Advanced Silicon Compiler in Prolog

I. Pyo, C. Su, I. Huang, R. Pan, Y. Koh
H. Chen, G. Cheng, C. Tsui, Alvin Despain

CENG Technical Report 93-22

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-6006

May 1993

ADVANCED SILICON COMPILER IN PROLOG

Iksoo Pyo, Ching-long Su, Ing-jer Huang, Ricky Pan, Yongseon Koh,
Hsu-tsun Chen, Gino Cheng, Chi-ying Tsui and Alvin M. Despain

Department of Electrical Engineering-System

University of Southern California

Los Angeles, CA 90089

ABSTRACT

An application specific microprocessor design is a continuous challenge since it has been a main issue of a customized chip. There have been lots of approaches to reach such a goal. However, they are limited at a specific design range. The vertical range of them belongs to one of four categories - Behavioral high-level synthesis, Topological logic-level synthesis, Geometrical circuit-level synthesis and Mixed level synthesis. The horizontal range of them is mainly based on many uncharacteristically not-well-formed heuristics and contraints.

The Advanced Silicon-compiler in Prolog (ASP) is a full-range hardware synthesis system. The goal of the ASP is to automatically synthesize a single VLSI chip processor from a high-level specification of the Instruction Set Architecture (ISA) written in a subset of standard Prolog. Our approach is to study a specialized, vertical slice of the design space that spans applications, language design, compiler design, instruction set design, microarchitecture, and VLSI implementation. The idea was to develop, all the same time, a hardware processor (e.g. PLM, VLSI-PLM, BAM, SLAM, ...) optimized for design automation tools written in Prolog, the corresponding software support modules (e.g. Compiler, Language, BM, DCG, ...), and a comprehensive design automation system (e.g. ASP I, ASP II, ASP III, ...) to help design our hardware systems.

1. INTRODUCTION

The ASP effort is part of the Aquarius Project, the goal of which is to produce high performance Prolog engines, realized in part with specialized high-quality microprocessors. Thus the focus of ASP is a single-chip microprocessor synthesis. ASP is also meant to test Prolog as an implementation vehicle for large programs.

Conceptually, each level of abstraction is composed of a simulator module, a translator module, an estimatior module, a design library module, a design program (engine) module and a design rule. This structure is illustrated in Figure 1.1. Each level accepts a specification in a formal specialized language and produces a more detailed and concrete specification in a different specialized language. In addition, in order to determine which design choices should be made, a benchmark program is also provided to each level so the developing architecture can be simulated and measured relative to the design choices.

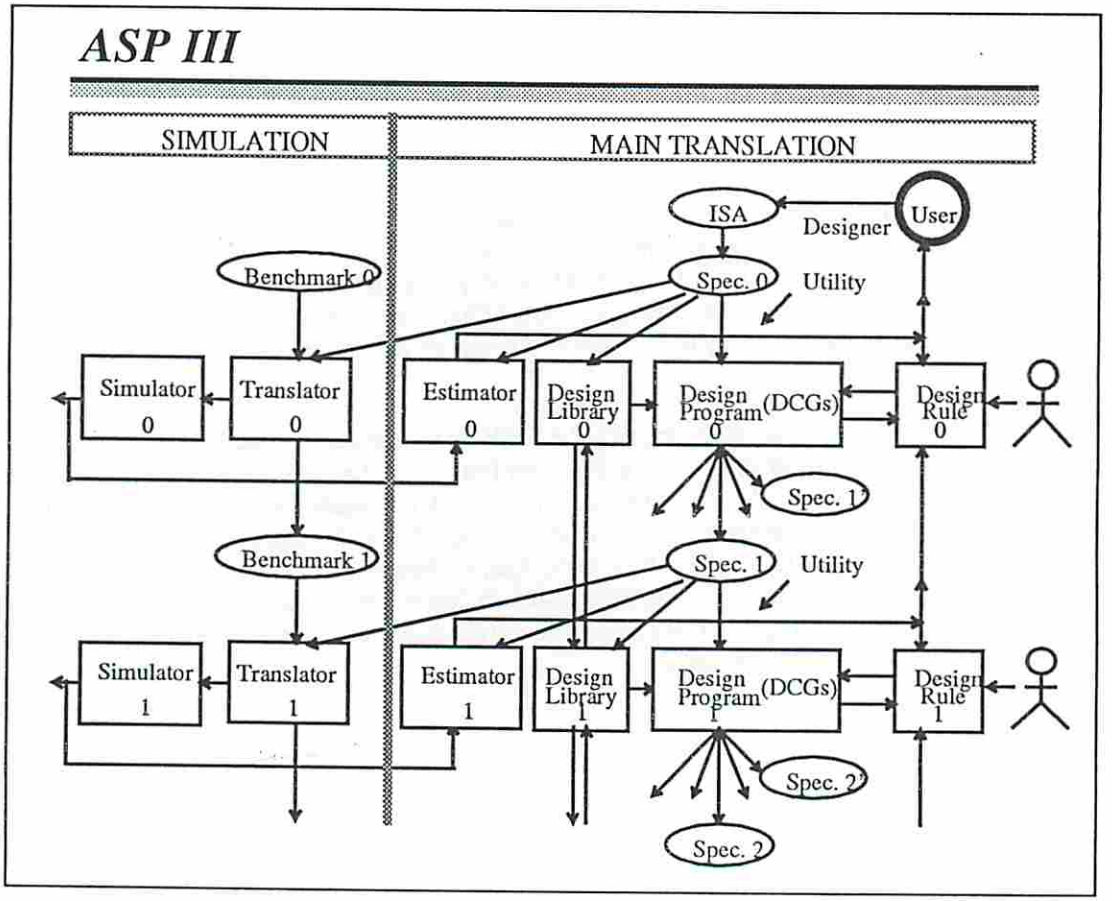


Figure 1.1 ASP III

The ASP system works with either the user constraint from outside world or the system estimated constraint from inside system. The more the designer constraints, the less the design space expands.

An efficient separation between main translation languages and design rules gives us a promise of an opti-

mized design decision at each level. Design rules from abstract behavior to concrete of each level are interpreted externally based on the target machine contraints or internally based on the estimator selected constraints. Human-driven constraints or design choices can be applied as the translation rules. However, the system produces multiple designs by default. The local optima will be selected by the chosen design rules which the estimator chooses along with the simulator results and the given specification. In this report, the experiences during expanding the specifications are described with detailed examples in Prolog, too.

In the ASP system, the synthesis phase of a compiler design for the target machine code can be easily simplified with the information of the global data/control flow analysis which is a level between parsing and scheduling. So, the HLL benchmarks described with an intermediate code are converted into a target machine code at the above level of ISA. Those target set of benchmarks will be translated into the proper level representation. This synthesis from the set of bechmarks to the target machine code generation is one of the many additional advantages which our ASP system can produce. So, the semantic gap between software and hardware engineer can be narrowed.

The way to make possible different designs from several specifications which were made by set of benchmarks is just a hack of translation with rules. This is an Application Specific Design (desired machine oriented design). It is quite difficult to figure out possible different designs from the same specification. This is an Optimized Contraint (performance) based Design. In OCD, the designer needs a criteria to compare at each design space. This complex criteria consists of the evaluation metrics which can be used as a selection barometer by an estimator. The estimator of each level evaluates based on the input specification and the simulation result. The selected design rules can guide the formal system to generate the local optima. And they can be feed back into the upper level whenever the constraints are not satisfied by any rules.

The ASP system is a design automation (DA) system as opposed to a computer aided design (CAD) system. This is illustrated in Figure 1.2. Here the level of abstraction is represented on the ordinate while the generality of the design is represented along the abscissa. The ASP design automation system is shown as the tall, narrow triangle, while more general, CAD tools are shown as short rectangles. ASP is composed of about fourty levels of abstraction. Each level is very tightly integrated into the levels above and below it. Collections of CAD tools almost never achieve such close integration.

Specification expansion between abstractions broads choices not only for an optimized translation decision but for an application specific design. Each specification is translated by following DCG (Definite Clause Grammar) rule which regulates each level's translation mostly by syntetic checking and translation rules.

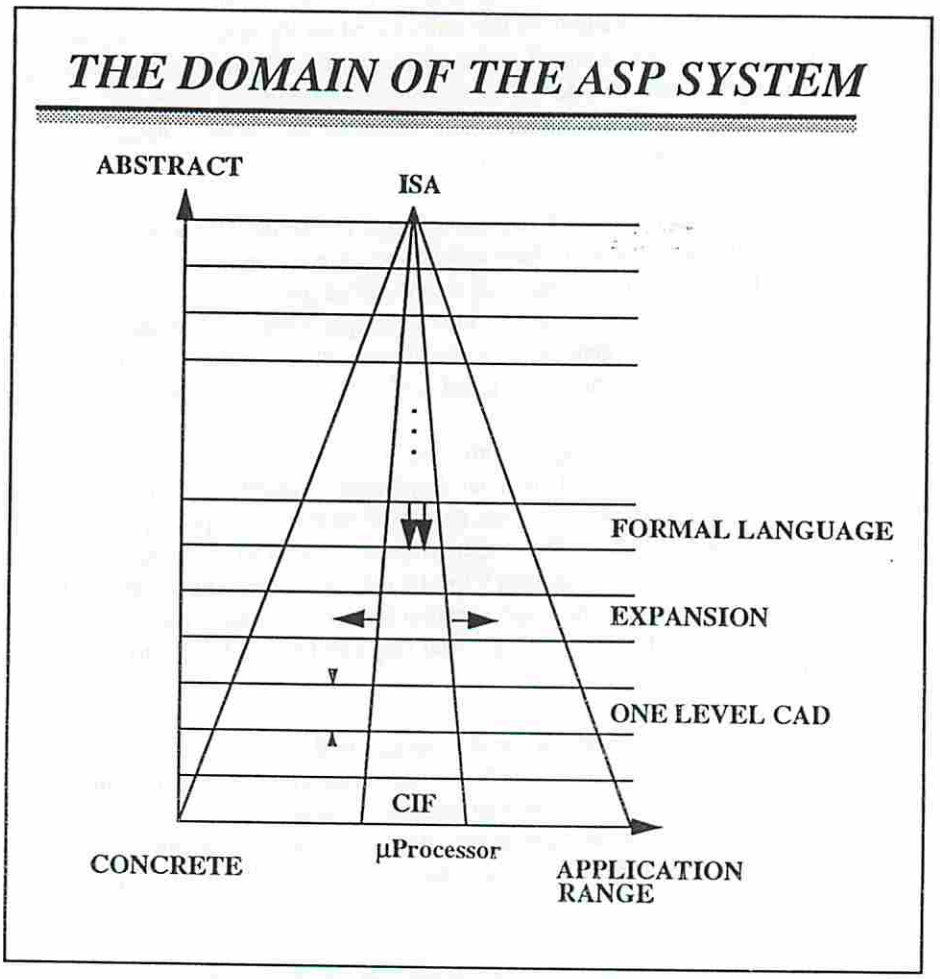


Figure 1.2 The Domain of the ASP System

2. FIPER

2.1 Formal Methods in DA

A critical issue in DA is to express the specifications for an object to be designed in a manner that can be "understood" by a computer program. For this purpose, the use of formal languages is absolute. If the formal approach is effective, significant advantages could be realized in creating design automation systems, and progress in developing a general thoery of design could be achieved.

The goal of the project is to determine the utility of formal methodology in design automation systems. More specifically, the usefulness of formal syntactics and semantics to define the meaning of specifications and the design process itself is to be determined.

By automating some of the more repetitive design functions more time is created for not only additional creative activities and for reducing the overall design time, but also for the evaluation of additional sucessive designs. This later capability, known as rapid prototyping is especially critical because it also allows the designer to get the specification right and to explore multiple design alternatives.

The cirtical idea in employing a formal high-level language (HLL) to express designs is that much of the repetitive, low-level, straight forward design tasks can be automated by incorporation into the compiler that converts the HLL into a more instantiated form. One view of an HLL compiler is that it is some form of an expert system that incorporated knowledge about how ceratin low-level programing tasks can be accomplished, i.e. designed.

A specification of an object to be designed can be given in a number of way. Industrial and military standards have been developed to make some of these specialized languages more specific and less ambiguous. However, with the exception of computer programming languages, these standrads are generally written in English and contain many ambiguities. One specific idea of this research is to employ formal language methods to define specialized specification languages in the area of design automation of digital system.

If specifications are to be given by a specialized language format, then there is a need to formally specify this language so that specifications in this language can be interpreted, simulated, transformed, analyzed and elaborated. This requires two major tasks, first the specification of the language syntax and second, the specification of the language semantics. For programming languages, syntax is generally specified in a standrad way known as the BNF (Backus Normal Form). Semantic definition is much more difficult and is sometimes only defined informally.

In this research we used a Definite Clause Grammar (DCG) (which is a form of attributed grammar) to formally specify both the syntax and semantics. In the DCG method the syntax specification is essentially the same as the BNF grammar and the semantic specification is accomplished by attaching "attributes" to each component of the BNF specification. The result is a relative semantic specification with the semantics defined only relative to the provided attributes. However, for a given specification, in more concrete and elaborated terms, than the specification itself.

2.2 Definite Clause Grammar

Definite clause grammars (DCG) are a form of attributed grammars implemented in Horn clause form, and in our case in the programming language Prolog. The use of Prolog is very convenient as the Prolog interpreter can be directly used to evaluate language expressions using the DCG specification of the language. Our goal is to transform an abstract design specification into a more concrete and elaborate set of design specifications automatically.

DCG system itself checks a syntex whenever it succeeds and adapts a semantic actions whenever it needs. Most of them are type checking. The utilization will be maximized when it is replaced with compact flow guidance. DCG system travels a node guided by rules and works as an action (translation) applier. In other words, DCG system builds a parsing map (tree or graph) as a structure, finds a road (path) to travel, and takes an action at each station (node). The rejection can happen whenever the system fails or an unacceptable condition occurs.

DCG does a hierarchical decomposition from Non-terminals to Terminals. The non-terminal symbol will be translated into a new terminal symbols by the translation rules which can be either design rules or synthesized (analyzed or flow) rules. DCG depends on the input list structure which determines the decomposition process. The system eventually reaches an actual terminal symbol which is a replace goal with lots of backtracking. DCG does not use a cut which can protect backtracking and assert/retract built-ins which can produce a lot of side effects.

DCG formalization has two major advantages. The one to one correspondance between BNF grammar and DCG is complete. And the designer has a flexibility to represent and express the language system in his own concept.

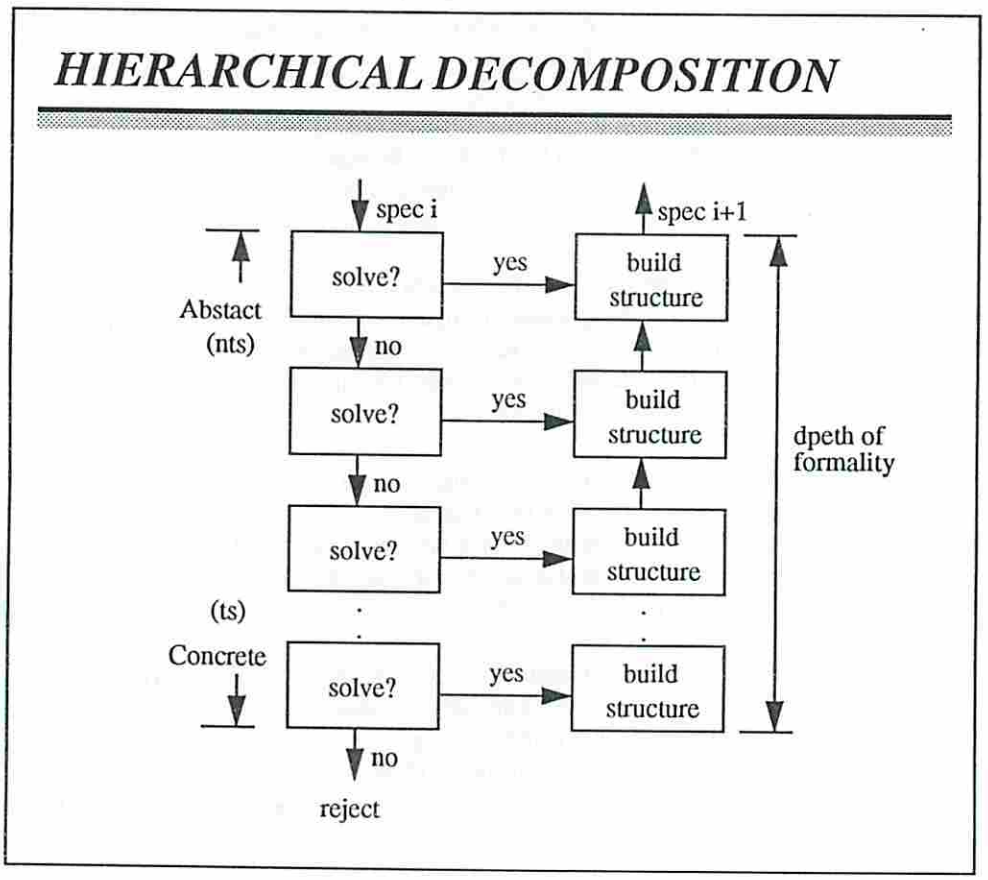


Figure 2.1 Hierarchical Decomposition

2.3 High Level Synthesis

The high level synthesis of the ASP system called FIPER generates a data path (a set of functional units) and control path (a set of control signals), controlled by a finite state machine, from an input specification in the form of

instruction set architecture definition (see Figure 2.2). FIPER of ASP performs two basic functions. It translates Porlog constructs into hardware equivalents, and it creates and locates hardware resources within various constraints. It uses a combination of compiler analysis and hardware knowledge. Algorithmic compiler techniques - dependency analysis, register allocation, and dependency-based scheduling - are used to produce a basic design with constraints. Hardware specific heuristics and knowledge about the characteristics of functional units are then used to generate a design within the constraints. The implementational view of FIPER is illustrated in Appendix A.

The FIPER operates in 5 phases. It scans the input Prolog for correctness in terms of the subset of Prolog it supports. It translates the Prolog into a data flow graph, an associated control flow graph, and a set of register transfers that embody the data flow graph. It scans the register transfers for global resource contraints. It schedules functional units and the movement of data between them. It allocates functional units.

Knowledge about functional units is packaged in a library, which also serves as the interface to lower synthesis levels. Each library member also can contain the logic equation and other information necessary for it to be realized as a circuit. For example, the global data/control flow analysis can use either the logic level specified library or the random logic generator specified truth table.

Formalization of this translation is defined by the DCG based on the translation rule. It has two very acceptable heuristics. First, a few words are reserved to detect mainly global data flow branching factor. Second, the major branch nodes have only the sequential execution cycle. In the following sections, we will describe the important issue/algorithm during the high-level sythesis, the correspondance level specifications and the DCG language of translation.

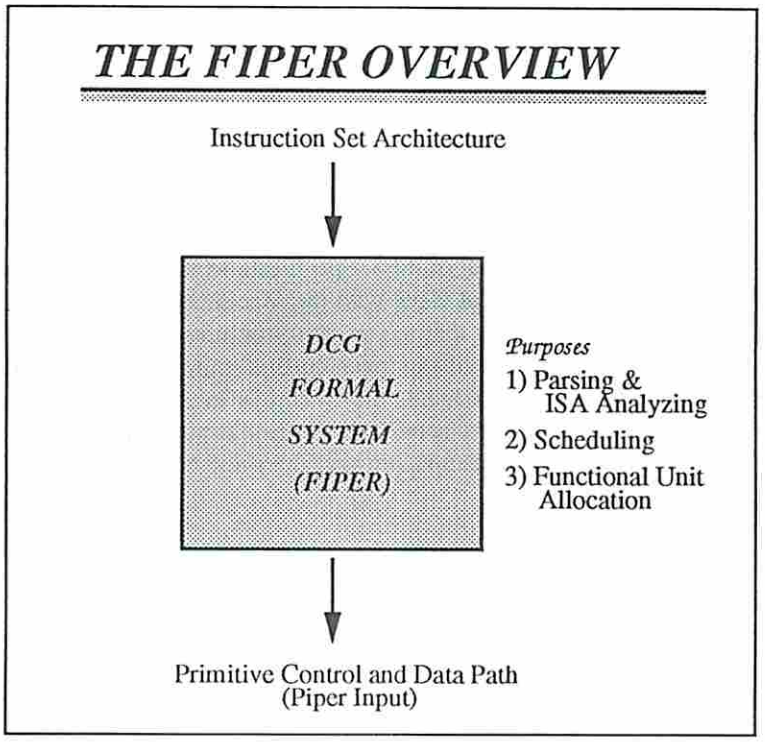


Figure 2.2 The FIPER Overview

2.4 Instruction Set Architecture

The structure of the Design Automation (DA) system we will employ will be layered. Each layer will accept a formal specification (in its input language) and will produce one or more output specifications in its output language. The output specification then becomes the input to the next layer.

The first part of the investigation is to define a DCG for a language to specify the instruction set architecutre (ISA) for a single-chip VLSI microprocessor. One way to represent a behavioral characteristics of a microprocessor is the linear recursive Instruction Set Architecture which has one tail recursion main. For example, consider a fragmentation of SM1 (Simple Machine written in a sub-set of Prolog). The complete ISA of SM1 is at the appendix B.

```
% Symbolic representation of Instruction Set Architecture

fsm(AC, PC , Memory):-
        fetch(PC, Op, Adr, Memory),
        P1 is PC + 1,
        execute(Op, Adr, AC P1, Memory, AC_next, PC_next, Memory_next),
        fsm(AC_next, PC_next, Memory_next).

fetch(PC, Op, Adr, Memory):-
        memory_read(PC, Memory, [Op,Adr]).

execute(add, X, AC, PC, Memory, AC_next, PC, Memory) :-
        memory_read(X, Memory, Data),
        AC_next is AC + Data.

class([register,master_slave,r1,16]).
class([counter,programmable,c1,16]).
class([memory,ram,m1,[16,64]]).

attributed([r1,ac,fsm,1]).
attributed([c1,pc,fsm,2]).
attributed([m1,m,fsm,3]).
```

The highest level of specification in ASP is executable Prolog. The system supports a subset of standard Prolog that roughly corresponds to the descriptive power of ISPS. Specifications in this subset can be both simulated and synthesized. This general approach is similar to that taken with the MacPitts system for LISP and with the Flamel system for Pascal. The restrictions on the ISA are that the specification must be deterministic (with only shallow backtracking), it must be only linear tail-recursive, without true recursion, and it can use only a limited set of built-ins and user-defined preserved words. On the other hand, the system also extends the standard Prolog built-in set, both to relax the above restrictions somewhat in a controlled way, and to support hardware-specific operations. In particular, built-ins are provided for creating and manipulating registers and bit fields with specified widths, and for managing the state those registers contain.

2.5 List Based Processing

The ISA specification is defined by a Definite Clause Grammer where Syntex defines the structure of the ISA specification and Semantics directs the translation. The DCG rules resides in the Design Program. The main transformation parses the input specification which was chosen by several constraints, simulation results and heuristics previously. The pre-defined library which will be generated automatically and the translation rules which will be generated at each level produce application specific intermediate list form.

The List representation has an advantage for firing rules not only as a favorite structure of DCG but also as a

simplified expression of Prolog which uses many built-ins for managing optimized control and data sturcture.

2.6 Data and Control Path

The output of the high level synthesis is the data and control path for the pipeline synthesis input and the logic level synthesis. It is a level we can see "What the design looks like" in the Figure 2.3.

```
% Control path

:- dynamic state/3.
state(bc(1),[move(pc,bus(unassigned),memAR),enable(pc,inc)],bc(2)).
state(bc(2),[enable(mem,mem_read)],bc(3)).
state(bc(3),[],switch(field(memDR,opcode),[case(store,bc(21)),case(shr,bc(19)),case(load,bc(17)),case(jmp,bc(15)),case(halt,bc(13))
,case(brn,bc(9)),case(and,bc(7)),case(add,bc(5))])).
state(bc(5),[move(field(memDR,2),bus(unassigned),memAR)],bc(6)).
state(bc(6),[enable(mem,mem_read)],bc(24)).
state(bc(7),[move(field(memDR,2),bus(unassigned),memAR)],bc(8)).
state(bc(8),[enable(mem,mem_read)],bc(25)).
state(bc(9),[],switch(ac<constant(0),[case(false,bc(1)),case(true,bc(11))])).
state(bc(11),[move(field(memDR,2),bus(unassigned),pc)],bc(1)).
state(bc(13),[],stop).
state(bc(15),[move(field(memDR,2),bus(unassigned),pc)],bc(1)).
state(bc(17),[move(field(memDR,2),bus(unassigned),memAR)],bc(18)).
state(bc(18),[enable(mem,mem_read)],bc(27)).
state(bc(19),[move(ac,bus(unassigned),shfone(shf1)),move(shfone(shf1),bus(unassigned),ac),enable(shfone(shf1),shr1)],bc(1)).
state(bc(21),[move(field(memDR,2),bus(unassigned),memAR),move(ac,bus(unassigned),memDR)],bc(22)).
state(bc(22),[enable(mem,mem_write)],bc(1)).
state(bc(24),[move(ac,bus(unassigned),port(alu(alu3),1)),move(memDR,bus(unassigned),port(alu(alu3),2)),move(alu(alu3),bus(unas
signed),ac),enable(alu(alu3),add)],bc(1)).
state(bc(25),[move(ac,bus(unassigned),port(alu(alu5),1)),move(memDR,bus(unassigned),port(alu(alu5),2)),move(alu(alu5),bus(unas
signed),ac),enable(alu(alu5),and)],bc(1)).
state(bc(27),[move(memDR,bus(unassigned),ac)],bc(1)).


% Data path

:- dynamic element/5, stateRegister/2, stateField/3.

element(reg,ac,[],[],[]).
element(reg,pc,[],[],[]).
element(reg,memAR,[],[],[]).
element(reg,memDR,[],[],[]).

stateRegister(ac,16).
stateRegister(pc,16).
stateRegister(memAR,16).
stateRegister(memDR,16).

stateField(memDR,opcode,15-13).
stateField(memDR,2,12-0).
```

2.7 Parsing

One way to represent a behavioral characteristics of a microprocessor is the linear recursive Instruction Set Architecture which has one tail recursion main. This parser has two levels of translation. First, read-in a standard Pro-

log ISA and build a list data structure. Second, check linear recurrence, apply translation rule and translate into acceptble spec with the rejection.

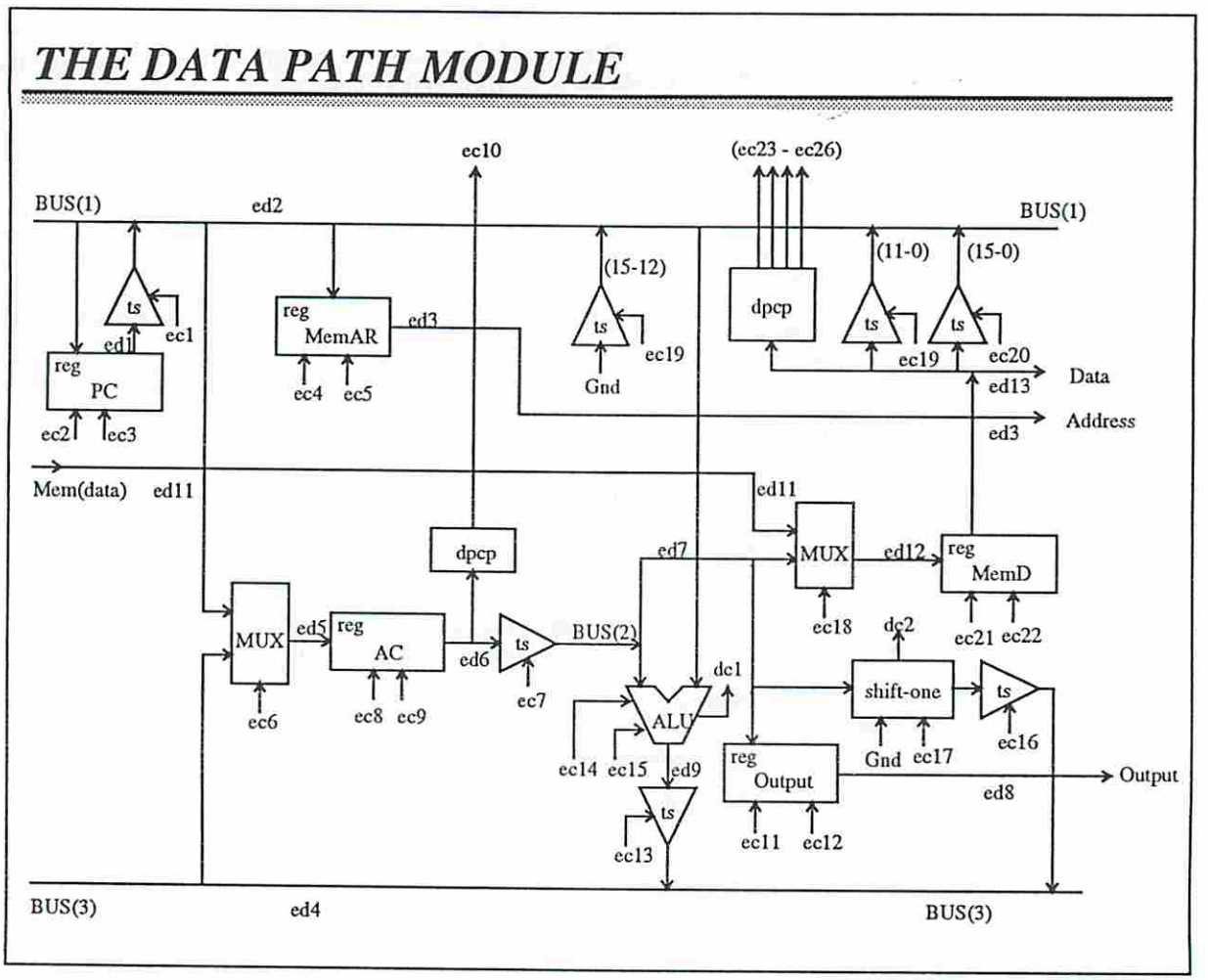


Figure 2.3 The Data Path Module

```
% <ISA_SPEC> --> <NR_CLAUSES>, <RC_CLAUSE>, <NR_CLAUSES>.
isa_spec(SPEC) --> nr_clauses(Nr1), rc_clause(Rc), nr_clauses(Nr2), {clean(Nr1,Nr3), clean(Nr2,Nr4), append(Nr3,[Rc],SPEC0),
append(SPEC0,Nr4,SPEC)}.

% <RC_CLAUSES> --> <HEAD>, [':-'], <RC_BODYS>.
rc_clause([Head, (:-) | Body]) --> head(Head), {main(Head)}, [':'], ['-'], rc_bodys(Head,Body1), {clean(Body1,Body)}.

% <NR_CLUASES> --> <NR_CLAUSE>, <NR_CLAUSES> | [].
nr_clauses([Nr|Nrs]) --> nr_clause(Nr), nr_clauses(Nrs) | [].

% <NR_CLAUSE> --> <HEAD>, [':-'], <NR_BODYS> | <HEAD>, ['.'].
nr_clause([Head, (:-) | Body]) --> head(Head), [':'], ['-'], nr_bodys(Head,Body1), {clean(Body1,Body)}.
nr_clause(Head) --> head(Head), ['.'].

% <HEAD> --> <STRUCTURE> | <ATOM>.
head(Head_list) --> structure(Head_list) | atom(Head_list).
```

```
% <RC_BODYS> --> <RC_BODY>, [','], <RC_BODYS> | <LRC_BODY>, ['.'].
rc_bodys(Head,[Body|Bodys]) --> rc_body(Body), [','], rc_bodys(Head,Bodys) | lrc_body(Head,Bodys), ['.'].

% <NR_BODYS> --> <NR_BODY>, [','], <NR_BODYS> | <NR_BODY>, ['.'].
nr_bodys(Head,[Body|Bodys]) --> nr_body(Head,Body), [','], nr_bodys(Head,Bodys) | nr_body(Head,Body), ['.'].

% <RC_BODY> --> <STRUCTURE> | <ARITH_OP>.
rc_body(Body) --> structure(Body) | arith_op(Body).

% <LRC_BODY> --> <STRUCTURE>, {is_first_recurrence}.
lrc_body(Head,Lrc_body) --> structure(Body), {is_first_recurrence(Head,Body,Lrc_body)}.

% <NR_BODY> --> <STRUCTURE>, {is_non_recursive} | <ARITH_OP> | <ATOM>.
nr_body(Head,Body) --> structure(Body), {is_non_recursive(Head,Body)} | arith_op(Body) | atom(Body1), {Body = [Body1]}.

% <STRUCTURE> --> [Functor], {is_atom}, ['('], <ARGS>, [')'].
structure(S) --> [Functor], {atom(Functor)}, ['('], args(Args), [')'], {clean(Args,A), append([Functor],A,S)}.

% <ARGS> --> <ARG_LIST>, [','], <ARGS> | <ARG_LIST>.
args([A|As]) --> arg_list(A), [','], args(As) | arg_list(A).

% <ARG_LIST> --> <ATOM> | ['['], <ARGS>, [']'] | <STRUCTURE>.
arg_list(A) --> atom(A) | ['['], args(A1), [']'], {clean(A1,A)} | structure(A).

% <ATOMS> --> <ATOM>, [','], <ATOMS> | <ATOM>.
atoms([A|As]) --> atom(A), [','], atoms(As) | atom(A).

% <VAR> --> [VAR], {is_var}.
var(Var) --> [Var], {var(Var)}.

% <ATOM> --> [ATOM], {is_atom}.
atom(Atom) --> [Atom], {atomic(Atom)}.

% <ARITH_OP> --> [NDATA], [is], [DATA1], <INFIX>, [DATA2].
arith_op([I,D1,D2,Ndata]) --> [Ndata], [is], [D1], infix(I), [D2].

% <ARITH_OP> --> [NDATA], [is], [DATA1], <INFIX>, <INFIX>, [DATA2].
arith_op([lor,D1,D2,Ndata]) --> [Ndata], [is], [D1], ['\'], ['/'], [D2].

% <ARITH_OP> --> [NDATA], [is], [DATA1], <INFIX>, <INFIX>, <STRUCTURE>.
arith_op([shr,D1,D2,Ndata]) --> [Ndata], [is], [D1], ['>'], ['>'], structure(D2).

% <ARITH_OP> --> [NDATA], [is], [DATA1], <INFIX>, <INFIX>, <STRUCTURE>.
arith_op([shl,D1,D2,Ndata]) --> [Ndata], [is], [D1], ['<'], ['<'], structure(D2).

% <ARITH_OP> --> [NDATA], [is], [DATA1], <INFIX>, <INFIX>, [DATA2].
arith_op([and,D1,D2,Ndata]) --> [Ndata], [is], [D1], ['/'], ['\'], [D2].

% <ARITH_OP> --> [NDATA], [is], [DATA1], <INFIX>, <INFIX>, [DATA2].
arith_op([I,D1,D2,Ndata]) --> [Ndata], [is], [D1], infix(I), infix(I), [D2].

% <ARITH_OP> --> [NDATA], [is], [DATA1], ['-'], [DATA2], ['-'], [DATA3].
arith_op([subc,D1,D2,D3,Ndata]) --> [Ndata], [is], [D1], ['-'], [D2], ['-'], [D3].

% <ARITH_OP> --> [NDATA], [is], [DATA1], <INFIX>, [DATA2], <INFIX>, [DATA3].
arith_op([addc,D1,D2,D3,Ndata]) --> [Ndata], [is], [D1], infix(I), [D2], infix(I), [D3].

% <ARITH_OP> --> [NDATA], [is], ['('], [DATA1], ['^'], [DATA2], [')'], ['\/'], ['('], [DATA1], ['^'], ['\'], [DATA2], [')'].
arith_op([xor,AC,D,Ndata]) --> [Ndata], [is], ['('], ['\'], [AC], ['/'], ['\'], [D], [')'], ['\'], ['/'], ['('], [AC], ['/'], ['\'], ['\'], [D],
[')'].

% <ARITH_OP> --> [DATA1], <INFIX>, [DATA2].
arith_op([I,D1,D2]) --> [D1], infix(I), [D2].
```

```
% <ARITH_OP> --> ['['], [DATA1], [','], [DATA2], [']'], <INFIX>, [DATA3].
arith_op([I,[D1,D2],D3]) --> ['['], [D1], [','], [D2], [']'], infix(I), [D3].

% <ARITH_OP> --> [DATA1], <INFIX>, [DATA2] ['->'], [DATA3], <INFIX>, [DATA4], <DIVIDER>, [DATA5], <INFIX>,
 [DATA6].
arith_op([I3,[->,[I1,D1,D2],[I2,D3,D4]],[I2,D5,D6]]) -->
[D1], infix(I1), [D2], ['-'], ['>'], [D3], infix(I2), [D4], divider(I3), [D5], infix(I2), [D6].

% <ARITH_OP> --> [DATA1], <INFIX>, [DATA2] ['->'], <ARITH_OP>, <DIVIDER>, <ARITH_OP>.
arith_op([I3,[->,[I1,D1,D2],I2],I4]) --> [D1], infix(I1), [D2], ['-'], ['>'], arith_op(I2), divider(I3), arith_op(I4).

% <ARITH_OP> --> [DATA1], <INFIX>, [DATA2] ['->'], <STRUCTURE>, <DIVIDER>, <STRUCTURE>.
arith_op([I3,[->,[I1,D1,D2],I2],I4]) --> structure(D1), infix(I1), [D2], ['-'], ['>'], structure(I2), divider(I3), structure(I4).

% <ARITH_OP> --> [DATA1], <INFIX>, [DATA2] ['->'], <STRUCTURE>, <DIVIDER>, <STRUCTURE>.
arith_op([I3,[->,[I1,D1,D2],I2],I4]) --> [D1], infix(I1), [D2], ['-'], ['>'], structure(I2), divider(I3), structure(I4).

% <ARITH_OP> --> <STRUCTURE>, ['\=='], [DATA2], ['->'], <ARITH_OP>, [','], <ARITH_OP>, <DIVIDER>,
<ARITH_OP>, [','], <ARITH_OP>.
arith_op([I3,[->,[noteq,D1,D2],I4,I5],I6,I7]) --> structure(D1), ['\'], ['='], ['='], [D2], ['-'], ['>'], arith_op(I4), [','], arith_op(I5),
 divider(I3), arith_op(I6), [','], arith_op(I7).

% <INFIX> --> ['*'] | ['+'] | ['^'] | ['>'] | ['='] | ['<'].
infix(I) --> ['+'], {I = add}.
infix(I) --> ['-'], {I = sub}.
infix(I) --> ['*'], {I = mul}.
infix(I) --> ['/'], {I = div}.
infix(I) --> ['>'], {I = shr}.
infix(I) --> ['='], {I = equal}.
infix(I) --> ['<'], {I = less}.

% <DIVIDER> --> [';'].
divider(D) --> [';'], {D = ';'}.

% Linear Recurrence Form Check
is_first_recurrence(Head_list,Body_list,List) :-
        Head_list = [Name1|Hs], Body_list = [Name2|Bs],
        length(Head_list,N1), length(Body_list,N2),
        (Name1 == Name2 ->
         (N1 == N2 ->
        (value(Num),
        (Num == 0 ->
         bind(Hs,Bs,List),
         asserta(value(1));
         List = [[wrong]],
         write('% Reject ISA: Another recurrsion'), nl,
         List = [[wrong]],
         asserta(value(-1))));
         write('% Reject ISA: Wrong arity'), nl,
        List = [[wrong]],
        asserta(value(-1)));
        (is_second -> List = [Body_list]; List = [[wrong]])).

is_non_recursive(Head_list,Body_list) :-
        Head_list = [Name1|_],
        Body_list = [Name2|_],
        Name1 \== Name2.

% Bind arguments of recursion
bind([A],[B],L) :- L = [[=,A,B]].
bind([A|As],[B|Bs],[L|Ls]) :- L = [=,A,B], bind(As,Bs,Ls).

check_isa :-
        (is_first ->
         nl, write('% Reject ISA: Not a recursion call at all');
```

```
        value(Flag),
        (Flag == 1 ->
        nl, write('% Accept ISA: the first recurrence form');
        nl, write('% Reject ISA: Not a first recurrence form'))).

% Value check of first recurrence form
is_second :- value(N), !, N \== 0.
is_first :- value(N), !, N == 0.
```

