at the circuit level tries to accommodate the desired function input. This results in the output of the available circuits from the circuit level. These available circuits and their characteristics, such as speed and power, are used to form a more detailed microarchitecture design.

A similar set of loops exists for interconnect design in the preliminary phase (Figure 4.1). The interrelationships between the microarchitecture and interconnect levels are not as apparent as those between the circuit and microarchitecture levels. They do exist though, and should therefore be



considered for a truly optimal design. An example of this relationship is the amount of communication between various subsystems of a processor. The desired speed of this communication is another example. These considerations might cause the interconnect designer to request extra interconnect layers and/or interconnect layers with low resistance and capacitance, from the process.

Just as the process accommodated the circuit designer requests whenever possible, it tries to meet the interconnect level requests. Knowing the process parameters and interconnect layers that are available, design at the interconnect level produces possible interconnect schemes for use in microarchitecture design.

One result of the preliminary phase is that the process will no longer change due to inputs from the design. However, it still might change if the external input, processing techniques, changes. This makes the designer's job harder. Assuming this does not happen though, inputs from the process level for all future phases are now external inputs. Loops involving the process level have been considered as appropriate, in this preliminary phase and eliminated from future phases. Process design does not complicate synthesis and analysis. It merely provides inputs which are used by the synthesis and analysis phases.

## **2. Synthesis**

Synthesis is the process of transforming a high level description of the behavior of a processor into a working VLSI circuit. The goal of synthesis is to provide a circuit that functions correctly. Correct functionality can mean different things at different stages in the design. In the early stages it may mean performing the required operations with no concern for speed or power dissipation. In other words, meeting the architectural requirements is the first step in attaining correct functionality. Later stages are concerned with meeting all system specifications also. Various types of analysis are done after synthesis

has generated the first proposed solutions. As a result of this, further synthesis is done to attain all unmet system requirements.

External inputs for synthesis differ somewhat from those of the preliminary phase. Process design occurs during the preliminary phase if the process is open to modification in response to the chip designers' needs. The process may still change after this but it is out of the control of the chip designers. However, the chip designers must still deal with any changes that may occur. Outputs of the process level become external inputs from the chip designers' points of view, after the preliminary phase. Therefore, the design rules, layers available, and devices available are all external inputs to design during the synthesis phase. At the high level, the behavioral description of the processor - architecture and system specifications - are external inputs. The VLSI circuit must realize this description. The circuits available and possible interconnect schemes from the preliminary phase are also inputs to microarchitecture design. They are from the preliminary phase and therefore, are considered as fixed inputs. They are needed to complete the details of the microarchitecture. The flow diagram for synthesis is shown in Figure 4.5. The area enclosed by the dashed lines represents the scope of design during synthesis. Arrows crossing the dashed lines are external inputs for the synthesis phase. Figure 4.6 shows the flow diagram for synthesis and includes only the design levels of concern to the synthesis phase. The process level has been removed from this diagram.

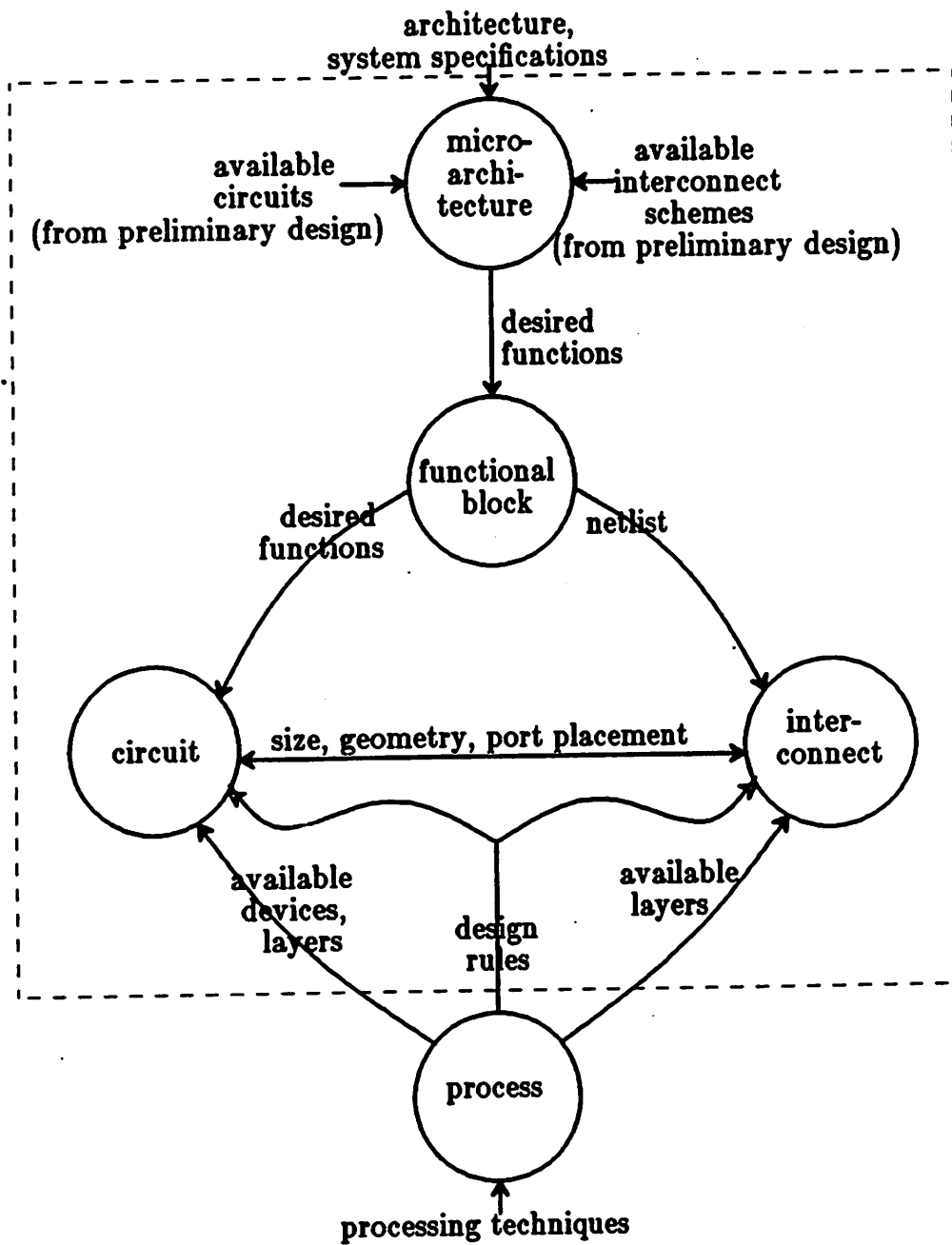


Figure 4.5- Design Levels During Synthesis

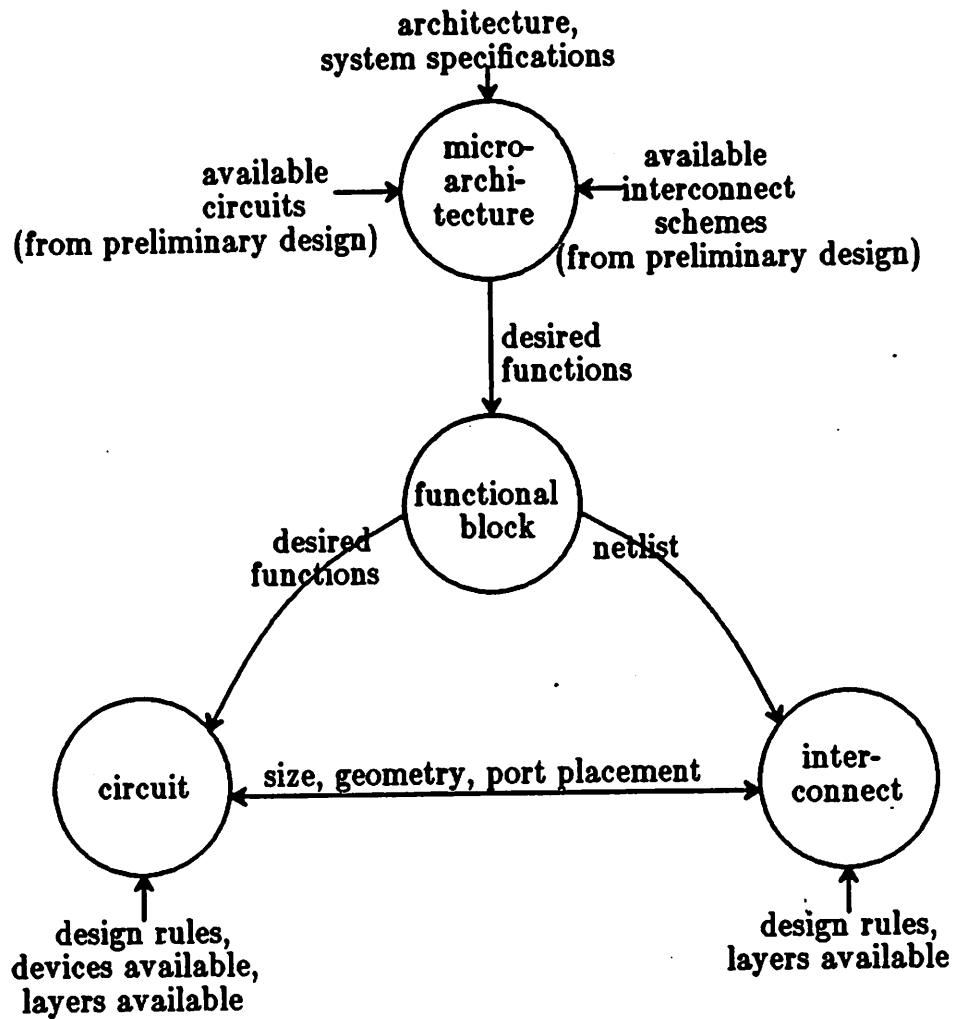


Figure 4.6- Synthesis- Full Custom Design

Synthesis is basically unidirectional. Design proceeds from the external and fixed inputs through various intermediate stages, to the circuit and interconnect levels without revisiting any intermediate stage. However, in a fully custom design the circuit and interconnect levels directly affect each other. A simple closed loop is formed by these two levels and is shown in Figure 4.6 by the bidirectional arrow between them. This makes design more complex than a completely unidirectional design, but it is still much simpler than the original problem.

Standard cell design would eliminate this bidirectional arrow (Figure 4.7). Size, geometry, and port placement of the circuit blocks are determined by the standard cells. The interconnect level has little effect on these characteristics. Thus, for standard cell design, the synthesis phase is purely unidirectional.

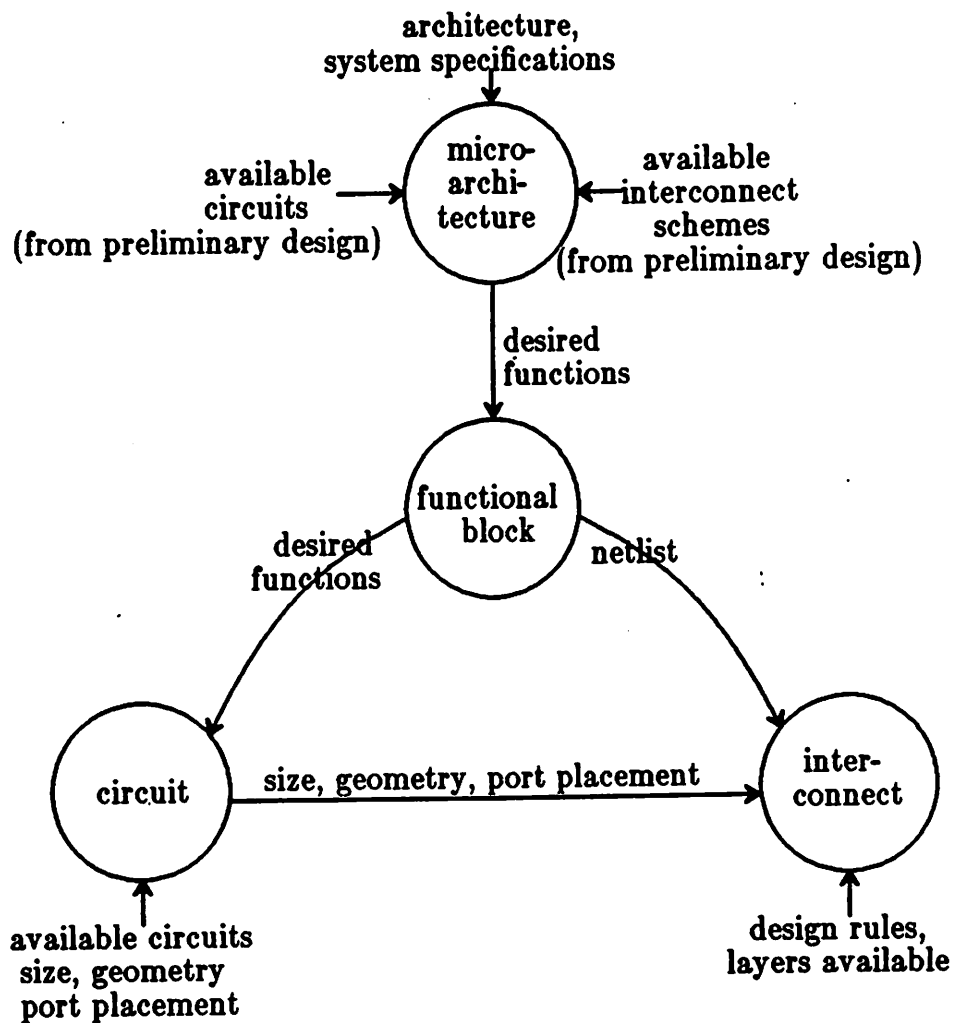


Figure 4.7- Synthesis- Standard Cell Design

Synthesis is commonly described as top down design with technology considerations. Design flows from the highest point – the architecture and system specifications external inputs – through the levels, from highest to lowest. Synthesis is completed when the lowest sublevel – layout – of the circuit and

interconnect levels have been finished. Technological considerations, in the form of design rules and available devices and layers, enter the design at the circuit and interconnect levels.

### **2.1. Microarchitecture Synthesis**

Synthesis starts with a description of the architecture and system specifications. Such things as the instruction set, trapping and interrupt situations, word size, data types, register organization, addressing modes, I/O protocols, bus protocols, coprocessor protocols, and timing specifications are important to the synthesis phase. Using this information and the available circuits and interconnect schemes from the preliminary phase, synthesis begins with microarchitecture design. Microarchitecture design defines the functional blocks of the processor and coordinates their operation. Functional blocks may or may not be the final circuit blocks. The circuit blocks are the actual blocks that are realized in the layout. Functional blocks are a different partitioning of these circuit blocks. One functional block may turn into one or many circuit blocks, or it may be optimal to combine several functional blocks into a single circuit block. An example of this are PLAs. A given function may correspond to a single functional block but be implemented with more than one PLA. Each PLA is a circuit block and the group of PLAs is the functional block. Another example is an ALU. An ALU combines several arithmetic and logical functions that may each have been specified by an individual functional block.

The microarchitecture is composed of various types of functional blocks as needed by the architecture and system specifications. From the instruction set the operations that the processor must perform, can be identified. Operation blocks corresponding to these operations must be included in the microarchitecture. The register organization will define many storage blocks that

are necessary. Trapping situations, interrupt detection, various protocols, and instructions that inspect conditions all must recognize specific situations. Condition detection blocks are needed for this. Word size determines the size of some blocks such as register blocks in the datapath. Control blocks are needed to coordinate the processor's functioning. These functional blocks are also known as the processor resources.

The microarchitecture must also coordinate the processor's operation. This is done by scheduling the usage of processor resources or functional blocks. This includes defining the parallelism in the processor and pipeline operation. Bus structures and temporary register blocks are also defined as needed, to ensure a smoothly functioning processor. An inspection of the architecture – the instruction set in particular – identifies the operations to be done and the order that they may be done in, for each instruction. For example a simple register add instruction first requires an instruction fetch. The operands are then read, added together, and the result is then written. Often there is more than one possible sequence of events for an instruction. Various operations may be done simultaneously. For example an instruction that does an add operation and a condition check on the operands, may do the two operations simultaneously or sequentially (Figure 4.8). Parallelism is given to a processor by identifying and deciding which functions are to be performed simultaneously. Increasing parallelism decreases the latency for each instruction. However, it also necessitates more hardware to provide the simultaneous processing capabilities.

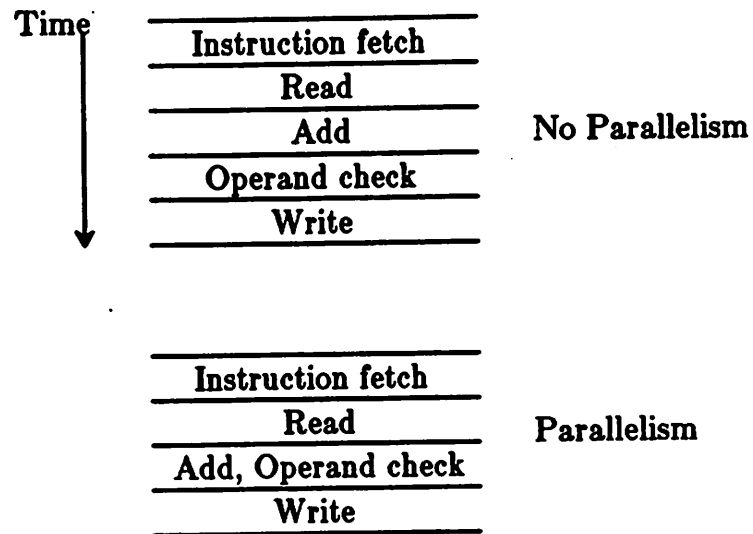
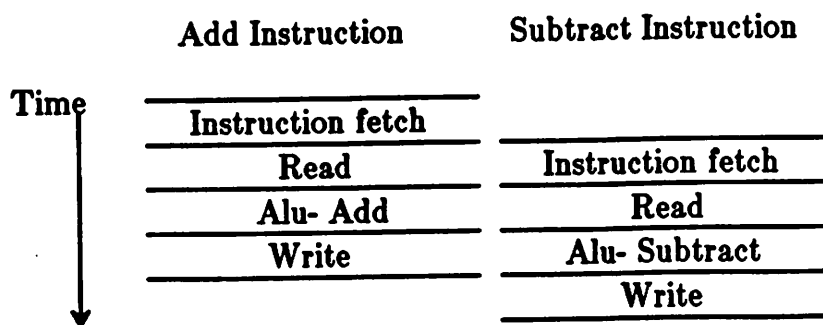


Figure 4.8- Parallelism vs. No Parallelism for an Operand Check

Often many instructions can use resources in the same order. When this is possible it is not necessary to let one instruction complete before the next one starts. It is only necessary to let one instruction finish with a given resource before the next instruction is allowed access to that resource (Figure 4.9). In this way execution of instructions may be overlapped and resources are more fully utilized. This is known as pipelining. Pipelining leads to a higher instruction rate and consequently higher processor throughput. However, it also leads to more complex bus structures and communications between the blocks in order to handle the increased processing activity. Parallelism and pipelining are two major concerns of microarchitecture design, for the coordination of resources.





Add followed by a Subtract

Figure 4.9- Pipelining

The detailed microarchitecture specifies the inputs, outputs, and timing of all functional blocks. To do this, key circuits from preliminary circuit design must be considered. Inputs, outputs, and clocking for some of these blocks will be determined by the circuits that realize them. One purpose of preliminary design was to have this information readily available for detailed microarchitecture design. Thus, the circuits available from the preliminary phase are fixed inputs to microarchitecture design during synthesis.

In a similar manner microarchitecture design may be limited by the available interconnect schemes. For example, the number of buses that can be routed across a bit slice of the datapath will place a limit on the bus structure. These schemes from the preliminary phase are another fixed input to microarchitecture synthesis.

## 2.2. Functional Block Synthesis

Microarchitecture design specifies the functional blocks during synthesis and the coordination between these blocks. The outputs of the microarchitecture level during synthesis, are the desired functions of the blocks. This includes the operations done within each block and the inputs and outputs of each block.

Block inputs include the data to be operated on and signals to control the block's functioning. Outputs of the blocks are the results and any conditions that are relevant to other blocks. The desired function outputs of the microarchitecture level are used as inputs by the functional block design level.

Functional block design is done after the microarchitecture design level has defined the functional blocks and their coordination. It maps the desired functions onto circuit blocks. After this mapping is done the desired function is known for each circuit block. Just as with the functional blocks, the desired functions include the operations performed and terminals for each circuit block. These desired functions are outputs of the functional block design level and are passed onto the circuit design level. The other output of functional block design for synthesis is the netlist. This specifies all connections between the circuit blocks. It may also specify restrictions on critical nets. The netlist is used as an input by the interconnect level.

A floorplan for the processor is developed through the functional block design. Once the actual circuit blocks and their terminals are known a tentative positioning of the blocks on the chip may be proposed. Space for the various interconnects is also designated on this floorplan. Another useful document from the functional block design is a block diagram of the chip. This diagram illustrates the circuit blocks and all major connections.

### **2.3. Circuit Synthesis**

Circuit design is carried out once the desired functions for each circuit block are known. Typically, the circuit block is first broken into smaller blocks or cells. Examples of cells are latch cells, register cells, ALU bit slices, drivers, and any circuitry that is replicated in regular structures such as PLAs and ROMs. If standard cell design is done, the layouts for the required cells are then chosen

from the cell library and assembled into the circuit block. In a fully custom design the cells are then expressed in greater detail by describing them in terms of the gates and transistors that compose them. Finally, the physical layout for the circuit block is achieved by using the design rules, devices available, and layers available - all inputs to circuit design from the process level. While doing the circuit block layout, consideration must be given to various outputs of the interconnect level. The interconnect scheme may desire certain ranges of sizes and shapes for the circuit blocks. This is to make routing simpler and have a minimum of wasted space. An example of this is pitch matching of cells so that circuit blocks may be placed adjacent to each other. Desired port placement is another input from the interconnect level. Typically it is desirable for control lines to enter the cells on a specified side of the cell. Ordering of the ports is also important when circuit blocks are placed adjacent to each other. The ports of one block should connect directly to the ports of the adjacent block with no extra routing between the blocks.

#### **2.4. Interconnect Synthesis**

During synthesis interconnect design consists of connecting all terminals of the circuit blocks to the appropriate places in the layout. A netlist from functional block design specifies all connections that must be made. The sizes and shapes of circuit blocks that must be routed around, are important when doing routing so that they may be avoided. This may not be all circuit blocks, especially when two levels of metal or polysilicon are available for routing. A knowledge of the placement of ports on the circuit blocks is important so that the proper connections are made. Interconnects are realized on the physical layout in the interconnect layers made available by the process. The minimum widths and spacings of the interconnects are determined by the design rules of the process.

### 3. Analysis

The third major phase of design is analysis. During analysis important characteristics of the design are evaluated. Three characteristics commonly evaluated are the speed, power, and area. The goal of the evaluation is twofold. First, it is important to know where the design stands for any of these characteristics. Typically, a value or range of values for these characteristics, is predicted. The second part of the evaluation is more subjective. Attempts are made to discover the reasons for any less than adequate characteristics and to pinpoint any bottlenecks. With the results of analysis further synthesis may be done to improve the design.

As just mentioned, many characteristics may be analyzed. Tradeoffs exist between the characteristics. For example, circuits that run faster usually consume more power.

System specifications dictate the priorities of the various characteristics. Usually it is harder to meet the requirements for some characteristics than for others. A separate flow diagram exists for the analysis of each characteristic. The structures of these flow diagrams are all similar but the quantities involved differ. Each characteristic is analyzed according to its flow diagram. The results of all analyses are compared and tradeoffs are made according to priorities and unmet requirements.

A flow diagram for the analysis of speed is shown in Figure 4.10. All levels of design are shown here. However, analysis is done only at the levels enclosed by the dashed lines. All process design that was sensitive to the chip designers' needs, was carried out during the preliminary phase. Figure 4.11 shows only the levels at which analysis is done. Analysis uses the process parameters as external inputs. Values for the characteristics of interest are derived from these parameters and the process design.

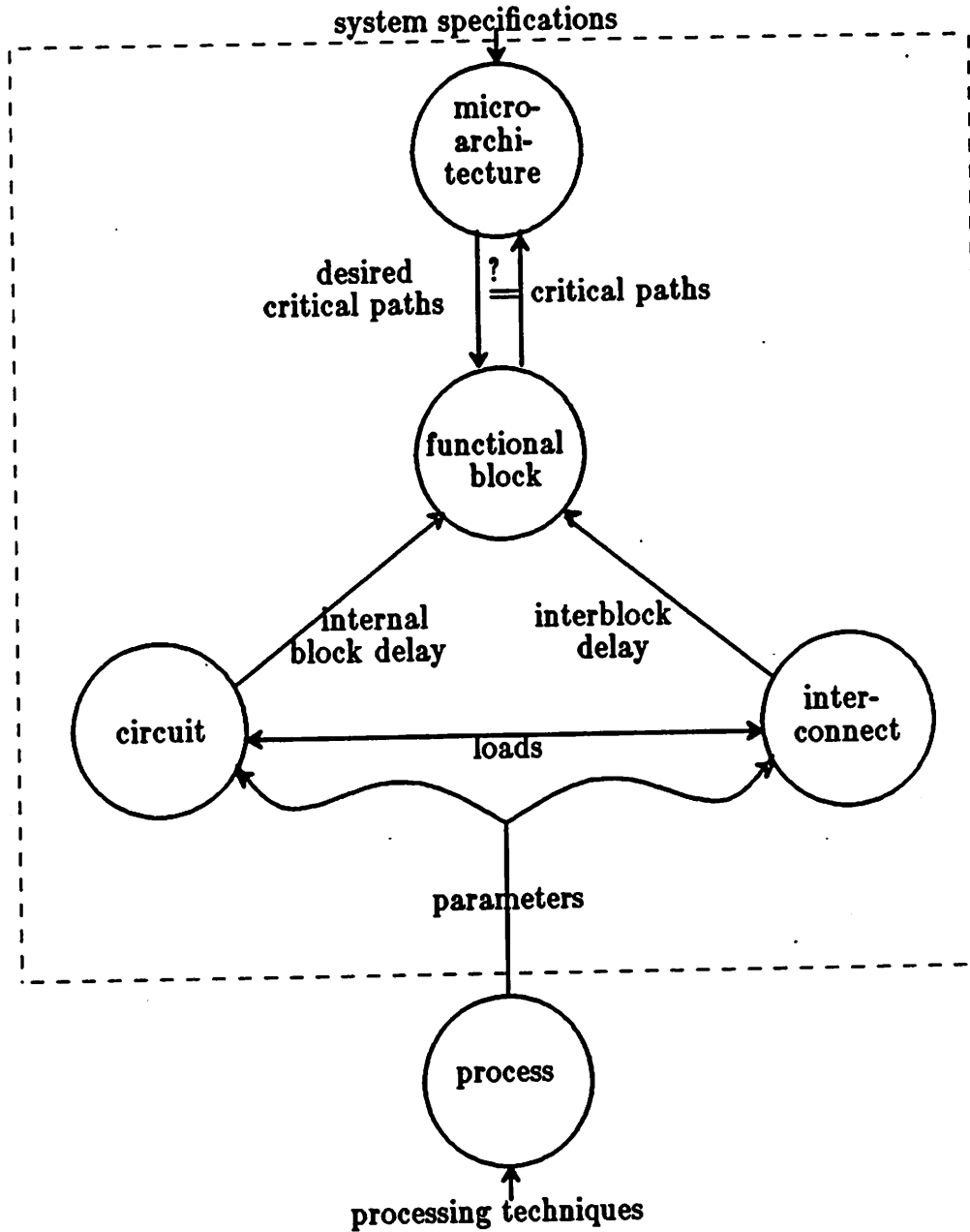


Figure 4.10- Design Levels During Speed Analysis

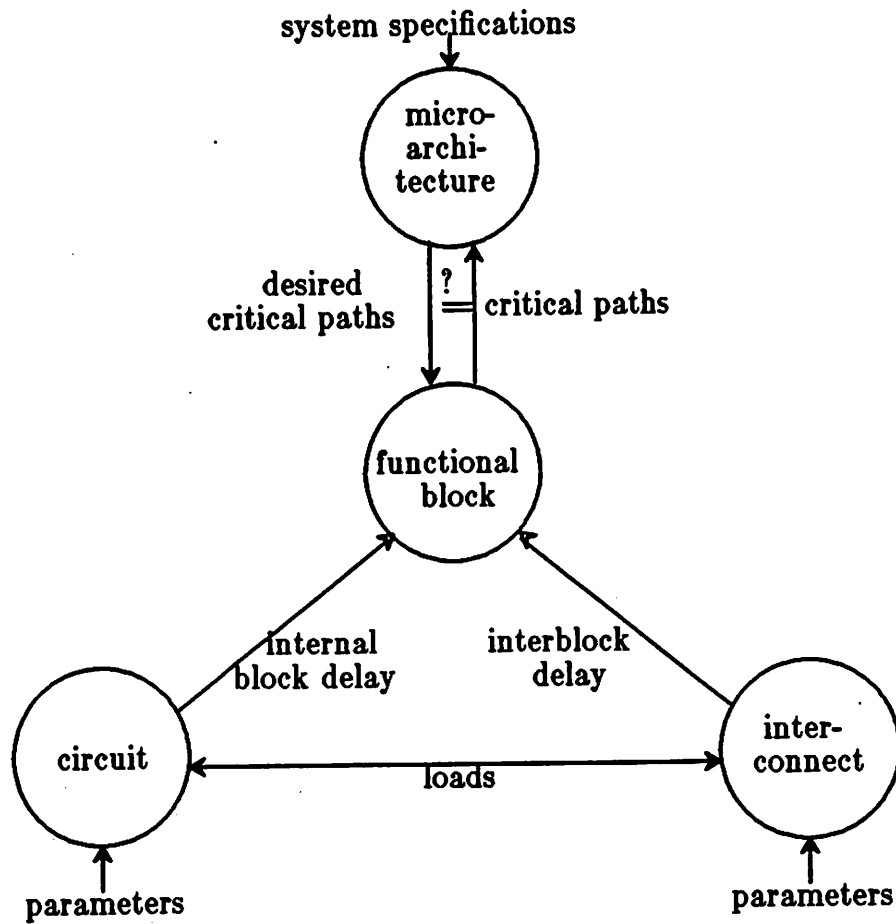


Figure 4.11- Analysis- Speed

Analysis is basically unidirectional also. However, there are two separate paths, both unidirectional (Figure 4.12). Values at the endpoints of the two paths are compared to determine if the characteristic meets its requirements. The exception to purely unidirectional analysis is found between the circuit and interconnect levels. A simple closed loop exists here, as indicated by the bidirectional arrow, due to circuit and interconnect loading.

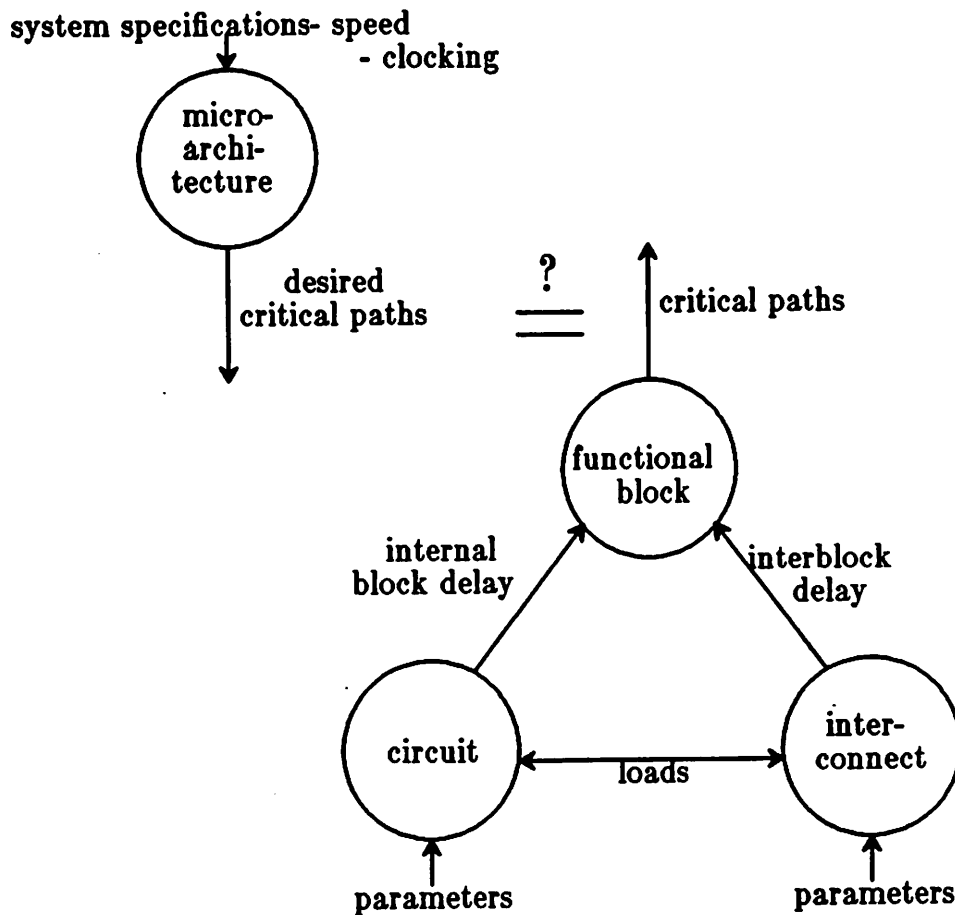


Figure 4.12- Unidirectional Paths of Speed Analysis

One unidirectional path represents a top down analysis. For speed, the desired critical paths are based on the system specifications. They are arrived at through an analysis of the microarchitecture. From synthesis, microarchitecture design proposed functional blocks and coordination of the blocks. One aspect of the coordination is a listing of all activities that must complete during each phase of the system clock. Analysis of the microarchitecture reveals the activities of each phase. A length of time is allotted to each phase by the system specifications. All activities of each phase must complete within the allowed time period. The critical paths are the activities that take the longest to complete. In Figure 4.12 the desired critical paths outputs of the microarchitecture level are these longest activities and the times available to them.

The other unidirectional path is a bottom up analysis that predicts the times for the critical paths from the processing parameters and an analysis of the lower design levels – circuit, interconnect, and functional block. Internal block delays are predicted for the circuit blocks using process parameters. Loading due to the interconnects affects this delay in a fully custom design. Output buffers of the block are designed according to the size of the load that must be driven. These buffers are part of the circuit blocks and the loading that they put on other parts of the block contributes to the total block delay. The speed of the block may also be affected by the transition times of input signals to the block. These transition times are determined by the output buffer of another circuit block, the loading of that buffer due to the interconnect, and the circuit being driven.

Interblock delays are also predicted during the bottom up analysis. These are the delays of the interconnects. Process parameters such as resistance and capacitance along with the interconnect dimensions, determine the maximum possible speed for each interconnect and load that the circuit block must drive. The speed of the interconnect and therefore the interblock delay is determined by the interconnect load and the circuit that drives it.

These internal block delays and interblock delays are used in an analysis of speed at the functional block level. The delay for each critical path can be predicted at this level, by adding together the delays for all circuit blocks and interconnects that compose the critical path. The predicted critical paths from this bottom up analysis are then compared to the desired critical paths of the top down analysis. Individual delays of the critical paths are examined to reveal the bottlenecks if a faster processor than what has been predicted, is desired.

As previously mentioned, this same type of analysis is carried out for all characteristics of interest. Figures 4.13 and 4.14 show flow diagrams for power and area analysis, respectively. Compromises between all characteristics must be



arrived at so that all system specifications are satisfied according to their priorities.

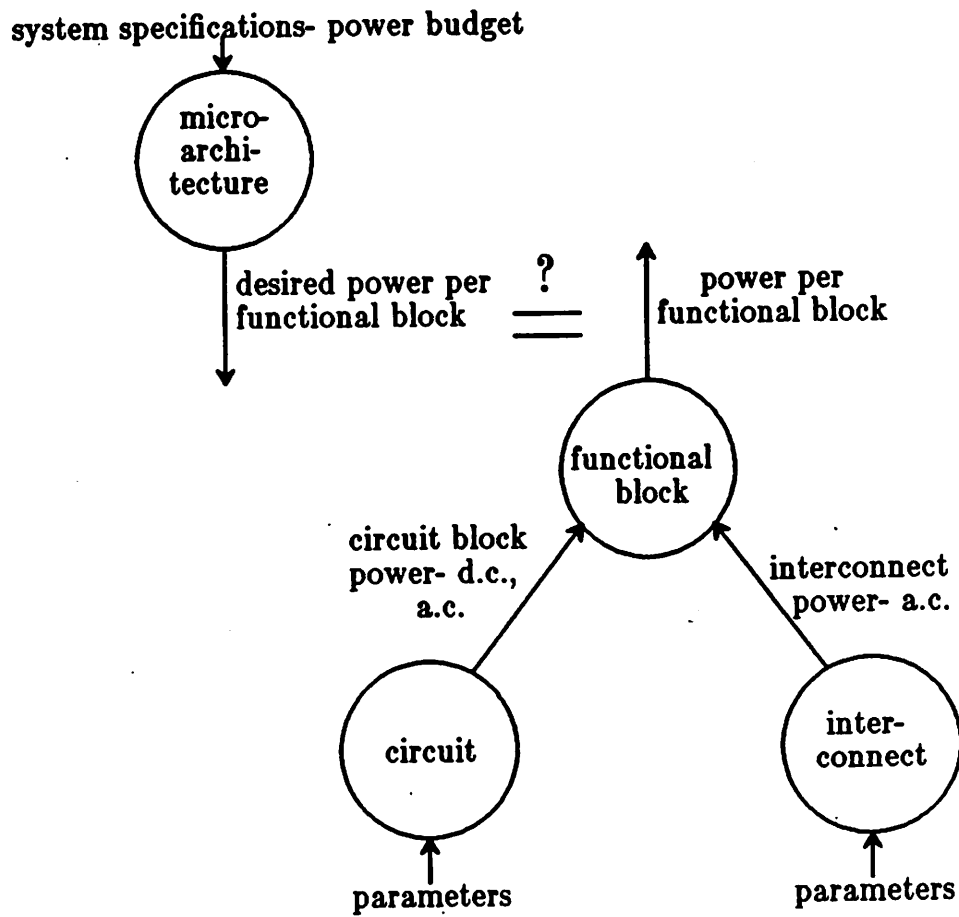


Figure 4.13- Analysis- Power

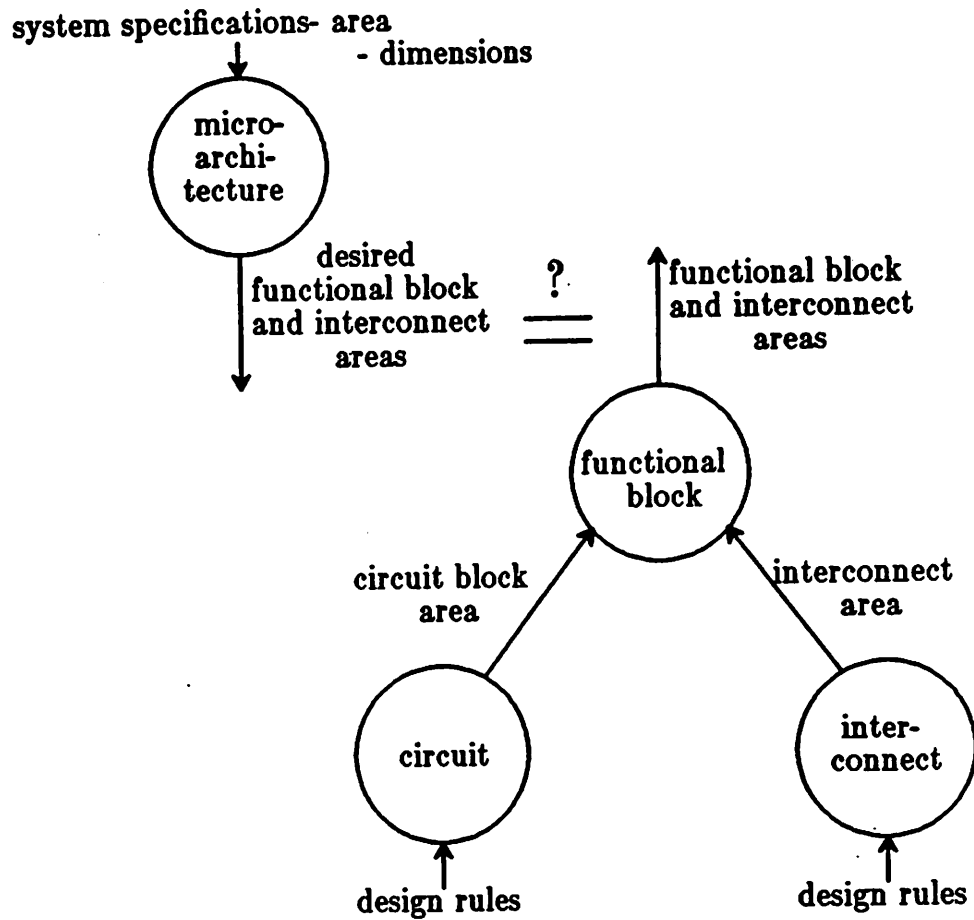


Figure 4.14- Analysis- Area

#### 4. Optimization

Optimization consists of taking a proposed solution to a design problem and improving one or more of its characteristics. In order to optimize a design, a proposed solution must first exist. Some amount of synthesis must be completed. All or part of the proposed solution is then analyzed in terms of any characteristics of interest. This analysis shows where the design stands for each of the characteristics and reveals areas of improvement. With this information further synthesis can be done to improve upon the original design. Another round of analysis is done after this synthesis. This may be followed by even more

synthesis if the results were not satisfactory. This series of analysis and synthesis phases is done until the design is considered satisfactory.

Each optimization step spans three phases (Figure 4.15). First an improvement is suggested through analysis. This improvement is then realized with a synthesis phase. The optimization is then completed with an analysis phase to evaluate the change.

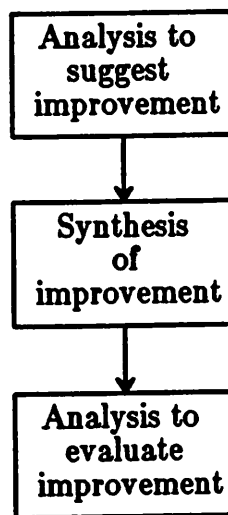


Figure 4.15- Optimization

Each analysis phase serves two purposes. First, it checks the results of the previous synthesis phase. This includes verification for proper functioning and evaluation of the characteristics of interest. Secondly, it reveals bottlenecks to be eliminated and improvements that may be done during the subsequent synthesis phase. Consequently, each analysis may be part of two optimization steps and optimization steps may overlap (Figure 4.16).

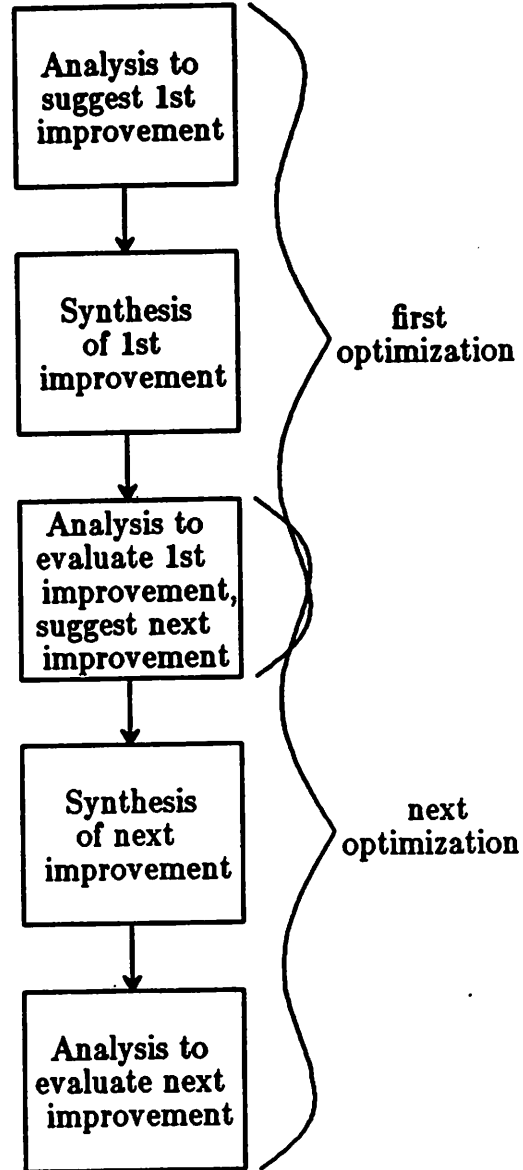


Figure 4.16- Sequential Optimizations

## 5. Methodology

Many design methodologies that are based on various sets of premises and priorities, exist for VLSI processors. Some treat the full design problem and others are restricted to sections of it. They all have some goals in common though. A correct design at a minimal expense are goals of all methodologies. In

terms of the four key issues of VLSI design (Chapter 1), this goal addresses the issues of correctness and time investment. However, after this the goals and premises of different methodologies diverge. The design methodology presented here is based on the following four premises:

1. Include all three design phases – preliminary, synthesis, and analysis.
2. Complete low level analysis is most time consuming.
3. Redo as little as possible – catch major mistakes early.
4. Accuracy of analysis depends on the accuracy of input data.

The first premise is that all three design phases should be included in the methodology. Each design phase considers some but not all of the interrelationships between the design levels. As discussed in Chapter 3, the many level interrelationships and tradeoffs of a VLSI design problem are distributed among the three phases – preliminary, synthesis, and analysis – so that the problem is more manageable. All three phases must be considered at least once during design so that no level interrelationships are overlooked. The synthesis and analysis phases are usually considered more than once so that optimal tradeoffs can be made. In this way the key issue of optimization is addressed.

The lower levels of a design contain more detail. The higher levels – microarchitecture and functional block – can be represented by things such as block diagrams. At the lower levels designs are represented by transistor level schematics or the physical layout. More detail implies that there is more data to be used in a complete analysis and therefore analysis will be more time consuming. For example speed simulation of the critical paths of a processor takes much longer using a low level simulator such as SPICE, than it would take using a simulator that assumes speeds for the functional blocks and then calculates critical paths from a block diagram.

VLSI processor design is inherently iterative. To get an optimal design, portions of a design may have to be done more than once. Therefore, some redesign time must be allotted. However, it is desirable to keep this to a minimum without sacrificing design quality. Therefore, mistakes that affect large portions of the design should be caught before large amounts of low level design are completed. Low level design, like low level analysis, involves much detail and can therefore be time consuming. If mistakes are caught early less time has been invested in the poor design and less effort is wasted in correcting the problem, than if they were caught late

The last premise is that the accuracy of predictions depends on the accuracy of the input data. If input data for analysis at any level is inaccurate, it can hardly be expected that the results will be reliable. This premise assumes that the method of analysis is dependable. If the method has limitations also, the analysis will be even less reliable. The implication of this is that the designers should recognize the accuracy of input data and analysis methods, and emphasize the results accordingly. For example, if it is known that the data or method gives results with a random 20% error, it is probably not advisable to make changes that would improve a section of the design by 10%.

The premise that major mistakes should be caught early along with the top down nature of synthesis suggests that portions of the analysis phase should be intermingled with portions of the synthesis phase. Synthesis begins at the highest level – microarchitecture. Analysis of proposed microarchitectures should be done before much design at lower levels is completed. In this way detailed design of poor solutions is avoided. However, without a complete low level design, assumptions must be made concerning the inputs when doing analysis at high levels. This means that the analysis will be less accurate and should be used accordingly when changing the design – premise 4. In this way it is possible to

catch major mistakes early but difficult to foresee the minor problems. But this is acceptable since the minor problems usually require less redesign than the major errors.

This top down order of synthesis and analysis means that higher design levels will be analyzed more times than the lower levels. As design progresses towards the lower levels, the inputs to higher level analysis can become increasingly accurate. High level analysis should be redone as inputs become more reliable to check that the high level design is still acceptable. Thus, high level analysis may be redone many times and the more time consuming low level analysis will be redone fewer times. This is desirable according to premise 2.

This also means that it may never be necessary to do a complete low level analysis of the entire processor. If analysis at each level includes all inputs as shown in the analysis flow diagrams, and yields all flow diagram outputs for that level, the complete analysis phase can be broken into analysis at each level. Figure 4.17 shows the individual steps that speed analysis may be broken into. The ordering of the steps is suggested by the unidirectional nature of analysis. As previously discussed, analysis contains two unidirectional paths. Both may be analyzed simultaneously and then the results are compared. The path involving the circuit, interconnect, and functional block levels is basically bottom up. This determines the analysis order for these steps, with the lower levels being analyzed first. The top down path only involves one level – the microarchitecture. In this way the complete low level analysis may possibly be avoided.

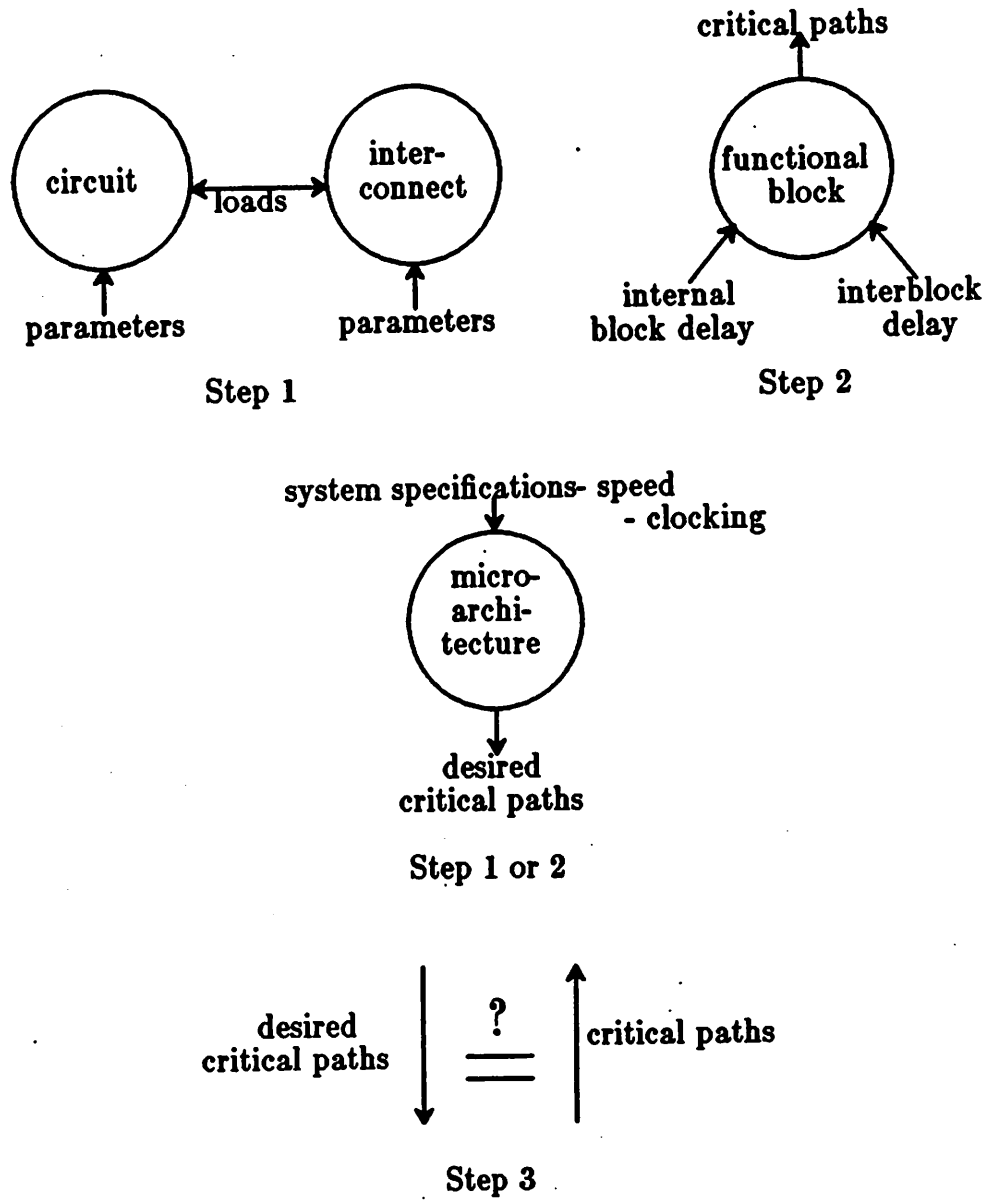


Figure 4.17- Steps of Speed Analysis

A methodology based on these premises is shown pictorially in the flow chart of Figure 4.18. Design is started with the preliminary phase as described earlier in this chapter. During the preliminary phase circuit and interconnect options are explored. Promising schemes for these two levels are identified so that microarchitecture design may be based on these possibilities.



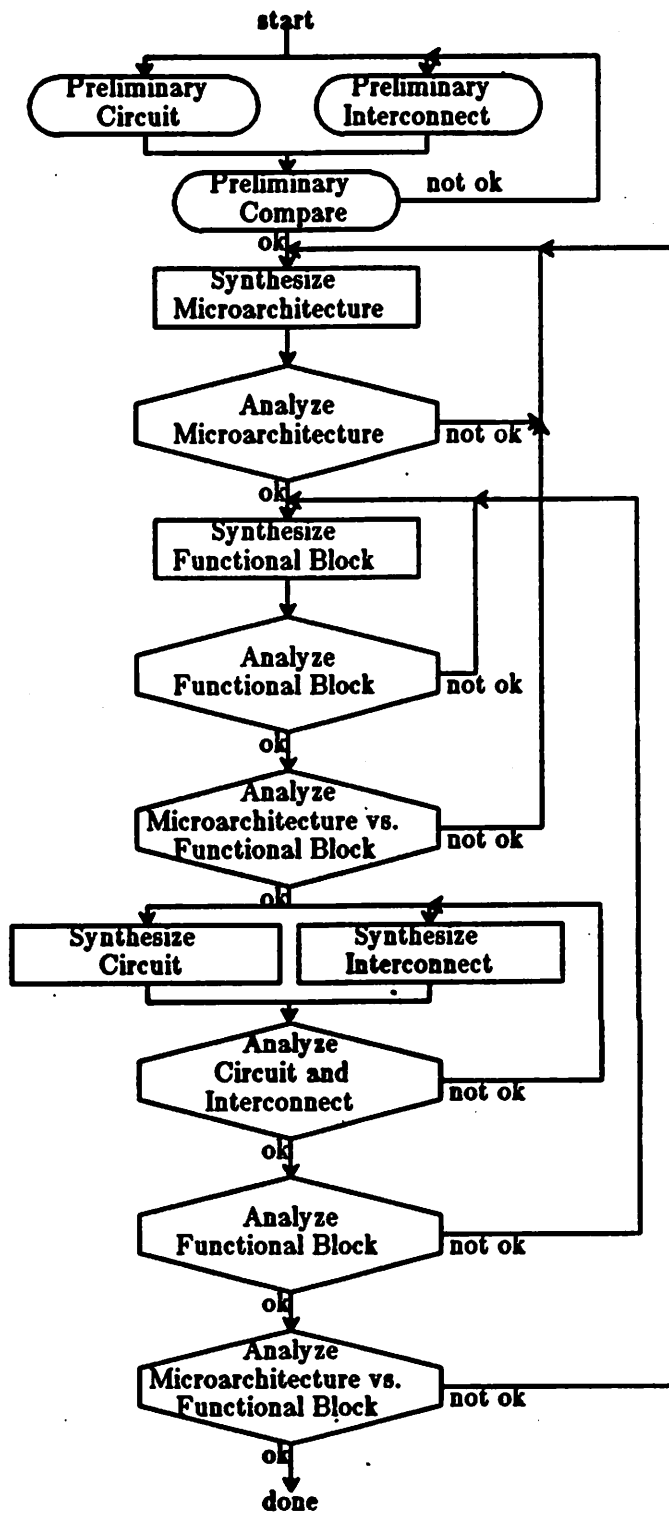


Figure 4.18- Methodology Flow Chart

Following the preliminary phase are alternating synthesis and analysis phases. Synthesis proceeds in a top down manner. Following synthesis at any given level is analysis and optimization of that level alone. When results at the single level are acceptable analysis and optimization move upwards one level at a time until all levels synthesized so far have been included. When the results of this analysis are acceptable, synthesis moves down to the next level and the process repeats itself.

Synthesis begins at the microarchitecture level. Microarchitecture analysis for all characteristics of interest follows synthesis at this level. Optimization is done until analysis of all characteristics is acceptable.

Synthesis then moves down to the next level – functional block. Again analysis and optimization are done at the functional block level for all characteristics of interest. When the results are satisfactory analysis is done including the microarchitecture level also. This consists of comparing the desired value for a given characteristic from microarchitecture analysis with the predicted value from functional block analysis. Design moves back to microarchitecture synthesis if the results of this comparison are not acceptable.

When the microarchitecture and functional block design are acceptable, synthesis moves to the circuit and interconnect levels. Again analysis and optimization are done within these levels until an acceptable solution is reached for all characteristics. Analysis then moves up a level to include the functional block level and finally the comparison with the microarchitecture level.

Each step in the flowchart has a corresponding flow diagram (Figure 4.19). Loops in the flow diagrams for each of the three phases are included in a single step. Loops signify iteration. It was possible to split many loops in the overall flow diagram between the natural phases. However, it is not obvious how to split up the remaining loops, so each loop is always included within one step. All

characteristics of interest are included within each analysis step. By considering all characteristics simultaneously the tradeoffs for optimality and the system priorities are emphasized.