2.8 Global Control Graph

We can assume the Finite State Machine consists of four main cycles - Instruction Fecth, Instruciton Decode, Operand Fetch and Execute. These cycles can be imbeded inside during the design phase. For example, the argument matching when an instruction execution in Prolog is a decoding phase in FSM. So, at one time, we have to extract them and represent as a global flow.

```
% <GLOB_SPEC> --> <GLOB_GOLAS>, <GLOB_SPEC> | [].
glob_spec([C|Cs],[R|Rs]) --> glob_goals(C,R), glob_spec(Cs,Rs) | [].

% <GLOB_GOALS> --> {is_main_goal}, [_], [':'], <DEPTH_GOAL>, ['.'], <DUM_GLS>.
glob_goals(Cs,CR) --> {main_goal(M)}, [[M|_]], [':'], depth_goal(C,R), {clean(C,Cs), clean(R,CR)}, ['.'], dum_gls.

% <DEPTH_GOAL> --> <DEPTH_BODY>, <DEPTH_GOAL> | [].
depth_goal([Cycle|Cycles],[R|Rs]) --> depth_body(Cycle,R), depth_goal(Cycles,Rs) | [].

% <DEPTH_BODY> --> [CYCLES], {is_assign_functor}.
depth_body([],[]) --> [Cycles], {Cycles=[=|_]}.
% <DEPTH_BODY> --> [CYCLES], {is_not_multi_cycle}.
depth_body(Cycle,[]) --> [Cycles], {Cycles=[Hc|_], is_multi_cycle(Hc, false), trans_rule(Cycles,Cycle)}.
% <DEPTH_BODY> --> [CYCLES], {is_multi_cycle}, <BFSS>.
depth_body(CCycles,CR) --> [Cycles], {Cycles=[Hc|Tc], is_multi_cycle(Hc, true), lst(Depth), bfss(Cyc, Hc, R, Tc, Depth, []),
clean(Cyc, CCycle), clean(R,CR),(CCycle=[CCycles|[]]; CCycles=CCycle)}.

% <BFSS> --> <BFS>, <BFSS> | [].
bfss([Bs|Bss], Hc, [R|Rc], Tc) --> bfs(Bs, Hc, R, Tc), bfss(Bss, Hc, Rc, Tc) | [].

% <BFS> --> {is_interrupt_goal}, [':'], <BFS_BODYS>, ['.'].
bfs(Bs, interrupt, FR, Tc) --> [[interrupt|[OP|Tc1]]], [':'], bfs_bodys(B,S), ['.'], {clean(B,CB), Bs = [[case,int,OP]|CB],
 syn_check_glob(R,Tc,[[OP]|Tc1]), clean(R,CR), clean(S,CS), append(CR,CS,FR)}.

% <BFS> --> {is_operand_fetch_goal}, [':'], <BFS_BODYS>, ['.'].
bfs(Bs, fetcho, FR, Tc) --> [[fetcho|[AddMode|Tc1]]], [':'], bfs_bodys(B,S), ['.'], {clean(B,CB), Bs = [[case,addmode,AddMode]|CB],
 syn_check_glob(R,Tc,[[AddMode]|Tc1]), clean(R,CR), clean(S,CS), append(CR,CS,FR)}.

% <BFS> --> {is_pc_increment_goal}, [':'], <BFS_BODYS>, ['.'].
bfs(Bs, incrementpc, FR, Tc) --> [[incrementpc|[AddMode|Tc1]]], [':'], bfs_bodys(B,S), ['.'], {clean(B,CB),
Bs = [[case,addmode,AddMode]|CB], syn_check_glob(R,Tc,[[AddMode]|Tc1]), clean(R,CR), clean(S,CS), append(CR,CS,FR)}.

% <BFS> --> {is_execute_goal}, [':'], <BFS_BODYS>, ['.'].
bfs(Bs, execute, FR, Tc) --> [[execute|[[OP]|Tc1]]], [':'], bfs_bodys(B,S), ['.'], {clean(B,CB), Bs = [[case,cr,OP]|CB],
syn_check_glob(R,Tc,[[OP]|Tc1]), clean(R,CR), clean(S,CS), append(CR,CS,FR)}.

% <BFS> --> {is_same_functor_goal}, [':'], <BFS_BODYS>, ['.'].
bfs(Bs, Hc, FR, Tc) --> [[Hc|Tc1]], [':'], bfs_bodys(B,S), ['.'], {clean(B,Bs), syn_check_glob(R,Tc,Tc1), clean(R,CR), clean(S,CS),
 append(CR,CS,FR)}.

% <BFS> --> {is_same_functor_fact}, ['.'].
bfs([Hc|Hcs], Hc, [], _) --> [[Hc|Hcs]], ['.'].
```

```
% <BFS> --> {is_dum_functor_goal}, [':'], <DUM_BODYS>, ['.'].
bfs([], _, [], _) --> [_], [':'], dum_bodys, ['.'].

% <BFS> --> {is_dum_functor_fact}, ['.'].
bfs([], _, [], _) --> [_], ['.'].

% <BFS_BODYS> --> <DEPTH_BODY>, <BFS_BODYS> | [].
bfs_bodys([C|Cs],[R|Rs]) --> depth_body(C,R), bfs_bodys(Cs,Rs) | [].

% <DUM_GLS> --> <DUM_GL>, <DUM_GLS> | [].
dum_gls --> dum_gl, dum_gls | [].

% <DUM_GL> --> [_], [':'], <DUM_BODYS>, ['.'] | [_], ['.'].
dum_gl --> [_], [':'], dum_bodys, ['.'].
dum_gl --> [_], ['.'].

% <DUM_BODYS> --> [_] | [_], <DUM_BODYS>.
dum_bodys --> [_].
dum_bodys --> [_], dum_bodys.

% Check multi cycle
is_multi_cycle(Hc, Bool) :- possible_multi_cycle(PMC), check_multi(Bool, Hc, PMC).
check_multi(false, _, []).
check_multi(Bool, Hc, [PMCH|PMCT]) :-
Hc == PMCH -> Bool = true; check_multi(Bool, Hc, PMCT).

syn_check_glob(_,[],[]).
syn_check_glob([R|Rs],[H|T],[H1|T1]) :-
H = H1, R = [], syn_check_glob(Rs,T,T1).
syn_check_glob([R|Rs],[H|T],[H1|T1]) :-
H \== H1, R = syn(H1,H), syn_check_glob(Rs,T,T1).
```

2.9 Transition Rules

This is the main issue of translation of the high level synthesis. The experimental and efficient rule set is a primary goal of this level. The basic micro processor rules contain a general, flexible and default translation guidance. The application specific design rule set can contain an ad hoc translation. Each translation subset can act as a major terminator which is an optimized design choice. However, the determination of "Which rule should be fired?" depends on the simulation and estimation result. The order of rule set is important.

```
% TDY version
trans_rule([memory_read,data,_,_],[[move,memDR,memAR],[mem_read,memAR,memDR1]]).
trans_rule([memory_read,data1,_,_],[[move,memDR,memAR1],[mem_read,memAR1,memDR1]]).
trans_rule([memory_read,x1,_,_],[[move,adrbuf2,memAR],[mem_read,memAR,memDR]]).
trans_rule([equal,AC_N,data1],[[move,memDR,AC]]) :- replace_goal(AC,AC_N).
trans_rule([add,PC,adrbuf,data],[[arith,+,PC,adrbuf,memDR]]).
trans_rule([add,D1,data1,D3],[[arith,+,D1,memDR,D4]]) :- replace_goal(D4,D3).
trans_rule([add,D1,adrbuf,x1],[[arith,+,D1,adrbuf,adrbuf1]]).
trans_rule([add,D1,D2,x1],[[arith,+,D1,D2,adrbuf]]).
trans_rule([addc,AC,_,_,R],[[arith,+,AC,memDR,c_in,R1]]) :- replace_goal(R1,R).
trans_rule([subc,AC,_,_,R],[[arith,-,AC,memDR,c_in,R1]]) :- replace_goal(R1,R).
trans_rule([sub,adrbuf,C,data],[[arith,-,adrbuf,C,memDR]]).
trans_rule([sub,adrbuf,C,x1],[[arith,-,adrbuf,C,adrbuf1]]).
trans_rule([sub,D1,data1,D3],[[arith,-,D1,memDR,D4]]) :- replace_goal(D4,D3).
trans_rule([sub,data1,D1,D3],[[arith,-,memDR,D1,D4]]) :- replace_goal(D4,D3).
trans_rule([sub,data,D1,D3],[[arith,-,memDR,D1,D4]]) :- replace_goal(D4,D3).
trans_rule([and,D1,data1,D3],[[arith,/\,D1,memDR,D4]]) :- replace_goal(D4,D3).
trans_rule([lor,D1,_,D3],[[arith,\/,D1,memDR,D4]]) :- replace_goal(D4,D3).
trans_rule([xor,D1,_,D3],[[arith,xor,D1,memDR,D4]]) :- replace_goal(D4,D3).
```

```
trans_rule([mul,D1,_,D3],[[arith,*,D1,memDR,D3]]).
trans_rule([div,D1,_,D3],[[arith,/,D1,memDR,D3]]).
trans_rule([memory_write,data2,Data,_,_],[[move,memDR1,memAR2],[move,Data,memDR2],[mem_write,memAR2,memDR3]]).
trans_rule([memory_write,data1,Data,_,_],[[move,memDR,memAR1],[move,Data,memDR1],[mem_write,memAR1,memDR2]]).
trans_rule([memory_write,data,Data,_,_],[[move,memDR,memAR1],[move,Data,memDR1],[mem_write,memAR1,memDR2]]).
trans_rule([memory_write,x1,Data,_,_],[[move,adrbuf1,memAR],[move,Data,memDR],[mem_write,memAR,memDR]]).
trans_rule([insert_status_reg,SR,D,cflag,_],[[move,D^carry,SR^0]]) :- assert((field(cflag,SR^0))).
trans_rule([extract,SR,cflag,_],[[move,SR^0,c_in]]) :- assert((field(cflag,SR^0))).

% SMP case
trans_rule([;,[->,[less,AC,data],[add,PC,1,_]],[equal,PCN,PC]],[[arith,AC<memDR],[case,AC<memDR,true],[inc,PC]]) :-
 replace_goal(PC,PCN).
trans_rule([;,[->,[less,AC,data1],[add,PC,1,_]],[equal,PCN,PC]],[[arith,AC<memDR],[case,AC<memDR,true],[inc,PC]]) :-
 replace_goal(PC,PCN).
trans_rule([;,[->,[equal,AC,C],[add,PC,D,PCN]],[equal,P,]],[[arth,A=nstant(C)],[case,AC=constant(C),true],[arith,+,PC,D,PC]]).
trans_rule([;,[->,[less,AC,C],[add,PC,D,PCN]],[equal,PCN,]],[[ari,<onstant(C)],[case,AC<constant(C),true],[arith,+,PC,D,PC]]).
trans_rule([;,[->,[equal,AC,],[equal,PCN,daa]],[equal,,]],[[arth,AC=onstant(C)],[case,AC=constant(C),true],[move,memDR,PC]])
 :- replace_goal(PC,PCN).
trans_rule([;,[->,[equal,AC,],[equal,PCN,daa1]],[equal,,]],[[arth,A=onstant(C)],[case,AC=constant(C),true],[move,memDR,PC]])
 :- replace_goal(PC,PCN).
trans_rule([;,[->,[less,AC,C],[equal,PCN,data]],[equal,,]],[[arith,<constant(C)],[case,AC<constant(C),true],[move,memDR,PC]])
 :- replace_goal(PC,PCN).
trans_rule([;,[->,[less,AC,C],[equal,PCN,data1]],[equal,,]],[[arith,<constant(C)],[case,AC<constant(C),true],[move,memDR,PC]])
 :- replace_goal(PC,PCN).

% Translation from Abstract to Concrete
trans_rule([memory_read,pc,_,_],[[move,pc,memAR],[mem_read,memAR,memDR],[move,memDR^1,cr]]).
trans_rule([memory_read,Address,_,_],[[move,Address,memAR],[mem_read,memAR,memDR]]).
trans_rule([memory_write,Address,Data,_,_],[[move,Address,memAR],[move,Data,memDR],[mem_write,memAR,memDR]]).
trans_rule([add,PC,1,p1],[[inc,PC]]).
trans_rule([add,D1,data,D3],[[arith,+,D1,memDR,D4]]) :- replace_goal(D4,D3).
trans_rule([add,D1,D2,D3],[[arith,+,D1,D2,D4]]) :- replace_goal(D4,D3).
trans_rule([sub,D1,data,D3],[[arith,-,D1,memDR,D4]]) :- replace_goal(D4,D3).
trans_rule([sub,D1,D2,D3],[[arith,-,D1,D2,D4]]) :- replace_goal(D4,D3).
trans_rule([and,D1,data,D3],[[arith,/\,D1,memDR,D4]]) :- replace_goal(D4,D3).
trans_rule([and,D1,D2,D3],[[arith,/\,D1,D2,D4]]) :- replace_goal(D4,D3).
trans_rule([shr,D1,[field,lowsix_bit,x1],D3],[[arith,>>,D1,adrbuf1^2^2,D4]]) :- replace_goal(D4,D3), assert((field(adrbuf^2^2,5
-0))).
trans_rule([shl,D1,[field,lowsix_bit,x1],D3],[[arith,<<,D1,adrbuf1^2^2,D4]]) :- replace_goal(D4,D3), assert((field(adrbuf^2^2,5
-0))).
trans_rule([shr,D1,[field,lowsix_bit,D2],D3],[[arith,>>,D1,D2^2^2,D4]]) :- replace_goal(D4,D3), assert((field(D2^2^2,5-0))).
trans_rule([shl,D1,[field,lowsix_bit,D2],D3],[[arith,<<,D1,D2^2^2,D4]]) :- replace_goal(D4,D3), assert((field(D2^2^2,5-0))).
trans_rule([equal,R1,data],[[move,memDR,R2]]) :- replace_goal(R2,R1).
trans_rule([equal,R1,D1],[[move,D1,R2]]) :- replace_goal(R2,R1).
trans_rule([equal,[R1,_],D1],[[move,D1^1,R1]]).
trans_rule([;,[->,[less,D1,C],[equal,D2,D3]],[equal,pc_next,]],[[arith,1<constant(C)],[case,D1<constant(C),true],[move,D3,D4]])
 :- replace_goal(D4,D2).
trans_rule([;,[->,[equal,D1,C],[equal,D2,D3]],[equal,pc_next,]],[[arth,1=onstant(C)],[case,D1=constant(C),true],[move,D3,D4]])
 :- replace_goal(D4,D2).
trans_rule([halt],[[stop,M]]) :- main_goal(M).
trans_rule([nop],[[nop]]).

% Translation from Abstract to Concrete (Register File Version)
trans_rule([register_read,Address,RF,Data],[[register_read,Address,RF,Data]]).
trans_rule([register_write,Address,RF,_Data],[[register_write,Address,RF,Data]]).
trans_rule([add,D1,D2,D3],[[arith,+,D1,D2,D3]]).
trans_rule([sub,D1,D2,D3],[[arith,-,D1,D2,D3]]).
trans_rule([and,D1,D2,D3],[[arith,/\,D1,D2,D3]]).
trans_rule([shr,D1,D2,D3],[[arith,>>,D1,D2,D3]]).

% Translation from Abstract to Concrete (TDY Version)
trans_rule([;,[>,[less,D1,C],[construct,[[field,highbyte,D2],[field,lowbyte,D1]],AB]],[construct,[[field,highbyte,_],[field,lowbyte,D1
]],AB]],[[arith,D1<constant(C)],[case,D1<constant(C),true],[move,constant(D2),AB^1],[move,D1,AB^2],[case,D1<constant(C),fal
se],[move,constant(127),AB^1],[move,D1,AB^2]]) :- assert((field(AB^2,7-0))), assert((field(AB^1,15-8))).
```

```
trans_rule([;,[>,[less,[field,lowbyte,BR],C],[construct,[[field,highbyte,0],[field,lowbyte,BR]],D]],[construct,[[field,highbyte,1],[field
,lowbyte,br]],D]],[[arith,BR^2<constant(C)],[case,BR^2<constant(C),true],[move,constant(0),D^1],[move,BR^2,D^2],[case,BR^2<
constant(C),false],[move,constant(1),D^1],[move,BR^2,D^2]]) :- assert((field(D^2,7-0))), assert((field(D^1,15-8))), assert((
field(BR^2,7-0))), assert((field(BR^1,15-8))).

% for TXI and TXP
trans_rule([;,[>,[note,[iel,ighye,XR],C],[equal,pc_ex,],[,XR,1,XRN]],[equal,pc_next,pc],[equal,XRN,XR]],[[arith,XR^1\==constant
(C)],[case,XR^1\==constant(C),true],[move,memDR,pc],[arith,-,XR,1,XR]]) :- assert((field(XR^2,7-0))), assert((field(XR^1,15-8))).
trans_rule([;,[>,[note,[iel,ighye,XR],C],[equal,pc_ex,x1],[,XR,1,XRN]],[equal,pc_next,pc],[equal,XRN,XR]],[[arith,XR^1\==const
ant(C)],[case,XR^1\==constant(C),true],[move,adrbuf1,pc],[arith,-,XR,1,XR]]) :- assert((field(XR^2,7-0))), assert((field(XR^1,15
-8))).

trans_rule([construct,[[field,highbyte,0],[field,lowbyte,br]],D],[[move,constant(0),D^1],[move,br^2,D^2]]) :- assert((field(D^2,7-0))),
 assert((field(D^1,15-8))).
trans_rule([construct,[[field,highbyte,0],[field,lowbyte,N2]],D],[[move,constant(0),D^1],[move,N2,D^2]]) :- assert((field(D^2,7-0))),
 assert((field(D^1,15-8))).
trans_rule([construct,[[field,highbyte,N1],[field,lowbyte,br]],D],[[move,N1^1,D^1],[move,br^2,D^2]]) :- assert((field(D^2,7-0))),
 assert((field(D^1,15-8))).
trans_rule([construct,[[field,highbyte,1],[field,lowbyte,N2]],D],[[move,constant(127),D^1],[move,N2,D^2]]) :- assert((field(D^2,7
-0))), assert((field(D^1,15-8))).
trans_rule([construct,[[field,highbyte,N1],[field,lowbyte,N2]],D],[[move,N1^1,D^1],[move,N2,D^2]]) :- assert((field(D^2,7-0))),
 assert((field(D^1,15-8))).
trans_rule([construct,[[field,highword,AC],[field,lowword,B]],D],[[move,AC,D^1],[move,B,D^2]]) :- assert((field(D^2,15-0))),
 assert((field(D^1,31-16))).
trans_rule([construct,MD,highword,D1],[[move,MD^1,D2]]) :- replace_goal(D2,D1), assert((field(MD^1,15-0))).
trans_rule([construct,MD,lowword,D1],[[move,MD^2,D2]]) :- replace_goal(D2,D1), assert((field(MD^2,31-16))).


% Translation from Abstract to Concrete (6502 Version)
trans_rule([add,PC,1,NPC],[[arith,+,PC,1,NPC]]).
trans_rule([addc,AC,D,C,R],[[arith,+,AC,D,C,R1]]) :- replace_goal(R1,R).
trans_rule([subc,AC,D,C,R],[[arith,-,AC,D,C,R1]]) :- replace_goal(R1,R).
trans_rule([equal,D1,D2],[[move,D2,D1]]).
trans_rule([insert_status_reg,SR,iflag,D,_],[[move,D,SR^2]]) :- assert((field(SR^2,2))).
trans_rule([extract_status_reg,SR,cflag,D],[[move,SR^0,D]]) :- assert((field(SR^0,0))).
trans_rule([test_status_reg,SR,iflag,D],[[move,D,cpr^1],[move,SR^2,cpr^2]]).
trans_rule([extract,PC,lowpc,D],[[move,PC^2,D]]) :- assert((field(PC^2,7-0))).
trans_rule([extract,PC,highpc,D],[[move,PC^1,D]]) :- assert((field(PC^1,15-8))).
trans_rule([construct,[[field,8,N],[field,lowbyte,SP]],D],[[move,N,D^1],[move,SP,D^2]]) :- assert((field(D^2,7-0))), assert((
field(D^1,15-8))).
trans_rule([construct,[[field,highbyte,N1],[field,lowbyte,N2]],D],[[move,N1,D^1],[move,N2,D^2]]) :- assert((field(D^2,7-0))),
 assert((field(D^1,15-8))).
trans_rule([check_flag,all,SR,_],[[check_flag,all,SR]]).

% Default Case
trans_rule(C,C).
```

2.10 Data and Control Dependency Analysis

A control dependency primarily compiles with the major branching sequences. In other words, the global branching nodes are separated by the control flowes. A data dependency acts on a principle of the out-in data flow and out-out. In/Out flatten data list makes a draft for the analysis.

```
% <DEP_SPECS> --> <DEP_SPEC>, <DEP_SPECS> | [].
dep_specs([F\Fs]) --> dep_spec(F), dep_specs(Fs) | [].

% <DEP_SPEC> --> [branch_node(BN)].
dep_spec(CF) --> [branch_node(BN)], {dep_io_lists(F,BN), clean(F,CF)}.
```

```
dep_io_lists(_,[]).
dep_io_lists([CFlat\CFlats],[H\T]) :-
H = [[case,D,B]\Comb],
flat_list(FComb,[],Comb,[]),
make_f_list(Flat,[case,D,B],FComb,[]), clean(Flat,CFlat), !,
dep_io_lists(CFlats,T).
dep_io_lists([L],[L]) :-
L = [H\_], is_list(H).
dep_io_lists(L,L).

% TDY case
make_f_list(_,_,[],[]).
make_f_list([[F,[arith,MDR1<C]],[F,[case,MDR1<C,TR], H5],[F,[case,MDR1<C,TR], H6],[F,[case,MDR1<C,FA], H7],[F,[case,M
DR1<C,FA], H8]\F6],F) --> [[arith,MDR<C]], [[case,MDR<C,TR], H1, H2], [[case,MDR<C,FA], H3, H4], {check_syn(H1,H5),
check_syn(H2,H6), check_syn(H3,H7), check_syn(H4,H8), check_syn(MDR,MDR1), !}, make_f_list(F6,F).
make_f_list([[F,[arith,AC\==C]],[F,[case,AC\==C,TR], H1],[F,[case,AC\==C,TR], H2]\F4],F) --> [[arith,AC\==C],[case,
AC\==C,TR], H1, H2], {!}, make_f_list(F4,F).
make_f_list([[F,[arith,AC=C]],[F,[case,AC=C,TR], H2]\F3],F) --> [[arith,AC=C]], [[case,AC=C,TR], H1], {check_syn(H1,H2), !},
make_f_list(F3,F).
make_f_list([[F,[arith,AC<C]],[F,[case,AC<C,TR], H2]\F3],F) --> [[arith,AC<C]], [[case,AC<C,TR], H1], {check_syn(H1,H2), !},
make_f_list(F3,F).

% <MAKE_F_LIST> --> [[_,Field,tag]], [[arith,tag=V]], {check_data_object}.
make_f_list([[F,[arith,Field=V]],[F,[case,Field=V,TF],H1]],F) --> [[_,Field,tag]], [[arith,tag=V]], [[case,tag=V,TF],H],
{check_syn(H,H1)}.
make_f_list([[F,[OP1,Field1,rf,rf(Field1)]],[F,[arith,rf(Field1)^1=V]],[F,[case,rf(Field1)^1=V,TF],[OP2,rf(Field1)^2,PC1]]],F) -->
[[OP1,Field,rf,[tag,_]]], [[arith,tag=V]], [[case,tag=V,TF],[OP2,_,PC]], {check_syn(Field,Field1), check_syn(PC,PC1)}.

% <MAKE_F_LIST> --> [[case\B],H], {check_synonym} \ [H], {check_synonym}, <MAKE_F_LIST> \ [H], {check_synonym}.
make_f_list([[F,[case\B],H1]],F) --> [[case\B],H], {check_syn(H,H1)}.
make_f_list([[F,H1]\F2],F) --> [H], {check_syn(H,H1), !}, make_f_list(F2,F).
make_f_list([[F,H1]],F) --> [H], {check_syn(H,H1)}.

% <FLAT_LIST> --> [H], <FLAT_LIST> \ [H].
flat_list(I,E) --> [H], {append(E,H,I)}.
flat_list(A,E) --> [H], {append(E,H,I)}, flat_list(A,I).

% Permutation of Set Elements
permut([],[]).
permut([L\T],[E\C]) :- member(L,E), permut(T,C).

depend_out(_,_,[]).
depend_out(R1,R2,[H\T]) :-
depend_out1(IR,OR,H,[],[]), clean(IR,ICR), clean(OR,OCR), !,
depend_out(ICRs,OCRs,T), append(ICR,ICRs,R1), append(OCR,OCRs,R2).

depend_out1(_,_,[],_,_).
depend_out1([C_in\C_ins],[C_out\C_outs],[Untagged\T],Out,Tag) :-
get_tagged_in_list(Untagged,In_Tagged),
get_tagged_out_list(Untagged,Out_Tagged),
depend_in_check(In_Depended,In_Tagged,Out,Tag),
depend_out_check(Out_Depended,Out_Tagged,Out,Nout,Tag,Ntag),
clean(In_Depended,C_in),
clean(Out_Depended,C_out), !,
depend_out1(C_ins,C_outs,T,Nout,Ntag).

depend_in_check([],_,_,[],[]).
depend_in_check([D1,D2,D3,D4,D5],[[H1,H2,H3,H4,H5]\T],Out,Tag) :-
depend_in(D1,[H1\T],Out,Tag), depend_in(D2,[H2\T],Out,Tag), depend_in(D3,[H3\T],Out,Tag), depend_in(D4,[H4\T],Out,Tag),
depend_in(D5,[H5\T],Out,Tag).
depend_in_check([D1,D2,D3,D4],[[H1,H2,H3,H4]\T],Out,Tag) :-
depend_in(D1,[H1\T],Out,Tag), depend_in(D2,[H2\T],Out,Tag), depend_in(D3,[H3\T],Out,Tag), depend_in(D4,[H4\T],Out,Tag).
depend_in_check([D1,D2,D3],[[H1,H2,H3]\T],Out,Tag) :-
depend_in(D1,[H1\T],Out,Tag), depend_in(D2,[H2\T],Out,Tag), depend_in(D3,[H3\T],Out,Tag).
depend_in_check([D1,D2],[[H1,H2]\T],Out,Tag) :-
```

```
depend_in(D1,[H1\T],Out,Tag), depend_in(D2,[H2\T],Out,Tag).
depend_in_check([D1],[H\T],Out,Tag) :-
depend_in(D1,[H\T],Out,Tag).

depend_in([],_,[],_).
depend_in([L1,T],[H\T],Out,Tag) :-
check_syn(H,H1),
member(H1,Tag),
find_member(Out,H1,L),
L = [_\L1].
depend_in([],_,_,_).

depend_out_check([],[H\T],[],[[H\T]],[],[H]).
depend_out_check([[L1,T]],[H\T],Out,Nout,Tag,Tag2) :-
H\== adrbuf, % TDY case
check_out_syn(H,H1),
member(H1,Tag),
delete(Tag,H1,Tag1),
append([H],Tag1,Tag2),
find_member(Out,H1,L),
delete(Out,L,Out1),
append([[H\T]],Out1,Nout),
L = [_\L1].
depend_out_check([],[H\T],Out,Nout,Tag,Ntag) :-
append([[H\T]],Out,Nout),
append([H],Tag,Ntag).

% Get a input tagged list
get_tagged_in_list([[case,CR,C],[F,AC<memDR]],[[CR,AC<memDR,memDR],[case,CR,C],[F,AC<memDR]]).
get_tagged_in_list([[case,CR,C],[F,B]],[[CR,B],[case,CR,C],[F,B]]).
get_tagged_in_list([[case,CR,C],[F,I,O]],[[CR,I],[case,CR,C],[F,I,O]]).
get_tagged_in_list([[case,CR,C],[F,I1,I2,[T,D]]],[[CR,I1,I2],[case,CR,C],[F,I1,I2,[T,D]]]).
get_tagged_in_list([[case,CR,C],[F,I1,I2,O]],[[CR,I1,I2],[case,CR,C],[F,I1,I2,O]]).
get_tagged_in_list([[case,CR,C],[F,OP,I1,I2,O]],[[CR,I1,I2],[case,CR,C],[F,OP,I1,I2,O]]).
get_tagged_in_list([[case,CR,C],[F,OP,I1,I2,I3,O]],[[I1,I2,I3],[case,CR,C],[F,OP,I1,I2,I3,O]]).
get_tagged_in_list([[case,CR,C],[case,A,T],[F,B]],[[CR,A,B],[case,CR,C],[case,A,T],[F,B]]).
%%
get_tagged_in_list([[case,CR,C],[case,A,T],[F,I,O^S]],[[CR,A,I],[case,CR,C],[case,A,T],[F,I,O^S]]).
get_tagged_in_list([[case,CR,C],[case,A,T],[F,I,O]],[[CR,A,I],[case,CR,C],[case,A,T],[F,I,O]]).
get_tagged_in_list([[case,CR,C],[case,A,T],[F1,I1,O1],[F2,I2,O2^S1]],[[CR,A,I1,I2],[case,CR,C],[case,A,T],[F1,I1,O1],[F2,I2,O2
^S1]]).
get_tagged_in_list([[case,CR,C],[case,A,T],[F1,I1,O1],[F2,OP,I2,I3,O2]],[[CR,A,I1,I2,I3],[case,CR,C],[case,A,T],[F1,I1,O1],[F2,
OP,I2,I3,O2]]).
get_tagged_in_list([[case,CR,C],[case,A,T],[F1,OP,I1,I2,O1]],[[CR,A,I1,I2],[case,CR,C],[case,A,T],[F1,OP,I1,I2,O1]]).
get_tagged_in_list([F,OP,I1,I2,O],[[I1,I2],F,OP,I1,I2,O]).
get_tagged_in_list([F,I,O],[I,F,I,O]).
get_tagged_in_list([F,B],[B,F,B]).

% Get a output tagged list
get_tagged_out_list([[case,CR,C],[F,B]],[B,[case,CR,C],[F,B]]).
get_tagged_out_list([[case,CR,C],[F,I,O]],[O,[case,CR,C],[F,I,O]]).
get_tagged_out_list([[case,CR,C],[F,I1,I2,[T,D]]],[D,[case,CR,C],[F,I1,I2,[T,D]]]).
get_tagged_out_list([[case,CR,C],[F,I1,I2,O]],[O,[case,CR,C],[F,I1,I2,O]]).
get_tagged_out_list([[case,CR,C],[F,OP,I1,I2,O]],[O,[case,CR,C],[F,OP,I1,I2,O]]).
get_tagged_out_list([[case,CR,C],[F,OP,I1,I2,I3,O]],[O,[case,CR,C],[F,OP,I1,I2,I3,O]]).
get_tagged_out_list([[case,CR,C],[case,A,T],[F,B]],[B,[case,CR,C],[case,A,T],[F,B]]).
%%
get_tagged_out_list([[case,CR,C],[case,A,T],[F,I,O^S]],[O,[case,CR,C],[case,A,T],[F,I,O^S]]).
get_tagged_out_list([[case,CR,C],[case,A,T],[F,I,O]],[O,[case,CR,C],[case,A,T],[F,I,O]]).
get_tagged_out_list([[case,CR,C],[case,A,T],[F1,I1,O1],[F2,I2,O2^S1]],[O2,[case,CR,C],[case,A,T],[F1,I1,O1],[F2,I2,O2^S1]]).
get_tagged_out_list([[case,CR,C],[case,A,T],[F1,I1,O1],[F2,OP,I2,I3,O2]],[O2,[case,CR,C],[case,A,T],[F1,I1,O1],[F2,OP,I2,I3,O2
]]).
get_tagged_out_list([[case,CR,C],[case,A,T],[F1,OP,I1,I2,O]],[O,[case,CR,C],[case,A,T],[F1,OP,I1,I2,O]]).
get_tagged_out_list([F,OP,I1,I2,O],[O,F,OP,I1,I2,O]).
get_tagged_out_list([F,I,O],[O,F,I,O]).
```

```
get_tagged_out_list([F,B],[B,F,B]).

% Find a member with tag and Extract that element
find_member([],_,_).
find_member([[H|T]|_],H,[H|T]).
find_member([_|T],H,I) :- find_member(T,H,I).

check_syn([C,C1,C4,C5],[C,C3,C4,C7]) :- syn(C1,C2), field(C2,C3), syn(C5,C6), field(C6,C7).
check_syn([C,C1^T|C3],[C,C2^T|C3]) :- replace_goal(C2,C1).
check_syn([C,C1|C4],[C,C3|C4]) :- syn(C1,C2), field(C2,C3).
check_syn(C1,C3) :- syn(C1,C2), field(C2,C3).
check_syn(rf(memDR^A)^_=_,rf(memDR^A)).
check_syn(rf(memDR^A)^_,rf(memDR^A)^_=_).

%%%TDY case
check_syn(adrbuf,adrbuf^_).
check_syn(adrbuf2,adrbuf1).
% Shift
check_syn(adrbuf^_^_,adrbuf).
check_syn(adrbuf1^_^_,adrbuf1).

check_syn(BN^carry,B^carry) :- replace_goal(B,BN).
check_syn(b^_,b).
check_syn(md^_,md).
check_syn(md1,md1^_).
check_syn(X<C,X1<C) :- syn(X,C2), field(C2,X1).
check_syn([arith,OP,I1,I2,Out],[arith,OP,I3,I2,Out]) :- syn(I1,C2), field(C2,I3).
%%%
check_syn(sr^_,sr).
check_syn(r1^1,r1).
check_syn(r2^1,r2).
check_syn(memDR^_,memDR).
% Twice Memory Read
check_syn(memDR,memDR1).
check_syn(memDR1,memDR).

% Default
check_syn(A,A).

check_out_syn(memAR1,memAR).
check_out_syn(memAR2,memAR).
check_out_syn(memDR1,memDR).
check_out_syn(memDR2,memDR1).
check_out_syn(memDR3,memDR2).
check_out_syn(ac<memDR,memDR).
check_out_syn(md2,md1^_).
check_out_syn(XR,XR^1\==constant(0)).
check_out_syn(A,A).
```

2.11 Binding Object (Functional Unit Allocation)

One of the important issue is the binding of a data object and a physical obejct. The data object is one which can be considered as a temporary result storage. It does not affects any real hardware unit or flowes. The physical obejct is one which is a real functioinal unit or at least a data/control flow. Sometimes, we have to utilize the physical resources for the design. In the following example,

```
r = [Tag, X],
Tag == 0 -> PC_next = X; PC_next = PC
```

Register field one of r (r^1) is data object Tag. In the data/control flow analysis, we just can not translate into

[move, r^1, tag] because there is no tag in real life. At the second condition structure, the compare of tag with constant 0 means [arith, =, tag, contant(0)]. Now, we have real problem. Where does the tag come? Can we put a cycle into move? The answer is to describe data obejct with an intermediate representation. After analyzing that representation, we can represent as followes.

```
[move, r1^1, arith_port1]
[move, constant(0), arith_port2]
[enable, arith_operation]
[[case, true], [move, r1^2, pc]]
[[case, false], []]
```

Empty list is a no operation (go to fetch cycle). Again, if you don't want to use an other register (or output latch) for the result of arith_operation, you can import case structure for the intermedite display.

2.12 Scheduling

It is an instruction state assignment of the global data/control flow analysis. There are lots of scheduling algorithm for this level. However, the optimal design should consider the hardware resources (internal buses, general registers, functional units, ...). These resource constraint scheduing is a basis of the data/control path.

```
asap(L) :-
        Next = [[inc,pc],[move,pc,memAR]],
        print_out(Next,1),
        process(L,Next,[],2).


process(_,[],_,_).
process(L,Done,Pass,N) :-
        N1 is N + 1,
        find_all_next(L,Done,Next_list),
        remove_dups(Next_list, Next_list1),
        append([Done],Pass,Pass1),
        check(Next_list1,L,Pass1,Next),
        print_out(Next,N), !,
        process(L,Next,Pass1,N1).

find_all_next(_,[],[]).
find_all_next(L,[X|Xs],Y1) :-
        (member([X,_],L) -> setof(A,A^member([X,A],L),Y); Y = []),
        find_all_next(L,Xs,Ys),
        append(Y,Ys,Y1).

check([],_,_,[]).
check([X|Xs],L,Pass,[X|Ys]) :- satisfy(X,L,Pass), !, check(Xs,L,Pass,Ys).
check([_|Xs],L,Pass,Ys) :- !, check(Xs,L,Pass,Ys).

satisfy(X,L,Pass) :- setof(A,A^member([A,X],L),B), check_member(B,Pass).

check_member([],_).
check_member([X|Xs],Pass):- check_arg(X,Pass), !, check_member(Xs,Pass).
check_arg(_,[]) :- fail.
check_arg([],[[]]).
check_arg([],_).
check_arg(X,[P|Ps]) :- (member(X,P) -> true; check_arg(X,Ps)).

print_out([],_).
print_out([X|Xs],N) :-
        X = [Cond,OP1,OP2,OP3],
```

```
            is_list(Cond),
            Cond1 =.. Cond,
            ch_int(OP1,Op1), ch_int(OP2,Op2), ch_int(OP3,Op3),
            P1 =.. Op1, P2 =.. Op2, P3 =.. Op3,
            print('state([‘), print(N), print(‘],[branch(‘), print(Cond1),
            print(‘,’), print(P1), print(‘,’), print(P2), print(‘,’), print(P3),
            print(‘)’), !, print_out1(Xs).
print_out([X\Xs],N) :-
            X = [Cond,Cond1,OP2], is_list(Cond),
            Cond2 =.. Cond, Cond3 =.. Cond1, ch_int(OP2,Op2), P2 =.. Op2,
            print(‘state([‘), print(N), print(‘],[branch(‘), print(‘[‘), print(Cond2),
            print(‘,’), print(Cond3), print(‘]’), print(‘,’), print(P2), print(‘)’), !, print_out1(Xs).
print_out([X\Xs],N) :-
            X = [Cond,OP1], is_list(Cond),
            Cond1 =.. Cond, ch_int(OP1,Operation), Operation1 =.. Operation,
            print(‘state([‘), print(N), print(‘],[branch(‘), print(‘[‘), print(Cond1), print(‘]’),
            print(‘,’), print(Operation1), print(‘)’), !, print_out1(Xs).
print_out([X\Xs],N) :-
            ch_int(X,XX), X1 =.. XX,
            print(‘state([‘), print(N), print(‘],[branch(‘), print(‘[]’),
            print(‘,’), print(X1), print(‘)’), !, print_out1(Xs).

print_out1([]) :- print(‘]).’),nl,nl.
print_out1([X\Xs]) :-
            X = [Cond,OP1,OP2,OP3],
            is_list(Cond),
            Cond1 =.. Cond,
            ch_int(OP1,Op1), ch_int(OP2,Op2), ch_int(OP3,Op3),
            P1 =.. Op1, P2 =.. Op2, P3 =.. Op3,
            print(‘,branch(‘), print(Cond1), print(‘,’), print(P1), print(‘,’), print(P2), print(‘,’), print(P3),
            print(‘)’), !, print_out1(Xs).
print_out1([X\Xs]) :-
            X = [Cond,Cond1,OP2], is_list(Cond),
            Cond2 =.. Cond, Cond3 =.. Cond1,
            ch_int(OP2,Op2), P2 =.. Op2,
            print(‘,branch(‘), print(‘[‘), print(Cond2), print(‘,’), print(Cond3), print(‘]’),
            print(‘,’), print(P2), print(‘)’), !, print_out1(Xs).
print_out1([X\Xs]) :-
            X = [Cond,OP1],
            is_list(Cond), Cond1 =.. Cond,
            ch_int(OP1,Operation), Operation1 =.. Operation,
            print(‘,branch(‘), print(‘[‘), print(Cond1), print(‘]’), print(‘,’), print(Operation1),
            print(‘)’), !, print_out1(Xs).
print_out1([X\Xs]) :-
            ch_int(X,XX),
            X1 =.. XX,
            print(‘,branch(‘), print(‘[]’), print(‘,’), print(X1), print(‘)’), !, print_out1(Xs).

ch_int([],[]).
ch_int([H\T],[H1\T1]) :- ch_syn_name(H,H1), !, ch_int(T,T1).

ch_syn_name(adrbuf1,adrbuf) :- !.
ch_syn_name(adrbuf2,adrbuf) :- !.
ch_syn_name(memAR1,memAR) :- !.
ch_syn_name(memAR2,memAR) :- !.
ch_syn_name(memDR1,memDR) :- !.
ch_syn_name(memDR2,memDR) :- !.
ch_syn_name(memDR3,memDR) :- !.
ch_syn_name(A,A) :- !.
```

2.13 Implementation Issues

We have completely reimplemented the ASP system since the prototype system was completed in 1987. The reimplementation effort was aimed at three problems; generality, maintainability, and speed.

Generality was a problem with the prototype system because it was not designed to support complex designs. Input specifications were constrained in form, and the module generator for the data path could not generate bit field selection logic. These limitation required new Prolog translation code and a new data path generator.

Maintainability was a problem because some of the code would not port from C-prolog to Quintus Prolog, which was necessary to take advantage of Quintus garbage collection. In particular, the topolog module generator had to be replaced because of this problem.

Execution speed was a problem in general with the lower level goals, which had to deal with thousands of geometric elements and were orders of magnitude slower than equivalent C programs. In particular, we have rewritten the compactor to use lists instead of assert and retract, in a successful effort to improve its performance.

In addition to addressing the above problems, the reimplementation also caused a change in the system's structure. In the prototype system behavioral and boolean sysnthesis were combined. In the current system they have been spilt, which clarifies the separate issues raised at each level.

2.14 Conclusion and Future Work

ASP formalization by DCG is an effort to fix a language format of a certain level. The whole hierarchical design was divided into three major group. The formatting between each link is critical for the point of view of implementation. VHSIC is the one of that effort. For the general and flexible formatting, we still need more experience.

The design choices at the high level synthesis is still a problem because it should be well characterized for the lower level design. Logic level systhesis has an ability to estimate among several functional unit modules. Regulating of library should be eleminated for the multiple design. Random logic library systhesis can support design choices without concering the only possible modules. This time, the estimation of functional unit based on the cell is critical.

The global control/data flow is analyzed based on the target machine architecture. So, translation rules can be set up for the design with minimizing effort. The target machine code generation can use those analysis.

The whole ASP system can be formalized from FIPER to STICK-PACK. DCG uses decomposition method from Non-terminal to replacible Terminal. One problem is to carry hugh list structure. It can generate some memory swap space lack problem. So, well organized algorithm should be used.

The run time suggestion is to use BFS even though the standard Prolog and DCG is characterized as a DFS. They need lots of symbol table with a careful format. But, it is about 14~20 times faster if there is a deep depth to branch.

Piper and Topolog will be formalized to get a fixed language format. And continuous feed back from the bottom will be continued.

3. PIPER

Piper is the high-level synthesis component of the Advanced Silicon Compiler in Prolog (ASP) system. The high level components (Viper / Piper) of ASP takes input of abstract instruction set architecture specified in a subset of Prolog, and generates the data path and control path of the pipelined microprocessor. Through levels of translations and synthesis, they also explore the major dimensions of the design space. The lower levels (Topolog, CPgen, and StickPack) of the ASP system further synthesize the data path and control path into blocks, transistors, and finally, the complete chip in CIF format.

The input to ASP is the ISA (instruction set architecture) specification of the microprocessor. Two of the most important features are: first, it is an algorithmic description of the behavior of the microprocessor. The specification language is carefully chosen out of a subset of Prolog such that, in addition to being a specification language, it can be simulated in standard Prolog environment. Secondly, the specification is abstract. The bit widths and values, concurrence, timing, and hardware entities (such as functional units, buses) are not present. Nor is the structure of pipeline specified.

The advantages are that the abstraction allows the designer concentrate on the behavior, and releases designer from specifying the hardware and control details. The simulability of the specification allows the early behavior verification and performance estimation. Since the specification is abstract, the design system has the flexibility of exploration various designs while satisfying the behavior specification and other design constraints.

3.1 System overview

This section gives an overview of the high level components of ASP. There are two major components: Viper and Piper. They automates the design of pipelined instruction set micro-processor from abstract ISA specification.

Viper is one of the high level components of ASP, and serves as the front end of Piper. It translates the ISA specification in Prolog into a sequential abstract finite state machine. Viper allocates the resource templates, and performs a preliminary scheduling (currently, 'as soon as possible' is implemented in Viper) based on the dependency relationship, and then produces a state transition graph which represents the sequential behavior of the machine under design. A state consists of a set of concurrent abstract register transfers whose resources are to be allocated later by Piper.

Piper takes the output of Viper, an abstract finite state graph, performs state decoupling (pipeline stage assignment), dependency resolution, bindings of operations to functional units, optimal allocation of resources for pipelined design, creation of interconnection, and finally generate the data path description in terms of modules and netlists, and the control path description in terms of symbolic boolean equations, control modules, and data path/control path interface.

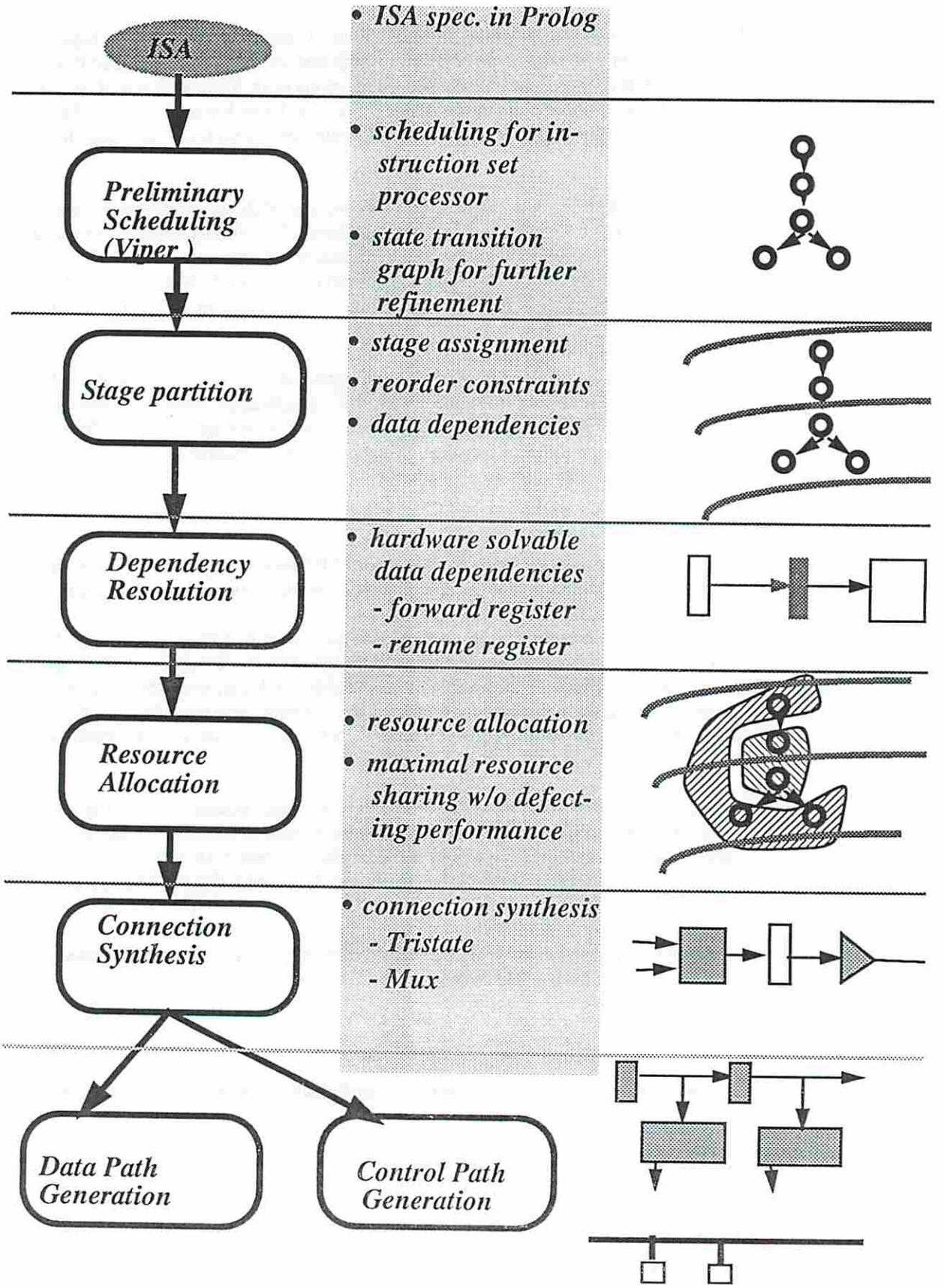
In addition to performing the synthesis task, Piper also characterizes and explores the pipeline design space, and estimates the impact of benchmarks on the pipeline designs.

3.2 System organization

This section and its subsections describe the sequence of transformation and synthesis which Piper performs.

In order, the major tasks which Piper performs are:

1. DAG transformation
2. Dependency analysis for pipeline synthesis
3. Stage assignment and benchmark analysis


ISA
• ISA spec. in Prolog
Preliminary Scheduling (Viper)
• scheduling for instruction set processor
• state transition graph for further refinement
Stage partition
• stage assignment
• reorder constraints
• data dependencies
Dependency Resolution
• hardware solvable data dependencies
- forward register
- rename register
Resource Allocation
• resource allocation
• maximal resource sharing w/o defecting performance
Connection Synthesis
• connection synthesis
- Tristate
- Mux
Data Path Generation
Control Path Generation

4. Dependency resolution
5. Resource allocation for pipeline synthesis
6. Connection synthesis
7. Control path generation and cost estimation
8. Data path generation and cost estimation

In the following subsections, we will examine these tasks in details.

3.2.1 Input to Piper

The inputs to Piper consist of two parts: specification of architected registers, and a finite state graph. The register specification describes the static aspect of the instruction set architecture, and defines the basic state space of it; in other words, the definition of registers which the designer specifies explicitly and are accessible by instructions. On the other hand, the finite state graph, representing the dynamic aspect, defines the sequential behavior of the instruction set architecture. Note that these two specifications are represented as Prolog programs and can be simulated with a simple simulation driver written in Prolog.

3.2.1.1 Specification of architected registers

The specification of architected registers at Piper's level consists of three types of predicates: register object, register width, and register fields.

The predicate '*element*' defines the object in the data path. Different types of data path objects are identified, created, or allocated in different phases of synthesis. The architected registers are the earliest data path objects identified. The format of '*element*' is:

element(<type>, <name>, [<bus in: to be allocated later>],[<bus out: to be allocated later>], [<control in: to be determined later>]).

For example,

element(reg,ac,[],[],[]).

In this statement, '*ac*' is a data path element of type '*reg*' (register). The connections (bus in, bus out, controls) are to be created after the resource allocation for pipeline synthesis.

The '*stateRegister*' predicate defines the bit width. '*stateRegister*' has two forms:

stateRegister(<name>,<width>).

stateRegister(<name>,<input | output | input/ouput>,<width>).

The second form further specifies the attribute of the register is an I/O register, i.e., a register with connection to out side of the microprocessor.

The optional predicate '*stateField*' specifies the fields of architected registers.

stateField(<name>,<fieldName>,<Highbit-Lowbit>).

For example, in sm1 [Appendix B2], register memDR has two fields: *opcode*, and *address*:

stateField(memDR, opcode, 15-8)

stateField(memDR, address, 7-0)

These three predicates define the basic state space of the instruction set architecture which is observable to the designer.

3.2.1.2 Finite state transition graph

The state transition graph describes the sequential behavior of the instruction set architecture. A state consists of a set of concurrent abstract register transfers, and the next state information. All the RTLs of the same state are executed concurrently. And the next state can be an unique state, or a branch switching on the value of architected register or output from templates of internal latches and abstract functional unit.

state(<id>,[<list of abstract RTLs>],<next state id>)

state(<id>,[<list of abstract RTLs>],switch(<target>,[case(<value>,<next state id>), ...]))

The following example specifies the concurrent RTLs of state *bc(1)* , and its next state as *bc(2).*

state(bc(1),[move(pc,bus(unassigned),memAR),enable(pc,inc)],bc(2)).

The next example indicates that the state *bc(3)* has no action (RTL) associated with it, and the next state depends on the value of field *opcode* of register *memDR.*

state(bc(3),[],switch(field(memDR,opcode),[case(stor,bc(21)), case(shr,bc(19)), case(output, bc(17)), case(load,bc(15)), case(jmp,bc(13)), case(brn,bc(9)), case(and,bc(7)), case(add, bc(5))])).

Please note that we use symbolic values in ISA specification, and these symbolic values are passed down to Piper. The symbolic values will be mapped to binary values and the sizes of the state register which holds those symbols will be determined when Piper generates its control path design as the input to CPgen (the control path generator of ASP).

The abstract RTL is either a data move between data path elements (architected registers and templates of other resources to be allocated), or the function specification of a resource template. In ASP, a legal function is a function which can be executed in one clock cycle. To represent a multiple-clock function in ASP, we divide it into subfunctions.

move(<source name>, <bus template>, <destination name>)

enable(<resource template>, <function>)

For example, the following set of concurrent abstract RTLs represent the behavior of 'r1 = r1 + r2'. Note that buses and functional unit (*fu(1)*) are templates. The *enable* statement specifies the function to be performed by *fu(1)* is *add.*

move(r1,bus(b1),port(fu(1),1))

move(r2, bus(b2),port(fu(1),2))

enable(fu(1), add)

move(fu(1), bus(b3), r1)

For complete example of the finite state transition graph, please refer to [APPENDIX B2].

3.2.2 DAG transformation

The first phase of Piper assigns a time dimension and a condition dimension to the state graph. The concept of time is implicit in the input of Piper. In the state transition graph, the trace of 'next state' is a linear temporal sequence. When a 'root' state appears as a next state of any current state, a 'loop' or 'recursion' is found. The condition dimension is defined by a conditional branch. The state transition graph is traversed, and each state is assigned a time stamp, and a template of control register is created for every branch in the state transition graph. There is case that a state may have different time stamps since it can be reached from more than one trace. In this case, the largest value is assigned as the value of time stamp for this state, and dummy states are inserted to the shorter traces.

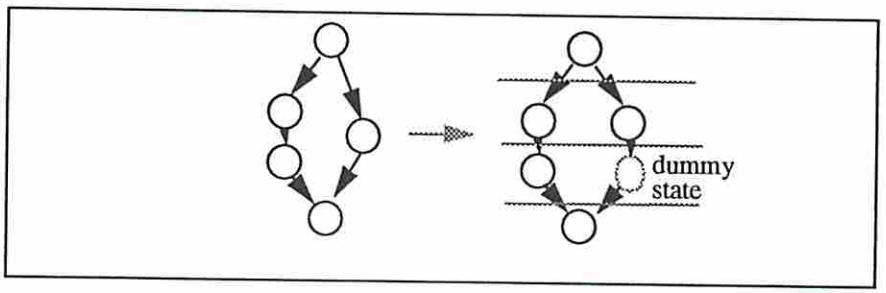


Figure 3.1 Dummy state assignment

The purpose of control registers is to hold the values that determine which branches to take in the state transition graph. Together with a system counter, they can identify the current active states in the state transition graph.

The formats of predicates for control registers are:

controlRegister(<template name>,<value>,<state>)

controlRegInfo(<template name>,level(<#>), width(<#>))

The first predicate specifies the name of control register template, its value, and the state associated with it. The second predicate records some of the important properties which will be used later for control path allocation.

In the sm1 example, two templates of control register are created:

controlRegister(controlReg1,field(memDR,opcode),bc(3)).

controlRegister(controlReg2,ac<constant(0),bc(9)).

controlRegInfo(controlReg1,level(1),width(3)).

controlRegInfo(controlReg2,level(2),width(1)).

Several dummy states are inserted into the state transition graph, for example:

state(bc(19),4,[controlReg1-shr], [move(ac,bus(unassigned),shfone(shf1)), move(shfone(shf1) , bus(unassigned),ac), enable(shfone(shf1),shr1)], dummyState6).

state(dummyState6,5,[controlReg1-shr],[],dummyState7).

state(dummyState7,6,[controlReg1-shr],[],bc(1)).

Two dummy states are inserted between states *bc(19)* and *bc(1)*.

Once the time stamps and control register templates are created, we are now ready for further processing.

3.2.3 Dependency analysis for pipeline synthesis

The dependency analysis for pipeline synthesis is more complicated than the traditional dependency analysis on a data/control flow graph (DCFG). In pipeline execution, there are several consecutive tasks being processed along the pipe. The subtasks of consecutive tasks are executed overlapped in time at different parts of pipe. Therefore, in addition to identifying the dependency relationship of micro-operations for a single task (for example, instruction), the dependency relationship of consecutive tasks (instructions) have to be satisfied as well. Since the main focus of ASP is on the synthesis of instruction set architecture, we can consider a task as an instruction execution without ambiguity.

There are two categories of dependencies we have to identify:

1. Intra-task dependencies (dependencies within single task execution): This is identical to the traditional dependency analysis. For Piper, we are interested in:

   data dependency (read after write)

   anti dependency (write after read)

   output dependency (write after write)

2. Inter-task dependencies (dependencies between consecutive tasks): The inter-task dependencies we are interested are:

   data dependency (read after write)

   anti dependency (write after read)

   output dependency (write after write)

These inter-task dependencies can be derived from the same data/control flow graph. The inter-task data dependency appears as the write-after-read pair per possible trace. If there are multiple *write* and *read* states on the same trace, the earliest *read* state and the latest *write* state are paired as the inter-task data dependency. When the *write* state is reachable from *read* state in the data/control flow graph, it is also an anti dependency of intra-task. In the case when the *write* state is not reachable from *read* state in the data/control flow graph, we call it as a pseudo-anti dependency of this graph.

The inter-task anti and output dependency appear as the read-after-write and write-after-write pair per possible trace, respectively. In the case where multiple *read/write* states exist on the same trace, the earliest and latest states are paired.

Table 3.1 summarizes the relationship between dependencies and the state transition graph.

Task relation	Dependency	In state transition graph
Intra-task	data dependency	read-after-write pair
	anti dependency	write-after-read pair
	output dependency	write-after-write pair
Inter-task	data dependency	farthest write-after-read pair/ per trace
	anti dependency	farthest read-after-write pair/ per trace
	output dependency	farthest write-after-write pair/ per trace

Table 3.1. Dependency and State transition grapth

3.2.4 Stage assignment

After the time stamps, control register templates are created, and the dependency information becomes available, we are ready to begin the synthesis of pipeline machine.

The first step is to partition the finite state graph into several disjointed state graphs. In other words, a pipelined machine is represented as a linear cascade of several finite state machines. Each stage of the pipe is a finite state machine which executes sequentially. The stages execute concurrently and synchronously while the sequential behavior specified in the ISA specification preserved.

For a finite state graph with critical path of time stamp s, i.e., a critical path of length s, there are s meaningful ways of partitioning the state graph into disjointed subgraphs. Of course, there are more than s ways of partitioning the graph, but lots of them won't result in efficient designs in the sense of both performance and cost; therefore these inefficient designs are not considered by Piper. It is sufficient to consider the following s ways of graph partitioning.

Briefly speaking, the stage assignment algorithm works, for each way of partitioning, by assigning states of m consecutive time stamps into a partition where $1 \leq m \leq s$. However, the stage assignment is subject to the dependency constraints. The intra-task output dependency enforces the involving states to be in the same partition (stage).

The maximal speedup of the m'th way of graph partitioning (pipeline stage assignment) is s/m. This is the performance given by the pipelining execution at full rate.

However, due to the inter-task data and output dependencies, the pipe may not execute at full speed all the time. For each pipeline stage assignment, Piper generates a set of reorder constraints for compiler specifying the number of *no-op* which have to be inserted between two consecutive instructions which are data or output dependent. The compiler may actually insert *no-op*'s or find some other independent instructions and insert them in between, subject to the reorder constraints.

When benchmarks are available, with the information provided by benchmark simulator, Piper can also estimate the average speedup of any pipeline stage assignment.

3.2.5 Dependency resolution

This phase solves the intra-task data dependency by creating forward registers. For each pair of intra data dependent register transfers, if there is one or more pipeline stages between them, we need to forward the data across the stages.

To create forward registers, we first find the farthest destination stage, for all intra-data dependencies of the same source register, and assign the distance as the forward distance. For every stage between the source and farthest destination, we create a forward register for the source.

Next, we have to create the forward controls. For each intra-data dependency spanning across stages, we create a forward register transfer for each stage in between. If the stage which contains the forward register transfer has more than one time step (pipeline cycle), then schedule this RTL to the time step which uses the least number of buses.

Conceptually, the forward register allocation is part of the resource allocation of Piper. However, the resource (forward register) allocated at this phase is not used directly to perform the function specified in ISA specification. In stead, it is used to help emulating the behavior of sequential machine (ISA specification) on a group of finite state machines which are linearly cascaded and execute concurrently and synchronously. Therefore, the process of forward register allocation is isolated from the resource allocation to be discussed in the next section, and is treated separately.

3.2.6 Resource allocation for pipeline synthesis

The functional units and buses are allocated for the pipelined machine obtained from the previous sections at this phase. The resources are allocated in the fashion that maximal degree of sharing is achieved such that the costs of functional units and buses are minimized.

Piper uses an unique two level hierarchy to model resources. ***Module*** is the object at the top level. It is a basic object in the data path, and is recognized by Topolog, an circuit domain component of ASP. ***Function*** is the object at the low level in the hierarchy, and represents the function performed by the module. A module may be able to perform multiple functions, and different modules may be capable of performing the same function. For example, 'bus' is an module which performs the function 'connect'; 'adder' is a module which is capable of functions 'add' and 'sub'; 'alu' is a module with functions 'add', 'sub', 'or', 'and', and 'xor'.

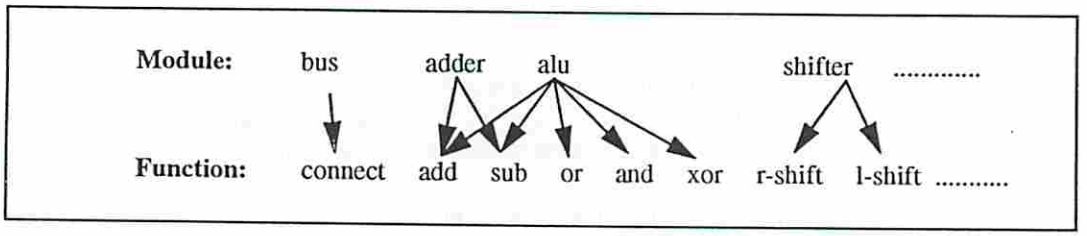


Figure 3.2 Resource model for Piper

One of the attributes associated with module is cost. Currently the cost is modeled as the transistor area. In the future version of Piper, power consumption will be taken into consideration. However, one difficulty of modeling the cost is that it is hard to find an unique measurement such that the costs of all modules are estimated upon it. For example, the relative costs of memory port and alu do not imply any useful suggestion since they serve different functions and are not compatible modules. Therefore it is not necessary to enforce an unique criterion to model the cost. The hierarchical model for resource provides a good solution to the cost modelling. Let M_i be a module, and F_j be a function, and F_{Mi} be the set of functions which M_i supports. And then we define the *relevant* group as following:

$$\text{relevant group } R_j \equiv \text{a set of modules such that for } (\forall M_p \in R_j, \exists M_q \in R_j, F_{M_p} \cap F_{M_q} \neq \varnothing)$$

In other words, for every module M_p of a particular relevant group, there is at least one module M_q of this group which is also capable of some of M_p's functions. This is equivalent to partitioning the module space into disjointed groups under the closure operation memtioned above. With the relevant group, now all we have to ensure is that the cost estimation has to be relatively accurate for modules within a relevant group. The costs of modules of different relevant groups are incompatable. For example, in figure 3.2, bus itself is in one relevant group; adder and alu consist in another relevant group.

The control registers are allocated at this phase too. The templates of control registers created in the phase 1 are now bound to objects of control registers. The goal is to allocated as less control registers as possible while the proper controls of pipelined machine is still maintained. For each stage, which is a finite state machine, we consider its state transition graph. A branch is characterized by the nest depth and the width of the control register which controls the branch. The exclusive branches can share the same control register if they require control registers of the same size. Exclusive branches are defined as branches which are not nested in the state transition graph. In the following figure shown is a finite state transition graph, with states omitted from the graph. The lines represent the traces of state transition. The heavy nodes (b1,b2,b3,b4) indicate the branching states. Of the exclusive pairs, branches b3 and b4 can share the same control register since they are exclusive, and they both need control register of size 1-bit.

However, while the control registers are shareable within a stage, it is not possible to share control registers across the stages since each stage is a finite state machine which executes independently and the exclusiveness of branches between stages is hard to ensure at design time in general. But in the case that only a finite set of benchmarks would be executed on the machine, it is possible to increase the degree of sharing by considering all finite state machines altogether at the cost of exponential computing time since we have to enumerate all possible state combination for all stages.

3.2.7 Connection Synthesis

The logical connections of modules are expanded into physical connections by adding tri-states and multiplexers into the data path.

For every bus connected to the output of a module, a tri-state is created, and if there are field definitions for the output of the module, then this tri-state is further decomposed into several tri-states corresponding to the number and positions of the fields defined for this module.

If there are multiple buses connected to the input of module, a multiplexer is needed to properly switch the buses. There is no need to split the multiplexer into fields.[1]

The register transfers in the states are properly modified as well to control the tri-state and multiplexer to create a dynamic data path.

The following two phases create structural descriptions for control path and data path such that they can be carried on by the lower level components of ASP: Topolog and CPgen.

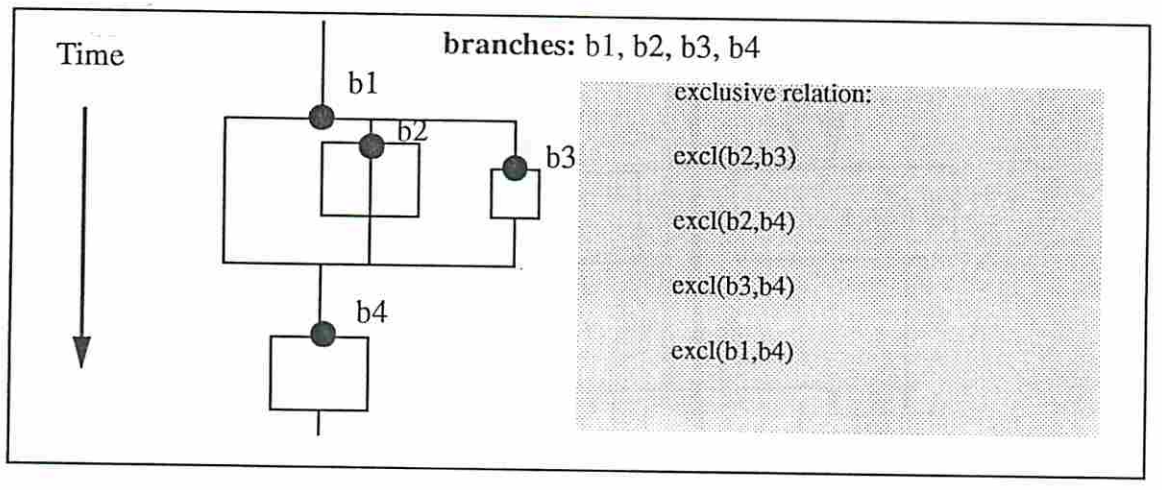


Figure 3.3 Exclusive branches in finite state transition graph

1. According to the ISA specification language, a register or an output ports can be accessed field-wisely; however, a register or an input port is treated as an entity such that it is not possible to set part of the contents, and leave the other parts unchanged. This limitation is due to the difficulty of looping back the un-changed contents to the same register or input port of functional unit. If it is necessary to be set fieldwisely, each field has to be defined as a register. And then we can combine those registers defining the fields, and feed them to the input port of functional unit.

3.2.8 Control path generation

In the following subsections, we describe the control model of the pipelined machine first, and then presentent the language which serves as the interface between Piper and CPgen.

3.2.8.1 Control model for pipelined machine

The underlying model of control path for pipelined machine is a set of linearly cascaded finite state machines. Each stage of the pipe is a finite state machine which executes sequential.

There is a global counter to drive and synchronize these finite state machines. For each stage, it serves as a timer to indicate the time step the stage is at. Each stage has its own control registers (state registers). The control registers and the global counter define the state space for each stage. A counter n works by repeatedly counting from 1 to n with the zero reserved for initialization and interrupt purposes.

The control registers get their contents from either data path or control registers of previous stage. Therefore, there are interfaces between control registers, and between data path and control registers.

For every stage, there is a combinational logic circuit which generates the outputs (control signals to data path) and sets the next state. Currently, PLA is used in CPgen to implement the logic.