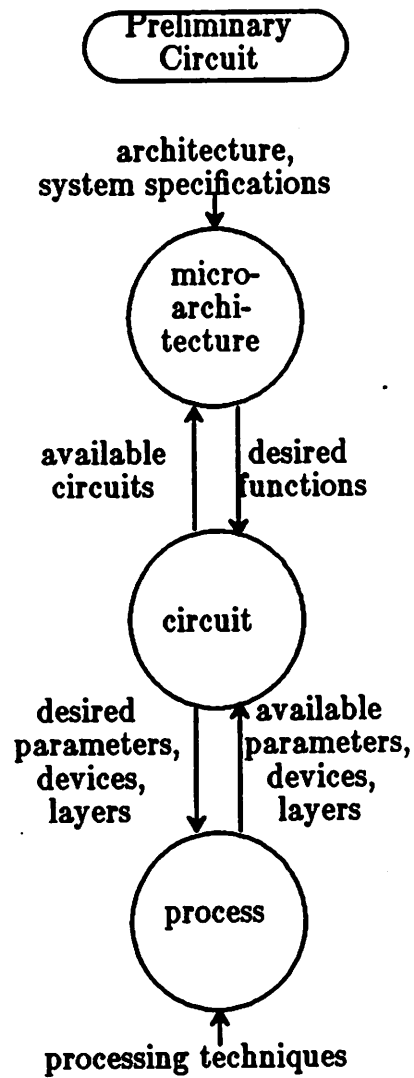


Figure 4.19a- Preliminary Circuit Step

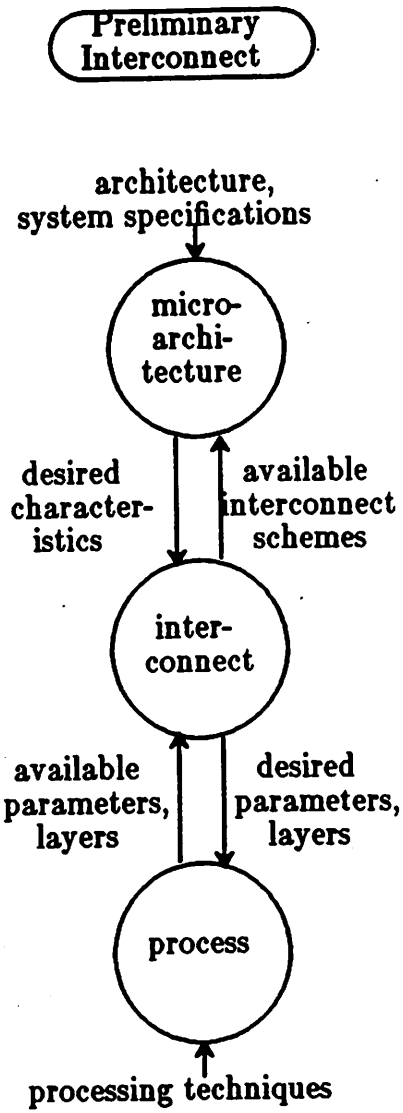


Figure 4.19b- Preliminary Interconnect Step



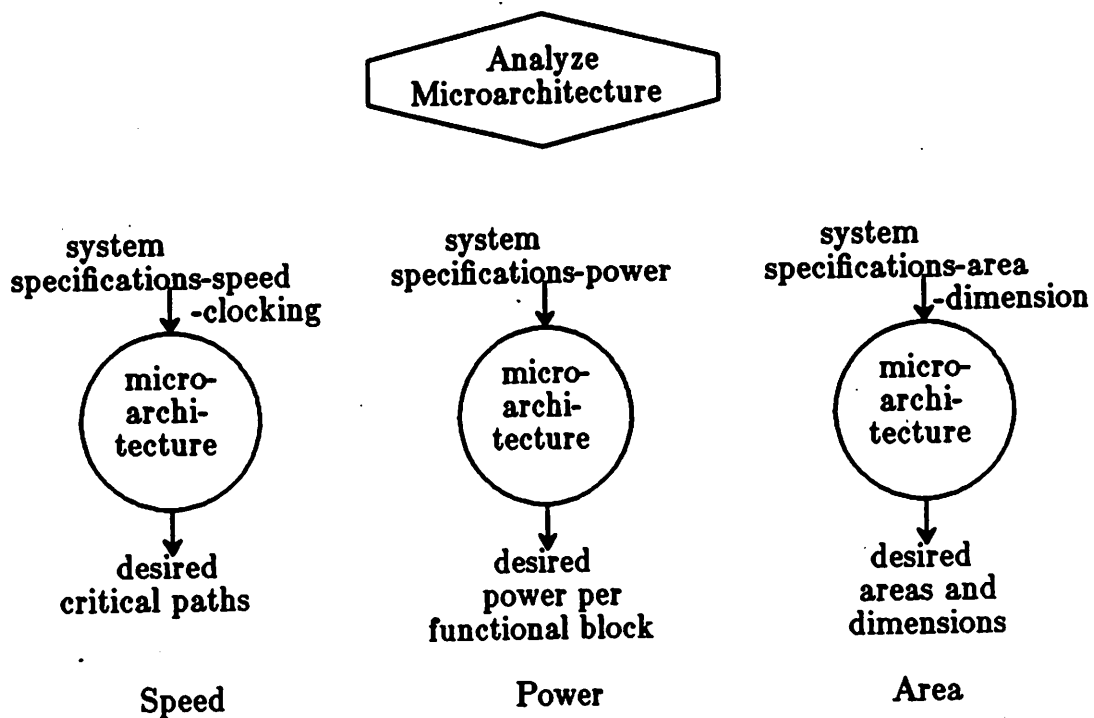


Figure 4.19e- Analyze Microarchitecture Step

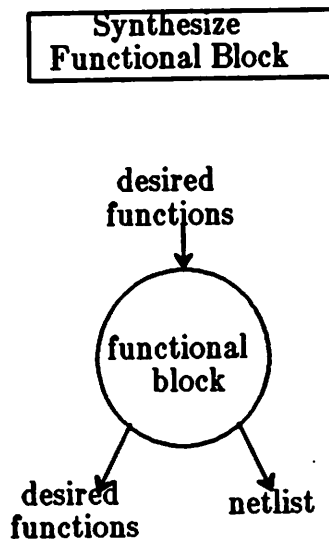


Figure 4.19f- Synthesize Functional Block Step

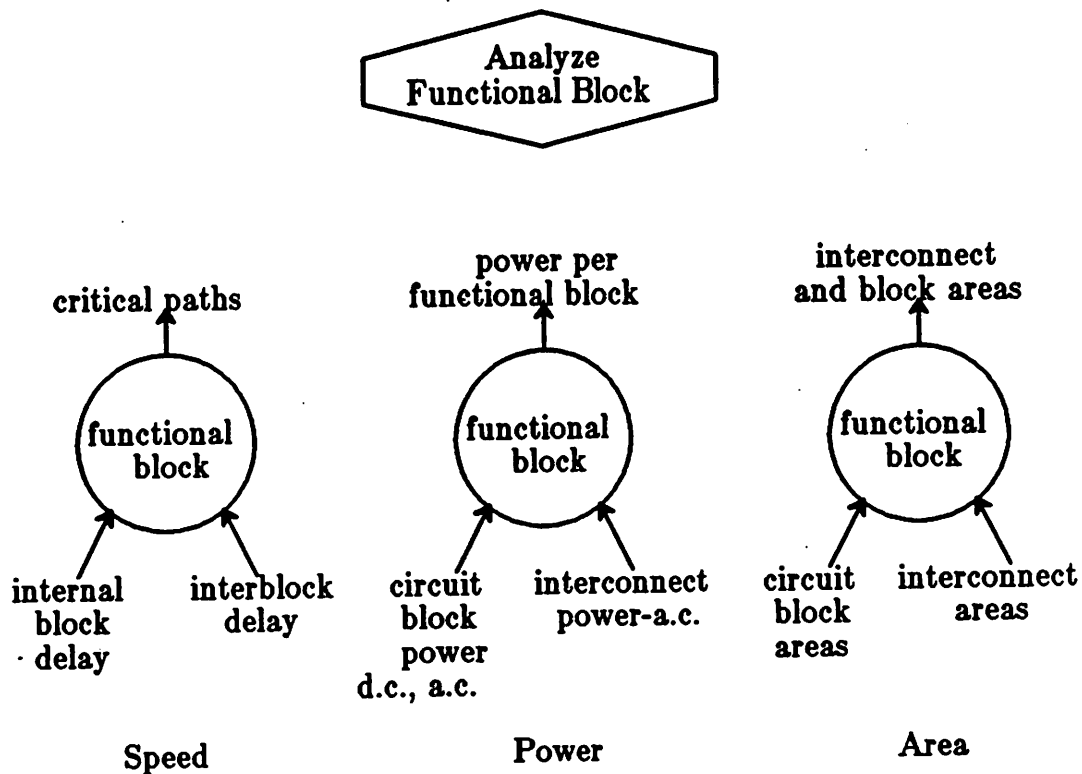


Figure 4.19g- Analyze Functional Block Step

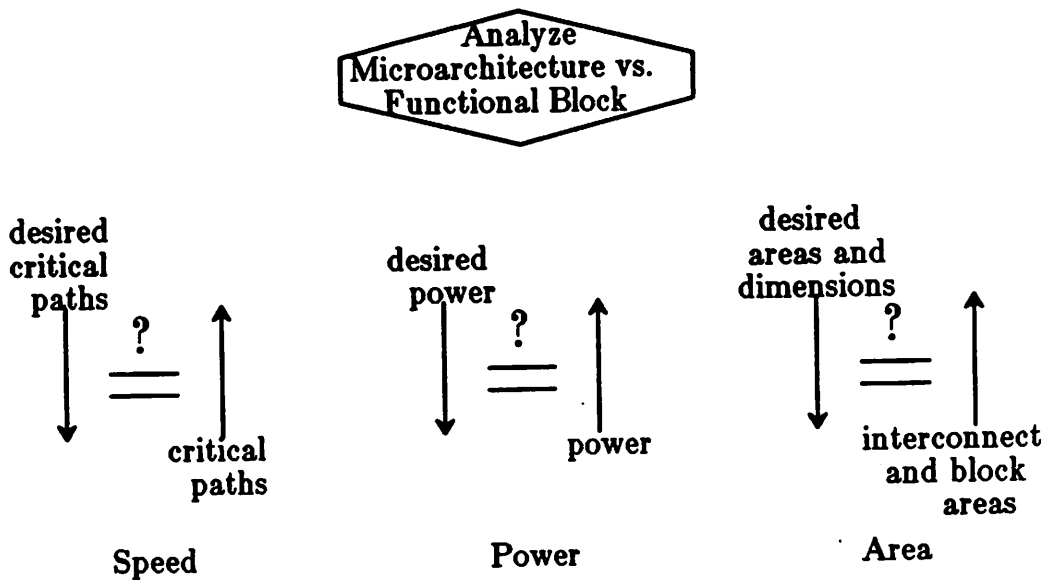


Figure 4.19h- Analyze Microarchitecture vs. Functional Block

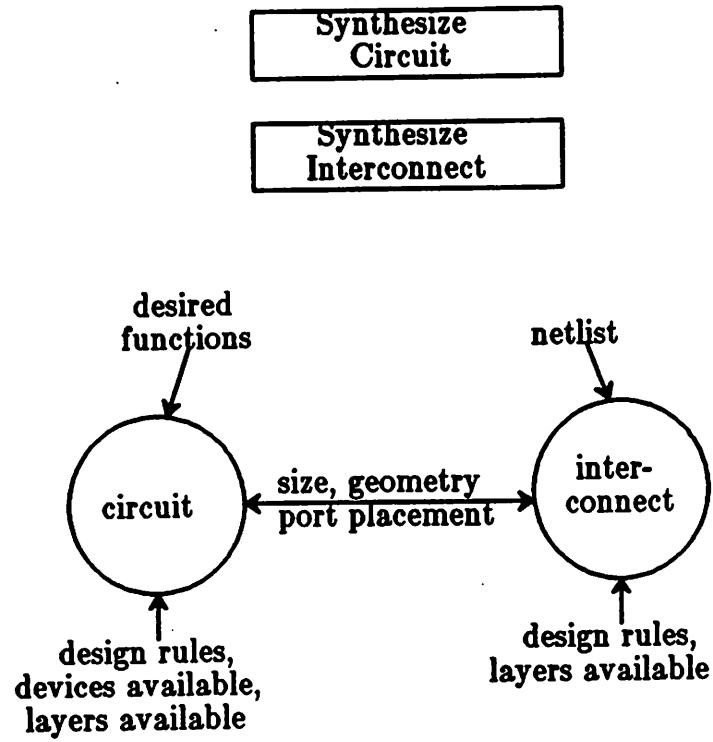


Figure 4.19i- Synthesize Circuits and Interconnects



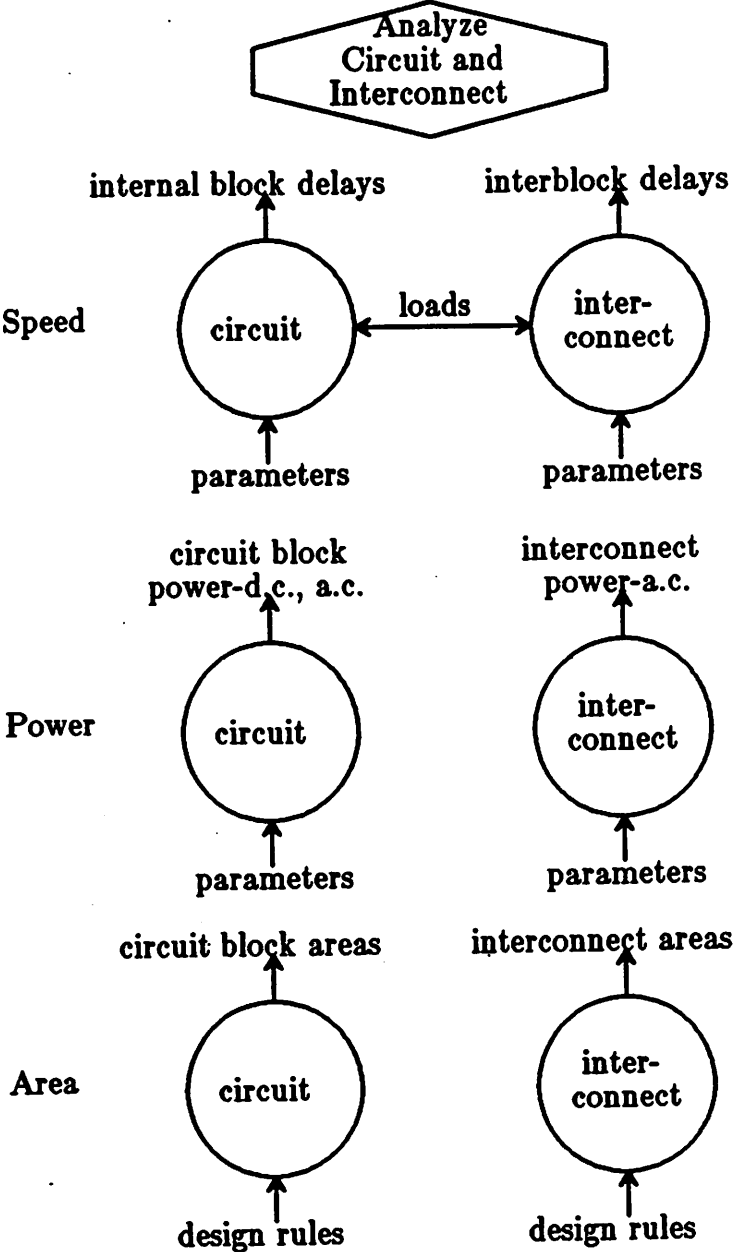


Figure 4.19j- Analyze Circuits and Interconnects

## Chapter 5

### External Inputs

### SOAR Case Study

External inputs are factors determined outside the realm of processor design, that must be considered when designing a VLSI processor. Chip design has little influence on these inputs, yet must meet their requirements and restrictions. For a fully custom VLSI processor, external inputs to the design are the *processor architecture, system specifications*, and available *processing technology*. SOAR was somewhat restricted from this. Fabrication was to be done by MOSIS. This meant that SOAR had to use whatever processes were available to MOSIS. Design of SOAR had no influence on the process. Thus, the outputs of the process level – *parameters, design rules, available devices, and layers* – all became external inputs. *Processor architecture* and *system specifications* were the other external inputs to be considered.

#### 1. Architecture

As previously mentioned, the architecture of a processor includes such things as data types, word size, addressing modes, register organization, the instruction set, and internally generated exceptions and traps. SOAR's goal was the efficient execution of Smalltalk, but without ignoring more general purpose languages such as C [Unga84]. The architecture of SOAR is a RISC architecture with internal opcodes and special features for the efficient execution of Smalltalk.

##### 1.1. Data Types

Data can be either tagged or untagged (Figure 5.1) [Blau83b], [Samp85]. Tagged data is useful for a Smalltalk processor while untagged data is best for a C processor.

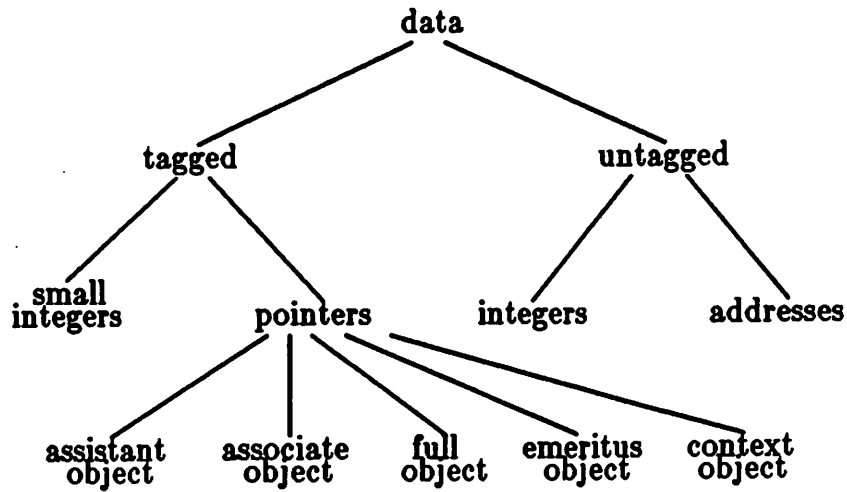


Figure 5.1- SOAR Data Types

Smalltalk is an object oriented language. The two main tagged data types are small integer objects and pointers to objects (Figure 5.1). Tag bits distinguish these data types (Table 5.1).

objects	tag
small integers	0
object oriented pointers	tag
assistant object	1000
associate object	1001
full object	1010
emeritus object	1011
context object	1111

Table 5.1- Tagged Data Types

Small integers are one type of tagged data. They have one tag bit and may range from  $-2^{31}$  to  $2^{31}-1$ .

The other types of tagged data are the pointers. Pointers are used to reference objects. Smalltalk has objects of several types and ages. These types and ages are reflected in the pointers to the objects. Pointer types are distinguished by their tags (Table 5.1). Assistant, associate, full, and emeritus objects are objects of various ages. These ages are significant to the storage reclamation inherent in Smalltalk [Unga84]. The last type of pointer is the pointer to a context object. Context objects are treated differently from other objects and this is the reason for their distinct pointer type.

Untagged data can also be of two main types (Figure 5.2). The first type is the integer. These can have values from  $-2^{31}$  to  $2^{31}-1$ . Addresses are the second major type. Potentially a 32 bit address space could be supported by untagged addresses. However, the system only calls for a 28 bit address space [Blau83a], [Blom83], [Brow85]. Thus, the extra four bits are meaningless for untagged addresses.

## 1.2. Word Size

All words on SOAR are 32 bits (Figure 5.2) [Samp85]. SOAR references 28 bits of virtual address space. Thus, only 28 bits of a 32 bit address word actually address memory. The other four bits are meaningless for addresses of instructions. Tagged data addresses are pointers and the extra four bits hold the tag that indicates the type of object being pointed to [Blau83b]. During untagged operation the extra four data address bits are meaningless.

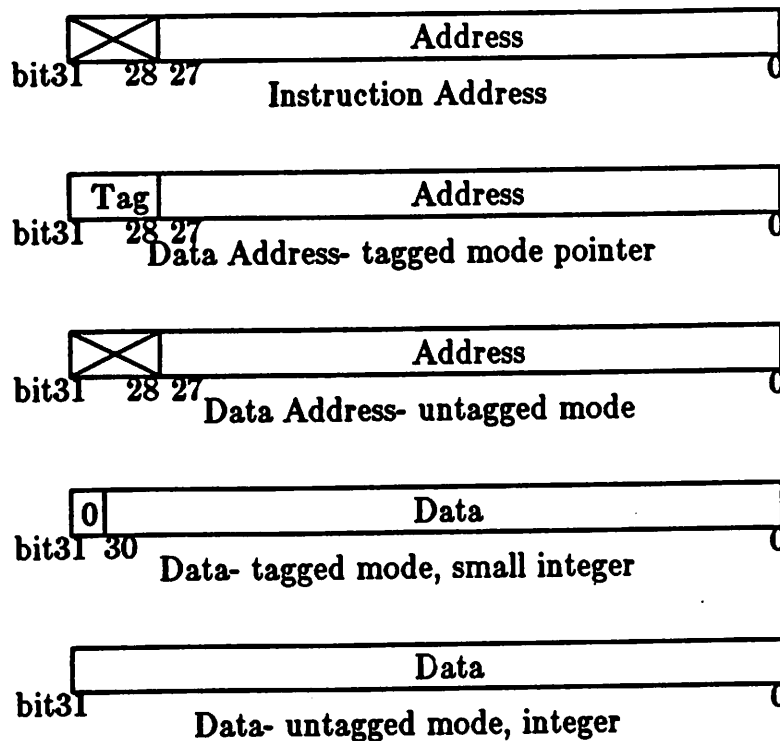


Figure 5.2- SOAR Words

Data words are all 32 bits also. Tagged data contains 31 bits of data and one tag bit indicating it is a small integer object. All 32 bits are data bits when operating in untagged mode.

### 1.3. Addressing Modes

SOAR is a register based processor. All operations use either register or immediate operands. Memory is accessed through loads and stores only. SOAR supports three types of addressing modes – absolute, relative, and indexed (Table 5.2). The address for absolute loads and stores is simply an immediate constant supplied by the instruction. Relative addresses are the value of the program counter offset by a constant. Indexed loads and stores use a value stored in a register and an offset to calculate the address. All offsets for stores originate in an immediate constant. For loads the offset may come from an immediate constant

or from a register.

Address mode	Address calculation
Absolute	0 + offset
Relative	program counter + offset
Indexed	register + offset

Table 5.2- Addressing Modes

#### 1.4. Register Organization

Any process running on SOAR has direct access to 32 registers, r0 through r31 (Table 5.3) [Samp85]. These 32 registers are divided into four groups of eight registers each, according to their function. These four groups are the globals, special registers, highs, and lows. The globals and specials are common to all subroutines. In other words, physically they are the same registers, independent of the invoked subroutine. The highs and lows are local registers for designated subroutines. Thus, the register that is r7 for one subroutine is not necessarily the same register physically, as r7 of another subroutine.

Register Type	Register	Contents
Global	r31	Scratch
	r30	Scratch
	r29	Scratch
	r28	Scratch
	r27	Scratch
	r26	Scratch
	r25	Scratch
	r24	Scratch
Special	r23	PSW
	r22	CWP
	r21	TB
	r20	SWP
	r19	SHA
	r18	SHB
	r17	PC
	r16	RZERO
High	r15	return address for this subroutine
	r14	return value
	r13	local
	r12	local
	r11	local
	r10	local
	r9	local
	r8	local
Low	r7	return address of traps, called subroutine
	r6	
	r5	
	r4	
	r3	
	r2	
	r1	
	r0	

Table 5.3- SOAR Registers

A key feature of the SOAR architecture is the register window scheme (Figure 5.3) [Kate83]. Local memory is split into eight banks of registers, each containing eight registers [Blak83]. At any time, the current process has access to two of these banks. One bank is designated as the high bank and gets accessed whenever r8 to r15 is referenced. The other bank is the low bank. Reference to r0 through r7 causes the low bank to be accessed. Together these two banks make up the register window for the current subroutine. When a call is executed, the current lows become the highs for the called subroutine and another bank becomes the new lows. In this way a new window is formed for the called routine. When doing a return the current highs become the future lows. The future highs are the highs that belong with these future lows. Thus, a return causes a previously defined window to become visible again. The sharing of highs and lows between subroutines leads to the term *overlapped register windows*.



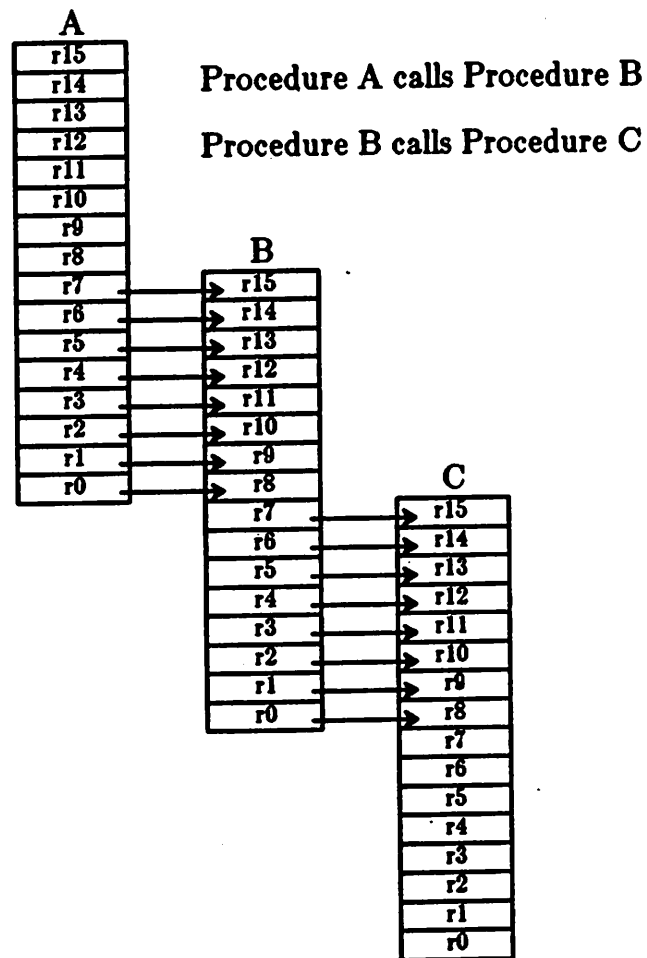


Figure 5.3- Register Window Scheme

A call occurring when all windows have been filled causes the saving of the oldest bank in memory. The area where it is saved is determined by two special registers, the saved window pointer (SWP) and current window pointer (CWP) as shown in Figure 5.4. Thus, every local register in every window has a corresponding memory address. Local registers from previous subroutines that are inaccessible by referencing r0 to r15, can be accessed by a load or store using their memory address.

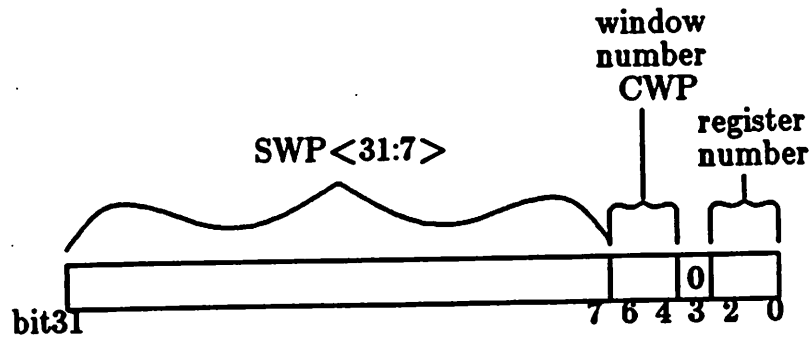


Figure 5.4- Memory Address of a Saved Register

The global registers, r24 to r31, are used for data storage. Their contents only affect the processor as operands. They can be used to hold such things as temporary results and global data, common to all subroutines.

The special registers, r16 through r23, all have functions other than just serving as operands. They are not for data storage, but determine the processor's functioning as described by Table 5.4. Processor state can be externally controlled by storing to these registers or examined by reading their contents.

Register	Name	Function
r23	PSW	process status word, shadow copy of destination, opcode
r22	CWP	current window pointer- points to one of eight windows
r21	TB	trap base register- used when forming trap vectors
r20	SWP	saved window pointer- points to last window saved
r19	SHA	shadow copy of A input to ALU, byte inserter/extractor
r18	SHB	shadow copy of B input to ALU, byte inserter/extractor
r17	PC	program counter
r16	RZERO	always zero

Table 5.4- Special Registers

### 1.5. Instruction Set

SOAR is based on a RISC architecture [Unga84]. It supports fewer types of instructions than would be found in a complex instruction set. Table 5.5 summarizes the instruction set [Samp85]. With a few exceptions an instruction may be started every clock cycle. In this way each instruction is similar to a microinstruction in a microcoded machine.

Most instructions have two modes of operation - untagged and tagged. Untagged instructions treat operands as if the tag bits are really part of the data. Tagged instructions examine the tag bits and complete according to whether or not the bits are correct for the operation. This is one of the features added for

Table 5.5- SOAR Operations

Instruction	I < 31:30 >	Opodes (octal)	Operations required
[%]CALL	00	00-37	addition decrement CWP
[%]JUMP	00	40-77	addition
[%]RET	01	10-17	addition
[%]SKIP	01	20	subtraction
[%]TRAP!	01	21-27	subtraction
[%]XOR	01	44	exclusive or
[%]AND	01	46	and
[%]OR	01	47	or
[%]ADD	01	50	addition
[%]SUB	01	52	subtraction
[%]SRL	01	40	logical right shift- 1 bit
[%]SRA	01	42	arithmetic right shift- 1 bit
[%]SL	01	51	addition
[%]INSERT	01	56	insert, zeroing
[%]EXTRACT	01	54	extract, zeroing
[%]LOAD	01	34	addition
[%]LOADC	01	35	addition
[%]LOADM	01	36	subtraction
[%]STORE	01	30	addition
[%]STOREM	01	32	subtraction
			decrement

the efficient execution of Smalltalk. The % in Table 5.5 indicates untagged operation. [%] indicates both modes of operation are possible. Instructions only operating in tagged mode are preceded by []].

Jumps and calls are distinguished from other instructions by bit 30 of the instruction (Figure 5.5). Bit 28 then further distinguishes jumps from calls. The remaining 28 bits of a jump or call instruction contain the absolute address of the target. This address is incremented and loaded into the program counter. In addition to this calls cause the register window to change, by decrementing the CWP, and the return address to be saved in the future r15 (current r7).

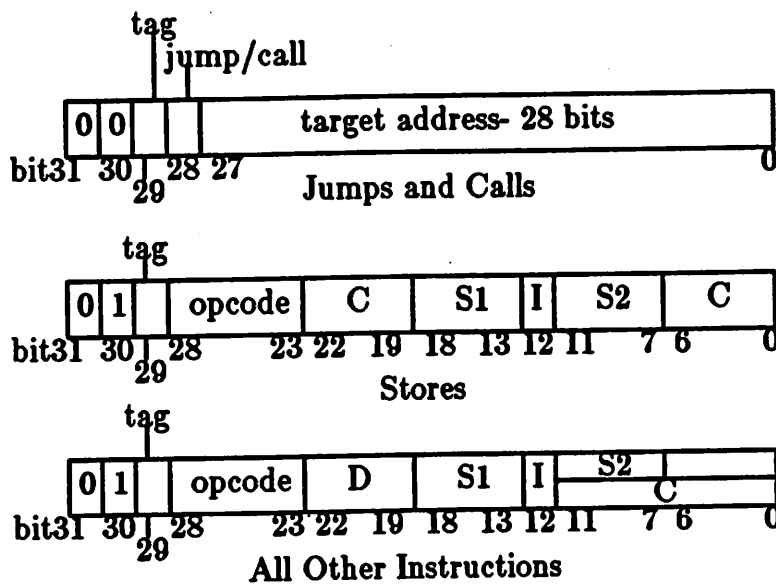


Figure 5.5- Instruction Formats

The remaining 29 bits of instructions other than jumps and calls, are split into fields as shown in Table 5.6 and Figure 5.5. The locations of these fields are the same for all types of instructions except stores (Figure 5.5). The operands are specified by the S1, S2, and C fields. The S1 operand is always found in a register. The second operand may be a register operand, S2, or a sign extended

constant, C, as indicated by the immediate bit. Stores require all three operands—S1, S2, and C. The D field specifies the destination register of the result. The opcode field indicates what operation is to be performed on the operands. When sign extension is performed on the immediate constant the four most significant bits become the tag of the constant and the fifth most significant bit is the sign bit that is duplicated (Figure 5.6).

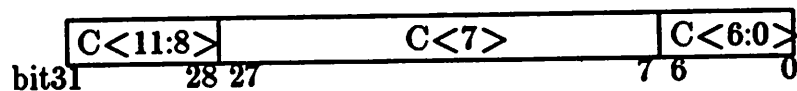


Figure 5.6- Sign Extension of the Immediate Field

Field	No. of bits	Specifies
Opcode	6	operation
D	5	destination register
S1	5	first operand register
I	1	immediate or register for the second operand
S2	5	second operand register
C	12	second operand immediate constant

Table 5.6- Instruction Fields

There are eight types of returns as shown in Table 5.5. The eight types of returns are formed by enabling any combination of three mechanisms— CWP increment, interrupt enabling, and register nilling— upon a return. When a call is performed the CWP is always decremented to attain a new register window.

Incrementing the CWP on a return will recover the old window, the window of the calling routine. Hardware interrupts may be optionally enabled upon a return if they were previously disabled. Registers r0 through r5 of the new window may be optionally set to the nil value, B0000000, upon a return—register nilling. Registers r6 and r7 contain the return value and saved address, respectively, and therefore can not be nilled. The return address is calculated by adding the two operands and placing the result in the program counter for all types of returns. Typically, the saved address—r7—is used for this calculation. The destination field is irrelevant for returns (but must be zero due to idiosyncracies in the microarchitecture).

Skip and trap instructions compare the two operands and then take a skip or trap if the result of the comparison is true. Skips and traps have no destination register so the destination field is used to specify the type of comparison to be done (Table 5.7). The skip instruction causes the instruction immediately following the skip to be skipped if the comparison is true. The trap instructions cause execution to jump to an address specified by the trap vector if the condition is true. There are seven trap instructions, each corresponding to a different trap vector. Other than this difference all seven types of trap instructions have the same function.

Mnemonic	D field (octal)	Condition
EQ, NE	04, 05	equal, not equal
LT, GE	02, 03	less than, greater than or equal
LE, GT	06, 07	less than or equal, greater than
LTU, GEU	12, 13	unsigned less than, greater than or equal
LEU, GTU	16, 17	unsigned less than or equal, greater than
NEVER, ALWAYS	00, 01	never, always
IN0, OUT0	12, 13	$0 < \text{1st operand} < \text{2nd operand}$
IN1, OUT1	22, 23	$1 < \text{1st operand} < \text{2nd operand}$

Table 5.7- Skip and Trap Instruction Condition Codes

All ALU operations, shift operations, and byte operations use the two operands, perform the designated operation on them, and store the result in the register specified by the destination field. Arithmetic operations are ADD and SUBTRACT. Logical operations supported by SOAR are AND, OR, and XOR. SOAR performs three types of one bit shifts – left arithmetic shifts, right arithmetic shifts, and right logical shifts. Byte operations performed by SOAR are EXTRACT and INSERT (Figure 5.7). Extract puts a specified byte of the first operand into the least significant byte of the result. All other bytes of the result are zeroed. Insert takes the least significant byte of the first operand and puts it into a specified byte of the result, zeroing all other bytes of the result. For both insert and extract the specified byte is determined by the two least significant bits of the second operand.



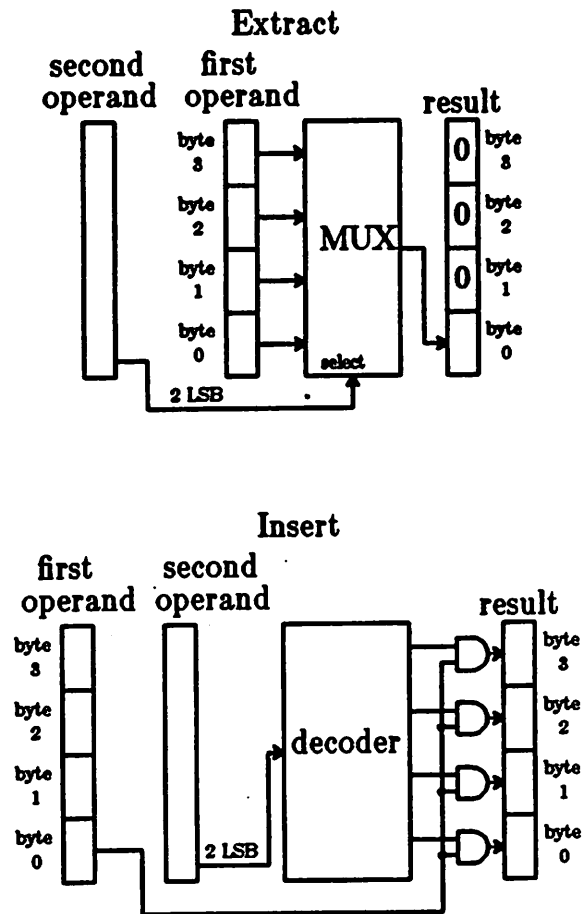


Figure 5.7- Byte Operations

The load and loadc instructions add the two operands to form an address for a memory fetch. The word fetched from memory is stored in the register specified by the destination field. Load multiple performs a series of up to eight loads. Destination registers for these loads are r0 through rn where n is specified by the destination field and must be less than eight. Addresses for these loads are evenly spaced in memory. This spacing is specified by the second operand. Load multiple can be described algorithmically:

```

x ← 1st operand
y ← 2nd operand
z ← destination field
Repeat
x ← x-y
r[z] ← M[x]
z ← z-1
until z < 0

```

Stores use the first operand and immediate constant, C, to form an address. The second operand is the data to be stored. Store multiple is the store counterpart to load multiple. Algorithmically it can be described:

```

x ← 1st operand
y ← immediate constant
z ← 2nd operand field
Repeat
x ← x-y
M[x] ← r[z]
z ← z-1
until z < 0

```

### 1.6. Internal Exceptions and Traps

A variety of situations can arise in SOAR that must be handled immediately. When such a trap situation arises, execution must switch to an appropriate software routine, trap handler, in order to handle the problem [Blau83b], [Unga84], [Samp85].

As previously mentioned traps can be caused by trap instructions. Illegal opcodes are recognized and cause traps to the appropriate handlers. This prevents hardware from trying to make sense of an illegal opcode and permits the instruction set to be extended by having the appropriate code sequence for a new opcode, in a trap handler.

Due to the register organization, registers must be written to memory if a call occurs and all register windows are filled. This is accomplished by a window overflow trap. As returns occur, old register windows are recovered. The oldest windows may have been written to memory and therefore must be fetched from memory when a return tries to recover them. A window underflow trap does this.

Instructions executing in tagged mode examine the operands' tags and cause traps if the tags are not correct. These traps were designed specifically for the efficient execution of Smalltalk. Tag traps can occur for loads, stores, arithmetic, logical, and shift operations according to Tables 5.8 and 5.9. Immediate operands are always assumed to be integers. Tagged ALU and shift instructions can only operate on two integers (Table 5.8).

1st operand tag	2nd operand tag	2nd operand	result
integer	integer	immediate	no trap
integer	integer	register	no trap
integer	pointer	immediate	no trap
integer	pointer	register	TAG TRAP
pointer	integer	immediate	TAG TRAP
pointer	integer	register	TAG TRAP
pointer	pointer	immediate	TAG TRAP
pointer	pointer	register	TAG TRAP

Table 5.8- ALU and Byte Operation Tag Traps

Tagged loads and stores need one of the operands used in the address calculation, to be a pointer and the other address calculation operand to be an integer (Table

5.9).

1st operand tag	2nd operand tag	2nd operand	result
integer	integer	immediate	TAG TRAP
integer	integer	register	TAG TRAP
integer	pointer	immediate	TAG TRAP
integer	pointer	register	no trap
pointer	integer	immediate	no trap
pointer	integer	register	no trap
pointer	pointer	immediate	no trap
pointer	pointer	register	TAG TRAP

Table 5.9- Load and Store Operation Tag Traps

Stores and returns can cause generation scavenging traps. Smalltalk uses these to reclaim storage space. Tagged returns require the first operand to be a pointer. Otherwise a trap occurs. Stores cause generation scavenging traps if the data being stored is a pointer to a context object or is a pointer to a younger object than the object where it is being stored.

The last type of trap designed specifically for Smalltalk is the software interrupt trap. A bit in the process status word indicates whether or not software interrupts may be taken. If a tagged call or jump occurs while software interrupts are enabled then a software interrupt trap occurs.

External situations may also cause traps. Data page faults, instruction page faults, and I/O interrupts are detected by SOAR and result in traps. All traps, both internal and external are prioritized. When two or more trap situations arise

simultaneously, the trap with the highest priority is handled first. If other traps still exist upon the return from this highest priority handler, they are then handled according to their priorities. Priorities for traps are shown in Table 5.10. Traps with the lowest *reason numbers* have the highest priorities.

Trap	Reason Number	Priority
Illegal opcode	0	Highest
Tag trap	1	
Software interrupt	2	
Window overflow	3	
Window underflow	4	
Data page fault	5	
Trap instruction	6	
Generation scavenging	7	
Instruction page fault	8	
I/O request	9	Lowest

Table 5.10- Trap Priorities

When a trap occurs SOAR automatically saves relevant state information. The operands being operated on, when the trap situation arises, are saved in shadow registers A and B, r19 and r18 respectively. The opcode being executed and the intended destination register are saved in the process status word. The value of the program counter is saved in r7. Hardware interrupts are automatically disabled when a trap occurs.

Program execution shifts to the beginning of a trap handler after a trap situation arises. The address of the start of the appropriate trap handler is formed by the concatenation of the trap base register (r21), the reason number, and the executing opcode (Figure 5.8).



Figure 5.8- Trap Handler Address

## 2. System Specifications

One of the goals of the Smalltalk project was to realize a working Smalltalk system without having to design a complete computer system. This is accomplished by using a SUN workstation to house a processor board built around the SOAR chip [Blom83], [Brow85]. Two separate processor boards were built. One board, *the cache board*, includes an 8Kbyte virtual address cache. The other board, *the Orion board*, directly accesses SUN memory. The SUN hardware is used to service all I/O devices – the disk controller, graphics interface, keyboard input, and mouse tracking. Extra memory boards are added to the SUN. The SUN's MC68010 processor and SOAR interrupt and communicate with each other via the custom designed SOAR processor boards. Thus, the hardware on these processor boards and the memory that SOAR has direct access to determined the system specifications for the SOAR processor design.

## 2.1. Memory Requirements

Performance studies indicated that the SOAR instruction set would require a virtual memory size of 20 to 40Mbytes [Blau83a]. A 64Mbyte virtual address space was chosen, requiring word addresses to be 24 bits wide. The architecture allows for expansion from this by providing 28 address bits. Physical memory in this system is 8Mbytes. On the *Orion board* a SOAR memory access requires translation of the virtual address using a single level page map, and then physical memory access through the Multibus [Blom83]. *The cache board* directly accesses the on board cache with the virtual address [Brow85]. A hit rate of greater than 90% is expected. It was predicted that 400ns. would be needed for either of these memory accesses. Thus, the system specification of a 400ns memory cycle was one of the external inputs to processor design.

Signals needed by the memory circuitry of the boards, include the 28 bit virtual address and a RD/WR\* signal. RD/WR\* indicates whether a read or write is being done at the given address. It is used to determine the board's state and to generate a write signal for memory. It must be valid early in the memory cycle and remain valid until the end of the cycle. Since much of the board design was done during chip design, it was not known exactly how early in the cycle this signal was needed. Thus, the external requirement for this signal was to have it available as soon as possible and hold it throughout the cycle.

## 2.2. Clocking

For board design simplicity, SOAR clock cycles and memory cycles are synchronous. One SOAR memory access can be done during each SOAR clock cycle. Therefore, the desired SOAR clock cycle was 400ns.

The basic SOAR cycle is split into underlapping clock phases. Hardware on both the processor and boards is clocked by these phases. A three phase clock

cycle was chosen in contrast to the four phase clock of RISC II [Kate83]. This was an attempt to avoid extra clocking overhead.

### 2.3. External Interrupts and Wait

Both boards signal interrupts to SOAR whenever a page fault or I/O request occurs [Blom83], [Brow85]. SOAR handles these interrupts using the same trapping mechanism as is used to handle internal trapping situations [Samp85]. SOAR distinguishes between data page faults and instruction page faults. These three external interrupts are assigned priorities and cause jumps to trap handlers, just as the internal traps do. Page faults and I/O requests are detected by the boards and the appropriate signal is asserted by the end of clock phase 2.

A variety of situations may arise that force SOAR to be put into a WAIT state. Cache misses, I/O requests, and direct accesses to main memory (*the cache board only*) all need WAIT states. During this WAIT the internal state of SOAR must not change. The boards supply a WAIT signal to SOAR that indicates a WAIT is necessary. The WAIT signal is asserted continuously during the WAIT. The boards detect a need for WAIT and the WAIT signal is asserted by the end of phase 2.

*The Orion board* requires SOAR to acknowledge a WAIT by asserting a wait acknowledge signal. Wait acknowledge is asserted when SOAR enters the wait state and remains asserted for the duration of the wait.

### 2.4. Fast Shuffle Control

In order for calls and jumps to be executed with minimal delay, the full absolute address of the target is one field of the call or jump instruction (Figure 5 [Samp85].5). Thus, the target address is immediately available to address memory without any computation. In theory, the processor could immediately load the



program counter with this address whenever a call or jump was detected. In practice however, the delay due to loading the program counter and driving the address pins would be too long. Therefore, a latch on the board- TARGET ADDRESS LATCH- captures the field that would contain the target address, on every incoming instruction- I/D asserted (Figure 5.9). Whenever a jump or call is detected on the processor- FSHCNTL asserted- this address is used to access memory, instead of the addresses coming from SOAR. This mechanism is known as a *Fast Shuffle* mechanism.

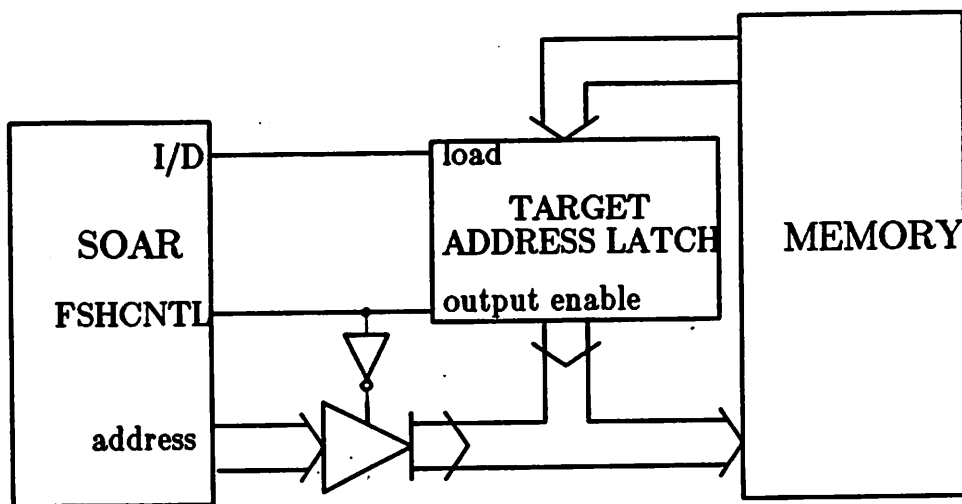


Figure 5.9- Fast Shuffle Mechanism

In order for this *Fast Shuffle* mechanism to work, SOAR must supply the board with two signals. The first signal is the I/D signal. I/D indicates whether the data coming into SOAR is an instruction or data. It is used to enable loading of the TARGET ADDRESS LATCH on the board. The TARGET ADDRESS LATCH must only be loaded on incoming instructions. Loading of this latch occurs in phase 3. Therefore, I/D must be valid by phase 3.

The second signal needed is FSHCNTL. This signal indicates that the off chip TARGET ADDRESS LATCH is to be used to access memory. The output of the TARGET ADDRESS LATCH is enabled according to this signal. This signal should be valid as early as possible in the SOAR cycle because a memory access takes the entire cycle.

## **2.5. Reset**

Both boards have the ability to reset SOAR by using the RESET input signal to SOAR [Blom83], [Brow85], [Samp85]. In this way SOAR is initialized to a known state. Resetting SOAR causes execution to begin at the designated reset address, FFFFFFF0, and interrupts to be disabled. SOAR executes NOPs during reset.

## **2.6. Loading Characteristics**

The two boards were designed using Schottky and standard TTL parts. Therefore, all outputs of SOAR are required to be able to drive at least one Schottky TTL load. This means that output drivers must be able to source  $50\mu\text{A}$  in the high state and sink 2mA in the low state. The output high and low levels must also correspond to TTL levels – greater than 2.7volts and less than .5volts respectively.

Output drivers are loaded down by package capacitances. They must drive the pin on the SOAR package, one other TTL package, and any board routing between the two packages at the required speed. This capacitance was assumed to be 20pF for simulations.

## 2.7. Size And Power

The size of the chip was restricted by processing equipment. The largest die size that MOSIS can handle is 7.8mm by approximately 10.5mm. If this restriction did not exist, the size would have been limited by the package cavity. SOAR fits into an 84 pin grid array package.

Another important restriction is the maximum power dissipation allowed. Excessive power dissipation can lead to device failure or can necessitate special cooling systems. The package rating of the 84 pin grid array limited the power dissipation to 2.5 watts.

## 3. Process

Process design provides the actual physical structures and their electrical characteristics, from which the processor will be made. It is at this level that much of the groundwork is laid for the final speed, area, and power of the processor. From the start it was known that SOAR would be fabricated through MOSIS, a DARPA sponsored fabrication service for universities. It solicits companies that are interested in fabricating university chips and characterizes their processes. This process data along with a schedule of process starts is available to universities. In order to have a chip fabricated, the project layout in CIF format, is sent to MOSIS along with required information about it. MOSIS then submits it to an appropriate company for fabrication. Therefore, in this situation all the outputs of the process design level – devices and layers available, parameters, and design rules – become external inputs for processor design.

MOSIS presently supports a variety of technologies – 4 micron NMOS, 3 micron NMOS, and 3 micron CMOS – with plans for smaller geometry CMOS processes in the future. Several suppliers service MOSIS for any one of these technologies. All lines target their parameters for similar values. Available from

MOSIS are target parameters for their technologies. MOSIS also supplies actual parameters for completed runs. Process parameter information for the SOAR design was taken from measurements of wafers from previous runs and target parameters. The measured parameters were more conservative and therefore were used for most simulations. Some simulations were done with the target parameters and then compared with the simulations using the measured parameters. A factor of slightly more than two was observed in the speed simulations.

### **3.1. Devices and Device Parameters**

SOAR was designed for MOSIS' 4 micron NMOS process. Available active devices in this technology are NMOS enhancement and depletion transistors. Measured threshold voltages for these devices were .6volts and -2.5volts respectively. Tables 5.11 and 5.12 summarizes the SPICE level 2 model parameters for these devices.

Parameter	Description
VTO	Threshold voltage with zero body bias
GAMMA	Body effect
TOX	Gate oxide thickness
KP	Transconductance- low field
UO	Mobility- low field
LD	Lateral diffusion
CJ	Junction area capacitance- zero bias
CJSW	Junction sidewall capacitance- zero bias
LAMBDA	Channel length modulation
VMAX	Maximum electron velocity to degrade mobility
UEXP	Fitting parameter for mobility degradation
UCRIT	Fitting parameter for mobility degradation

Table 5.11- Device Parameters

Parameter	Enhancement		Depletion		Units
	Measured	Target	Measured	Target	
VTO	.6	.8	-2.5	-2.6	V
GAMMA	.40	.65	.51	.56	$\sqrt{V}$
TOX		850		850	Å
KP	17.2	34	18	35	$\mu\text{A}/\text{V}^2$
UO	350		366		$\text{cm}^2/\text{Vs}$
LD	.5		.5		$\mu$
CJ	$1.3 \times 10^{-8}$		$1.6 \times 10^{-8}$		$\text{F}/\text{cm}^2$
CJSW	$3.5 \times 10^{-10}$		$3.5 \times 10^{-10}$		F/m
LAMBDA	.01		.015		$\text{V}^{-1}$
VMAX	$4 \times 10^4$		$3 \times 10^4$		m/s
UEXP	.23				
UCRIT	$2.6 \times 10^5$				V/cm

Table 5.12- 4micron NMOS parameters

The speed difference between simulations done with target and measured parameters is due primarily to the discrepancies of the transconductance parameters, KPs.

Available passive devices include resistors and capacitors. Except for one type of capacitor, these devices are all parasitics and formed by a single mask layer. These parasitics are considered in the next section since only one mask layer is needed to form them. The exception to this is a capacitor formed by the implant, active area, and polysilicon layers (Figure 5.10). The polysilicon layer is the positive plate of the capacitor and the implanted area is the negative plate. The implanted active area is the same structure as the channel of a depletion

transistor. The polysilicon layer acts the same way as the gate of the depletion transistor. Therefore, as long as the polysilicon layer is more positive than the implant area, the implant area or negative plate will be conductive, just as a depletion transistor's channel is conductive when its gate is more positive than its source. The capacitance of this device is determined by the gate oxide thickness and is  $.41\text{ff}/\mu^2$ .

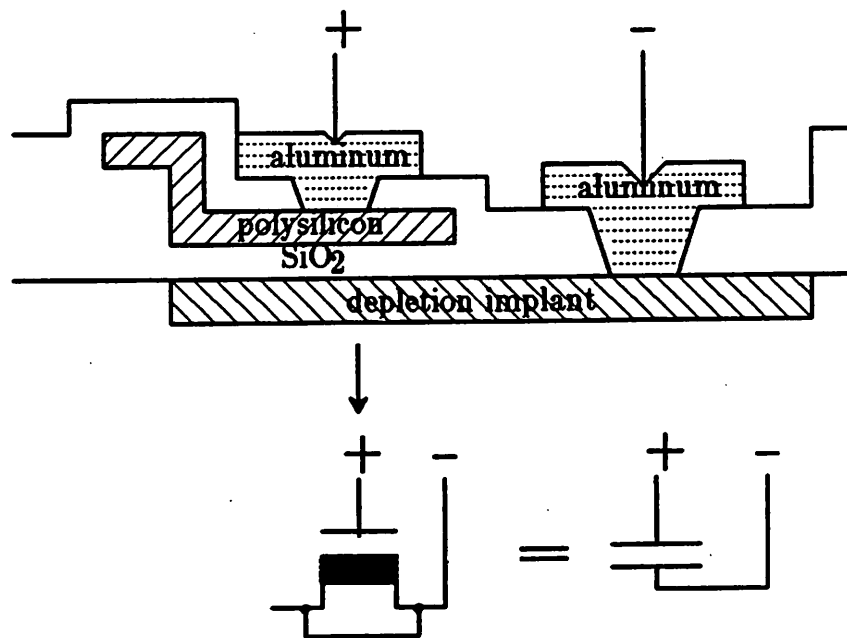


Figure 5.10- NMOS Capacitor Device

### 3.2. Layers and Layer Parameters

Mask layers of the MOSIS 4 micron NMOS process are summarized in Table 5.13. Two contact layers are available. The standard contact layer connects metal and diffusion or polysilicon. Buried contacts are also available to connect polysilicon directly to diffusion. Three interconnect layers are available – metal, diffusion, and polysilicon.

Layer	Resistance	Capacitance
Active/Diffusion	$20\Omega/\square$	$.16\text{ff}/\mu^2$ area $.35\text{ff}/\mu$ perimeter
Implant		
Buried Contact		
Polysilicon	$50\Omega/\square$	$.41\text{ff}/\mu^2$ gate $.06\text{ff}/\mu^2$ field
Contact		
Metal	$.03\Omega/\square$	$.05\text{ff}/\mu^2$
Overglass		

Table 5.13- 4micron NMOS Layers

### 3.3. Design Rules

Design rules for the MOSIS 4 micron NMOS process are lambda based, Mead Conway style design rules with lambda equal to two microns [Mead80]. For a complete description of these design rules see Appendix A. Principal single layer rules are summarized in Table 5.14. Minimum transistor dimensions are 4 microns for both the length and width.



Layer	Minimum Width	Minimum Spacing
Active/Diffusion	$2\lambda$	$3\lambda$
Polysilicon	$2\lambda$	$2\lambda$
Metal	$3\lambda$	$3\lambda$
Contact	$2\lambda \times 2\lambda$	

Table 5.14- Principal Single Layer Design Rules

#### 4. References

[Blau83a] Blau, R.; 'Paging on an Object-oriented Personal Computer', M.S. Report, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., 1983.

[Blau83b] Blau, R.; 'Tags and Traps for the SOAR Architecture', (Unpublished) Proceedings of 292R- Smalltalk on a RISC Architectural Investigations, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., April 1983.

[Blak83] Blakkan, J.; 'Register Windows for SOAR', (Unpublished) Proceedings of 292R- Smalltalk on a RISC Architectural Investigations, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., April 1983.

[Blom83] Blomseth, R.; Davis, H.; 'The Orion Project- A Home for SOAR', (Unpublished) Proceedings of 292R- Smalltalk on a RISC Architectural Investigations, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., April 1983.

[Brow85] Brown, E. W.; 'A Virtual Memory CPU Board with a Large Cache', M.S. Report, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., January 1985.

[Kate83] Katevenis, M. G. H.; 'Reduced Instruction Set Computer Architectures for VLSI', Ph.D. Thesis, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., 1983.

[Mead80] Mead, C.; Conway, L.; 'Introduction to VLSI Systems', Addison-Wesley Publishing Co., Reading, Ma., 1980.

[Samp85] Samples, A. D.; Klein, M.; Foley, P.; 'SOAR Architecture', T.R. UCB/CSD/85/226, University of California, Berkeley, Ca., March 1985.

[Unga84] Ungar, D.; Blau, R.; Foley, P.; Samples, A. D.; Patterson, D.;

'Architecture of SOAR: Smalltalk on a RISC', 11th Annual International Symposium on Computer Architecture, Ann Arbor, Mi., June 1984.

## Chapter 6

### Preliminary Design

### SOAR Case Study

Processor design is started with the preliminary phase after specifying the problem and the external inputs. As previously described, possible circuit and interconnect schemes are explored during this phase. Figure 6.1 shows the portion of the methodology that corresponds to the preliminary phase. According to the proposed methodology the two sides of the preliminary phase flow diagram are first considered separately during this phase. The flow diagrams for this are shown in Figures 4.19a and 4.19b for a full custom chip. SOAR was somewhat restricted from this. As previously discussed, the process was not open to modifications by the chip designers. Therefore, the process design level does not exist on the SOAR flow diagrams. Outputs of the process level are external inputs as outlined in the previous section of this case study. Figures 6.2a and 6.2b show the flow diagrams for the first part of the preliminary phase of the SOAR design. As previously discussed and indicated by the flowchart, the *preliminary circuit* and *preliminary interconnect* methodology flow diagrams are considered separately and therefore, may be considered simultaneously (Figure 6.1). Initial design proposals for the microarchitecture, circuit, and interconnect levels were developed for SOAR during the first part of the preliminary phase.

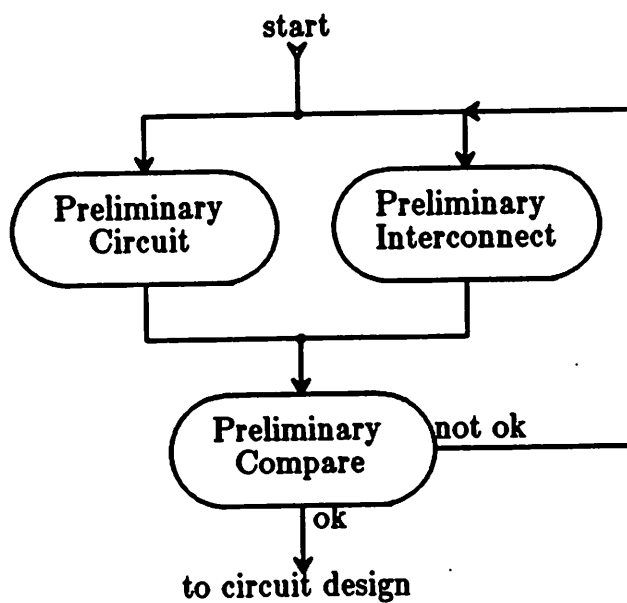


Figure 6.1- Preliminary Phase

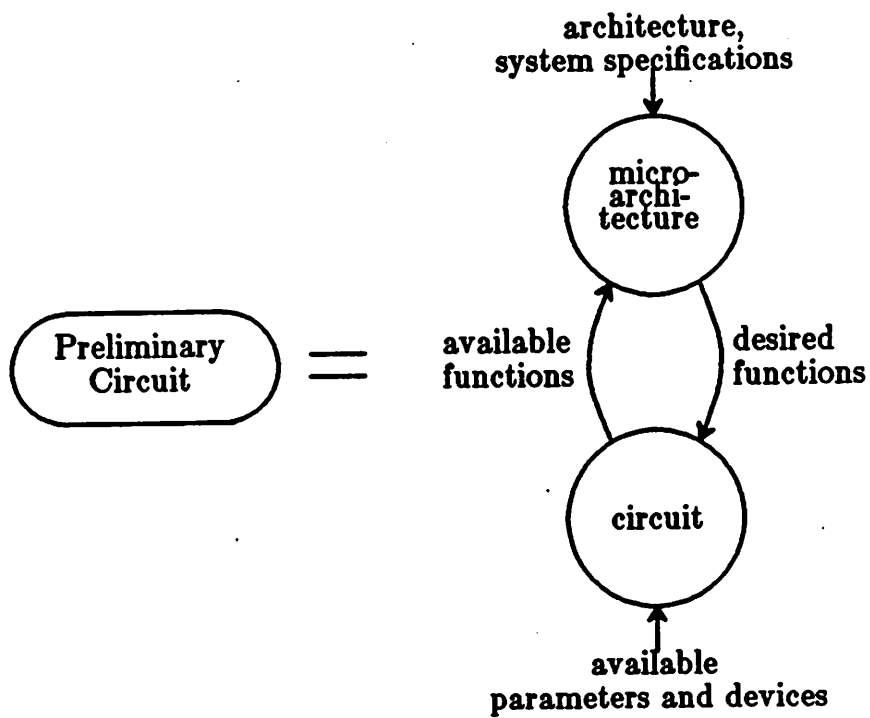


Figure 6.2a- Preliminary Circuit Step

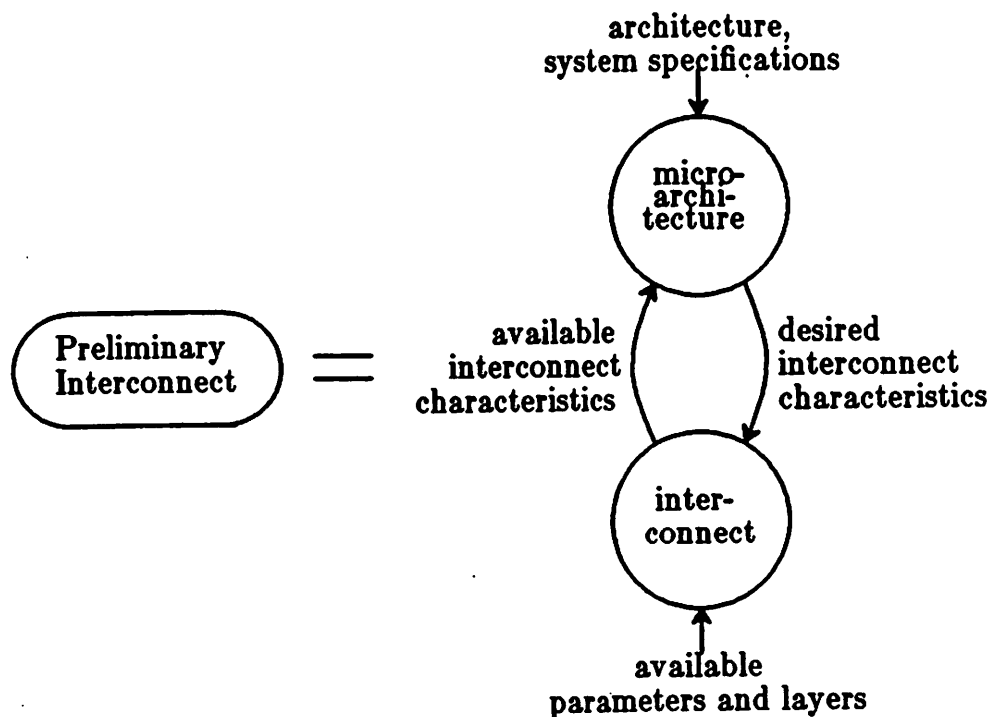


Figure 6.2b- Preliminary Interconnect Step

Design moves to the *preliminary compare* step after the *preliminary circuit* and *preliminary interconnect* steps. Design levels common to both the *preliminary circuit* and *preliminary interconnect* steps are compared during the *preliminary compare* step (Figure 4.19c). The *preliminary compare* step for SOAR (Figure 6.3) is simpler than that of a full custom design (Figure 4.19c). Once again, this is because the process was not open to modification by SOAR designers. Only the microarchitecture level was compared in the *preliminary compare* step for SOAR. If this comparison reveals no discrepancies between the designs resulting from the *preliminary circuit* and *preliminary interconnect* steps the preliminary phase is completed. If discrepancies do exist design must return to the *preliminary circuit* and *preliminary interconnect* steps until the discrepancies are resolved.

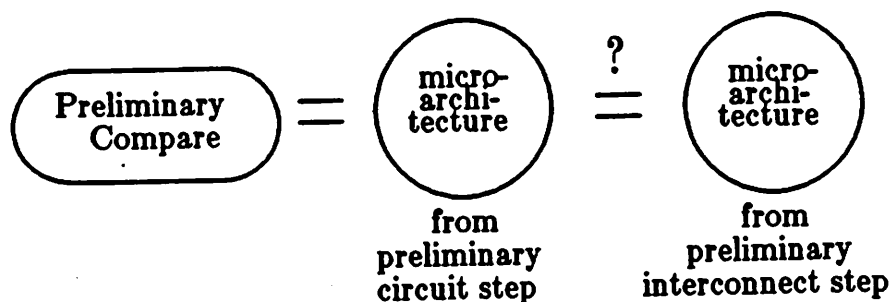


Figure 6.3- Preliminary Compare Step

## 1. Preliminary Circuit

The *preliminary circuit* flow diagram is shown in Figure 6.2a. As previously discussed, there are two aspects to microarchitecture design – specification of the functions to be done by the processor and coordination of the units that perform them. Therefore, early in this step as part of microarchitecture design, the instruction set is examined to determine the operations that must be performed and the order that they may be performed in. The operations that must be performed become the *desired functions* output of the microarchitecture level. The order that they may be performed in, provides a starting point for the behavioral aspect of microarchitecture design.

### 1.1. Desired Functions

#### 1.1.1. ALUs

The SOAR architecture was first examined to determine the necessary arithmetic and logic units. The instruction set was most important in determining the operations to be performed. This instruction set was discussed in detail in the preceding section on external inputs [Samp85]. Table 5.5 summarizes the instructions and the operations that they require. A list of operations can be

extracted from this (Table 6.1).

Operation	Instructions
Addition	CALL, JMP, RET, ADD, SLL, LOAD(C), STORE
Subtraction	SKIP, TRAP <sub>i</sub> , SUB, LOADM, STOREM
And	AND
Or	OR
Exclusive or	XOR
Logical right shift- 1 bit	SRL
Arithmetic right shift- 1 bit	SRA
Insertion, zeroing	INSERT
Extraction, zeroing	EXTRACT
Decrement- 5 bit latch	LOADM, STOREM
Increment- 3 bit latch	RET
Decrement- 3 bit latch	CALL

Table 6.1- Operations Required

In addition to the functions of Table 6.1, an inspection of the instruction fields reveals the need for a sign extender. The second operand is either specified by a register or a 12 bit constant that requires sign extension.

The register organization of SOAR was a second part of the architecture that required arithmetic units. As previously described, the registers are organized into windows. The accessible register window changes with a procedure call or return. A call decrements the window number and a return optionally increments the window number. There are eight windows and the current window is pointed

to by a 3 bit latch – the current window pointer (CWP). Consequently, this 3 bit latch must contain an incrementer/decrementer.

Registers may be specified by their register number if they are in the current window or by the memory location that they map into. Register contents of previous windows may be in the register file or may have been written to memory. A special purpose comparator is used to determine the location of the contents of registers from previous windows. This is the saved window pointer comparator (SWP comparator).

The final arithmetic unit required by the architecture is an incrementer for the program counter. This is used to sequentially address memory for instructions.

### **1.1.2. Storage**

The SOAR architecture suggested two main types of storage circuits – temporary registers and the static register file cells. The special registers are temporary registers. These include the shadow registers, program counter, process status word, current window pointer, saved window pointer, and trap base latch. Temporary registers are also used to latch instructions. Much of the state of SOAR is contained in these latches. It was desirable to be able to single step SOAR during testing. Therefore, it was decided that these latches would be master/slave latches with a builtin refresh ability.

The other type of storage cell is the register file cell. The register file cell is a static RAM cell. It was desirable to avoid the refresh and clocking complexities of dynamic memories and therefore static memory cells were chosen.



### 1.1.3. Random Logic

An inspection of the architecture and system requirements showed that there would be a significant amount of random logic in SOAR [Samp85]. Instructions and register specifiers must be decoded. Many types of traps must be detected. This necessitates tag examination, detection of illegal opcodes, and detection of window underflow and overflow. Conditions must be checked on skip and trap instructions. To save design time and minimize errors a regular structure was desirable for implementing the random logic functions. CAD tools were available for automatic PLA generation. Therefore, it was decided that as many of these functions as possible would be implemented with PLAs.