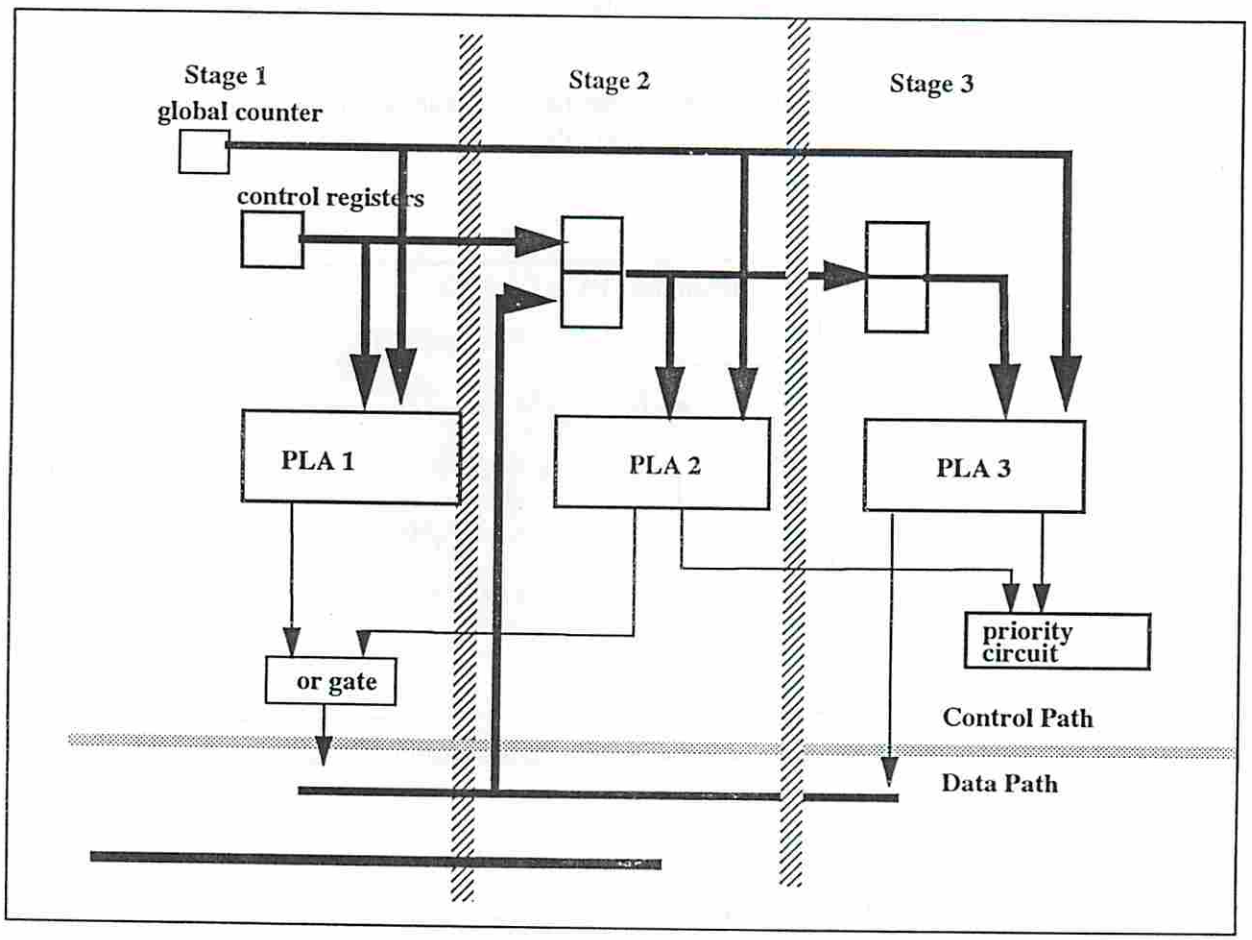


Figure 3.4 Control model for pipelined machine

Since we adopt the distributed control methodology, there are cases where several stages want to control the same object in the data path at the same time. This conflict is resolved by, on one hand, generating the reorder constraints to instruct the compiler to properly schedule instructions by simply inserting 'nop' or reordering instructions; and on the other hand, using priority circuits to grant the control signals from the deepest stage the priority. The reason is that even though the compiler delays the next instruction, there is still an instruction 'no-op' entering the pipe which may enable the control signals. Therefore, a priority circuit is necessary to handle this case.

Finally, the non-conflicting and conflict-resolved control signals from different stages have to be or'ed together if they control the same function in the data path. 'OR' modules are allocated for this purpose.

3.2.9 Structural description of control path

The control information includes a global counter, a specification of stage number, control registers, boolean equations, symbol binding, the data/control path and control/data path interfaces, and the interconnection.

In detail, the control information is represented in the following predicates:

counter(<integer>)

no_of_stage(<integer>)

controlRegister(stage(<integer>), <name>)

boolean(stage(<integer>), <function>, [<Sum of Product>])

symbolTable(<data path object>, <control register name>, [<symbol-value>])

module(<priority | or>,<name>,<I/O>, [<input signal>], [<output signal>],[],[])

dpcp(<data path object>, <bus>, <bit-range>,<control register>)

connect(<source control register>, <destination control register>)

The naming scheme for control registers is that control registers of different stages have the same name if the same data is moved between them. In other words, the control registers which hold different time instances of data from the same data path object have the same name. An index is associated to differentiate them. For example, the predicate *controlRegister(stage(2), foo)* specifies a control register *foo* at stage two, and the predicate *controlRegister(stage(3), foo)* specifies a control register *foo* at stage three which gets its content from the control register of the same name (*foo*) at stage two.

The boolean equation includes a dependent variable (*function*) and its independent variables in sum-of-product form. In the sum-of-product, the variable appears as multiple-valued logical variable, and these values are symbols which are bound to integers in the *symbolTable* predicates. The integer is further mapped to binary representation by CPgen which is the logic and topology synthesizer of control path. An index is used to identify the stage which the boolean equation is associated with. For example,

boolean(stage(2),enable(ac,load),[[controlReg1-add,clock-2], [controReg1-and,clock-2]
, [controlReg1-load,clock-2]])

where *enable(ac,load)* is the function to be activated in stage two when its boolean equation is satisfied. The boolean equation is represented in a sum-of-product form. The inner list represents a product term; the outer list represents the sum. *enable(ac,load)* is activated if the value of global counter is 2 and the content of the control register *controlReg1* of stage two is either *add, and,* or *load.*

An example of symbol table looks like:

symbolTable(field(memDR(1),opcode),controlReg1,[nop-0, add-1, and-2, brn-3, jmp-4,
load-5, output-6, shr-7, stor-8])

The *dpcp* and *connect* define the pipes of control register. *dpcp* specifies a path of data movement from data path to control register in the control path. The width of the control register is derived from the width of the data in the data path. *connect* indicates how the content of control register is forwarded to next stage. Therefore, the property of control register of current stage is succeeded from the property of control register with the same name in the previous stage. Examples of *dpcp* and *connect* are:

dpcp(field(memDR(1),opcode),bus(1),15-8,controlRegister(stage(1),controlReg1))

connect(controlRegister(stage(1),controlReg1),controlRegister(stage(2),controlReg1))

The control register *controlReg1* of stage one gets its content from the *opcode* field of data path register *memDR(1)*. The bit-range is from bit 8 to bit 15. And the content of control register *controlReg1* of stage one is forwarded to control register *controlReg1* of stage two at the end of pipeline cycle.

It is straight forward in the module specifications for priority circuit and or gate. The forth and fifth arguments stand for the input and output lists of the module, respectively. A function has form of:

enable(<module name>, <function>)

In case that more than one control module generates the same control signal, an index is used to associate the control signal to the control module. For example,

control(stage(2),enable(pc,load))

represents a function *enable(pc,load)* which is generated by stage two; and

control(priority(d,pc),enable(pc,load))

represents a function *enable(pc,load)* which is output of control module *priority(d,pc)*

3.2.10 Data path generation

The data path information is translated into a structural hardware description that the lower level components of ASP can use. This includes a netlist based module description, interfaces of data and control signals in and out of data path, and the external pins.

A module has a type, a name, and nets (buses) which connect to the inputs and outputs, and the names of control signals.

In detail, each module has the form:

module(<type>, <name>,<not used>, [<busIn>], [<busOut>], [<controlIn>], [<controlOut >])

where the bus-in and bus-out are lists of nets with bit-range specification. For example, the following predicate indicates a module *mux(memAR(1))* with type as two-input multiplexer. The inputs are nets *bus(1)* and *bus(2)*, both with bit-range of 15-0. The output is net *in(memAR(1))* with bit-range 15-0. There is one control signal to this multiplexer: *mux(mux(memAR(1)),fn)*.

module(mux(2), mux(memAR(1)), x-x, [bus(1):[15-0], bus(2):[15-0]], [in(memAR(1)):[15-0]], [mux(mux(memAR(1)),fn)], [])

The predicates datapath/2 and datapath/3 define the boundary interface of data path. The boundary interface includes the signals running between data path and control path, and the signals running between data path and external pins. The formats are:

datapath(<incontrol | outcontrol>, <signal>)

datapath(<indata | outdata>, <bit-range>, <net>)

The first form specifies the control signals running from control path to data path (*incontrol*) , and vice verse (*outcontrol*). The second form specifies the data signals running from external pins to data path (*indata*), and vice verse (*outdata*). Since the data nets usually have width of more than one bit, we need a bit-range specification here.

The names and types of pads are defined by the pad/2 predicates.

pad(<net>, < in | out | bi>)

There are three types of pads: input, output, and bi-direction pads. In the case of bi-directional pad, an additional predicate bi_pad/3 to identify the input and output ports and the selecting signal:

bi_pad(<name>, <width>, [<input>, <output>, <selecting signal>])

For complete example of data path description, please see appendix B2.

3.3 Design space exploration

-to be continued-

3.4 Benchmark analysis

-to be continued-

3.5 Discussion on the design methodology

-to be continued-

3.6 Piper system files

The section lists the functionality and synopsis of important files and programs of Piper.

3.6.1 System maintenance

Name	Description
makefile	maintaining design and system consistency
localmakefile	local information for makefile
npmakefile	makefile for non-pipelined design
pipercompiler.pl	source code of prolog 'compiler' for Piper
pipercompiler.go	driver for pipercompiler.pl, invoked by 'prolog < pipercompiler.go'

3.6.2 Executable programs (unix utilities)

Name	Synopsis (used as unix commands)	comment
make	make Spec=<Spec>	
pipercompiler	pipercompiler <Piper Program>	program name w/o suffix .pl

3.6.3 Executable programs (Piper tools: listed in execution order)

Name	Synopsis (used as unix commands)	comment
depth	depth <Spec>	
dep	dep <Spec>	
stage	stage <Spec>	
	stage <Spec> <Option>	Option= seq ...non-pipelined design Option= <integer> ...pipeline cycle
stageperf	stageperf <Spec>	
ddr	ddr <Spec>	
prepalloc	prepalloc <Spec>	
width	width <Spec>	
palloc	palloc <Spec>	
postpalloc	postpalloc <Spec>	
pcpdp	pcpdp <Spec>	
cpstyle	cpstyle <Spec>	
pelab	pelab <Spec>	
cpext	cpext <Spec>	
pdpe	pdpe <Spec>	

3.6.4 Environment & Library

Name	Description
piperenv.pl	To be loaded before loading any Piper system. Setting up the environment and libraries
piperLib.pl	Library (for programming purpose)
piperLocalLib.pl	Local library predicates
piperDesignLib.pl	Design library

3.6.5 Prolog programs of Piper System

Name	Description
depth.pl	Time and condition dimensions assignment
dep.pl	Dependency analysis for pipeline synthesis
stage.pl	Stage constraint generation and stage assignment
stageperf.pl	pipeline performance analyzer
ddr.pl	Data dependency resolution
width.pl	Data path width analysis
prepalloc.pl	preprocessing of bus usage
palloc.pl	Resource allocation for pipelined microprocessor
postpalloc.pl	postprocessing of bus usage
pcpdp.pl	Pipelined data/control path construction
cpstyle.pl	Control path cost estimation
pelab.pl	connection synthesis
cpext.pl	structural description for control path
pdpe.pl	structural description for data path

3.7 Reference

[1] William Bush et al., "A Prototype Silicon Compiler in Prolog," *Technical report, report no. UCB/CSD 88/476,* Computer Science Division, University of California, Berkeley, December 1988

[2] Alvin Despain et al., "Aquarius," *Computer Architecture News,* March, 1987

[3] Alvin Despain, "The Design System (ASP) of the Aquarius Project," *Proceedings of the Second International Workshop on VLSI Design,* December, 1988

[4] Nohbyung Park and Alice C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," *Transactions on Computer-Aided Design, Vol 7. No. 3,* March 1988

[5] Michael C. McFarland, Alice C. parker and Raul Camponsano, "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE, Vol. 78, No. 2,* February 1990

[6] R. Camposano, "From Behavior to Structure: High-Level Synthesis," *IEEE Design & Test of Computers,* October 1990

[7] GioVanni De Micheli, "The Olympus Synthesis System," *EEE Design & Test of Computers,* October 1990

[8] Vijay K. Raj, "DAGAR: An Automatic Pipelined Microarchitecture Synthesis System," *Proceedings International Conference On Computer Design,* 1989

[9] Forrest Brewer and Daniel Gajski, "Chippe: A System for Constraint Driven Behavioral Synthesis," *IEEE Transactions on Computer-Aided Design, Vol. 9, No. 7,* July 1990

[10] Yoheved Dotan and Benjamin Arazi, "Concurrent Logic Programming as a Hardware Description Tool." *IEEE Transactions on Computers, Vol. 39, No. 1,* January 1990

[11] William Bush et al., "Experience with Prolog As a Hardware Specification Language," *Fourth Symposium on Logic Programming,* 1987

[12] Chia-Jeng Tseng et al., "Mind: A Module Binder for High Level Synthesis," *Proceedings International Conference On Computer Design,* 1989

[13] Pierre B. Paulin and John P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design, Vol. 8, No. 6,* June 1989

[14] Richard J. Cloutier and Donald E. Thomas, "The Combination of Scheduling, Allocation, and Mapping in a Sinble Algorithm," *Proceedings of Design Automation Conference,* 1990

[15] Barry M. pangrle and Daniel D. Gajski, "Slicer: A State Synthesizer for Intelligent Silicon Compilation," *Proceedings International Conference On Computer Design,* 1989

[16] Mick Tredennick, "How to Flowchart for Hardware," *Computer,* December 1981

[17] David Ku and Giovanni De Micheli, "Relative Scheduling under Timing Constraints," *Proceedings Design Automation Conference,* 1990

[18] Cheng-Tsung Hwang et al.., "Optimum and Heuristic Data Path Scheduling under Resource Constraints," *Proceedings Design Automation Conference,* 1990

4. TOPOLOG

The Topolog system is the major circuit-design component of the ASP silicon compiler. This is the second major level which is the circuit or functional domain in the ASP[5]. The purpose of this domain is to present the behavioral component with abstract components. Hence, this level attempts to synthesize and connect the finite state machine and functional units generated by the behavioral level. This level encompasses the traditional tasks of logic synthesis, placement and routing, and module generation.

Topolog is the combination of four major parts which are module generator (module library), cell library, cell placement and cell routing. Topolog has nine pipeline stages which are signal_preprocess, map_unit, cell_net, pre_min_cut, min_cut, placer, pre_route, router and combine. Each major parts in Topolog will be described in the following paragraphs.

The input of Topolog is the modules description and the output of Topolog is in SIP (Stick-pack In Prolog) format. The Topolog is the transformation process from circuit functional description to physical description in transistors level. In the other word, Topolog is the combination of logic synthesis and layout engine.

4.1 Module Generator

The module generator is the first step of Topolog. It can be called module library. It included three pipeline stage, signal_preprocess, map_unit and cell_net.

The module generator provides the mapping from modules into a list of blocks which is map_unit.pl program. In this mapping, a block can be viewed as a cell name and its external signals. The module library is shown in Figure 4.1. The input for the module library is .mod file from Piper. The format for the module is:

```
module (Type, instance, bitrange, external_signals).
```

The **signal_preprocess.pl** program generates an unique and simple alias for each distinct external signal. The external signal names generated by Piper are generally long with unnecessary information for Topolog. So long as the names are unique, simpler ones are more appropriate for Topolog processing.

The **map_unit.pl** is a module generation program. It matches the module name with module library and breaks it down (or synthesize) to several basic cells. It also distributes the cells to each bit slice. This is done in a predicate called map_unit :

```
map_unit(Module,First-bit,Last-bit,Current-bit,Signal-list,ID,In,Out):-

        add_block(Current-bit,In,(cell,[cell-signals]),In1),
        add_block(Current-bit,In1,(cell,[cell-signals]),In2),
        add_block(Current-bit,In2,(cell,[cell-signals]),In3),

        Next is Current-bit + 1,
        map_unit(Module,First-bit,Last-bit,Next,Signal-list,ID,In3,Out).
```

The output of map_unit.pl is a list of blocks in each bit slice of the datapath. The format is:

```
bit(Bitnum, Blocklist).
.....
```

The gen_priority.pl is a priority circuit generator. Priority Circuit resolves the priority of the control signals over the same module generated by different control PLAs of each stage. The priority circuit is part of control path, and it can be implemented in many ways: a PLA, a random logic, or a specialized circuit. The specialized circuit approach was chosen because it was believed to minimize the area and delay. The output is a single bit slice of blocks. The similarity of the output format to the datapath allows us to use the same placement and routing as the datapath. But this approach has disadvantages, too. We might have a very long or very short bit slice that affects the floor planing.

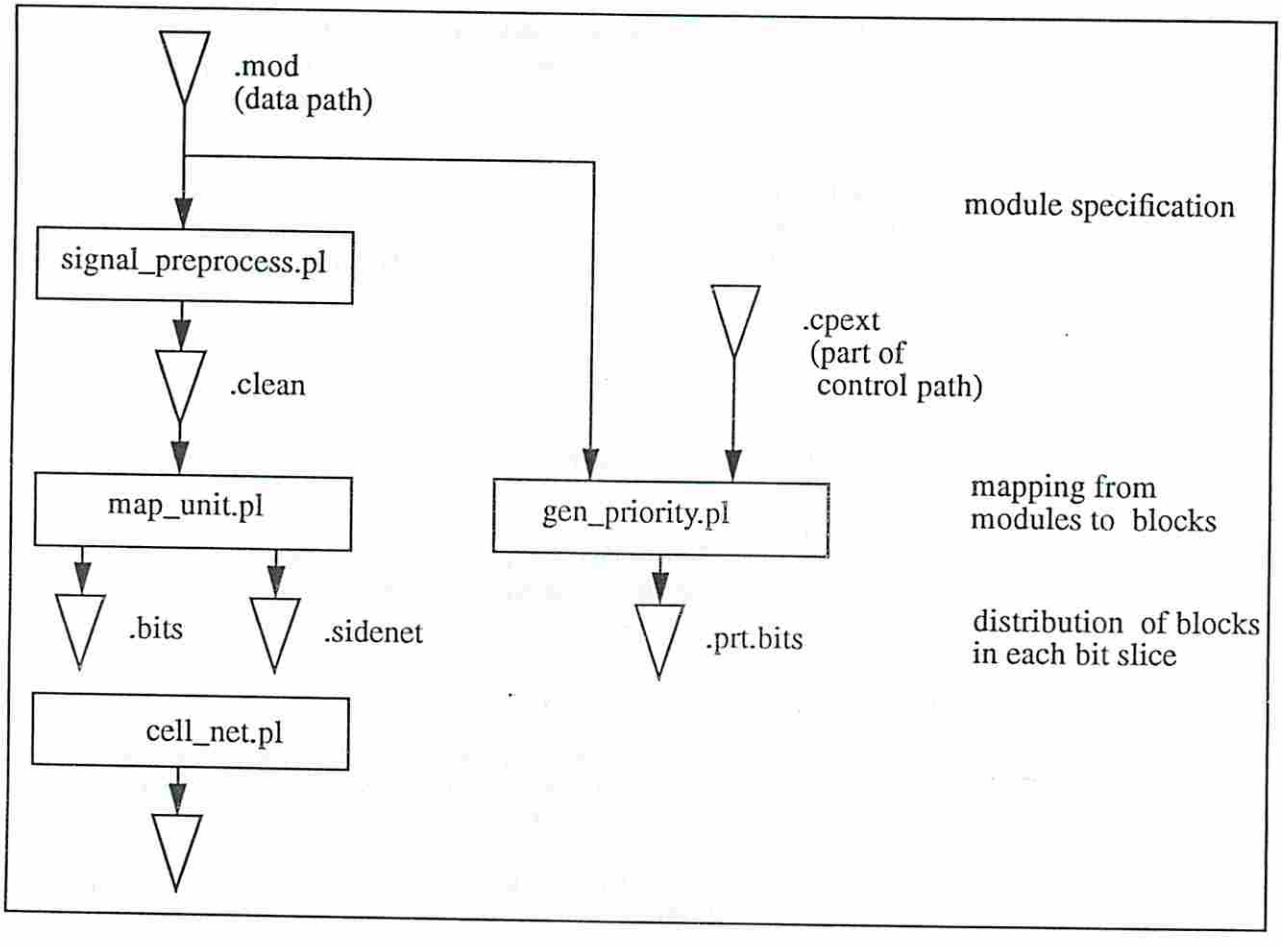


Figure 4.1 module library

The third pipeline stage is **cell-net.pl** which transfers the block list and the external net of the block to a net list format and gives each block a unique name and identify the constraint block at this process. The constraint block is either top block or bottom block or through block which should be placed relatively close to the same constraint block in the other bit slices.

The format of the cells is :

cell(Block_name, Block_ID_number, External_net, Size).

The cell information will be further transfered to more efficient format for placement to use. The router and the placer will read in this information to get the block from cell library according to the block_name. We will describe this process in detail in the following paragraph.

The process of adding a module is as the following:

step 1. Draw a circuit diagram and identify the cells needed.
step 2. Check the cell library and create a cell if needed.
step 3. Perform circuit level simulation to decide the transistor size.
step 4. Distribute the blocks in the bit slices and add it into map_unit.pl program.

The part of logic synthesis has done for adding new module into library. For example, map_unit can support all kind of 'constant' module by read in constant(N) where N is a integer number and map_unit will produce a module called constant with binary number equal N.

The module also can become a submodule for other big module to call. For example, the 'smult' module is a multiplier which need a lot of register, mux, and adder. The map_unit for this module can call other submodule like adder to support it instead of design it in detail again.

4.2Topolog Library

The Topolog library provides the mapping from modules to physical layout of the cells. The Topolog library can be viewed as two parts: the cell library and the module library (See Figure 4.2). The Topolog library has been updated to obtain correct functionality and efficiency in area and delay. The underlying philosophy behind the new library is adopting the bottom up methodology, as well as the top down. That is, the module generation is affected by the lower level characterizations of the basic cells.

Many problems were solved by introducing the lower level information of the basic blocks. The previous library has limitation to specify modules. It was either awkward or impossible to have multiple feed-through in different bit slices, which was required to specify the carry chain of the ALU or the comparators. Three special blocks, top, bottom and through, made it possible to specify multiple feed-through signals in different bit slices.

The modules in the previous library was composed of unit size (2 lambda by 3 lambda) transistors and only the transistors in the critical path were resized by transizer. The SPICE simulation on the modules has shown that modules with unit size transistors do not generally operate correctly. In the new library, each transistor in the cells are given a certain size that has been decided upon a circuit-level simulation of the modules. The modules in the new library have been designed to exhibit more efficiency. The pass transistor logic was adopted in the newly designed modules to enhance efficiency in area and delay.

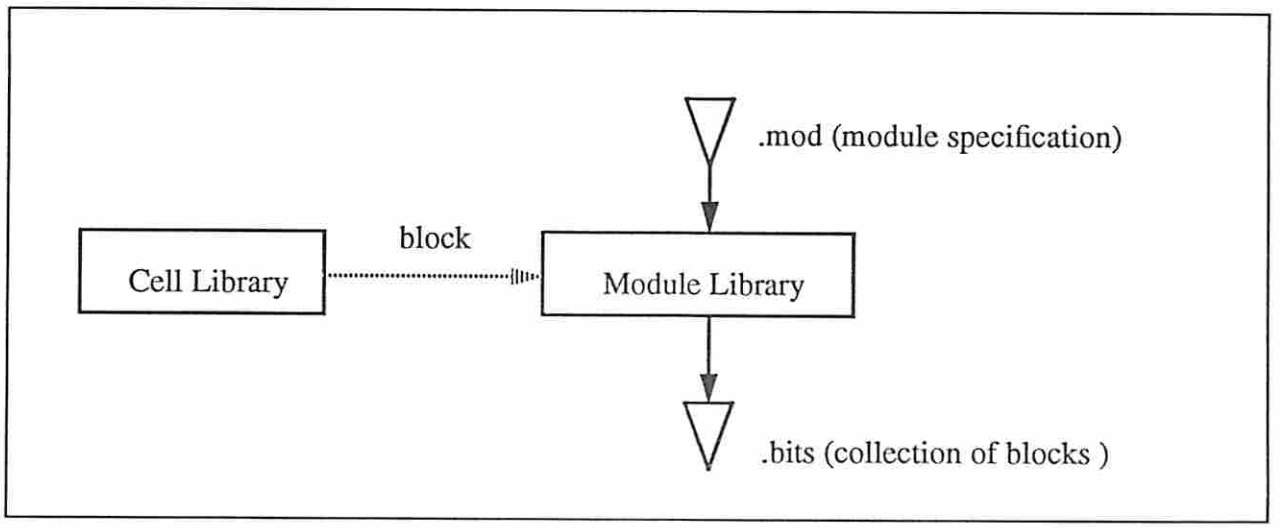


Figure 4.2 Topolog Library

4.3 Cells

The cell library contains the physical layout information for the basic cells. The cells are various gates such as and, or, nand, nor, pass transistors, as well as special cells for control signals such as top, bottom and through. The cells are kept in different files to make adding and maintaining the library easy. A cell is created when it is needed to specify a certain module. The cells that we support are listed in topolog.lib (See Figure 4.3).

```
% libcell(English_description_of_function, name_of_cell).

libcell(cmos_pass_transistor, cpass).
libcell(pair_of_wire_ored_cmos_pass_transistor, cpasspair).
libcell(nmos_pass_transistor, npass).
libcell(inverter, inv).
libcell(a_2_input_and, and2).
libcell(a_2_input_or, or2).
libcell(a_2_input_nand, nand2).
libcell(a_3_input_nand, nand3).
libcell(a_4_input_nand, nand4).
libcell(a_2_input_nor, nor2).
libcell(a_3_input_nor, nor3).
libcell(a_4_input_nor, nor4).
libcell(a_2_input_xor, xor2).
libcell(a_2_input_var_nmos_ulm, nulm).

libcell(thru_block, thru).
libcell(bot_block, bot).
libcell(top_block, top).
```

Figure 4.3 Topolog.lib

Each cell is an array of transistors, with a row of p-transistors at the top and a row of n-transistors at the bottom. The transistors within the cells are optimally placed using the algorithm by Uehara and Vancleemput so that the area and the internal routing is minimized. Those cells whose name ends with i, are the inverted cells. These cells have p-transistors at the bottom and n-transistors at the top. They are used to route Vdd and GND easily. That is, if the regular cells are distributed in one bit slice, then the inverted cells are distributed in the next bit slice.

The transistors and terminals in a cell are represented using the following sip format:

trans(Type,Source-pos,Gate-pos,Drain_pos,Width,Length>, Source-signal,Gate-signal,Drain-signal).

Type = nd / pd

Source-pos, Gate-pos, Drain-pos = pt(X,Y)

pin(Side,Location,Layer,Electrical-node,Label).

Side = hiy / loy

extsig(External-signal-list).

extsigin(External-Insignal-list).

extsigout(External-Outsignal-list).

maxx(Maximum-x-number,Cell-name).

maxy(Maximum-y-number,Cell_name).

map(Side,Virtual-grid).

Side = top / bot

Trans is used to represent a transistor of a gate. The arguments <width> and <length> are the scale factors for the transistor size. They are multiples of the width and length of a minimum size transistor(2 lamda for the length, 3 lamda for the width). The transistors in a cell are the minimum size transistors that guarantee any module, which is composed of the cell, to work correctly with balanced rising and falling delay. The SPICE simulation is performed on a module to decide the transistor size of the cells that are contained in the module, assuming 50fF for the load capacitance. Each net of a transistor has a unique name. Internal nets are denoted by i(N).

Pin is used to specify the terminals for the control signals in top, bot and thru blocks. *Extsig* is used when a module is mapped into blocks. *Extsigin* and *extsigout* are used for switch-level simulation. The virtual y-grid is 4 for the top and 2 for the bot.

Figure 4.4 shows an example of a 2-input nand cell

```
trans(pd, pt(1,4), pt(2,4), pt(3,4), 2, 1, vdd, in1, out).
trans(pd, pt(5,4), pt(4,4), pt(3,4), 2, 1, vdd, in2, out).
trans(nd, pt(3,2), pt(2,2), pt(1,2), 2, 1, i(1), in1, out).
trans(nd, pt(5,2), pt(4,2), pt(3,2), 2, 1, vss, in2, i(1)).

extsig([out, in1, in2]).
extsigin([in1, in2]).
extsigout([out]).

maxx(5, nand2).
maxy(4, nand2).
map(top, 4).
map(bot, 2).
```

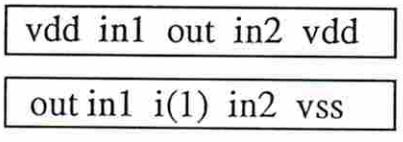


Figure 4.4 Two-input nand cell

4.3. Cell Library

The cell library is composed of blocks which is another level of abstraction of the cells. A block contains additional information on terminals, boundary nodes, internal nodes, external nodes, and so on. The additional information is extracted from the cell. Each step in Topolog has different view of the blocks. For example in module generation, a block is nothing but a cell name and external signals.

The library has the following format:

```
cell(Name, (Els, (Ts, Bs), (Tp, Bp)), ((Topterms, Botterms), (Tpg, Bpg)),
    ((Tf, Tl, Bf, Bl), Intnodes, Esigs), Size).
```

where the variables corresponds to the following informations.

```
cell(Name, (Elements, Transistors, Pins), (Terminals, Power_Grounds),
    (Boundary_info, Internal_nodes, External_nodes), Size).
```

Cell library structure is shown in Figure 4.5. The single file topolog.cells is a library representation of the cells. To add a cell in the library, create the cell with .lsip extension and run flp.pl. This will create inverted cells, by flipping every cell upside down. The inverse cells are used in the adjacent rows from the normal row in order to share the power line and ground line in between them.

An entry of created cell and the inverted cell of it should be made in the topolog.lib file. The prohram genlib.pl will create the topolog.cells from the cells (that has .lsip extension) and topolog.lib file.

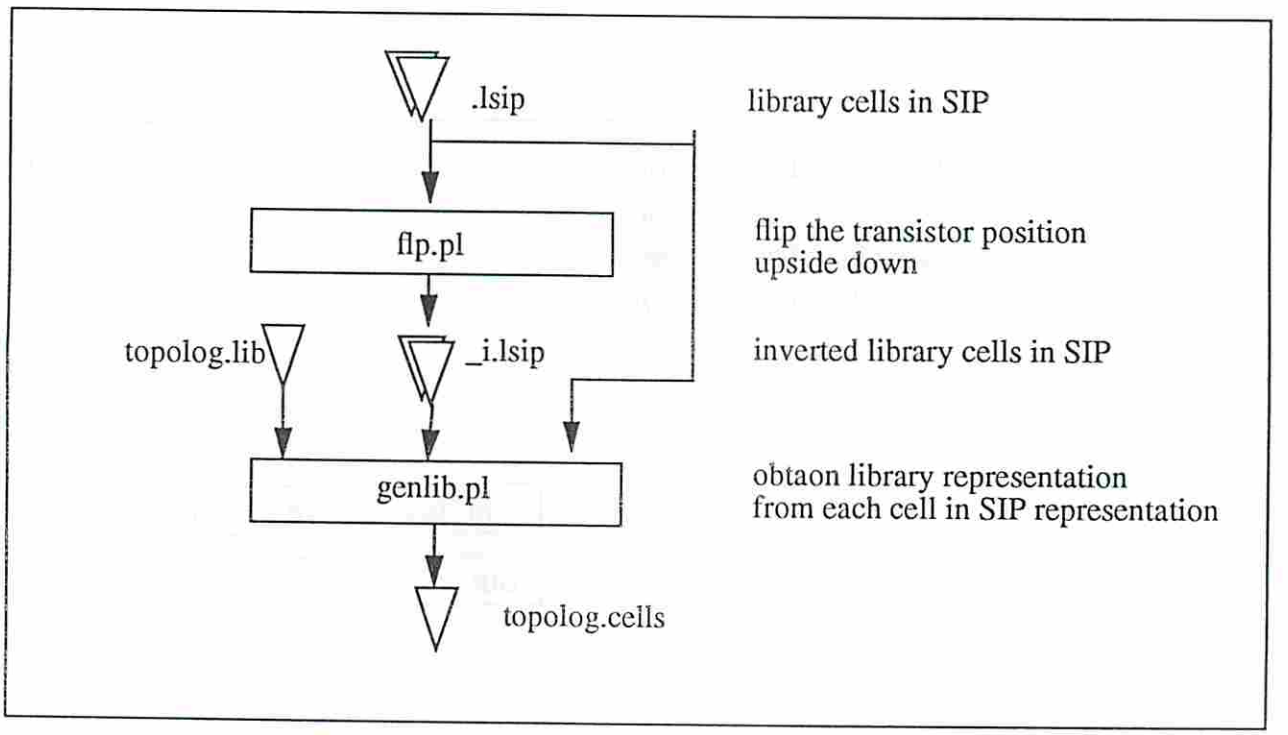


Figure 4.5 Cell library

4.3.1.Modules in Topolog

The modules in the previous library had problems on functionality and performance. All the modules were redesigned to ensure its functionality and to enhance efficiency. An effort has been made to optimize the circuit by using as many pass transistors and NAND gates. The functional units for sm1 and sm2 are static modules operated by a single-phase system clock. The modules supported by Topolog are :

alu, mux(2) .. mux(6), register, increment register, counter, comparator,

tri-state, adder, multiplier, priority circuit, etc.

Multiplexer

The 2-input multiplexer, mux(2), is a basic building module for a N-input multiplexer, mux(N). The previous multiplexer was composed of logic gates. Cascading this type of multiplexer creates considerable delay when we the number of input is large. The pass transistor logic was used in the new multiplexers. The resulting circuit takes up less area and operates faster.

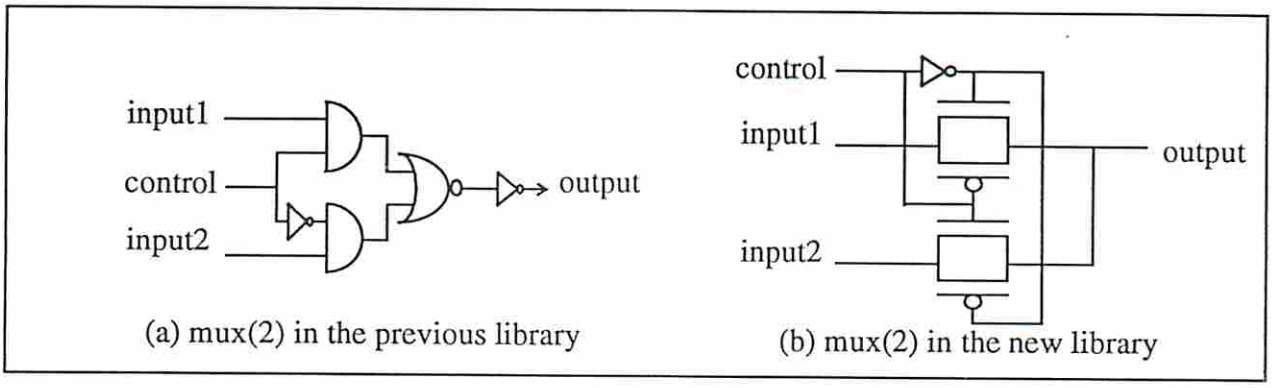


Figure 4.6 The multiplexer design

7-function ALU

The ALU operations that is required for sm1 and sm2 are ADD, SUB, INC and AND. The 6502 processor needs 3 more functions, DEC,OR and XOR. The current alu in the library is subset of the 7-function ALU for 6502. Our ALU is based on a ripple carry chain. Two gates, a AND and OR per bit are involved in the propagation path in ordinary ripple carry chain. It has been shown that we can reduce the propagation delay using associativity, resulting a 3 -input NAND gate per bit in the propagation path. This provides a speed up of 2 compared with the ordinary ripple carry chain.

The ALU is shown in Figure 4.7. It is composed of namely 3 parts: carry input multiplexing, carry chain, and output part. An effort has been made to optimize the circuit. Pass transistors are used in the carry input multiplexing and the output circuit instead of logic gates to make the circuit faster. The carry circuit is optimized using as many NAND gates as possible. The SPICE simulation on those 3 parts has been done to decide the transistor size to ensure correct functionality.

The output of alu using pass gate instead of using or-gate to group the multi-function output is mainly because that we can easily adding and removing the function from alu. For alu synthesis purpose, we can create almost any kind of function combination alu by using the pass gate to group the function together in the future.

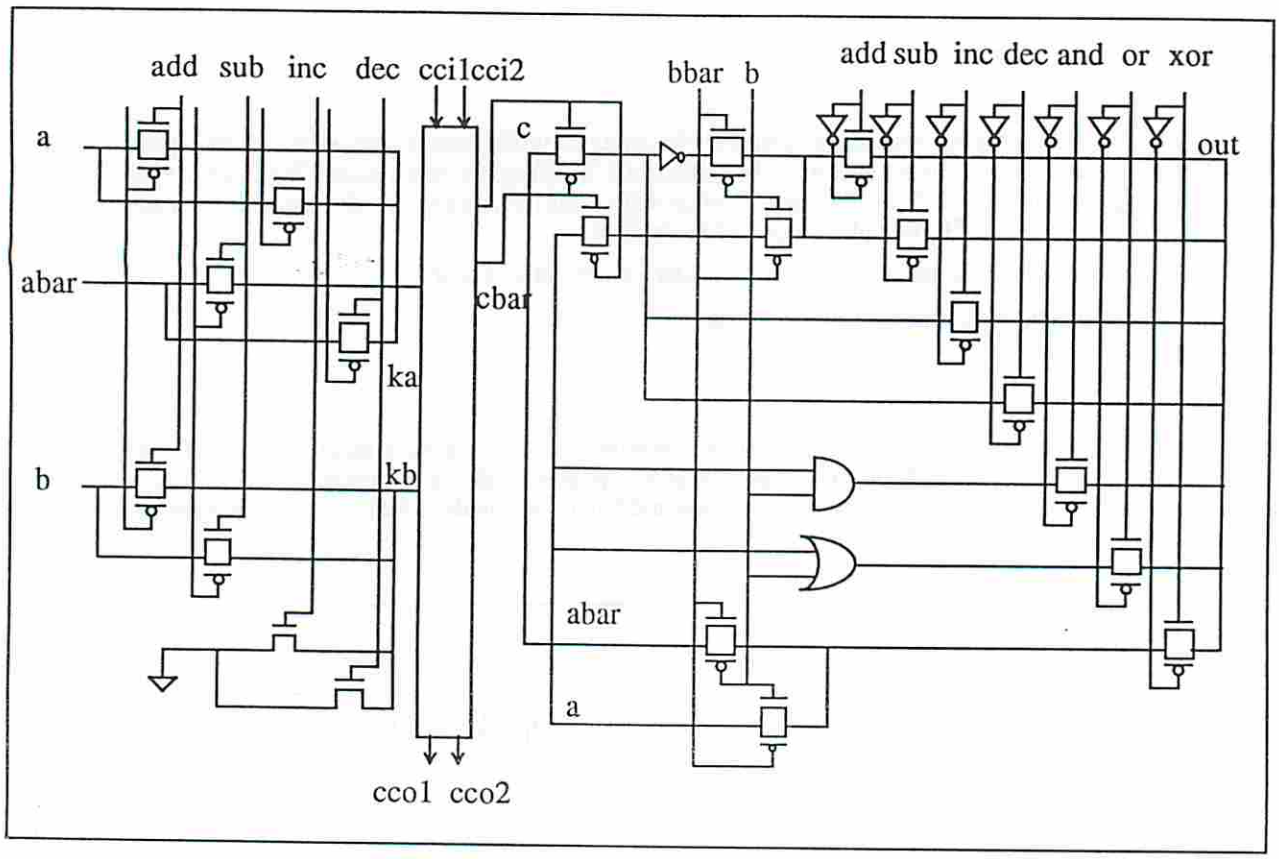


Figure 4.7 Seven-function ALU

Register

The register design is shown in Figure 4.8 . This register is master-slave type, and it is a fully CMOS circuit. The master is enabled only when both the clock and load signal is high. Otherwise output is refreshed. A reset function can be obtained easily by replacing the inverter in the slave with a NOR gate.

The transistor sizing was turned out to be important because of the feedback loop The SPICE simulation has been done to decide the minimum transistor size for 50fF load capacitance. It was found that the the width factors for the pass transistors are 5 for the p-type transistor, and 2 for the n-type.

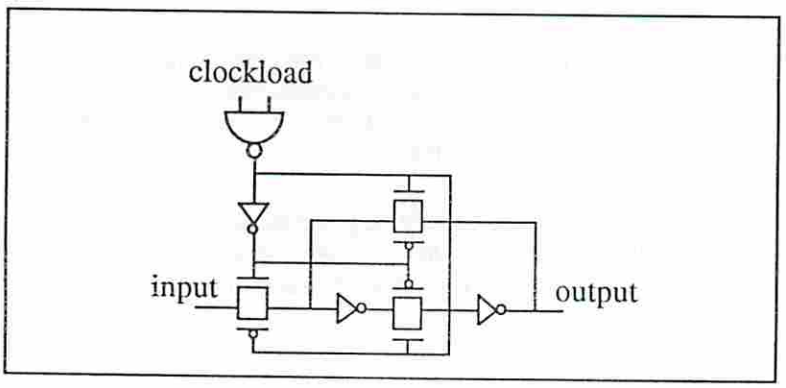


Figure 4.8 The register design

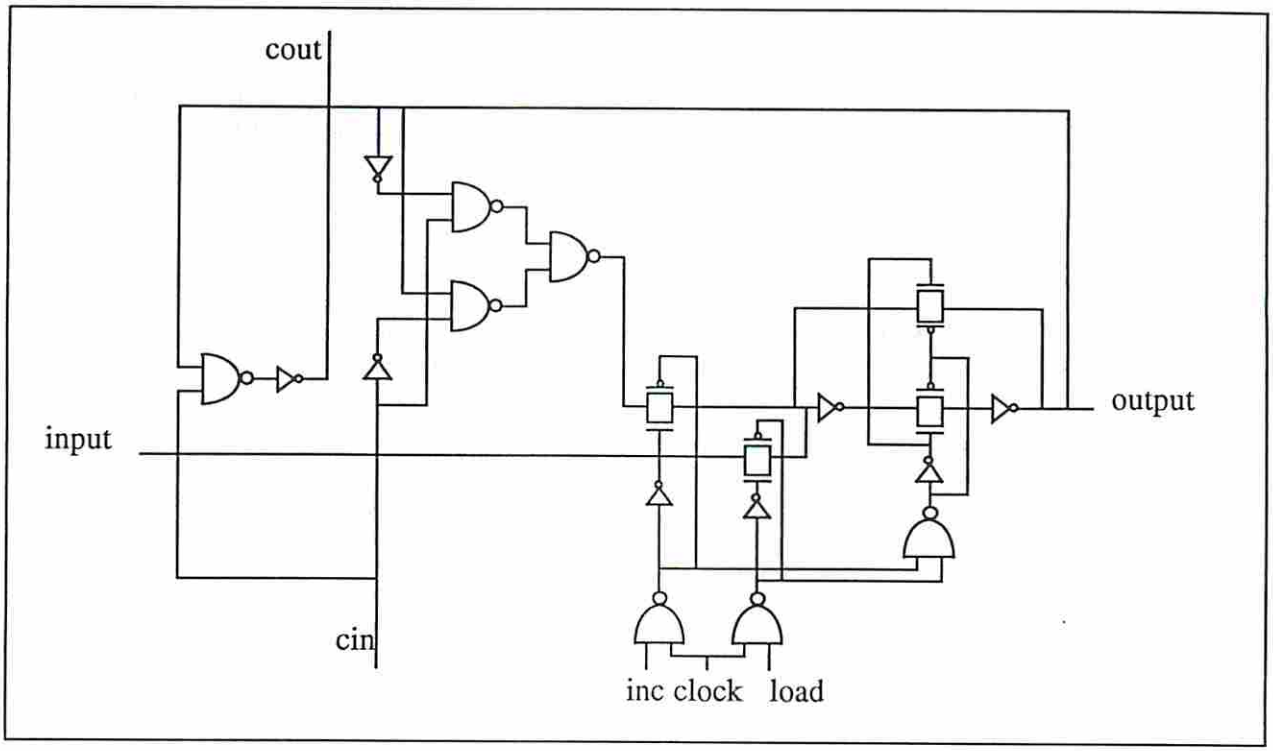


Figure 4.9 Increment register

Increment register.

The increment register is used for program counter. It is composed of a half adder and a register with multiplexer. The one bit slice of the increment register is shown in Figure 4.9. The master of the register is connected to a multiplexer. When LOAD is active, the multiplexer selects the input. When INC is active, the multiplexer selects the sum from the half adder. The half adder also produces COUT. When none of them are active, OUT is refreshed via a feedback. Note that the initial carry to the first bit should be set to zero. Reset could be included in the same manner as in the register.

Priority circuit.

Priority Circuit resolves the priority of the control signals over the same module generated by different control PLAs of each pipeline stage. Thus the priority circuit is part of control path and it is natural to implemented by a control PLA. However, a decision was made to implement the priority circuit by a specialized circuit because this approach was believed to minimize the area and delay. The specialized priority circuit is a single bit slice of blocks, which will be placed and routed by Topolog.

The circuit type is quite irregular. A different design (e.g., number of pipeline stages) results in a different type of circuit. It is impossible to have all the types of priority circuit in the module library. Hence, they are synthesized from their module specification in the .cpext and .mod. The .cpext does not give complete specification for the priority circuit synthesis. The .cpext is created even before the control signal names are decided for .mod. So the signal names in .cpext are in their primitive form, which cannot be readily used. Hence, only the information on conflicting signals and their stages are extracted from .cpext, while the actual external signal names are obtained from .mod

There are two categories of priority circuit: one type for the decoded signals, and the other for the encoded signals. A decoded signal is active if it is 1(high), or not active if it is 0(low). When a control signal is encoded, it is impossible to tell whether it is active or not. Another control signal is required to indicate that an encoded signal is active. Most control signals are decoded. Only the select signals for multiplexers are encoded. Currently, Piper generates an active signal per each set of select lines for a multiplexer.

Figure 4.10 (a) is an example of a priority circuit for decoded controls. It has 4 distinct signals (c1 to c4) out of 3 pipeline stages. If any of the signals from a stage is active (Nor/Not gate output is low), all the signals in the lower stages are killed (the n-pass transistor is switched to ground), otherwise, they pass through the cmos pass transistors. This is done from the second lowest stage to the highest stage, resulting prioritized signals. That is, only one of the 6 signals remain active. Finally, the signals are grouped by bring the same signals.

Figure 4.10 (b) shows an example of a priority circuit for encoded signals. The signal s is the select signal for a 2-input and an initial carry 1 (See Figure 4.10 (a)). Nor the output gives A=B and the carry out gives A>B or A<B. Another method is to compare individual bits using XOR gates and see if all of them are equal.

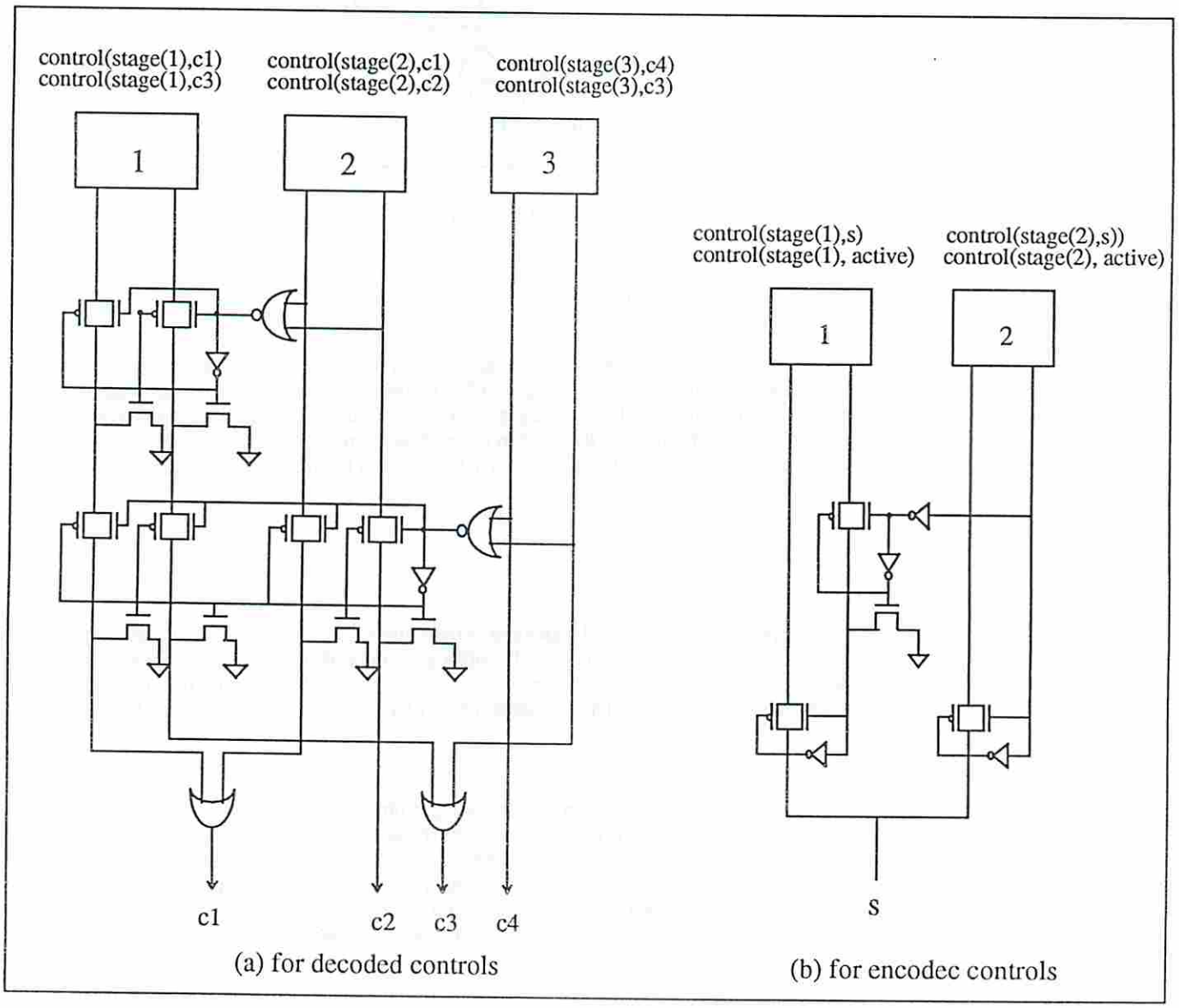


Figure 4.10 Prioritry Circuit

4.3 Placement

The placement problem is formulated as a set of cells which must be placed in a rectangular planar area. The placement in Topolog is implemented by min-cut placement[1] which is based on partitioning graphs to have minimum edges crossed [3]and place the logic circuit in bit slice style. Before the placement, The cell-net format has been transferred to object-to-object format which is

obj(name,[[connected objects..],weight ,[connected objects ..], weight, ...]).

This format can express multi-point net[4] and can weight the net which higher weight net will has shorter distance after placement. Because of these two feature, the min-cut placement algorithm must be modified in order to fit in the new model. Another feature in this placement algorithm is the non-movable elements has been identified in a special group that allow the element has the constraint that only allowed to place closed to some particular element. The bit slice structure of the circuit design using through block , top block and bottom block to connect signal between the bit slice will need the constraint block which constraint the top, bottom or through block location relatively close to the block which is going to connect.

The min-cut placement is based on the min-cut partition algorithm which is to find the minimum cut of partitioning a graph into two subsets. It started from an initial random partition and search the best pair of elements to exchange in order to reduce the number of net been cut. The exhaustive search which has the time complexity of n^2 will spent too much time in search when the circuit become very large. The placement algorithm in Topolog has employed two different search strategy, one is the exhaustive search another is quick search. The quick search is trying to find the best element in each group and exchange these two. The method can reduce the search time complexity from n^2 to n but the pair found by using quick search may not the best pair to exchange. But for very large circuit which will use a lot of process time, this method is faster and still can find a good placement result.

The output of the placement is an ordered list of blocks. The format is:
block_order([ordered list of blocks]).

The program plm.pl will read in this block_order and change the block number to the block name and the signals. The program pin.pl will identify the routing pins for each bit slice and store these routing pins in a file for router to read in to do the routing part. The program structure of the placement is shown in Figure 4.12.

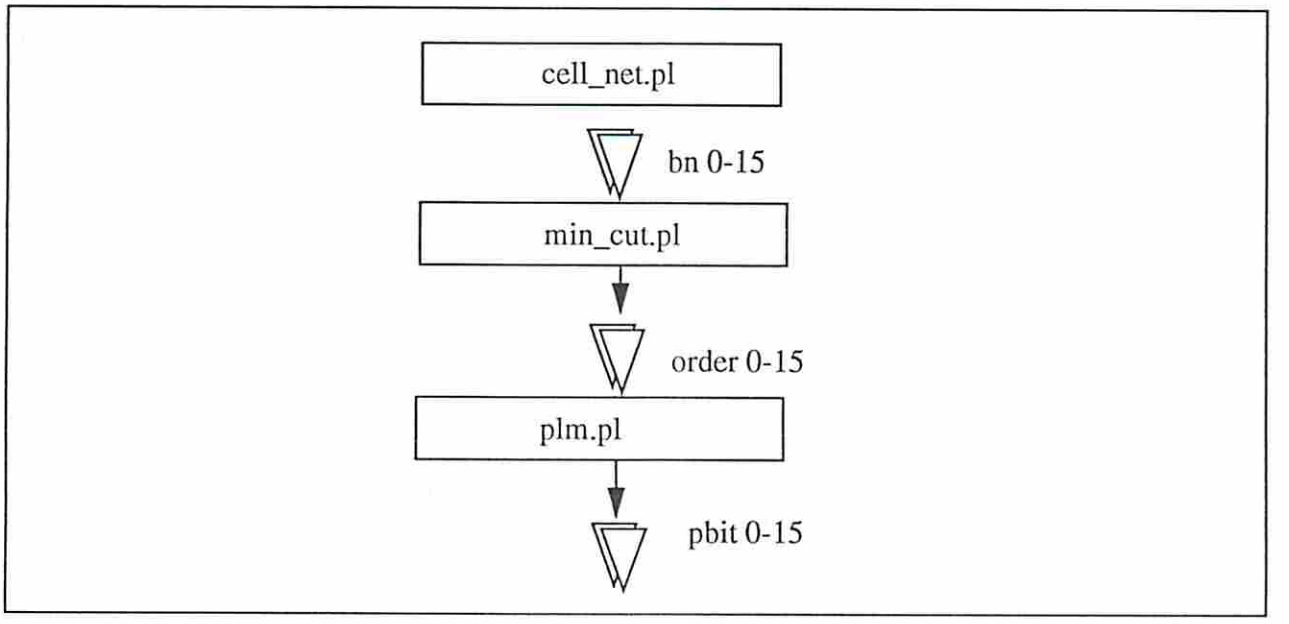


Figure 4.11 Placement Programs

4.4 Routing

The final job in Topolog is connect the signals between the transistors by wires and contacts. The algorithm we used to route the signals is channel router [5]. We use this algorithm is because it is efficient and simple and guarantee 100-percent completion if constraints are noncyclic.

The whole process include four different program to accomplish this goal. The programs are pin.pl, strip.pl router.pl combine.pl. The pin.pl identifies the routing pins in the cells and put it into router.pl for routing part only. the strip.pl place the transistor in the right location according to the placement result. the router.pl simply performs the channel route. The combine.pl combine the all bit slice SIP files which the location may need some adjust. the whole picture of process flow is shown in Figure. 4.13. The strip.pl and pin.pl are all inside strip.pl now for programming convenience.

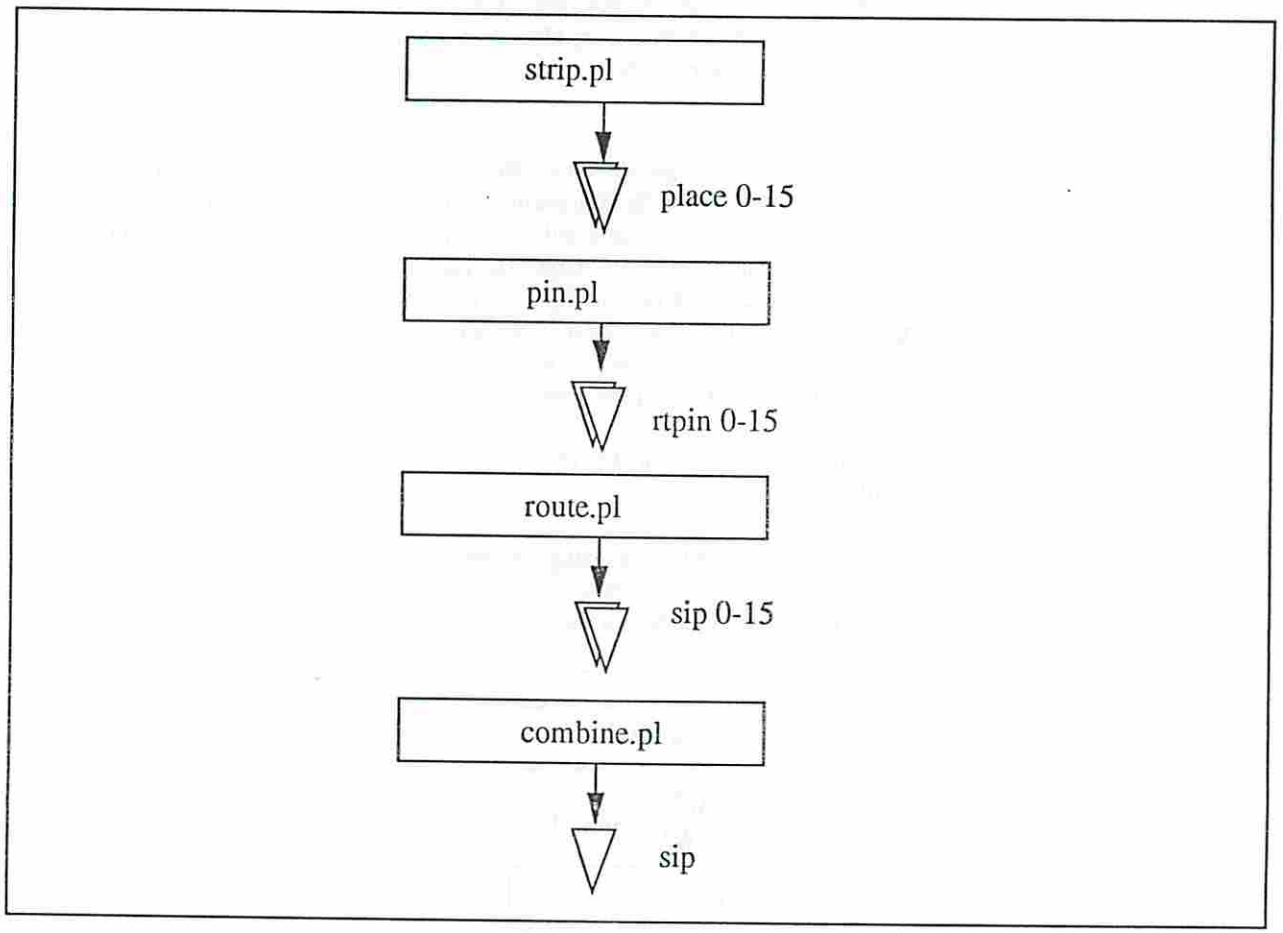


Figure 4.12 Routing Programs

The final output of the router is the final result of Topolog. The output is in SIP format which is Stick-pack In Prolog. There are for type of predicates in SIP as we mention before, trans, wire, cont and pin which stand for transistor, wire, contact and output pin. These sixteen files (for sixteen bit slice) will feed into stick-pack program which is the third level of ASP system for further process which will be describe on next chapter.

Reference

1. Melvin A. Breuer, "Min-cut Placement", VLSI circuit layout theory and design, IEEE press.

2. W.F. Clocksin, C.S. Mellish, "Programming in Prolog".

3. B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs", Bell Sys. Tech. J., vol. 49 February 1970, pp291-308.

4. D. G. Schweikert and B. W. Kernighan, "A proper model for the partition-ing of electrical circuits", proc. 9th Design Automation Workshop, June 1972, pp57-62.

5. P. C. McGeer, W. R. Bush, G. Cheng, A. M. Despain, "Prolog for VLSI Layout: Experiences in the Design and Implementation of Topolog, A Prolog Based module Generation and Layout System", UCB/CSD 87/363

6. Takeshi Yoshimura and Ernest S. Kuh, "Efficient Algorithms for Channel Routing" IEEE Transactions on Computer-Aided Design of Integrated Circuit and Systems, Vol. CAD-1, No. 1, Jan. 1982.

5. TRANSIZOR

When a VLSI circuit is designed, we must consider not only the functional correctness but also timing behavior, chip area and power consumption. Usually, users will give some specification on maximum time delay. In order to meet the delay constraint, we need to re-size(sizing) some transistors[4][23][24][25].

Increasing the size of transistors in a VLSI circuit tends to decrease the time delay through the circuit, but at the cost of increasing its area and power consumption. However, beyond a certain point, larger transistors actually increase the time delay. So, how to find a optimal transistor size of the circuit become very significant in the VLSI circuit design domain.

Human designers avoid considering all transistors in the circuit by using intuition and heuristics. After some initial configuration is chosen, simulations and timing analyses are run on the circuit to find its critical paths (the paths through the circuit whose delay exceeds the constraints) and the designer reduces their delay sufficiently. Now some other paths may be critical, so the process iterates until the maximum delay through the circuit is satisfactory. No formal attention is paid to transistor area.

In a large circuit, however, there may be many paths each requiring more time than permissible. If an iteration of such critical path heuristic is required for each path, the total computation required may be immense. One solution is to consider more than one critical path at once, such as simulated annealing method. However, human designer can not consider the circuit in such a global way, we need to employ some design automation system.

The design automation system ASP(Advanced Silicon compiler in Prolog)[1] includes a transistor sizing program named TranSizor, which is incorporated into ASP for optimizing the delay and area of the microprocessor chip design. In prototype ASP system (ASP I), there is a similar sizing program called MOST[2]. However, the difference between these two programs is MOST runs stand-alone with small number of transistors, but TranSizor runs in ASP II with thousands of transistors (SM1 about 3000 transistors, for example).

TranSizor reads unsized transistors in SIP(Sticks In Prolog) format from the files produced by Topolog and then outputs sized transistors to the files fed to Sticks Pack. Figure 5.1 shows the position of TranSizor in ASP system (i.e. Topolog -> TranSizor -> Sticks Pack). From the figure, we can see the width of transistor is changed from 1 to 3 for n-type transistor and from 1 to 6 for p-type transistor.

The major algorithms using in TranSizor are Simulated Annealing (SA) and Critical-Path Heuristic (CPH) [3][14][16]. In SA algorithm, with a input delay constraint, Transizor can find the optimal transistor size for the certain delay constraint and chip area. On the other hand, in CPH algorithm, TranSizor will find the minimum delay of the circuit (as fast as possible) but does not consider the chip area. Of course, the SA algorithm needs more execution time than the CPH algorithm, but SA will consider the area within the optimization, which CPH will not. It's hard to say which method is better, so Transistor just includes these two methods and let users choose alternatively[6].

TranSizor uses PTA (Prolog Timing Analyzer, similar to Crystal[12][13]) to calculate the time delay of each path. The delay model in current version of PTA is a simple lumped RC delay model. An interesting feature of PTA is its ability to provide symbolic equations for the resistance and capacitance at each path. From this, we can solve the differential equation of each path to find the minimum delay transistor size.

5.1. Preliminary

5.1.1 Terminology

The usual statement of a problem is "Assign sizes to the components of circuit X so that all its outputs are produced by time T." T is called *maximum delay* or *delay constraint*, and this entire process is called *sizing* the circuit. The actual delay through the circuit is the delay given a particular assignment of sizes to its components, while the size

or area of a circuit is the sum of the component areas.

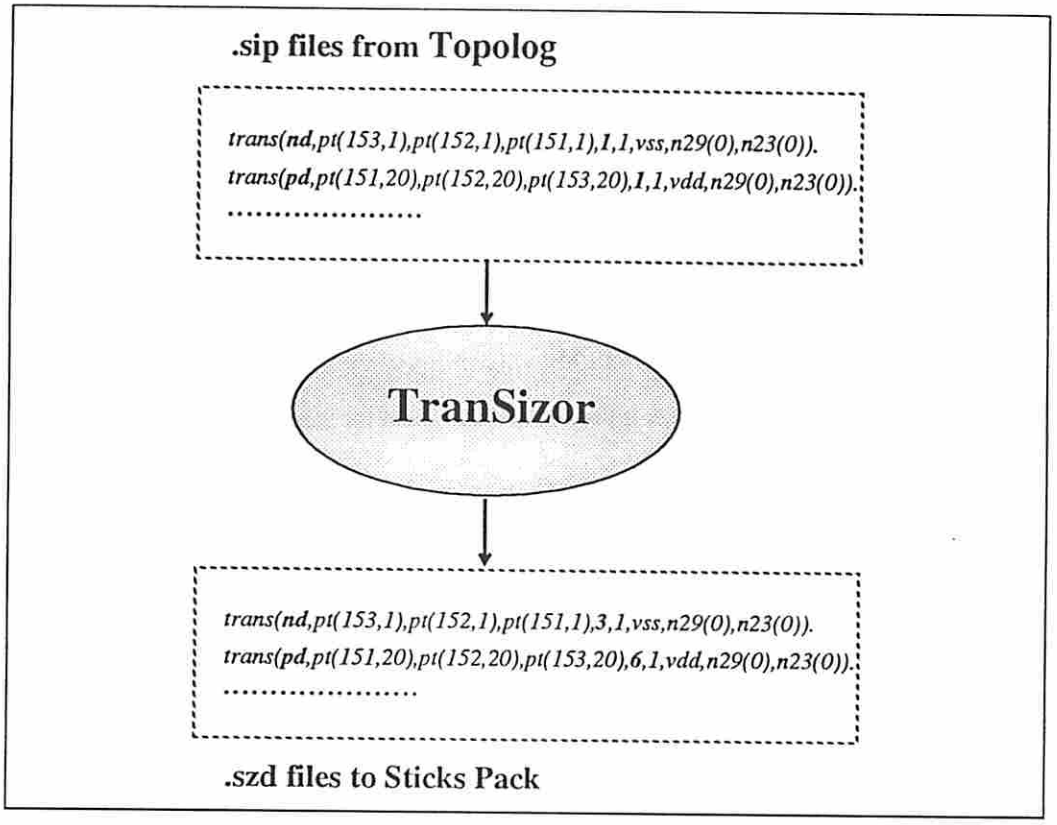


Figure 5.1. TranSizor in ASP system

A path will be made up of several *stages* (hierarchical *cells* in this report). A stage is a chain of transistors from previous stage to next stage. In each cpu cycle (*state* in this report), a path may come from stage of some register to other stages which also end in some register.

Computing the path delay through the circuit is the job of timing analyzer. Essentially, what needs to be done is find possible path delay in each cpu state, and then take the maximum of all these path delays. The path with the lowest delay is referred to as the *critical path*. In a ISA cpu circuit, the critical path may occur in ALU carry-chain or in memory access.

A related problem is that of *minimizing delay* through the circuit. In this case, there is no explicit delay constraint; the goal is to make the circuit run as fast as possible. Additionally, there may be an area constraint, or some maximum permissible transistor area.

5.1.2 Transistor size and delay

The most previous work, the path is viewed as being made up of logic gates, rather than individual transistors.Most authors assume that signals generated by gates in the path are not used anywhere but their successor, and that inputs similarly do not come from outside the path[15][17][18][19]. Furthermore, the ASP system deals with ISA microprocessor, TranSizor will find the possible paths in each cpu cycle instead of searching all the paths in the circuit. These assumptions allow efficient trace of the paths and avoid some potential fault paths.

The most obvious approach for sizing is simply to use an already-existing general purpose optimization package along with a highly accurate timing analyzer such as SPICE. However, too much time spent in simulation makes this approach impractical. TranSizor employs symbolic derivatives to avoid expensive numerical computation.

The other way of avoiding the high computational cost is to use a simplified model for transistors. At some cost of accuracy, this saves greatly in computation, especially if symbolic derivative information can be calculated. When using a simple RC model (e.g. lumped RC model), it is possible to derive the equations for path delay in terms of the transistor sizes. Consider the critical path as made up of n stages (cells in this report), each of which in turn drives the next stage (i.e. cell), and let Di be the delay of stage i. Then[21]

$$D_{tot} = \sum_i D_i$$

and, from the lumped RC model

$$D_i = R_i \times C_i$$

Now let T be all the transistors making up stage i; Rother and Cother stand for the interconnect and output resistances and capacitances, then

$$R_i = \sum_{t \in T} R_t + R_{other}$$
$$C_i = \sum_{t \in T} C_t + C_{other}$$

The resistance of a transistor is inversely proportional to its size Si ; the capacitance directly. So we can get

$$D_i = \left(\sum_{t \in T} \frac{k_1}{S_t} + R_{other} \right) \times \left(\sum_{t \in T} k_2 \times S_t + C_{other} \right)$$

Hence, we can sum all Di to get the equation for the total path delay. More detailed calculation will be showed in PTA part. Furthermore, from this equation, if all S values are bound to single symbolic value S_s , we can derive the delay as follow

$$D_{tot} = k_1 \times S_s + k_2 \times \frac{1}{S_s} + k_3$$

So before some certain point of S_s, the increasing of transistor size will decrease the path delay. However, beyond that certain point of S_s , the increasing of transistor size also increases the path delay[20]. This shows that we can get the minimum path delay on that certain point. Combined with symbolic derivative technique, TranSizor can find the minimum path delay very efficiently.

5.2 TranSizor Overview

Figure 5.2 shows the overview of TranSizor. The input files come from Topolog in SIP format and output files go to Sticks Pack in same SIP format but with sized transistor. In system view, there are four levels in TranSizor. Each level reads the input language which come from previous level and produces output language which feed into next level.

In first level, the input language is defined in SIP (e.g. trans/9). The make sips program (makesips.pl) will generate the unique number for each transistor as its name and add the unmodified status for unsized transistors to build the output language SST (Sip STructure).

In second level, the inputs are (1) SST from first level, (2) state/3 in .pcp file and (3) module/7 in .clean file. The path pre-process program (ppp.pl) will find out the data signal sequence from each state/3 (i.e. cpu cycle) by referring to module/7 and trace all possible paths by matching the signal names. Then, path pre-process program will output PAT(PATh) for next level.

In third level, the input language is PAT (e.g. path/2) with data signal sequence and paths represented in cell format. The prolog timing analyzer program (pta.pl) will calculate the delay of each path by delay model and cell delay accumulation. Then, pta will produce DEL(DELay) language for the next level.

In last level, the input language is DEL(e.g. delay/2) with path information and its delay value. There are two programs in this level - SA program (anneal.pl) and CPH program (heuristic.pl). SA will build the cost function of delay and area, then generate the optimal size of the transistors by SA algorithm based on the cost function. On the other hand, CPH will build the symbolic equation of each path and solve the differential equation to find the size for minimum path delay. However, both of them will produce SZD (sized transistors) language to Sticks Pack.

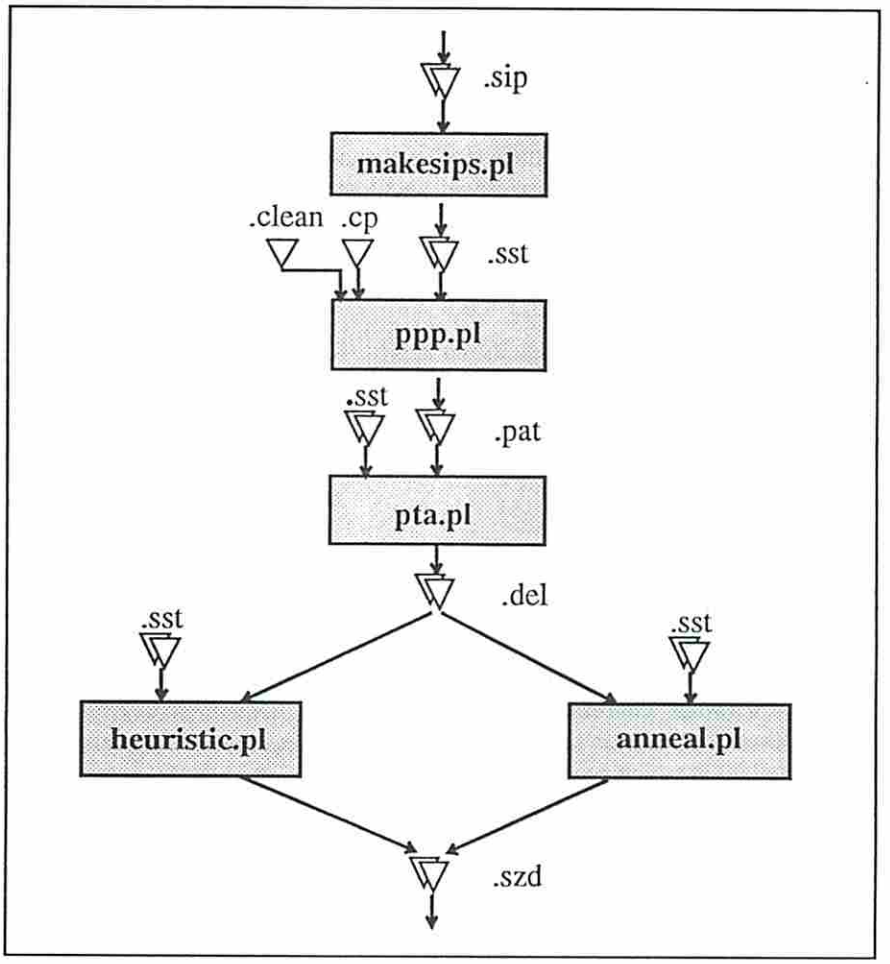


Figure 5.2. TranSizor Overview

5.3 Critical Path Heuristics (CPH)

This method mimic human designer in solving the critical-paths. Given a delay constraint T, designer uses the simulation tool (e.g. SPICE or Crystal) to find out the paths which exceed value T, and then do sizing on those paths to meet the delay constraint T. However, when these path delays decrease below T, some other path delays may exceed

T, so designer needs to re-run the simulation and re-size the transistors. This process will iterate until the maximum delay through the circuit is satisfactory. This basic idea is as follow

```
heuristic(Circuit,Constraint) :-
        analyze(Circuit,CriticalDelay),
        sizing_if_necessary(Circuit,CriticalDelay,Constraint).

sizing_if_necessary(Circuit,CriticalDelay,Constraint) :-
        CriticalDelay <= Constraint. % Successful!

sizing_if_necessary(Circuit,CriticalDelay,Constraint) :-
        Delay > Constraint,
        sizing_by_heuristic(Circuit,NewCircuit),
        heuristic(NewCircuit,Constraint).
```

Two criteria can be applied to heuristic are efficiency and optimality. In a real chip design, absolutely optimum circuit is not required. If a heuristic does give satisfactory results, efficiency becomes important. A major factor in efficiency is the number of timing analyses required, so many of the different approaches are attempts to reduce this number. Actually, avoiding some reanalyses may save lots of execution time and enhance the efficiency.