PLAs implement an AND function followed by an OR function (Figure 6.4). The number of inputs to each AND and OR is variable but basically PLAs are two level AND/OR functions. Register file decoding is naturally a single level AND function. Thus, PLAs are not ideally suited for use as register file decoders. Due to the large register file and therefore large number of register file decoders, it was decided that PLAs would not be used for the register file decoders. These decoders would be custom designed.

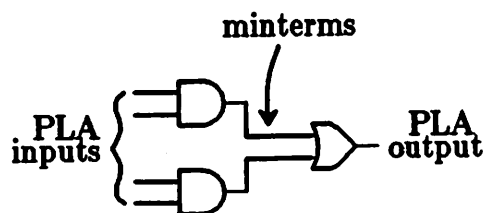


Figure 6.4- PLA Function

#### 1.1.4. Drivers

The fourth major category of necessary circuits were drivers. Drivers were needed for many large loads – datapath control lines, register file word lines, PLA outputs, and as pad drivers.

The datapath width was 32 bits as required by the word size. Most control and word line drivers were therefore required to drive a high fanout and the accompanying routing across the datapath. Many of the control signals had to be gated with clocks. Depending on the signal, control lines could be enabled with either a high logic level or low logic level. Control line drivers had to be designed for all of these possibilities.

PLA outputs typically had few gates to drive but large routing loads. This necessitated drivers, but these drivers did not need to be as strong as the control and word line drivers due to the reduced fanout.

The strongest drivers were needed to drive the output pads. These drivers had to drive the package pins and meet Schottky TTL requirements. These requirements were summarized in Chapter 5, Section 2.6. Data outputs had to be tristated; other outputs did not need tristating.

Table 6.2 summarizes the main types of drivers and their approximate loads. Power consumption was a major consideration in driver design since large numbers of drivers were needed.

Driver	Fanout	Routing Load	Pins
Control line	high	high	no
Word line	high	high	no
PLA	low	high	no
Pad	low	high	yes

Table 6.2- Driver Requirements

### 1.1.5. Summary

Table 6.3 summarizes the *desired functions* input to the circuit design level during the *preliminary circuit* step. Circuit design began in response to these inputs. These functions covered most of the total circuit design. Characteristics of these circuits had a major impact on the microarchitecture. During this preliminary phase the microarchitecture is still fairly flexible. By considering these characteristics intelligent decisions in the microarchitecture may be made. However, as the total chip design was completed other unforeseen random logic circuits appeared. This unforeseen random logic was minor, as it should be, and had no noticeable affect on the microarchitecture.

Desired Functions	Circuit Type
Addition	ALU
Subtraction	ALU
And	ALU
Or	ALU
Exclusive or	ALU
Logical right shift- 1 bit	ALU
Arithmetic right shift- 1 bit	ALU
Insertion, zeroing	ALU
Extraction, zeroing	ALU
Decrement- 5 bits	ALU
Increment- 3 bits	ALU
Decrement- 3 bits	ALU
Sign extension	ALU
SWP comparator	ALU
Increment- 28 bits	ALU
Temporary registers	Storage
Register file cell	Storage
PLAs	Random Logic
Register file decoders	Random Logic
Control line inverter	Driver
Control line NOR	Driver
Control line OR	Driver
Word line inverter	Driver
Pad Driver	Driver
Pad tristate driver	Driver
Other drivers	Driver

Table 6.3- SOAR Desired Functions (Preliminary Phase)

## 1.2. Circuits Available

### 1.2.1. ALUs

SOAR's main ALU is a full 32 bit ALU that performs addition and subtraction. Subtraction is performed with the same hardware as the addition by using a ones complement algorithm. Many methods exist for performing addition [Sher84a] [Whal84]. All have their advantages and disadvantages. Tradeoffs between such things as speed, power, area, and layout regularity greatly affect VLSI adder design. For SOAR, a fast but relatively simple adder was desired [Bose83]. It was not desirable to spend large amounts of area and design time on complex carry lookahead circuitry. With this in mind, a carry bypass scheme used by Siemens Research Laboratories was chosen [Pomp82]. Carry lookahead is done for each block of four bits. Figure 6.5 shows the original carry lookahead circuitry for a four bit block. The Cin line is precharged high for increased speed during evaluation.

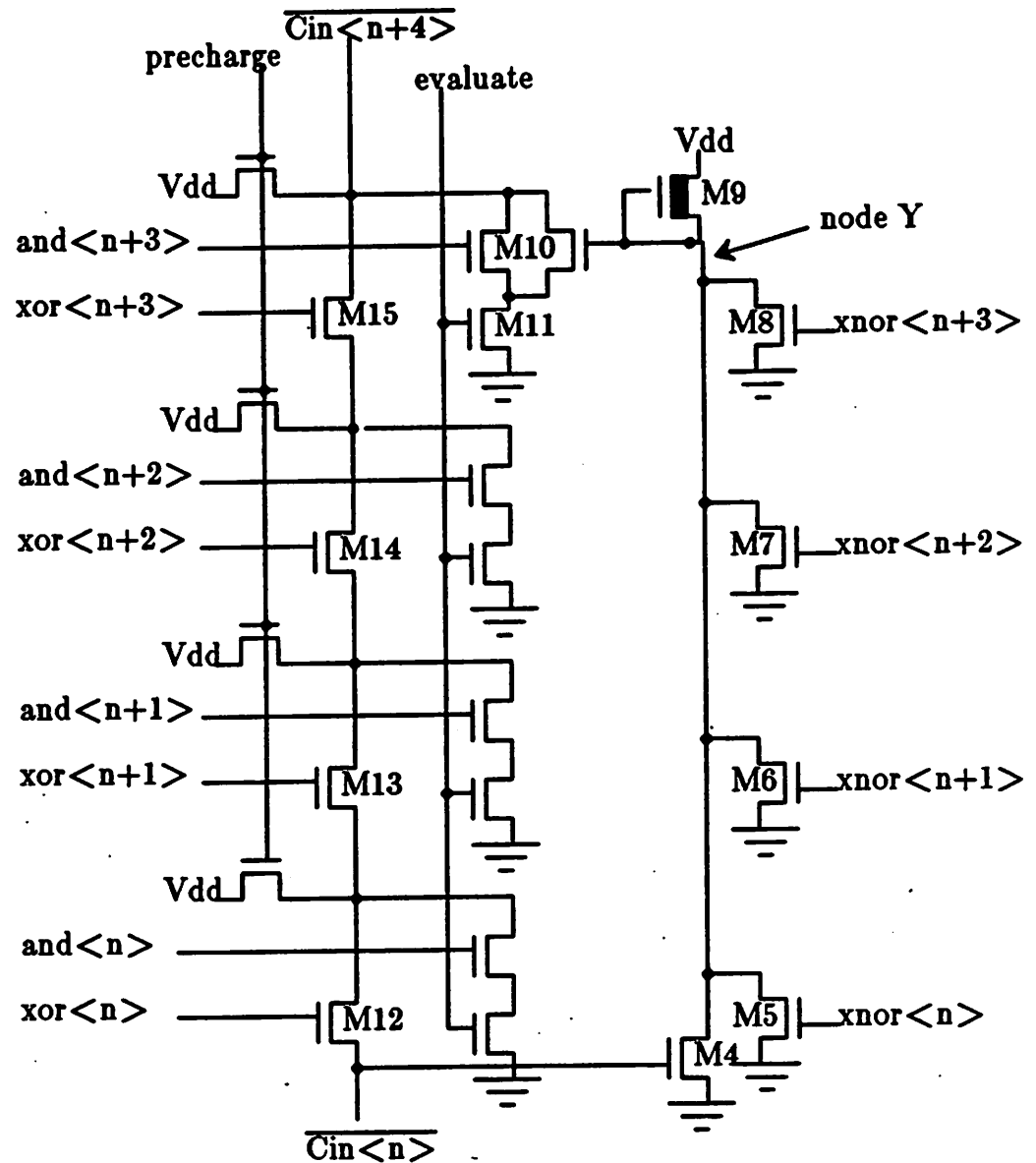


Figure 6.5- Four Bit Carry Generation- Siemens Scheme

For a single bit, the carry is described by:

$$C_{out}\langle n \rangle = C_{in}\langle n+1 \rangle = A\langle n \rangle B\langle n \rangle + A\langle n \rangle C_{in}\langle n \rangle + B\langle n \rangle C_{in}\langle n \rangle$$

A single gate to implement this function is shown in Figure 6.6. Node X corresponds to the precharged  $\overline{C_{in}\langle n \rangle}$  signal. The single bit carry of Figure

6.6, may be implemented with precharging of the  $\overline{\text{Cin}}_{\langle n \rangle}$  and  $\overline{\text{Cin}}_{\langle n+1 \rangle}$  signals as shown in Figure 6.7. The function  $A+B$  formed by transistors M1 and M2 in Figure 6.6 must be replaced by  $A \oplus B$  - M3 - when  $\overline{\text{Cin}}_{\langle n \rangle}$  is precharged. If it is not,  $\overline{\text{Cin}}_{\langle n \rangle}$  will be determined by  $A_{\langle n \rangle}$  and  $B_{\langle n \rangle}$  for the case when both  $A_{\langle n \rangle}$  and  $B_{\langle n \rangle}$  are high, which is incorrect.

These individual carry blocks are chained together to form  $\text{Cin}$  for each bit of the adder. When chained together the M3 transistors of each bit form a series of pass transistors that  $\overline{\text{Cin}}$  must propagate through - transistors M12 through M15 in Figure 6.5. Propagation through a large number of pass gates is inherently slow. Therefore, hardware to bypass these pass gates is added to each group of four bits - transistors M4 through M11 of Figure 6.5. When all four pass gates are on,  $\overline{\text{Cin}}_{\langle n \rangle}$  will slowly propagate through transistors M12, M13, M14, and M15 to become  $\overline{\text{Cin}}_{\langle n+4 \rangle}$ . Meanwhile,  $\overline{\text{Cin}}_{\langle n \rangle}$  is rapidly inverted once to become  $\text{Cin}_{\langle n \rangle}$  at node Y, and then again to form the  $\overline{\text{Cin}}_{\langle n+4 \rangle}$  output.

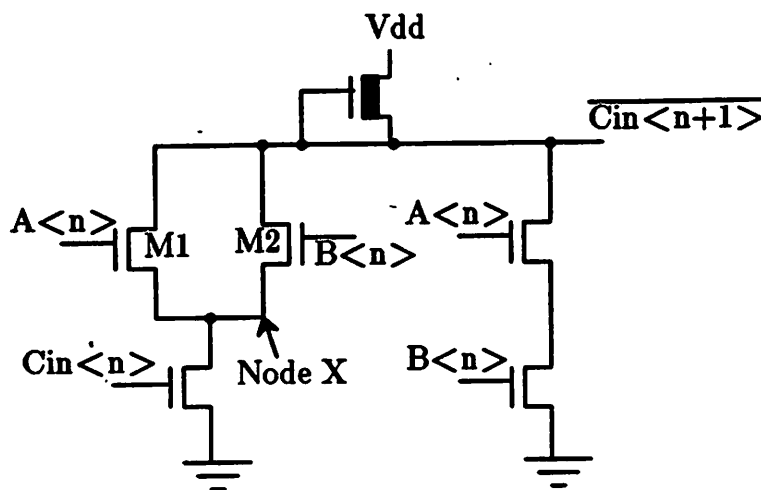


Figure 6.6- Single Bit Carry- No Precharge

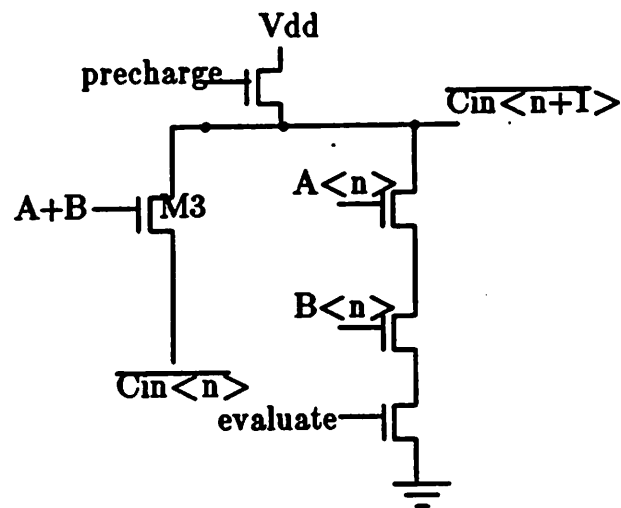


Figure 6.7- Single Bit Carry- Precharge

This carry bypass scheme meets the requirements of minimum extra complexity and area spent on lookahead circuitry. The bypass circuitry incorporates eight extra transistors for every four bits and may be simply duplicated for each group of four bits. The area occupied by the bypass circuitry accounted for 10% of the SOAR ALU. Pomper, et. al. reported a 35ns, 32 bit carry propagation time for their NMOS technology, using this scheme [Pomp82]. Allowing for technology differences, less than 100ns was expected for a 32 bit add with this scheme on SOAR [Bose83].

Another common method of carry generation is the ripple carry technique. This scheme uses no carry lookahead hardware. The carry output of any given bit is simply used as the carry input for the next most significant bit. The advantage of ripple carry adders is their simplicity. However, this type of adder is inherently slower for a large number of bits. Therefore, on SOAR it was used only when few bits were to be added – the 5 bit decrementers – and for large adds when speed was not thought to be a factor – the 28 bit incrementer for the program counter and the SWP comparator.



The SWP comparator compares the saved window pointer and an address of a memory reference to determine if the contents of the memory location are stored on chip or off chip. The saved window pointer points to the address of register 0 of the last window written to memory. The comparison done by the SWP comparator is:

$$[\text{SWP}\langle 27:4 \rangle - \text{address}\langle 27:4 \rangle - 1]\langle 27:7 \rangle = 0$$

A ripple carry adder is used to perform this 24 bit computation and then bits 7 through 27 of the answer are NORed together to check for 0 (Figure 6.8).

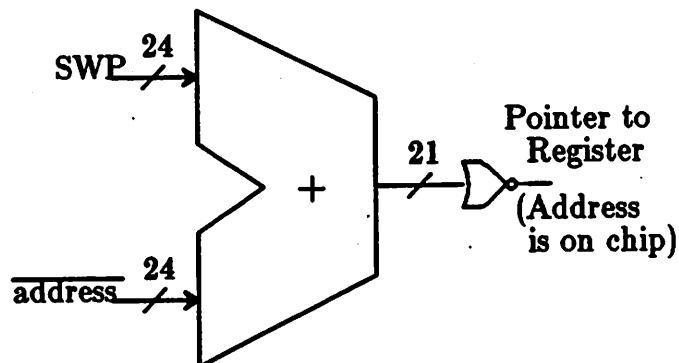


Figure 6.8- SWP Comparison

Another adder required by SOAR was a 3 bit incrementer/decrementer. Since only three bits were involved it was decided that it would not be very difficult to compute the output for all three bits in parallel. Circuits for this are shown in Figure 6.9.

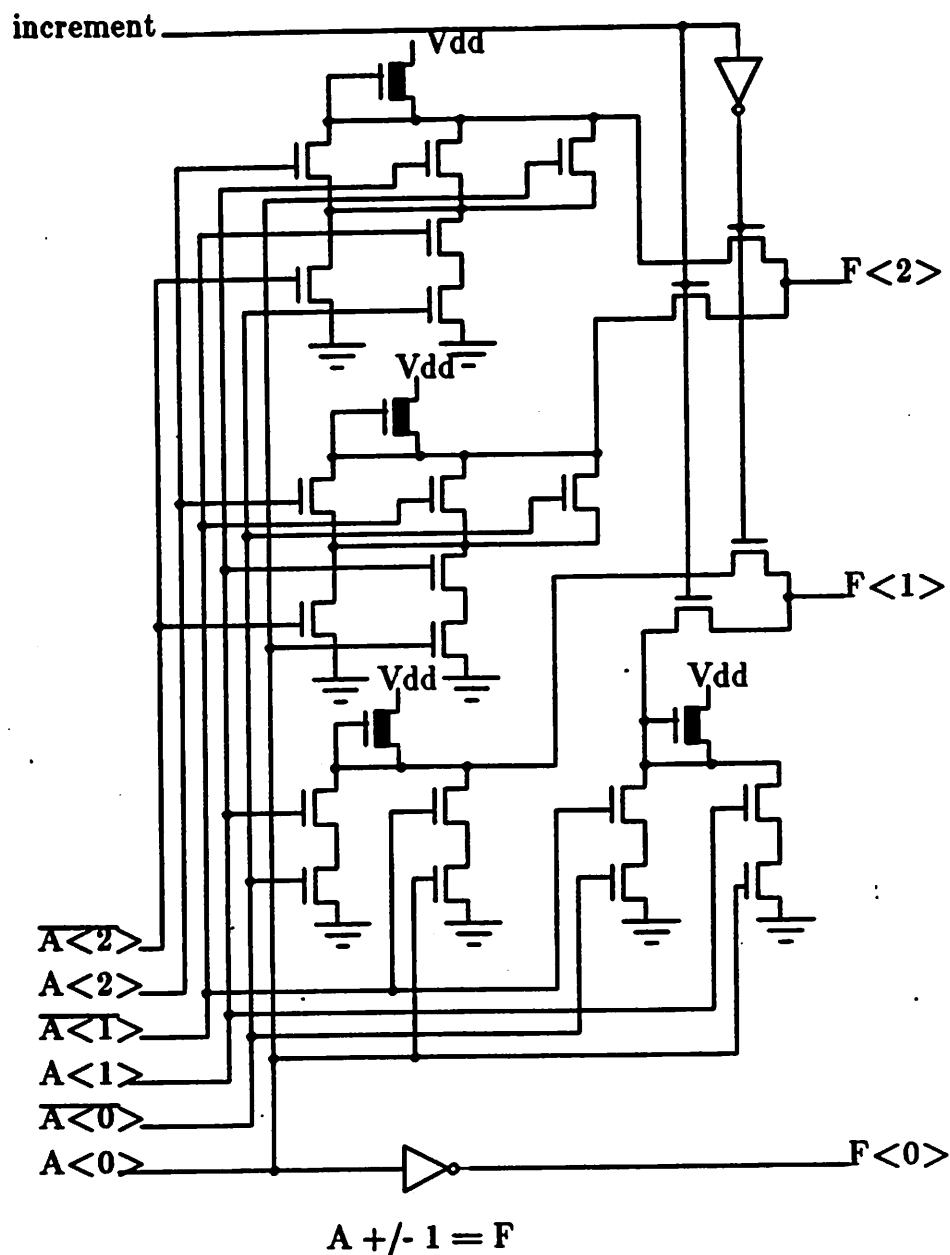


Figure 6.9- Three Bit Incrementer/Decrementer

Byte operations require an inserter and extractor, as previously described (Ch.5, Sec.1.5). Figure 5.7 shows a block diagram for insertion and extraction. Insertion and extraction are performed on the first operand. The byte to be inserted or extracted is specified by the two least significant bits of the second operand. Figure 6.10 shows the portion of the inserter/extractor that the first

operand flows through. Control lines are set according to Table 6.4. Control lines ex3, ex2, ex1, ex0/in, in3, in2, in1, and in0/ex cause the appropriate input byte to be routed to the correct output byte. Signals in3\*/ex, in2\*/ex, in1\*/ex, and in0\* cause all other output bytes to be zeroed. The two least significant bits of the second operand are decoded to set the appropriate control lines.

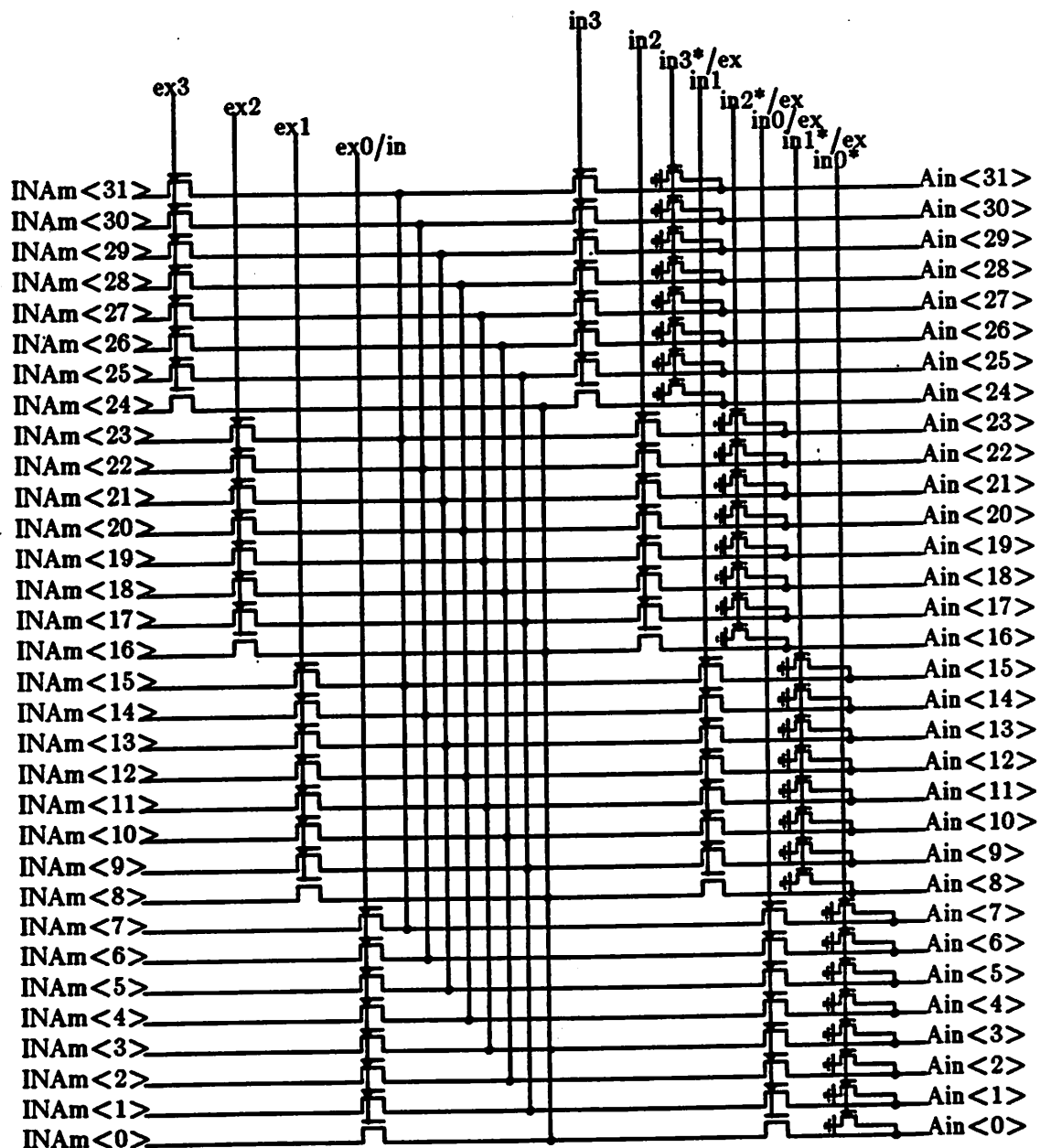


Figure 6.10- Inserter/Extractor

Operation	2 LSBs of 2nd Operand	Enabled Control Lines
Extract	0	ex0/in, in0/ex, in3*/ex, in2*/ex, in1*/ex
Extract	1	ex1, in0/ex, in3*/ex, in2*/ex, in1*/ex
Extract	2	ex2, in0/ex, in3*/ex, in2*/ex, in1*/ex
Extract	3	ex3, in0/ex, in3*/ex, in2*/ex, in1*/ex
Insert	0	ex0/in, in0/ex, in3*/ex, in2*/ex, in1*/ex
Insert	1	ex0/in, in1, in0*, in2*/ex, in3*/ex
Insert	2	ex0/in, in2, in0*, in1*/ex, in3*/ex
Insert	3	ex0/in, in3, in0*, in1*/ex, in2*/ex

Table 6.4 Insert/Extract Control Lines

Sign extension is also performed by an array of pass transistors (Figure 6:11). As previously discussed (Ch.5, Sec.1.5), bit 7 of the immediate constant is the bit to be sign extended. Bits 8 through 11 of the immediate constant become the tag bits of the 32 bit sign extended immediate (Figures 5.5 and 5.6). The location of bits 7 through 11 of the immediate depend on the instruction type – store or non-store. Store instructions select bits 19 through 22 of the instruction as the tag bits and bit 18 as the sign extension bit. For other instructions bits 8 through 11 become the tag bits and bit 7 becomes the sign extended bit.

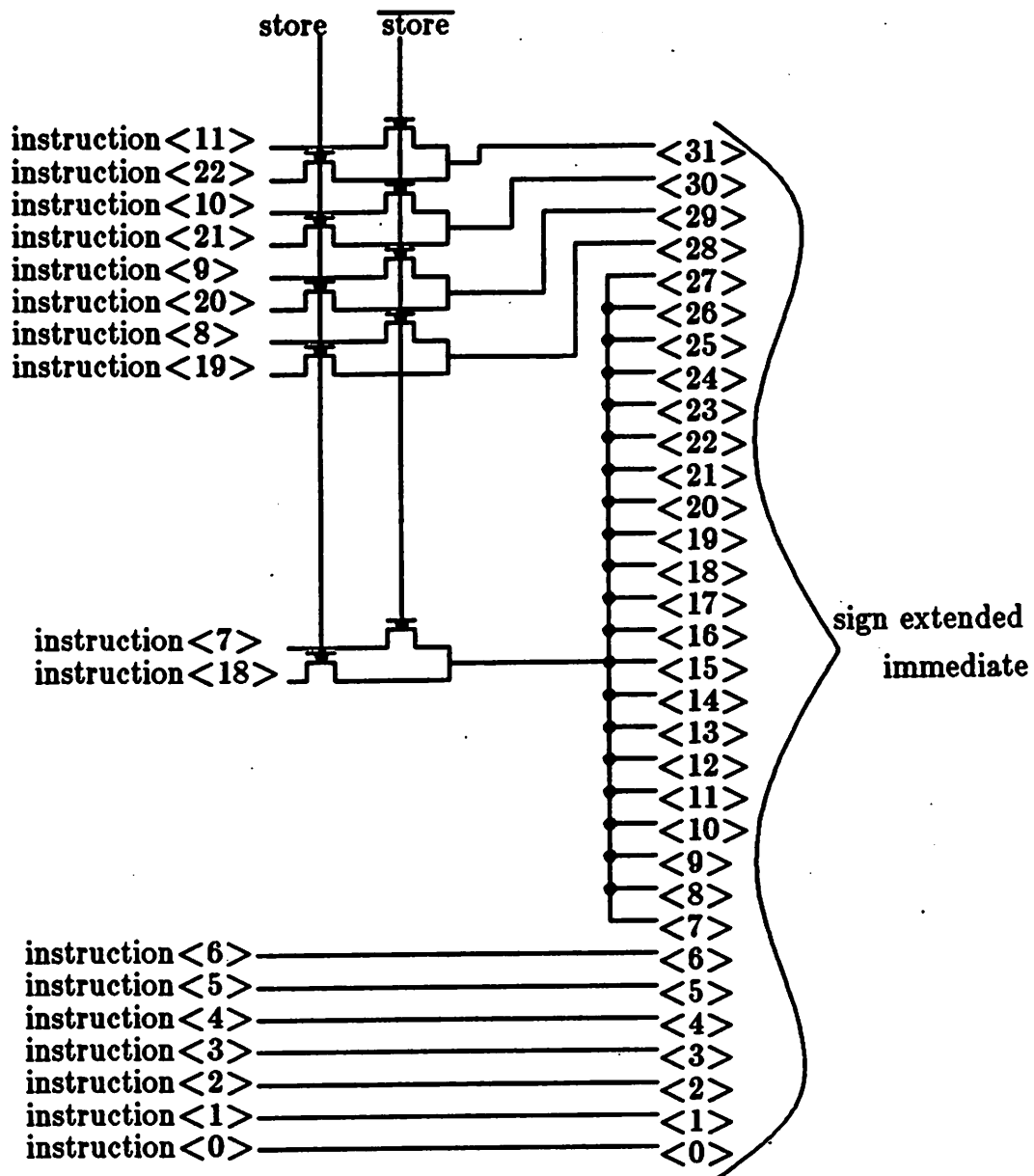


Figure 6.11- Sign Extension

### 1.2.2. Storage

A basic master/slave latch with a built in refresh capability was designed for the temporary registers (Figure 6.12). Multiple load transistors can be used to selectively load data from one of many inputs. Data is loaded into the master section by enabling one load transistor. The step transistor allows data to be loaded into the slave section from the master section. Enabling the refresh

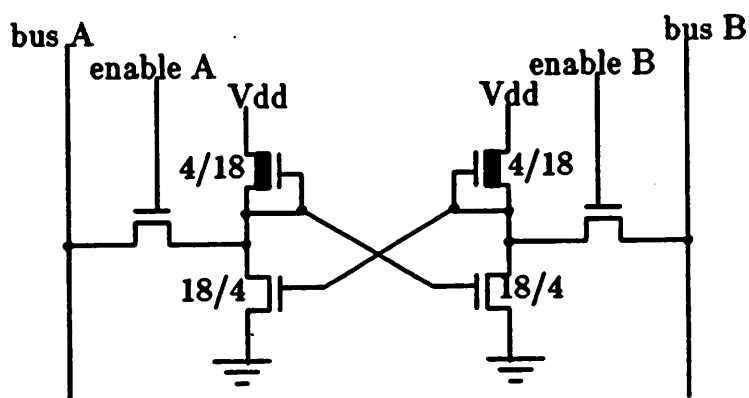


Figure 6.13- Static RAM Cell

Both types of storage cells read data onto precharged buses. The speed of the read operation may be approximated using a simple resistor/capacitor model for the bus (Figure 6.14). Figures 6.15 and 6.16 show the propagation delay of the read operation as a function of  $R_{bus}$  and  $C_{bus}$ . According to SPICE simulations, propagation delays of 24ns to 44ns could be expected for the latch cell and 14ns to 40ns for the register file cell, for loads up to 2.5pF and 10K $\Omega$ .

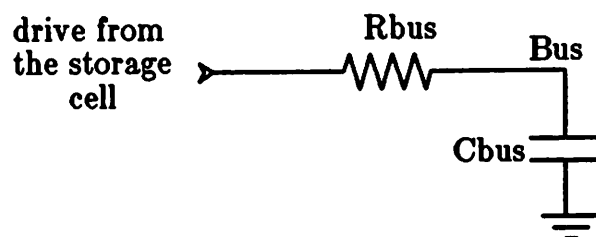


Figure 6.14- Simple RC Bus Model

transistor causes the master node to be updated from the slave section. Data from the slave section is read onto a precharged bus by a low logic level at the read enable input. Due to the master/slave arrangement reads and writes to the same register may be done simultaneously. A read retrieves the old data from the slave section while a write puts new data into the master section.

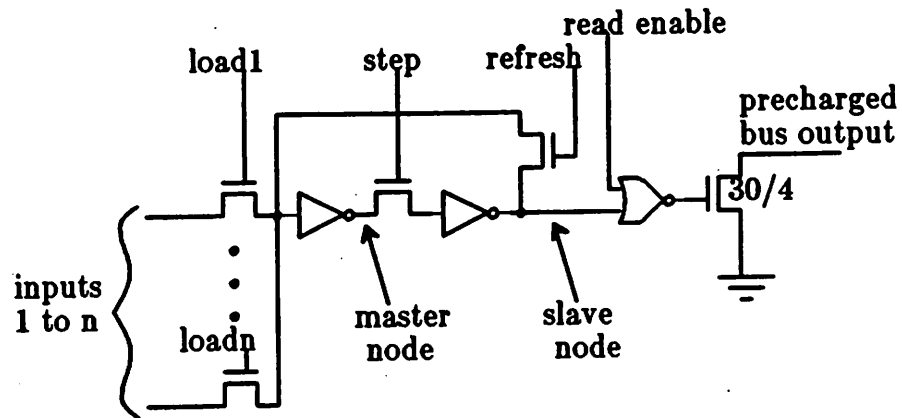


Figure 6.12- Master/Slave Latch

Static RAM cells were used for the register file. A previous Berkeley microprocessor, the RISC II, also had a large register file, similar to that of SOAR [Sher84b]. The register cell design of RISC II was successful and therefore used on SOAR (Figure 6.13). Writes are accomplished by putting the data that is to be written on busA and its complement on busB. Both enableA and enableB are enabled during a write. This is necessary to override the old data of the cell. Reads are accomplished by selecting either enableA or enableB depending on which bus is to be used for the read. Thus, reads from two of these registers may be done simultaneously. Only one write can be done at a time. Unlike the master/slave latch, reads and writes may not be done simultaneously.

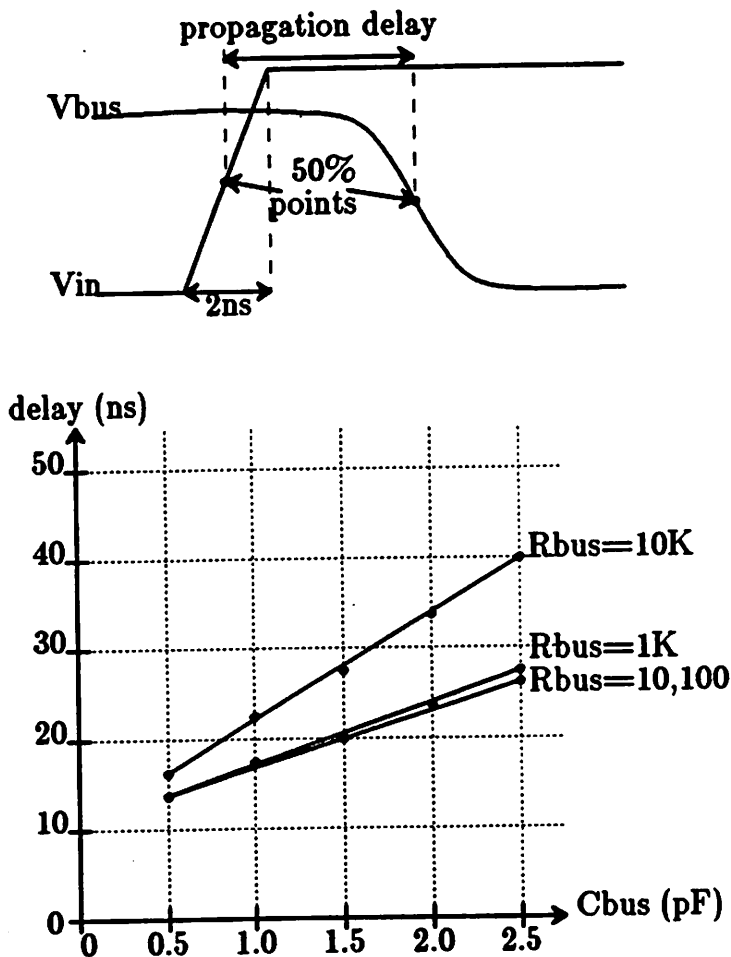


Figure 6.16- Static RAM Propagation Delay

### 1.2.3. Random Logic

PLAs are used for much of the random logic on SOAR. Using CAD tools available at Berkeley, PLAs can be generated automatically [VanD82] [Laru83] [Scot85]. PLAs are composed of two level AND/OR functions that are realized by NOR/NOR configurations on SOAR (Figure 6.17). Input buffers to the PLAs are two sequential inverters (Figure 6.18). Buffered true and complement forms of the input signal are available from this input circuit. This input buffer was the only one incorporated by Berkeley's PLA CAD tools at the time of the SOAR design. Since then other input buffers have been made available to the CAD tools [Ober85].



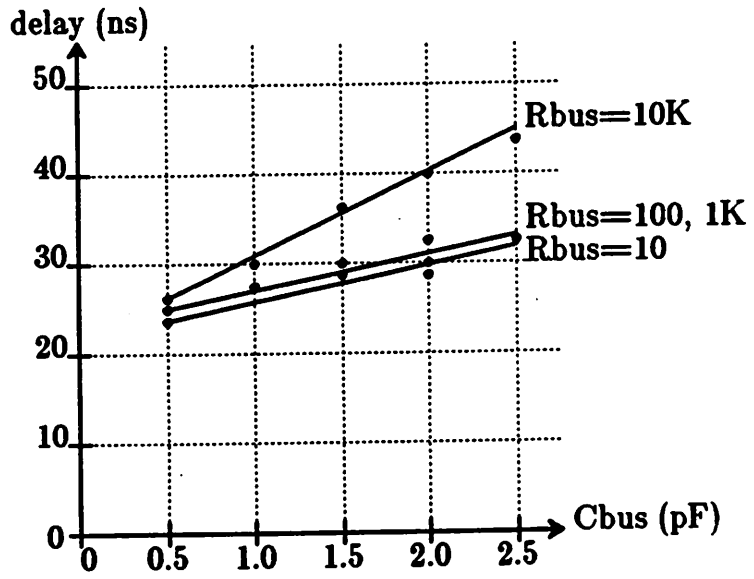
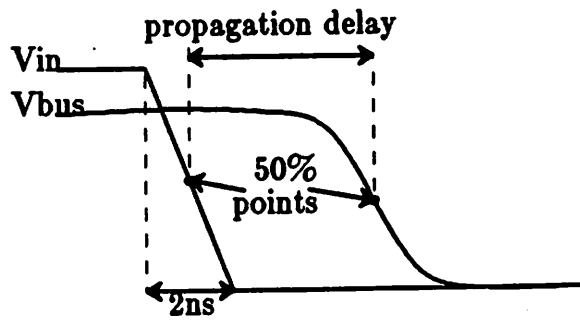


Figure 6.15- M/S Propagation Delay

No. of Minterms	Speed
15	~40ns
45	~100ns
100	~200ns

Table 6.5- PLA Speed Estimates

Preliminary circuit design of the register file decoders was greatly influenced by the register window organization and the large number of registers – 72. Nine windows of eight registers each exist in the register file. The first significant implication of this is that 7 bits of register address must be decoded to access the specified register. Address decoding is inherently an AND function. Thus, decoding could have been implemented in one level with either 72 seven input NAND gates or 72 seven input NOR gates (Figure 6.19). Single level decoding with either NAND or NOR gates would have required a large area due to the large number – 72 – of high fan in – 7 – gates. The single level NAND decode also had the disadvantage of being slow due to high fan in. The NOR method would have had high power dissipation since all gates except for the selected one would have been on.

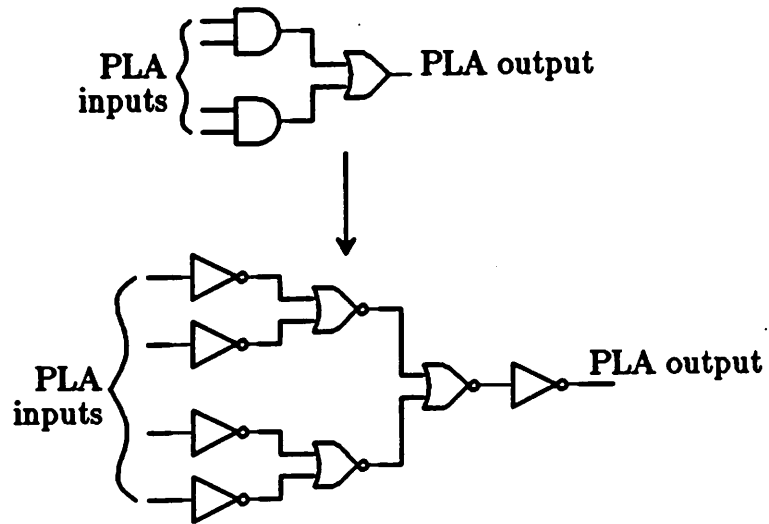


Figure 6.17- PLA Function

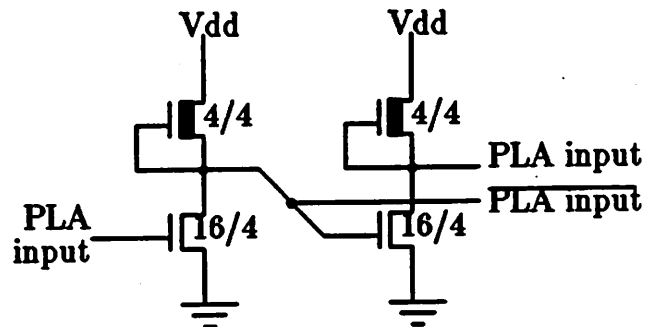


Figure 6.18- PLA Input Buffer

Speed estimates for the PLAs were obtained with CRYSTAL simulations of three sizes of test PLAs [Oust85]. Table 6.5 summarizes these speed estimates.

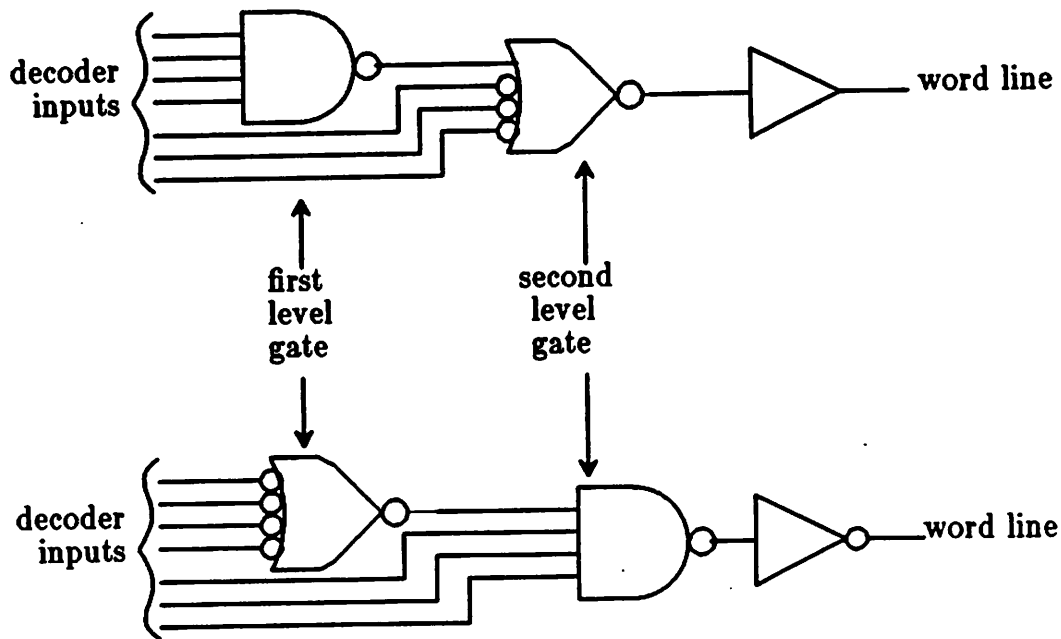


Figure 6.20- Two Level Decode

Figure 6.21 shows the basic SOAR decoding scheme used for the 64 local registers. Two NOR gates are used for the first level of the decode. They are enabled if the register number specifies a local register. These NOR gates decode the three address bits of the current window pointer that specify the register window. One NOR decoder selects a given window when that window corresponds to the low registers - r0 to r7. The other NOR decoder selects the same window when the CWP has been decremented and the same 8 registers correspond to the high registers - r8 to r15. The second level of the decode, the NAND gate, decodes the register number within the window.

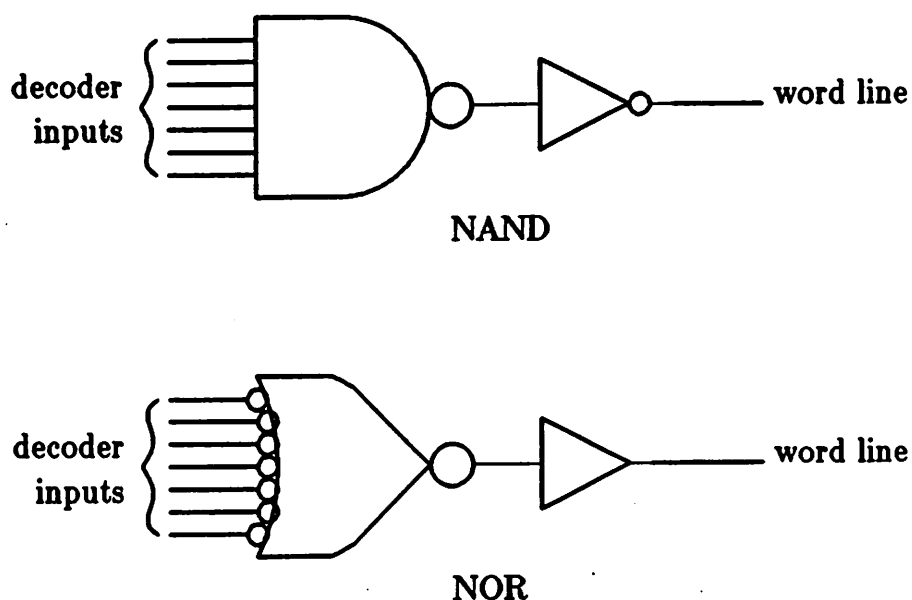


Figure 6.19- Single Level Decode

Two methods of two level decoding were considered (Figure 6.20). Both methods involve two sequential 4 input gates. The second level gate enables the word line driver of the register file. Therefore, one second level gate exists for each register: 72 total second level gates are required. The first level gate has an enable input and three address inputs. Thus, only eight first level gates are required. The first method consists of a four input NAND gate followed by a four input NOR gate. This again has the disadvantage of high power consumption because all 72 second level NOR gates are on - low output - except the one for the selected register. One first level gate is on, resulting in a total of 72 gates consuming power. In the second two level method a four input NOR gate is followed by a four input NAND gate. At any one time, one NAND gate and seven NOR gates will be on resulting in a total of eight gates consuming power. This scheme was chosen for SOAR due to its lower power consumption.

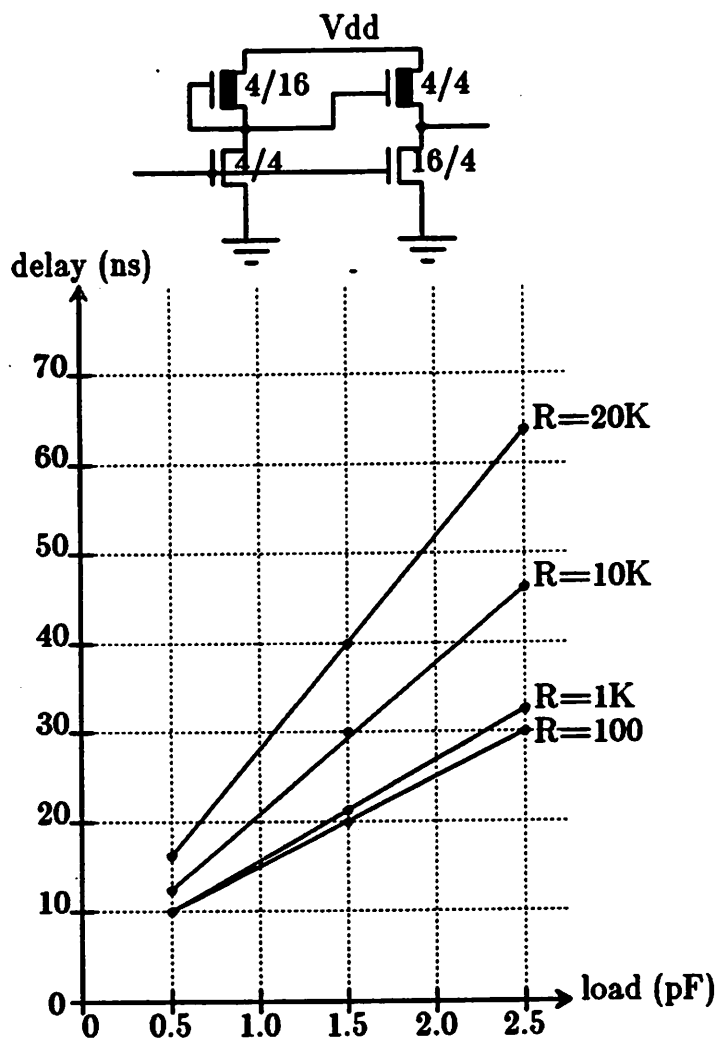


Figure 6.22- Inverting Driver and Propagation Delay- D1

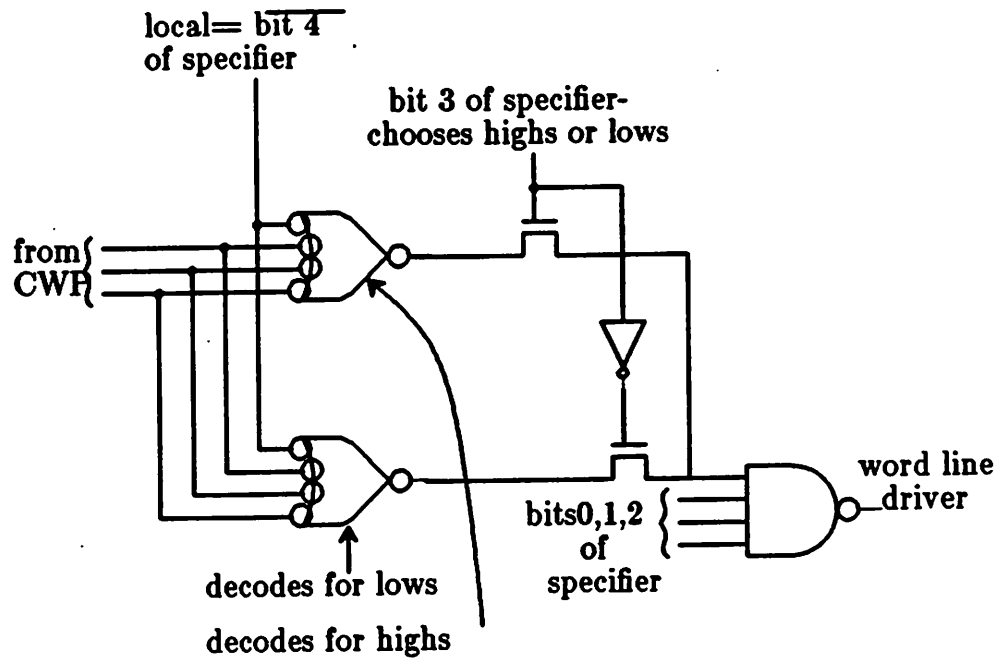


Figure 6.21- SOAR Register File Decode

#### 1.2.4. Drivers

Typical Mead-Conway drivers and their propagation delays for various loads are shown in Figures 6.22, 6.23, and 6.24. Other important characteristics for the evaluation of a driver is its power dissipation and output high level. Drivers D1 and D2 dissipate static power in both the input and output stages. The output stage must be large to drive the load and this results in high static power dissipation in the output stage. Output high levels of D1 and D2 are  $V_{dd}$ . In contrast to D1 and D2 is D3. D3 has no static power dissipation in the large output stage, resulting in lower power dissipation than in D1 and D2. However, its disadvantage is that the high output level is a threshold voltage below the supply voltage.

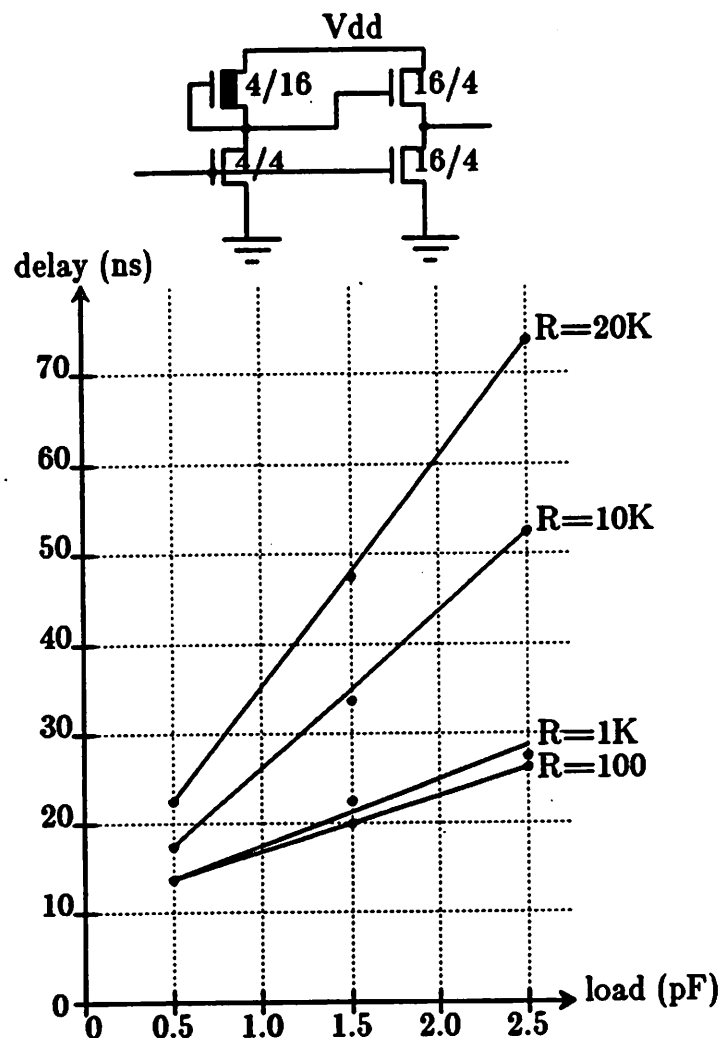


Figure 6.24- Inverting Driver and Propagation Delay- D3

As previously described, many large fast drivers were needed for control lines, word lines, and the output pads. Therefore, it was desirable to design drivers with low power dissipation and an output high level of  $V_{dd}$ . Bootstrap drivers meet these requirements. A basic two stage bootstrap driver is shown in Figure 6.25. The only static power dissipation in these bootstrap drivers is in the small input stage. Thus, power dissipation is low as in driver D2. As the output of this type of driver rises, the drain of transistor M2 rises above  $V_{dd}$  due to the capacitive coupling between the drain and  $V_{out}$  through the bootstrap capacitor,  $C_b$ . The source of M2 follows the drain above  $V_{dd}$ . Thus, the gate of M1 rises



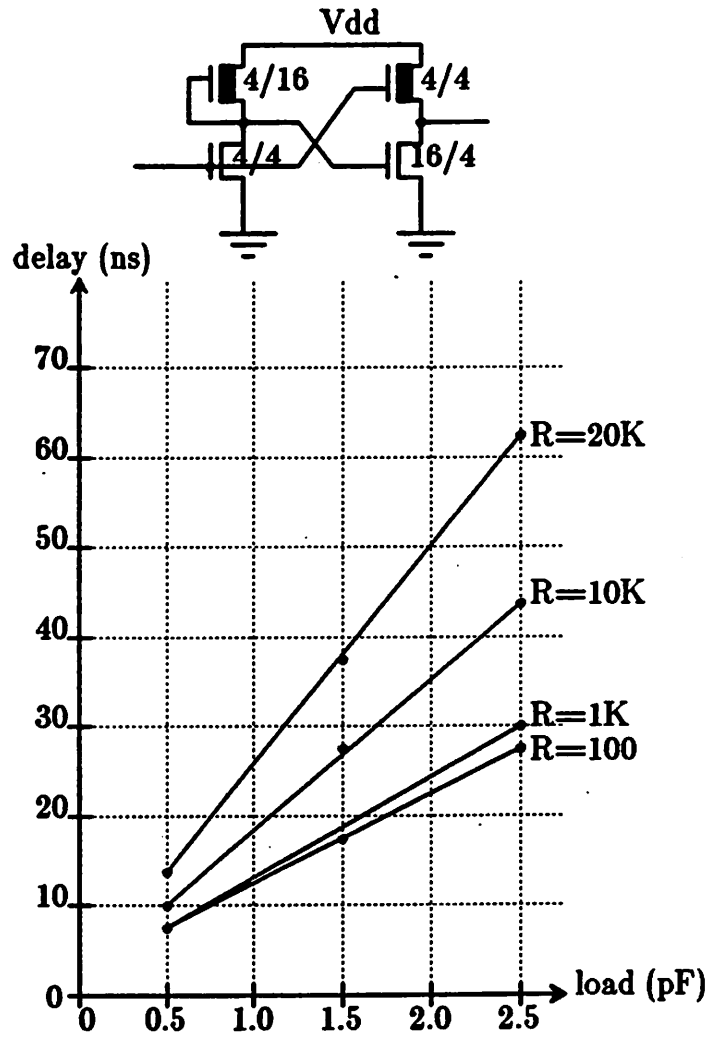


Figure 6.23- Non-Inverting Driver and Propagation Delay- D2

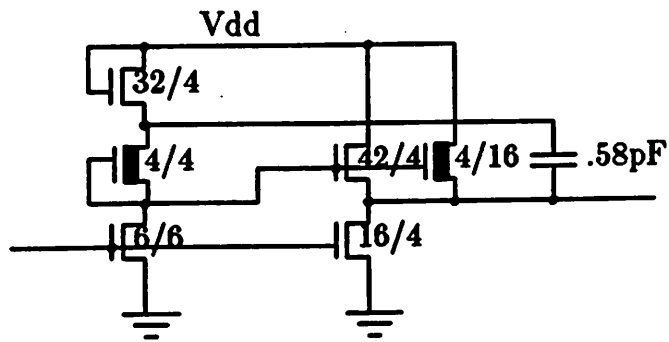


Figure 6.26a- Control Line Inverter

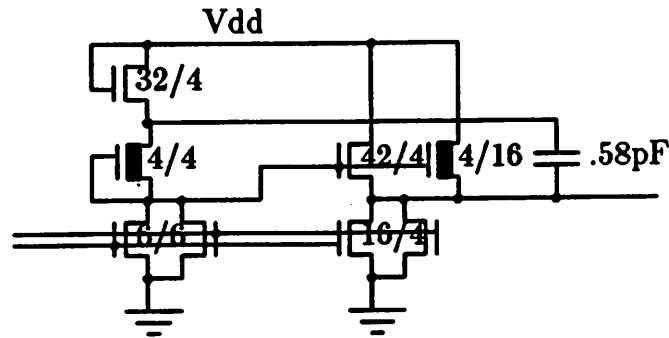


Figure 6.26b- Control Line NOR

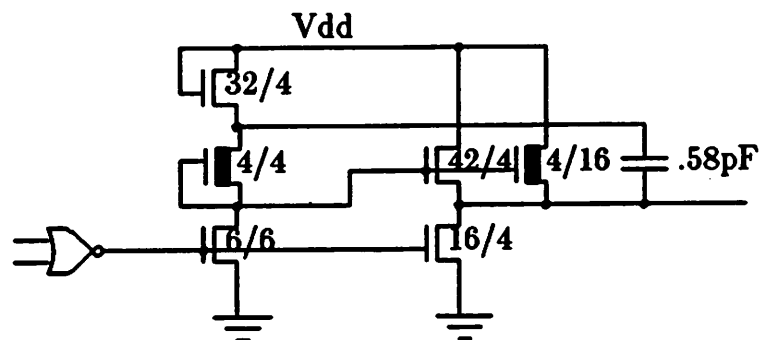


Figure 6.26c- Control Line OR

above  $V_{dd}$ , allowing the output to rise to  $V_{dd}$ . The drawback of these drivers is that they require more area to accommodate the bootstrap capacitor.  $C_b$  must be large enough to quickly supply charge to charge up the gate capacitance of  $M_1$ , and the parasitic capacitances  $C_{p1}$  and  $C_{p2}$ . The diffusion to substrate parasitic capacitance,  $C_{p3}$ , is proportional to the size of  $C_b$  and adds to the load of this driver. A variety of bootstrap drivers were designed to meet the desired driver functions of Table 6.3 (Figure 6.26) [Kong85]. SPICE simulations of the propagation delays as functions of loads for these drivers are shown in Figure 6.27.