Heuristics fall into two major categories: transistor-level and critical-path. Transistor-level heuristics deal with one transistor at a time. A timing analysis is performed, and then one transistor is resized. This scheme is very simple and no complicated equations need to be solved. However, the problem of this scheme is the lack of efficiency, particularly if a full timing analysis needs to be performed after sizing each transistor. Some possible way is sizing several transistors before performing another timing analysis, but this will increase the complexity.

Critical-path heuristics mirror human designers' strategies. The critical-path is found, then re-sized the path so that it meets the delay constraint. The process is repeated until all path delays have been reduced sufficiently. This reduces the number of analyses of the circuit, but comes out with some path interaction problems. Most obvious is if a transistors is on two different paths, how to do sizing for it?

5.3.1 Transistor-level Heuristics

There are two questions in this approach: what transistor to resize and how to do the resizing. For first one, there are several ways to choose what transistor to resize. Most easy way is random choice. This is extremely simple, and fast to compute, but gets trouble in an unacceptably large number of timing analyses. Another way is to find the transistor which contributes the most delay and resize it. Once again, thought the computation of this method is not too expensive, it is difficult to say just what "contribute the most delay" means. In the final configuration, some transistors contribute more delay than others, however, in some case, it is better to leave it unsized and resize another transistor instead. We can consider a method in which every transistor contributing more than a specified amount of delay is increased in size, or where every transistor whose resizing would decrease delay is resized.

There is no need to restrict these techniques to a single transistor. More than one transistor can be resized in each pass. Dealing with multiple transistors simultaneously is always advantageous. Very little addition work needs to be done and the number of analyses is almost always reduced. For second question, there are several ways to decide how much to adjust the size of the transistors. The most simple way is increasing the size of the transistor by one unit. This method turns out efficiently.

5.3.2 Critical-Path Heuristics

The basic idea of finding the critical path and then resizing it is simple. The question is how to do the resizing. Two approaches are useful: (1) Numeric solution of the path delay equations for optimal size. (2) A simplified numeric solution, given some assumptions about the eventual sizes. Of course, these methods are more complicated and time-consuming than the ways of sizing individual transistors. From an efficiency standpoint, then, the question is whether

the additional computation done here is compensated by a corresponding decrease in the number of timing analyses required.

The real difficulty in critical-path heuristics is the interaction between different paths. What is to be done if a transistor has had a size assigned to it in one path, and then is a component of another critical path considered later. There are two possible strategies for dealing with this situation. One is to allow each transistor to be sized only once; once it has a size assigned to it, it is fixed This method is quite easy to implement, but sometimes needs special consideration when the transistor is on two critical path simultaneously.

The other method allows transistors to be resized as many times as necessary. However, resizing a transistor affects the delays along paths which have previously been sized, potentially requiring once again sizing those paths. Sizing one path undoes the effect of sizing the other. Even worse, the size of some transistor is increased to reduce the delay of some path, the other path delay may be increased. And then creates the possibility of a loop between two path.

5.3.3 Implementation of CPH

In the implementation of CPH, TranSizor deals with a simple microprocessor chip SM1. Assuming the delay time of memory access is smaller than ALU carry chain delay, then we can focus on the delay of ALU carry chain. There are sixteen bit slice (row) in SM1, it's impractical to find all the paths in ALU cycle and calculate the delays for the paths. However, PTA will find the critical path in each row and accumulate them to build the circuit critical path. So TranSizor will deal with the critical paths in each row and sum the delay of sized critical path delay in each row to achieve the final critical path delay (minimum delay) of the circuit. Furthermore, PTA will produce the symbolic equations for the paths, and then TranSizor will solve the differential equations of critical path to find the sizes for transistors in the path to achieve minimum delay. The detailed algorithm is as follow

1. *Get the Initial Delay of each path from PTA.*

2. *Deal with paths in single row (say row N).*

    2.1. *Build Symbolic Equation for Critical-Path (C.P.) in row N.*

    2.2. *Solve the Differential Equation to find the Size which achieves minimum delay of Critical-Path.*

    2.3. *If exists other path (O.P.) delay > minimum of C.P. delay in row N then solve that path by 2.1 and 2.2.*

        2.3.1. *if minimum of O.P. delay > minimum of C.P. delay then minimum of C.P. delay <- minimum of O.P. delay else get minimum value of C.P. delay in row N.*

3. *Apply same method to each row, find minimum of C.P. delay in each row.*

4. *Add the carry chain delay of ALU to get the minimum value of Critical-Path delay.*

Figure 3 shows the detailed flow chart for CPH algorithm. According to the algorithm, CPH needs to find the critical path first

```
critical_path(DelayList, CriticalPath, CPDelay) :-
        find_critical_by_row(DelayList, CPDelayList),
        get_critical_path(CPDelayList, CriticalPath, CPDelay).
```

where DelayList is sixteen row path delays list produced by PTA. Procedure find_critical_by_row reads in DelayList and calls find_critical_path to extract the CPDelayList composed by critical path in each row. And then procedure get_-

critical_path gets the critical-path and delay value from CPDelayList. The procedure find_critical_path looks like

```
find_critical_path(Row, [HDelay|RowDelays],InCPDelay, RowCPDelay) :-
        compare_delay(HDelay, InCPDelay, TempCPDelay),
        find_critical_path(Row, RowDelays, TempCpDelay, RowCPDelay).

compare_delay(Delay, InDelay, Delay) :-
        Delay > InDelay.

compare_delay(Delay, InDelay, InDelay).
```

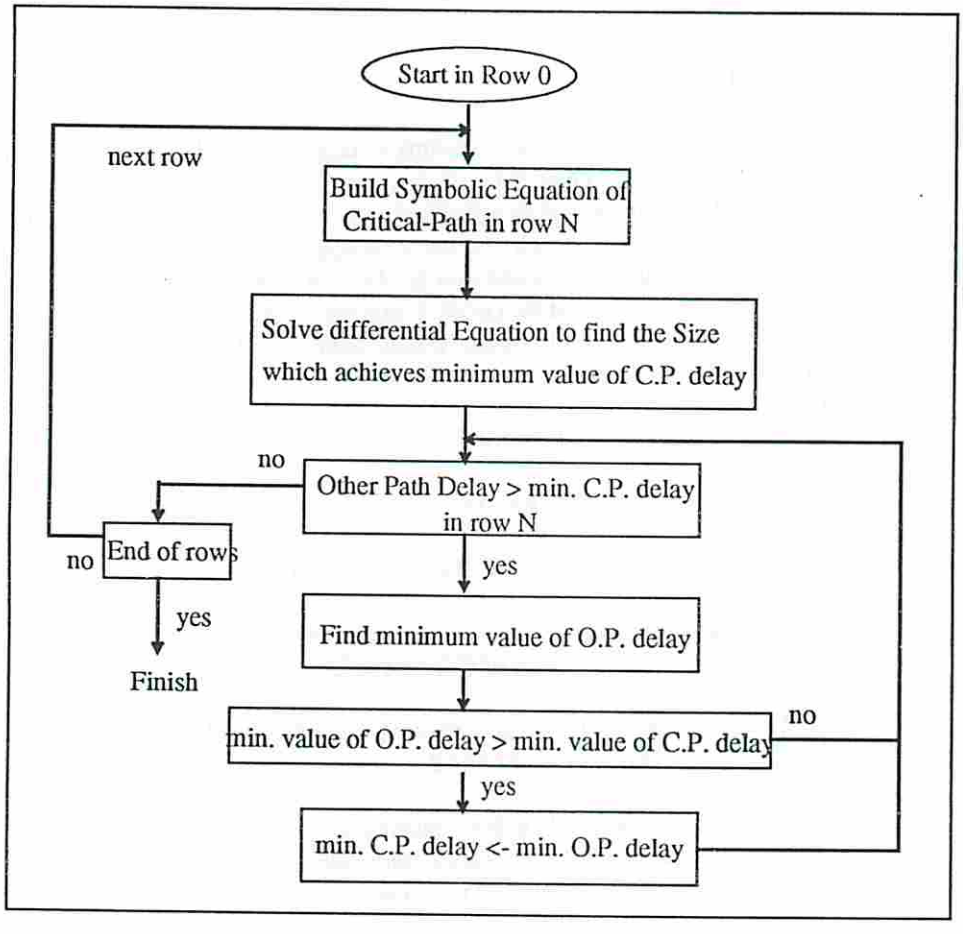


Figure 5.3. CPH algorithm flow chart

When critical path is found, CPH will get the symbolic equation of the path. Next step is solving the differential equation and find the transistor size for the minimum path delay. Actually, from section 5.1.2, the equation is convex function, so we can calculate the path delay when size increases from 1 and find the minimum value of path delay and the certain size of transistors.

```
find_minimum_delay(Eq,InSize,NewSize,CPDelay, MinDelay) :-
        evaluate(Eq,InSize,ThisDelay),
        ThisDelay < CPDelay,
        TempSize is InSize +1,
        find_minimum_delay(Eq,TempSize,NewSize,ThisDelay,MinDelay).
```

```
find_minimum_delay(Eq,Size,Size,Delay, Delay).
```

Then CPH needs to find out all other paths which delay exceeds the minimum delay value of critical path. In each row, there are four categories of paths: (1)the path through the carry of alu, (2) the path from one end to the carry, (3)the path from carry to the end and (4)other paths do not belong to previous three types. Each type needs to be considered when performs the 2.3 part of the CPH algorithm. Furthermore, dealing with the carry related (i.e. from the carry of previous row) paths, the previous row delay needs to be accumulated to the path delays. So CPH algorithm 2.3 can be implemented as

```
sizing_in_row(Sst,NewSst,DList,PreDelay,NewPreDelay,CPDelay,NewCpDelay) :-
        through_delay(Sst,TSst1,DList,CPDelay,RowDelay),
        end_to_delay(TSst1,TSst2,DList,PreDelay,CPDelay,RowDelay,NewPreDelay),
        to_end_delay(TSst2,NSst,DList,PreDelay,CPDelay,NewCPDelay),
        other_delay(DList,NewCPDelay).
```

where the Sst is the original sip structure list, DList is the delay list, PreDelay is the delay of previous row and NSst is the sized sip structure list. Actually the size of transistors are changed in previous three procedures, the last procedure is just for the verification purpose.

```
through_delay(Sst,NewSst,[HD|Delays],CPDelay,RowDelay) :-
        is_through_delay(HD),
        (HD =< CPDelay -> NewDelay is CPDelay, TSst = Sst;
                          solve_this_path(Sst,TSst,HD,NewDelay)),
        through_delay(TSst,NewSst,Delays,NewDelay,RowDelay).

end_to_delay(Sst,NewSst,[HD|Delays],PreDelay,CPDelay,RowDelay,NewPreDelay) :-
        is_end_to_delay(HD),
        TestDelay is PreDelay + CPDelay,
        (HD =< TestDelay -> NewDelay is CPDelay, TSst = Sst;
                          solve_this_path(Sst,TSst,HD,NewDelay)),
        end_to_delay(TSst,NewSst,Delays,PreDelay,CPDelay,NewDelay,NewPreDelay).

to_end_delay(Sst,NewSst,[HD|Delays],PreDelay,CPDelay,NewCPDelay) :-
        is_to_end_delay(HD),
        ThisDelay is HD + PreDelay,
        (ThisDelay =< CPDelay -> NewDelay is CPDelay, TSst = Sst;
                                 solve_this_path(Sst,TSst,HD,NewDelay)),
        to_end_delay(TSst,NewSst,Delays,PreDelay,NewDelay,NewCPDelay).

other_delay([HD|Delays],CPDelay) :-
        (HD < CPDelay -> true;
                         print(' *** Fail in Other Path Delay ***')),
        other_delay(Delays,CPDelay).
```

The procedure solve_this_path gets the symbolic equation of the path and find the minimum delay of that path. So, from CPH algorithm, TranSizor can size the circuit successfully and efficiently.

5.4 Simulated Annealing (SA)

5.4.1 Basic Algorithm

In global view, we can deal with whole transistors in the same time and get the optimal size transistors for the circuit. Simulated annealing algorithm is a good approach for this purpose. In this algorithm, a new configuration may

be accepted even if it increases the cost; the chance of this occurring ids controlled by a parameter called the temperature, which steadily decreases. This prevents getting trapped in a local minimum due to an unfortunate choice of initial configuration.

The SA algorithm divides into an outer loop, which gradually decreases the temperature, and an inner loop, which performs a number of iterations at each temperature. At each iteration, a new configuration is generated, its cost is evaluated, and then the acceptance function determines whether or not the configuration is accepted. Any configuration decreasing the cost will of course be accepted; different acceptance functions give different chances of accepting configurations which increase the cost. The usual acceptance function is

$$exp\,(-\frac{\Delta\ cost}{T})$$

The algorithm looks like

```
anneal(Circuit,Constraint) :-
        initialize(Circuit,Configuration,Delay,Temperature),
        outer_loop(Configuration,Delay,Constraint,Temperature).

outer_loop(Configuration,Delay,Constraint,Temperature) :-
        Delay =< Constraint. % success !!

outer_loop(Configuration,Delay,Constraint,Temperature) :-
        Delay > Constraint,
        interate_at_temperature(Temperature,N),
        inner_loop(Configuration,NewConfig,N,Temperature,Cost,Delay,NewDelay),
        update_temperature(Temperature,NewT),
        outer_loop(NewConfig,NewDelay,Constraint,NewT).

inner_loop(Configuration,NewConfig,N,T,Cost,Delay,NewDelay) :-
        generate(Configuration,TempConfig),
        cost(TempConfig,TempCost,TempDealy),
        accept(TempCost,Cost,T),
        N1 is N - 1,
        inner_loop(TempConfig,NewConfig,N1,T,TempCost,TempDelay,NewDelay).

inner_loop(Configuration,NewConfig,N,T,Cost,Delay,NewDelay) :-
        N1 is N - 1,
        inner_loop(Configuration,NewConfig,N1,T,Cost,Delay,NewDelay)
```

The algorithm can terminate in two condition: (1) it succeeds if delay is reduced below the desired goal(meet the delay constraint and optimum area), and (2) it fails if some failure criterion is met. A reasonable criterion is unable to achieve successful configuration after a certain number of times through the main loop.

A configuration is simply an assignment of sizes to the transistors. New configurations are generated by random perturbations of each size. Usually, using interger(e.g. 1,2 or 4) for perturbations makes the transistor sizes also integer. The cost of a configuration consists of a penalty for exceeding the specified delay, and another term relating to the total size of the transistors. So, we can find the optimal solution between delay and area from this algorithm. Before getting the cost, PTA calculates the path delays for cost function This is the most time consuming part. Many parameters can be varied in an attempt to improve performance. They are the initial temperature and configuration, the rate at which the temperature decreases, the proper number of iterations at a given temperature, the acceptance chance, cost function, perturbation number, and generation procedure. From a carefully consideration, the SA algorithm will be efficient.

5.4.2 Cost Function

A major determining factor in the performance of the algorithm is the cost function. The idea is charging a penalty for a delay exceeding the constraint. If the desire were simply to reduce circuit delay to a minimum, then the penalty could just be the delay. Since the problem is only to meet a specified criteria, no bonus is given for reducing delay below this bound.

Penalty = max(Delay-Constraint, 0)

Secondly, the transistor size also needs to include in the cost function which prevents very large circuits. The cost function may become

Cost = Penalty + TotalSize

Of course, we can weight either delay penalty or totalsize to satisfy the design goal. So cost function becomes

$$Cost = (k_1 \times Penalty) + (k_2 \times TotalSize)$$

The process essentially divides into two steps: first the sizes of the transistors increased as delay is reduced to the constraint, and then the total size component of the cost takes over, and the sizes are gradually reduced. From this process, a satisfactory solution is reached, but rather slowly, since essentially only one critical path is being considered at one time.

Another approach is dealing with multiple paths simultaneously. The improvement is to consider all paths which exceed the delay constraint in the cost function. Assume P stands for the set of all penalty paths, then

$$Cost = (k_1 \times \sum_{i \in P} Penalty_i) + (k_2 \times TotalSize)$$

again, we can choose different value of k1 and k2 to either weight delay penalty or total size However, the most critical path can be weighted more heavily than others if the time delay is the most important factor of design.

5.3 Implementation of SA

The problem domain in SA is still SM1, again, PTA will find the path delays in the same way described in section 4.The detailed SA algorithm looks like

1. *Get an Initial Configuration C.*
2. *Get an Initial Temperature T > 0.*
3. *While not yet Success, do the following.*
   - 3.1. *Perform the following loop N times.*
     - 3.1.1. *Generate a random neighbor C' of C.*
     - 3.1.2. *Let Δ = cost(C') - cost(C).*
     - 3.1.3. *If Δ <= 0 (downhill move), Set C = C' with probability 1.*
     - 3.1.4. *If Δ > 0 (uphill move), Set C = C' with probability exp(-Δ/T).*
   - 3.2. *Set T = rT (reduce Temperature).*
4. *Return C.*

Figure 4 shows the detailed flow chart for SA algorithm. The acceptance chance is defined as *exp(-Δ/T)* and

the cost function is defined as 2*DelayCost+1*SizeCost, where DelayCost is the accumulation of Penalty and SizeCost is the accumulation of width in each transistor. However, weighting more either in DelayCost or SizeCost will get different results.

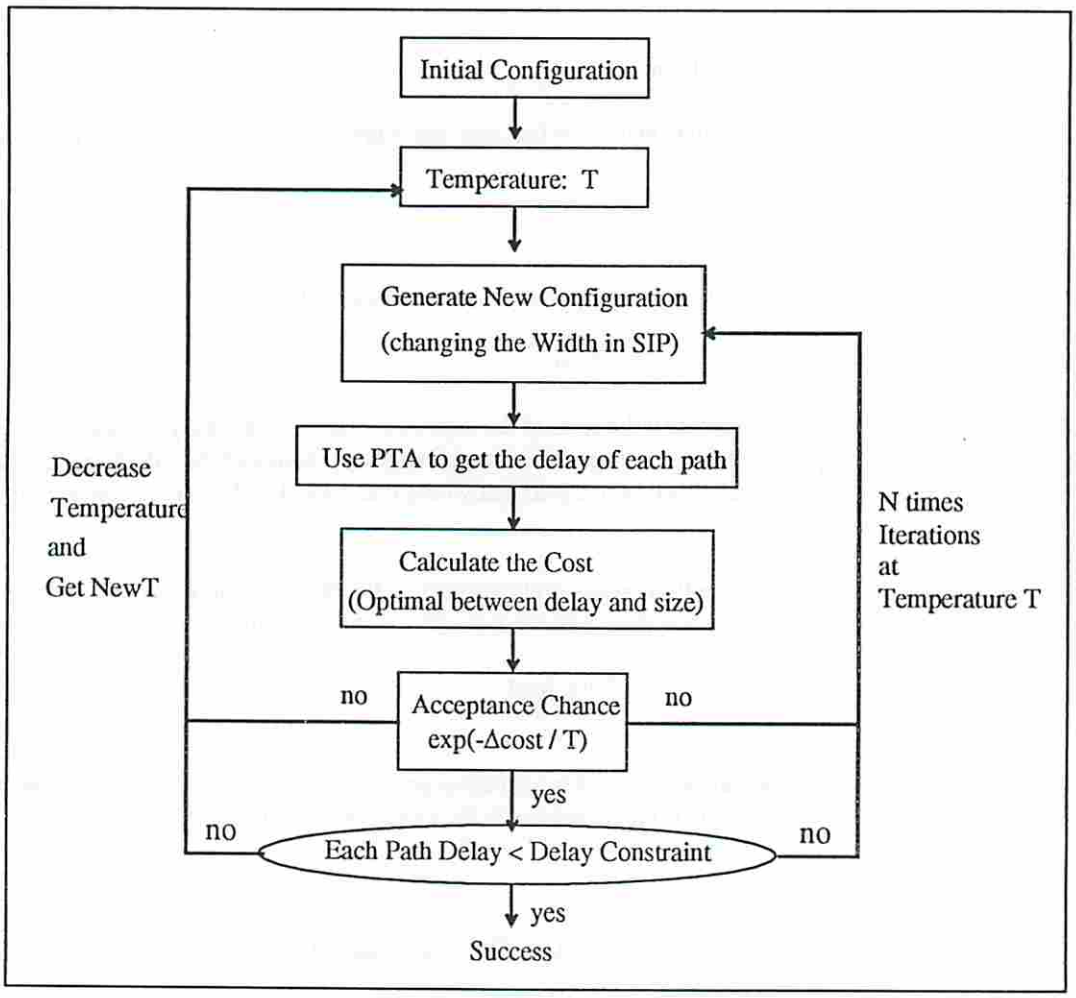


Figure 5.4. SA algorithm flow chart

First, the procedure initialize finds the initial configuration and its cost, and then get the initial temperature as

```
initialize(Sst,DelayList,Constraint,Temperature,Cost) :-
        cost(Sst,DelayList,Constraint,Cost),
        init_temperature(Cost,T).
```

where Temperature is the function of Cost. And procedure Cost is as follow

```
cost(Sst,DelayList,Constraint,Cost) :-
        delay_cost(DelayList,Constraint,DelayCost),
        size_cost(Sst,SizeCost),
        make_cost(DelayCost,SizeCost,Cost).
```

procedure delay_cost accumulates the penalty of all the paths and size_cost sums up all the transistors' width.Then, procedure make_cost uses the equation of 2*DelayCost+1*SizeCost.

Another important procedure is generate which produces new configuration in each time. SA uses a random

function to generate new configuration from accepted configuration.

```
generate([HSstlSsts],[NewHSstlNewSsts]) :-
        minimun_gate_size(Min),
        maximum_gate_size(Max),
        perturbation(Mn,Max,Size+Change),
        NewSize is Size + Change,
        change_size(NewSize,NewHSst),
        generate(Ssts,NewSsts).
```

5.5 Prolog Timing Analyzer(PTA)

PTA is the heart of TranSizor. It provides information for both the CPH and SA top ends. Although it is both slower and somewhat less accurate than Crystal, it still has several interesting features. In particular, for consistency, it is written in Prolog; it is turned to repeated use as part of TranSizor, and thus preprocesses as much of its input as possible; it deals with multiple paths simultaneously and keeps essential information for them; it provides symbolic equations for the delays of the paths; and it builds paths in hierarchical cell structures for easy computation. So, TranSizor uses PTA instead of employing other analyzer like Crystal. Moreover, SM1 is a cpu chip. In each cycle (state), there are some input signals and output signals which needs to provide as input data for Crystal, if Crystal is chosen. This work is tedious and needs to translate signal format at every time. But, on the other hand, PTA can get these information from upper level of ASP system (e.g. Piper) very easily and build the paths for future use, since this preprocessing cost only needs to be paid once.

PTA in TranSizor consists of three major parts. Two of them are preprocessing task (i.e. only need to do once), they are: (1) make sips program, and (2) path pre-process (ppp) program. These two build the paths of the circuit, these path structures will never change during the sizing work, since the modification is only on transistor sizes in each path. Another one is (3) pta program. It calculates the path delays and executes every time on new configuration of the circuit. Following sections discuss detailed implementation of PTA.

5.5.1 Make Sips Program

Make sips program (makesips.pl) gets the input of unsized transistors from .sip files. The format of input is

trans(Type,Spt,Gpt,Dpt,Width,Length,Ssig,Gsig,Dsig).

where Type is n-type or p-type transistor, Width is the transistor width (will be changed finally), Length is the transistor length(keep unchanged). For further processing of transistors, make sips program gives each transistor a unique name. In CPH algorithm, TranSizor allows transistor to be resized only once. So a status field is appended for distinguishing modified transistors from unsized transistors. However, in SA algorithm, TranSizor allows unlimited modification, so this field is kept unchanged. And the format of output Sst is the list of sip/3 as

sip(Name,trans(Type,Spt,Gpt,Dpt,Width,Length,Ssig,Gsig,Dsig),Status).

where (1)Name is [Sequent number in Row, Sequent Row Number], (2)Unsized Transistor is the same as input, (3)Status stands for the modification status and 0 stands for unmodified status. This item will be changed if the size is changed. Figure 5.5.1 shows the input and output examples for make sips program.

5.5.2 Path Pre-Process (PPP) Program

Path Pre-Process program (ppp.pl) is important part of PTA. It builds the paths in cell format for PTA further processing. ASP system deals with ISA microprocessor, TranSizor needs to get more information from Piper and Topolog. In .pcp file produced by Piper, state stands for one cpu cycle. For example, moving data from accumulator, through bus to output register is a state. PPP traces all possible paths in every single state. However, it's possible there

are more than one move in a state. Even in a parallel processing, PPP must find all the move information to build the paths. Furthermore, in .clean file from Topolog, module stands for each element module for CPU, just like accumulator, ALU, ...etc. PPP processes each state, combining the mapping of module for data signal information, to get data signal sequences for tracing the paths. After getting data signal sequences, PPP traces the data signal in SST list and constructs the paths in cell format.

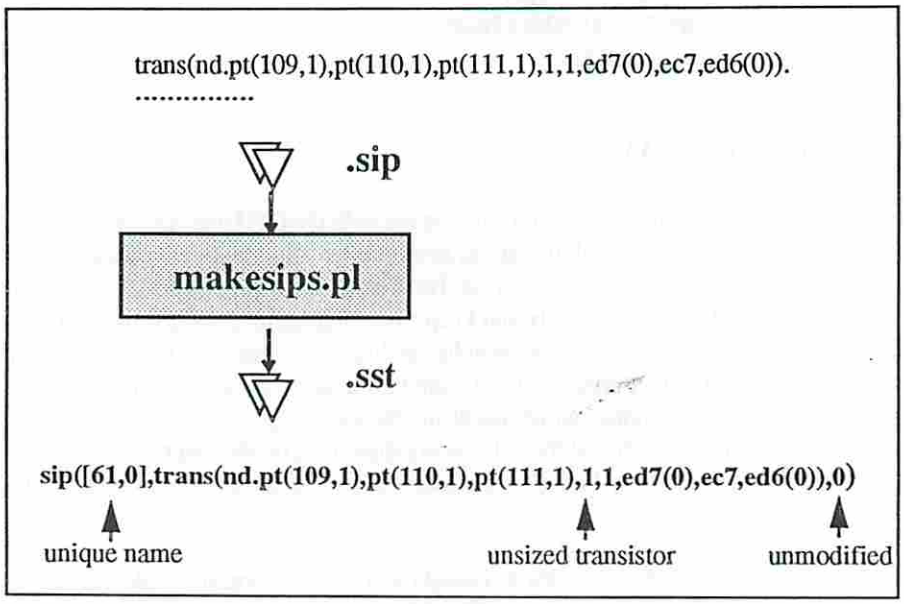


Figure 5.5.1 Overview of make sips program.

The input of PPP program is SST (list of sip/3) from make sips program. Besides, the list of state/3 from .pcp file and the list of module/7 from .clean file are also the inputs as

[sip([0,0], trans(...), 0), sip([1,0],trans(...),0), sip(...), ...].
[state(...,[move(ac,bus(2),outputReg)],...),state(...,[move(...),...), ...].
[module(ts,ts(pc,bus(1)),15-0,[ed1],[ed2],[ec1],[]), module(...), ...].

And the output Pat is a list of path/2, the format looks like

path([ed6,ed7,ed8], [cell([[61,0],[69,0],inverter([62,0],[63,0])]),cell([inverter([64,0],[65,0])]),
cell([[70,0],inverter([74,0],[75,0])]), cell([inverter([76,0],[77,0])])]).

Figure 5.5.2 shows the inputs and output of PPP program. The PPP program divides into three small level. The first sub-level produces the possible paths in one state in the format of PathInfo (the list of pathInfo/3). The second sub-level reads in PathInfo from first sub-level and produces output in the format of CombinePath (the list of path/2). Then, the third sub-level reads in CombinePath from second sub-level and produces the output in the format of Path (the list of path/2 again but in cell format). This translation can be written as

1-sub-level -> [pathInfo(ed6,ed7,[[61,0]]), pathInfo(ed7,ed8,[...]), ...].
2-sub-level -> [path([ed6,ed7,ed8],[[61,0],[69,0],...]), path([...],[...]),...].
3-sub-level -> [path([ed6,ed7,ed8],[cell([[61,0],[69,0],...]), cell([...]),...]), path([...],[...]), ...].

First Sub-Level

The first sub-level also divides into two main procedures: one is make_signals which makes the data signals sequence and related control signals, another one is make_possible_paths which produces all possible pathInfo

between two data signals by referring the data signals and control signals.

```
process_state(Sst,[state(_,Moves,_)|States], InitPathInfo,PathInfo) :-
        make_signals(Moves,DataSigs,ControlSigs),
        make_possible_paths(Sst,DataSigs,ControlSigs,TempPathInfo),
        process_state(Sst,States, TempPathInfo,PathInfo).
```

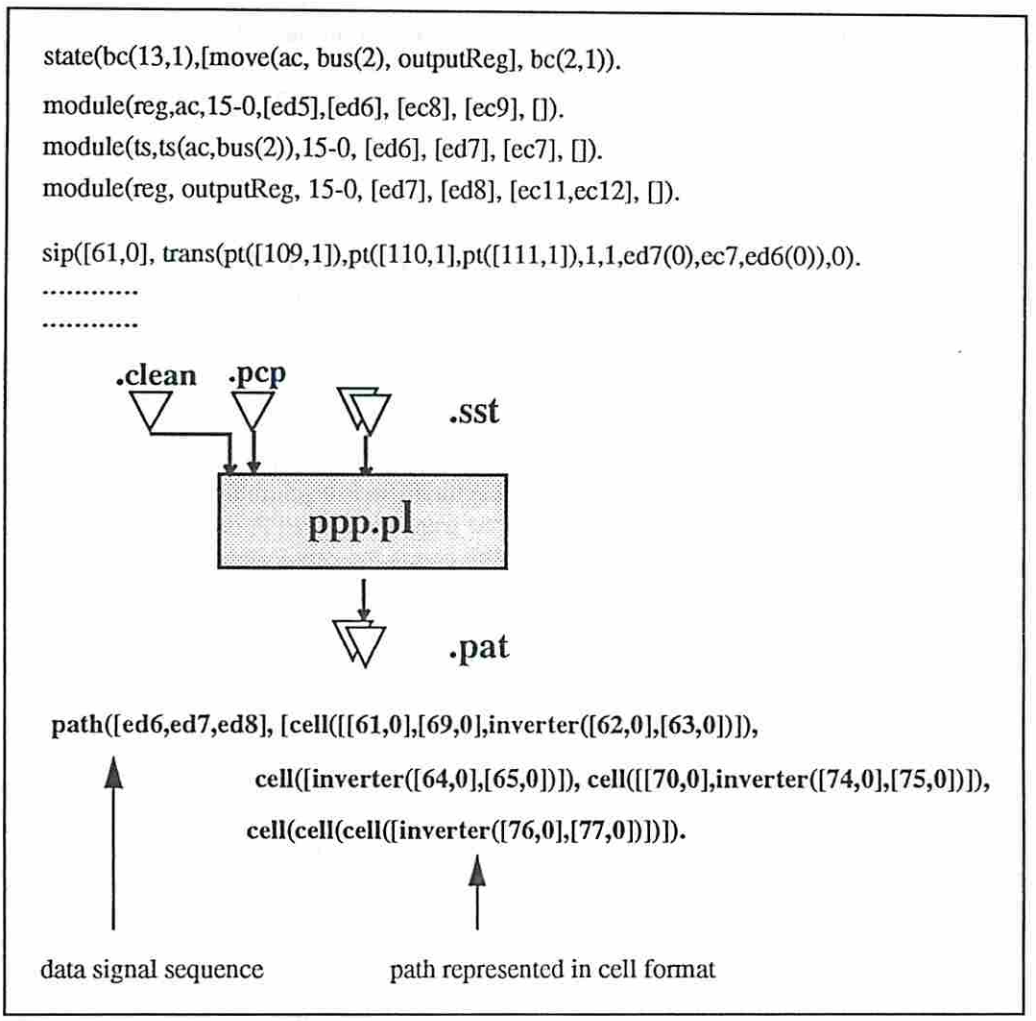


Figure 5.5.2. Overview of PPP program.

In each state, procedure make_signals extracts the move information and maps the information into the element modules. Then, the data signal sequence and the related control signals of the state are available for next procedure. For example, from figure 5.5.2, the state is moving data from accumulator(ac) through bus to output register (outputReg). Then, mapping module ac, module ts and module outputReg gets the data signal sequence [ed6,ed7,ed8] and active control signals [ec7,ec11,ec12] of this state.

After getting the data signal sequence and related control signals of the state, procedure make_possible_paths finds out all possible path information(pathInfo) between any two consecutive data signals.

```
make_possible_paths(Sst,[D1,D2|DataSigs],PathInfo) :-
        trace_paths(Sst,D1,D2,PInfo),
        make_possible_paths(Sst,[D2|DataSigs],[PInfo|PathInfo]).
```

The trace of PathInfo is depending on signal name matching in SIP format. For example, the data signal sequence [ed6,ed7,ed8] needs to referring the SIP information below

sip([61,0],trans(nd,pt(109,1),pt(110,1),pt(111,1),1,1,**ed7(0),ec7,ed6(0)**)).

sip([69,0],trans(nd,pt(124,1),pt(125,1),pt(126,1),1,1,**n19(0),ec11,ed7(0)**)).

sip([62,0],trans(nd,pt(112,1),pt(113,1),pt(114,1),1,1,vss,**n19(0),int0at112no2**)).
sip([63,0],trans(pd,pt(114,20),pt(113,20),pt(112,20),1,1,vdd,**n19(0),int0at112no2**)).
sip([64,0],trans(nd,pt(115,1),pt(116,1),pt(117,1),1,1,vss,**int0at112no2,n20(0)**)).
sip([65,0],trans(pd,pt(115,20),pt(116,20),pt(117,20),1,1,vdd,**int0at112no2,n20(0)**)).

sip([70,0],trans(nd,pt(127,1),pt(128,1),pt(129,1),1,1,**n22(0),ec12,n20(0)**)).

sip([74,0],trans(nd,pt(136,1),pt(137,1),pt(138,1),1,1,vss,**n22(0),int0at136no2**)).
sip([75,0],trans(pd,pt(138,20),pt(137,20),pt(136,20),1,1,vdd,**n22(0),int0at136no2**)).
sip([76,0],trans(nd,pt(139,1),pt(140,1),pt(141,1),1,1,vss,**int0at136no2,ed8(0)**)).
sip([77,0],trans(pd,pt(139,20),pt(140,20),pt(141,20),1,1,vdd,**int0at136no2,ed8(0)**))

From the bold characters, pathInfo(ed6,ed7,[[61,0]]) is produced and pathInfo(ed7,ed8,[.....]) is also produced from the signal name matching. The figure 5.5.3shows the path of [ed6,ed7,ed8]. The data signal may come from drain to source or reverse (there is no obvious direction concept for drain and source) for a pass transistor or the data signal will come to the gate side (e.g. inverter or nand gate). In MOS circuit, there is no current from gate to source or drain. Thus, the circuit can be divided into several cells if we cut it in the gate points. In PPP, it is smart enough to detect the gate input transistors and separates them from other transistors by giving the new format of gate(Name) or inverter(-Name1,Name2). The output of this sub-level becomes the list of pathInfo between two data signals as

[pathInfo(ed6,ed7,[[61,0]]), pathInfo(ed7,ed8,[[69,0], inverter([62,0],[63,0]),inverter([64,0],[65,0]),[70,0], inverter([74,0],[75,0]),inverter([76,0],[77,0])]), pathInfo(...), ...]

The tracing for path information is time consuming, especially when the number of transistors increase, the executing time will increase exponentially. Some improvement is made in PPP: (1) if the signal to gate is 1 for p-type transistor or 0 for n-type transistor, the gate is an open-gate. The program doesn't need to trace through it. This scheme can be implemented by getting the related active control signals from procedure make_signals. (2) The tracing of path never goes through the same transistor twice to avoid the infinite loop. This can be achieved by simply recording the traced transistors. (3) The tracing of path ends either in second data signal or Vdd/GND. When PPP traces to Vdd or GND, this path is terminated without tracing from Vdd/GND to other signals.

It's difficult to trace all the path information in ALU carry chain. A more reasonable way is separated the path information by bit slice (row). In this scheme, the PPP program needs to detect the carry signals and then trace the path not only from one data signal to the other but from one data signal to carry out, from carry in to carry out and from carry in to the other data signal.