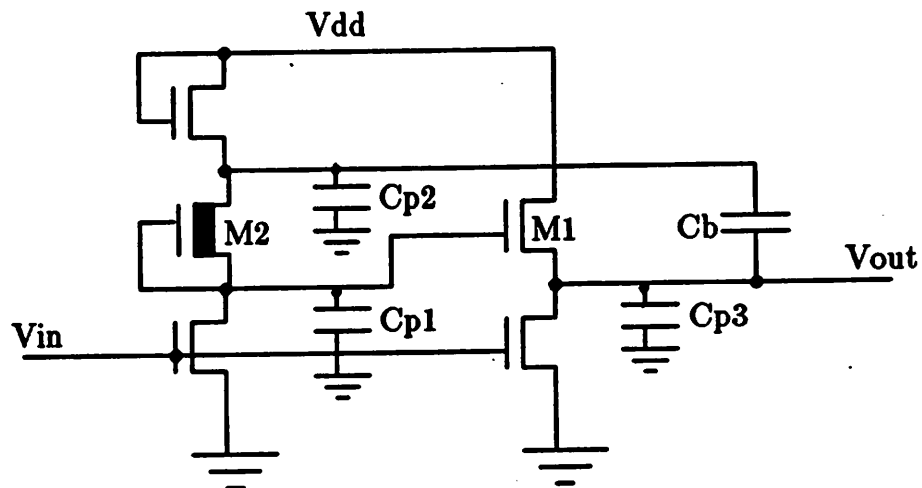


Figure 6.25- Two Stage Bootstrap Driver

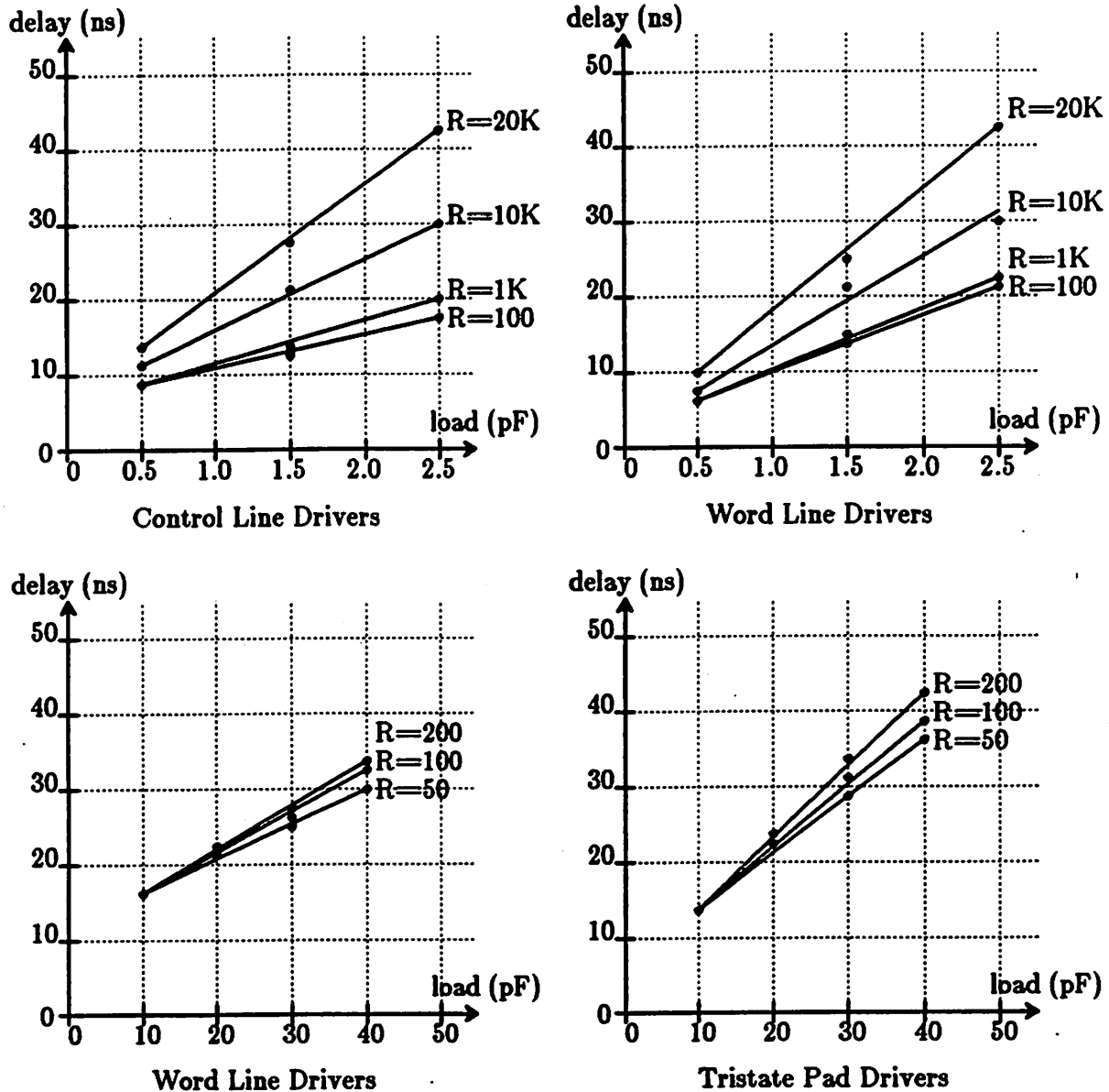


Figure 6.27- Propagation Delays for SOAR's Bootstrap Drivers

### 1.3. Initial Microarchitecture Design

Early in the *preliminary circuit* step the instruction set was analyzed to discover the operations to be performed and the order that they could be performed in. The required operations were used by the circuit design level, as just described. Meanwhile, the coordination of these operations provides a starting point for microarchitecture design. Figure 6.28 shows one possible

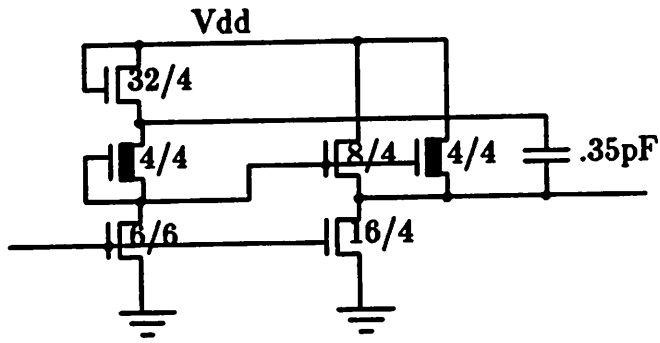


Figure 6.26d- Word Line Inverter

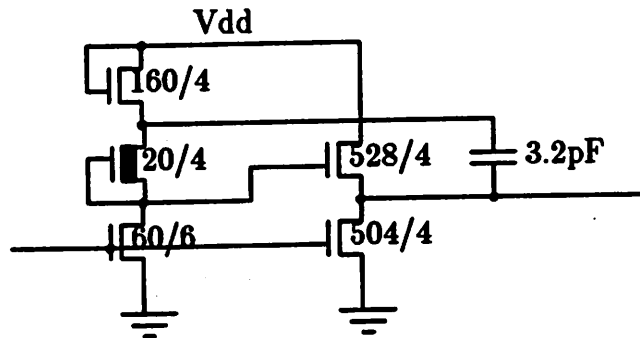


Figure 6.26e- Pad Driver

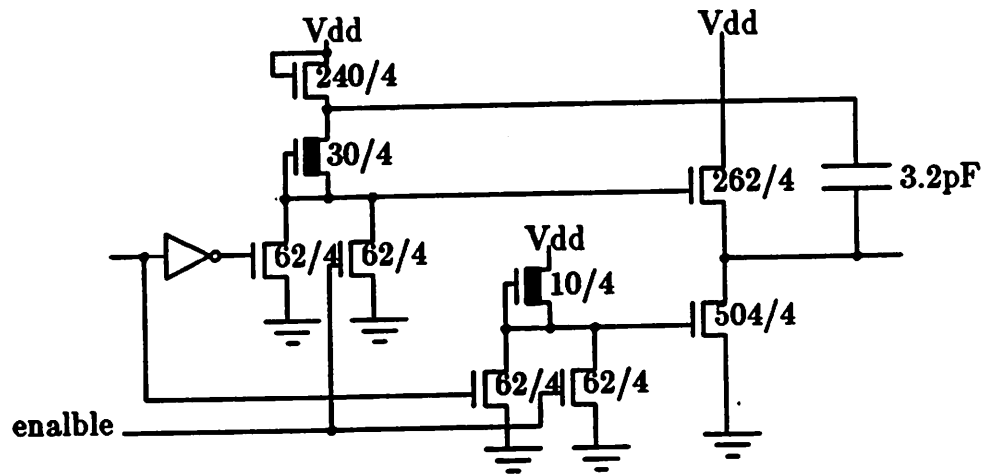


Figure 6.26f- Tri State Pad Driver

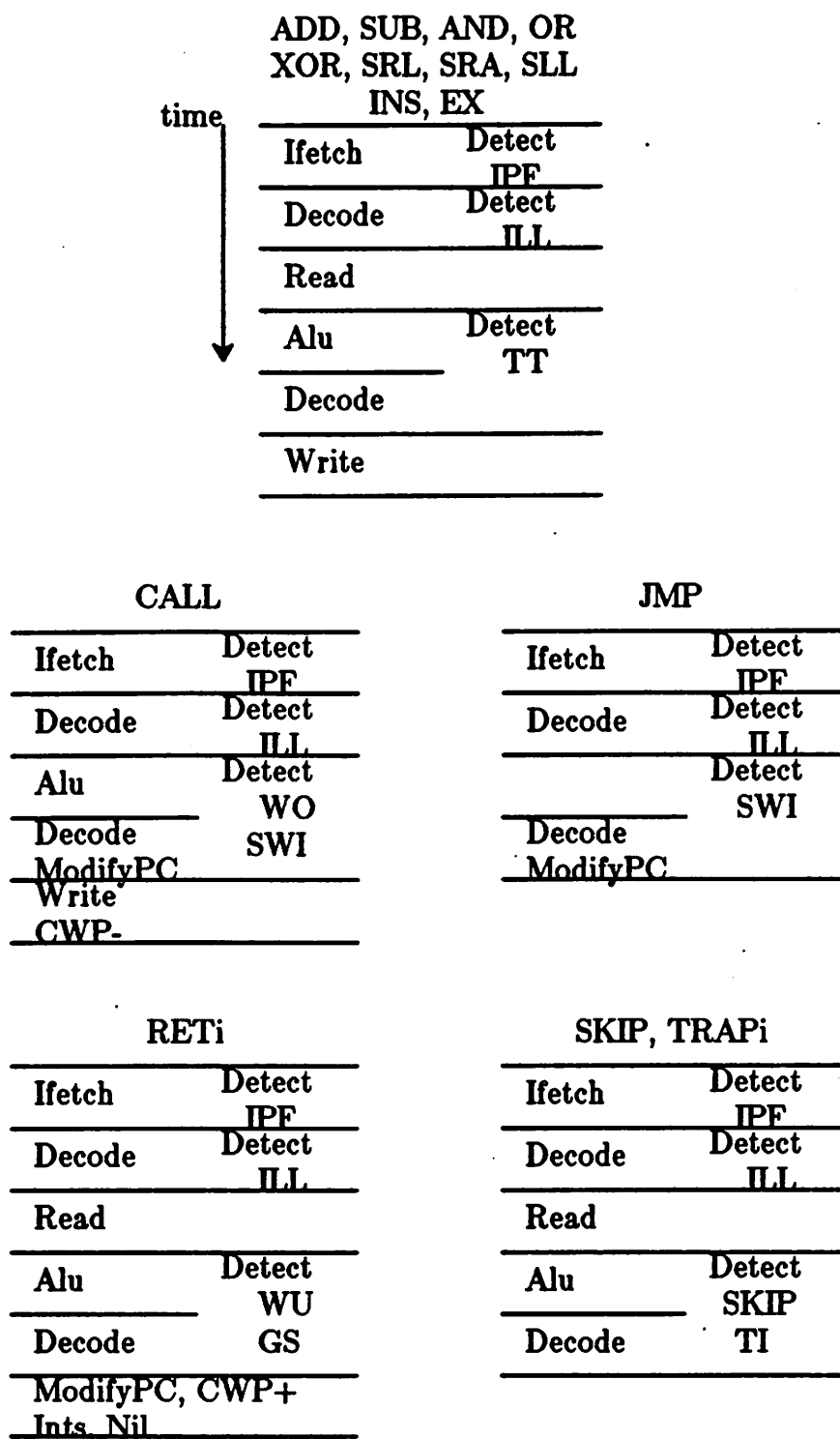


Figure 6.28- Ordering for the Basic Pipeline Functions

ordering for the basic required functions of each instruction type. Trap detection can take place any time after the information that must be examined to determine if a trap situation exists, is available. Another variation, is that decoding could have been done only once for each instruction instead of twice. Thus, many variations are possible. However, certain functions can have only one possible order. For example, Ifetch, decode, read, alu, and write must be ordered:

1. Ifetch
2. Decode
3. Read
4. Alu
5. Write

Address  $\leftarrow$  TB (traps only)

Instruction latch  $\leftarrow$  Memory[PC] then

PC  $\leftarrow$  PC+1

Decode op  $\leftarrow$  \* Instruction latch[opcode field]

S1  $\leftarrow$  \* Instruction latch[1st operand field]

S2  $\leftarrow$  \* Instruction latch[2nd operand field]

D  $\leftarrow$  \* Instruction latch[destination field]

Read ALUinput1  $\leftarrow$  r[S1]

ALUinput2  $\leftarrow$  r[S2] or

ALUinput2  $\leftarrow$  immediate

DataOut  $\leftarrow$  r[S2] (stores only)

Alu ALUoutput  $\leftarrow$  ALUinput1 op ALUinput2

Write r[D]  $\leftarrow$  ALUoutput

r[D]  $\leftarrow$  DataIn (loads only)

r[15]  $\leftarrow$  PC (calls only)

r[7]  $\leftarrow$  PC (traps only)

Compare ConditionValid  $\leftarrow$  \* ALUoutput

DetectTrap TRAP  $\leftarrow$  \* op,r[S1],r[S2],ALUoutput,PSW,CWP,SWP,  
ConditionValid,PageFault,IO

DetectSkip SKIP  $\leftarrow$  \* op, ConditionValid



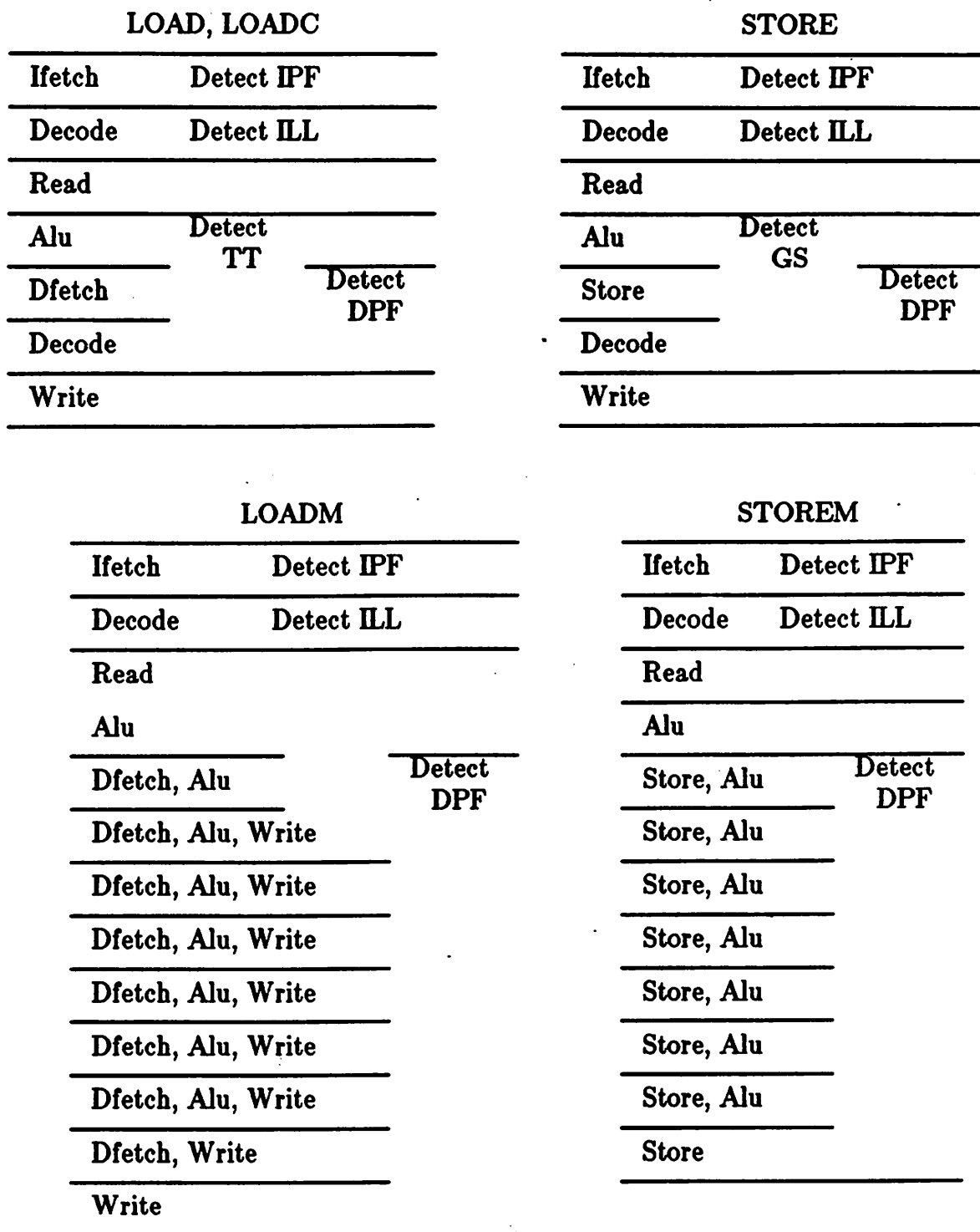


Figure 6.28- Ordering for the Basic Pipeline Functions (cont.)

Each of the pipeline functions of Figure 6.28 may be expressed algorithmically:

Ifetch    Address  $\leftarrow$  PC

when more than one pipeline function occurs simultaneously. For example, trap detection on SOAR can be done in parallel with instruction execution for most instructions. In a pipelined processor several instructions are in various stages of completion at any one time. Each instruction in the pipeline is using a portion of the processor resources, to complete the basic pipeline function that it is processing. When all instructions have finished the individual pipeline functions that they are working on, they synchronously move on to the next pipeline functions and a new instruction is started. Thus, processor resources are more fully utilized and the instruction rate is higher than on a processor that requires an instruction to complete before the next one starts. In order for a processor to be cleanly pipelined, most instructions must be able to execute with the same ordering of functions. In this way resource conflicts between instructions do not exist and the pipeline can run smoothly and continuously. An inspection of the order of functions for the SOAR instruction set (Figure 6.28) reveals the following function sequence to be common for most instructions:

1. Ifetch
2. Decode— for read and alu
3. Read
4. Alu
5. Decode— for write, DetectTrap
6. Write

This ordering was chosen as the basis of the SOAR pipeline.

A second consideration in the pipeline design is the way in which the basic functions of the instructions will overlap. An estimate of the relative speeds of each basic function, in addition to the function order, is needed to propose the overlapping. This is one way that the *circuits available* input influences microarchitecture design. As each of these functions is realized by circuits, the

PC  $\leftarrow$  PC+1

Nil r[0to5]  $\leftarrow$  B0000000 (nil value)

CWP+ CWP  $\leftarrow$  CWP+1

CWP- CWP  $\leftarrow$  CWP-1

Ints PSW<1>  $\leftarrow$  1

modifyPC PC  $\leftarrow$  ALUoutput

Dfetch Address  $\leftarrow$  ALUoutput then  
DataIn  $\leftarrow$  Memory[Address]

Store Address  $\leftarrow$  ALUoutput then  
Memory[Address]  $\leftarrow$  DataOut

Precharge All buses  $\leftarrow$  FFFFFFFF

\*Assigned through random logic

All trap detection has been assigned to the DetectTrap function. Each type of trap examines the relevant inputs on the right side of the TRAP statement and generates a TRAP through random logic.

Once an ordering of the functions for each instruction has been proposed, the pipelining and parallelism of the processor may be proposed. Parallelism arises

(Figure 6.30).

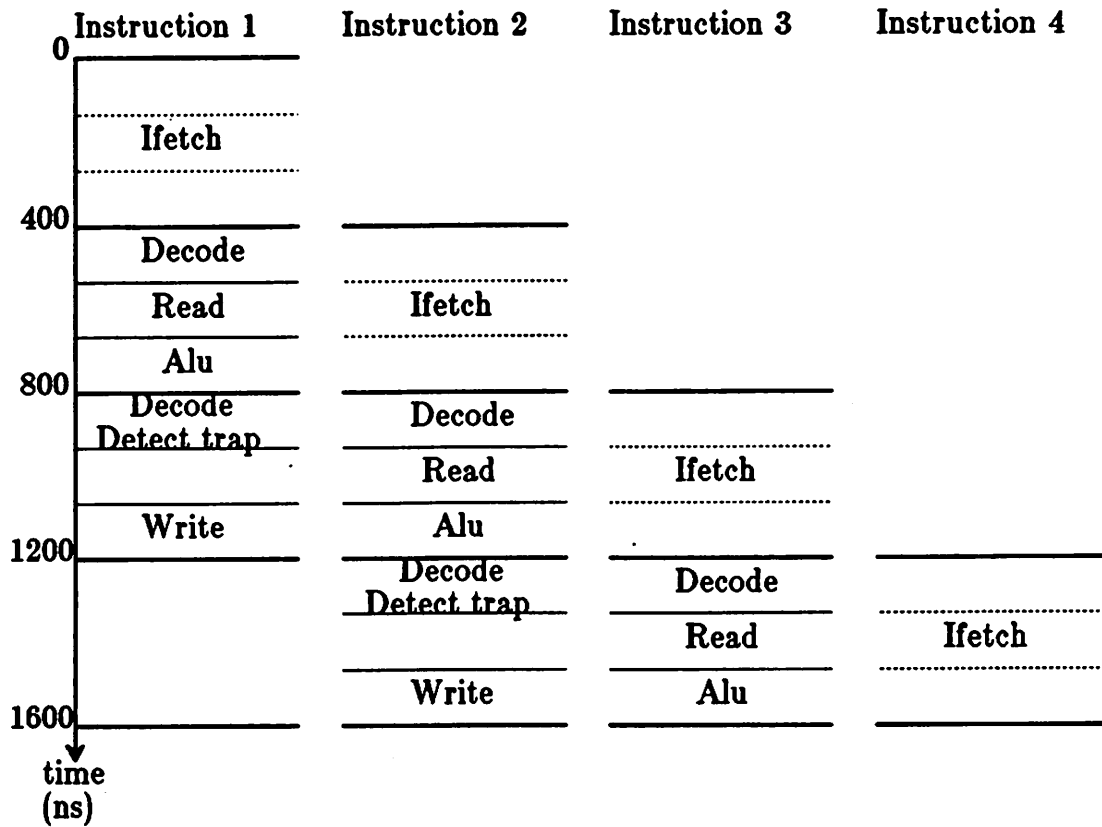


Figure 6.29- Pipeline- Realized SOAR

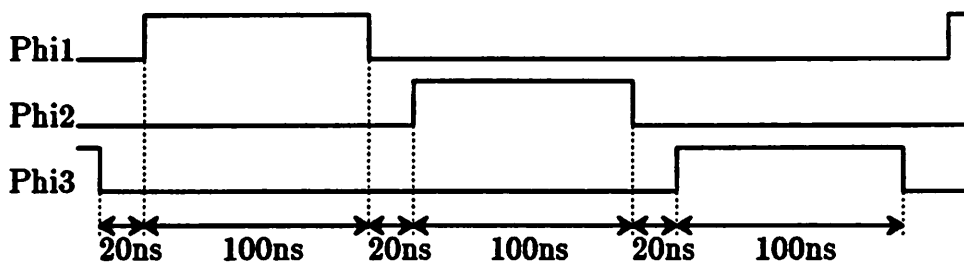


Figure 6.30- Proposed Clocking- Realized SOAR

speeds of these functions can be more accurately determined through speed analysis of the circuits. For SOAR, the overlap of the functions was proposed before the *circuits available* input existed. Rough estimates of the function speed were used to develop the pipeline overlaps (Table 6.6). Fetches were estimated to require four times as much time as any other operation and therefore became the pipeline bottleneck. Each instruction required a fetch since there was only one memory port and no on chip instruction storage. Therefore, instructions could not be started more frequently than once every 400ns.

Function	Speed Estimate	Reason
Ifetch	400ns	system specifications
Precharge	100ns	?
Decode	?	Ignored
Read	100ns	RISC II
ALU	100ns	Siemens paper
Write	100ns	RISC II
DetectTrap	?	Ignored

Table 6.6- Original Speed Estimates

Figure 6.29 shows the assignment of functions to time slots and the overlapping of functions for sequential instructions. Each instruction fetch takes a memory cycle as described in the system specifications (Ch.5, Sec.2.1). Each cycle is then divided into three phases of approximately 100ns each, for decode, read, and the alu operation during cycle 2 of an instruction, and for decode and write during cycle 3. Underlaps of 20ns between phases were proposed for the clocks

schemes during the first cut, as done in SOAR, as long as the designers do not forget that this first cut at timing is only as good as their estimates of circuit speeds and clock requirements.

The second pass at microarchitecture design during the *preliminary circuit* step has a more accurate idea of circuit speed and clocking due to the circuits that are available and their timing characteristics. For the following discussion of the second pass at microarchitecture design speed estimates from the *circuits available* input are used. Table 6.7 summarizes the speed estimates of the circuits described in this chapter, Section 1.2 that contribute to the basic pipeline functions – Ifetch, decode, read, alu, and write.

Circuit	Speed Estimate (ns)
PLA	100-200
Register file cell	20-25
Latch cell	30-45
ALU	100
D1, D2	50-70
Word line driver	20-40
Control line driver	20-40

Table 6.7- Circuit Speed Estimate Summary

Table 6.8 summarizes the times for each of these basic pipeline functions based on speed estimates of the circuits. The Ifetch primarily involves circuits outside the processor. Decode assigns values to control and word lines from the instruction fields, through the PLAs. These values must be driven to the places where they

#### 1.4. Further Preliminary Microarchitecture Design

With a knowledge of the *available circuits and their characteristics* from initial circuit design the microarchitecture design may be further refined intelligently. This microarchitecture design in response to initial circuit design closes the loop in the flow diagram for the *preliminary circuit* step (Figure 6.2a). As indicated by the loop's existence, it is possible to go through a series of iterations in the microarchitecture and circuit design if the original conceptions of design at these two levels are very incompatible. At the other extreme, if the original proposals are totally compatible, the microarchitecture would not need to be refined at all in response to the *circuits available* input from the circuit level. Realistically, most design will require at least one microarchitecture refinement after the available circuits and their characteristics are known. Unfortunately, in the SOAR project this was largely overlooked and would have made a difference in a few key places. Consequences of this were that some optimality was lost and some major redesign was done after most low level design was complete. The time consuming low level redesign time could have been saved and a more optimal processor achieved if the microarchitecture had been refined in response to the circuit design at this stage. Although this was not done, a discussion of how it might have been done for SOAR and future similar processors will be carried out here for the purpose of a complete case study.

#### 1.5. Pipeline

The first cut at the microarchitecture design defined the functions that the circuits must perform and an ordering of these functions. The second part of microarchitecture design involves the coordination of these functions. A knowledge of the speed and clocking of the circuits is needed to smoothly coordinate the functioning of the circuits. It is acceptable to propose timing

Function	Circuits	Circuit Speeds	Total Speed
Ifetch	system	400ns	400ns
Decode	PLAs	100-200ns	150-270ns
	D1, D2	50-70ns	
Read	word line drivers	20-40ns	50-85ns
	register file cell	20-25ns	
	or		
	control line drivers	20-40ns	
	latch cell	30-45ns	
Alu	ALU	100ns	100ns
Write	control line drivers	20-40ns	50-85ns
	latch cell	30-45ns	
DetectTrap	PLAs	100-200ns	100-200ns

Table 6.8- Pipeline Function Speed Estimates

Using these speed estimates and the proposed assignment of phases into cycles it can be seen that the cycle time was to be limited by cycle 2 – decode, read, and alu (Table 6.9a). The cycle time can be estimated to be from 360 to 515ns because of cycle 2. Table 6.10 shows speed estimates for the basic pipeline functions during the first and second passes of microarchitecture design. Decode time was overlooked during the first pass. Knowledge of PLA speeds shows that the decode time would be a significant contributor to cycle time.



are used by drivers such as D1 and D2. For the read function two operand sources are considered - register file cells for the global and local registers, and latch cells for the special registers. A read occurs when either word or control lines enable the operand registers and the registers subsequently drive the buses to the ALU inputs. The alu function involves the execution of the specified operation. Writes are performed when control line drivers enable the output of the ALU output latch onto buses, to be stored in the destination register. The most complex trap detection is done primarily by the random logic of PLAs.

<b>Cycle</b>	<b>Functions</b>	<b>Function Times</b>	<b>Cycle Time</b>
<b>1</b>	<b>Ifetch</b>	<b>400ns</b>	<b>400ns</b>
<b>2</b>	<b>Decode</b>	<b>150-270ns</b>	<b>360-515ns</b>
	<b>Read</b>	<b>50-85ns</b>	
	<b>Alu</b>	<b>100ns</b>	
	<b>Underlaps</b>	<b>60ns</b>	
<b>3</b>	<b>DetectTrap or</b>	<b>100-200ns</b>	<b>260-415ns</b>
	<b>Decode and</b>	<b>150-270ns</b>	
	<b>Write</b>	<b>50-85ns</b>	
	<b>Underlaps</b>	<b>60ns</b>	

**Table 6.9a- Required Cycle Times- Realized SOAR**

With the goal of a shorter cycle time in mind, other cycle and phase assignments can be proposed. In an attempt to have all worst case cycle times no worse than the time for Ifetch - 400ns - the alu function may be moved to cycle 3 (Figure 6.31). The second decode and the alu followed by trap detection functions are done in parallel during the first part of cycle 3. The write function that previously occupied the later part of cycle 3 is moved to the first part of cycle 4 so that it will not conflict with the read at the register file. Each cycle is now composed of two phases (Figure 6.32). Phase 1 is limited by the alu function followed by the trap detection function to be 200 to 300ns and phase 2 by the read that takes 50 to 85ns. The total processor cycle time is now estimated at 290 to 425ns, which is closer to the limiting 400ns memory cycle time (Table 6.9b).

<b>Function</b>	<b>1st Pass Speed Estimate</b>	<b>2nd Pass Speed Estimate</b>
<b>Ifetch</b>	<b>400ns</b>	<b>400ns</b>
<b>Precharge</b>	<b>100ns</b>	<b>60ns</b>
<b>Decode</b>	<b>ignored</b>	<b>150-270ns</b>
<b>Read</b>	<b>100ns</b>	<b>50-85ns</b>
<b>Alu</b>	<b>100ns</b>	<b>100ns</b>
<b>Write</b>	<b>100ns</b>	<b>50-85ns</b>
<b>DetectTrap</b>	<b>ignored</b>	<b>100-200ns</b>

**Table 6.10- Pipeline Function Speeds**

Cycle	Functions	Function Times	Cycle Time
1	Ifetch	400ns	400ns
2	Decode Read Underlaps	150-270ns 50-85ns 40ns	240-395ns
3	Alu and DetectTrap or Decode Underlaps	100ns 100-200ns 150-270ns 40ns	240-340ns
4	Write Underlaps	50-85ns 40ns	90-135ns

Table 6.9b- Required Cycle Times- Optimized SOAR

registers, and interconnect resources such as buses, word lines, and memory ports. Table 6.11 shows the basic pipeline functions, the circuit and interconnects that they use, and their clock phase assignments. This analysis is carried out for the realized and optimized versions of SOAR. Tables 6.12a and b pictorially show the allocation of these resources for both versions of SOAR.

Basic Pipeline Function	Resources	Clock Phase	Clock Phase
		Real SOAR	Optimized SOAR
Ifetch	Address Memory Ports	1	1
	Data Memory Ports	3	2
Decode	PLAs	1	1
Read	Register File	2	2
	Word Lines- read	2	2
	Bit Lines- read	2	2
Alu	ALU	3	1
Write	Register File	3	1
	Word Lines- write	3	1
	Bit Lines- write	3	1
DetectTrap	PLAs	1	1

Table 6.11- Basic Pipeline Resources

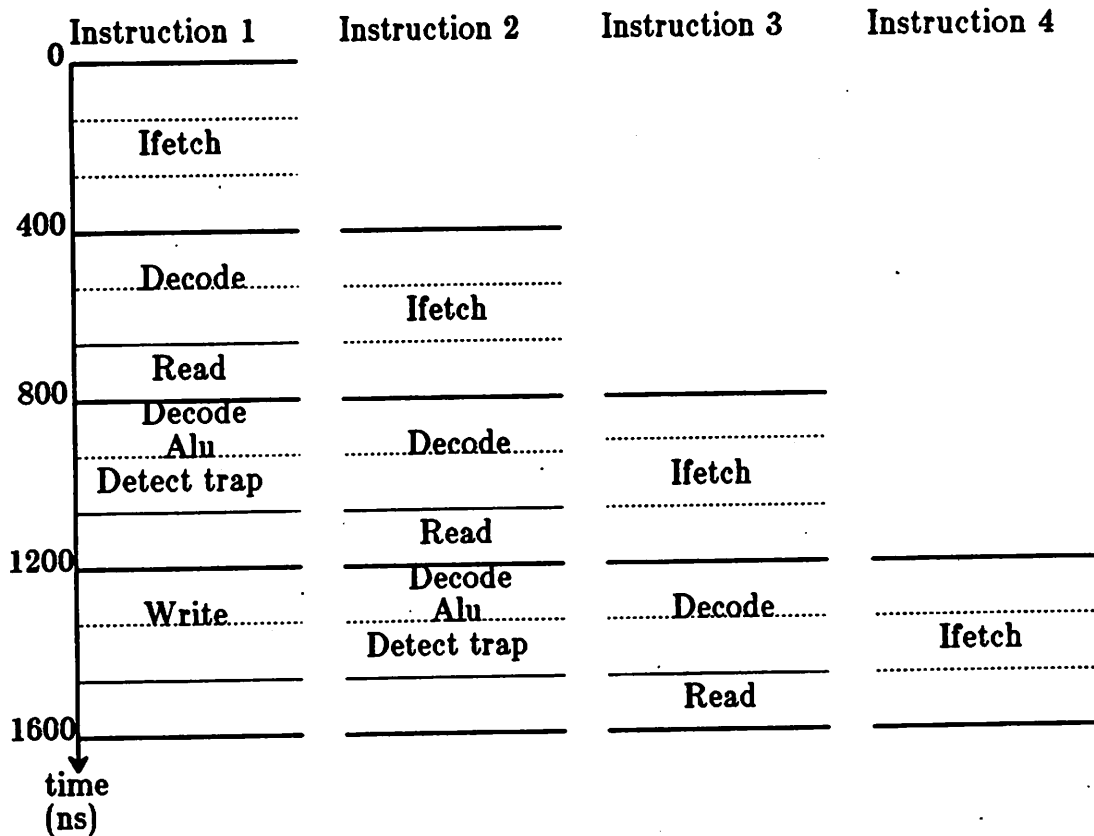


Figure 6.31- Proposed Pipeline- Optimized SOAR

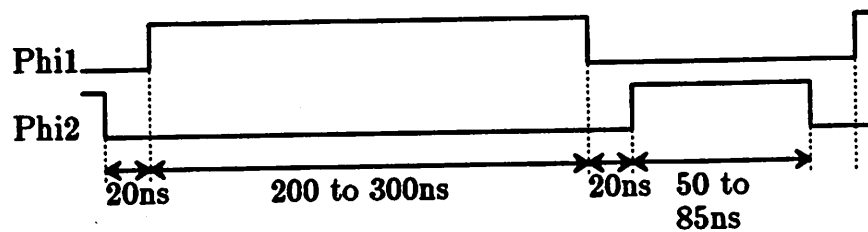


Figure 6.32- Proposed Clocking- Optimized SOAR

### 1.6. Resource Allocation

Once the overlapping of the basic pipeline functions has been determined and a clocking scheme proposed, resources can be allocated to specific time slots or clock phases. Resources to be allocated include circuits such as the ALU and

register file resources have less idle time than other resources. It must be remembered that all speed and clocking estimates are very rough at this point in the design. If future detailed analysis of the register file timing reveals a need for more time, less free time is available and it may be difficult to accommodate the register file. Therefore, a more detailed analysis of the register file resources should be done at this stage in the design. A detailed timing analysis of the register file resources requires a knowledge of the register file circuit (Figure 6.33). Table 6.13 shows the order of events for the register file and the operation of the word and bit lines. A read requires that the bitlines be precharged initially. This is because the small depletion mode pullup transistors of the register file cell are too weak to quickly pull the bit lines high. The larger pulldowns can quickly pull the bit line low during a read. During precharge the word lines must be disabled to prevent the stored data from being destroyed. When reading from the register file the appropriate word line is driven high and bit lines are selectively discharged according to the data in the register being read. Word lines must then be disabled before writing to prevent a false write to the register that has just been read. When writing to the register file word lines are enabled and large drivers are used to overpower the inverters of the cells being written to.



Resource	Phase 1	Phase 2	Phase 3
Address Ports	X		
Data Ports			X
PLAs	X		
Register File		X	X
Word Lines		X	X
Bit Lines		X	X
ALU			X

Table 6.12a- Resource Allocation- Real SOAR

Resource	Phase 1	Phase 2
Address Ports	X	
Data Ports		X
PLAs	X	
Register File	X	X
Word Lines	X	X
Bit Lines	X	X
ALU	X	

Table 6.12b- Resource Allocation- Optimized SOAR

From these tables it can easily be seen that all major resources, except the register file resources, are used only once each cycle. The register file resources, including word and bit lines are the most heavily utilized resources – twice each cycle. The

are compatible with the proposed clocking schemes. The realized scheme has phase 1 available for precharge while the optimized version must use the 1/2 underlap for both word line disabling and precharging. However, the optimized version has less resource idle time. The optimized design will have a faster cycle time due to a more complete utilization of the resources. In particular 100% utilization is planned for the most heavily used resource – the register file.

Step	Real SOAR Clock Phase	Optimized SOAR Clock Phase
Precharge	1	1-2 underlap
Read	2	2
Disable	2-3 underlap	2-1 underlap
Write	3	1
Disable	3-1 underlap	1-2 underlap

Table 6.14- Register File Clock Phase Assignments

A final detailed analysis of the register file timing compares the times required by the register file steps with proposed clock phase lengths for the optimized SOAR (Table 6.15). The read and first disable are compatible with their clock phase lengths. However, phase 1 is much longer than necessary for the write, and the following underlap is too short for the disable and precharge. Further optimization requires the balancing of these discrepancies. Phase 1 can be split into  $\text{ph1lw}$  and  $\text{ph1p}$  for the write and precharge operations respectively (Figure 6.34). Other operations assigned to phase 1 can span  $\text{ph1lw}$  and  $\text{ph1p}$  (Figure 6.35). The total time required by  $\text{ph1lw}$  and  $\text{ph1p}$  for the register

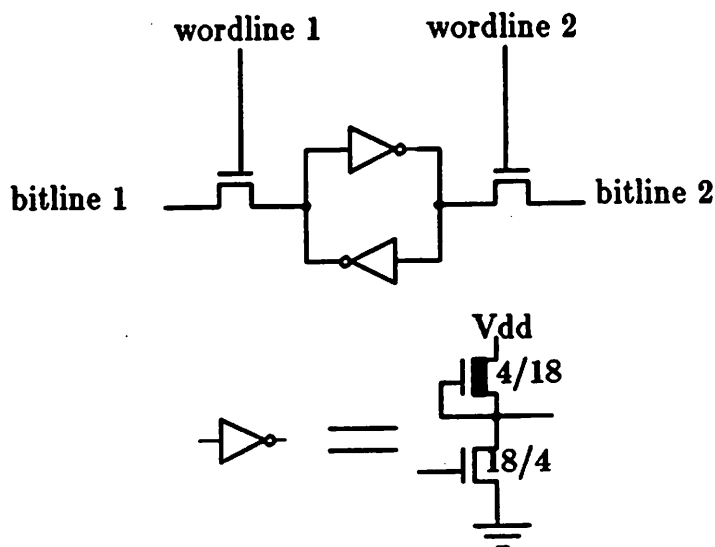


Figure 6.33- Register File Memory Cell

Event	Word Lines	Bit Lines
Precharge	disabled- low	drive high
Read	enable- drive high	selectively discharge
Disable	disable- drive low	idle
Write	enable- drive high	drive with data
Disable	disable- drive low	idle

Table 6.13- Register File Operation

A complete read/write cycle is inherently a five step process with this register file cell. These steps must be assigned to the processor clock phases and register file circuit design must compatibly provide the required functioning. Clock phase assignments for these steps are shown in Table 6.14 for both the real and optimized SOAR. Disabling is done during the underlap time between the clock phases. Referring back to Tables 6.12a and b, it can be seen that both versions

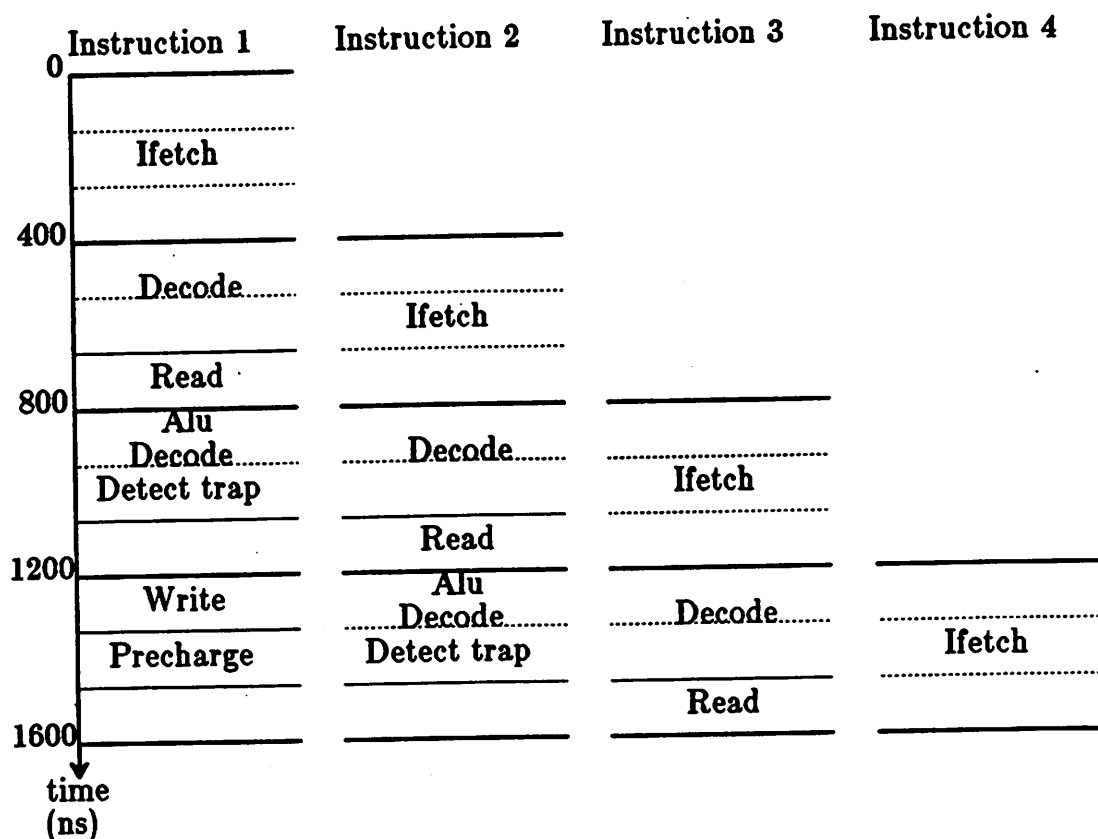


Figure 6.35- Pipeline-Optimized SOAR

### 1.7. Pipeline Exceptions

When designing the pipeline, a sequence of basic functions was chosen that would accommodate most instructions. On SOAR all arithmetic, logical, shift, and byte instructions could be handled by the proposed sequence easily (Figure 6.28). Returns, call, and jumps fit the proposed sequence with minor modifications (Figure 6.36). For these three types of instructions, the alu function is used to compute the target instruction's address. The time slot after the ALU computation and before the write is used to modify the program counter using the result of the target address computation. Nilling is a simultaneous write operation to multiple registers and therefore is assigned to the write time slot. Changing the CWP can be assigned to the read or alu time slot for both calls and

operations is 130 to 165ns. Thus, decode will still limit the total phi1 time with its requirement of 150 to 270ns. Adding this extra clock phase allows the register file disable and precharge steps to be moved into the register file idle time following a write, that is required by the slower decode function.

Step	Required Time	Proposed Phase
Read	50-85ns	2 (50-85ns)
Disable	20ns	2-1 underlap (20ns)
Write	50-85ns	1 (150-270ns)
Disable	20ns	1-2 underlap (20ns)
Precharge	60ns	1-2 underlap (20ns)

Table 6.15- Register Operations and Timing

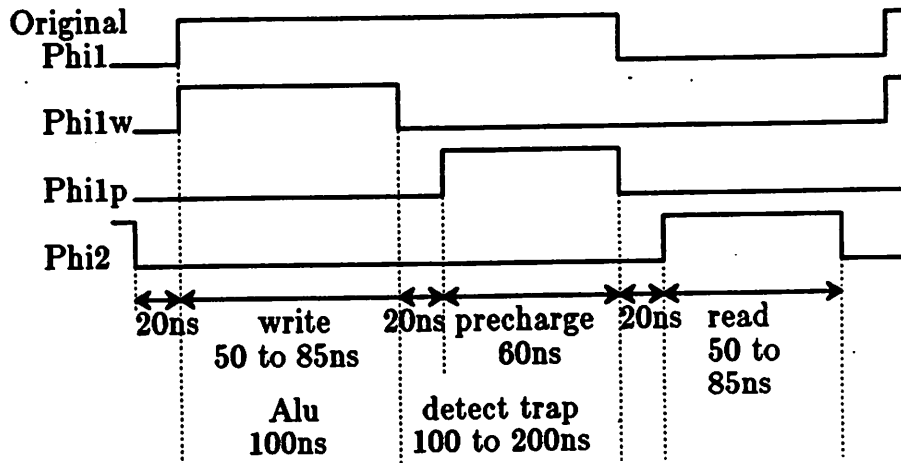


Figure 6.34- Clocking- Optimized SOAR

Load, loadc, and store require

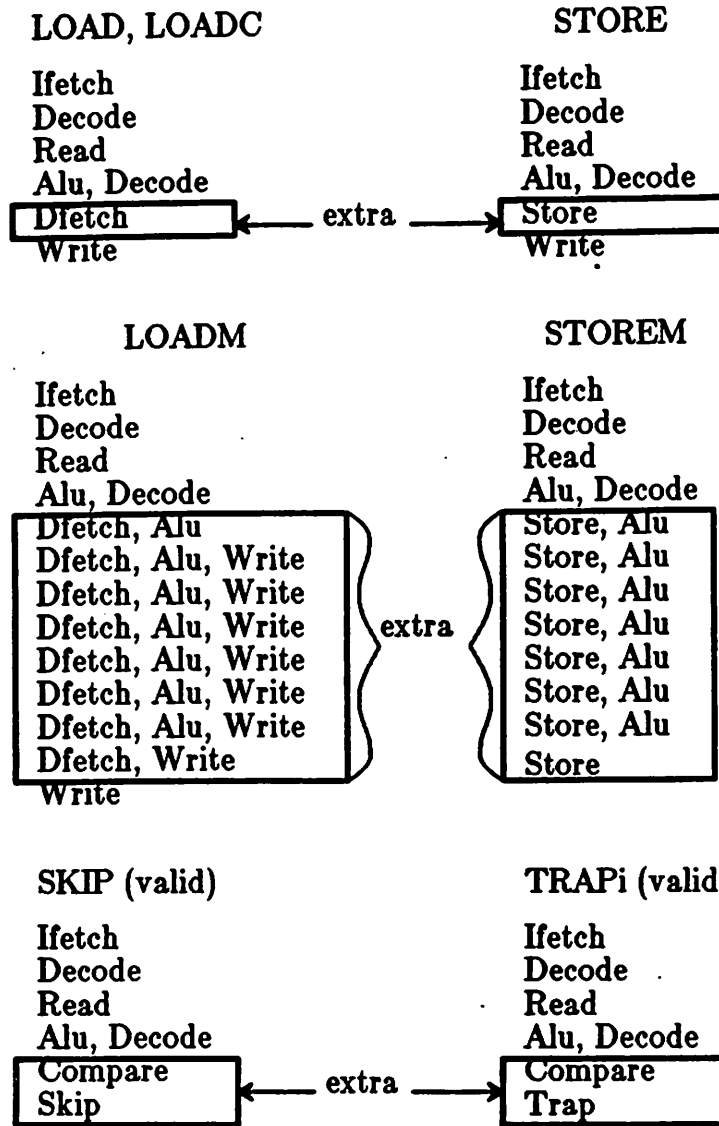


Figure 6.37- Instructions Requiring Extra Time Slots

one extra memory access cycle for their pipeline functions – Dfetch and store. Load multiple and store multiple perform up to eight loads or stores and therefore need up to eight extra memory access cycles. The memory is 100% utilized so each extra memory cycle adds a cycle to the time needed to complete the instruction. All skip and trap instructions require a compare operation. When

returns. Thus calls, returns, and jumps can easily fit into the proposed pipeline and need no extra cycles for their completion.

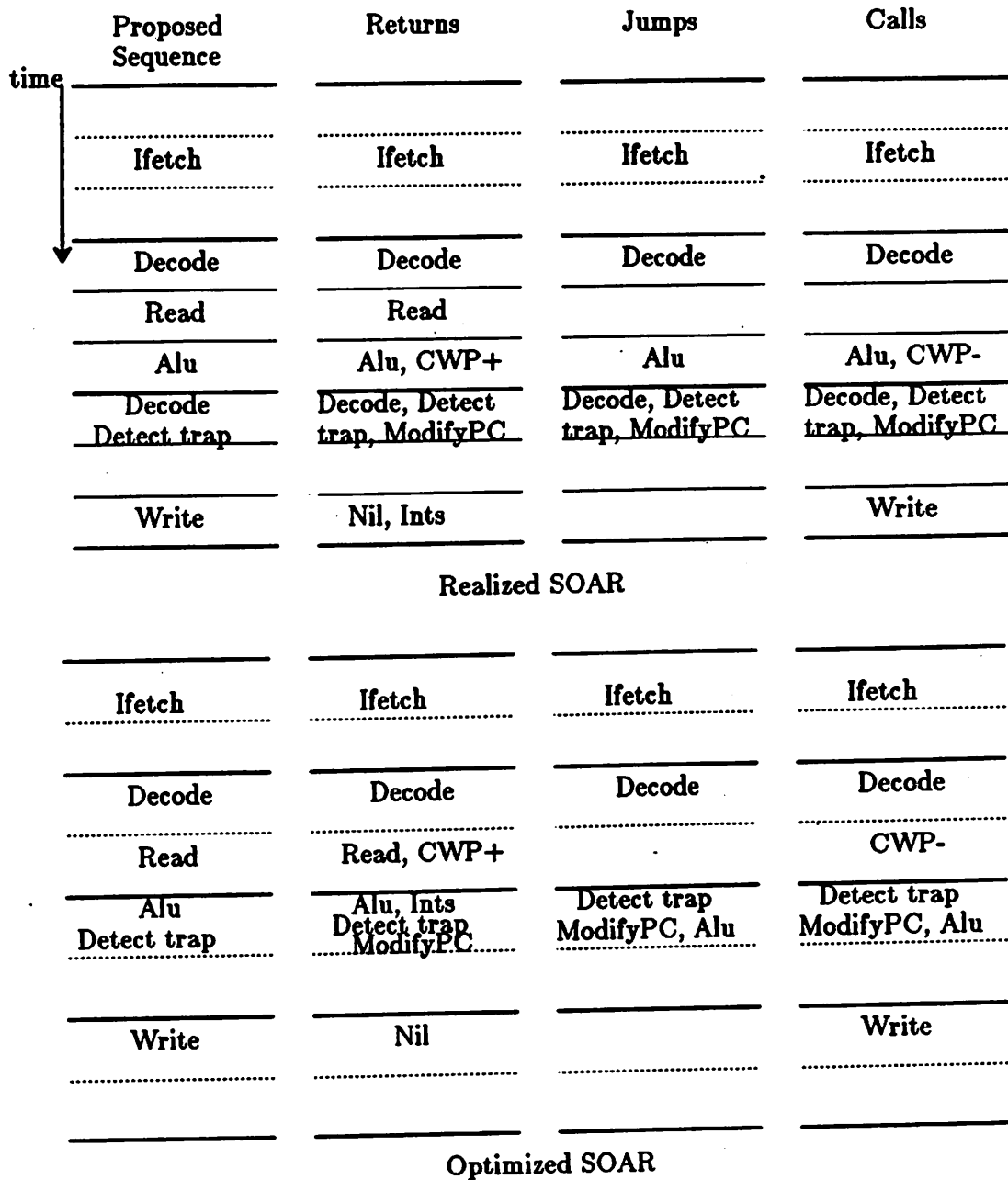


Figure 6.36- Returns, Calls, and Jumps in the Proposed Pipelines

In contrast to this are skips, trap instructions, loads, and stores. They require extra pipeline functions and therefore need extra time slots (Figure 6.37).

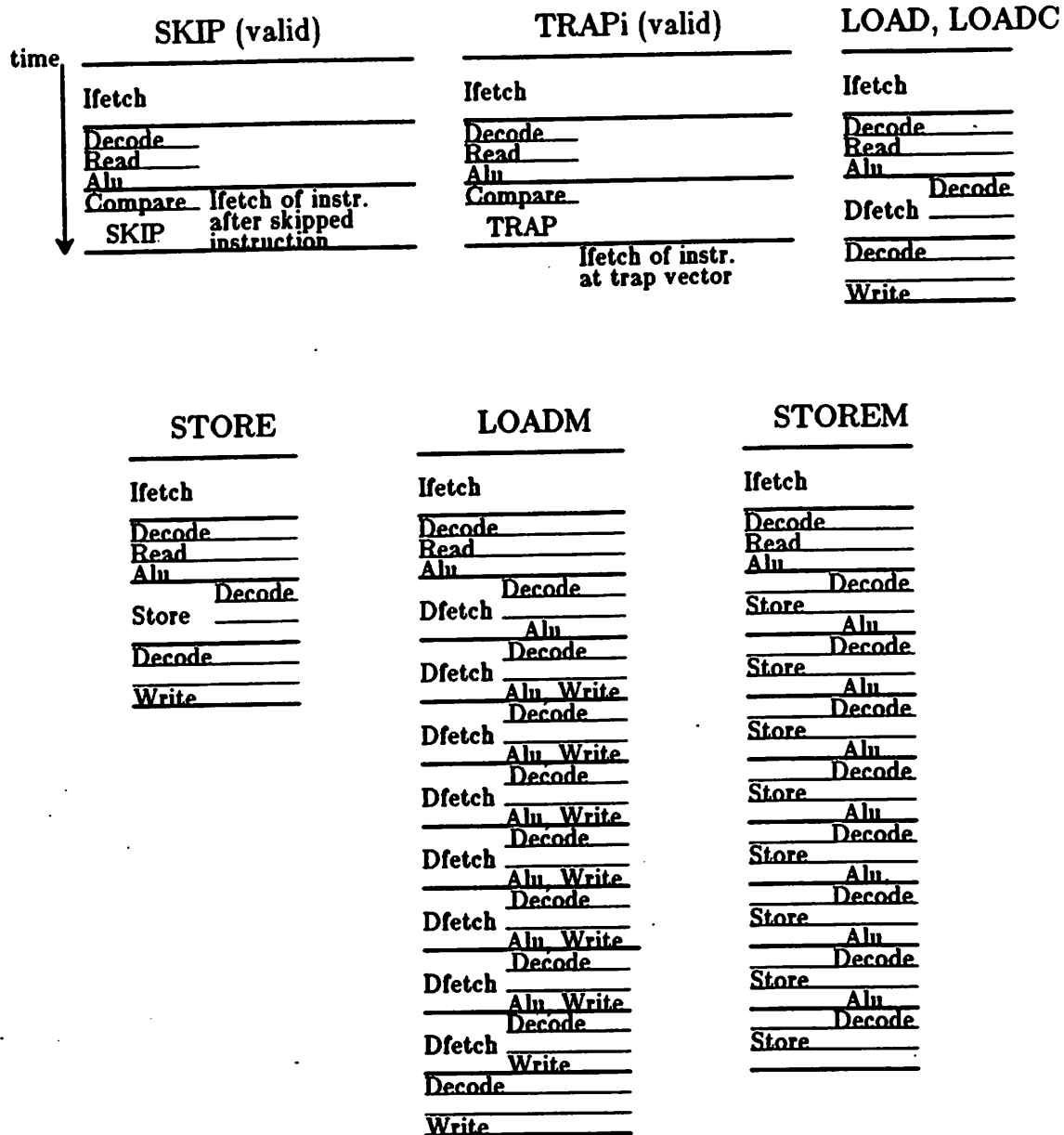


Figure 6.38- Multicycle and Conditional Instructions

At this point in the microarchitecture design, schemes are proposed to handle the instructions that require extra cycles [Pend84]. It is desirable to handle these more complex situations with the same pipeline and decoding mechanisms that will be used for the simpler instructions. In this way extra hardware and complexity are kept at a minimum. To do this, these multicycle instructions are broken into a series of single cycle opcodes. Each single cycle opcode can be



the results of this compare are true extra cycles are needed to complete the skip or trap. A skip requires one extra cycle – the cycle of the instruction to be skipped. Traps require two extra cycles due to the compare, insertion of the trap vector into the memory address latch, and fetching of the instruction at the vector address. Cycle and clock phase assignments for these instructions are shown in Figure 6.38 for the realized SOAR. Analogous assignments may be made for the optimized SOAR.

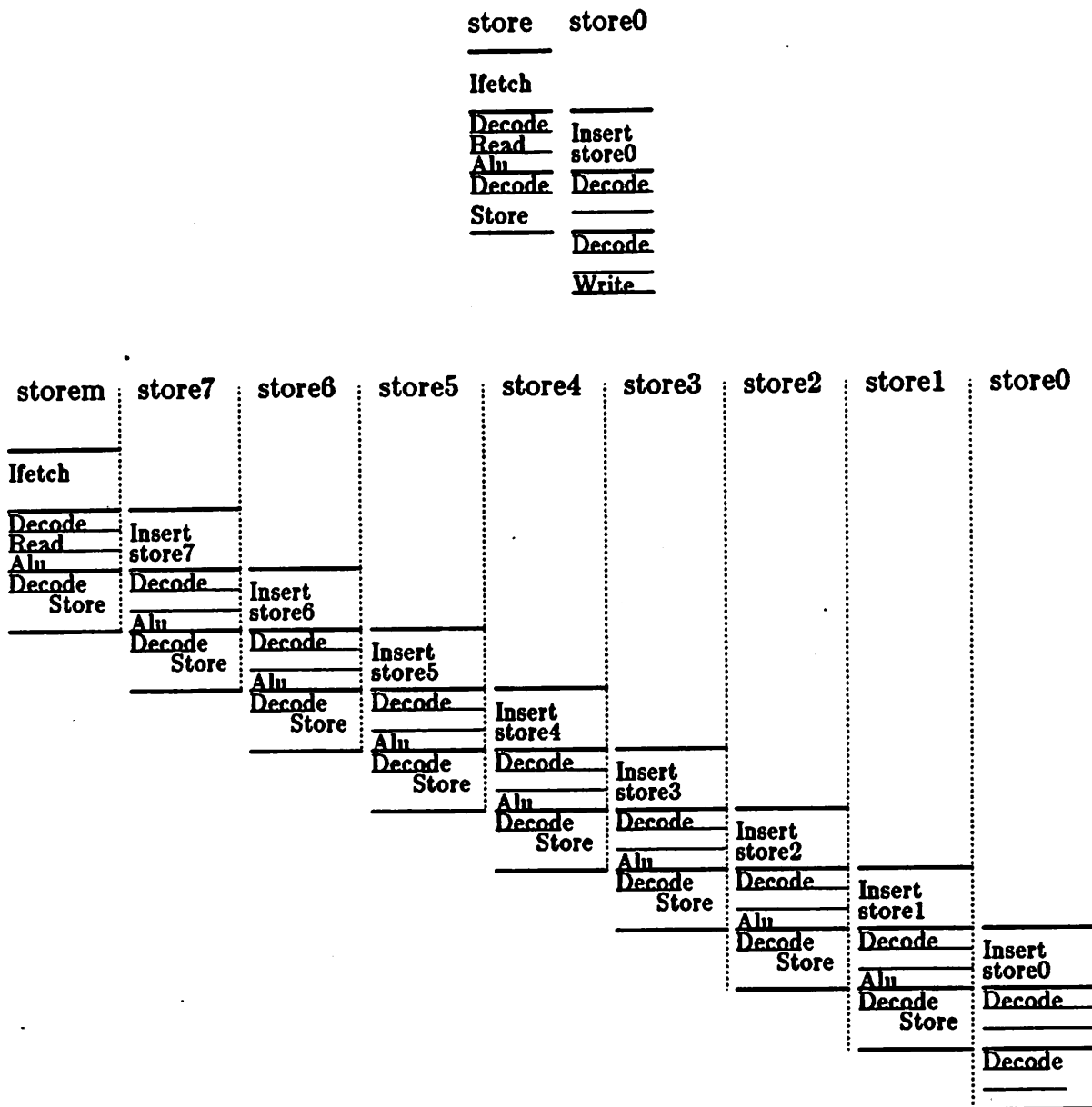


Figure 6.39- Internal Opcodes for Multicycle Instructions (cont.)

Load and loadc cause the insertion of the load0 opcode into the pipeline. This insertion is analogous to an instruction fetch except that load0 is generated on the processor. Load0 sets control lines through the standard decoding mechanisms, for the later parts of the load instruction. Both load and load0 require the same number of cycles as the standard instructions. Any nonstandard pipeline functions that they require, such as Dfetch, are compatible with the

accommodated by the basic pipeline functions. Thus, pipeline flow is uninterrupted in these situations. Resource allocation for multicycle instructions will fit into the same scheme as for the simpler one cycle instructions (Figure 6.39).

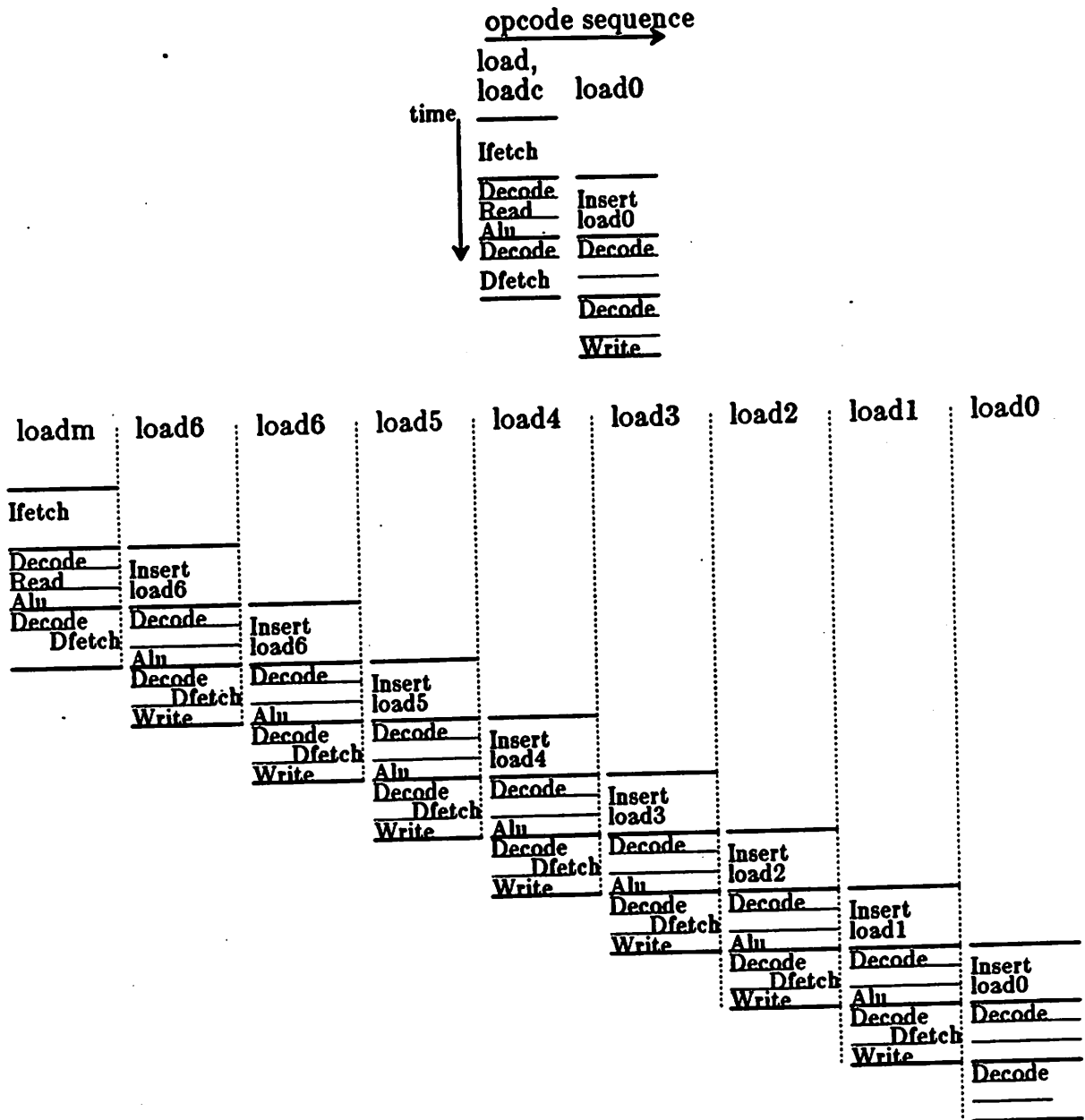


Figure 6.39- Internal Opcodes for Multicycle Instructions

pipelining of the standard functions. Stores are handled in a similar manner to loads by the store, store0 instruction sequence. The fetched store instruction causes the internally generated store0 opcode to be inserted into the pipeline.

The SOAR architecture also called for more complex multicycle instructions – load multiple and store multiple. These instructions provide for the loading or storing of up to eight regularly spaced memory locations. One extra cycle is required for each load or store. Therefore, a loadm or storem generates an internal opcode for each load or store. Each of these opcodes is accommodated by the standard pipeline and causes the insertion of the next internal opcode or an instruction fetch when the last load or store has been completed.

In addition to multicycle instructions, traps and skips that arise should fit into the standard pipeline (Figure 6.40). The compare function uses the ALU result to determine if a trap or skip should be taken. If it is to be taken an internal opcode is inserted into the pipeline – SKIP, TRAP, and NOP. The inserted opcode is decoded and control lines are set so that the skip or trap is taken.

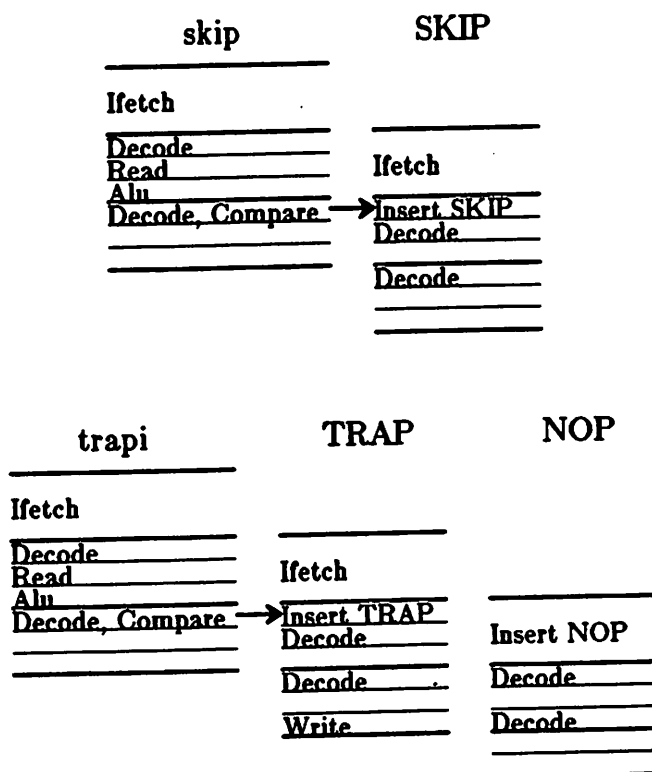


Figure 6.40- Internal Opcodes for Conditional Instructions

### 1.8. Preliminary Circuit Summary

The *preliminary circuit* step for SOAR starts with an examination of the instruction set. The operations needed to perform each instruction are listed. These operations become the *desired functions* input for circuit design. The goal of preliminary design at the circuit level is to satisfy the desired functions and characterize the circuits' timing so that the behavior aspect of microarchitecture design has an estimate of the speeds and clocking requirements of the component circuits.

A second aspect of the instruction set examination is to describe each instruction algorithmically. Each algorithmic statement corresponds to a pipeline function. Some of these pipeline functions for a given instruction may occur simultaneously, others must fall into a sequence for correct completion of the

instruction. Thus, several possible orderings for the pipeline functions of a given instruction may exist.

Once the pipeline functions and potential sequences have been identified for each instruction, the parallelism and pipelining of the processor is proposed. Pipelining evolves from the recognition of sequences of pipeline functions that are common to most instruction. Parallelism emerges from the groups of pipeline functions that may be done simultaneously for any single instruction and from the overlapping of sequential instructions.

The pipeline is composed of a sequence of pipeline functions that is common to most instructions. Once such a sequence has been proposed the overlapping of these basic pipeline functions for sequential instructions must be determined. To do this, an estimate of the time required by each basic pipeline function is needed. This time estimate is obtained by knowing the circuits that are needed for each function and their speeds. This is where the *available circuits* and their characteristics make a significant contribution to the microarchitecture design. The basic pipeline functions of an instruction are organized into sequential groups that all require about the same amount of time to complete. This time is the processor cycle time. It is desirable to keep the cycle time short so that a high instruction rate may be attained. This implies that the pipeline function groups should be small. Therefore, the minimum cycle time is limited by the longest basic pipeline function. If one basic pipeline function is much longer than any of the others and appears to significantly extend the cycle time it may be desirable to try to pipeline that function.

The organization of the basic pipeline functions into groups allows each pipeline function to be assigned to a time slot – a designated cycle and clock phase within that cycle. Knowing the circuits that are used by each function, the circuits and interconnects are also assigned time slots for their use. This is known

as resource allocation. Some resources, such as the register file on SOAR, may be used by more than one function. Thus, the resources are examined and it is noted when each one is busy. In this way conflicts may be discovered and avoided when more than one function uses the same resource. This analysis also exposes another basic limit to the cycle time - fully utilized resources. During any given machine cycle all basic pipeline functions are performed - not for a single instruction, but for all instructions that are in various stages of completion. Therefore, all functions that use a resource must fit their use of that resource into a single cycle and the total time that a resource is busy due to the requirements of all functions becomes a limit on the minimum cycle time. Thus, when a resource is fully utilized - busy 100% of the time - a minimum cycle time has been achieved. If it is desirable to reduce the cycle time from this, the fully utilized resource must be duplicated in some way to accommodate the requirements of all the pipeline functions.

Once the resources with the heaviest use have been identified, their speeds and clocking requirements are examined in detail to ensure that they are compatible with the proposed clocking. To attain a minimum cycle time the resource with the heaviest utilization must be used 100% of the time. Modifications can be made at this point if they are possible, to bring this utilization closer to 100%. However, the accuracy of speed estimates should be taken into consideration so that speed requirements are not made unrealistically tight at this point.

Lastly, instructions and situations that do not automatically fit into the basic pipeline must be provided for. It is desirable to accommodate these situations without any unnecessary hardware complexity or cycle time increases. The concept of internal opcodes allows multicycle instructions and special conditions to be handled within the basic pipeline and resource allocation framework. In this

way the pipeline operates continuously and smoothly.

## 2. Preliminary Interconnect

SOAR was to be implemented in a technology that had one level of metal. Therefore, other interconnect layers were needed. Both polysilicon and diffusion were considered. Polysilicon was chosen due to its lower capacitance and RC time constant, allowing for faster signal propagation than in diffusion. Table 6.16 shows resistance and capacitance contributions of a minimum size square of interconnect made from each layer. Polysilicon has significantly less capacitance and a lower overall time constant than diffusion. Thus, polysilicon is faster when the interconnect speed is limited by the capacitive loading or the time constant.

Layer :	R	Area C	Perimeter C	Total C	RC
Polysilicon	50	.96fF	0	.96fF	48fs
Diffusion	20	2.56fF	5.60fF	8.16fF	163fs

Table 6.16- Minimum Square ( $4\mu\times 4\mu$ ) Resistances and Capacitances

Interconnects in the datapath consisted of control and word lines crossed at right angles by the data buses. Metal was chosen for the data buses and polysilicon for the control and word lines. Polysilicon control and word lines cross the 32 bit datapath. The length of these lines contributes a high resistance and capacitance to their time constants. The propagation delays due to the time constants are significant. Consultation with the circuit design reveals the dimensions of these lines for one bit -  $88\mu\times 4\mu$ . Both resistance and capacitance are proportional to the total length. Thus, the limiting RC time constant is



proportional to the square of the length. At this point it was proposed to split the datapath into two 16 bit halves, reducing the RC time constant by a factor of four (Table 6.17). Load capacitances for the control and word lines are due to one minimum size,  $4\mu\text{x} 4\mu$ , gate per bit and  $12\mu\text{x} 4\mu$  gate per bit respectively. A register file read was modeled and SPICE simulations for the word line and bit line speeds were carried out (Figure 6.41, Table 6.18).

	R	Cparasitic	Cload	Ctotal	RC
32 bits	35.2K	.68pF	.63pF	1.31pF	46.1ns
16 bits	17.6K	.34pF	.32pF	.66pF	11.5ns

Table 6.17a- Word Line Time Constants

	R	Cparasitic	Cload	Ctotal	RC
32 bits	35.2K	.68pF	.21pF	.89pF	31.3ns
16 bits	17.6K	.34pF	.11pF	.44pF	7.8ns

Table 6.17b- Control Line Time Constants

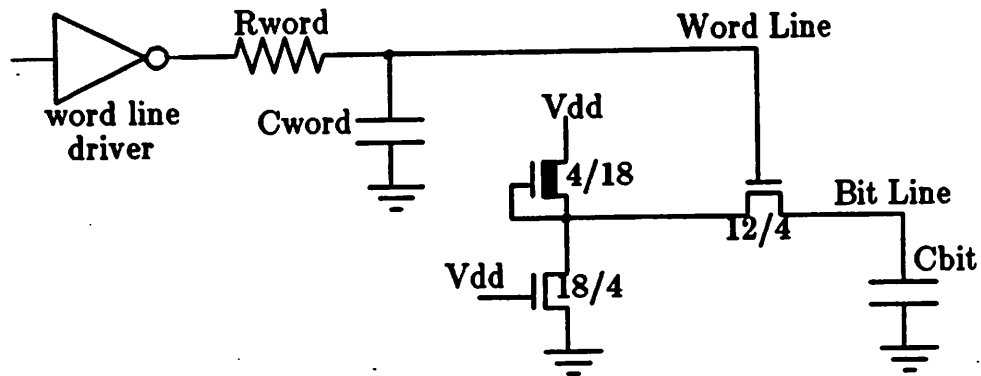


Figure 6.41- Circuit Model for Register File Read

	Rword	Cword	Cbit	Word Line Propagation Delay	Bit Line Propagation Delay
32 bits	35.2K	1.31pF	1.5pF	38ns	56ns
16bits	17.6K	.66pF	1.5pF	13ns	30ns

Table 6.18- Read Propagation Delays

The propagation delay for the word lines of the split datapath was one third of the propagation delay of the full 32 bit datapath. Consequently, bit line delays for reads were estimated to be twice as fast with the split datapath according to simulations.

Splitting the datapath means that a few signals that are critical to the processor cycle time, such as the ALU carry line, have to connect between the two halves of the datapath. These signals, were identified, and are routed solely in metal.

Finally, a preliminary pinout was proposed. System specifications called for separate data and address pads. Therefore, it was proposed that data pads would

be on one side of the chip and address pads on the opposite side. This seemed to lead to a simpler routing problem.

As a result of the *preliminary interconnect* step the SOAR datapath was split into two 16 bit halves. Drivers and decoders are duplicated for each half. Control hardware is placed between the two datapath halves.

### 3. Preliminary Compare

The *preliminary compare* step for SOAR involved comparing the microarchitecture design from the *preliminary circuit* step with the simulated speeds from the *preliminary interconnect* step. Interconnect design estimated the speed of a read with the split datapath. A 30ns propagation delay from the word line driver input to bit line discharge was estimated. This falls within the 50-85ns read speed range that was used for microarchitecture design in the *preliminary circuit* step.

Interconnect design also analyzed the time needed to enable control lines. A 12ns propagation delay was predicted for this. This is compatible with the 20-40ns control line driver speed estimate that was used for microarchitecture design in the *preliminary circuit* step.

No contradictions in the microarchitecture design were discovered during the *preliminary compare* step. This concludes the preliminary design phase for SOAR. Processor design now moves to the alternating synthesis and analysis steps.

### 4. References

[Bose83] Bose, B. K.; Mattausch, H.; Schallenberger, B.; 'VLSI Design Consideration for SOAR', (Unpublished) Proceedings of 292R- Smalltalk on a RISC Architectural Investigations, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., April 1983.

- [Kong85] Kong, S. I., 'Some Design Techniques for High Performance MOS Circuits', M.S. Report, EECS Dept., University of California, Berkeley, Ca., January, 1985.
- [Laru83] Larus, J. R.; 'SPLAT: A Tool for Automatically Extracting PLAs', Final Report CS292X, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., May 1983.
- [Ober85] Obermeier, F.; Katz, R.; 'PLA Driver Selection: An Analytic Approach', Proceedings of the 22nd Design Automation Conference, Las Vegas, June 1985.
- [Pomp82] Pomper, M.; Beifuss, K.; Horninger, K.; Kaschte, W.; 'A 32 bit Execution Unit in Advanced NMOS Technology', IEEE J. Solid State Circuits, V.17, N.3, June 1982.
- [Samp85] Samples, A. D.; Klein, M.; Foley, P.; 'SOAR Architecture', T.R. UCB/CSD/85/226, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., March 1985.
- [Scot85] Scott, W.; Hamachi, G.; Ousterhout, J.; Mayo, R.; ed. '1985 VLSI Tools: More Works by the Original Artists', T.R. UCB/CSD/85/225, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., Feb. 1985.
- [Sher84a] Sherburne, R. W.; 'Processor Design Tradeoffs in VLSI', Ph.D. Thesis, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., April 1984.
- [Sher84b] Sherburne, R.; Katevenis, M.; Patterson, D.; Sequin, C.; 'A 32bit NMOS Microprocessor with a Large Register File', Digest of Technical Papers, IEEE International Solid State Circuits Conference, San Francisco, Feb. 1984.
- [VanD82] VanDyke, K. S.; 'SLANG a Logic Simulation Language', M.S. Report, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., June 1982.
- [Whal84] Whalen, S. P.; 'CMOS Adder Designs for High Performance Microprocessors', M.S. Report, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., August 1984.

## Chapter 7

### Microarchitecture Design

#### SOAR Case Study

Microarchitecture synthesis and analysis begin after the preliminary steps have been completed, as discussed in Chapter 4, Section 5 (Figure 4.18). Figure 7.1 shows the portion of the methodology flowchart that corresponds to these steps, and the associated pieces of the flow diagrams.

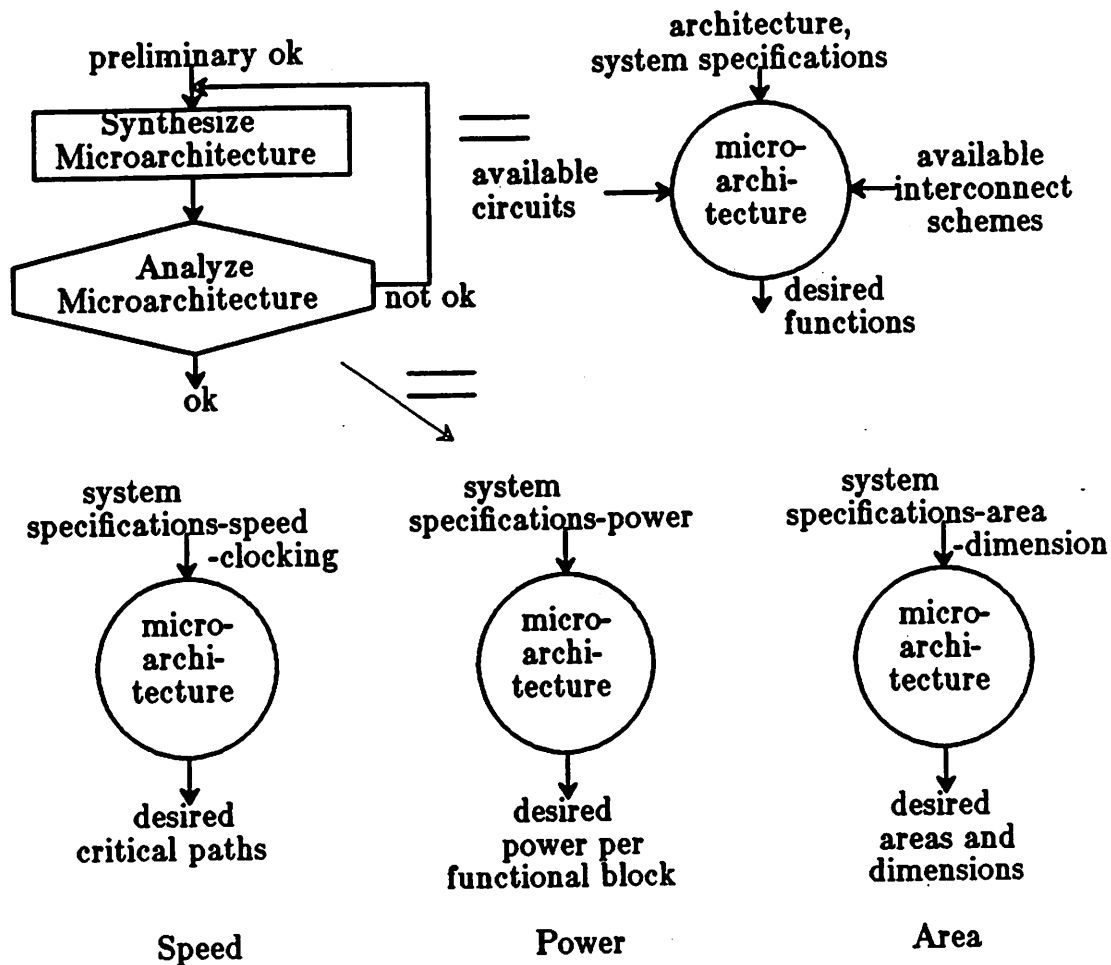


Figure 7.1- Microarchitecture Synthesis and Analysis

## 1. Microarchitecture Synthesis

The goal of microarchitecture synthesis is to completely specify the *desired functions* from the architecture description and system specifications. To do this two aspects of microarchitecture are considered – the necessary functional blocks and the coordination of these blocks. Typically, some of this is done during the preliminary phase, as discussed in the previous chapter. The preliminary phase identified most of the types of functional blocks, defined the basic pipeline, and scheduled the major resources used by this pipeline.

On the SOAR project there was little distinction between microarchitecture and functional block synthesis. The flow diagrams for these two steps were merged into one (Figure 7.2). All inputs were considered and therefore important information was not overlooked. However, with no methodical way of concentrating on the microarchitecture design, it is difficult to streamline the microarchitecture. To illustrate the complete design methodology, the microarchitecture design will be carried out here. In subsequent sections functional block synthesis will be carried out according to this methodology. The pipeline of the realized SOAR from the preliminary phase will be used for this microarchitecture and functional block synthesis. Thus, the inputs to microarchitecture and functional block synthesis for this study will be the same as they were on the SOAR project. In this way a valid comparison can be made between the functional block level designs of the realized SOAR processor and this methodology after functional block synthesis.

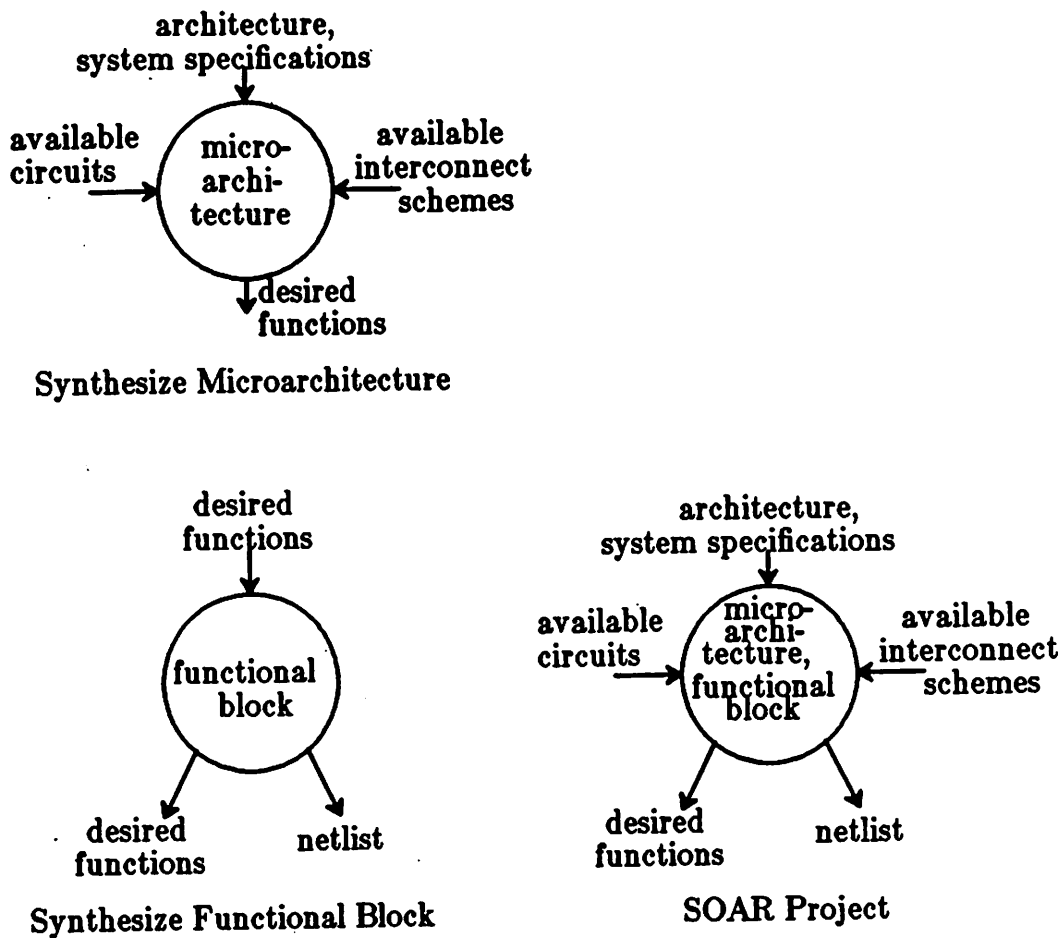


Figure 7.2- Microarchitecture and Functional Block Synthesis

One goal of microarchitecture synthesis is to define the specific functional blocks, their types, sizes, inputs, and outputs. This is done by examining each aspect of the architecture and system specifications, and determining their requirements. As each functional block is identified, its type is determined. Functional block types are chosen from the *circuits available* designed during the preliminary phase. Often a block's size may also be discovered in the architecture. For example, word size determines the size of most datapath blocks. The sizes of the instruction fields indicate the sizes of various control latches. Once this is done, the data inputs and outputs are listed for each block. Control line requirements are determined from the circuit that implements the block type,

the inputs, and outputs of the block. Using the basic pipeline functions and their clock phase assignments, the timing of input loading and output enabling is also listed. The conditions under which inputs are loaded and outputs are enabled is also determined. Together these conditions and timing define the control line inputs to the block. These conditions must be generated somewhere and therefore define outputs of other blocks.

A second goal of synthesis is to coordinate all functional blocks. This includes completely scheduling all resources according to the pipelining and parallelism. Major resources have already been scheduled during the preliminary phase. Now all remaining resources must be scheduled. Bus structures are planned so that enough lines of communication exist for all transactions that occur. Temporary registers, as required by the pipeline, are also identified and added to the list of functional blocks. Timing of all control lines is finally determined according to the control lines' purposes and the pipeline. For example, if a given control line loads an operand and operands are read and available for loading in phase 2, then the control line is enabled on phase 2.

### **1.1. Functional Blocks**

Many of the specific functional blocks for SOAR were already identified during the preliminary phase (Ch.6, Sec.1.1). Table 6.3 lists the desired functions of preliminary design. Many of these correspond directly to functional blocks. An inspection of the instruction set reveals the need for the operations of Table 6.1. Each of these corresponds to a functional block (Table 7.1). Arithmetic, logical, and shift operations must accommodate both 31 and 32 bit data. Two 5 bit decrementers are needed – one for load multiple and one for store multiple. A condition comparator that checks for the conditions of Table 5.7 is needed for skip and trap instructions. The instruction latch becomes an operand for jumps



and calls. A sign extender is needed when the instruction contains a 12 bit immediate field. A hardwired constant - B0000000 - is needed for nilling on returns.

Block	Type	Size (bits)
Add	addition	31, 32
Subtract	subtraction	31,32
And	and	31,32
Or	or	31,32
Xor	xor	31,32
Srl	logical right shift	31,32
Sra	arithmetic right shift	31,32
Insertter	insertion	32
Extractor	extraction	32
CWP+	incrementer	3
CWP-	decrementer	3
D-	decrementer	5
S2-	decrementer	5
Condition comparator	PLA	
Instruction latch	m/s latch	32
SignExt	sign extension	32
Nil	constant	32

Table 7.1- Functional Blocks from the Instruction Set

SOAR's register organization reveals many functional blocks also (Table 7.2). Each register corresponds to a functional block. The 8 global and 64 local registers are realized with static RAM register cells. These are 32 bits wide as determined by the data word size. The special registers are based on the master/slave latches. Various aspects of the architecture call for the eight special registers. Trapping necessitates the PSW, TB, and the shadow registers - SHB, SHA, shOPC, and shDST. The window organization of the local registers calls for the SWP, CWP, and SWP comparator. Rzero is a register that always contains zero. Sizes of the special registers are determined by their functions. Decoders are needed to address the registers specified by the S1, S2, and D fields of the instruction. The decoding scheme described in the preliminary phase is used for the globals and locals (Ch.6, Sec.1.2.3). PLAs are used to select the special registers.

Block	Type	Size (bits)
Global registers	static RAM	8x32
Local registers	static RAM	64x32
Rzero	constant	32
PC	m/s latch	28
SHB	flow through latch	32
SHA	flow through latch	32
SWP	m/s latch	32
TB	m/s latch	18
CWP	m/s latch	3
PSW- PSW	m/s latch	2
shOPC	m/s latch	8
shDST	m/s latch	5
Decoders- Rfile	decoders	
Decoders- specials	PLAs	
SWP comparator	special adder	24

Table 7.2- Functional Blocks from the Register Organization

Trapping and exception mechanisms require another set of functional blocks (Table 7.3). Trap detection can be thought of as needing one block to detect each type of trap and a final block to generate the trap signal (Figure 7.3). Once a trap has been detected, its priority must be encoded as part of the trap address. The trap base latch and shadow opcode complete the trap vector address.

Block	Type	Size (bits)
Illegal Opcode Detection	random logic	?
Tag Trap Detection	random logic	?
GS Trap Detection	random logic	?
SWI Detection	random logic	?
Window Overflow Detection	random logic	?
Window Underflow Detection	random logic	?
Trap Instruction Detection	random logic	?
Data Page Fault Detection	random logic	?
Instruction Page Fault Detection	random logic	?
I/O Interrupt Detection	random logic	?
Trap	random logic	?
Priority Encoder	random logic	?

**Table 7.3- Functional Blocks from the Trap Mechanism**

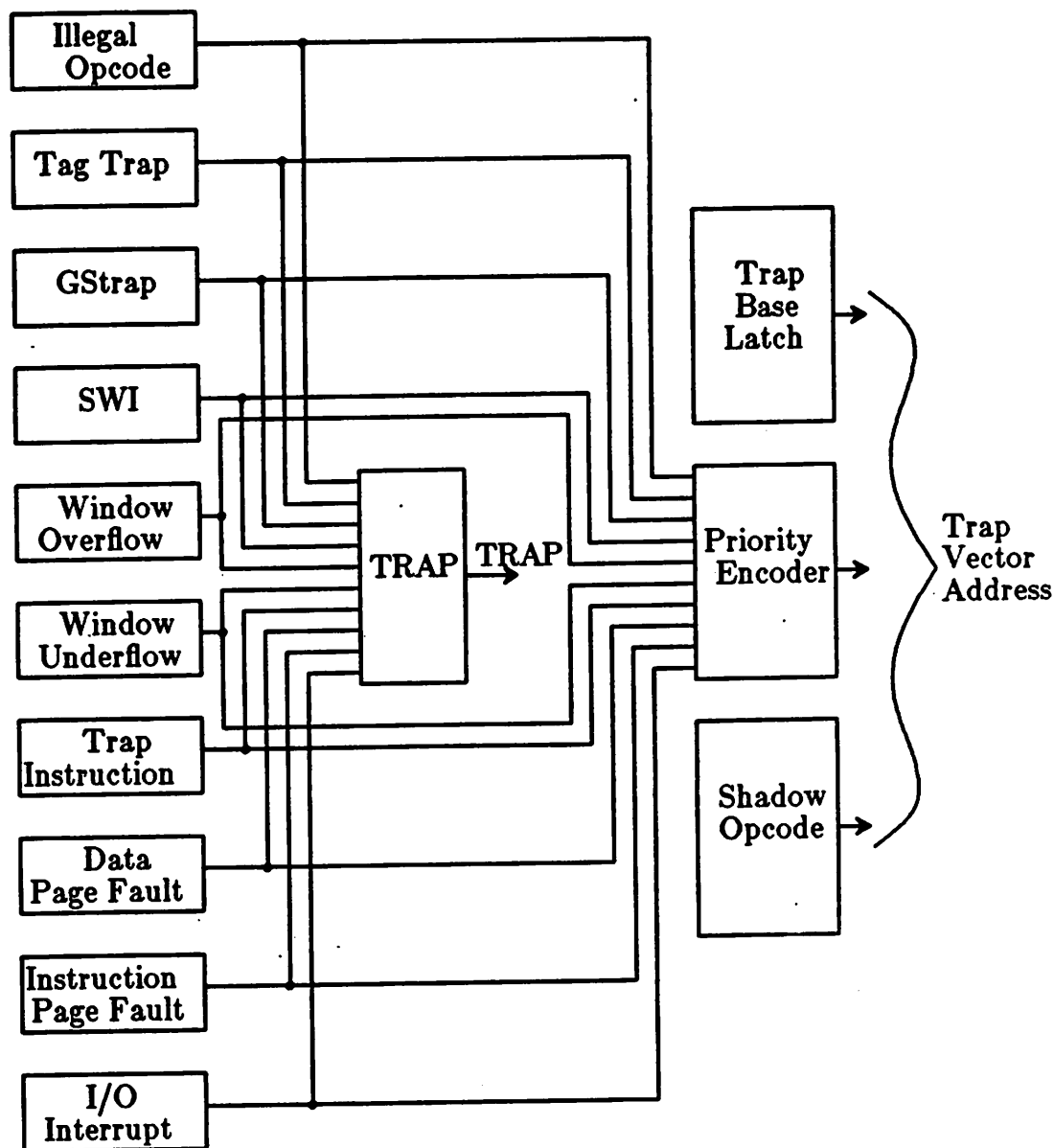


Figure 7.3- Trap Functional Block Diagram

System requirements generate another set of functional blocks (Table 7.4). Data and address outputs must be able to drive TTL loads and therefore require 32 bit driver blocks. Random logic blocks must generate the internal WAIT and RESET signals from the corresponding external signals. The output signals I/D, RD/WR, FSHCNTL, and WAITACK are also generated from random logic. A program counter incrementer is needed to correctly address memory.

Block	Type	Size (bits)
Data drivers	tristate pad driver	32
Address drivers	pad driver	32
WAIT	random logic	
RESET	random logic	
I/D	random logic, pad driver	
RD/WR	random logic, pad driver	
FSHCNTL	random logic, pad driver	
WAITACK	random logic, pad driver	
PC+	incrementer	28

Table 7.4- Functional Blocks from the System Requirements

A final set of functional blocks is needed due to the processor's pipelining. Whenever data is available from one resource on a specified clock phase but used by another resource during a later phase, the possibility exists that the originating resource will have already been re-used by the time the requesting resource needs the data. Thus, the requested data will have been lost unless some temporary storage is provided for it. The need for temporary registers can readily be seen in diagrams such as Figure 7.4. In Figure 7.4 the pipelining for all instructions that use the write pipeline function are shown along with the pipelining of instructions that overlap the write. For the instruction being analyzed, the origination of information used in the write is circled and an arrow drawn to the clock phase of the write that uses it. If the originator is used between the time it generates the information and the completion of the pipeline function that uses the information, a temporary latch will be needed. Reuse is indicated by the dotted circles in

Figure 7.4. To detect any reuse of the originator, all pipeline functions of overlapping instructions are checked. For example, during arithmetic, logical, and shift operations the ALU generates data in cycle 2, phase 3. This information is then written in cycle 3, phase 3. However, the following instruction uses the ALU in its cycle 2, phase 3, before the write has completed. Therefore, a temporary register is needed to store the ALU result before the write, so that another alu operation may start. Similarly, an instruction fetch retrieves the destination of the write that is then decoded during the 2nd decode - cycle 3, phase 1. Meanwhile, another instruction fetch has been completed. Upon completion the destination field will change. Thus, a temporary register is needed for the destination field of the original instruction. Similar diagrams are shown in Figure 7.4 for all instructions using the write pipeline function. Temporary latches are needed for fetched data, register data to be stored, and the program counter. This same analysis is done for all pipeline functions to identify the temporary storage blocks needed (Table 7.5).

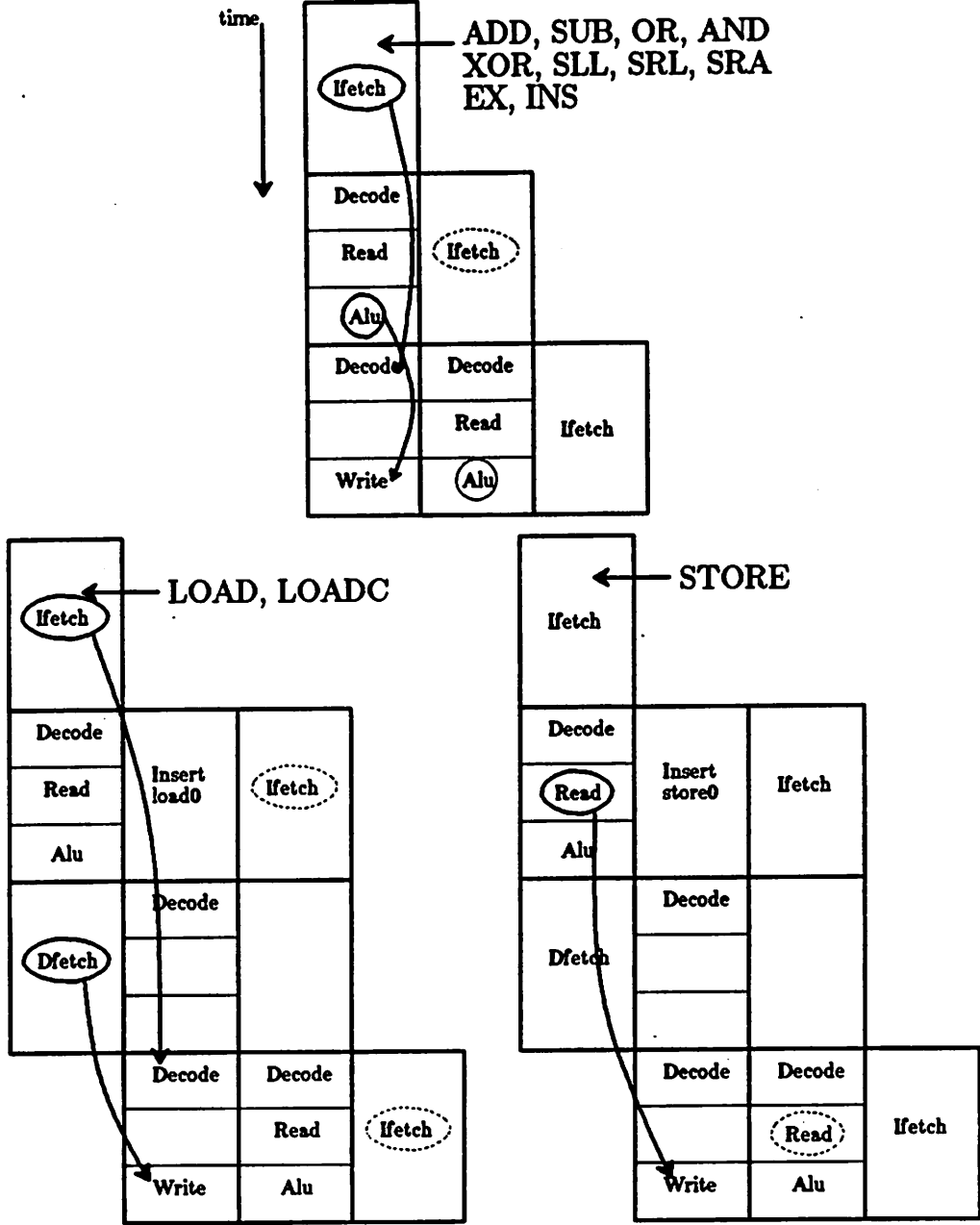


Figure 7.4- Temporary Storage Identification



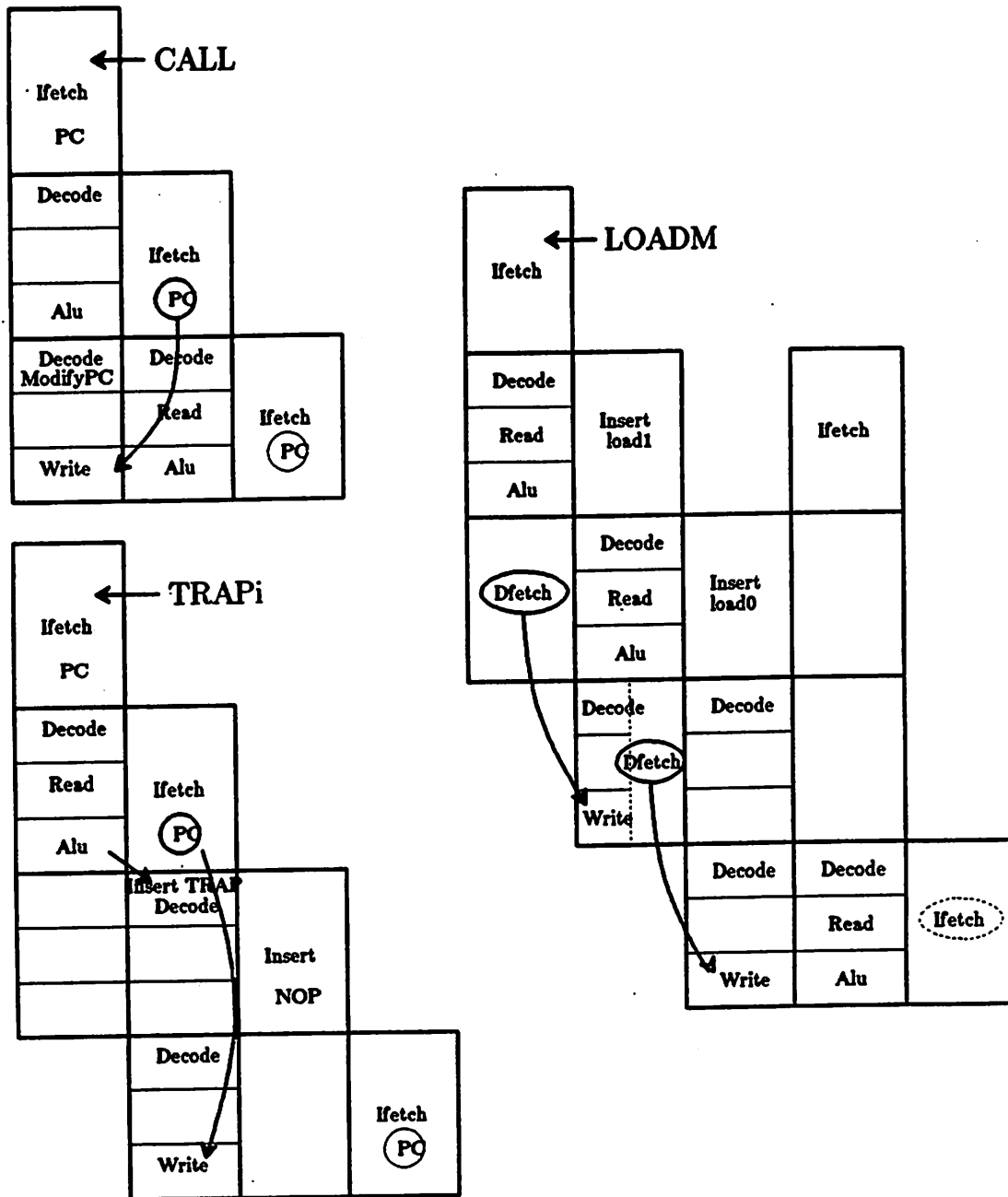


Figure 7.4- Temporary Storage Identification (cont.)

Block	Needed by	Type	Size (bits)
TempALUoutput1	write, Dfetch, store	m/s latch	32
TempDataIn	write	m/s latch	32
TempPC	write	m/s latch	32
TempStoreOperand	write, store	m/s latch	32
TempInstrLatch	2nd decode	m/s latch	13
TempALUinput1	ALU	m/s latch	32
TempALUinput2	ALU	m/s latch	32
TempALUoutput2	compare	latch	2
TempOpcode	DetectTrap	m/s latch	9
TempTags	DetectTrap	latch	8
TempPSW	DetectTrap	latch	1
TempCWP	DetectTrap	latch	3
TempSWP	DetectTrap	latch	3

Table 7.5- Functional Blocks for Temporary Storage

Tables 7.1 through 7.5 summarize all functional blocks that are identified by an inspection of the architecture, system specifications, and processor pipelining. These functional blocks will be organized into circuit blocks during the functional block synthesis step.

## 1.2. Bus Structures

The bus structures of a processor must provide for all the communication required by the pipeline functions. Processor instructions were broken into pipeline functions (Ch.6, Sec.1.3). Therefore, by satisfying the communication

needs of the pipeline functions, the needs of the instructions are also satisfied. Information transfer is indicated by the arrows in the algorithmic description of each pipeline function. Therefore, each arrow must be assigned a time slot on a bus structure. The inputs to the bus structure are indicated by the functional blocks of the right side of the statement. The destinations of the bus structure are the functional blocks of the left side of the statement. For example, the write pipeline function was algorithmically expressed by:

$$r[D] \leftarrow \text{ALUoutput}$$

The right side - ALUoutput - indicates that an ALUoutput block is the origin of the data. Consultation with the lists of functional blocks reveals that TempALUoutput1 was intended to hold the ALU data for subsequent writes. The destination block is a register -  $r[D]$ . This could be in one of the 72 global or local register blocks or any of the special registers. Thus, a bus structure must exist between TempALUoutput1 and these registers (Figure 7.5). All portions of this bus structure are reserved for writes in clock phase 3 - the write phase.

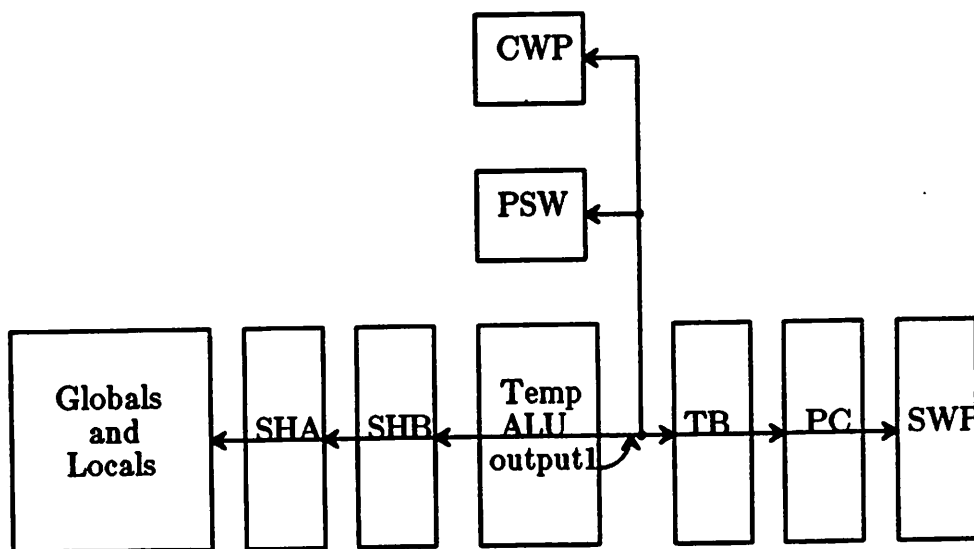


Figure 7.5- Bus Structure for  $r[D] \leftarrow \text{ALUoutput}$

To design the processor bus structure all pipeline functions are analyzed in this way. A table is compiled that lists all necessary lines of communication with their origins, destinations, and time slot usage (Table 7.6). Assignments through random logic— the decode function for example – are not included in this table. These assignments are not simple interconnects but signify PLAs and other logic circuits.

Pipeline Function	Originating Block	Destination Block
Write- phase 3	TempALUoutput1	globals, locals, PC, SHB SHA, TB, SWP, PSW, CWP
	TempDataIn	globals, locals
	PC, TempPC	locals
Ifetch- phase 1	PC	address drivers
phase 3	data drivers	instruction latch
phase 3	PC+	PC
Read- phase 2	globals, locals, PC, SHB, SHA, TB, SWP, CWP, PSW	TempALUinput1
	TempALUoutput1	TempALUinput1
	globals, locals	TempALUinput2
	instruction latch	TempALUinput2
	SignExt	TempALUinput2
	TempALUoutput1	TempALUinput2
	globals, locals	TempStoreOperand
DetectTrap- phase 3	TB	PC
DetectSkip- phase 3	PC+	PC
Nil- phase 3	Nil	locals
CWP+- phase 3	CWP+	CWP
CWP-- phase 3	CWP-	CWP
modifyPC- phase 3	Add	PC
Dfetch- phases 1,2,3	TempALUoutput1	address drivers
phase 3	data drivers	TempDataIn
Store- phases 1,2,3	TempALUoutput1	address drivers
phases 1,2,3	TempStoreOperand	data drivers
phase 3	TempStoreOperand	locals
ALU- phase 2	TempALUinput1	add, subtract, and, or xor, srl, sra, inserter extractor
phase 2	TempALUinput2	add, subtract, and, or xor, srl, sra, inserter, extractor
phase 3	add, subtract, and, or, xor, srl, sra, inserter, extractor	TempALUoutput1,2

Table 7.6- Lines of Communication

Bus structures are now proposed that account for every entry in Table 7.6. The possible solutions are limited by restrictions to microarchitecture design from the preliminary phase – characteristics of *available circuits* and *interconnect schemes*. For SOAR, important limitations were:

1. Two read ports from the register file using the two bit lines.
2. One write port to the register file using the same two bit lines.
3. Space for only one bus to cross the ALU blocks– Add, Subtract, Xor, Or, And, Srl, Sra, Inserter, Extractor.
4. Data pads on one side of the chip and address pads on the opposite.

Bus structures may be represented in functional block diagrams that show the functional blocks and buses that connect them. For SOAR, the functional block diagram of Figure 7.6 is proposed. This is an extension of Figure 7.5 that includes all the buses required by Table 7.6.

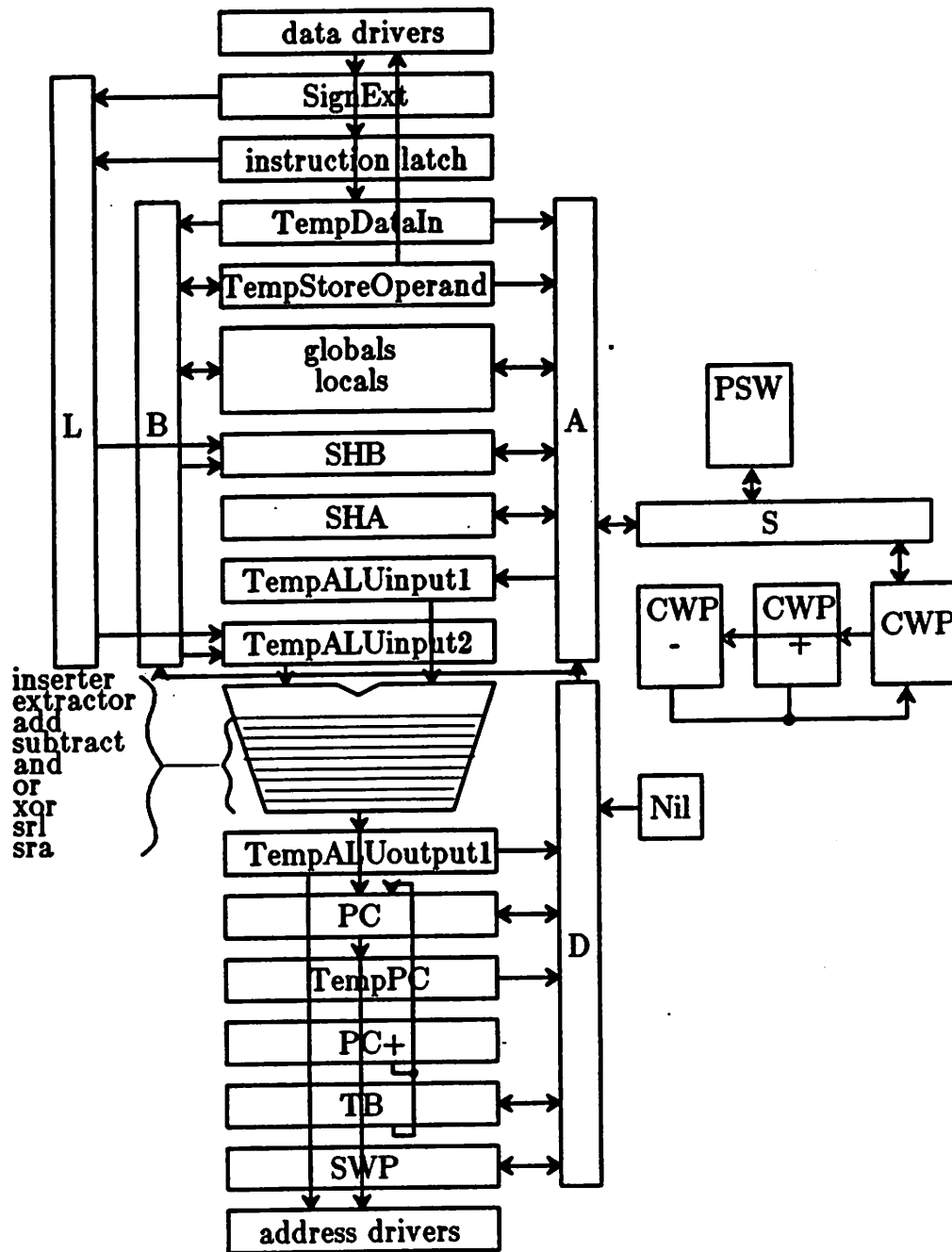


Figure 7.6- Proposed SOAR Bus Structure and Functional Block Diagram

### 1.3. Resource Usage

Bus usage for SOAR is shown in Table 7.7. When more than one pipeline function uses the same bus time slot, the pipeline functions must be mutually exclusive. If they are not, a conflict would exist. For example, both write and nil use busD during phase 3. However, nil uses it during cycle 3 of a return while write uses it during cycle 3 of an alu, call, trap, load, or store instruction. Any two of these instructions can not be in cycle 3 simultaneously and therefore there is no conflict for busD by the nil and write functions.

Bus	Phi1	Phi2	Phi3
BusA	Precharge	Read	Write, Nil, Store
BusB	Precharge	Read	Write, Nil, Store
BusD	Precharge	Read	Write, Nil
BusL	Precharge	Read	unused
BusS	Precharge	Read	unused
DataIn	unused	Store	Ifetch, Dfetch, Store

Table 7.7- Bus Structure Usage

At this time resource usage for all resources should be examined to ensure that no conflicts exist. Major resource usage was analyzed in the preliminary phase (Ch.6, Sec.1.6). Similar analysis is now done for all remaining resources, revealing no conflicts.

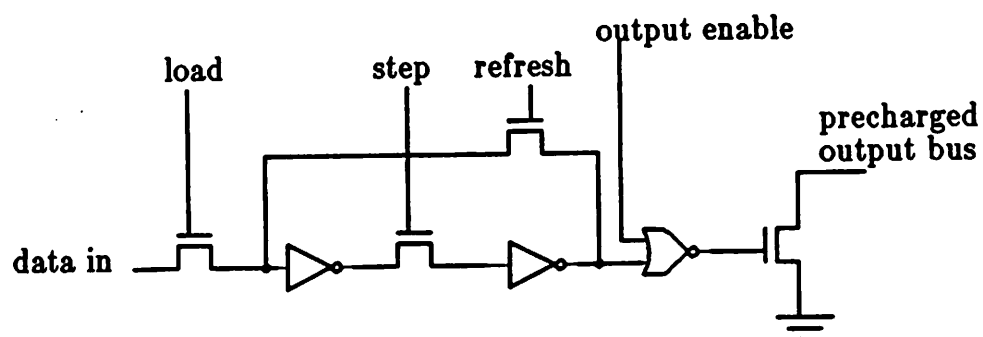


#### 1.4. Complete Functional Block Input and Output Specification

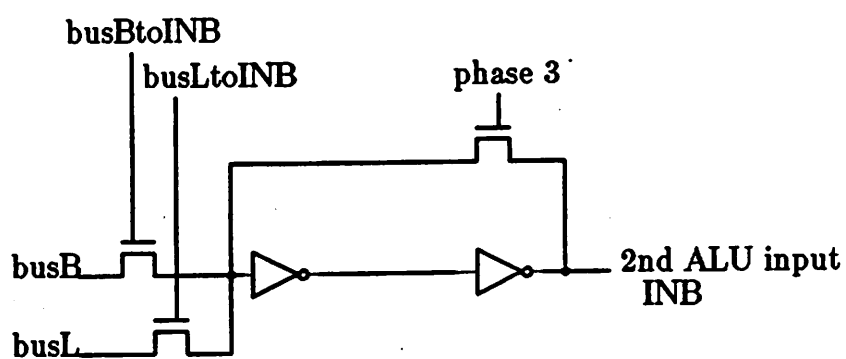
To complete the details of microarchitecture synthesis all inputs and outputs of each functional block must be specified. At the start of microarchitecture synthesis the circuit type of each block was identified. This circuit type may need modification to provide the exact function called for by the functional block. Modifications may include variations on the number of input and output ports or internal clocking. Data inputs, control line inputs, and data outputs of the customized block are identified. The conditions that determine the control line inputs are also identified. These conditions must be generated somewhere and therefore identify condition outputs of other blocks.

The TempALUinput2 provides a good example of this procedure. It was identified as a 32 bit master/slave latch. Figure 7.7a shows one bit of a master/slave latch from preliminary circuit design. Consultation with the functional block diagram (Figure 7.6) and algorithmic statements of the pipeline functions (Ch.6, Sec.1.3) shows that this latch needs two inputs – one from busL and one from busB. Loading occurs due to a read which is in phase 2. Thus, both load control lines are enabled during phase 2. The output of this latch is the only connection to the second ALU input. Therefore, no enable circuitry is needed on the output. The ALU operation involves precharged logic and occurs in phase 3. Therefore, the ALU inputs must be set up in phase 2. Consequently, the output of this latch must be ready in phase 2. This means that the step transistor is eliminated since the inputs are loaded and must flow through to the output in phase 2. Refreshing always occurs in phase 1 or phase 3. Conditions for loading busB are during the read function of non-immediate ALU, byte, skip, trap, return, and load instructions. BusL is loaded during the read phase of a call, jump, or store instructions, and immediate ALU, byte, skip, trap, return, and load instructions. Table 7.8 summarizes the inputs and outputs of the

TempALUinput2 block.



(a) Generic M/S Latch



(b) Customized Latch

Figure 7.7- M/S Latch Customization to form TempALUinput2

Inputs	Outputs	Control Lines	Enable Conditions	From
BusB BusL	INB	busBtoINB  busLtoINB	phi2 and imm and (op1= add,sub,or,and,xor,srl, sra,insert,extract,skip, trapi,reti,load,loadc, loadm)  phi2 and [(imm and (op1= add,sub,or,and,xor,srl, sra,insert,extract,skip, trapi,reti,load,loadc, loadm))or(op1=call,jmp, store,storem)]	decode PLA  decode PLA

Table 7.8- Inputs and Outputs- TempALUinput2

To completely specify the microarchitecture this is done for all functional blocks. All conditions must be generated somewhere. Therefore, if functional blocks that generate conditions have not been specified they must be specified now. For SOAR this means adding decode PLA blocks to the list of functional blocks.

### 1.5. Microarchitecture Verification

After the microarchitecture has been specified it must be checked to make sure that it executes instructions and responds to system inputs as it should. This verification process is crucial to finding and eliminating errors in the microarchitecture. Verification is usually done with the aid of CAD tools. Three

things are needed for verification:

1. A microarchitectural or functional simulator
2. A complete microarchitecture description, compatible with the simulator
3. A set of test programs or diagnostics

A functional simulator has the capability of generating new node values from previous and present node values. Changes to the input nodes – clocks, data inputs, etc. – cause changes in many internal nodes. The functional simulator generates these new node values. Typically, input nodes including clocks are changed simultaneously on the clock's edge. The simulator then evaluates all nodes affected by the input nodes and the nodes that they in turn affect, until all nodes have settled. The simulator is then ready to accept another set of inputs. Thus, a test program is stepped through by clock phases. The SOAR project used the SLANG functional simulator [VanD82].

The functional simulator expects a microarchitecture description containing information like that of Table 7.8 and Figure 7.7b, for all functional blocks. The SLANG description of the TempALUoutput2 latch is given:

```
(defnode INB
  (depends phi2 busL busB phi3 busLtoINB busBtoINB)
  (update
    (If3way phi3
      INB
      (If3way busLtoINB
        busL
        (If3way busBtoINB
          busB
          INB
          UNK)
        UNK)
      UNK)))
```

The output node is given in the defnode statement. Input nodes are listed in the depends clause. The update statements are a description of the circuit in Figure 7.7b. Pass transistors are described by the If3way statements. Higher level abstractions of functional blocks will speed up the simulation but care must be taken so that accuracy is not sacrificed. For example, the register file for SOAR was simply described by an array. Individual gates were not simulated. Appendix B contains the SLANG description of SOAR.

To completely test the microarchitecture a set of diagnostic programs that exercise all possible situations and features of the architecture, is needed. For SOAR a list of items to test was compiled and diagnostics were written to cover these situations.

Table 7.9 lists the diagnostics used to test the instruction set. Common instructions - add, subtract, jump, and call - and common situations - sign extension of an immediate - happened throughout the diagnostic set and therefore did not need separate diagnostics dedicated to them. The largest effort here was put into completely testing all conditions (Table 5.7). The multitude of return instructions and forwarding situations were the next largest efforts for the instruction set diagnostics.

Table 7.10 shows the diagnostics used to test the register organization. Window management was the most complex register feature tested. This was followed by the memory mapping of on chip registers - pointer to register.

Diagnostics to test the trap mechanisms are shown in Table 7.11. The most complicated trapping diagnostics were due to the variety of tag and generation scavenging traps, and the window management diagnostics.

Diagnostics for the external interrupts are shown in Table 7.12. Most difficult to test was the WAIT signal. It could arise under many circumstances and had to be tested for all these possibilities.

Feature	SOAR Cycles	% of Diagnostics
conditions	888	17.1
return	292	5.6
forwarding	195	3.7
skip	104	2.0
trap instructions	78	1.5
store	46	.9
loadm, storem	44	.9
or, xor, and	40	.8
insert, extract	38	.7
srl, sra, sll	30	.6

Table 7.9- Instruction Set Diagnostics

Feature	SOAR Cycles	% of Diagnostics
windows	728	14.0
pointer to register	347	6.7
special registers	108	2.1

Table 7.10- Register Organization Diagnostics

Feature	SOAR Cycles	% of Diagnostics
window overflow, underflow	728	14.0
tag traps	623	12.0
-alu, shifts	255	4.9
-loads	180	3.5
-overflow	130	2.5
-skips	58	1.1
generation scavenging	294	5.7
-store	265	5.1
-return	29	.6
illegal opcodes	209	4.0
priority mechanism	149	2.9
software interrupt	97	1.9
trap instructions	78	1.5
I/O interrupt	61	1.2
page faults	55	1.1

Table 7.11- Trap Mechanism Diagnostics

Feature	SOAR cycles	% of Diagnostics
wait	645	12.4
I/O interrupts	61	1.2
page faults	55	1.1

Table 7.12- External Signal Diagnostics

Table 7.13 summarizes these diagnostic categories. The major portion of the diagnostic effort was spent testing the trapping mechanisms - 44.3%. Following this was the instruction set which was dominated by the condition testing. A total of 5177 SOAR cycles made up the diagnostic set. A few diagnostics - such as the window management tests - fell into more than one category. This is why the total is less than the sum of the individual categories.

Diagnostic Category	SOAR Cycles	% of Diagnostics
Trap mechanism	2294	44.3
Instruction set	1755	33.8
Register organization	1183	22.8
External signals	761	14.7

Table 7.13- Diagnostic Categories



## 1.6. Microarchitecture Synthesis Summary

Detailed microarchitecture design was completed in the *microarchitecture synthesis* step. This detailed design considered the external inputs – *architecture* and *system specifications* – and internal fixed inputs from the preliminary phase – *available circuits* and *interconnect schemes*.

Using these inputs and the pipeline design from the preliminary phase, all functional blocks were identified. Their function was expressed first in terms of the available circuits and then by customization of these circuits. Bus structures were designed to accommodate all data transfers implied by the algorithmic descriptions of the pipeline functions.

The complete microarchitecture description included listings of the data and control line inputs to each functional block, along with the data and condition line outputs of the blocks. This and the blocks' operations completed the detailed microarchitecture description.

Finally, the detailed description was verified for correctness. This was done by running diagnostic programs on a simulator that functioned according to the detailed microarchitecture description. Many bugs were discovered and corrected due to this verification process. The verification process including diagnostic development, simulation, and debugging took 7 to 8 months.

## 2. Microarchitecture Analysis

Analysis can be done for any characteristic of the processor (Ch.4, Sec.3). Typically, analysis includes, but is not limited to, the speed, power, and area characteristics. For SOAR the primary concern was processor speed. Area and power limits were looser and had lower priorities.

## 2.1. Speed Analysis

Speed analysis, at the microarchitecture level, consists of examining all processor activities to determine the clock phases that are allotted to each activity. The circuit blocks and interconnects that realize a given activity must be fast enough to complete the activity in the allotted time. The flow diagram for this is shown in Figure 7.8. The '?' indicates a comparison between the time allotted to each activity, according to its clock phases, and the predicted times, according to the speed of the components of the activity. The circuit block and interconnect composition of each activity is not yet known, so analysis at this point consists of identifying the clock phases allotted to each activity – the section of Figure 7.8 enclosed by the dashed lines. This information is recorded and used in later comparisons when the composition of each activity is known.

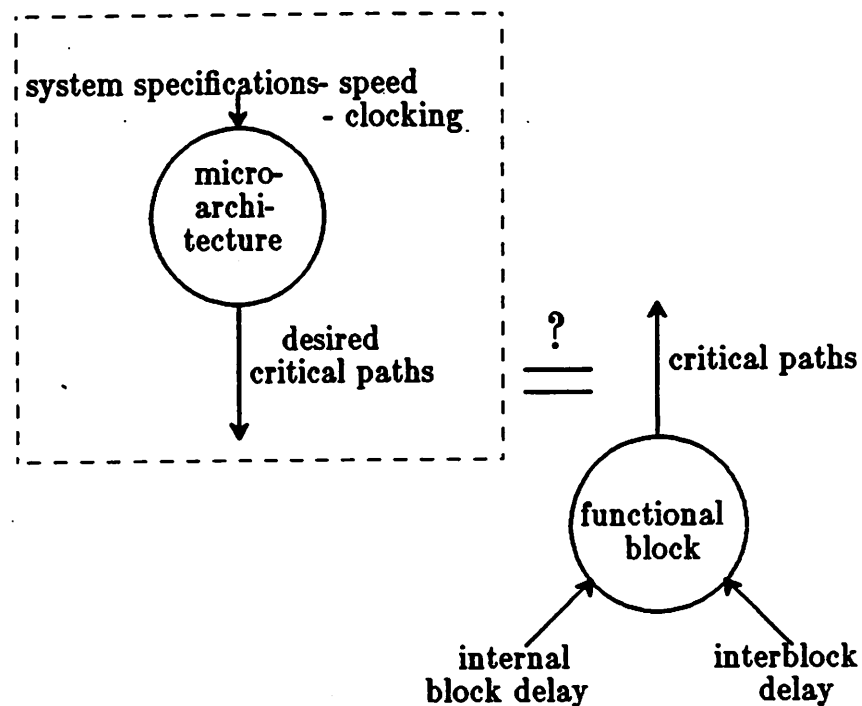


Figure 7.8- Speed Analysis After Microarchitecture Synthesis

During microarchitecture synthesis, inputs and outputs of all functional blocks have been identified and recorded in tables such as Table 7.8. The processor's activities lead to the setting of these signal lines. To consider all activities it is sufficient to consider either all inputs for all functional blocks or all outputs. This is because the inputs of a given block are generated from one or more outputs of other blocks. When considering the timing of an input to a block it is necessary to trace it back to its origins - outputs of other blocks. Therefore, outputs will automatically be considered, as necessary, as the designer proceeds through the list of inputs. Similar arguments can be made for the consideration of all outputs. In this methodology the inputs are used because of the data organization. Tables such as Table 7.8 show the outputs of other blocks that constitute an input, but do not hold information on inputs to other blocks that use the given block's outputs. Therefore, information is available to trace back from the inputs but not forward from the outputs.

The inputs in Table 7.8 contain two types of information. First, they contain timing information. This appears either as explicitly mentioned clock phases or implicitly through the functional description of the block - Figure 7.7b or the SLANG description. Secondly, they contain signals that are outputs of other blocks. These outputs of other blocks are generated by the functions of the other blocks. They are therefore the result of some processor activity. The timing information of each input indicates when these results will be used as an input. Therefore, the timing information signifies the time when the activity that generates the result, must be completed. The timing of the start of an activity is found by tracing the result portion of the input signal back to its origins. Typically, the activity starts on a clock edge indicated by the functional description of an originating block.

To illustrate this microarchitectural speed analysis the TempALUinput2 functional block is used as an example. Table 7.8 shows four inputs to this block – busB, busL, busBtoINB, and busLtoINB. Explicit timing information is shown for the two control lines. Both busBtoINB and busLtoINB are gated with phi2. Other information forming these control lines are conditions generated by the decode PLA. An inspection of Figure 7.7b shows that the clock phase phi2 is used to load new data if the conditions are correct. Therefore, the conditions must be determined by the start of phi2 to make sure that data is not erroneously loaded. Thus, the rising edge of phi2 signifies the end of the time allotted to the decoding for these two control lines (Figure 7.9). The falling edge of phi2 signifies the completion of the load. Therefore, the data on busB or busL must be valid by this time. Thus, the end of the read activity is the falling edge of phi2.