Second Sub-Level

The second sub-level reads in PathInfo(pathInfo/3), then combines the pathInfo to build the real paths by matching the data signal sequence again. (e.g. from [ed6,ed7] to [ed7,ed8]). Of course, ALU data signal sequence needs special attention just like in the first sub-level. Procedure combine_paths produces the output CombinePath (path/2) which is ready to feed into next sub-level.

[path([ed6,ed7,ed8], [[61,0],[69,0],inverter([62,0],[63,0]),inverter([64,0],[65,0]),[70,0], inverter([74,0],[75,0]),inverter([76,0],[77,0])]), path([...],[...]), ...].

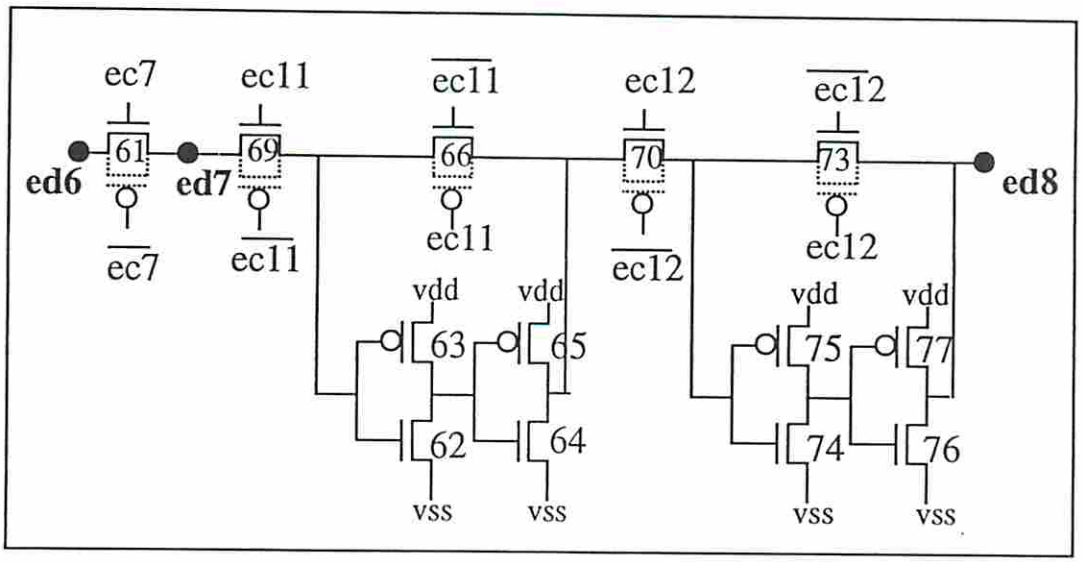


Figure 5.5.3 The path of [ed6,ed7,ed8].

Third Sub-Level

Again, this sub-level reads in CombinePath from previous sub-level and produces the output paths in cell format (path/2). The rule employed in this sub-level is the MOS circuit characteristics which say that the path can be cut into several cells in the points of inverter or gate because there is no current from gate to drain or source in MOS[22]. The procedure cell_paths parses the path list to single path and the procedure cell_single_path cuts the path into cell representation. The final output of PPP program is the list of paths in cell format

[path([ed6,ed7,ed8], [cell([[61,0],[69,0],inverter([62,0],[63,0])]), cell([inverter([64,0],[65,0])]),
cell([[70,0],inverter([74,0],[75,0])]), cell([inverter([76,0],[77,0])])]), path([...],[...]), ...].

5.5.3 PTA Program

PTA program iterates several times when CPH algorithm and SA algorithms are running. The efficiency of it is even more important than previous two pre-processing tasks. When transistor sizes change, PTA gets involved for the calculation of the path delays. PTA reads in the paths and produces the delays for the paths (path/2). Figure 5.5.4 shows the input and output of PTA.

The delay model currently used for transistors is the lumped RC model, which views the entire resistance and capacitance of a stage as concentrated at the end of the stage[7][8][9][10][11]. This model is not very accurate, however, the gain is speed. Within ASP, TranSizor is meant to be run before layout takes place. This implies that the exact lengths of the interconnections are not known, and so some estimates have to be made on the parasitic resistances and capacitances. Since the uncertainty of the parasitics limits the accuracy of any delay computations, there is no point in spending extra effort to arrive at similarly inaccurate results.

Figure 5.5.5 shows the lumped RC model in one basic cell. For transistor T, the delay is

$$Dt = Biggerdelay(In, Tin) + (Rx + \sum_{i \in I} Ri + Rin) \times (Cx + \sum_{i \in I} Ci + \sum_{f > t} Cf + Cout)$$

where 'I' is interconnections, 'Cout' is drive capacitive load, 'Rin' is the input resistance and '>' means 'follow in the path'. Then, the delay of this basic cell becomes

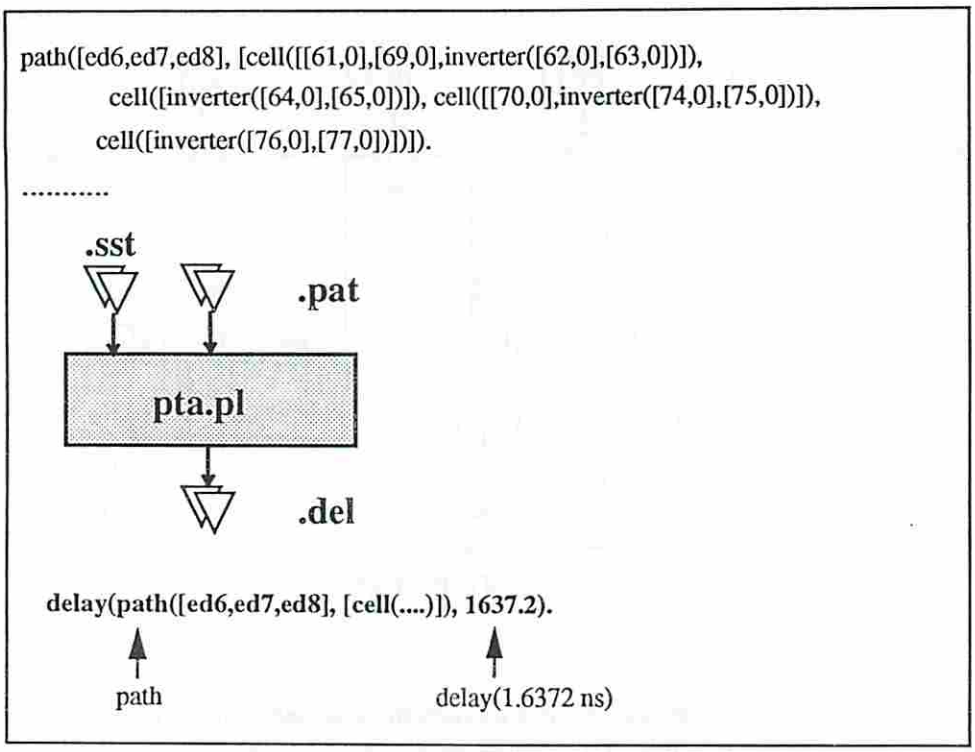


Figure 5.5.4 Overview of PTA program

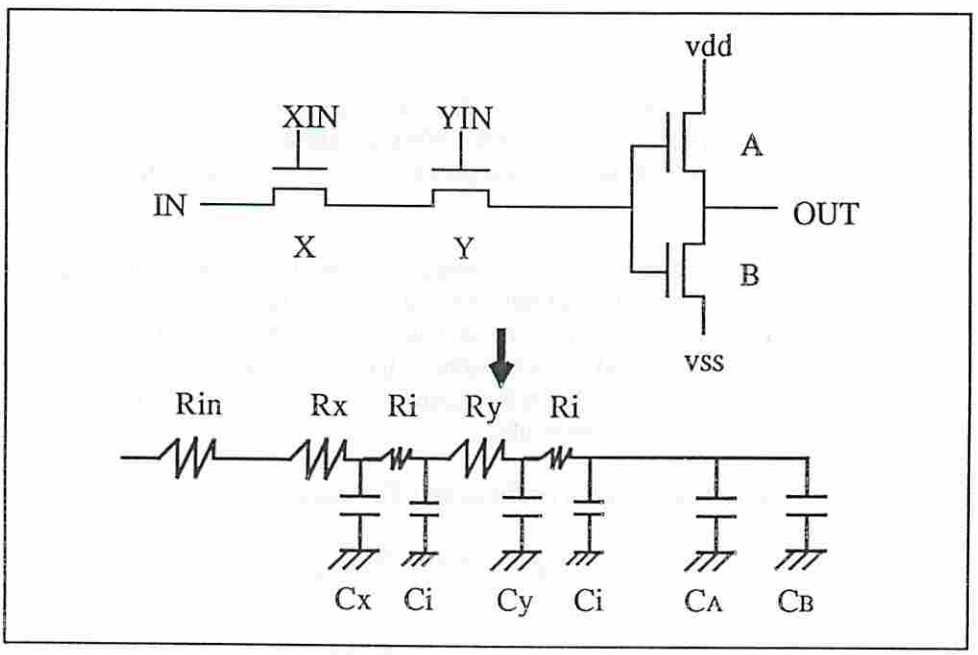


Figure 5.5.5. Lumped RC model (one cell graph)

Dcell = Bigger_delay (Dx, Dy).

*Dx = Bigger_delay(In, Xin) + (Rx+Ri+Ry+Ri+Rin) * (Cx+Ci+Cy+Ci+Cout)*
*Dy = Bigger_delay(In+dx,Yin) + (Ry+Ri+Rin+Rx+Ri) * (Cy+Ci+Cout)*

where Cout = CA + CB and dx is negligible. When all the cell delays are available, PTA accumulate the cell delays in a single path to make the path delay. The procedure process_single_path looks like

```
process_single_path(Sst,[cell(Cell)|Cells],PreCell,PreDelay,Delay) :-
        get_cell_delay(Sst,Cell,PreCell, CellDelay),
        process_single_path(Sst,Cells,Cell,CellDelay,Delay)
```

where PreCell is used for input resistance and PreDelay is delay accumulation before this cell.

As described in PPP section, TranSizor finds all possible in single bit slice(row), but not crossing between bit slices(rows). So PTA uses some technique to deal with the ALU carry chain which crosses between the bit slices. Figure 10 is a simple example for PTA to obtain the critical path for whole cpu circuit. Assuming there are sixteen bit slices and carry chain looks like figure 5.5.6. First, find the longest path delay from Cin0, A and B to Cout0 in bit slice 0 (row 0). And then this delay (Cout0) becomes Cin1 delay in row 1 (the second bit slice). At this time, Cin1 may be the crit ical one comparing to A and B. Applying this method till row 15 (the last row), PTA can find the critical path delay of the carry chain circuit, which is the output signal S15 delay.

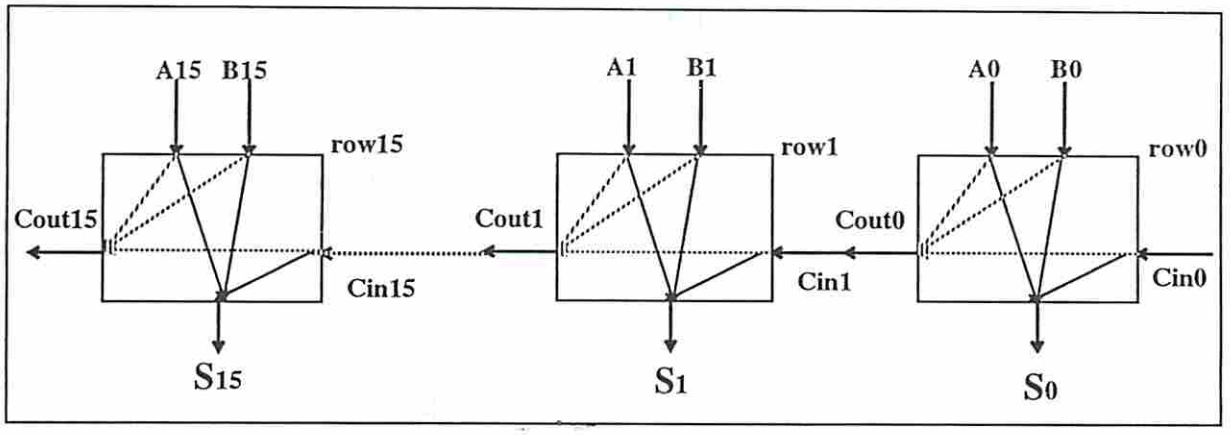


Figure 5.5.6. Delay in ALU carry chain.

5.6 Results

TranSizor deals with SM1 (Instruction Set Architecture Microprocessor) which has about 3000 transistors.- First, PTA finds the critical path of whole circuit in ALU carry chain. The delay of critical path is about 32 nsec.(32.425 ns). This path goes from bit slice 0 (row 0) to bit slice 15 (row 15) as

[cell([[149,0],gate([86,0])]),cell([[90,0],[94,0],inverter([83,0],[84,0])]),
cell ([gate([88,1])]),cell([[90,1],[92,1],inverter([83,1],[84,1])]),.....,
cell([[90,14],[92,14], inverter([83,14],[84,14])]),
cell([gate([106,15])]),cell([[110,15],[116,15],[122,15], [126,15],inverter([180,15],[181,15])]),
cell([[167,15],gate([40,15])]),cell([[42,15], inverter([35,15],[36,15])]),
cell([[45,15],inverter([49,15],[50,15])]),cell([inverter ([51,15],[52,15])]),
cell([[53,15],inverter([57,15],[58,15])]),cell([inverter([59,15], [60,15])])].

Secondly, TranSizor uses critical-path heuristic (CPH) algorithm and simulated annealing (SA) algorithm for sizing SM1. The results are listed below. The comparison of these two algorithms are focused on (1) delay reduction, (2) size increase and (3) CPU time.

Critical-Path Heuristic (solving the symbolic equations)

Delay Reduction 25% (to 24.464 ns)
Size Increase less than 10%
CPU Time 4055 second (1 hour and 8 minutes)

Simulated Annealing (cost function is 2*DelayCost + 1*SizeCost)

(1) Maximum Perturbation is 1
Delay Reduction 14% (to 27.983 ns, delay constraint is 28 ns)
Size Increase 23% (from 2900 to 3575)
CPU Time 26293 second (7 hours and 18 minutes)

(2) Maximum Perturbation is 2
Delay Reduction 15% (to 27.435 ns, delay constraint is 28 ns)
Size Increase 56% (from 2900 to 4546)
CPU Time 12016 second (3 hours and 20 minutes)

(3) Maximum Perturbation is 4
Delay Reduction 14% (to 27.981 ns, delay constraint is 28 ns)
Size Increase 62% (from 2900 to 4702)
CPU Time 21960 second (6 hours and 6 minutes)

where the CPU time is based on Sun 4-460 workstation and the codes are written in Prolog, and the results of SA algorithm are based on the average values of three run in each different perturbation value (i.e. 1,2,4).

5.7 Conclusion

From the results of section 7, CPH algorithm performs better than SA algorithm not only in delay reduction, size increase but CPU time. Particularly, using symbolic equations, the CPH algorithm can achieve 25% enhancement in about one hour for 3000 transistors. In lumped RC delay model, this is the best delay reduction we can achieve. Moreover, the size increase of final circuit is less than 10%. This makes CPH algorithm more promising.

The SA algorithm's performance is also acceptable. Especially, when maximum perturbation is 2, the CPU time is three hours and 20 minutes and delay reduction is up to 15%. But in the other two cases, the CPU times are too long to be practical. While dealing with larger circuits, the SA algorithm may need more time in execution.

In ASP design automation system which deals with ISA microprocessor, TranSizor provides CPH algorithm and SA algorithm for sizing The user can select either algorithm to satisfy the design goal. However, from the result, TranSizor prefers CPH algorithm.

5.8 Future Work

Combine CPH and SA

One promising area of exploration is the integration of SA algorithm with CPH algorithm for sizing. The CPH can generate an acceptable solution and then rely on SA to find the proper solution. More simply, CPH may be used to give a starting configuration for SA and, finally, a post-processing CPH may be used to improve the solution generated by SA. Such a post-processing CPH might be useful in reducing the transistor sizes. Since area minimization is

less important than reducing the delay to the constraint, transistors not on the critical path tend to be larger than they need to be. Detecting and then examining these transistors is helpful to reduce the transistor sizes from SA.

Accurate models of Delay, Area and Power

PTA's accuracy can be improved by incorporating the distributed RC model and taking waveform shape into account. This model is only slightly more complex than the lumped RC model, and the sizing techniques should continue to perform much the same. The gains will, of course, be limited by the fact that interconnect lengths are only estimates in TranSizor of ASP design automation system, but should make PTA's accuracy competitive to other timing analyzers. Furthermore, while TranSizor considers the area and power dissipation, more accurate models are also necessary.

Combine Sizing with Placement, Routing and Compaction

The drawback of estimates of interconnect length can be solved by considering sizing together with placement, routing and compaction. This means to combine the TranSizor with Topolog and Sticks Pack in ASP system. First, do rough sizing before placement and routing, so placement and routing program (Topolog) can also consider the delay of wires as a factor[5]. Second, perform the sizing but make more accurate estimate of the parasitic resistance and capacitance. After compaction taking place, TranSizor can calculate the accurate delay, area and power dissipation of the VLSI circuit. If the circuit can't meet the user spec and constraint, we need to iterate this procedure until success.

5.9 Reference

[1] Aspians, "The Advanced Silicon-compiler in Prolog (ASP II)", 1990.
[2] J. Pincus, "Transistor Sizing", UCB/CSD 86/285, 1986.
[3] R. Brayton and A. Sangiovanni-Vincentelli "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, 1984.
[4] R. Brayton and A. Sangiovanni-Vincentelli "A Survey of Optimization Techniques for Integrated-Circuit Design", Proceedings of IEEE, 1981.
[5] T. Hu and E. Kuh "Theory and Concepts of Circuit Layout" in "VLSI Circuit Layout: Theory and Design", IEEE Press, 1985
[6] N. Weiner and A. Sangiovanni-Vincentelli "Timing Analysis In A Logic Synthesis Environment ", 26th IEEE DAC, 1989.
[7] B. Hoppe, "Optimization of High-Speed CMOS Logic Circuits with Analytical Models for Signal Delay, Chip Area, and Dynamic Power Dissipation",IEEE Trans. on CAD, 1990
[8] F. Obermeier "An Open Architecture for Improving VLSI Circuit Performance", UC Berkeley PhD thesis, 1989.
[9] F. Obermeier and R. Katz " An Electrical Optimizer that considers Physical Layout" , 25th IEEE DAC,1988
[10] D. Marple " Performance Optimization of Digital VLSI Circuits" , Stanford Univ. PhD thesis, 1986.
[11] D. Marple " Transistor Size Optimization in the Tailor Layout System ", 26th IEEE DAC,1989.
[12] J. Ousterhout "Crystal : A Timing Analyzer for nMOS VLSI Circuits", 1985.
[13] J. Ousterhout "Switch-Level Delay Models for Digital MOS VLSI", 21th IEEE DAC, 1984.
[14] W. Kao "Algorithms for Automatic Transistor Sizing in CMOS Digital Circuits", 22nd IEEE DAC,1985.
[15] L. Glasser "Delay and Power Optimization in VLSI Circuits ", 21st IEEE DAC, 1984.
[16] C. Lee "An Algorithm for CMOS Timing and Area Optimization ", IEEE Solid State Circuits, 1984
[17] M. Matson "Macromodeling of Digital VLSI Circuits ", 22nd IEEE DAC, 1985.
[18] M. Matson "Optimization of Digital VLSI Circuits", Proc. Chapel Hill Conf. on VLSI, 1985.
[19] J. Fishburn "TILOS: A Posynomial Approach to Transistor Sizing", IEEE ICCAD, 1985.
[20] K. Hedlund "Aesop: A Tool for Automated Transistor Sizing", 24th IEEE DAC, 1987
[21] M. Cirit "Transistor Sizing in CMOS Circuits ", 24th IEEE DAC, 1987.
[22] M. Hofmann "Delay Optimization of Combination Static CMOS Logic ", 24th IEEE DAC, 1987.
[23] C. Mead and L. Conway "Introduction to VLSI System ", Addison Wesley.
[24] L. Glasser "The Design and Analysis of VLSI Circuits ", Addison Wesley.
[25] N. Weste "Principles of CMOS VLSI Design ", Addison Wesley.

TranSizor:

A Transistor Sizing program in ASP II.

Hsu-tsun Chen

Abstract

This work focuses on the problem of automated optimization of VLSI CMOS microprocessor circuit designed by the ASP design automation system. TranSizor runs after placement and routing and before the compaction in ASP. Several methods of choosing optimal sizes of transistors considering both delay and area are discussed, especially the Critical-Path Heuristic algorithm and Simulated Annealing algorithm which are using in this sizing program alternatively. TranSizor uses PTA timing analyzer to provide the delay information to implement these two approaches. The performance of unsized circuit can be enhanced about 25% in an hour execution of TranSizor. Using the simple lumped RC model and symbolic equations, Critical-Path Heuristic method performs more efficiently than Simulated Annealing method not only in more delay reduction but less size increase.

6. STICKS-PACK

In this section we present Sticks-Pack, a design environment for VLSI circuit layout generation written exclusively in Prolog. Not only does Prolog provide a relational database for VLSI objects, it also provides a syntax well suited for expressing both algorithm and rules.It takes in the output from Topolog which is a bunch of sip (Stick in Prolog) files of all the macro-module of the chips and output a CIF of the whole chip. Sticks-Pack has the following tools which are integrated together to form the system:

- a technological independent compactor;
- a power/ground wire size estimator;
- a joiner that joins together cells generated by the compactor;
- a global placer;
- a global router;
- a channel router;
- a pad-frame generator;

6.1 The System

Current layout systems are composed of programs that have been written independently of each other. This often results in a duplication of work and a need for conversion of format between programs. Sticks-Pack does not have this problem since it is designed to be an integrated system. For example, while spacing the elements from a cell file, the compactor saves all the elements on the border of the cell into a border file for the joiner/ pitchmatcher. The joiner can then space/pitchmatch the cells properly without again searching through each cell for border elements which have to be joined between cells. This is in contrast to other systems where the compactors is written independently of the joiner. Also a language which is a subset of Prolog is used to represent the design objects and hence all the programs have the same base to operate.

Previous approaches to integrated VLSI design system were generally based upon conventional programming languages using custom data managers with strict data format([1], [2]). Objects in these data managers can only be generated or accessed through a fixed data field. It is very inflexible and sometime inefficient to access certain data objects according to different requirement. By using relation database inherent in Prolog, SP allows generation of design object with any arbitrary number of data fields. Furthermore, individual data field may be represented by objects. This allows special fields to be parameterized. Also objects that have special relationship(e.g in the same net or in the same layer) can be grouped and accessed much easily. This gives the CAD designer a simple but powerful method of accessing data.

6.2 The Compactor

The SP compactor takes in a cell defined in the Sticks in Prolog (SIP) language and creates a mask level representation (CIF) for the cell. Virtual grid symbolic layout [3] is used as the backbone of the abstract representation of the cell element. Every element is placed in a virtual grid first without actual physical dimension between the grids. The compactors tries to put the elements as close as possible without violating the design rule and then assigning actual physical distance to each virtual grid. The compactor employs a one-dimensional two-pass compaction approach. An algorithm similar to zone refinement [4] is used to get the rough spacing of the elements. For each compaction pass, the elements lies on a specific virtual grid is compared with those compacted which form the 'floor' or

'fence'. The elements are then moved directly across the 'molten region' to the floor where the space is minimized without violated the spacing requirement. Diagonal constraints are resolved during the space check phase.

The structure of the compactor program is shown in figure 2.1.

function call	Function	Files
addsubcon	add substrate contact	addsubcon.pl
netex	net extraction	cnetex.pl
wireJoin	joining of wires which are connected to the same net	coverwire.pl
cmp	main compaction call	lists.pl
xcompact	x-direction compaction	lists.pl
xcoordex	x-direction forward compaction	lists.pl
updatefence	update the fence stru.	lists.pl
getrow	compact row by row	lists.pl
xcoordexb	x-direction backward compaction	lists.pl
(similar to xcoordex)		
ycompact	y-direction compaction	lists.pl
(similar to xcompact)		
genbox	generate box for each element according to the physical dimension got from compaction	expand.pl
expand	write out in cif format doing well generation	expand.pl

Figure 6.1 structure of the compactor

The main program of the compactor is "compact". It first reads in the .sip file of the module that needs to be compacted. Then it calls "addsubcon" to generate the substrate contact for those transistor contacts that are connected to Vdd or Vss. Substrate contact are added to minimize the latch-up effect which will cause damage to the chip due to electrostatic charge. Then "netex" is called to extract all the elements that are connected to the same net and assign the same netname to the elements that connected to the same net. Next step is to call "wireJoin" to join the overlapping wires and adds pseudo 'contacts' to provide a flat surface underneath the vias.

The main compaction routine is inside the "cmp" program which calls the "xcompact" and "ycompact" to compact the module in both x direction and y direction respectively. Each compaction takes two passes, one forward and one backward. For each pass the following algorithm is use. A fence structure which represents the physical distance mapping of each virtual grid is built and initialized. The fence structure has 4 layers of element to consider and has the following form:

fence(p([poly_fence]),dif([diffusion_fence]),m1([metal-1 fence]),m2([metal-2 fence]))

After the initial fence is built, the module is compacted row by row. The elements of each row is retrieved by the "getrow" function. Each element in the row is compared to those relevant elements in the current fence to determine the minimum spacing of the element to the existing fence. The maximum spacing obtained from the comparison determine the actual physical location of the grid. After one row is processed, the fence structure is updated by adding the elements of the row just processed and removing those elements which will not affect the spacing of the remaining uncompacted row.

The calculation of the distance that an element has to be placed is illustrated in figure 2.2.

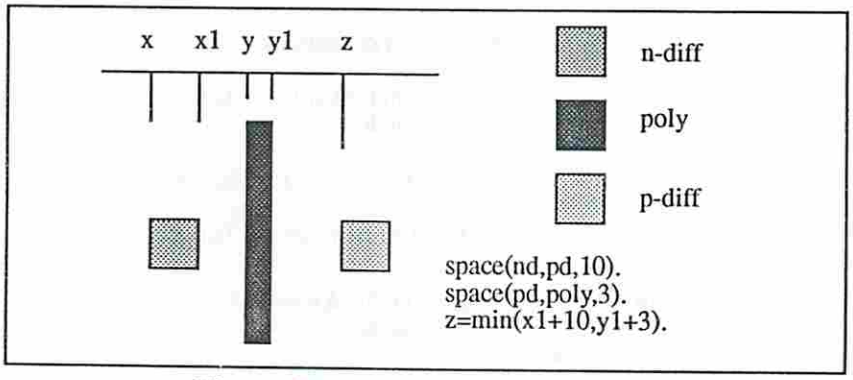


Figure 6.2 calculation of spacing

The compactor is technological independent since all the design space rules are all included in a design rule check file and is consulted in separately. If different technology is used, we just consult in the relevant design rule check file. Also by treating each layout element as an object, the compactor can easily interpret new layout objects such as bipolar transistor elements to suit mixed technology process. Example of the technology file is shown in figure 2.3.

6.3 The Joiner

Large layouts in SP are realized by joining small cells together with the joiner. Examples are joining the sixteen bits slices of the datapath to form a full datapath and joining the PLA with the control registers to form a full control path. Leaf cells are compacted individually and are the building blocks for larger modules. When joining the leaves cell together it is much better to abut them rather than calling a router to route the signals between them because smaller area will be resulted. This can be realized since the pins that needed to be connected between two leaves are all laid on the opposite side and are in right order. Two methods can be used in joining cells, pitchmatching and river routing. Pitchmatching cause expansion in one axis while river routing cause expansion in the other. Previous joiners have either exclusively pitchmatching or river routing. The joiner here connects signals between a given pair of cells by either pitchmatching or river routing, whichever is more area efficient. The joiner operates in the

```
% this is drc.pl

% MOSIS rules (in 2 units per mosis unit).

% Width rules (half of actual)

width(nd, 2).
width(ndtrans, 3).
width(pdtrans, 3).
width(pd, 2).
width(p, 2).
width(m2, 3).
width(m1, 3).
width(m1m2, 4).
width(m1p, 4).
width(m1nd, 4).
width(m1pd, 4).
% Spacing rules (full distances)

maxspace(m1, 12).
maxspace(m2, 16).
maxspace(pd, 48).
maxspace(nd, 48).
maxspace(pdtrans, 48).
maxspace(ndtrans, 48).
maxspace(p, 12).
maxspace(m1m2, 16).
maxspace(m1p, 12).
space(L1,L2,Space):-
 subspace(L1,L2,Space);
 subspace(L2,L1,Space), !.

subspace(m1m2, m1nd, 10):- !.
subspace(m1m2, m1pd, 10):- !.
subspace(m1p, p,6):- !.
subspace(m1m2, flat, 2):- !.
subspace(m1p, pdtrans, 8):- !.
subspace(pdtrans, m1p, 4):- !.
subspace(pdtrans, m1nwell, 6):- !.
subspace(ndtrans, m1p, 8):- !.
subspace(m1p, ndtrans, 4):- !.
subspace(m1m2, nd, 6):- !.
subspace(m1m2, pd, 6):- !
```

Figure 6.3 example of drc file

physical domain rather than the virtual grid domain for tighter results. This also allows cells of various virtual grid heights and widths to be joined.

The pitchmatching program is included in the file pitchmatch2.pl. It first read in the .space, sip,and the .term files of the cells that are going to pitchmatch. The order of the pins that are going to be connected must be the same for the two sides that are going to be joined. The physical space for each pair of pins that are going to be joined are matched. The actual physical space to be stretched for each cell is calculated. If the stretching due to the matching of a certain pair of pins exceed an estimated cost constraints, the pin pairs will not be matched. Instead they will be connected by the router which will be called after the pitchmatching phase. After all possible pairs are matched, the space and term files are updated. The pitchmatcher is capable to identify equivalent locations of a pin so that when matching a pairs of pins, it will select the optimum pin locations such that the stretching of cells will be minimized.

6.4 Power and Ground Wire Sizer

During the Topolog synthesis, there is no idea on the power consumption of each gates. Transizer is used to size the transistor to minimize the timing for the critical path. After all the transistors' sizes are determined, the power and ground wire size has to be re-determined in order to satisfy the voltage drop requirement and current density requirement. If the current density is too high electromigration effect may occur. First the total current drawn from the power and ground are estimated and the then the size of the wire is estimated according to this. The current is estimated by counting the number of p-transistor and n-transistor connected to Vdd and Vss respectively. Then the current drawn by each transistor is estimated according to its size. The total current is obtained by adding up current drawn by all transistors. This figure is too conservative since it assumes all transistors are on and drawn maximum current. An utilization factor is applied on this figure to obtain a more reasonable maximum current. Empirically we are using 0.5.

The above program is contained in the file 'pgwiring.pl'. The main function call is:-

```
updateallpowerbus(Name,S,E):-
    getcurrent(Name,0),     % get the current of the power and ground wire
    updatebussize(Name,1,E),!. % update the size of the bus
        "getcurrent" calls the "findcurrent " to find the current:
findcurrent(Vtype,Ttype,Current):-
   bagof([W,L],(Pt1,Pt2,Pt3,A,B)^trans(Ttype,Pt1,Pt2,Pt3,W,L,Vtype,A,B),List),
   findcurrent1(Ttype,List,0,Current),!.

findcurrent1(_,[],C,C).
findcurrent1(Type,[H\T],OldCurrent,NewCurrent):-
   H =[W,L],
   Ratio is W/L,
    current_for_min_size(Type,MinCur),
   C1 is MinCur * Ratio,
  TempCurrent is OldCurrent + C1,
  findcurrent1(Type,T,TempCurrent,NewCurrent),!.
```

6.5 Global Placer

After cells are compacted and joined to form larger modules, the larger modules have to be placed in the core and connected. The objective for the global placement is to place all modules in a way such that the total core area and routing between modules are minimized. Other criteria such as minimized the routing length of some criteria signal such as clock can be added to constrain the placement and routing phase.

Since the number of the modules generated is not very larger under the current design style of ASP, we are

adopting a simple but powerful placement algorithm. Basically it is a constructive placement algorithm. It chooses the best on among those not yet placed and placed in the best location according to some estimator function. The program is written in a way that we can plug in different estimator function for different placement strategy. Current we are using the area of the core and total routing wire length as the estimator.

The modules are placed in a way such that a slicing structure is maintained[5]. Slicing structure is preferable because it facilitate the later routing stage. Channel definition and channel ordering [6],[7],[8]are made easier. Also re-routing is needed due to routing area congestion because for a slicing structure, there is an optimal channel routing order that the modules can be shifted in the case of the routing region is not large enough without redoing the previous routing.

The basic procedures of the Placer is shown in figure 2.4.

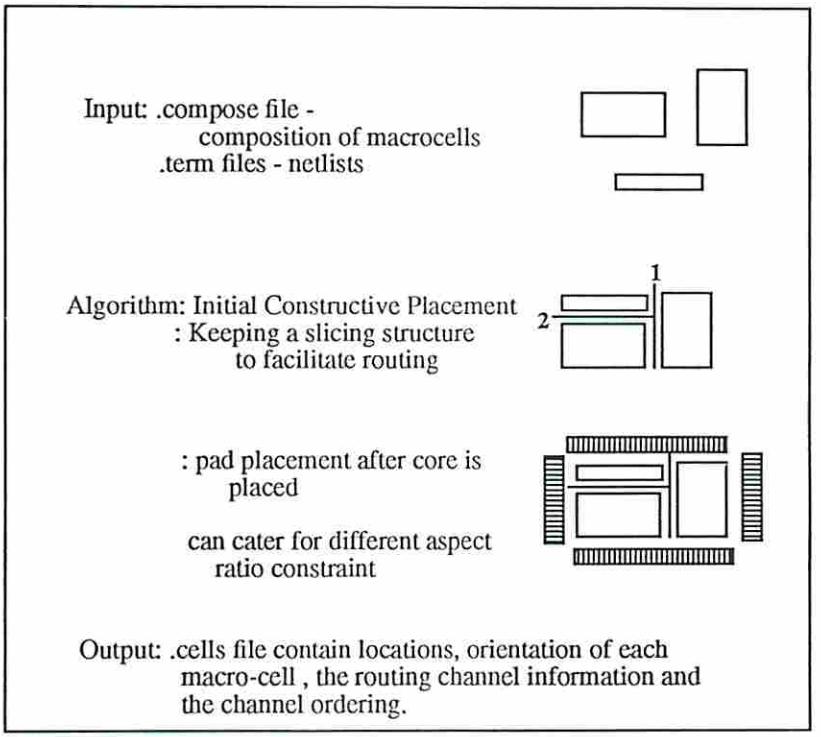


Figure 6.4 Basic procedures of the Placer

The main program "place" is contained in the file 'newplacer.pl'. It calls two major programs: "placer" which does the placement work and "channe_define" which does the channel definition and channel ordering work. The "placer" program is as follows:-

```
placer(OldSel,Rest,Placed,OldPlaced,NewPlaced,OldPossLoc,NewPossLoc):-
        selmodtoplace(OldSel,Rest,Placed,Sel),
        placemod(OldPlaced,TempPlaced,Rest,TempRest,Sel,OldPossLoc,TempPossLoc),
        placer(Sel,TempRest,NewP,TempPlaced,NewPlaced,TempPossLoc,NewPossLoc),!
```

selmodtoplace chooses from Rest which is the set of unplaced modules the next module (Sel) to place. It uses an force-directed algorithm is used. The resultant force obtained from subtracting the attractive forces from the rest of unplaced modules from the attractive forces from the placed module is calculated for each potential module to be selected. The one that has the maximum force is selected. The force is simply modelled by the no.of connection between modules.

The program placemod is used to place the selected module. It selects a best location to place, then check whether the new placement will result in any overlapping or non-slicing structure, if not, try next best location. After the module is placed, all the location of the placed modules are updated and new possible locations to place a new module are generated. New positions are generated at those corners resulted from placement.

```
placemod(OldPlaced,NewPlaced,OldRest,NewRest,Sel,OldPossLocs,NewPossLocs):-
        findloc(Sel,OldPlaced,OldPossLocs,Loc,Orientn,TempPlaced),
        updatePart(TempPlaced,OldRest,Sel,Loc,Orientn,NewPlaced,NewRest),
        updatePossLoc(NewPlaced,OldPossLocs,Sel,Loc,NewPossLocs,Orientn),!.
```

The best location is determined by the estimator function which is now based on the area of the bounding box of the placed cells and the total connection wire length.

```
findloc(Sel,Placed,PossLocs,Loc,Orient,NewPlaced):-
        findbestloc(Sel,Placed,PossLocs,[[[],999999,999999,[0,0,0]]],ListofLoc)
        checkoverlap(Sel,ListofLoc,TempOrient,Placed), % check cells overlapping
        checkslice(Sel,ListofLoc,TempOrient,Placed,NewPlaced),!. % check slicing structure
```

After all the modules are placed, the channel_define is called to do the channel definition and channel ordering. In addition the pads placement is done in this stage. After the channel is ordered, the channel graph is built to facilitate the later global routing stage.

```
channel_define(Name):-
        padorcorebound(Mode), % decide whether the chip is core bound mode or pad bound mode
        place_ext_pin(Name,Mode),  % place the pads in accordance with the mode
        channel_order(Chan_order,Ch_no), % order the channel
        form_channel(Name,Chan_order,0,Ch_no,[],NewChan,Cell_list,_Newcell_list,Mode), %
                        % finding the details of the channels e.g. pins on the two sides of the channel,
                        % the modules and their corresponding sides that form the channel.
        build_channelgraph(NewChan,X_chan,Y_chan,[],AdjaGraph),!. % building the channel graph.
```

The way to order the channel is shown in figure 2.5 and an example of the channel graph is shown in figure 6.2.