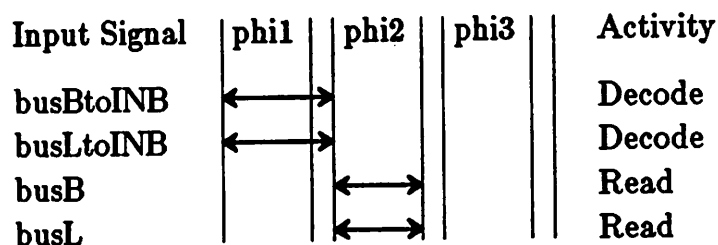


Figure 7.9- Settling Time Slots for Inputs to TempALUinput2

The starts of these decode and read activities are found by tracing the result portions of the input signals back to their origins. The control lines contain conditions that are formed combinatorially from data in the instruction latch by the decode PLA. The functional representation of the instruction latch (Figure 7.10) shows that this data is available at the start of clock phase phi1. Thus, decode begins at the rising edge of phi1. BusB can be generated in a few ways. It can be generated by a register file read, forwarding from the TempALUoutput1 or

TempDataIn latches (Figure 7.11). All of these activities are initiated by the rising edge of phi2. Similarly, busL is used to transfer data from either the sign extender or instruction latch (Figure 7.12). Both of these data transfers also begin with the rising edge of phi2 also. Therefore, the rising edge of phi2 is the start of the read activity.

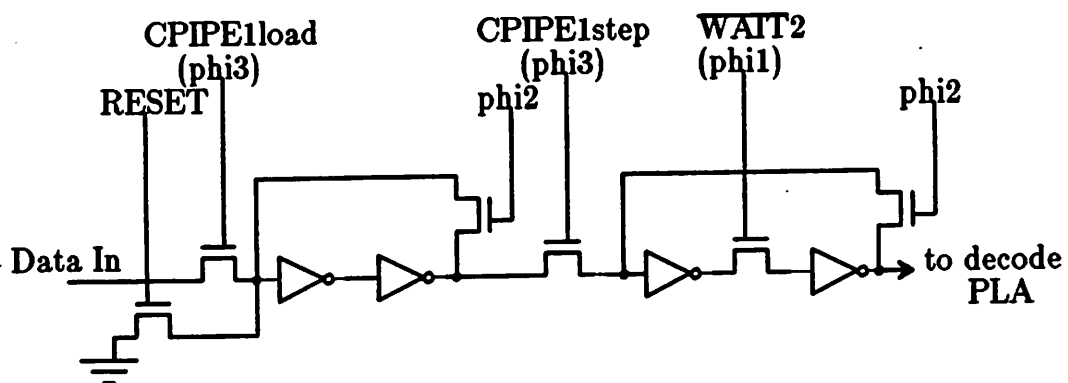


Figure 7.10- Instruction Latch Functional Representation

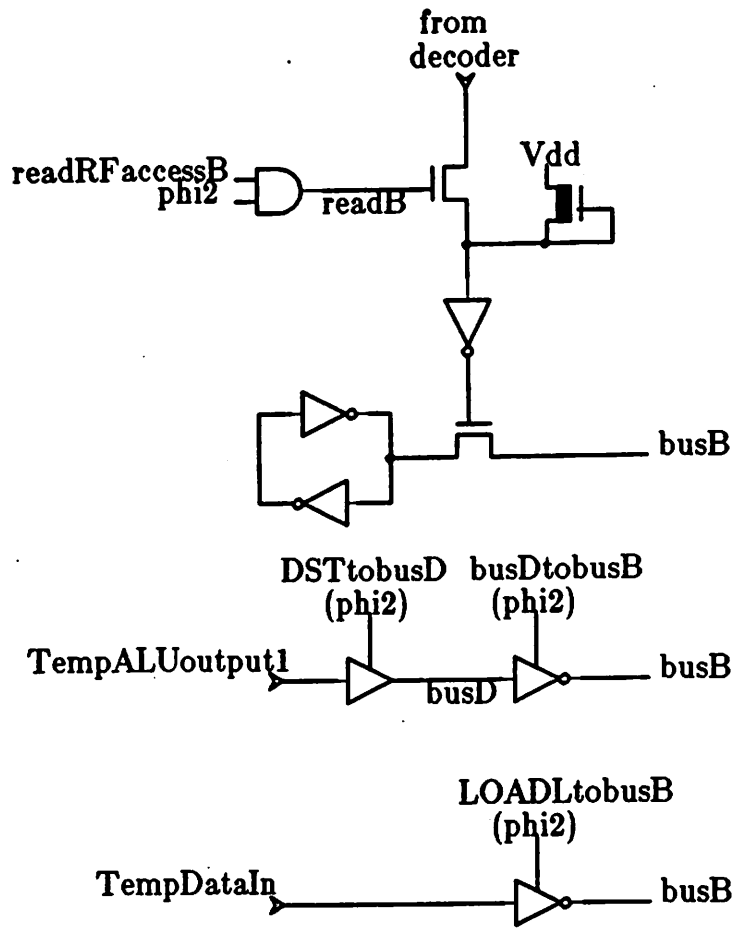


Figure 7.11- BusB Sources for a Read

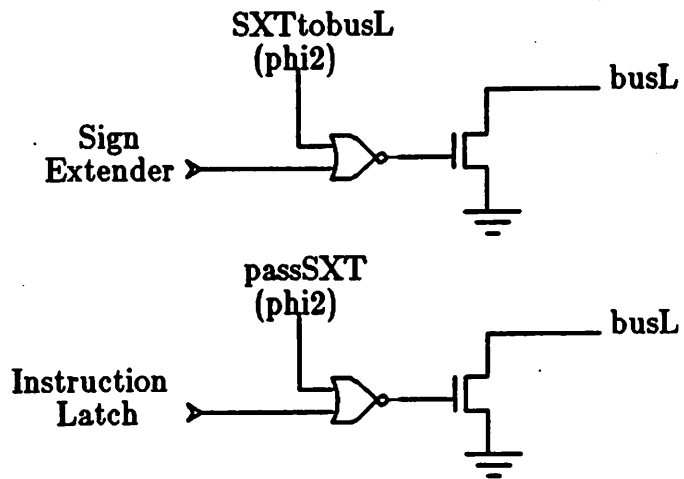


Figure 7.12- BusL Sources for a Read

Figure 7.9 indicates the activities that correspond to the settling of each input along with the clock phases allotted to the activities. This type of analysis is carried out for all blocks. As synthesis progresses estimates of the times needed for these signals to settle will become available. These estimates are compared with the allotted clock phases and their lengths. These comparisons can reveal signals that are not fast enough according to their time allotments and places where the time allotments are longer than necessary.

This method of analysis reveals the timing of all signals. For the processor to function correctly all signals must function properly. Speed problems with any signal, no matter how minor the signal is, can limit the speed of the entire processor. Therefore, a method is needed to analyze all signals early in the design process, not just major signals. In this way unobvious speed problems in the microarchitecture will not be overlooked. Steps may be taken to correct any problems before large amounts of work would have to be redone for the corrections. This microarchitectural analysis of the clock phase allotments for the settling of all functional block input signals becomes the *desired critical paths* output of the microarchitecture level in the speed analysis flow diagram (Figure 7.8). Later in the design this output is compared with the increasingly accurate estimates of the critical paths.

## 2.2. Area and Power Analysis

SOAR also had area and power specifications. Normally, area and power analysis consists of allotting area, dimensions, and a power budget to each functional block (Figure 4.19e). The assumption for SOAR was that these specifications were loose and that as long as the total area and power met the specifications, it did not matter how these characteristics were divided up. Thus, the original system specifications for power and area were passed through the

microarchitecture level with no analysis (Figure 7.13).

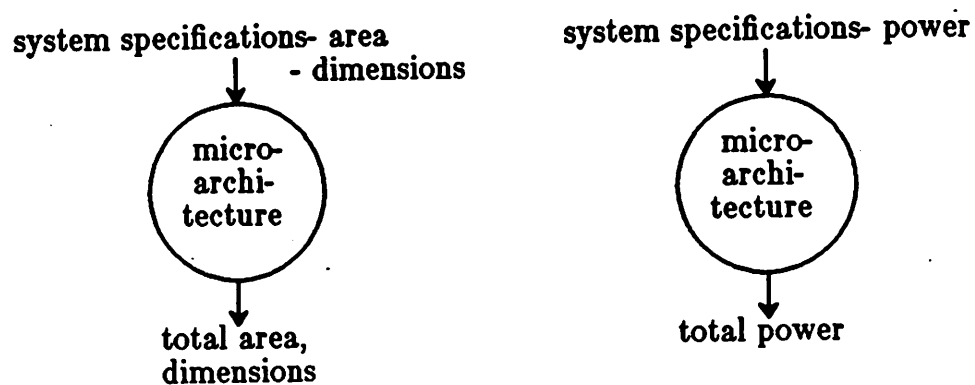


Figure 7.13- Power and Area Analysis After Microarchitecture Synthesis

### 3. References

[VanD82] Van Dyke, K. S.; 'SLANG a Logic Simulation Language', M.S. Thesis, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., June 1982.



## Chapter 8

### Functional Block Design

#### SOAR Case Study

After synthesis and analysis is completed at the microarchitecture level, design moves to the functional block level (Figure 4.18). The flow diagram for the functional block steps are shown in Figure 8.1. *Functional block synthesis* consists of mapping the functional blocks into the actual circuit blocks. As this is done, the processor interconnects between the circuit blocks are defined. Analysis at this level first considers the circuit blocks and interconnects alone. Then the individual characteristics are combined to arrive at the global characteristics. These global characteristics are compared to their desired values that were arrived at through microarchitecture analysis.

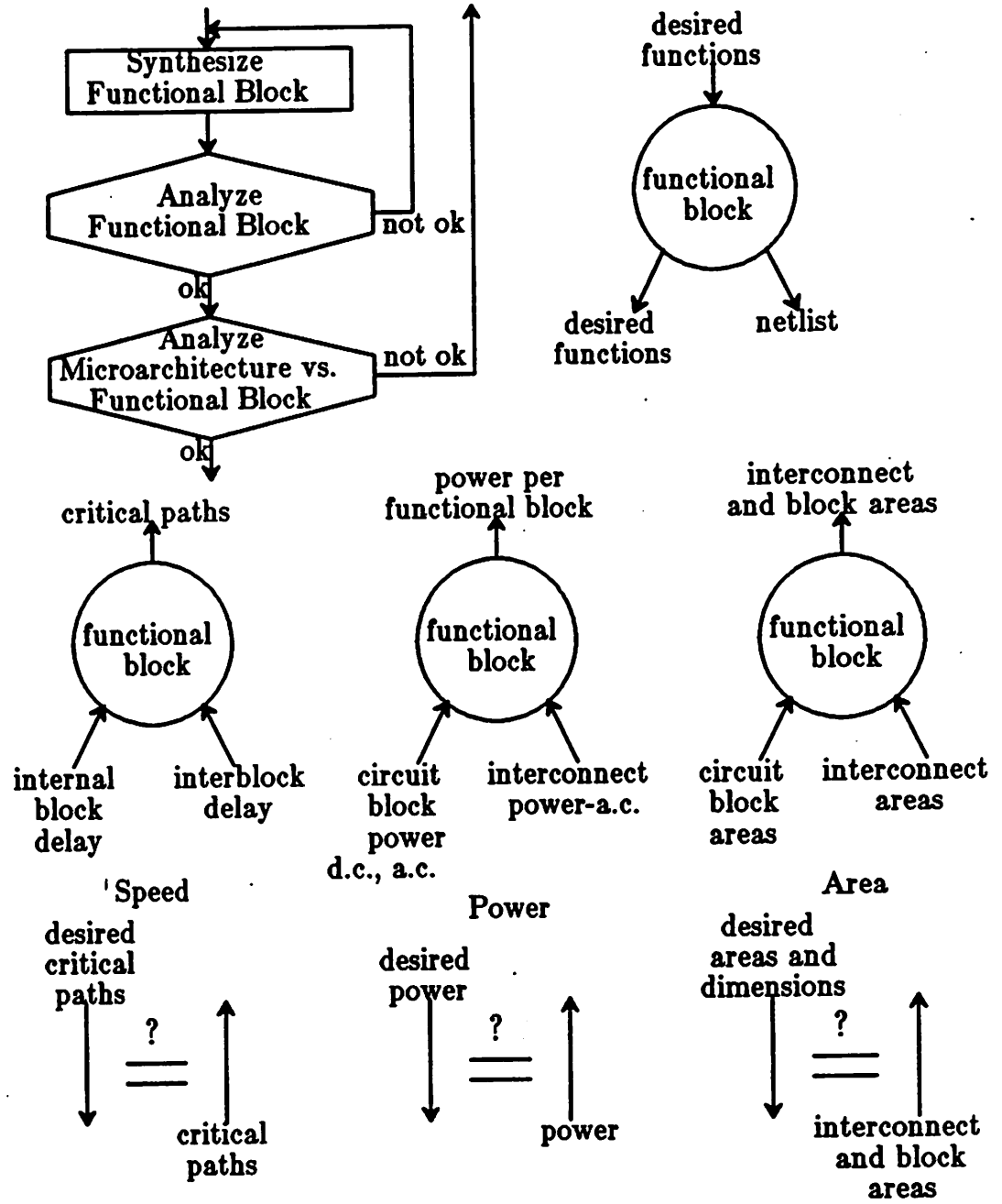


Figure 8.1- Functional Block Synthesis and Analysis

## 1. Functional Block Synthesis

As previously discussed, there was no real distinction between microarchitecture and functional block synthesis on the SOAR project. In this section the results of microarchitecture synthesis as it might have been done will be used (Ch.7, Sec.1). This includes the list of functional blocks along with the detailed descriptions of their inputs, outputs, and function. Using these inputs to functional block design, synthesis will be discussed to illustrate the methodology on the SOAR processor.

When assigning functional blocks to circuit blocks, there is not always a one to one correspondence between functional blocks and circuit blocks. Sometimes many functional blocks can be incorporated into one circuit block – merging of functional blocks – or one functional block may be mapped into more than one circuit block – splitting of functional blocks.

### 1.1. Merging

It is desirable to merge functional blocks, since it can lead to less circuitry and fewer and shorter interconnects. The most obvious candidates for merging are functional blocks that use all or part of the same circuitry and have the same inputs and outputs. In this way unnecessary duplication of circuitry is avoided. Table 8.1 shows the functional blocks that were merged on SOAR for this reason. In the adder, AND, OR, and XOR are formed and used to compute sums. Subtract is done using the ones complement of the second operand and performing an add. Therefore, these five functions exist in the adder. In the shifter, both shift functions are the same except for the most significant bits that are shifted in. Therefore, the functional blocks srl and sra were combined to form the shifter circuit block. Both insert and extract use similar hardware (Figure 6.10). The TempDataIn and TempStoreOperand blocks are both temporary

latches that hold external memory data. One is used during loads and the other during stores. Since they have the same function and are never both used at the same time they may be merged. Either the nil constant or the TempALUoutput1 latch data is written into the register file, so the nil constant was merged with the TempALUoutput1 latch.

Functional Block	Circuit Block
Add	Adder
Subtract	
And	
Or	
Xor	
Srl	Shifter
Sra	
Insert	EX/INS
Extract	
TempDataIn	LOADL
TempStoreOperand	
Nil	Destlatch
TempALUoutput1	

Table 8.1- Common Circuitry Merging

Functional blocks that share the same inputs and outputs but have different internal circuitry may also be merged into one circuit block. The internal circuitry of the circuit block is the collection of circuitry from each functional block. Thus, total circuitry is not reduced. However, the functional blocks have the same inputs and outputs and can therefore share interconnects, reducing routing. Table 8.2 shows blocks that were merged due to shared inputs and outputs.

Functional Block	Circuit Block
Adder Shifter	ALU
Global registers Local registers	Register file
Tag Trap Detection Trap Instruction Detection GS Trap Detection	TiTtGsDetection
Window Overflow Detection Window Underflow Detection	Window logic
Data Page Fault Detection Instruction Page Fault Detection	Page fault detection

Table 8.2- Shared Inputs and Outputs Merging

Functional blocks may also be merged when most or all inputs and outputs of one block are connected to only one other block. This first block provides an auxillary function for the second block. Routing is minimized by combining such blocks. A good example of this on SOAR, are the CWP, CWP+, and CWP- blocks (Table 8.3). CWP+ and CWP- increment and decrement the CWP, respectively. Their inputs are the current CWP value and their outputs become the updated CWP value when the conditions are right. D- and S2- are decrementers for the D and S2 fields of the instruction latch, respectively. PC+ increments the program counter - PC. WAIT and WAITACK are both derived from the WAIT input of the processor.

Functional Block	Circuit Block
CWP CWP+ CWP-	CWP
D- Instruction Latch (D field)	DST1
S2- Instruction Latch (S2 field)	SRC2
PC PC+	firstPC, PCIncr
WAIT WAITACK	WAIT logic

Table 8.3- Auxillary Block Merging

A final reason to merge blocks is when a subset of the inputs or outputs of one block are used solely by another block. This is perhaps the weakest reason for merging. When considering these types of merges, many possibilities exist. Tradeoffs have to be made between the routing that is eliminated and any extra routing that is generated by the merge and the resulting larger block size that may be harder to place and route than two smaller blocks. Merges of this type on SOAR are shown in Table 8.4. These consist mainly of latches for various pieces of combinatorial logic.

Functional Block	Circuit Block
Condition Comparator TempALUoutput2	Condpla
TiTtGsDetection TempTags TempOpcode	TTrap Detection
TempCWP TempSWP Window logic	WTrap Detection

Table 8.4- Some Outputs=Inputs Merging

Tables 8.1 through 8.4 summarize the functional block merges of SOAR. There were four types of merges, listed in decreasing order of importance:

1. Common circuitry
2. Shared inputs and outputs
3. Auxillary blocks
4. Some outputs = inputs

The purpose of merges is to eliminate redundant circuitry and minimize routing. This leads to a more compact, faster, and lower power processor.

## 1.2. Splitting

Merging is complemented by functional block splitting. The goals of splitting are the same as for merging – less circuitry, increased speed, fewer interconnects,



and flexibility in placement. However, a split will typically improve one or two factors and make the others worse. Thus, the tradeoffs have to be analyzed carefully.

The first type of split involves duplication of circuitry. This is done when the extra circuitry leads to faster circuits and less routing. The instruction latch was the only functional block duplicated on SOAR. One copy was placed in the datapath to generate immediate operands. The other copy was part of the control section. Its various fields are decoded to set control and word lines.

A second way to split blocks is to leave the circuitry unchanged but put parts of the block in different places. This is done if the various parts do not affect one another and interface to disjoint blocks. By placing each piece close to where it is needed, routing is reduced and speed is increased. Table 8.5 summarizes the blocks of SOAR that were split for this reason. The PSW, instruction latch and TempInstrLatch were split into separate latches for each of the fields that they included. The decoding functional block was split into blocks for control line gating and the control line drivers and blocks for the combinatorial logic - the decode PLAs.

Functional Block	Circuit Block
PSW	PSW
	shDST
	shOPC
Instruction latch	SRC1
	SRC2
	DST1
	CPIPE1
TempInstrLatch	DST2
	CPIPE2
Decoding	Decode PLAs
	Control line gating and drivers

Table 8.5- Block Splitting, Same Circuitry

Another reason to split blocks is to reduce circuitry. This leads to smaller and/or faster functional blocks. The prime candidates for this were the PLAs. Instruction decoding, condition checking, and trap detection on SOAR are done primarily with PLAs. Originally signals were assigned as outputs of the PLAs according to the functions that generated them. Six PLAs were identified with these functions (Table 8.6). As the size of a PLA increases its speed decreases. Sizes and speed estimates of the original PLAs are shown in Table 8.7.

Original PLA	Functional Blocks
Cpla1	1st decode except register access
Cpla2	2nd decode except register access
Condpla	Condition comparator, TempALUoutput2
Illpla	Illegal opcode detection
Apla	decode- register access
Tpla	Tag trap detection, trap instruction detection, TempTags, GS trap detection, TempOpcode, priority encoder

Table 8.6- Original PLA Functional Blocks

Original PLA	Inputs	Minterms	Outputs	Delay
Cpla1	10	80	39	210ns
Cpla2	7	19	10	67ns
Condpla	11	34	2	103ns
Illpla	8	8	1	46ns
Apla	18	34	24	110ns
Tpla	35	?	9	?

Table 8.7- Original PLA Sizes and Speeds

Tpla detects some of the trapping conditions and encodes the trap priority. These functions were naturally much greater than a single 2 level AND/OR function and were not able to be generated due to memory limits on our machines. The large first decode PLA - cpla1 - would have limited clock phase 1. In the interest

of being able to generate all PLAs and speed up the processor, the PLAs were reorganized into smaller, faster PLAs.

The most obvious PLAs to split are those whose outputs can be grouped so that each set of outputs is formed from inputs that do not contribute to another output set (Figure 8.2). Thus, each set of outputs will also have its own set of minterms that are derived from its inputs. When this is the case, the original PLA is split into multiple PLAs, each corresponding to one set of outputs. Each input will be an input to only one of the new PLAs.

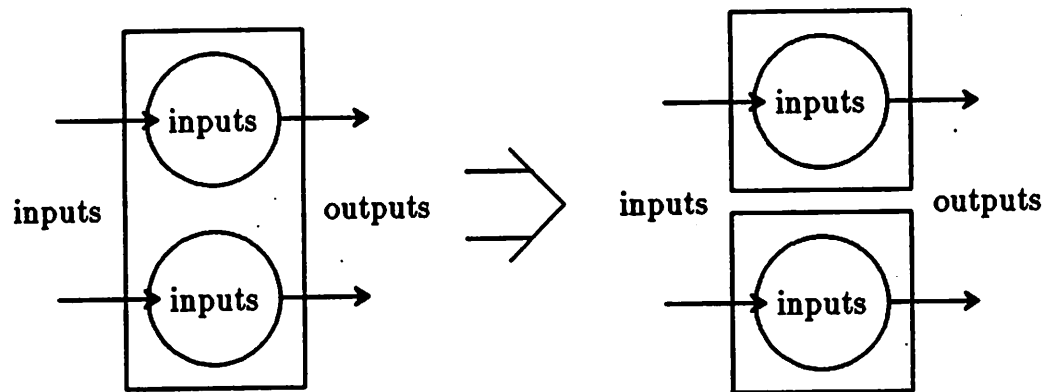


Figure 8.2- No Common Inputs PLA Split

Sometimes it is the case that the outputs can be grouped so that each set of outputs has its own distinct set of minterms, but sets of inputs are not distinct (Figure 8.3). A given input may contribute to more than one minterm group. When this is the situation, a new PLA is formed for each set of minterms. Inputs are then used by one or more of the new PLAs.

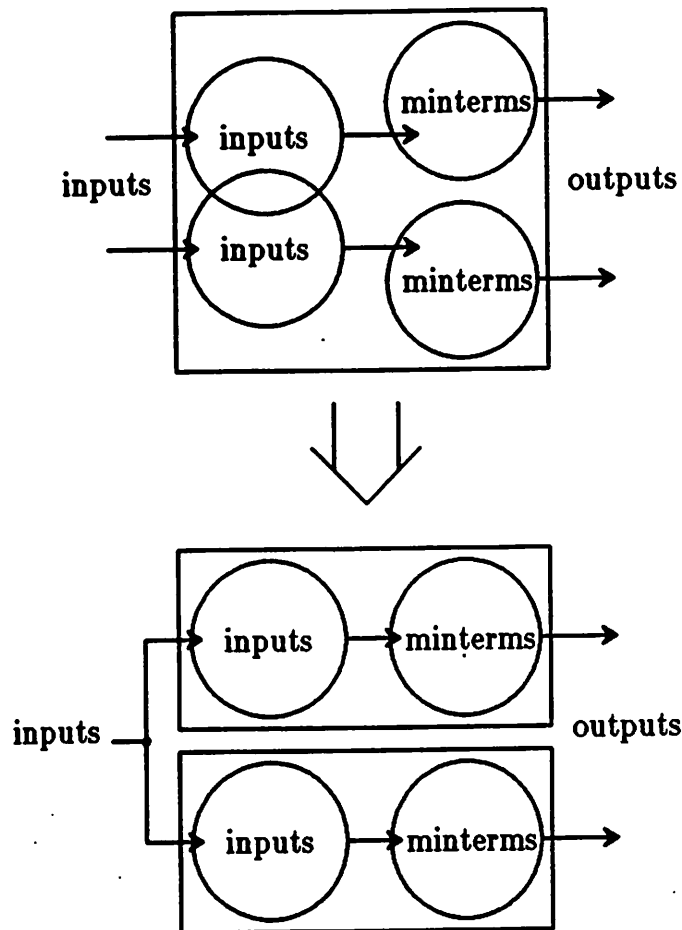


Figure 8.3- No Common Minterms PLA Split

The third type of PLA split leads to the formation of sequential PLAs. PLAs implement a 2 level AND/OR function. Logic functions that naturally are more than two levels may be reduced to this format but often at the cost of many more minterms. The PLA size may balloon and multiple sequential PLAs will be faster. This was the case with the tpla. Its most complex outputs were three 2 level AND/OR functions and an intermediate inversion. The attempt to fit it into a single 2 level AND/OR function generated enough terms to exceed machine memory limits.

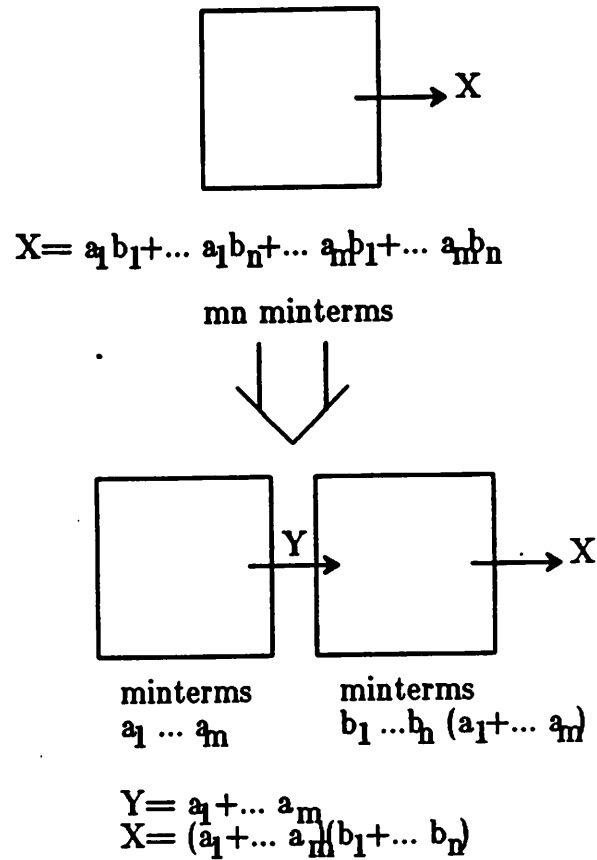


Figure 8.4- Factoring to Split PLAs

The six SOAR PLAs were split as shown in Figure 8.5 and summarized in Table 8.7. Cpla1 was split into two approximately equal PLAs by identifying two groups of minterms. They are distinct except for one minterm. Seven of the nine inputs are shared by both PLAs. Tpla was split by first identifying a distinct set of inputs and outputs that became tpla1. The remaining outputs are formed by factoring the original seven level functions into three sequential PLAs and a small amount of logic between tpla and tpla2. Apla2 was first split off from apla. It shares one input and no minterms with the other parts of apla. The remaining outputs are formed from the same inputs but may be split into two distinct sets of minterms. This resulted in apla and apla1.

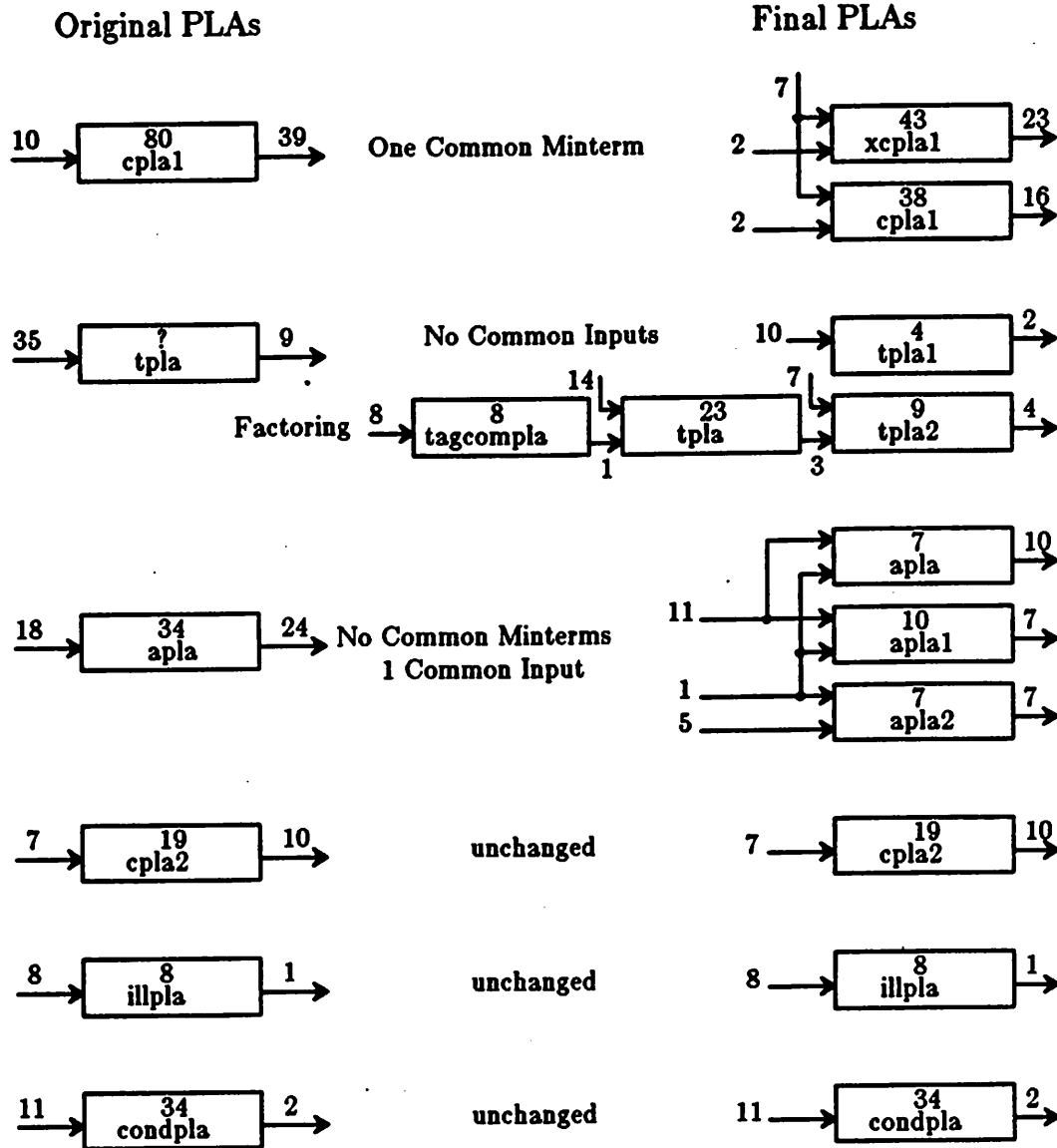


Figure 8.5- PLA Splits of SOAR

Original PLA	Final PLA	Inputs	Minterms	Outputs	Delay
Cpla1	Cpla1	9	38	16	102ns
	Xcpla1	9	43	23	85ns
Cpla2	Cpla2	7	19	10	67ns
Condpla	Condpla	11	34	2	103ns
Illpla	Illpla	8	8	1	46ns
Apla	Apla	12	17	10	42ns
	Apla1	12	10	7	50ns
	Apla2	6	7	7	32ns
Tpla	Tpla	15	23	5	54ns
	Tpla1	10	4	2	45ns
	Tpla2	10	9	4	54ns
	Tagcompla	8	8	1	42ns

Table 8.7- PLA Splitting

The slowest PLA in the decode path is cpla1. The delay through cpla1 is 102ns, making decode 107ns faster than before the split. The other important result of the split is that the trap mechanism logic can be realized with sequentially organized PLAs.



### **1.3. Summary**

Table 8.8 summarizes the original functional blocks and their circuit block assignments. It also categorizes the blocks according to their types: datapath, control, or control line driver.

Functional Block	Circuit Block	Type
Data drivers	DataOut	datapath
SignExt	SXT/DIL	datapath
Instruction latch	DIL	datapath
Instruction latch	CPIPE1	control
Instruction latch	SRC1	control
Instruction latch, S2-	SRC2	control
Instruction latch, D-	DST1	control
TempDataIn, TempStoreOperand globals, locals	LOADL Register file	datapath datapath
SHA	SHA	datapath
SHB	SHB	datapath
TempALUinput1	INAm	datapath
TempALUinput2	INBm	datapath
Insertor, Extractor	EX/INS	datapath
Add, Subtract, And, Or, Xor, Srl, Sra	ALU	datapath
Rzero	Precharge	datapath
TempALUoutput1, Nil	Destlatch	datapath
PC, PC+	firstPC, PCincr	datapath
TempPC	lastPC	datapath
TB	TB	datapath
SWP	SWP	datapath
SWPcomparator	SWPcompare	datapath
Address drivers	AddressOut	datapath
Decoders	Decoders	datapath
PSW	PSW, shDST, shOPC	control
CWP, CWP-, CWP+	CWP	datapath
TempInstrLatch	DST2, CPIPE2	control
Decoding, I/D, RD/WR	Driver1-8	driver
	Forwarding comparators, apla, apla1, apla2, cpla1, xcpla1,	control control cpla2
WAIT, WAITACK	WAIT logic	control
Condition comparator, TempALUoutput2	Condpla	control-trap
Illegal Opcode Detection	Illpla	control-trap

Table 8.8- Circuit Block Summary

Functional Block	Circuit Block	Type
Tag Trap Detection, TempOpcode Trap Instruction Detection, GS Trap Detection, TempTags	Tagcompla, tpla, TTrapLogic	cont.-trap
Window Overflow Detection, Window Underflow Detection, TempCWP, TempSWP	Window logic	cont.-trap
SWI Detection	SWI Detection	cont.-trap
Data Page Fault Detection	Page Fault Detection	cont.-trap
Instruction Page Fault Detection		
I/O interrupt Detection	I/O interrupt detection	cont.-trap
Trap	Trap	cont.-trap
Priority Encoder	tpla2	cont.-trap
Reset	Reset	control
FSHCNTL	FSHCNTL	control

Table 8.8- Circuit Block Summary (cont.)

The *desired function* of each circuit block is an output of functional block design and is used in design by the circuit level. The function of each circuit block is derived from the functional block description that the circuit block originated from. Appendix C contains functional descriptions in the form of logic diagrams for all circuit blocks.

A netlist is another output of functional block design. It is used by the interconnect design level. It can be derived from tables such as Table 7.8 for each circuit block or from the logic diagrams.

Circuit block diagrams and a floorplan of the processor were then developed. Figure 8.6 shows the circuit block diagram of the realized SOAR datapath and its bus structure. The control line driver interface between the control and datapath sections is shown in Figure 8.7. The trap and skip mechanisms of the control section are shown in Figure 8.8. Figure 8.9 shows the remaining parts of the control section. This includes the PSW, instruction latch and some register access decoding.

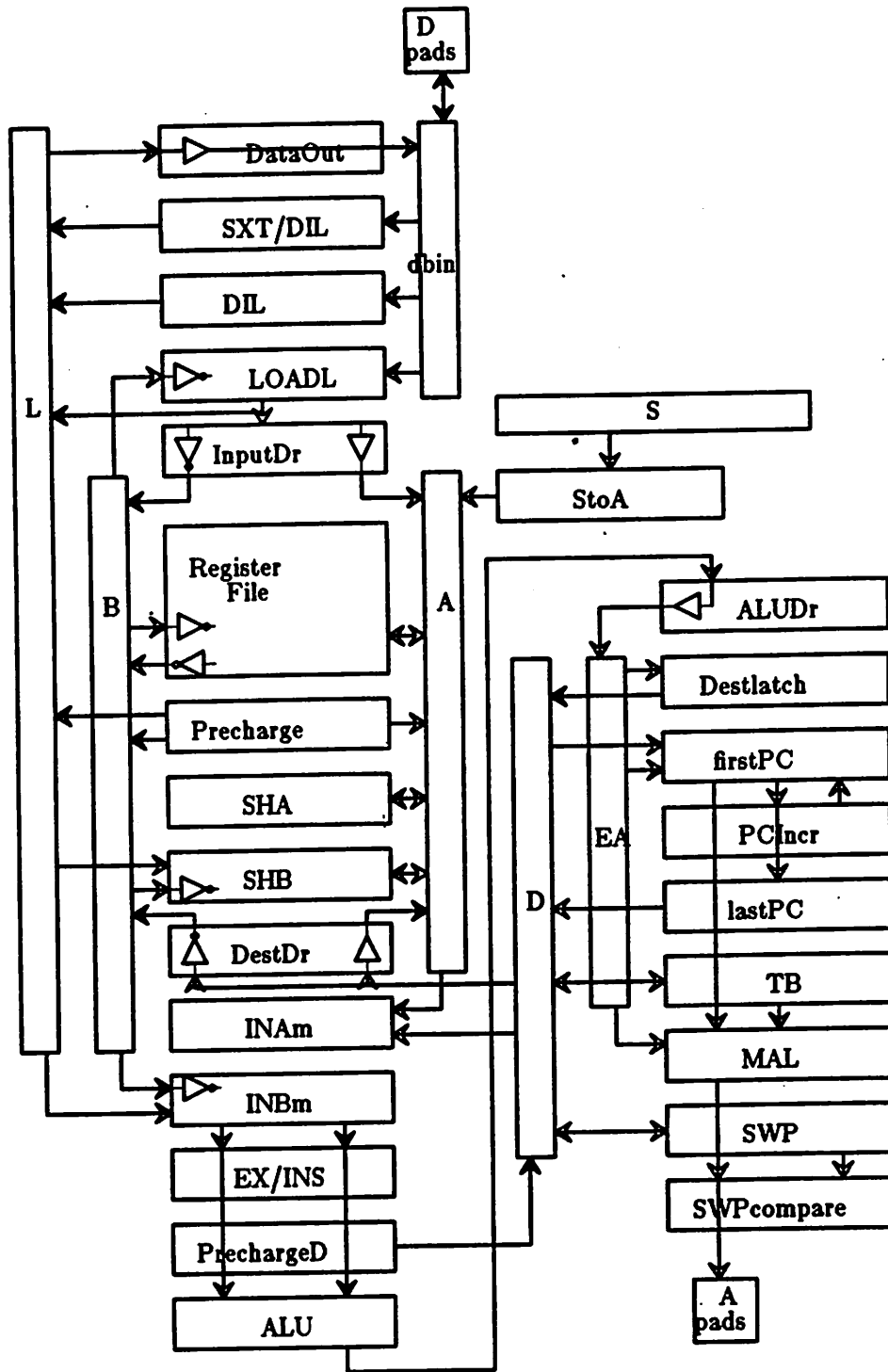


Figure 8.6- Datapath- Data Transfer (Realized SOAR)

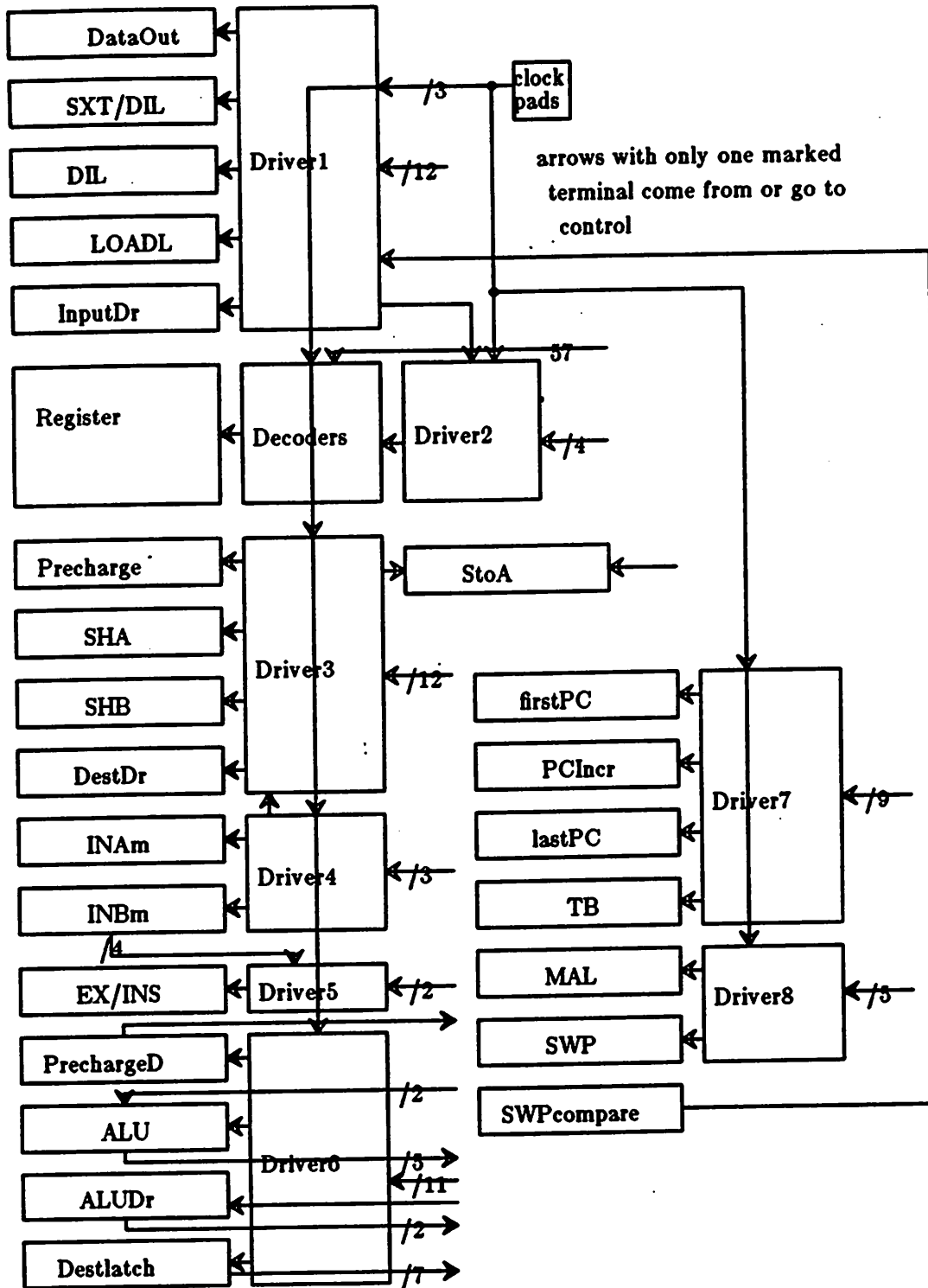


Figure 8.7- Datapath- Control Lines (Realized SOAR)

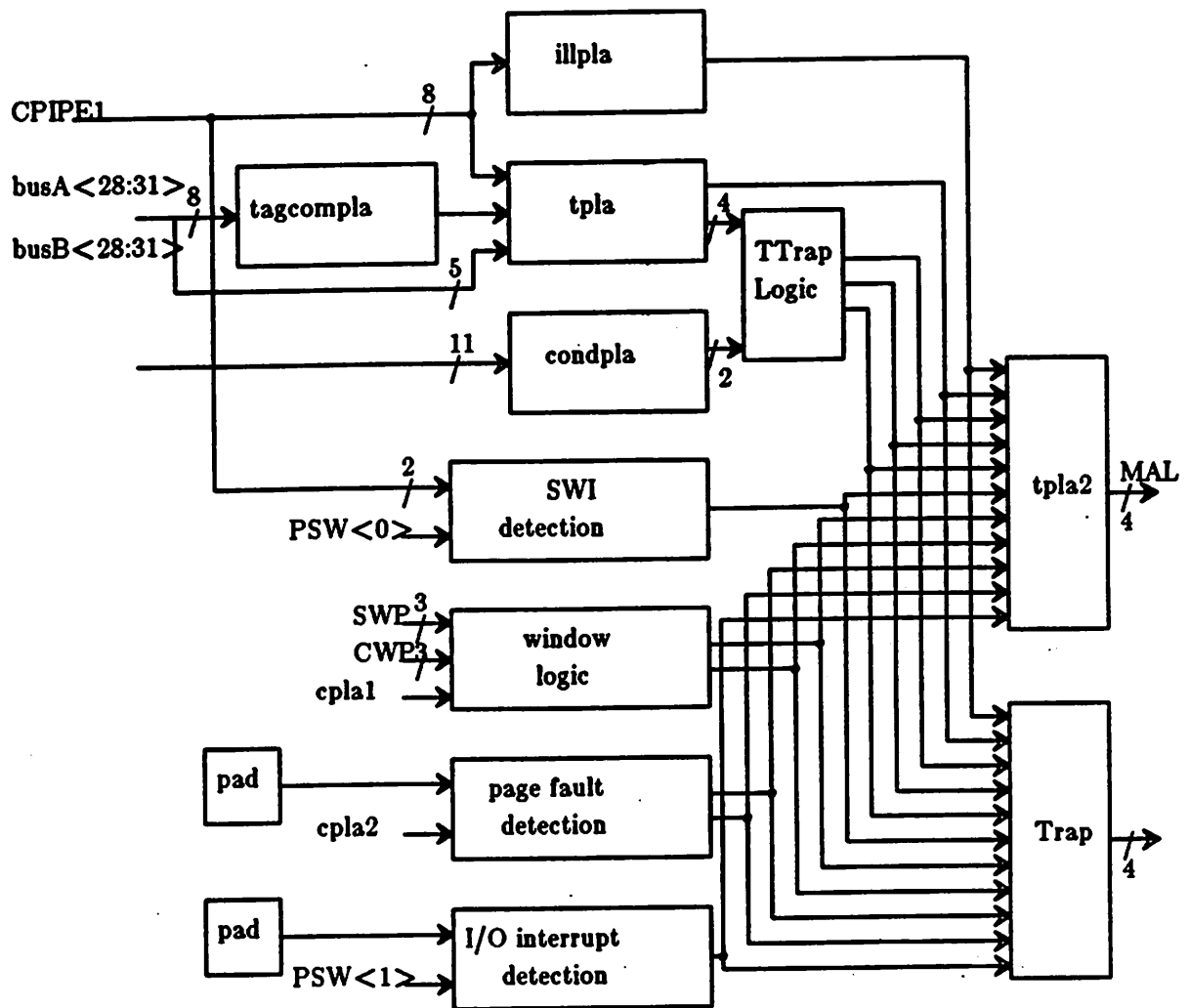


Figure 8.8- Control- Trap Section (Realized SOAR)

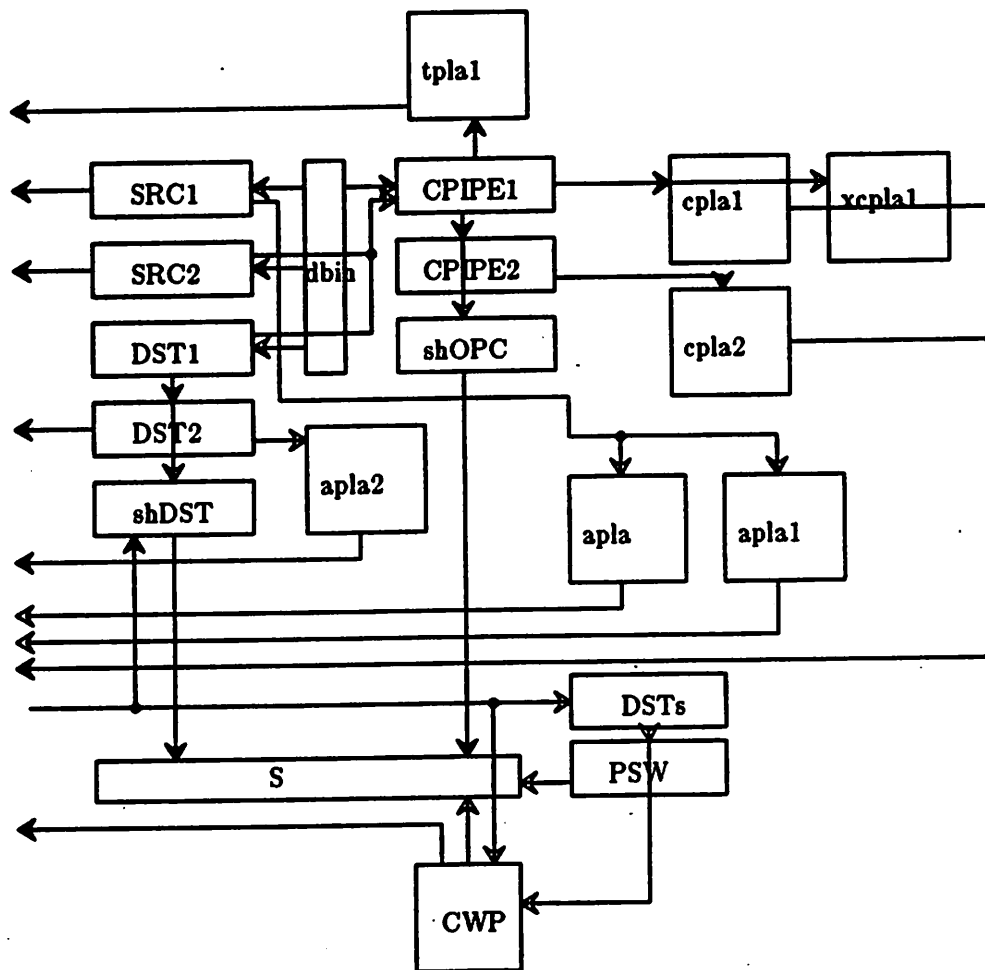


Figure 8.9- Control- Other (Realized SOAR)

The realized SOAR datapath contains a memory address latch – MAL. The MAL is not on the list of functional blocks that were developed through this methodology. The PC and TempALUoutput1 latch duplicate the function of the MAL. Thus, the MAL could have been omitted. This was the only difference between the circuit block diagrams of the realized SOAR processor and a SOAR processor developed through this methodology, using the realized SOAR pipeline.

## **2. Functional Block Analysis**

Functional block analysis leads to a comparison of desired values for the quantities being evaluated, with predicted values for the same quantities. Desired values were arrived at through microarchitecture analysis. After functional block synthesis the circuit block and interconnect structure can be determined for all signals. An estimated value for each characteristic being analyzed is determined for each circuit block and interconnect. When analyzing speed, power, and area, estimates of the delay, power dissipation, and area of each circuit block or interconnect is needed. These estimates are totaled up to arrive at an estimate of the speed of each signal, total power, and total area.

### **2.1. Speed Analysis**

On the SOAR project complete functional block analysis of speed was not done at this stage in the design. Functional block synthesis was followed by synthesis at the circuit and interconnect levels with no intervening speed analysis. Detailed speed analysis was then done and revealed several unacceptably long delays. At this time functional block analysis was done to discover the reasons for these delays. For the purpose of illustrating this methodology, functional block speed analysis, as it should be done, will first be discussed. This will be followed by the results of functional block speed analysis as it was done. It was done correctly but if it had been done at the proper time – before detailed circuit and interconnect synthesis – it would have saved many hours of detailed redesign at the circuit and interconnect levels.



### 2.1.1. Method

As previously discussed, each input signal to a block is identified with a processor activity. Microarchitecture analysis identified the start and finish of each of these activities. This defines the allowed settling time of the input signal corresponding to an activity. During speed analysis at the functional block level, the circuit blocks and interconnects that the signal passes through when settling are first identified. This is done with the help of the netlist and leads to diagrams such as Figure 8.10, for each signal. In Figure 8.10 the signal being analyzed passes through three blocks and the interconnects between them as it settles. To predict the settling time, the individual block and interconnect delays for this block structure are totaled. A second part of functional block analysis examines these block structures to uncover unnecessary components and excessively long paths. All this is done in the *analyze functional block* step (Figure 8.1).

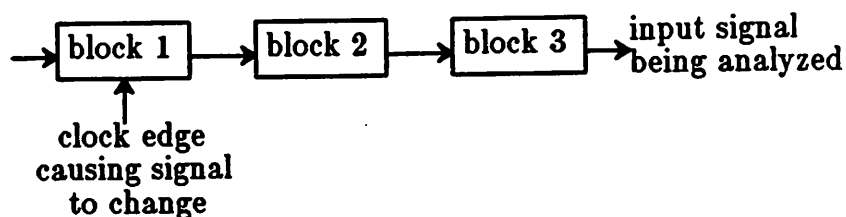


Figure 8.10- Block Structure of a Signal

The design process then moves to the *analyze functional block vs. microarchitecture* step. The total delays from the previous step are compared to the desired delays. The desired delays were found from previous microarchitecture analysis. They were expressed in terms of allotted clock phases for the settling of input signals (Figure 7.9).

This speed analysis can best be illustrated with an example. The TempALUinput2 functional block became the INBm circuit block. It has four inputs – busBtoINB, busLtoINB, busB, and busL – as previously discussed. Using the netlist it can be seen that the ungated version of busBtoINB comes from xcplal. The function of xcplal is examined and shows that inputs from CPIPE1 form the busBtoINB output (Figure 8.11). The rising phase 1 clock edge at CPIPE1 signals the start of this decode activity (Figure 7.10). Similar analysis using the netlist and circuit block functions results in the block structures for the remaining input signals to INBm – busLtoINB, busL, and busB. Two decoded signals are combined to form busLtoINB. BusL is driven with information from either the sign extender – SXT/DIL – or the instruction latch – DIL. Either SXT/DIL or DIL is enabled by a control line from the Driver1 circuit block. Data from one of the registers, Destlatch, or LOADL is put onto busB during the read activity. These blocks are also enabled by control lines from driver blocks.

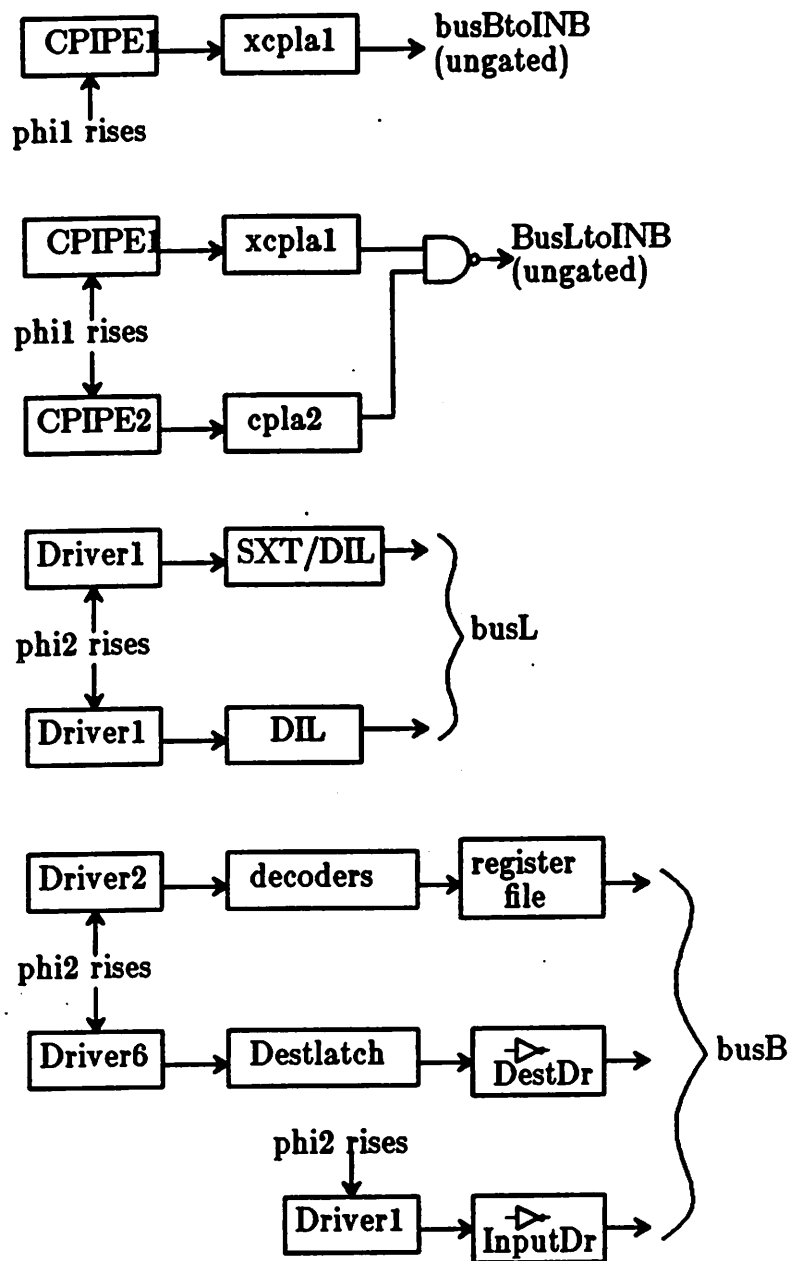


Figure 8.11- Functional Block Structure of Inputs to TempALUinput2

Estimates of the delays through these circuit blocks and interconnects between them are needed to compute the settling time of each signal (Table 8.9). Some of this information is available from the characteristics of circuits that were designed during the preliminary phase. PLA delay estimates are readily available due to automatic layout generation of the PLAs from logic descriptions, and

timing verification of the layout using CRYSTAL. Other delays are unknown and very rough guesses must be used. Enabling of the control latches - CPIPE1 and CPIPE2 - should be relatively fast. Interconnect speeds depend on the capacitive loads and composition of the interconnects. This can vary greatly depending on the final layout and is therefore left unknown, as indicated by the '?'. Delays for circuits that drive buses, control, and word lines were calculated using the approximate loads, during preliminary circuit design. Thus, circuit block and interconnect delays are lumped together for these situations.

Block or Interconnect	Delay (ns)	Origin
CPIPE1	~0	guess
Xcpla1	100	CRYSTAL
CPIPE2	~0	guess
Cpla2	70	CRYSTAL
Driver1 + control line	20-40	preliminary phase
SXT/DIL + bus	30-45	preliminary phase
DIL + bus	30-45	preliminary phase
Driver2 + control line	20-40	preliminary phase
Driver6 + control line	20-40	preliminary phase
Decoder + word line	20-40	preliminary phase
Register file + bus	20-25	preliminary phase
Destlatch + bus	30-45	preliminary phase
DestDr + bus	20-45	preliminary phase
InputDr + bus	20-45	preliminary phase
Interconnect- CPIPE1 to xcpla1	?	
Interconnect- xcpla1 to INBm	?	
Interconnect- CPIPE2 to cpla2	?	
Interconnect- cpla2 to INBm	?	

**Table 8.9- Circuit Block and Interconnect Speed Estimates**

Two types of analysis are done with this block structure and delay information. The most obvious analysis is to sum up the delays for each signal. This leads to an estimation of the settling times (Table 8.10).

Signal	Delay
BusBtoINB	100ns + 2 interconnects
BusLtoINB (xcpla1)	100ns + 2 interconnects or
(cpla2)	70ns + 2 interconnects
BusL	50-85ns
BusB (register file)	60-105ns or
(Destlatch)	70-130ns or
(LOADL)	40-85ns

Table 8.10- Input Signal Delays (INBm)

The second type of analysis is more subjective. The block structures for each signal are examined to identify unnecessarily long paths for signal settling (Figure 8.11). In this example the control lines – busBtoINB and busLtoINB – involve a minimum number of blocks. A latch holds the opcode. This opcode goes through one decode PLA and then to its destination. The number of blocks needed to drive busL is also minimal – a driver block that enables the block holding the data to be read onto busL. Two of the three paths to drive busB are also minimal – the register file and InputDr paths. In the third path the Destlatch first drives busD then the DestDr block transfers the data from busD to busB. This last step is unnecessary if the microarchitecture can be rearranged so that INBm can be loaded from busD. Referring to the first part of functional block analysis (Table 8.10), it can be seen that the delay of this third path is estimated to be the longest settling time for busB. Thus, both types of functional block analysis indicate that the processor speed can probably be improved by rearranging the microarchitecture so that busD can load INBm if this is found to be a limiting

critical path when microarchitecture versus functional block analysis is done.

These two types of delay analysis should be done for the input signal of all circuit blocks.

Results of this analysis are then compared with the desired critical paths from microarchitecture speed analysis (Ch.7, Sec.2.1) and redesign is done as necessary (Table 8.11). All signals except busB, for the INBm circuit block compare favorably with the previous microarchitecture analysis. BusB is estimated to take longer than the allotted phi2 phase - 50-85ns. The unminimal busB path from the Destlatch requires 70-130ns. Even the minimal bus B path from the register file needs 60-105ns. Thus, even with the proposed microarchitecture rearrangement the specification will not be met. At this point all critical paths for phi2 should be inspected. If phi2 must be extended even further due to some other signal it may not be worthwhile to speed up the reads onto busB. If these reads prove to be the limiting path for phi2 then either phi2 must be extended to accommodate them or faster circuits must be designed so that the specification can be met or some compromise between these two solutions must be agreed upon.

Signal	Allotted Clock Phases	Estimated Delay	Compare
BusBtoINB	phi1 + underlap: 170-290ns	100ns + 2 interconnects	ok
BusLtoINB	phi1 + underlap: 170-290ns	100ns + 2 interconnects	ok
BusB	phi2: 50-85ns	70-130ns	not ok
BusL	phi2: 50-85ns	50-85ns	ok

Table 8.11- Microarchitecture vs. Functional Block Analysis

### 2.1.2. SOAR Speed Analysis

Functional block speed analysis of the PLAs was done after functional block synthesis on the SOAR project. As previously described, the PLA functions were originally grouped into six PLAs. Three of these PLAs were then split into smaller PLAs. As described in section 1.2 of this chapter, the motivations for two of these splits – cpla1 and apla – were the long delays through the original PLAs. Tpla was split because it was unable to be generated. It would have been extremely long if it was generated and consequently, unacceptably slow. Thus, the PLAs followed the design steps of Figure 8.1:

1. Synthesize functional blocks
2. Analyze functional blocks– not ok
3. Synthesize functional blocks
4. Analyze functional blocks– ok

To analyze the PLA blocks, the layout was used since it was easily available due to CAD tools.

Other functional block speed analysis was done after detailed circuit and interconnect design. It revealed unacceptable bottlenecks on the chip and resulted in microarchitecture redesign. If done at the right time, it would have identified the problems earlier, before the time consuming circuit and interconnect design and layout had been completed. As previously discussed (Ch.6, Sec.1.7), multicycle instructions on SOAR were implemented by an internally generated series of single cycle opcodes. This was the result of the microarchitecture redesign after functional block speed analysis. Before the analysis an interlock mechanism was used to hold the opcode of the multicycle instruction in the instruction latch for the required number of cycles (Figure 8.12).



Interlock		Internal Opcodes
loadm	cycle 1	loadm
loadm	cycle 2	load6
loadm	cycle 3	load6
loadm	cycle 4	load5
loadm	cycle 5	load4
loadm	cycle 6	load3
loadm	cycle 7	load2
loadm	cycle 8	load1
loadm	cycle 9	load0

Figure 8.12- Instruction Latch Opcodes for a Multicycle Instruction- Loadm

Figure 8.13 shows the original interlock mechanism. CPIPE1 is the instruction latch holding the multicycle opcode, loadm. This opcode is decoded by a decode PLA and a signal indicating a multicycle opcode is being processed, is generated for the interlock logic. DST1 holds the register specifier. During loadm registers are written, with the loaded data, in descending sequential order. So the new value of DST1 comes from the decremter during loadm interlocks. The interlock logic generates the signal that selects the decremter for DST1 when a multicycle opcode is being processed. When DST1 is zero, the multicycle instruction has been completed and various normal processor activities - increment the PC, load CPIPE1, load the MAL from the PC, etc. - are resumed. The control lines for these activities are set by a decode PLA.

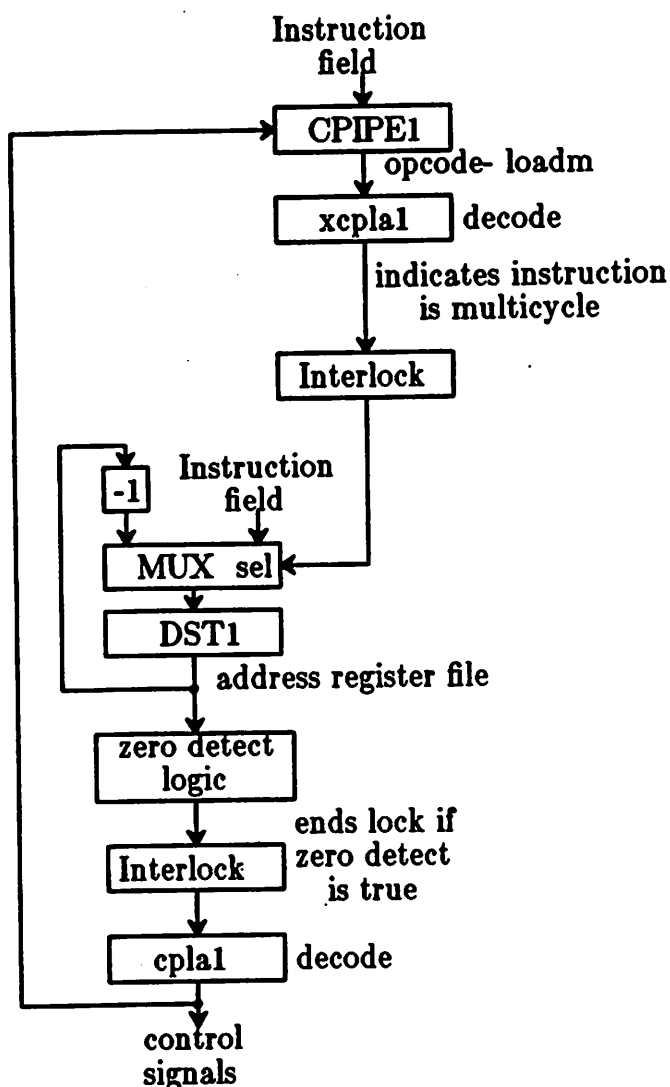


Figure 8.13- Interlock Mechanism Block Structure

There were several control lines that were set by this interlock mechanism. This path is composed of seven circuit blocks and the routing between them. As these signals settle, they must go through the largest decode PLAs – cpla1, xcpla1 – twice, the interlock logic twice, and assorted other pieces of logic. It was not possible to put all seven blocks in close proximity. Therefore, interconnect loading and delays are significant. When this path is compared to the minimal decoding paths of busBtoINB and busLtoINB (Figure 8.11), it is apparent that it is extremely long.

The internal opcode mechanism for interlocks is shown in Figure 8.14. The instruction latch, CPIPE1, holds the opcode. However, unlike the interlock situation, this opcode changes during the multicycle instruction (Figure 8.12). All control lines are set according to the opcode. The opcode for each cycle is unique and therefore the end of the multicycle instruction is indicated by the last opcode - load0. Control lines that increment the PC and load CPIPE1 from the instruction field are enabled when this last opcode is decoded by cpl1, and disabled during the other opcodes of the multicycle instruction. The decremter is used to address the register file, just as with the interlock method. The decremter also supplies the unique piece of the internal opcode for every cycle of the multicycle instruction. The internal opcode is loaded into CPIPE1 at the start of the each cycle. In this way a unique internal opcode is generated for each cycle of the multicycle instruction.

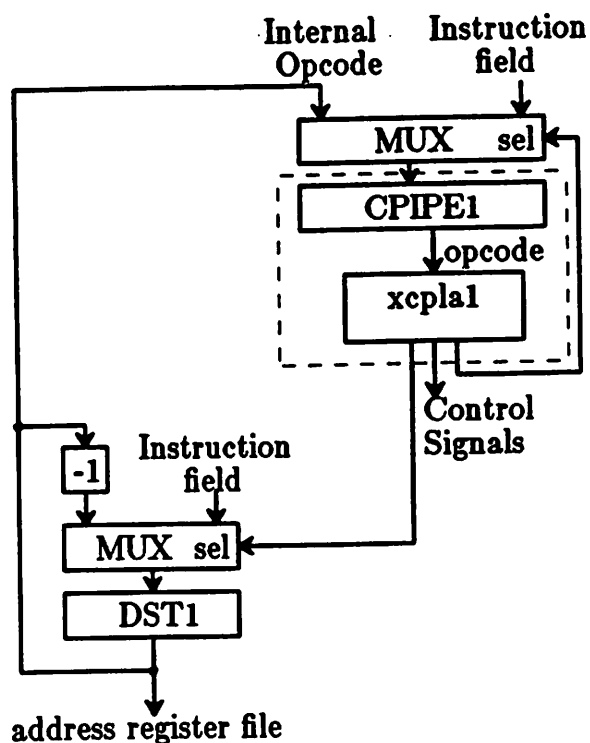


Figure 8.14- Internal Opcode Mechanism Block Structure

Using the internal opcode mechanism, control lines that were previously set with the interlock mechanism, now settle through a path composed of CPIPE1 and one large decode PLA – indicated by the dashed lines of Figure 8.14. This is the minimal block structure needed to form control lines from an opcode.

A second non-minimal block structure for instruction decoding occurs when control lines must go through two sequential PLAs, as they are being set (Figure 8.15). This is the situation for control lines that are involved in routing the ALU output (Destlatch) or data input latch (LOADL) directly to the ALU input – forwarding – and when writing to the special registers. Forwarding occurs when a source register for an instruction in the read phase, is the same as the destination register of the previous instruction. The source of the forwarded data is either the Destlatch or LOADL latch, depending on whether the previous instruction was a register to register instruction or a load. Cpla2 determines this, according

to the previous opcode that is in CPIPE2. The signal from cpla2 is combined in apla1 with the signals from the forwarding comparators to generate the forwarding control lines. These lines are used during the read phase – phase 2 – and must therefore have settled by the start of phase 2 (Table 8.12). When compared with the minimal decode that has the same time allotment, the forwarding lines have a slightly longer PLA delay and an extra interconnect delay. This indicates a potential bottleneck to decoding. Therefore, it was decided that the two PLAs – cpla2 and apla1 – should be placed close together. This requirement becomes an input to interconnect design.

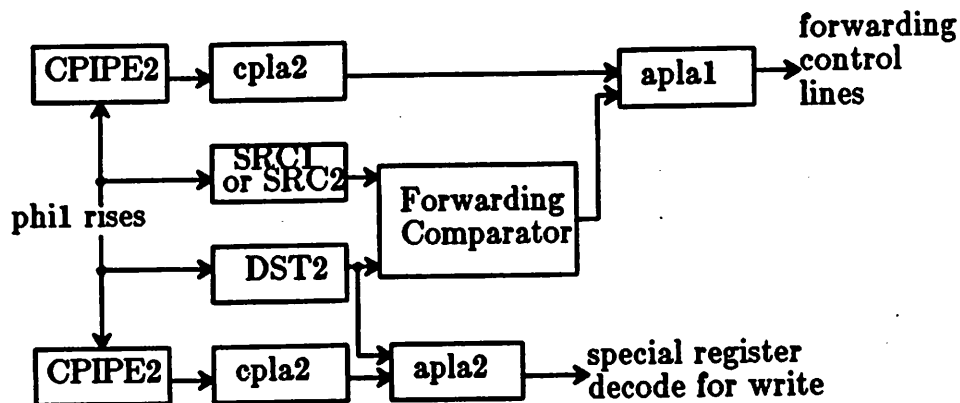


Figure 8.15- Another Non-minimal Block Structure for Instruction Decode

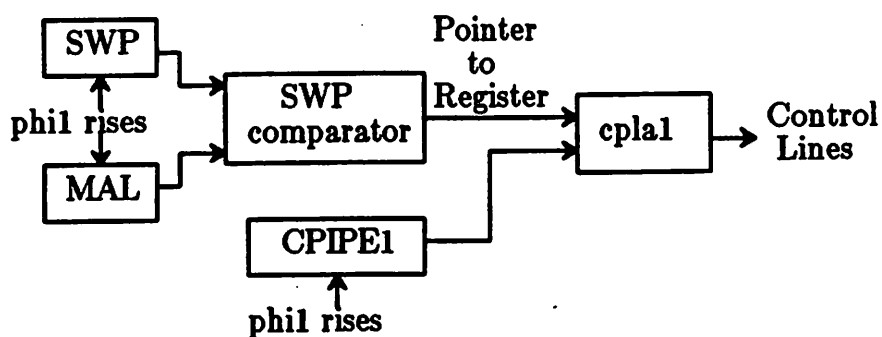
Signal	Allotted Clock Phases	Estimated Delay	Compare
Forwarding	$\text{phi1} + \text{underlap} = 170\text{-}290\text{ns}$ .	$120\text{ns} + 3\text{interconnects}$	ok
Write specials	$\text{phi1} + \text{phi2} + 2 \text{ underlaps} = 240\text{-}395\text{ns}$	$102\text{ns} + 3\text{interconnects}$	ok
Pointer to reg.			
Original	$\text{phi1} + \text{phi2} + 2 \text{ underlaps} = 240\text{-}395\text{ns}$	$350\text{ns} + 2 \text{ interconnects}$	not ok
Revised	$\text{phi1} + \text{phi2} + 2 \text{ underlaps} = 240\text{-}395\text{ns}$	$250\text{ns} + 1 \text{ interconnect}$	ok
Trap			
Thru cpla1	$\text{phi3} + \text{phi1} + \text{phi2} + 2 \text{ underlaps} = 340\text{-}495\text{ns}$	$300\text{ns} + 6 \text{ interconnects}$	not ok
Thru cpla1	$\text{phi3} + \text{phi1} + 1 \text{ underlap} = 270\text{-}390\text{ns}$	$300\text{ns} + 6 \text{ interconnects}$	not ok
Not thru cpla1	$\text{phi3} + \text{phi1} + 1 \text{ underlap} = 270\text{-}390\text{ns}$	$200\text{ns} + 4 \text{ interconnects}$	ok

Table 8.12- Functional Block Analysis of Key Paths

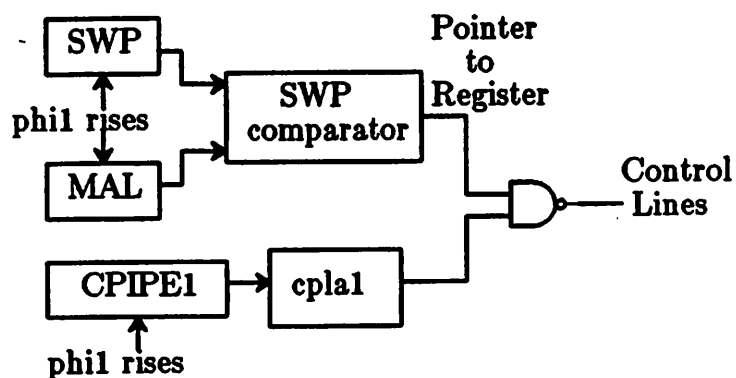
Writing to the special registers also requires two sequential PLAs (Figure 8.15). Cpla2 decodes the opcode and indicates a write is to be done. Apla2 is the decoder for the special registers. This write is done in phase 3. Thus, both phi1 and phi2 are allotted to the decode (Table 8.12). This decode requires 100ns of PLA delays and three interconnect delays. It should easily fit into its 240 to 395ns time slot.

A third non-minimal block structure and therefore potential problem for instruction decoding, existed with control lines that depended on the pointer to register signal (Figure 8.16). Pointer to register is asserted whenever a memory access refers to an on chip register. The SWP comparator is a 24 bit ripple adder that includes zero detection on the output. This comparator was estimated to have a 250ns delay. The control lines that depend upon pointer to register must settle by the start of phase 3 (Table 8.12). The original microarchitecture

proposal had these control lines settling through the SWP comparator and cpla1 sequentially. This was estimated to have a delay of 350ns plus two interconnect delays. The allotted time for settling was 240-395ns. This indicated a problem and the microarchitecture was rearranged so that signals would go through the comparator and cpla1 simultaneously (Figure 8.16b). The two resulting signals are combined at the control line drivers to form the control signals. With this modification the overall delay was reduced by 100ns and an interconnect delay (Table 8.12).



(a)- Before Microarchitecture Rearrangement



(b)- After Microarchitecture Rearrangement

Figure 8.16- A Third Non-minimal Block Structure for Instruction Decode

The block diagram of the trapping mechanism shows seven possible paths that may all result in a trap (Figure 8.8). The path that causes the longest delay is the path through the condition PLA. The block structure for this is shown in Figure 8.17. Signals on this path start to change at the beginning of phase 3 as the ALU compares the two operands. The condition PLA evaluates the ALU result according to the type of comparison being done, and generates a signal indicating whether a trap should occur or not. When a trap occurs an internal opcode is placed into the opcode latch (CPIPE1) and then decoded to set control lines. All but two of these control lines are used in phase 3. The allotted time for these control lines is 340 to 495ns (Table 8.12). The estimated delays of the PLA outputs are 300ns plus the delays of six interconnects. This is close to the limit, making it important to minimize the interconnect delays. Two control lines must be generated by phase 2. This would not be possible if they were PLA outputs. Therefore, they are generated directly from the trap signal.

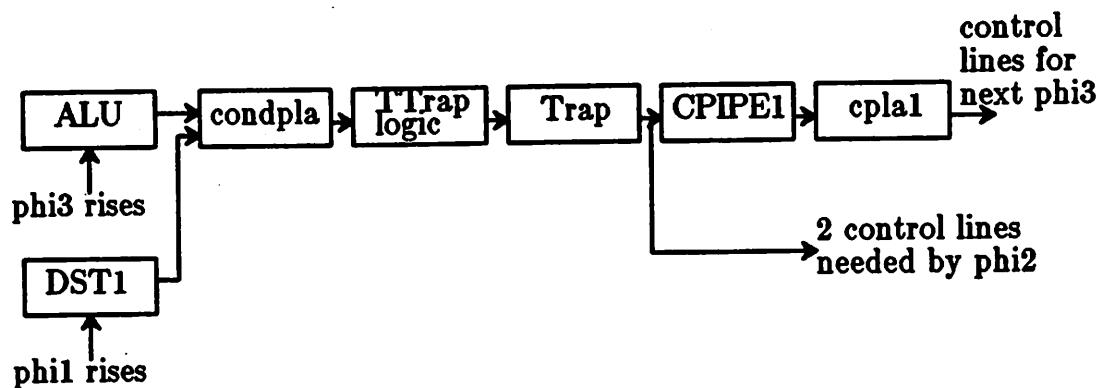


Figure 8.17- Longest Block Structure of the Trap Mechanism

Functional block speed analysis provided insights into the bottlenecks of SOAR when speed analysis of the completed layout was carried out. This led to revisions in the microarchitecture and further functional block analysis. When this analysis met the requirements of the microarchitecture, the circuit and



interconnect design and layout was redone. This redesign proceeded according to the methodology.

## **2.2. Power Analysis**

Power analysis of SOAR was done at the functional block level after functional block synthesis. DC power dissipation levels of the circuits developed during the preliminary phase, were used as inputs to functional block analysis. AC power dissipation for bus and interconnect loads of approximately 1pF, were calculated based on a 500ns cycle time and proved to be negligible. Table 8.13 summarizes the estimated power dissipation for SOAR from functional block analysis. The large register file and assortment of PLAs contributed most to power consumption – 25% and 21% respectively. At this point the total power estimate for SOAR Was 600mW.

Block Type	Power (mW)	%
Datapath latches	60	10
Bus drivers	22	4
Register file	150	25
Word line drivers	86	14
Control line drivers	38	6
Decoders- register file	14	2
EX/INS	4	1
ALU	11	2
Control latches	11	2
Pad drivers	77	13
PLAs	129	21
<b>Total</b>	<b>602</b>	<b>100</b>

Table 8.13- Power Estimate from Functional Block Analysis

## **Chapter 9**

### **Circuit and Interconnect Design**

#### **SOAR Case Study**

Once a functional block design that is expected to meet the microarchitecture requirements has been proposed, design moves to the circuit and interconnect levels. Figure 9.1 shows the section of the methodology that corresponds to circuit and interconnect design. Circuit and interconnect design are done simultaneously due to the close ties between the two levels. Both levels contribute to the final layout of the processor. During synthesis the sizes and geometries of the two individual levels must fit together and meet the area specification of the chip. Port placement on the circuit blocks influences interconnect layout. During analysis, interconnect loading affects the speed of the circuit block outputs. Circuit block inputs at the terminations of interconnects affect the maximum possible speed of the interconnect.

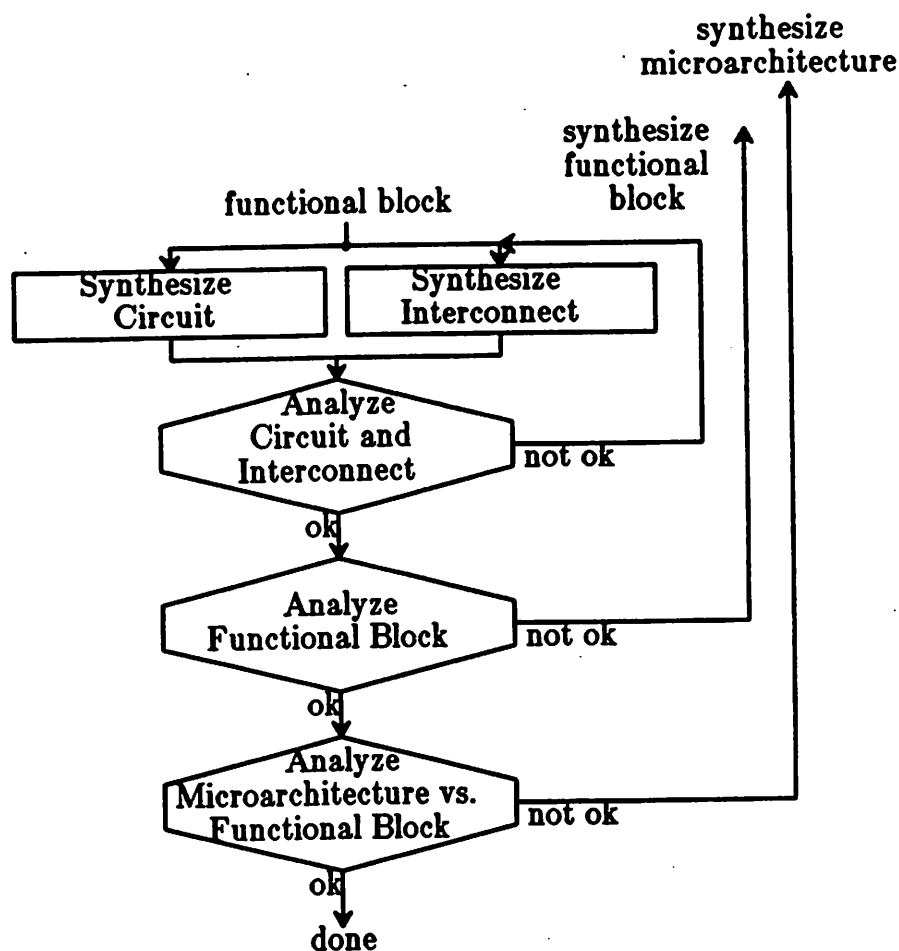


Figure 9.1- Circuit and Interconnect Design

### 1. Circuit and Interconnect Synthesis

The first design step at the circuit and interconnect level is the synthesis of the circuit blocks and the routing between them (Figure 9.1). Figure 9.2 shows the flow diagram for this step. The *desired functions* inputs to circuit design is the set of logic diagrams that describe the circuit blocks. The netlist input to interconnect design is derived from the circuit block descriptions. Interconnect characteristics take the form of speed requirements for key interconnects. These are interconnects on signal paths for which it is questionable as to whether they will meet their timing requirements. Special attention to interconnects on these

paths can make the difference. Requirements were:

1. Minimize routing between cpla2 and apla1.
2. Minimize routing on the trap instruction detection path.
3. Carry lines made entirely of between the two datapath sections.

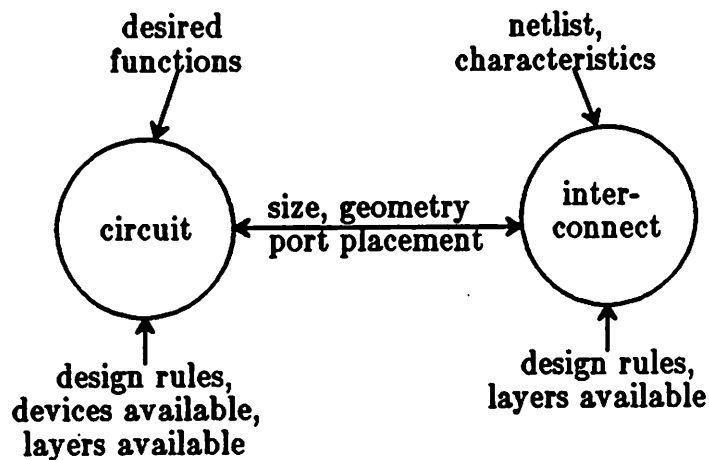


Figure 9.2- Circuit and Interconnect Synthesis

Circuit design consisted of customizing the circuits designed during the preliminary phase so that the logic diagrams for all circuit blocks were realized. Datapath cells were pitch matched, and their inputs and outputs were placed so that adjacent blocks could be butted up against each other. The entire chip was routed according to the netlist and the routing requirements from previous design steps. This completed the first cut at circuit and interconnect synthesis.

## 2. Circuit and Interconnect Analysis

After this design moved to the step that analyzed the circuit blocks individually. Theoretically interconnects should have been analyzed individually also, but there was no clear way to do this for speed analysis. Figure 9.3 shows the complete flow diagram for this step. As previously discussed, speed was the

primary characteristic of interest. Area and power specifications for SOAR were much looser. Thus, analysis concentrated on speed analysis of the circuit blocks, as shown by the dashed lines in Figure 9.3.

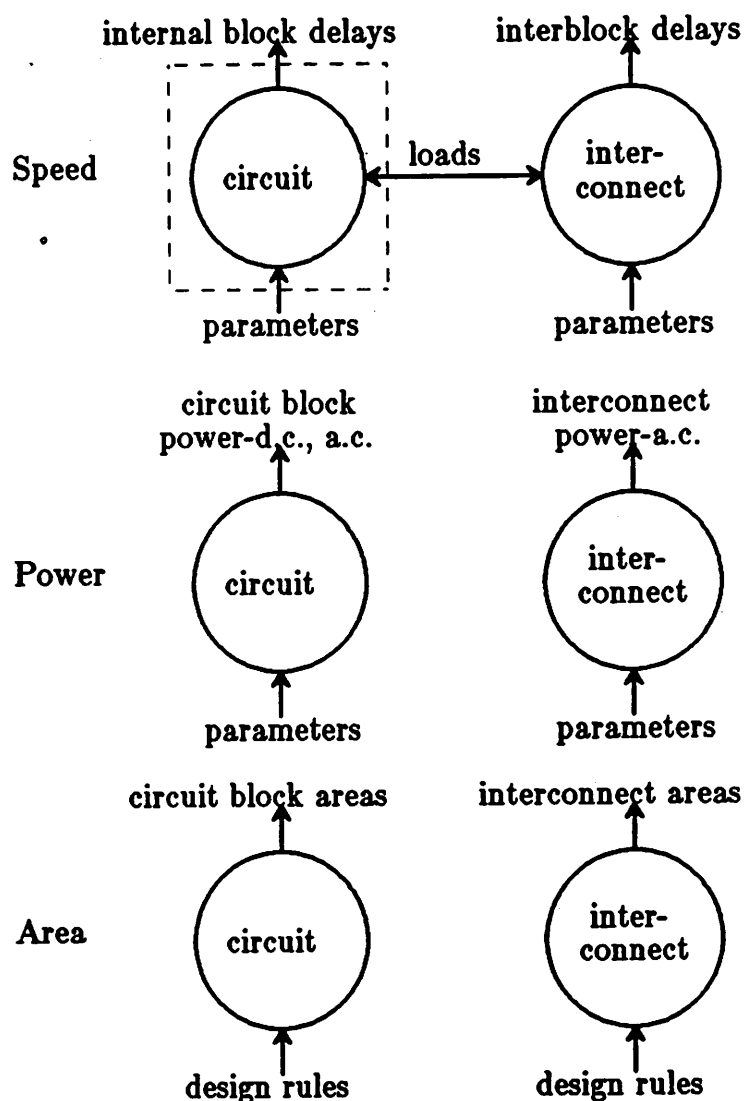


Figure 9.3- Circuit and Interconnect Analysis

Circuit blocks were typically 32 bit blocks and therefore were too complex for practical SPICE analysis. The timing verifier - CRYSTAL - was used to predict circuit block speeds [Oust85]. The goal of circuit analysis during this step, was to analyze the speed from the extracted circuit layout, for all circuit blocks

on the signal paths outlined during functional block analysis. The layout adds many parasitic elements, both capacitances and resistances, to the circuits. These unfortunately, do not always have a negligible affect on performance. Since layout does not necessarily exist during the preliminary phase, they were not accurately modeled in early speed estimates. Now that complete layouts exist for circuit blocks, more accurate speed estimates can be made. PLAs were generated automatically during previous steps and therefore their speeds have already been estimated from the layout (Table 8.7). Speeds according to CRYSTAL, of key circuits are summarized in Table 9.1.

Block	Estimated Delay	Allotted Delay
ALU	247ns	100ns
Inserter/extractor	157ns	100ns
SWP comparator	601ns	240-395ns
Decoders	438ns	170-290ns
Sign extender	305ns	170-290ns
PC incrementer	350ns	395-515ns

Table 9.1- Speed Estimates of Key Circuits (CRYSTAL)

## 2.1. ALU

The first major problem was the ALU. The original ALU layout had a simulated delay of 400ns. This was primarily due to some large diffusion areas in the carry circuitry, and their associated parasitic capacitances – node Y in Figure 6.5 [Kong84]. Figure 9.4 shows the modified carry circuitry. The parasitic

capacitance of node Y was greatly reduced in the layout of this circuit. The output node of the ALU also had a large capacitance associated with it due to the layout. Therefore, a buffer was added to drive the ALU output (Figure 9.6). With these changes and other minor modifications, the ALU delay was reduced to 247ns. This is still more than the desired 100ns, but was accepted for this design.

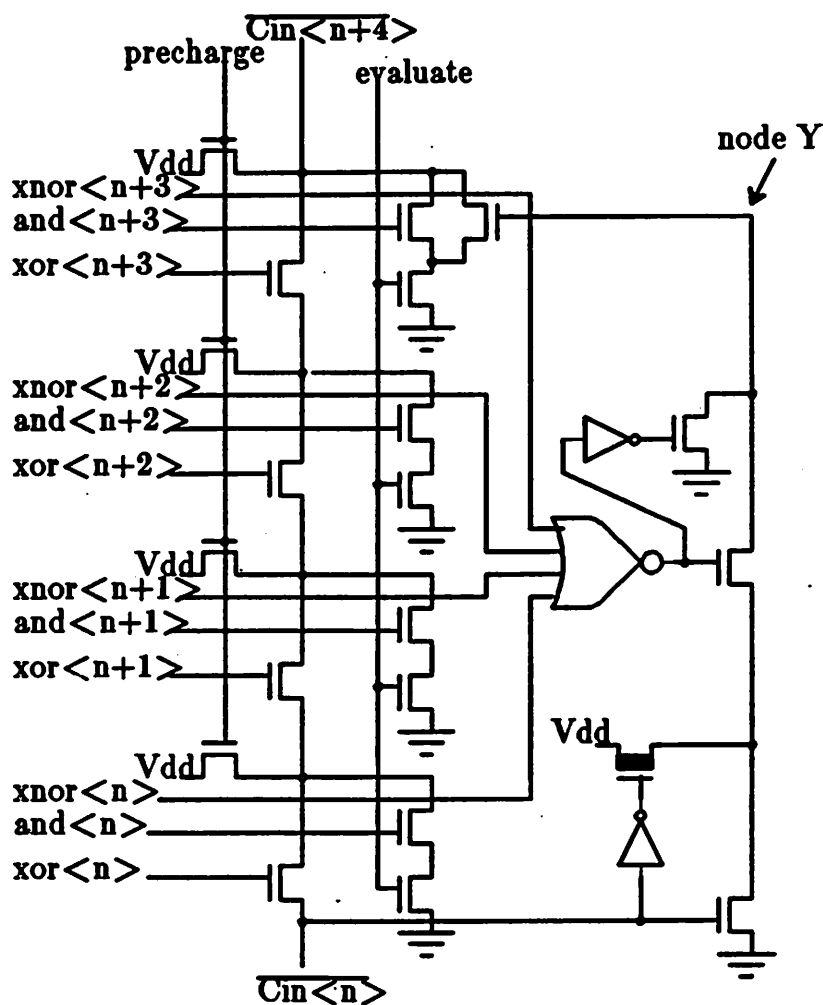


Figure 9.4- SOAR ALU

Table 9.2 lists the circuit components used during addition, with this type of ALU. As previously discussed, the ALU is organized into eight four bit blocks - nibbles. Each nibble has a carry bypass line used to rapidly propagate the



incoming carry across it, when conditions are correct - path a in Figure 9.5. Carry propagation across a nibble was reduced from 43ns to 14ns by including this bypass scheme. Each ALU output bit requires an input carry to compute its result value. Within a carry block, the slowest input carry to be generated, is the one that requires propagation across three bits - path c in Figure 9.5. The delay of this path is 32 ns. After the input carry for a given bit has settled, another 47ns is needed to compute the result for that bit - path a in Figure 9.6. This result is then driven onto the ALU output node 17ns later - path b in Figure 9.6. Table 9.3 gives a breakdown of the circuit components for the worst case situation of the slowest bit - the second most significant bit.

Circuit Component	Delay	Figure
Nibble with carry bypass	14ns	9.5, path a
Nibble without carry bypass	43ns	9.5, path b
Slowest carry generation within a nibble	32ns	9.5, path c
Compute result from carry	47ns	9.6, path a
Drive ALU output load	17ns	9.6, path b
Drive split datapath line	35ns	

Table 9.2- Speed Estimates of ALU Circuit Components (CRYSTAL)

Operation Component	Delay	% Total Delay
Generate slowest carry inside first nibble	32ns	13.0
Propagate across 4 nibbles	42ns	17.0
Drive split datapath	35ns	14.1
Propagate across 3 nibbles	42ns	17.0
Generate slowest carry inside last nibble	32ns	13.0
Compute result from carry	47ns	19.0
Drive ALU output load	17ns	6.9
<b>Total</b>	<b>247ns</b>	<b>100</b>

**Table 9.3- Speed Estimates of ALU Operation Components (CRYSTAL)**

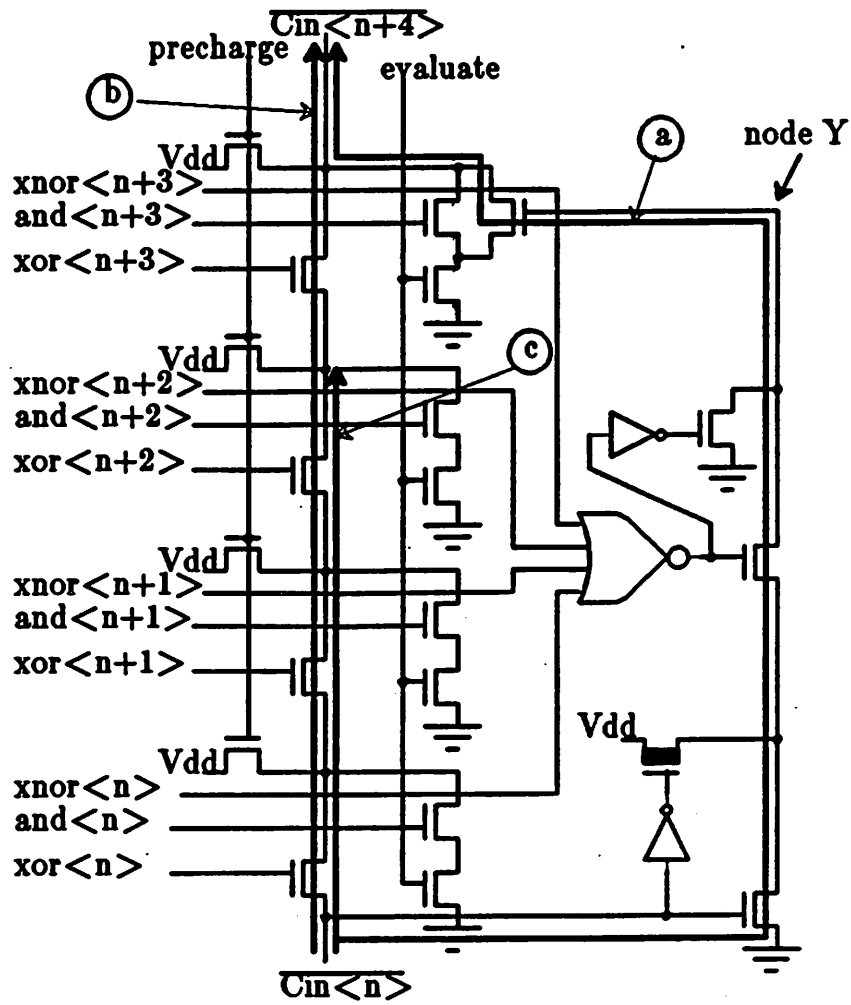


Figure 9.5- SOAR ALU- Carry Paths

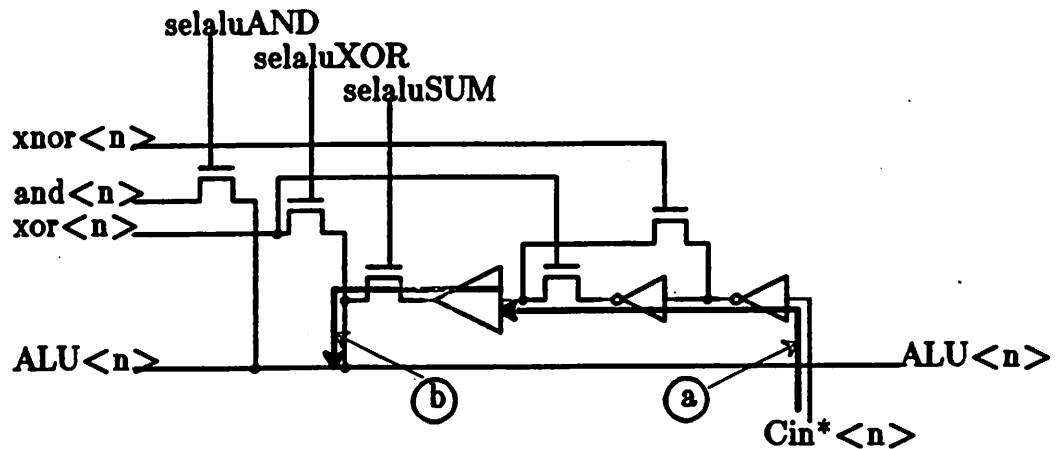


Figure 9.6- SOAR ALU- Sum Generation

When the original 100ns estimate was made, there was no layout and therefore the parasitic capacitances were not known. The original estimate also only accounted for carry propagation across eight nibbles. Carry propagation across eight nibbles including parasitic capacitances, is 112 ns. As can be seen from the difference between total ALU delay and delay for eight nibbles, overhead delays contribute significantly to the total delay – 54%. Overhead delays are due to the carry line that crosses the split datapath, generation of the slowest carry in the first and last nibbles, computation of the result from this carry, and the load of the output bus.

If 247ns for ALU operation had been considered unacceptable, design would have returned to the circuit synthesis step. The ALU could have been redesigned to increase its speed. One way to do this using existing circuitry, is to add a carry select scheme to the carry computation. To implement a carry select scheme, an ALU is divided into carry select blocks. The size of these blocks does not have to be the same as the block size for the carry bypass or carry lookahead. The carry output of each carry select block is computed for two situations. It is computed assuming the carry input to the select block, is 'one' and also 'zero' (Figure 9.7).

This is done for all carry select blocks simultaneously. Then the correct carry output of a given select block is chosen by the carry output of the next lower order block.

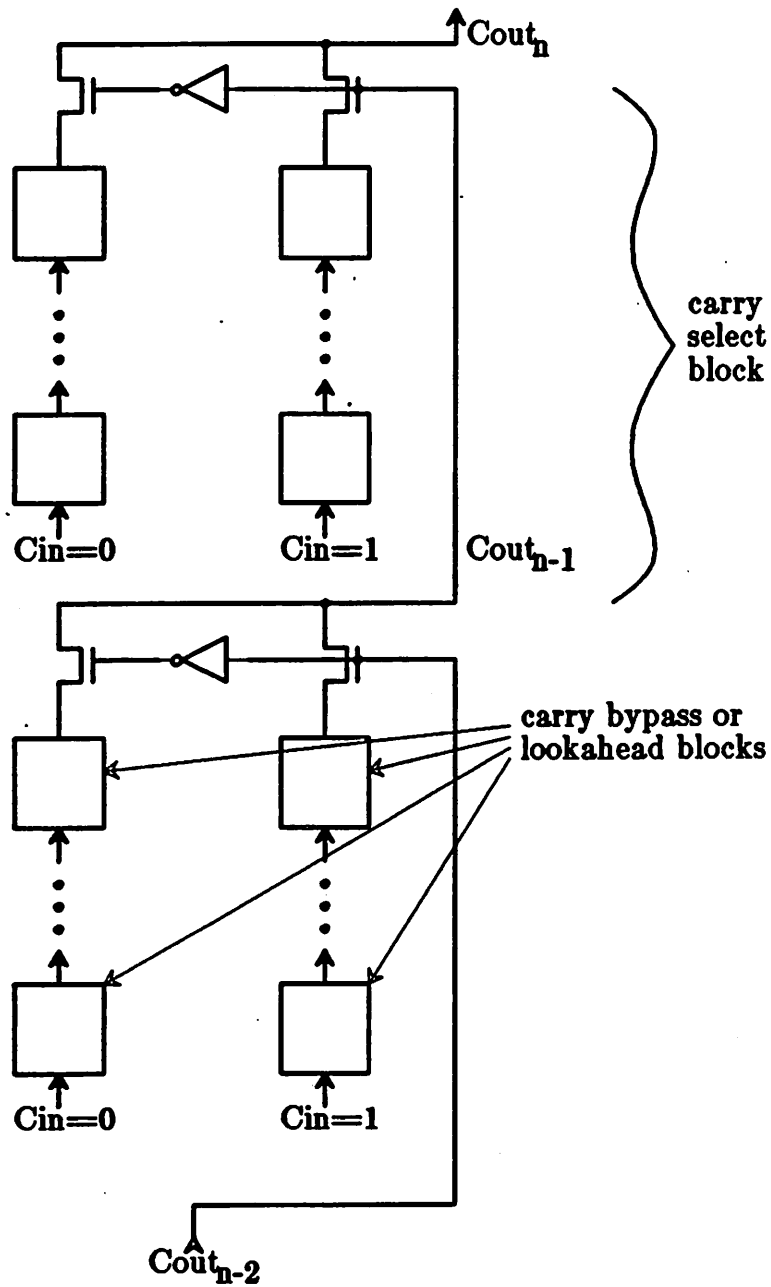


Figure 9.7- Carry Select Scheme

Maximum speed for carry computation with carry select depends on:

1.  $n$  = Total number of bits
2.  $m$  = Number of bits per carry bypass or lookahead block
3.  $p$  = Number of carry select blocks
4.  $a$  = Delay across a carry bypass or lookahead block
5.  $i$  = Maximum delay inside a carry bypass or lookahead block
6.  $s$  = Carry select delay as one select block selects another

Using these parameters:

$$n/mp = \text{number of bypass or lookahead blocks per select block}$$

The worst case computation time for one carry select block is due to the worst case computation for the lowest order bypass or lookahead block, followed by carry propagation across the remaining bypass or lookahead blocks.

$$i + (n/mp - 1)a = \text{carry computation time for one select block.}$$

All select blocks perform this computation for the cases of  $\text{carry}_{in} = 1$  and  $\text{carry}_{in} = 0$ , simultaneously at the start of the carry calculation. To complete the calculation, each lower order select block, starting from the lowest, chooses the carry output of the next higher order select block, ending with the highest order block. Thus, the total delay is:

$$i + (n/mp - 1)a + (p - 1)s = \text{total computation time}$$

For SOAR:

$$n = 32$$

$$m = 4$$

$$i = 32\text{ns}$$

$$a = 14\text{ns}$$

$$s = \text{unknown, assume } 10\text{ns}$$

$$p = \text{to be determined so that the overall delay is minimum}$$

With four bits per carry bypass block and 32 total bits, possible values for  $p$ , the number of carry select blocks, are 2, 4, and 8. Using the expression for total carry computation time, the delay for each of these possibilities is calculated (Table 9.4). The optimum scheme for SOAR would have been to have had four carry select blocks, each with two nibbles or 8 bits each. This would have reduced the ALU delay by 72ns, to 175ns - a 29% improvement.

Carry Scheme	Carry Computation Time
$p=2$ , carry select	84ns
$p=4$ , carry select	76ns
$p=8$ , carry select	102ns
no carry select	130ns

Table 9.4- Carry Delay for Carry Select Schemes

## 2.2. Inserter/Extractor

The inserter/extractor is assigned to the same time slot as the ALU. With a simulated speed of 157ns, it is slower than the desired speed. However, the desired speed was based on the predicted ALU speed. The inserter/extractor, at 157ns, is faster than the newly estimated ALU speed – 247ns. Therefore, it did not need modifications.

## 2.3. SWP Comparator

The second major problem was the SWP comparator. Stronger gates were substituted wherever possible in the layout. The comparator is loaded from the MAL and SWP in 24ns. It includes a 24 bit ripple carry adder. Each pair of bits has a delay of 36ns (Table 9.5). After the sum has been calculated, 106ns are needed for the output signal to settle, bringing the total delay to 601ns. Any further improvement would have required a carry scheme other than the ripple carry.

Circuit Component	Delay	% of Total Delay
Load comparator	24ns	4.0
12 x delay/2 bits	12x36ns= 432ns	71.8
Drive split datapath	39ns	6.5
Compare signal	106ns	17.7
Total	601ns	100

Table 9.5- Speed Estimates of SWP Comparator Components (CRYSTAL)



## 2.4. Register File Decoders

The slowest path through the decoders was 438ns. This was after circuit modification. Buffers and larger gates were added as necessary to drive large loads. One contribution to the decoder delay comes from the ability to access a register using its memory mapped address from the Destlatch (Figure 9.8). The basic decoders are shown in Figure 6.21 and within the dashed lines of Figure 9.8, and discussed in Chapter 6, Section 1.2.3. A mux selects either the register specifier field or the computed memory address as the decoder input. This muxing involves a significant amount of chip area devoted to routing. The capacitive loads of this routing contribute to the delays. The delay through this mux and its routing are 108ns (Figure 9.9). Four of the mux outputs must go through the NOR section of the decoders, another mux, and associated routing. The NOR section, including buffers, contributes 72ns; the second mux and associated routing contributes another 204ns. The output of the NOR section then goes through the NAND decoders that have a delay of 54ns. As can be seen from the breakdown of the decoder delay, mux and routing delays contribute significantly to the overall speed. 71% of the delay was due to the muxes and routing (Table 9.6).

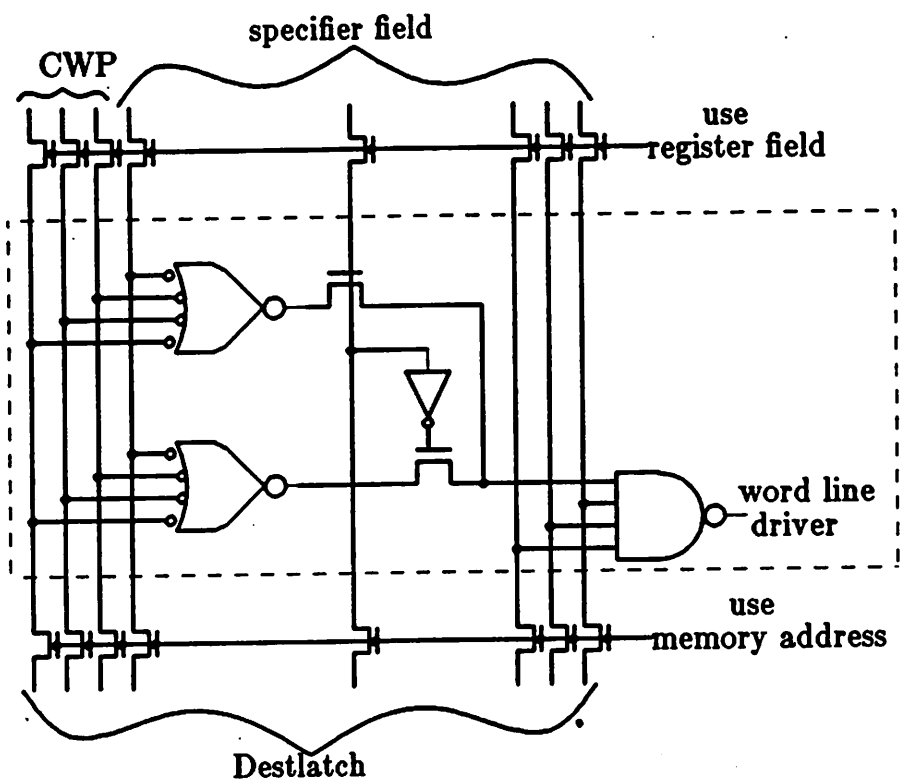


Figure 9.8- Register File Decoders

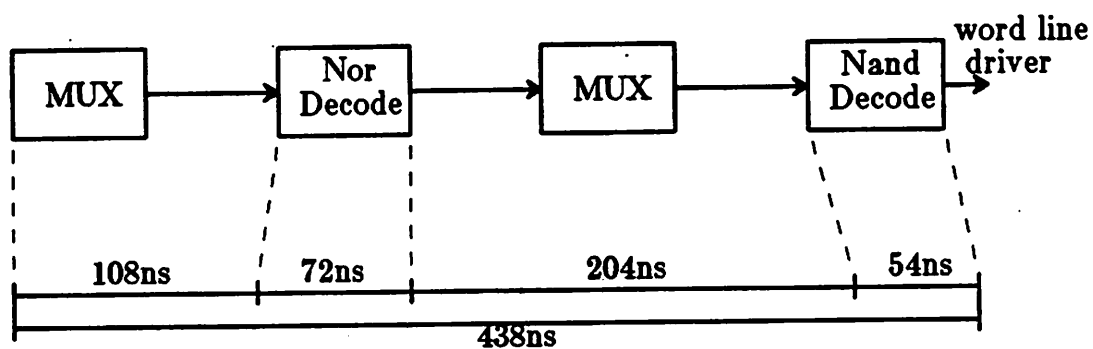


Figure 9.9- Register File Decode Block Structure

Circuit Component	Delay	% of Total Delay
Memory mapping mux and routing	108ns	24.7
NOR decode	72ns	16.4
Window mux and routing	204ns	46.6
NAND decode	54ns	12.3
Total	438ns	100

Table 9.6- Speed Estimates of Decoder Components (CRYSTAL)

## 2.5. Sign Extender

The sign extender was estimated to have a 305ns delay after larger buffers were added to drive large routing loads. Instruction decode and sign extension both occur in phase 1. The bits to be sign extended depend on whether or not the instruction is a store instruction (Figure 6.11). A mux selects these bits. The instruction is first decoded to determine the mux select line. Table 9.7 gives the breakdown for the sign extension delay.

Circuit Component	Delay	% of Total Delay
Interconnect from instr. latch to decode PLA	115ns	37.7
Decode PLA	63ns	20.7
Interconnect to datapath	92ns	30.2
Control line driver	10ns	3.3
Sign extend	25ns	8.2
Total	305ns	100

Table 9.7- Speed Estimates for Sign Extension Components

Only 25ns (8.2%) of the delay is due to actual sign extension. The rest of the delay is due to the two types of sign extension and decode that they require. 68% of the total time is due to routing delays.

## 2.6. Summary

This analysis step revealed three major contributors to speed estimates that were difficult to estimate in previous steps:

1. Parasitics within circuits
2. Interconnect parasitics
3. Overhead circuitry

Parasitics exist within circuits due to large diffusion areas and high resistance signal lines. Large diffusion areas contribute significantly to the loads of any gates whose outputs must drive the large diffusion. High resistance signal lines were usually less significant on SOAR. They are usually in series with a transistor of the gate driving the signal line and therefore the total resistance is the sum of the parasitic and the 'on' resistance of the transistor. Thus, the parasitic must be

compared with the transistor resistance to determine its significance. Routing loads within circuit blocks can be significant whenever a circuit block covers a large area. This was the case for the decoders. Routing capacitances are due to interconnect areas and the resulting capacitances to the substrate. Interconnect resistances are the result of polysilicon crossovers. Parasitic resistances of interconnects were frequently significant on SOAR – the crossover resistance value was comparable to the ‘on’ resistance of transistors in the interconnect driver. Until a layout exists, it is difficult to know the values of parasitics. Necessary overhead circuitry is known before the circuit and interconnect steps. However, it proved to be easily overlooked in circuit blocks that were composed of a repeated series of one cell type – the adders. Overhead circuitry contributed significantly to delays in these circuits.

### 3. Functional Block Analysis

After the final *circuit and interconnect analysis* step, design moves to the *functional block analysis* step (Figure 9.1). Theoretically, functional block analysis should be able to be done by considering the block structure of each signal just as it was done before circuit and interconnect synthesis (Figure 9.10). Now that circuit and interconnect synthesis has been done, accurate estimates of the internal block delays and interblock delays should be available from the circuit and interconnect analysis step. However, on the SOAR project individual interconnect delays were not computed. Therefore, functional block analysis incorporated circuit and interconnect analysis also (Figure 9.11). The timing verifier, CRYSTAL, was used to estimate the critical paths from the layout.

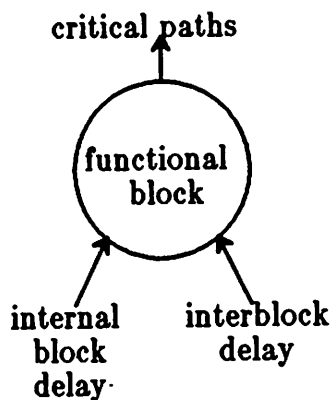


Figure 9.10- Functional Block Analysis Only (Speed)

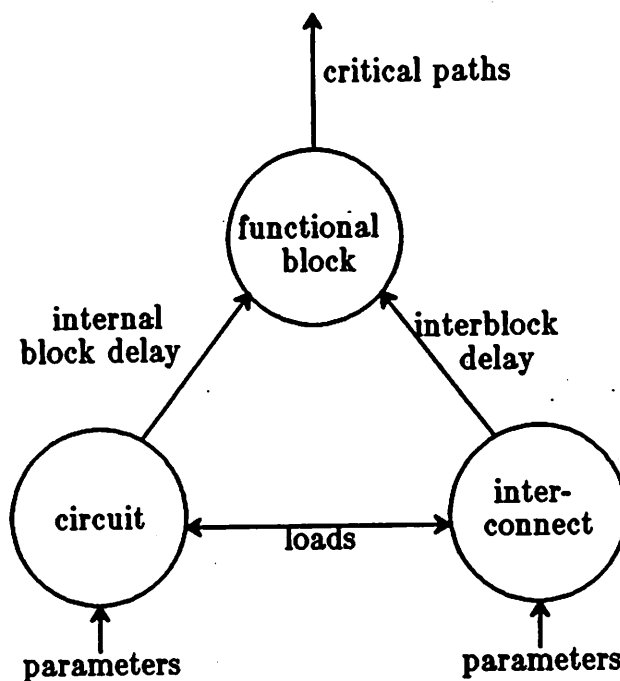


Figure 9.11- Functional Block Analysis with Circuit and Interconnect Levels

Delay analysis was done for all signals that were analyzed during the previous functional block analysis step (Ch.8, Sec.2.1). First, signals that were allotted a single phase and the following underlap, were analyzed. Signals that were the bottlenecks of the individual phase lengths were identified. Following the methodology, design returned to the functional block synthesis step (Figure

9.1). This option to redesign the circuit block structure was considered but no obvious improvements were found. Thus, design proceeded to the circuit and interconnect step once again. Stronger buffers were substituted at the PLA outputs for the limiting signals that included PLAs in their paths (Figure 9.12). Larger drivers and gates were put into datapath blocks that had to drive buses on limiting signal paths in the datapath. This included the block that drove the ALU output bus and blocks in the signal path that transferred data from busD onto either busA or busB. The transfer of data from busD to busA and busB was first identified as a potential bottleneck during functional block design (Ch.8, Sec.2.1.1). Analysis of the extracted layout confirmed this bottleneck.

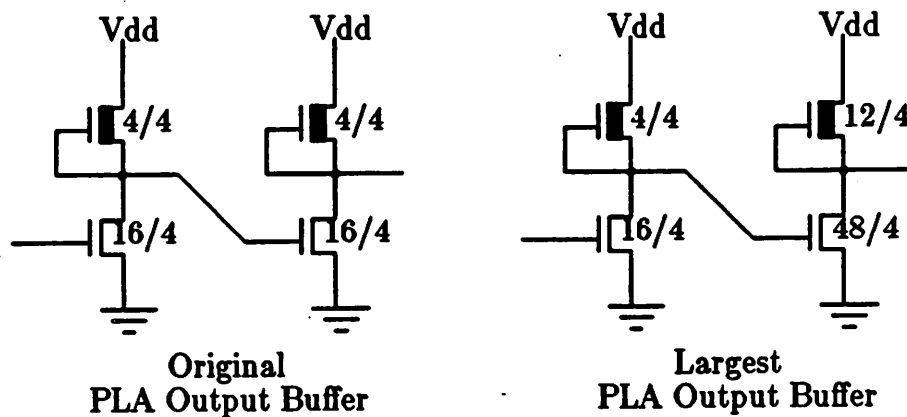


Figure 9.12- PLA Output Buffer Optimization

The first instruction latch drove routing, five PLA inputs, and two latch inputs. This was a considerable load and these signals were consequently slow. To correct this, the outputs of this instruction latch were split into two branches, each with its own buffer (Figure 9.13). One branch drove one PLA and the two latches. The other branch drove the remaining four PLAs. Table 9.8a gives the delay breakdown for the slowest signal paths with this scheme. These include the first instruction latch and the slowest PLA - cpla1. If the PLAs had not been

split during functional block synthesis, the instruction latch load would have been less and the delay of the interconnect between the instruction latch and decode PLA would have been smaller. The breakdown for this option is shown in Table 9.8b. The smallest possible load - short routing to one PLA - provides the lower bound here. This data is from the second instruction latch which is in close proximity to the one PLA it drives. In reality, the first instruction latch would have driven three PLAs and two latches, if the PLAs had not been split. Thus, the interconnect speed would have fallen somewhere between this lower bound and the present speed. With the split PLA scheme, the interconnect speed is slower but the PLA speed more than makes up for the difference. The net result is that without the split PLA scheme instruction decode would have been 18-30% slower.



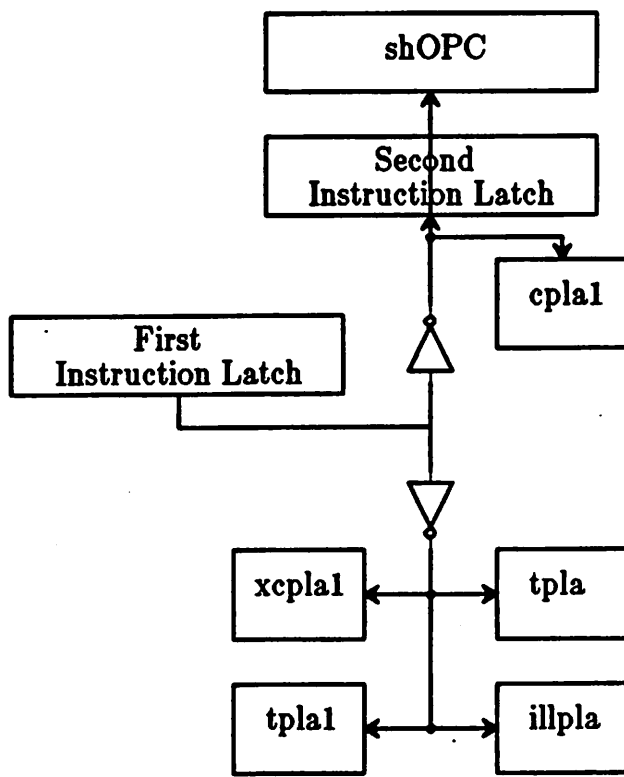


Figure 9.13- Instruction Latch Output Buffering

Component	Delay
Load instruction latch	36ns
Drive buffer from the instruction latch	98ns
Drive PLA input from the buffer	12ns
Cpla1 (split)	102ns
<b>Total</b>	<b>248ns</b>

Table 9.8a- Speed Estimate for Decode with Split PLAs (CRYSTAL)

Component	Delay
Load instruction latch	36ns
Drive PLA input from the instruction latch	56-110ns
Cpla1 (unsplit)	210ns
Total	302-356ns

Table 9.8b- Speed Estimate for Decode with Unsplit PLAs (CRYSTAL)

This scheme of buffering each branch of signal lines that split as they go to their various destinations, is used on other slow signals also. Nilling signals, CWP updating signals, and the critical signals between cpla2 and apla1 are other places where these repeaters are incorporated.

#### 4. Microarchitecture vs. Functional Block Analysis

The previous section discussed functional block analysis that led to revisions in the circuit and interconnect designs. These revisions were followed by a second *functional block analysis* step. Design then moved to the *microarchitecture vs. functional block analysis* step (Figure 9.1). In this step the results of functional block analysis are compared to the microarchitecture requirements.

##### 4.1. Phase 1 Analysis

Signals that were allotted phase 1 and the following underlap to settle were the sign extended immediate and control lines for:

1. Shadowing
2. Forwarding
3. Special register decode
4. ALU input latch loading
5. Reads to buses A, B, and L

Figure 9.14 is a histogram of the final settling times for these signals. These are the settling times after stronger buffers were substituted to drive large interconnect loads. The ranges of settling times for the various categories of signals are shown below the histogram. The paths for phase 1 are well balanced except for the forwarding signals and the one shadow signal. Transistor ratios of the gates at the destinations of these slower signals were designed so that the gate outputs switch at lower voltages. The effect of this is that the destination gates will switch on even though the input signal is farther from its final value than for gates with standard transistor ratios.

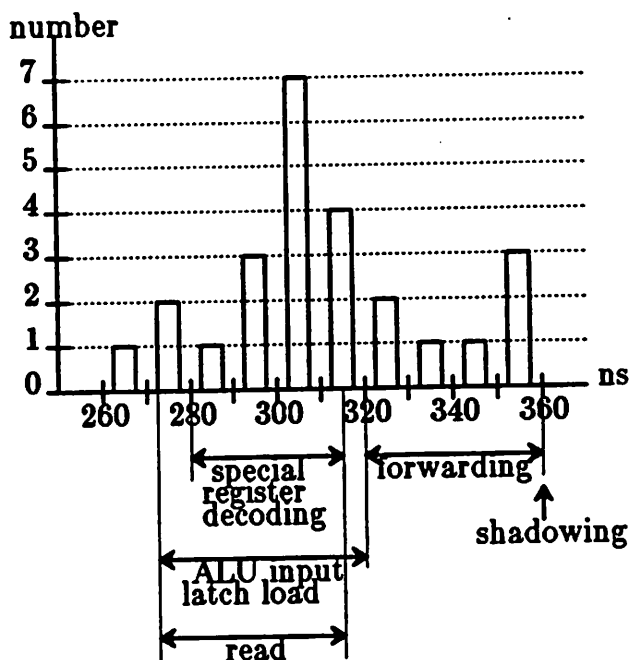


Figure 9.14- Phase 1 Signal Settling Times

Except for sign extension, all signals settling in phase 1 are the result of decoding using PLAs. Significant fractions of these settling times were due to routing delays; an average of 51%, with a standard deviation of 6%, of the delay times were due to routing. The range for routing delays was 34% to 61%. Knowing the settling times for signals allotted to phase 1 and the following underlap, a more accurate estimate of the length of phase 1 may be made. Phase 1 plus the subsequent underlap require 320ns.

#### 4.2. Phase 2 Analysis

During phase 2 operands are read onto buses A, B, D, L, and S, and latched at the ALU input. Bus values are latched at the end of phase 2 and therefore only the phase 2 high time is allotted to the settling of these buses (Ch.7, Sec.2.1). Phase 2 is limited by signals that are first read onto one bus and then transferred to either busA or busB. BusD must be transferred onto busA when the TB, SWP,

or PC is used as an operand. Forwarding requires the ALU output latch (Destlatch) data to first be read onto busD and then transferred to either busA or busB. This takes 202ns (Table 9.9). 182ns is required for busA to settle when data from busS is transferred to busA. If these buses could be directly loaded into the ALU input latches, the longest settling time for any bus loading the input latches, would be 163ns. This occurs when busA or busB is driven by the data input latch (LOADL). Register file data is driven onto the buses in 154ns. With the present scheme, 202ns is required for phase 2.

Signal	Origin	Delay
BusA,B	busD	202ns
BusA	busS (CWP, PSW)	182ns
BusA,B	data latch- LOADL	163ns
BusA,B	register file	154ns
BusA	SHA, SHB	141ns
BusD	ALU output latch, TB, SWP, PC	<124ns
BusL	sign extender, instruction latch	<124ns

Table 9.9- Speed Estimates for Read (CRYSTAL)

Before phase 3 begins, logic in the ALU must settle. This logic is used by the carry chain. The carry chain is precharged before phase 3 and then evaluated during phase 3. Any unsettled carry chain inputs at the start of phase 3 can cause false discharging of the carry chain. There is no way to recover from this which leads to incorrect results. In Table 9.10 these are the XNOR, XOR, and AND signals.

Signal	Delay
XNOR	335ns
XOR	329ns
AND	288ns
tbusA<28:31>	318ns
tbusB<28:31>	318ns
tagcompare	325ns

**Table 9.10- Speed Estimates for Phase 2 + Underlap (CRYSTAL)**

Operand tags – tbusA and tbusB – are also latched at the start of phase 3 for trap detection. Tagcompare is formed by the tagcompla from the operand tags. These signals – tbusA, tbusB, and tagcompare – depend on the operands and therefore start to settle at the beginning of phase 2. They must finish settling by the start of phase 3. Thus, phase 2 and the following underlap must total to 335ns.

During the underlap between phase 2 and phase 3 the register file word lines must be disabled. This was estimated to take 125ns.

### **4.3. Phase 3 Analysis**

During phase 3, buses A, B, and D are driven with data that is being written. Delays for this are similar to bus delays for the read operation - 200ns (Table 9.11). Addition is also performed in the ALU during phase 3. The ALU result must then be driven onto the EAbus, to be loaded into the PC or MAL, before the end of phase 3. This requires 255ns, of which 247ns is the ALU delay. Thus, phase 3 must be 255ns long.

Signal	Delay	Pipeline Function
BusA,B	200ns	Write
BusD	124ns	Write
EAbus	255ns	Alu

Table 9.11- Speed Estimates for Phase 3

Again the word lines must be disabled between phase 3 and phase 1. This requires a 125ns underlap between these two phases.

#### 4.4. TRAP Analysis

The previous three sections have discussed single phase length and underlap requirements. However, this does not cover all signals. Several signals that span more than one phase exist. Trap mechanism signals fall into this category. SOAR had ten different types of traps (Ch.5, Sec.1.6). Trap detection occurs through a variety of paths as shown in Figure 8.8. Once the trap is detected, tpla2 encodes its priority and the Trap circuit block generates a signal that loads the internal TRAP opcode into the first instruction latch (CPIPE1). Figure 9.15 shows a more detailed circuit block structure for each type of trap detection. Outputs of these block structures are used by tpla2 and the Trap circuit block. The clock phase that initiates trap detection varies according to the type of trap. Illegal opcode traps, window traps, and software interrupt traps are initiated when the first instruction latch changes – at the beginning of phase 1. The start of phase 2 triggers page fault interrupts, IO interrupts, tag traps, and generation scavenging traps. Trap instructions and overflow tag traps use the ALU results and therefore are initiated on phase 3. Settling times for the various trap

detection signals also vary. Consequently, the completion time of trap detection depends on the type of trap.

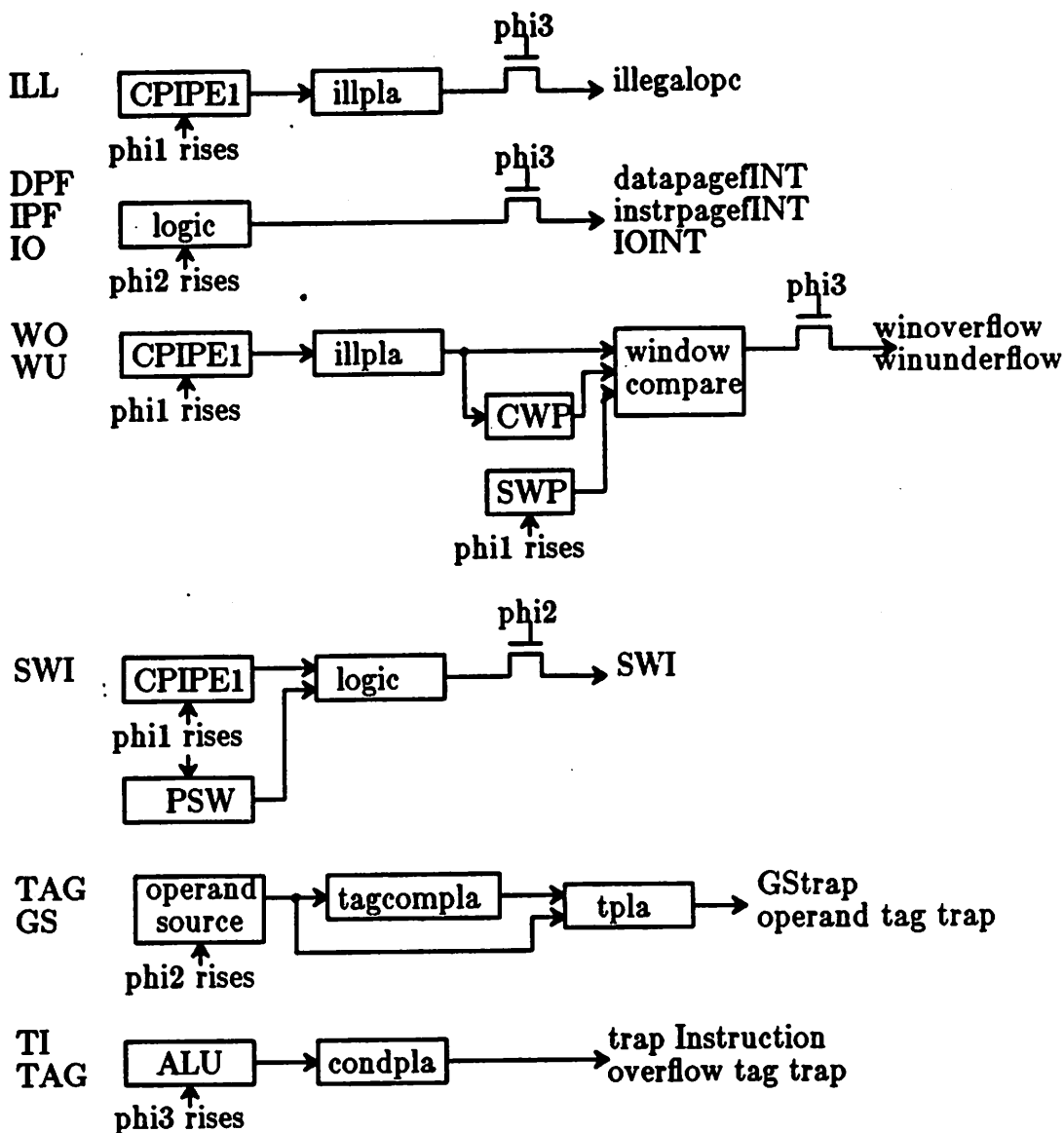


Figure 9.15- Trap Block Structures

The Trap circuit block samples the trap detection signals on phase 1 and subsequently causes the TRAP internal opcode to be loaded into the first instruction latch when a trap is detected. This opcode is then decoded to set control lines. Therefore, the TRAP internal opcode should be loaded as early in



phase 1 as possible. Figure 9.16 shows the timing of the various types of traps. Clock phase lengths are those that were determined by single cycle analysis. Most types of traps are detected by early in phase 3 and therefore are not limiting factors. The exceptions to this are trap instructions and overflow tag traps. These both depend on the ALU output which must then be interpreted by the condition PLA (condpla). Consequently, the signal (CPIPEtrap) from the Trap circuit block that loads the first instruction latch settles 154ns after the start of phase 1. This is 125 ns after the instruction latch would have been loaded if there had not been a trap. Thus, control lines settle 125ns later when a trap is detected. This presents no problem for control lines that are used in phase 2 because these control lines determine what is to be read and operated on, and do not change the state of SOAR. However, phase 3 control lines are used for writing and loading of state registers. Therefore, this 125ns delay must be considered when analyzing control lines used during phase 3.

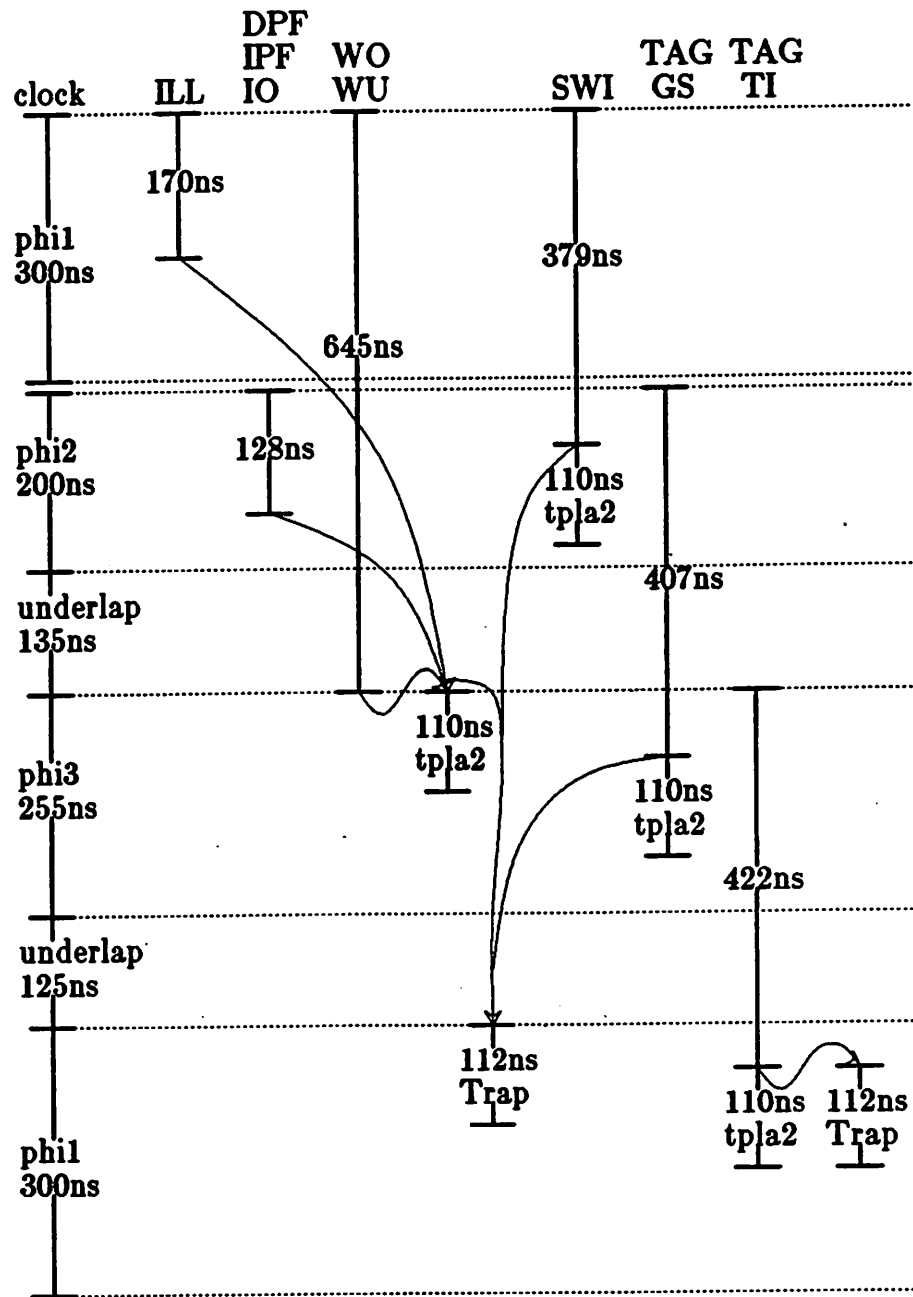


Figure 9.16- Trap Timing

#### 4.5. Decode for Phase 3 Analysis

Phase 3 is the phase during which ALU operations are completed, results are written, and latches are loaded with future values. Control lines for phase 3 direct:

1. ALU operation
2. Writes
3. Addressing latch loads
4. Internal opcode loads
5. Other state latch loading

Figure 9.17 is a histogram of the delays for these signals. Settling times range from 226ns to 432ns when a trap has not occurred during the previous cycle, and 351ns to 557ns when a trap has occurred. These signals are allotted phase 1, the phase1/2 underlap, phase 2, the phase2/3 underlap for settling. According to the restrictions on single cycle lengths, this allotted time is 655ns. Therefore, the requirements of this decode are not a limiting factor for phase lengths. The slowest of these signals will still be ready 100ns before it is needed.

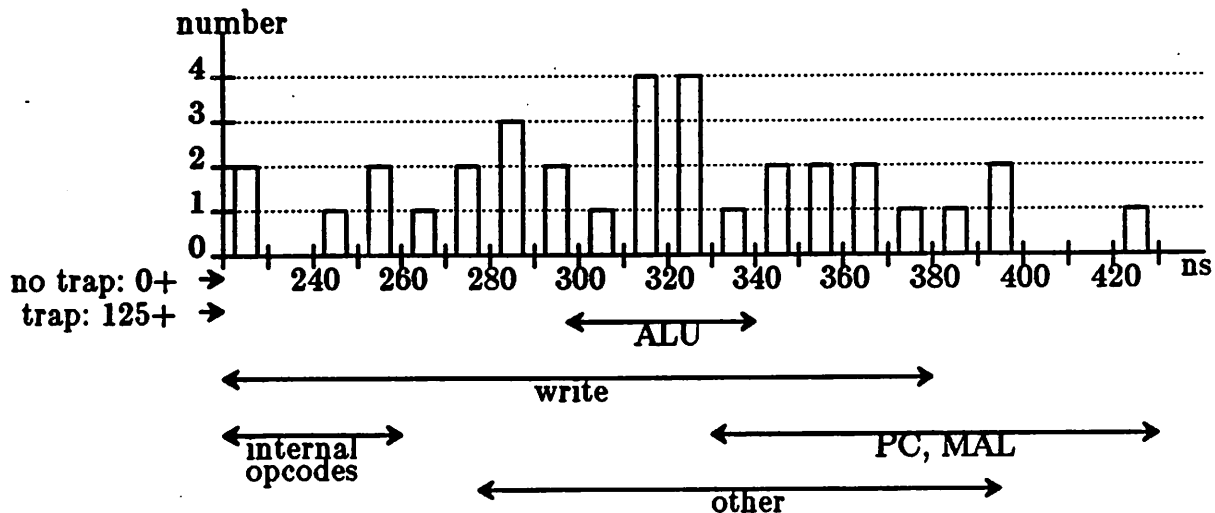


Figure 9.17- Decode Settling Times for Phase 3

As seen in the histogram the delay ranges for the different types of phase 3 control lines overlap significantly. The longest delays are for signals that control the loading of the addressing latches - PC, MAL. The fastest signals are those

that load internal opcodes into the first instruction latch.

#### **4.6. Microarchitecture Analysis**

Functional block analysis provided speed estimates of all signal paths. These estimates led to minimum lengths for individual clock phases and groups of clock phases (Table 9.12a,b). The requirements of the signals that spanned multiple phases were consistent with the individual phase requirements for all signals except register file decoding for reads and the SWP comparison (Table 9.12b).

Previous circuit analysis of the register file decoders showed that they needed 440ns to complete a decode (Sec.2.4). With the register specifier available at the start of phase 1 and a read occurring during phase 2, only phase 1 and the following underlap are available for decoding of the operand specifiers. Instruction decode requires this period to be 320ns. Thus, register decode for reads would have extended phase 1 by 120ns, to 440ns.

The SWP comparator was also simulated during circuit analysis. Its speed was estimated to be 600ns (Sec.2.3). An extra 150ns is needed to route its result across the chip to where it is used. This brings the total time for the comparison to 750ns, which is 95ns longer than the time available according to the single phase requirements. Extending phase 1 by 120ns for the register decode would have given SWP comparison 775ns to complete, which would have satisfied its requirements.

Phases	Required Length	Reason
$\Phi 1 + ul1$	320ns	instruction decode (reads)
$\Phi 2$	200ns	read
$\Phi 2 + ul2$	335ns	ALU setup
ul2	125ns	disable word lines
$\Phi 3$	255ns	ALU
$\Phi 3$	200ns	write
ul3	125ns	disable word lines
Cycle	1035ns	Total

Table 9.12a- Individual Phase Length Requirements

Phases	Required Length	Length from Single Phase Requirements	Reason
$\Phi 1 + ul1 + \Phi 2 + ul2$	555ns	655ns	instruction decode
$\Phi 1 + ul1$	440ns	320ns	register decode (reads)
$\Phi 1 + ul1 + \Phi 2 + ul2$	440ns	655ns	register decode (writes)
$\Phi 1 + ul1 + \Phi 2 + ul2$	750ns	655ns	SWP comparison

Table 9.12b- Multiple Phase Length Requirements

Phases	Required Length	Length from Single Phase Requirements	Reason
$\Phi 3 + \text{ul}3 + \Phi 1 + \text{ul}1$	585ns	700ns	reg. dec.-read
$\Phi 3 + \text{ul}3 + \Phi 1 + \text{ul}1 + \Phi 2 + \text{ul}2$	990ns	1035ns	SWP compare

Table 9.12c- Modifications to Multiple Phase Requirements

To satisfy these two signals the cycle time would have been extended by 12%, from 1035ns to 1155ns. However, the microarchitecture analysis step provides the option of returning to microarchitecture design if the analysis results are unsatisfactory. This option was explored and with a few minor microarchitecture modifications it was possible to fit these two signals into the single phase requirements.

Originally register decode started on phase 1. However, the register specifiers become available during the previous phase 3. So the start of the register decoding for reads was moved to phase 3 (Table 9.12c). An extra 145ns is now needed for read decoding to allow address calculation to settle for the case of memory mapped register reads. However, even with this longer decode, the read decode now finishes 115ns before it is needed.

Similarly, the SWP comparison also began at the start of phase 1. The data used for this comparison was loaded into the slave sections of the MAL and SWP at the start of phase 1. However, it is loaded into the master sections during the previous phase 3. The comparator was modified to compare the data from the master sections. In this way the comparison also starts one phase earlier, in phase 3 (Table 9.12c). Address calculation occurs during this phase 3 and provides the MAL data that is used in the comparison. This causes the start of the comparison

to be delayed by 240ns to allow portions of the address calculation to complete. This brings the total comparison delay to 990ns which is less than the 1035ns provided by the single phase requirements.

Another possible microarchitecture modification is to eliminate the transfer of data from busD and busS, onto the ALU input buses during the read operation. As previously discussed, this would reduce phase 2 by 20% from 200 to 160ns. The overall impact on the 1035ns cycle time would have been a 4% improvement. This change would have been more complicated and the benefit did not seem great enough for it to be worth implementing.

Another microarchitecture modification concerns the loading of the shadow latches - SHA, SHB, shDST, and shOPC. The shadow latches capture the operands, opcode, and destination field of an instruction that causes a trap. They are always loaded during the read phase - phase 2 - except for the cycles after a trap has occurred (Figures 9.18 and 9.19). In this way the information from the instruction causing the trap, is saved. This presents two problems. The control line that loads these latches must be ready by phase 2. Simulation showed that this was one of the slowest signals - 360ns. On SOAR transistor ratios were adjusted so that this would not be a limit, but a much cleaner solution exists. The second problem arises because these registers may be used as operands. When used as operands, they will be read as they are being loaded on phase 2. They are read onto precharged buses. A race condition exists between the load enable line and the read enable line. Thus, when shadowing is enabled (PSW<1>), reading them leads to unknown operands. This may not affect much of the software but does add inconsistency to the architecture.

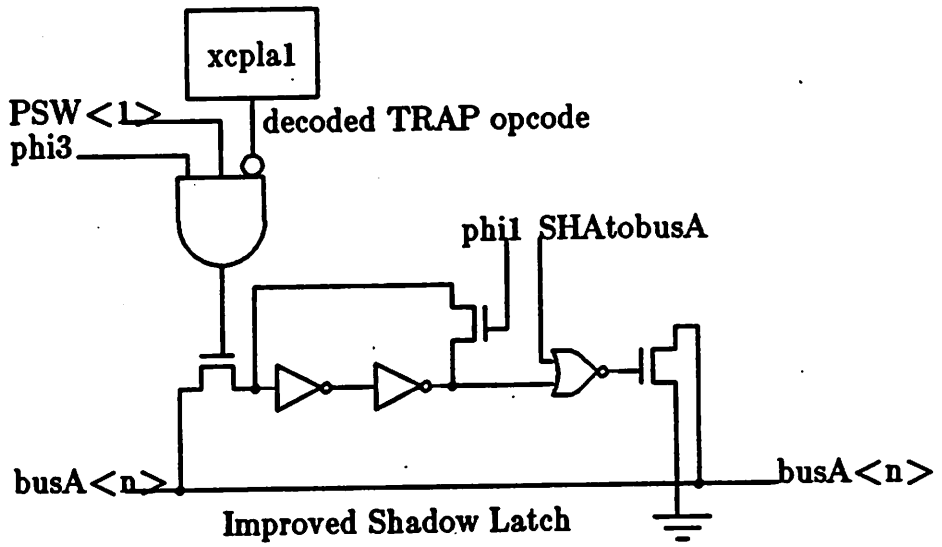
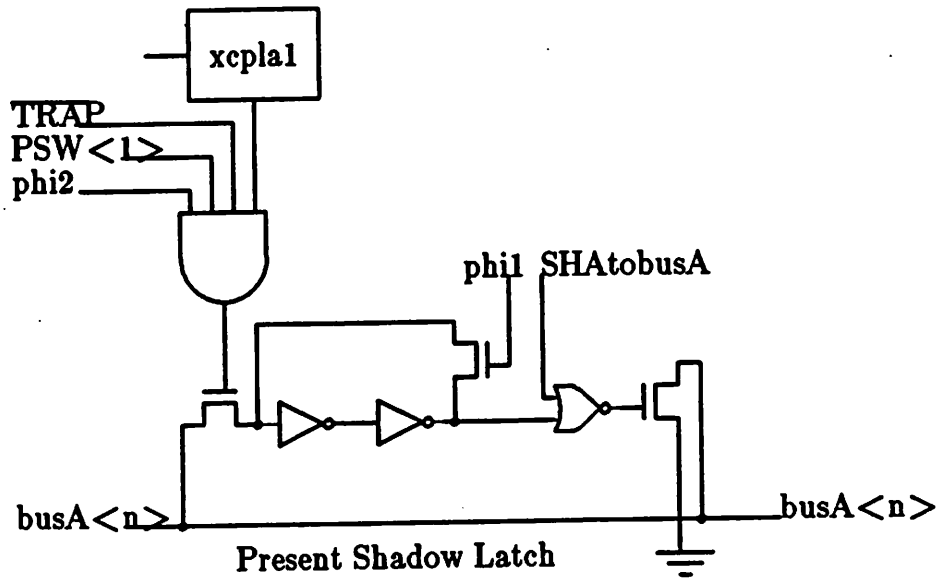


Figure 9.18- Shadow Latches



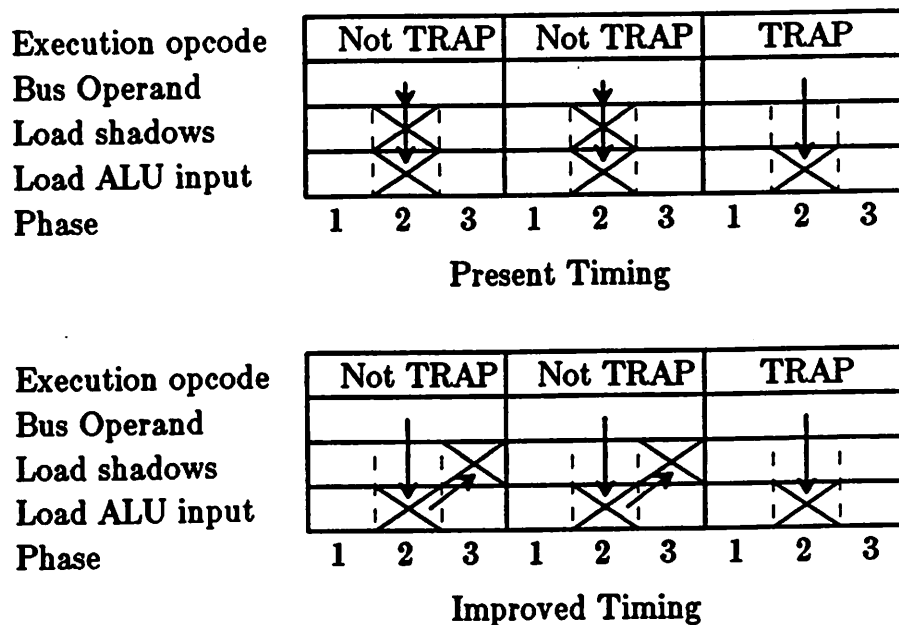


Figure 9.19- Shadow Latch Timing

A better solution is to load them on phase 3 from the ALU input latches or instruction latch (Figures 9.18 and 9.19). By postponing the load to phase 3, the slow control signal has ample time to settle. The race condition between loading and reading the shadows is also eliminated, since loads and reads no longer occur simultaneously.

Once these microarchitecture changes were decided upon, the functional block and circuit block descriptions were modified. The modifications were verified using SLANG. Design and layout at the circuit and interconnect levels was then updated and verified. After these modifications, the limits on the SOAR cycle time are due to read, ALU setup, ALU operation, instruction decoding, and word line disabling (Tables 9.12a,b,c). Extra time is available for register decoding, the SWP comparison and writes. Table 9.13 lists the final clocking according to CRYSTAL speed estimates.

Phase	Length
Phase 1	300ns
Phase1/2 underlap	20ns
Phase 2	200ns
Phase2/3 underlap	135ns
Phase 3	255ns
Phase3/1 underlap	125ns
Total	1035ns

Table 9.13- Phase Lengths, Realized SOAR

### 5. Optimized Pipeline Analysis

SOAR was designed and sent to fab before this methodology was solidified. Further design of SOAR using this methodology revealed an optimized SOAR pipeline (Ch.6, Sec.1.5). Using the more detailed speed estimates that became available after circuit and interconnect design, the optimized SOAR speed may be estimated. Table 9.14 shows the requirements for the lengths of the individual phases and combinations of phases. Using these requirements the lengths of the individual phases may be determined (Table 9.15).

Phases	Required Length	Single Phase Requirements	Reason
$\Phi 1w+ul1w+\Phi 1p+ul1p$	320ns	440ns	instruction decode
$\Phi 2$	200ns	200ns	read
$\Phi 2+ul2$	335ns	335ns	ALU setup
ul2	125ns	135ns	disable word lines
$\Phi 1w$	200ns	200ns	write
ul1w	125ns	125ns	disable word lines
$\Phi 1w+ul1w+\Phi 1p$	255ns	415ns	ALU
$\Phi 1p$	90ns	90ns	precharge
ul1p	20ns	25ns	skew
$\Phi 1p+ul1p+\Phi 2+ul2$	440ns	450ns	register decode-write
$\Phi 1w+ul1w+\Phi 1p+ul1p$	440ns	440ns	register decode-read
full cycle	750ns	775ns	SWP compare

Table 9:14- Phase Length Requirements, Optimized Pipeline

Phase	Length
$\Phi_{1w}$	200ns
$\Phi_{1w}/\Phi_{1p}$ underlap	125ns
$\Phi_{1p}$	90ns
$\Phi_{1p}/\Phi_{2}$ underlap	25ns
$\Phi_{2}$	200ns
$\Phi_{2}/\Phi_{1w}$ underlap	135ns
Total	775ns

Table 9.15- Phase Lengths, Optimized Pipeline

The optimized SOAR is limited by the register file operation, register file decode, and ALU setup. As previously discussed, the register file has no idle time. The time needed to cycle through the register file operations – precharge, read, word line disable, write, word line disable – becomes the processor cycle time. Time requirements and clock phase assignments for register file decode, ALU setup, and the SWP comparison are compatible with the register file clocking, with little wasted time. Thus, the limiting paths of the chip would have been well balanced. Instruction decode and ALU operation do not limit the cycle time as they do in the realized SOAR. The total cycle time is estimated to be 775ns, which is 260ns faster than the realized SOAR. This would have been a 25% improvement over the realized SOAR.

## 6. Split Datapath Analysis

The SOAR datapath was split into two 16 bit halves to reduce the time needed to drive the word and control lines. SPICE simulations were done using circuit models from the extracted layout for the 16 bit and 32 bit datapath widths. Operations affected by this split are shown in Table 9.16. The unsplit datapath would have required an extra 25ns for the enabling of control and word lines, and disabling of word lines. However, the ALU would not have had a carry line to drive across the chip and therefore would have been 35ns faster. With the realized SOAR pipeline, the phase2/3 underlap would have been limited by the longer word line discharge, not the ALU setup time as it is with the split datapath (Table 9.17a). Writes, not ALU operation, would have limited phase 3 on a SOAR with an unsplit datapath and the same pipeline. The cycle time would have been 1070ns – 3.4% longer.

Function	Split Datapath	Unsplit Datapath
Read	200ns	225ns
Disable word lines	125ns	150ns
Write	200ns	225ns
ALU	255ns	220ns

Table 9.16- Function Times

The affect of the split datapath on a processor with the optimized SOAR pipeline is greater (Table 9.17b). This is because the register file limits all phase lengths. The register file benefits most from the split datapath. The ALU, which is hurt by the split datapath, does not limit any phase lengths. SOAR, with the

optimized pipeline, but unsplit datapath would be 115ns or 14.8% slower than it would be with the split datapath. The unsplit datapath cycle time would be 890ns as opposed to 775ns for the version with the split datapath.

Phase	Limit	Unsplit vs. Split Datapath Time
$\Phi 1 + ul1$	decode	same
$\Phi 2$	read	+25ns
ul2	disable word lines	+15ns
$\Phi 3$	write	-30ns
ul3	disable word lines	+25ns
Cycle		+35ns = +3.4%

Table 9.17a- Unsplit Datapath Phase Limits, Realized SOAR Pipeline

Phase	Limit	Unsplit vs. Split Datapath Time
$\Phi 1w$	write	+25ns
ul1w	disable word lines	+25ns
$\Phi 1p$	precharge	+25ns
ul1p	skew	same
$\Phi 2$	read	+25ns
ul2	disable word lines	+15ns
Cycle		+115ns = +14.8%

Table 9.17b- Unsplit Datapath Phase Limits, Optimized SOAR Pipeline

## 7. References

[Kong84] Kong, S. I.; Private Communication, August 1984.

[Oust85] Ousterhout, J.; 'Using CRYSTAL for Timing Analysis', 1985 VLSI Tools: More Works by the Original Artists, T.R. UCB/CSD/85/225, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., Feb. 1985.

## **Chapter 10**

### **Results**

#### **SOAR Case Study**

This chapter discusses results of the SOAR processor in three separate areas:

1. Methodology
2. Processor
3. Architecture

The first section summarizes optimizations for SOAR that were revealed by this methodology. Speed and functionality were the primary concerns during the design of SOAR; power and area had looser restrictions. Thus, the optimizations discussed in this section are for speed improvements. Simulations were used for detailed delay analysis. Therefore, all delays in this section are simulated delays.

The second section discusses results from testing of the fabricated SOAR. The testing strategy and initial test setup are first described, followed by a report on the basic functionality. Again, the speed was the primary concern (after functionality) during testing. The speed restrictions for each phase are analyzed. Delays reported in this section are the measured delays of the fabricated processor. Differences between the measured and simulated delays are discussed in the section on process effects – Section 2.4.

The architecture of SOAR was designed for a high performance Smalltalk system. SOAR included several features to facilitate this. The impact of these architectural features on the implementation is discussed in the last section of this chapter. These features are individually analyzed for their contributions to the area, circuitry, complexity, and cycle time of SOAR.



## 1. Methodology Results

The SOAR design involves many optimizations for speed improvement. Some of these optimizations were implemented; others were not discovered until this methodology was developed. Unfortunately, that was after fabrication was underway. Table 10.1 lists the major optimizations. These optimizations and their impact on processor speed have already been discussed individually. Table 10.2 summarizes the effects of these speed improvements. All delay times in this section are the result of CRYSTAL simulations.

Optimization	Implemented	Discussion
Internal opcodes	yes	Ch.6 Sec.1.7, Ch.8 Sec.2.1.2
Pipeline	no	Ch.6 Sec.1.5, Ch.9 Sec.5
Balanced critical paths	yes	Ch.8 Sec.2.1.2, Ch.9
Split PLAs	yes	Ch.8 Sec.1.2, Ch.9 Sec.3
ALU- carry select	no	Ch.9 Sec.2.1
No bus to bus transfer	no	Ch.8 Sec.2.1.1
Split datapath	yes	Ch.6 Sec.2, Ch.9 Sec.6

Table 10.1- SOAR Optimizations

Optimization	Cycle Time Reduction	% Reduction
Internal opcodes	~1500ns	145
Pipeline	260ns	25
Balanced critical paths	120ns	12- realized
Split PLAs	54-108ns- realized	5-10
	0ns- optimized	0
ALU- carry select	54ns- realized	5
	0ns- optimized	0
No bus to bus transfers	40ns	4- realized, 5- optimized
Split datapath	35ns- realized	3.4
	115ns- optimized	14.8

Table 10.2- Simulated Cycle Time Reduction Due to Optimization

The most significant improvement was due to the use of internal opcodes. This cut the decode time from about 1800ns to 300ns - a factor of six improvement for the length of phase 1. SOAR would have had a cycle time of about 2.5 $\mu$ sec if internal opcodes had not been used - 145% slower.

The next major improvement comes from the optimized pipeline - 25%. Unfortunately, this was discovered after submittal for fabrication. If SOAR had been implemented with the optimized pipeline a faster ALU and the split PLAs would not have made a difference in the cycle time. This is because instruction decode and ALU computation do not limit any phase lengths in the optimized version (Ch. 9, Sec. 5).

The third greatest reduction in cycle time is due to the balancing of critical paths that are significantly affected by the interconnect delays. Table 10.3 lists

critical path optimizations that are not part of other optimizations and the techniques used to eliminate the problems. These optimizations involve minor revisions at most design levels – microarchitecture, functional block, circuit, and interconnect – and are discussed in detail in Chapter 9. They are good examples of the ‘not ok’ methodology paths in Figure 4.18. Together these solutions resulted in a 120ns reduction in the cycle time of the realized SOAR – 12%.

Critical Path Problem	Solutions
SWP comparison	Microarchitecture revision Faster circuits
Register file decode	Separate source and destination decoders Microarchitecture revision Faster circuits
Decode delay to PLAs – forwarding signals	Extra buffering Transistor sizes at interconnect terminations

Table 10.3- Critical Path Optimizations

PLA splitting on SOAR reduced the cycle time by 5 to 10%. This was due to a shorter instruction decode time – phase 1.

Other optimizations to the realized SOAR include a carry select scheme in the ALU, elimination of bus to bus transfers on reads, and the split datapath. Each of these optimizations reduces the cycle time of the realized SOAR by 5% or less. According to simulations, SOAR with the realized pipeline, would have a cycle time of 941ns – 9.1% faster – if these other optimizations were all included.

With the optimized pipeline the split datapath has a greater impact on the cycle time - 14.8%. If the optimized pipeline had been implemented and bus to bus transfers on reads avoided, a cycle time of 735ns could have been achieved. This would be 29% faster - 300ns - than the simulated cycle time of the realized SOAR.

Analysis of SOAR according to this methodology shows that in many places sound design decisions were made:

1. Use of internal opcodes
2. Split datapath
3. Choice of circuit blocks- except for the MAL
4. Split PLAs
5. Separate source and destination decoders
6. Balanced critical paths

However, the methodology also reveals significant improvements that were overlooked on the realized SOAR:

1. More efficient pipeline
2. Carry select scheme on the ALU
3. No bus to bus transfers on reads

## **2. Processor Results**

SOAR was initially fabricated during the winter and spring of 1985 by both MOSIS and Xerox. Both facilities used  $\lambda=2$  microns, leading to minimum line widths of 4 microns, on the original runs. The die size including scribe lines, is 432mils by 320mils. Dies passing visual inspection were packaged at the fabrication facility and then sent to Berkeley. These chips are packaged in an 84 pin grid array with a large cavity, made by Kyocera.

The debugging and testing strategy for SOAR relied heavily on CAD tools and a complete diagnostic set. As previously described (Ch. 7, Sec. 1.5), diagnostics were written to test all SOAR operations and features. This diagnostic set was used to verify the design at all levels through the use of CAD tool simulators. These diagnostics were run on the design representations associated with each design level (Table 10.4). Results from the different levels were compared and verification was complete when all results were consistent.

Level	Simulator
Architecture	Daedalus
Microarchitecture	SLANG
Functional Block	SLANG
Circuit- extracted layout	ESIM
Interconnect- extracted layout	ESIM

Table 10.4- Verification Simulators

Inconsistent comparisons resulted from bugs in several places:

1. Simulator
2. Diagnostic
3. Design

Computer simulation of complete designs of large chips requires large amounts of memory and CPU time. Therefore, writers of simulators must make generalizations and assumptions where they judge that it is safe, so that simulators are practical. However, this can lead to subtle bugs in the simulators and result in inconsistent comparisons. Diagnostic bugs arose from

misunderstandings of fine points in the architecture. Many of these were due to the processor pipelining. Ideally, the pipelining should be invisible at the architecture level but in practice this is not always true. Finally, there were many bugs in the design itself. Consistent comparisons do not guarantee that there are no bugs but it is unlikely that all simulators can miss the same bug. Bugs may also still exist due to oversights in the diagnostic set.

In addition to simulation a test die that included risky circuits, was designed, fabricated, and tested. This test die included all bootstrap drivers [Kong85]. It verified their functions and speeds.

Because of the large simulation effort and test die characterization [Kong85], SOAR was submitted for fabrication with confidence that the basic functions would work. These included reading the register file and all state registers (special registers) and input/output functions. With this functionality the chip can be debugged. Instructions can be loaded, executed, and results outputted. When bugs arise SOAR can be run in small loops that isolate the bug and read out the contents of all registers for analysis. Extra hardware for testing, such as scan in/scan out hardware, was not included on SOAR. The testability of SOAR relies on:

1. Accessibility of all state registers
2. Thorough CAD simulations to ensure basic functionality
3. Test dies to prove high risk circuits

The testing strategy followed was to first test SOAR for basic functionality and characterize its speed and I/O timing on a simple board. After basic functionality and pad timing was verified, SOAR was incorporated into the SUN workstation on the *Orion* board. The complete diagnostic set (Ch. 7, Sec. 1.5) was then downloaded into *Orion's* memory and used to complete the functional verification. After this, the final step was to bring the Smalltalk software up on

the complete system. The remainder of this section describes the initial tests and results.

### **2.1. Test Setup**

To initially characterize the processor a simple test board was built and driven by a digital analyzer – Tektronix DAS-9100. Figure 10.1 is a schematic of this board. The DAS supplies the clock cycle, wait, reset, page fault, and I/O interrupt signals. All clock phases and the underlap between phase 2 and phase 3 are generated by monostable multivibrators – 26S02s. The input clock cycle triggers a string of four of these multivibrators for these signals. The underlap between phase 3 and phase 1 is determined by adjusting the cycle time after all phase lengths and other underlaps have been set. The underlap between phase 1 and phase 2 is too short to be generated by a multivibrator and is therefore determined by inverter delays and the loading on the inverters. The phase lengths have an adjustment range of 24–460ns (Table 10.5). The underlap between phase 1 and phase 2 may be set between 10ns and 25ns. The phase 2/phase 3 underlap is adjustable between 30ns and 235ns. This test board also buffers the WAIT, RESET, PAGE, and I/O inputs with inverters.

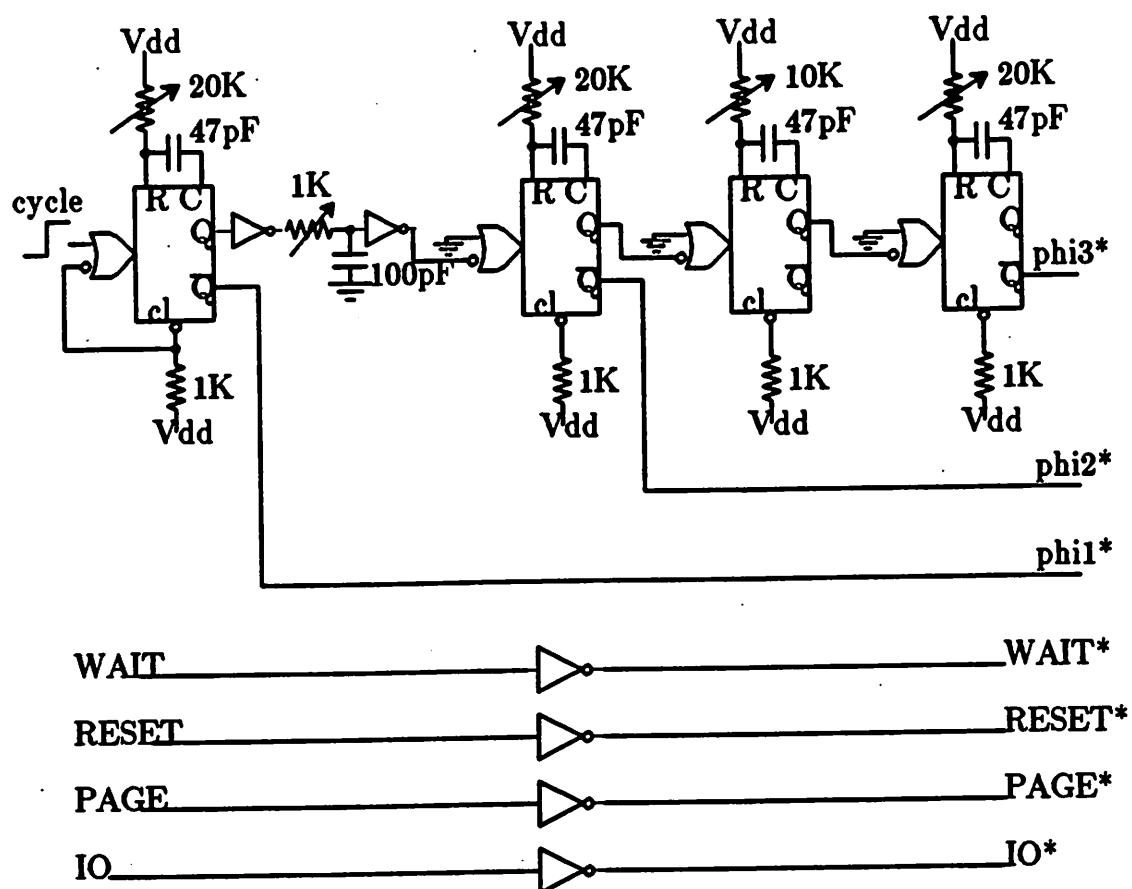


Figure 10.1- Test Board Schematic

Phase	Range
Phase 1	24-460ns
Phase1/2 underlap	10-25ns
Phase 2	24-460ns
Phase2/3 underlap	30-235ns
Phase 3	24- 460ns

Table 10.5- Test Board Clock Phase Adjustment Ranges



The complete test setup is shown in Figure 10.2. In addition to driving the test board, the DAS directly drove the data inputs of SOAR - D00-D31. Instruction sequences were programmed into the DAS and supplied to the chip through these pins. Separate power supplies were used for board circuitry, the processor supply voltage, and substrate bias. In this way clock levels and SOAR supply voltages could be independently adjusted if necessary. All signals could be monitored on the oscilloscope. Results of processor operations were read off chip through the address outputs using the call, jump, and return instructions. The data acquisition channels of the DAS collected and displayed this information.

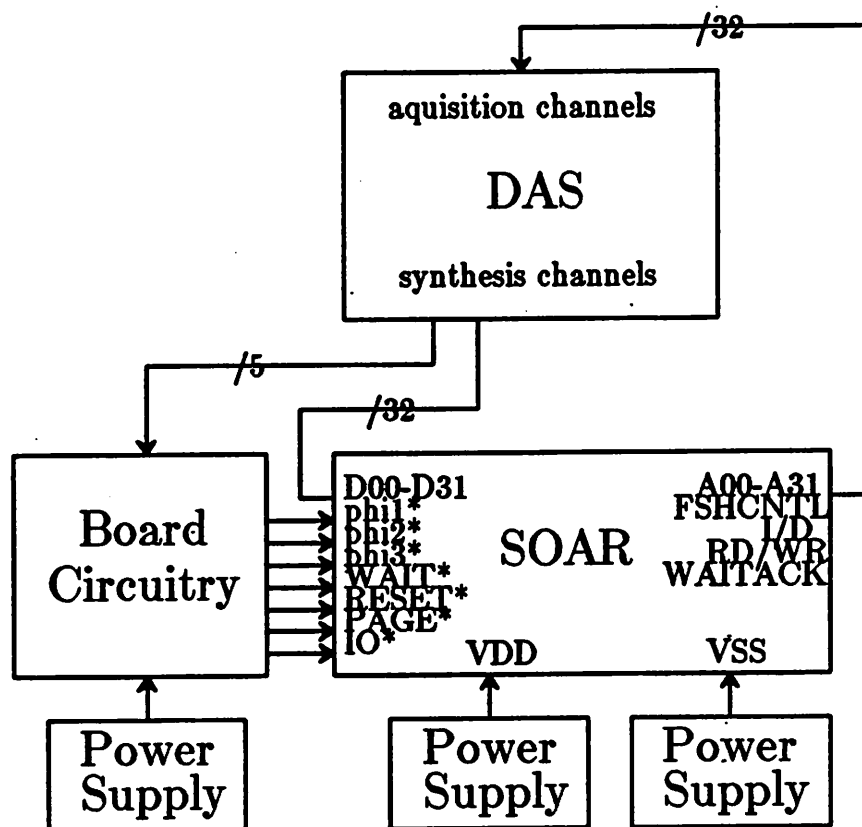


Figure 10.2- Test Setup

## **2.2. Functionality**

Using this setup the functionality of a significant portion of SOAR was verified. Simple programs that initialize SOAR after resetting and jump to test code showed the following features to be functional:

**Jumps and calls- address formation**

- saving of the return address on calls
- CWP change on calls

**Returns- address calculation**

- interrupt enable
- CWP change

**Loads- address formation**

- data capture

**Shifts****Adds****Inserts****Extracts****Ors****Reads- register file**

- special registers
- sign extension of immediates

**Writes- register file**

- special registers

**Forwarding- load forwarding**

- ALU forwarding

**Trap mechanism- vector formation**

- priority encoding
- saving of the return address
- shadow register operation

**Trap types- illegal opcode**

- I/O interrupt
- page faults

### External signals- WAIT

- RESET
- FSHCNTL
- I/D
- WAITACK

The only feature tested that did not work was the nilling option of the return instruction. Investigation of this problem showed that the drivers that write to the register file were not designed to be strong enough to write to six registers simultaneously. Functions that were not tested initially were stores, load multiple, pointer to register, skips, and some types of traps. It was judged that enough worked to incorporate SOAR into the *Orion* board and complete the testing there.

### 2.3. Speed

The minimum cycle time was determined by writing test programs that exercised the critical paths and reducing the phase lengths until failure occurred. This was done with SOAR at its stable operating temperature, a supply voltage of 5 volts, no substrate bias, and clock high levels of 4.5 volts. Phase lengths were measured from the midpoints of the transitions on the oscilloscope trace and were repeatable to within 5ns. The minimum cycle time is 400ns for the MOSIS chips and 330ns for the Xerox chips by this method. Table 10.6 summarizes the minimum phase lengths. The following sections analyze the individual phase lengths in detail for the MOSIS chips. Pad timing is shown in appendix D.

Phase	MOSIS	Xerox
Phase 1	90ns	55ns
Phase1/2 underlap	<10ns	0ns
Phase 2	90ns	90ns
Phase2/3 underlap	<25ns	<25ns
Phase 3	145ns	125ns
Phase3/1 underlap	40ns	35ns
Total	<400ns	<330ns

Table 10.6- Measured Cycle and Phase Lengths

### 2.3.1. Phase 1

The length of phase 1 is determined by instruction decode and sign extension (Ch. 9, Sec. 4.1). The slowest signals that must settle in phase 1 are the forwarding and shadow control lines (Figure 9.14). As previously described, the gates at the terminations of these signals were designed so that these signals would not limit phase 1. A program that exercised forwarding and the same program with NOPs inserted to avoid forwarding require the same phase 1 length (within the accuracy of the measurements) - 85-90ns (Table 10.7). Shadowing also proved to not be a limiting factor to the phase 1 length. The many other control lines of phase 1 have settling times within the same ranges according to simulations (Figure 9.14).

Feature	Minimum Phase 1 Length
Forwarding	85ns
No forwarding	90ns
Shadowing	<85ns

Table 10.7- Measured Minimum Phase 1 Length

### 2.3.2. Phase 2

Operand reads are assigned to phase 2 (Ch. 9, Sec. 4.2). There are several operand sources and simulated access times vary accordingly (Table 9.9). Test programs were written to exercise these operand reads (Table 10.8).

Signal	Origin	Measured Phase 2
BusA	ALU forwarding to busD	70ns
BusA	SWP to busD	75ns
BusB	ALU forwarding to busD	80ns
BusA	CWP to busS	70ns
BusA	Load forward	65ns
BusB	Load forward	60ns
BusA,B	Register file	90ns

Table 10.8- Measured Minimum Phase 2 Length

The slowest reads are on register file accesses - 90ns. According to simulations the register file cell is faster than the master slave latch cell (Ch. 6, Sec. 1.2.2). Therefore, this extra delay is due to the extra time needed to enable the word line drivers before they can drive the word lines. At the register file, phase 2 gates a driver that enables the word line drivers. In contrast to this, phase 2 directly gates the control line drivers for latch accesses, resulting in a faster access.

The reads that involve the transfer of data from one bus to another bus all require 70-80ns. This is the situation during ALU forwarding and when using the CWP, PSW, SWP, TB, or PC as operands.

The fastest operand accesses occur when data is driven directly onto the bus to the ALU input - 60-65ns. This occurs during load forwarding, shadow register operands, immediate operands, and the zero operand.

### **2.3.3. Phase 3**

The slowest path of phase 3 involves the ALU. It occurs when carry must propagate across seven nibbles and the three least significant bits of the most significant nibble (Ch. 9, Sec. 2.1). This requires 145ns (Table 10.9). Phase 3 measurements were taken for carry propagation across 3 to 7 nibbles. Using this data, the delay of the carry bypass is calculated to be 8.8ns with a standard deviation of 4.8ns.

Event	Measured Phase 3
ALU to Destlatch- 7FFFFFFF+1	145ns
ALU to Destlatch- 07FFFFFF+1	140ns
ALU to MAL- 07FFFFFF+1	120ns
ALU to MAL- 007FFFFF+1	105ns
ALU to MAL- 0007FFFF+1	100ns
ALU to MAL- 00007FFF+1	<90ns
Carry bypass speed	8.8ns

Table 10.9- Measured ALU Limits to Phase 3

The other major function of phase 3 is the write of the result from the previous instruction. Table 10.10 shows the required phase 3 lengths for different types of writes. The slowest write is to the register file - 110ns. Data is first put onto bus D and then transferred to buses A and B to get to the register file. Special registers that are written to directly from busD - such as the SWP - require 100ns. 90ns is needed to write to the CWP which is located in the control section.



Write Destination	Measured Phase 3
Register file- busD to busA,B	110ns
SWP- busD	100ns
CWP- control	90ns

Table 10.10- Measured Write Limits to Phase 3

#### 2.4. Process Effects

The process design level was a fixed input during SOAR design. The importance of this design level to the characteristics of the processor can be seen by comparing the measured characteristics to simulation results. Simulations were based on a more conservative set of parameters than those of the fabricated processor.

Transistor parameters used in the simulations and measured on wafers containing the fabricated SOAR are shown in Tables 10.11a and 10.11b. These transistor

Parameter	Simulation	Fabricated MOSIS	Fabricated Xerox
VTO (V)	.6	.93	.65
GAMMA ( $\sqrt{V}$ )	.40	.41	
KP ( $\mu\text{A}/\text{V}^2$ )	17.2	32.3	49.5
UO ( $\text{cm}^2/\text{Vs}$ )	350	654	
LAMBDA ( $\text{V}^{-1}$ )	.01	.021	

Table 10.11a- Enhancement Transistor Parameters

Parameter	Simulation	Fabricated MOSIS	Fabricated Xerox
VTO (V)	-2.5	-3.3	-4.2
GAMMA ( $\sqrt{V}$ )	.51	.43	
KP ( $\mu\text{A}/\text{V}^2$ )	18	31.4	44.8
UO ( $\text{cm}^2/\text{Vs}$ )	366	900	
LAMBDA ( $\text{V}^{-1}$ )	.015	0	

Table 10.11b- Depletion Transistor Parameters

parameters determine the current that is available to charge and discharge capacitive loads. In the saturated region a first order approximation for transistor current is:

$$I_{\text{sat}} = KP/2(W/L)(V_{\text{GS}} - V_{\text{T}})^2[1 + \text{LAMBDA}(V_{\text{DS}})]$$

Tables 10.12a and 10.12b show the saturation currents for low  $V_{\text{DS}}$  and no

substrate bias ( $V_T = V_{TO}$ ) using the simulation and measured parameters. The enhancement and depletion MOSIS transistors conduct 1.6 and 3.1 times as much current in the saturation region as the transistors used in simulations. Xerox transistors were also much stronger than the transistors of the simulations - 2.8 and 7.0 times as strong for the enhancement and depletion devices, respectively. In the linear region transistors can be described by their 'on' resistance:

$$R_{on} = 1/g_m = 1/[KP(W/L)(V_{GS} - V_T)]$$

The 'on' resistances of the fabricated transistors are much less than those of the transistors used in simulations - 24% to 58% of the simulation resistances. Thus, the fabricated transistors conduct more current than the simulation transistors in all situations of interest.

	$I_{sat}/W/L$	Normalized $I_{sat}$	$R_{on}/L/W$	Normalized $R_{on}$
Simulation	.166ma	1.0	13.2K	1.0
MOSIS (fab)	.268ma	1.6	7.6K	.58
Xerox (fab)	.468ma	2.8	4.6K	.35

Table 10.12a- Enhancement Transistors -  $V_{GS}=5.0$ volts

	$I_{sat}/W/L$	Normalized $I_{sat}$	$R_{on}/L/W$	Normalized $R_{on}$
Simulation	.056ma	1.0	22.2K	1.0
MOSIS (fab)	.174ma	3.1	9.6K	.43
Xerox (fab)	.393ma	7.0	5.3K	.24

Table 10.12b- Depletion Current Sources

The rise and fall times of many nodes are limited by the capacitive loads of the nodes and the currents available to drive these loads. Capacitance depends on area and process dependent capacitance parameters. Capacitance parameters used in simulations and for the fabricated devices are shown in Table 10.13. Depending on the size, shape, and type of capacitor, fabricated node capacitances may range from .7 to 1.4 times those of the simulations. Nodes dominated by gate capacitance have higher capacitances in the fabricated devices – 1.22 to 1.37 times as much for the MOSIS and Xerox devices, respectively. This is the situation for the output nodes of high fanout gates. Another common situation is an interconnect node that is predominately a field capacitor. The fabricated field capacitors have similiar values to those of the simulations – .83 to 1.17 as much capacitance.

Capacitor	Simulation	MOSIS		Xerox	
		value	normalized	value	normalized
Gate	.41fF/ $\mu^2$	.56fF/ $\mu^2$	1.37	.50fF/ $\mu^2$	1.22
Field	.06fF/ $\mu^2$	.07fF/ $\mu^2$	1.17	.05fF/ $\mu^2$	.83
Diffusion side	.35fF/ $\mu$	.50fF/ $\mu$	1.43		
Diffusion area	.16fF/ $\mu^2$	.11fF/ $\mu^2$	.69		

Table 10.13- Capacitance Parameters

When charging a node, current is supplied by a depletion transistor used as a current source. In this situation the current available to drive the load is 3.1 to 7.0 times higher for the fabricated devices than for the simulated devices (Table 10.12b). This more than offsets the higher capacitances of the fabricated devices, resulting in shorter rise times. For example, gate dominated capacitances will be charged approximately 2.3 times faster on MOSIS devices than in the simulations.

Nodes are discharged through enhancement pulldown devices. Fabricated pulldown devices are stronger than those of the simulations - 1.6 to 2.8 times stronger (Table 10.12a). This also more than offsets the higher capacitances of the fabricated devices but not as dramatically as for the depletion devices. Thus, fall times on the fabricated devices are slightly faster than those of the simulations.

Interconnect resistances are also process dependent. Resistivities for the fabricated devices and simulations are shown in Table 10.14. Signal lines that could not be routed in metal are routed in polysilicon. Polysilicon for both the MOSIS and Xerox processes, has a much lower resistivity than anticipated - 40% to 48% of the simulation value. Signal lines that have delays limited by their RC

time constants are consequently faster on the fabricated devices than in the simulations. This is the situation for control lines, word lines, and interconnects with long polysilicon crossovers.

Resistor	Simulations	MOSIS	Xerox
Diffusion	20Ω/□	26Ω/□	19Ω/□
Polysilicon	50Ω/□	20Ω/□	24Ω/□

Table 10.14- Polysilicon and Diffusion Resistivities

The shorter cycle times of the fabricated devices – 400ns and 330ns for the MOSIS and Xerox devices respectively – than of the simulated device – 1035ns – can be traced to the differences in process parameters. The stronger depletion and enhancement transistors of the fabricated processors result in shorter rise and fall times than in the simulations. Lower polysilicon resistivities also contribute to the shorter cycle times of the fabricated processors.

## 2.5. Summary

The SOAR processor chip was started in the spring of 1983 with the primary goals of an instruction set targeted to compiled Smalltalk and a cycle time of 360ns (Table 10.15). These goals are critical for overall system performance. Allowing for fabrication variations, a cycle time of 330ns to 400ns was achieved with 4micron NMOS technologies. Functionality according to the target instruction set was verified except for the register nilling option on returns. Thus, the SOAR processor is close to being fully successful in meeting the two primary goals. Other goals are not crucial for system performance but are a matter of

practicality – power, size, packaging, and design time. The die size was limited by the process technology in the shorter direction to 320mils. Allowing for some slack space between the chip and cavity wall, the 440x440mil package cavity limited the chip length to 435mils. Originally design time was predicted to be twice that of RISCII due to the increased complexity of SOAR. These goals were all met except for power consumption.

Characteristic	Goal	Realized
Instruction set	see Chapter 5	all but register nilling
Cycle time	360ns	330-400ns
Power	2.5W	2.75-3.00W
Die size	320x435mils	320x432mils
Transistors	50,000	35,700
Process	4micron NMOS	4micron NMOS
Package	84 PGA	84 PGA
Design time	67.2 person months	38.5 person months

Table 10.15- SOAR Processor Characteristics

### 3. Architecture Results

Although this methodology is primarily concerned with tradeoffs between the five design levels – microarchitecture, functional block, circuit, interconnect, and process levels – tradeoffs with the architecture and system level are also important. The impact of the architecture and system requirements on the implementation is important in design decisions concerning the architecture and

system.

SOAR included many features to increase Smalltalk execution speed [Unga85]. Some of these prove to greatly reduce the number of cycles needed to execute programs and others are less significant. To further understand the worth of these features, their effects on the speed, power, area, and complexity of the chip should be considered.

### 3.1. Overview

SOAR is based on a RISC style of architecture [Kate83]. RISCs traditionally have a relatively small control section – 10% of RISCII for example. The benefits of this are a fast and relatively simple processor. SOAR is a more complex processor than previous RISCs, partially due to the features that enhance Smalltalk execution. The control section is therefore larger – 29% – but still less than half of the processor (Table 10.16).

Section	$M\lambda^2$	% Area
Datapath	11.05	51.2
Control- circuits	4.45	20.6
- routing	1.82	8.4
Periphery	4.28	19.8
Total	21.60	100.0

Table 10.16- Major SOAR Sections



### 3.2. Area and Geometry

Table 10.17 lists the SOAR features that enhance Smalltalk execution and potentially require special purpose circuitry. Transistor counts and the areas occupied by these features are listed. The features requiring the most extra area are the shadow registers, register windows, byte instructions, and the pointer to register capability. The loadc and sll instructions use no extra circuitry. Together, all Smalltalk features use 12.4% of the chip area (excluding the periphery) and 13.9% of the transistors.

Feature	Number of Transistors	% Transistors	Area (M $\lambda^2$ )	% Area (internal)
Register windows	969	2.71	.430	2.48
Inline cache	24	.07	.029	.17
Byte instructions	308	.86	.345	1.99
Tagged integers	211	.59	.076	.44
Forwarding	136	.38	.045	.26
Fast Shuffle	18	.05	.014	.08
Tagged immediates	79	.22	.034	.20
Nilling	254	.71	.038	.22
Trap instructions	123	.35	.048	.28
Loadm/storem	269	.75	.105	.61
Pointer to register	705	1.98	.230	1.33
Vectored traps	453	1.27	.171	.99
Generation tags	165	.46	.065	.38
Loadc	0	0	0	0
Shadow registers	1159	3.25	.478	2.76
Sll	0	0	0	0
Extracodes	99	.28	.034	.20
<b>Total</b>	<b>4972</b>	<b>13.94</b>	<b>2.142</b>	<b>12.36</b>

Table 10.17- Transistor Count and Area of Smalltalk Features

### 3.3. Complexity

Table 10.18 outlines the complexity of these added features. Complexity can be measured in several ways. One measure is the number of circuit blocks that the feature is distributed among. All blocks must be correctly connected and this leads to routing complexity. Routing delays contribute to settling times and therefore a large number of circuit blocks indicates a potentially long critical path. These paths require extra attention so that they do not limit processor speed. Register windows, shadow registers, and nilling require the most circuit blocks - ten or more.

Feature	Number of Circuit Blocks	Number of Hand Drawn Transistors	% Diagnostics
Register windows	20	149	14.9
Inline Cache	4	8	2.2
Byte instructions	6	128	.7
Tagged integers	8	25	12.0
Forwarding	5	106	3.7
Fast Shuffle	4	4	0
Tagged immediates	4	12	0
Nilling	10	10	2.8
Trap instructions	6	14	1.8
Loadm/storem	6	16	.9
Pointer to register	7	78	6.7
Vectored traps	4	0	3.2
Generation tags	5	8	6.0
Loadc	0	0	0
Shadow registers	11	50	.9
Sll	0	0	.2
Extracodes	3	1	4.3
<b>Total</b>			<b>60.3</b>

**Table 10.18- Complexity of Smalltalk Features**

A second measure of complexity is the number of transistors that must be drawn by hand. These require extra design time and are more prone to bugs due to human error. Thus, they also increase debugging time. The features that are

most complex by this measure are register windows, byte instructions, forwarding, pointer to register, and the shadow registers. Many features required ten or less hand drawn transistors – inline caching, fast shuffle, nilling, vectored traps, generation tags, loadc, sll, and extracodes.

A third indication of complexity is the number of diagnostics needed to test the feature. This is perhaps more a measure of architectural complexity than implementation complexity. The diagnostic effort for a given feature is a reflection of the variations and subtleties of that feature. The tagged integers and register windows require the most diagnostics – 12.0% and 14.9% respectively. Also requiring a large number of diagnostics are the pointer to register, generation tags, and extracodes features – 6.7%, 6.0%, and 4.3% respectively. All other features need fewer diagnostics.

### **3.4. Speed**

As previously discussed, the cycle time is limited by instruction decoding, register file reads, and ALU operation. None of these critical paths involve the Smalltalk features. Therefore, although these features have added area and complexity, they do not appear to affect the cycle time. This is due to the effort put forth to identify and balance the critical paths (Chapter 9). Two critical paths that would have caused a significant speed reduction – 12% – are the register file decode and SWP comparison. These can be traced to the register windows and pointer to register feature, respectively.

Although the Smalltalk features do not explicitly appear in any of the critical paths, they may add to signal delays because of the increase in chip size due to the area that they occupy. This effect is difficult to measure. Some of the extra circuitry fits into areas that would otherwise have been left empty and therefore did not increase the processor size; other additional circuitry probably increased

Feature	% Slowdown if omitted	% Transistors	% Area	Complexity Index
Register windows	46	2.71	2.48	10.0
Inline cache	33	.07	.17	1.3
Byte instructions	33	.86	1.99	4.0
Tagged integers	26	.59	.44	4.6
Forwarding	15	.38	.26	4.0
Fast shuffle	11	.05	.08	.8
Tagged immediates	9.6	.22	.20	.9
Nilling	4.3	.71	.22	2.5
Trap instructions	3.9	.35	.28	1.7
Loadm/storem	3.4	.75	.61	1.6
Pointer to register	3.1	1.98	1.33	4.4
Vectored traps	2.9	1.27	.99	1.4
Generation tags	1.3	.46	.38	2.3
Loadc	.46	0	0	0
Shadow registers	.04	3.25	2.76	3.2
SII	0	0	0	0
Extracodes	0	.28	.20	1.5

Table 10.19- Effects of Smalltalk Features

From these processor considerations, the greatest benefits for the lowest costs are due to the inline caching and fast shuffle features. Register windows, tagged integers, and byte instructions have large benefits but also have relatively high costs.

In the 3 to 5% range for speed increase all costs are relatively low except those of the pointer to register feature. Pointer to register is costly by all measures and reduces the cycle count by only 3.1%. Vectored traps are also fairly costly in terms of circuitry and area and only contribute a 2.9% speedup.

The features that contribute less than 1% to the speed increase cost very little except for the shadow registers. The shadow registers involve a large amount of circuitry and area, and are moderately complex. These shadow registers stand out as the first feature to eliminate in any redesign.

#### 4. References

[Kate83] Katevenis, M. G. H.; 'Reduced Instruction Set Computer Architectures for VLSI', Ph.D. Thesis, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., 1983.

[Kong85] Kong, S. I.; 'Some Design Techniques for High Performance MOS Circuits', M.S. Report, EECS Dept., University of California, Berkeley, Ca., January 1985.

[Unga85] Ungar, D. M., 'The Design and Evaluation of a High Performance Smalltalk System', Ph.D. Thesis, Computer Science Division, EECS Dept., University of California, Berkeley, Ca., 1985.