6.6 Router

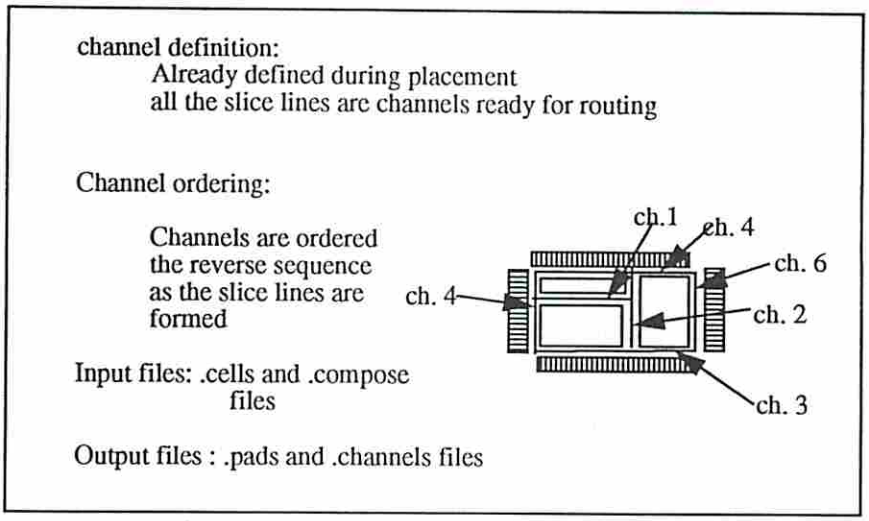


Figure 6.5 Channel ordering

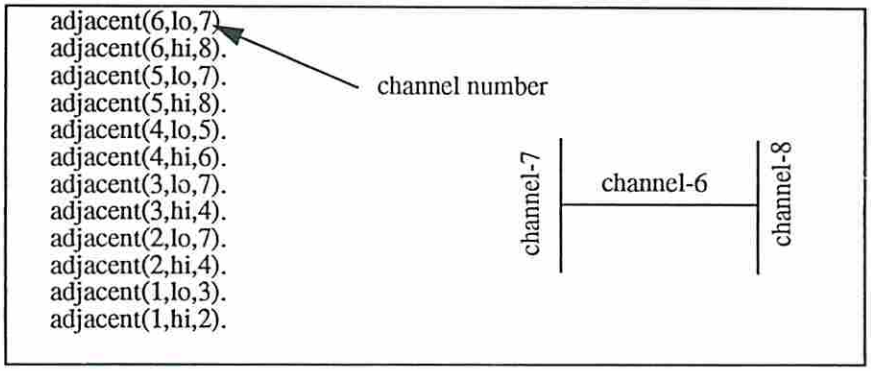


Figure 6.6 Channel graph

Since the modules are placed in a slicing structure, channel definition is not necessary. Channel ordering is easy to obtained by reversing the slicing order of the slicing structure[7]. Global routing is done first by finding the shortest path of each net. Steiner tree is formed to find the shortest path for each path. Then channel assignment is done for each net.

The global router program is stored in 'globroute.pl' and the main program is "main". The program is shown in the following:

```
main(Cell):-
        readcells(Cell),
        allnodes(Nodelist, Eqs),   % find out all nets to be routed, multipoint nets and equivalent nets
                                   % are identified so that no redundant routing will be resulted
        tell(debugs),
        routenets(Nodelist, List),  % net routing
```

```
routeqs(Eqs, List, Flist),   % routing of the equivalent nets
sides(Flist),                % make sidenets for each channels
told,
makerfiles,
make_cells(1, no, Cell), !.  % call the detail channel router to route each channels and finally
                             % produce the ultimate supercell
```

Breadth first search is used to find the shortest path in the net routing phase :-

```
% assigns the channels travelled through for a single net. Input
% is a list of channels the net endpoints lie in, and output is
% all the channels the nets have to pass through.

routenet([Channel],ChanAssign):-
        ChanAssign = [Channel].

routenet([Head|RestChanList],ChanAssign):-
        routenet(RestChanList,TmpAssign),
        bfs([[Head]],TmpAssign,Path),
        (var(Path)->
        This = [];
        This = Path),
        append(This,TmpAssign,ChanAssign).
```

After global routing is done, the detailed router is called to route each channel according to the channel order. Channel router is employed for the detailed router. Left edge greedy algorithm is used for the channel router.

6.7 Pad frame synthesis

The Pad frame synthesis has two phases: pad placement and pad frame generation. After the core of the chip is completed, the pads are placed according the final location of the pins that the pads are connected to. The pad placement algorithm is an assignment problem algorithm. An estimator based on the estimated wire length is used to give an approximate goodness figure for assigning a pad to one side of the chip. The problem is solved by finding an assignment of each pad to one of the sides which has an overall shortest connection wire length. Power and ground pads are inserted between pads at a ratio of 1:8.

After the pads are placed, the pad frame generator is called to generate the pad frame by retrieving the corresponding primitive pad cells from the pad library according to the pad placement and joining them together to form a pad. The 4 pad frames are connected together to give an inner power and ground ring for the power distribution. The padframe generator is also divided into two phases: the first phase is the generation of the abstract pad frame. It identifies the type of the pads and get the appropriate pad from the pad library. Figure 2.7 shows a list of pads that are cur-

rently supported in the library. The final location of the pads cannot be known until the actual dimension on the core is known, for metal has to be generated to create the power and ground ring if the chip is core bound. The actual size of the core is known after all the internal channels are routed. After the core routing is finished, the pad-frame generator is called to generate the physical pad frames. The final phase will then be the routing of the four peripheral channels.

The padframe generator program is stored in the makeframe.pl files and the main program is called create-padframe.

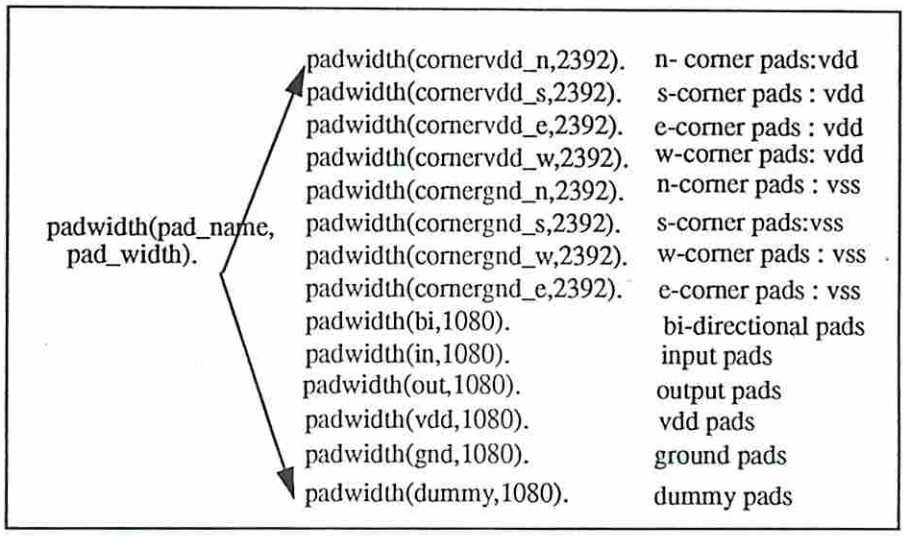


Figure 6.7 example. of pad library

6.8 The Design Environment

There are many characteristics of CAD elements that make them difficult to represent in a database[9],[10]. Each element has many features that associate it with other elements. For example, a wire may be related to other wires by nod, by layer, and by location. A CAD tool should be able to select elements by any features as well as assign new features and relations.

A VLSI database must:

- Provide a method for representing objects and structures as well as relations between objects.
- Provide an abstraction to allow the user to access data efficiently without burdening the user with details of operation.
- Interface well with the programming environment. The programming language must be powerful enough to manipulate data efficiently.

The relational database inherent in Prolog is well suited for meeting these requirements.

6.8.1 Prolog as a Database

To model the many complex CAD structures as well as the relationships between structures, many CAD environments use object oriented databases. CAD elements, whether they be geometry for a compactor, transition states for a simulator, or logic expressions for a logic minimizer, can all be expressed in terms of objects. Relationships between the elements can be expressed in terms of groups. For example, elements in a cell can be grouped by node or by location as well as layer. Current object oriented databases for CAD have strict set relations. For example. many databases categorize wires by layer but not location. To find wires of the same layer, one simply calls a generator that returns instances of wars that are of the queried layer. But to find wires of the same grid, one cannot simply generate wires based upon the grid information, but must generate wires by layer and filter out the wires that are not of a common grid. Data in Prolog can be linked by both structure and value. Thus the procedure fro generating all wires on the metal-1 layer is the same as the procedure for generating all wires on row 5, or wires of node vdd, or all wires of row 5 and node vdd in metal-1. Data can also be store in structures (such as binary trees or sorted lists) for faster access. These constructs provide the ASP Prolog database with a flexible syntax. Elements ranging from behavioral descriptions to logic equations to an ALU layout are all directly expressed in and referenced through Prolog.

6.8.2 Sticks in Prolog

Sticks in Prolog (SIP) is a grid based sticks representation in Prolog that supports hierarchy and parameterized elements. Module generators or human designers generate SIP files which are converted to mask geometry by the Sticks-Pack compactor. In SIP, VLSI elements are modeled as facts. Attributes for the elements are represented as atoms within the facts. There are four types of facts in SIP:

wire(Layer,pt(X1,Y1),pt(X2,Y2),Width,Netname).

cont(Type,pt(X1,Y1),Size,Netname).

trans(Type,pt(Xs,Ys),pt(Xg,Yg),pt(Xd,Yd),Width,Length,Sorucename,
Gatename,Drainname).

pin(Direction,pt(X1,Y1),Layer,Width,Netname,Netname,Element).

Layer can be one of the atoms *m1,m2,p,pd,nd.* These represent the physical layers of the element (metal-1, metal-2, poly, p-diffusion, n-diffusion).

Contact types can be one of the atoms *m1m2, m1p, m1pd, m1nd, m1pdnwell, m1ndpwell* (metal-1-to-metal2, metal-1-to-poly, metal-1-to-pdiff, metal-1-to-ndiff, n-substrate, p-substrate).

Width, Length, and *X* and *Y* coordinates are integers. *pt(X,Y)* represents a point location at virtual grid (X,Y). The Netname field is an atom that is the node name of the element, representing its connectivity. Elements of the same node are electrically connected. Nodal information is supplied by the cell generator or can be extracted by a net extractor.

Transistors have 3 point locations, one for the source, one for the gate, and one for the drain. They also have three nodes, one for each terminal.

The *Element* field for pins contains the element that the pin is attached to.

For example the following defines an inverter in SIP:

wire(m1,pt(0,0),pt(0,5),2,vdd).
wire(m1,pt(0,1),pt(2,1),2,vdd).

```
wire(m1,pt(10,0),pt(10,5),2,vss).
wire(m1,pt(10,1),pt(8,1),2,vss).
wire(m1,pt(8,3),pt(2,3),2,out).
wire(m1,pt(6,3),pt(6,5),2,out).
wire(p,pt(8,2),pt(2,2),1,in).
wire(p,pt(6,0),pt(6,2),1,in).
trans(nd,pt(2,1),pt(2,2),pt(2,3),4,2, vdd, in out).
trans(nd,pt(8,1),pt(8,2),pt(8,3),2,2, vss, in out).
cont(m1pd, pt(2,1), nof, vdd).
cont(m1nd, pt(8,1), nof, vss).
cont(m1pd, pt(2,3), nof, out).
cont(m1nd, pt(8,3), nof, out).
```

6.8.3 Prolog Programming for CAD

There has been a growing trend in CAD to develop tools that use both algorithmic and rule-based programming styles [11],[12]. Algorithms are generally fast, but are inefficient at handling problems that have many special cases. Rule-based systems area well suited for solving problems with many special cases or problems that are not well defined. Rule-based systems have generally been slow. The rules must be looked up and efficient management systems have not yet been developed. Some CAD problems have algorithmic solutions (such as simulation), but most of them are computationally expensive (such as routing and logic minimization), and can be solved by a host of approximation techniques, including rule-based heuristics.

Prolog provides an environment for both algorithmic and rule-based programming styles. Its clausal nature allows rules to be easily updated or modified. Algorithms can be expressed quickly and easily, which makes Prolog an ideal language for rapid prototyping.

A current philosophy in CAD systems is to develop CAD tools that are "technology independent" or "technology insensitive". Tools have been developed with information regarding technology expressed as a set of parameters, with the data for a certain technology loaded from a technology file. Because of this, the tools have not been able to utilize fully benefits that certain technologies have to offer. For example, in the automatic generation of random logic metal-1 and metal-2 vias are expensive in area. In certain technologies, the area of diffusion between two series transistors is an ideal site for the via as metal-1 and metal-2 are both routable to the site and the spacing between the transistors is about the same as the size of the via. Such a condition is difficult to express algorithmically and is very technology dependent, but would be useful in minimizing area. The SP compactor supplements its set of technology parameters with a set of rules that allow the compactor to compact cells more tightly.

Ref.

[1] 'Data Management and Graphics Editing in the Berkeley Design Environment'; D. Harrison, P. Moore, R. Spickelmeier, A.R. Newton; Proceedings of the IEEE International Conference on CAD, November, 1986

[2] 'A Symbolic Design System for Integrated Circuits'; K.H. Kellar, A.R. Newton; 19th Design Automation Conference, 1982

[3] 'Virtual Grid Symbolic Layout', N. Weste, 18th Design Automation Conference, 1981, pp. 225-233

[4] 'Two Dimensional Compaction by Zone Refining'; H. Shin, A. Sangiovanni-Vincentelli; 23rd Design Automation Conference, June 1986

[5] 'Automatic Floorplan Design', R. Otten; 19th Design Automation Conference, 1982, pp261-267

[6] 'Routing Region Definition and Ordering Scheme for Building Block Layout', W.M. Dai, T. Asano, and E.S. Kuh; IEEE Transaction on Computer-Aided Design 3/3 (1984).

[7] VLSI Placement and Global Routing Using Simulated Annealing,C. Sechen,1988 Kluwer Academic Publishers

[8] 'Order of Channels for Safe Routing and Optimal Compaction of Routing Area', U. Kajitani, IEEE Transaction on Computer-Aided-Design 2 (1983)

[9] New Features for a Relational Database System to Support Computer Aided Design'; A. Guttman; Ph.D. thesis, U.C. Berkeley, 1984

[10] ' A Database Approach for Managing VLSI Design Data'; R.H. Katz; 19th Design Automation Conference, 1982

[11] 'An Expert System Paradigm for Design'; F.D. Breweer, D.D. Gajski; 23rd Design Automation Conference June 1986

[12] 'Rule Based and Algorithmic Approach fro Logic Synthesis'; T.Yoshimura, S. Goto; IEEE International Conference on CAD, November 1986

7. CPGEN

The basic architecture of the ASP-synthesized chips has two major parts: the datapath and the control path. CPGEN (stands for Control Path Generator) is a tool to generate the control from the high level description down to a PLA layout in CIF format. In addition to a CIF file, an interface file which contains all the connection of control signals from the control path to the datapath is produced.

7.1 Control Path Design Strategy

Instead of viewing the control path as a finite state machine with a combinational logic part and state register, we introduce a counter concept here. The states of the machine are encoded in the time cycle that the machine is currently at. So the output of the machine which depends on the input and the states are now depended on the input and the time cycle. The time cycle is determined by the no. of clock cycle for each instruction to complete. The no. of cycle depends on several factors, the specification of the instruction and whether there is any pipelineing and how deep the pipeline is. So implicitly the state of the machine is specified by the content of the instruction register and the specific time cycle the machine is at. The following illustrates how this works:

counter(4).

boolean(stage(1),enable(pc,load),[[controlReg1-jump,clock-4],[controlReg1-jumpid1,clock-4],[controlReg1-jumpid2,clock-4]]).

The above specifies each time cycle lasts for 4 clock cycles and the signal enable(pc,load) is on when the time cycle is at the 4th cycle and the instruction register content is either jump, jumpid1 or jumpid2.

One of the advantages of this scheme is that during implementation, we don't need to have a separate state register and also the input to the PLA is much smaller for a complex control path which will have hundreds of states. This will in turn reduce the area of the control path. But there is a disadvantages of this implementation. When we detect branching and jump to a new instruction, we have to wait until the time clock count back to one. If we have a deep state graph and a lot of branching, the performance of the chip will be degraded a lot.

7.2 Pipelining structure control strategy

For a pipeline structure, there are two types of control path generation scheme that we can use: Centralized and Distributed. For centralized scheme, we are using a single controller for the whole chip while for distributed control scheme, basically we partition the control path into pieces in accordance with the no. of pipelines we have, i.e. each pipeline stage is controlled by its own controller. Communication between different stages is implemented by signal passing between different controller. Connection between different stages is specified in the .cpext file as follows:

connect(controlRegister(stage(1)controlReg1),controlRegister(stage(2),
controlReg1)).

This specifies the output of the control register-1 of stage(1) is connected to the input of the control register-1 of stage(2). Connection between datapath and control path is specified as follows:

dpcp(field(memDR(1),opcode),bus(1),15-8,controlRegister(stage(1),controlReg1)).

This specifies the memory data register's opcode field is connected to the control register-1 of stage(1) control path through the *bus(1)*.

Distributed control scheme is usually more efficient than centralized control when the controller is implemented by PLA. In ASP we are using distributed control in our pipeline controller synthesis.

For distributed control, the interface between the datapath and control path becomes a little bit complicated. The control signal driving the module in the datapath, instead is come from a single controller, may now be driven by separate controller. For those signal lines which do not have conflicts (i.e. although they are driven by different controllers, the timing of the activation by the controllers is not at the same cycle), a OR-gate is required to OR all those signal lines from different controller. (WIRED-OR is not feasible since we are using C-MOS technology). But for those that have conflicts, a priority circuit is required to distinguish which controller should have the priority to drive that particular signal. Piper identifies those signals that need a priority circuit and specify it in the.cpext file. CPgen will then call a PCgen(priority circuit generator) to generate it. The specification of a priority circuit is as follows:

module(priority,priority(d,pc),in(2)/out(2),[control(stage(2),enable(pc,load)),control(stage(1),enable(pc,inc))],[enable(pc,inc),control(priority(d,pc),enable(pc,load))],[],[]).

7.3 Extracting information from Piper output

The control information from Piper is in the form of Boolean equation. The form of the Boolean Equation is in the form as follows:

boolean(Stage_no,Output_signal_name,Or_term_List).

The Stage_no states explicitly which pipeline stage this boolean equation is related to. Output_signal_name is the name of the control signal that should be activated with the Or_term. Or_term_List is a list of or_terms which individually contains a list of and _terms. The format of Or_term_List is as follows:

[[controlReg1-jump,clock-4],[controlReg1-jumpid1,clock-4],[controlReg1-jumpid2,clock-4]]

control reg1=jump ^ clock at 4th cycle

7.4 Overview of CPGEN

The overview of CPGEN is illustrated by figure 1.1:

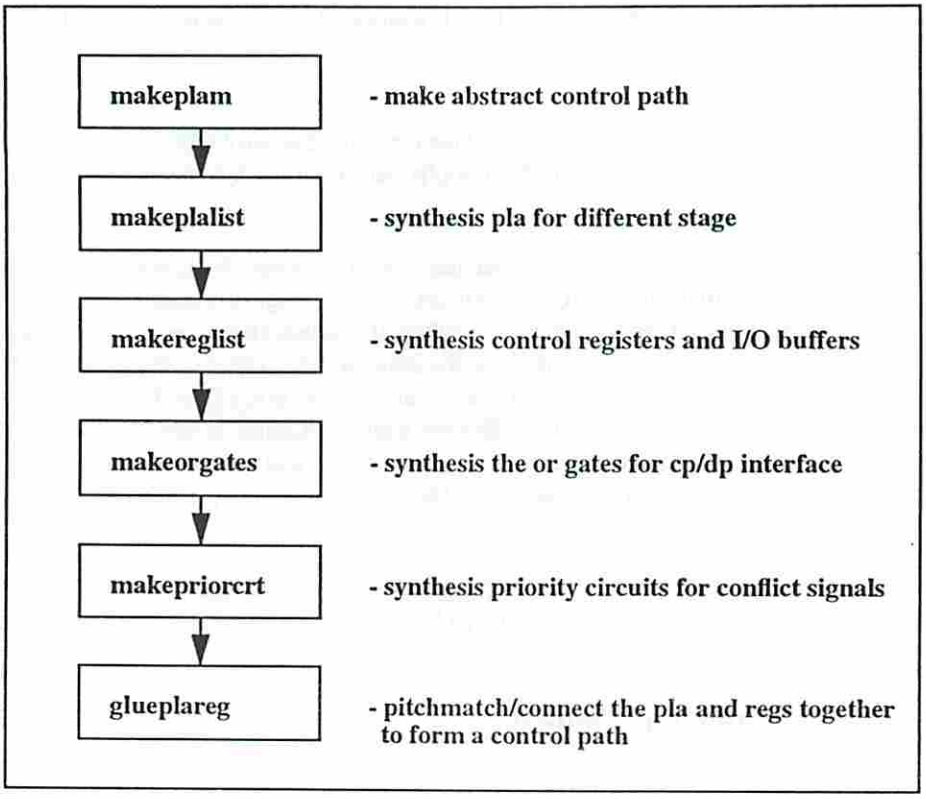


figure 7.1 Overview of CPgen

7.4.1. Building the abstract control path

First the Boolean equations has to be extracted from the control path specification which is in the .cpext file. The inputs, outputs, and_terms, and or_terms are identified by the following functions.

```
** makePLAOrOutputs(Stage,Name,HName) :-
    (setof([Action,And_Or_list],boolean(Name,Action,And_Or_list),Action_list); Action_list),
     makePLAOrOutput(Stage,Action_list,HName),!.

 makePLAOrOutput(_,[],_).
 makePLAOrOutput(Stage,[[Action,And_Or_list]|T],HName):-
     makealias(And_Or_list,And_term_list,HName),
     makePLAAction1(Stage,Action,And_term_list,HName),
     makePLAOrOutput(Stage,T,HName),!.
makePLAAction1(Stage,enable(Unit,Value),OrList,HName):-
     findprimunit(Stage,Unit,Value,ActionList),
     assertPLAaction(OrList,ActionList,HName),
     assertPLAOrOut(Stage,OrList,enable(Unit,Value),ActionList,HName),
     assert(equivaction(enable(Unit,Value),ActionList,HName)),!.
```

makePLAOrOutputs calls setof to get all the actions (which corresponds to activated signals) and the list of and_or_-term. It then calls makePLAOrOutput to form the outputs. makePLAOrOutput is a recursive function which call makealias to build the and term list. Then makePLAAction1 is called which calls findprimunit to map the action name (in the form of enable(_,_)) to the actual signal names.

Then the PLA or terms are asserted in the database as follows:

plaOrOut(int(1,cpr(cpr(1),equal)),[bc19,bc20,bc23,bc24,bc25,bc26],sm2acp1).

```
**makePLAAndOutputs(Stage,HName) :-
        (setof([And,Alias],symbalias(And,Alias,HName),AndList); AndList = []),
         makePLAAndOutput(Stage,AndList,HName), !.
  makePLAAndOutput(_,[],_).
  makePLAAndOutput(Stage,[[And,Alias]|T],HName) :-
        formstatebit(Stage,And,[],BitList),
        assert(plaAndOut(Alias,BitList,HName)),
        makePLAAndOutput(Stage,T,HName),!.
```

makePLAAndOutputs calls setof to get all the and terms and their alias (e.g. bc1,bc2). Then it calls makePLAAndOutput which calls formstatebit to map the input of the and term to the input of the PLA. Then the and_terms are asserted in the database in the following format: (Note: No logic minimization is incorporated in this stage except sharing of and_term is identified.)

plaAndOut(and_term_name, [input_list],name_of_control_path).

E.g. plaAndOut(bc6,[inv(cregout(1,1)),inv(cregout(1,2)),inv(cregout(1,3)),inv(cregout(1,4)),cregout(1,5),cregout(1,6),clk(1),inv(clk(2)),inv(clk(3))],sm2acp1).

```
** makePLAInputs1(Stage,HName):-
        bagof(List,X^plaAndOut(X,List,HName),InList),
        union(InList,PLAInList),
        writeInList(PLAInList,HName),!.
```

makePLAInputs calls bagof to get all the input lists from the and_term_list. Then it calls writeInList to assert the PLA input to the database in the following format :

plaIn(InputName, ControlPathName).

e.g. plaIn(cregout(1,1),sm2acp1).

Input registers and Control registers specifications are also generated in the above phase. Then writePLA is called to write the specification of the PLA to a .pla file and the specification of the registers to a .reg file for further processing.

7.4.2 Synthesis of the PLA

After the abstract PLA is formed and the PLA specification is written to the .pla file, "makecp" calls the program "makeplalist" to synthesis all the PLAs.The "makeplalist" program is described in the following sub-section.

7.4.2.1 Technology

The PLA is implemented as two NOR arrays for high speed application[1]. The inputs and outputs are inverted to preserve the AND-OR structure. pseudo-nMOS design is employed for its simplicity and small size. A disadvantage is the static power dissipation of the NOR gates due to the pull-up transistors[2]. The structure of pseudo-nMOS PLA circuits is illustrated in figure 1.2.

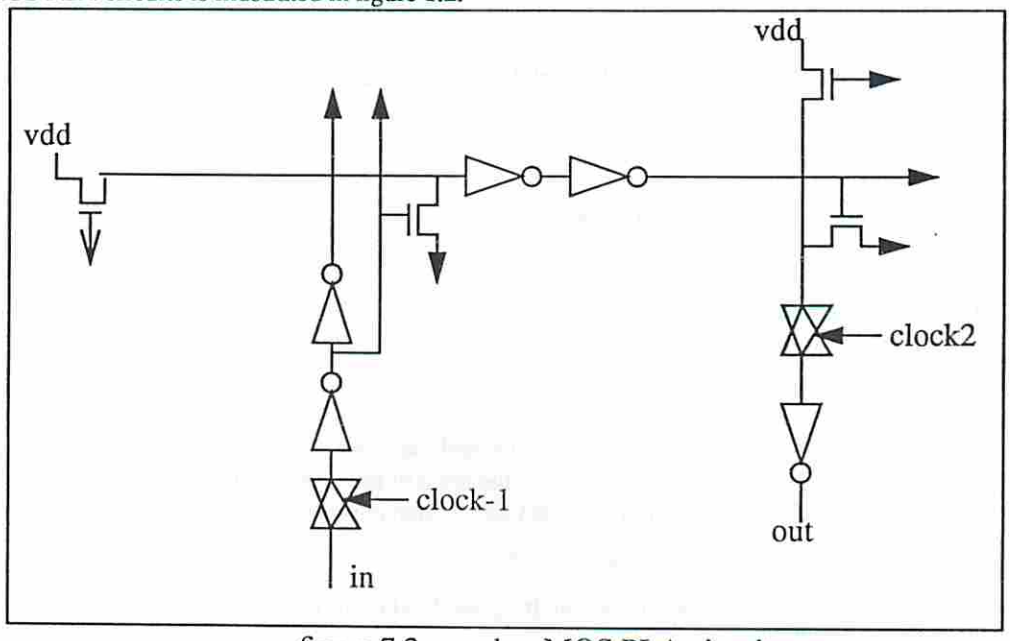


figure 7.2 pseudo-nMOS PLA circuit

7.4.2.2 PLA folding

"makeplalist" calls "plafold" to do the PLA folding. "plafold" first calls "formConandMata" to form the personality matrix according to an initial input/output ordering specified in the specification.

Usually the personality matrix is so sparse that a straight forward mapping from the matrix to silicon area result in a waste in silicon area (in processor controller design, especially true for the OR-plane). Therefore PLA folding is employed to fold the input and output column. Simulated annealing is employed to do the folding. Starting from a random ordered initial personality matrix, possible solutions are generated by switching the position of two rows. The simulated annealing program is shown as follows:

```
simulated_anneal(Prev_Cost,OldConf,Old_Mat,OldCost,OldT,NewCo\nf,New_Mat,NewCost,NewT):-
    iteration(Old_Mat,N),
      gen_and_eval(N,OldConf,Old_Mat,OldT,OldCost,NextConf,Next_Mat,NextCost),
      coolingschedule(OldT,NextT),
```

```
(steady(Prev_Cost,OldCost,NextCost),NewConf=NextConf,New_Mat=Next_Mat,
    NewCost is NextCost ,NewT is NextT;
simulated_anneal(OldCost,NextConf,Next_Mat,NextCost,\NextT,NewConf,New_Mat,NewCost,NewT)),!.
```

iteration determines the no. of each iteration done in each temperature. gen_and eval is the main loop in generating possible solution, evaluate it and decide whether to accept it or not. After N iteration at a temperature, the temperature changes according to the cooling schedules. Then the stopping criteria is checked, if satisfied, stop and report the solution, otherwise, continue the simulated annealing in the new temperature.

The gen_and eval use a quick estimator function(eval_cost) which based on the possible column that can be folded evaluate the relative goodness of the solution compared to the previous one. Depended on the current temperature and the relative goodness, the new solution is checked by an "accept function" to determine whether to accept or reject.

7.4.2.3. Basic Floor Planning of the PLA

PLA is popular because of its regularity and hence easier to be generated automatically. From a personality matrix, the PLA can be generated by tiling some pre-existed primitive cells according to the matrix[1].

"makeplalist" calls a " plaguefold" program to generate the arrays of primitive cells that are used to implement the PLA according to the ordering of the inputs, outputs and the and_term which are determined by the result of the PLA folding. The arrays are made according to the following basic floorplan adopted in our PLA generator which is shown in figure 1.3.

After the arrays are formed, "makeplalist" calls an tiling program "tile" which retrieves the relevant primitive cells from the library and tile then together according to the arrays.

A .sip file is produced and the compactor is called to compact the PLA.

7.4.3. Synthesis of Registers

Control Registers together with input/output drivers are synthesis in this stages. The main program name is "makereglist". Since the PLA is folded, to follow the positions of the inputs and outputs, there are also two sets of registers: top-register and bot-register. Also since the width of the register is much wider than that of the PLA, the top/bot register are further split into two registers in order to have a better aspect ratio when joining the PLA and registers. The decision to split the register into two depending on the number of register cells in each register. The above can be shown in the followings:

```
makereglist(_,[]).
makereglist(Name,[H\T]):-
    splittopbot(H),     % split the register into top and bot register
```

OC	OUTPUT_INV	TR	INPUT_INV	AC
OR	OR_CELL	A_O_BUF	AND_CELL	A_PU
OR	OR_CELL	A_O_BUF	AND_CELL	A_PU
OR	OR_CELL	A_O_BUF	AND_CELL	A_PU
OR	OR_CELL	A_O_BUF	AND_CELL	A_PU
OR	O_PU	A_O_R	AR	AR
ORC	OUTPUT_INV	A_O_R	INPUT_INV	ARC

OR : Or Plane routing
ORC : Or Plane routing corner
OUTPUT_BUF : Output inverter
OR_CELLS: Or Plane On and Off cells
O_PU: Or Plane Pull-up cells
TR: Power routing cells
A_O_BUF: And Plane OrPlane communication and buffer cells
A_O_R: And Plane/Or Plane routing cell
INPUT_INV: input inverter
AND_CELL: And Plane On/Off cells
AR: And Plane Power Routing cell
AC: And Plane corner
A_PU: And Plane Pull-up cells
AR: And Plane routing cell
ARC: And Plane routing corner

figure 7.3 Basic floorplan of PLA

```
makename(H,top,Top),
makename(H,bot,Bot),
makereg1(Name,H,top,unflip),
makereg1(Name,H,bot,flip),
makereglist(Name,T),!.
```

"makereg1" is the main program to synthesis the register. Similar strategy is used as the synthesis of PLA. An array of register cells is formed. The type of register cell depends on the signal that goes into the cell. Generally input signal goes to a Master-Slave register with load control and clocked by Φ1 and an output signal goes to a Φ2 driven Master-Slave register. The input is clocked in Φ1 at which the datapath completes its operation and the output is clocked at Φ2 at which the data is latched in the master of the registers of the datapath.A clock generator is added in each register since a two phase clocking scheme is used. The size of the drivers is decided according to the load

that the signal is going to drive.

After the arrays of register cells is generated, a similar procedure as the PLA synthesis is followed, i.e. calling the tiler to retrieve the basic register cells from the library and tile them together to form the .sip file. Then the compactor is called to compact each register. After that the two registers in either top/bot registers are joined together by both pitchmatch and channel routing strategy (discussed in the Joiner section of Stick-Pack).

7.4.4 Or-gates and priority circuit synthesis

The specification of the or gates and priority circuits are included in the .cpext file in the following format:

module(priority,circuit_name,I/O_number,input_list,output_list).

e.g.module(priority,priority(d,pc),in(2)/out(2),[control(stage(2),enable(pc,load)),-control(stage(1),enable(pc,inc))],[enable(pc,inc),control(priority(d,pc),enable\(pc,load))],[],[]).

module(or,circuit_name,I/O_number,input_list,output_list).

e.g. module(or,or(enable(pc,load)),in(2)/out(1),[control(priority(d,pc),enable(pc,-load)),control(stage(1),enable(pc,load))],[enable(pc,load)],[],[]).

The or gates are synthesized in the same way as the register. The program "makeorgates" serves this purpose. The priority circuits are generated by calling the Topolog tools to synthesize.

7.4.5. Joining the PLA and the registers

The final control path is obtained by joining the PLA and the registers together. Since the signals between the PLA and the registers are already in order, so a pitchmatch program is called to join them together. The program "glueplareg" is called to serve this purpose. It calls the pitchmatcher and the router alternatively to join the cells.

```
glueplareg1(Name,H):-
   tell(pitchtemp),
   write(pitchmatch(Name,x,P,hiy,R,loy)),write('.'),nl,
   told,
   unix(shell('~tsui/working/Glob/pitchmatch < pitchtemp')),
   tell('mkgridtemp'),
   write(mkgrids(Temp,R,loy,no,P,hiy,no,S,h)),write('.'),nl,
   told,
   unix(shell('/home/zelea3/tsui/Plagen/Plagen/router < mkgridtemp')),
   tell(pitchtemp),
   write(pitchmatch(Name,x,R1,hiy,Temp,loy)),write('.'),nl,
```

```
told,
unix(shell('~tsui/working/Glob/pitchmatch < pitchtemp')),
tell('mkgridtemp'),
write(mkgrids(H,Temp,loy,no,R1,hiy,no,S1,h)),write('.'),nl,
told,
unix(shell('/home/zelea3/tsui/Plagen/Plagen/router < mkgridtemp')),
!.
```

Reference

[1] 'Principles of CMOS VLSI Design, A Systems Perspective', Neil Weste

[2] 'Computer Aids for VLSI Design'; Steven M. Rubin

8. Simulation, Analysis and Verification

In this section we present a simulation system (including the simulation, analysis and verificationof the corresponding level), the left side part of ASP structure in the Figure 1.1. As above description, ASP is an application-driven silicon compiler which takes ISA (Instruction Set Architecture) specification as input, synthesizes specifications from abstraction to more details, based on those benchmark programs at the corresponding levels, and produces the CIF (Caltec Institute Form). The method we used for the simulation system is quite different from those general purpose simulation tools because we strongly combine the simulation and synthesis system from the very abstract level specification (ISA) to the layout specification (CIF).

In the previous version of the ASP, the simulation system was not included, since the translation of the benchmark programs into the corresponding levels is a problem. Another unsolved problem is to find how benchmark programs effect logic-level and circuit-level simulators and analizors to guide the systhesizors during the translations.

In the current version of ASP, We include the functional- and behavioral-level simulation system after we refined the high-level synthesis programs to be embeded with the simulation system. For logic-level and circuit-level simulations, we used those genera- purpose tools like (ESIM and SPICE) to verify the design.In the near future, we hope we can include the logic-level and circuit-level simulation system into the next version of the ASP.

8.1 Overview

The simulation system in ASP is composed by two phases, the retargetable code generator and VLSI circuit simulation. The retargetable code generator is presented in the section 8.2. This is important for building a fully application-driven silicon compiler, since the benchmark programs in high-level language must be compiled into the target machine code where the VLSI simulation system can take as input. Another critical idea for ASP is the simulation is strongly combined into the synthesis programs. The simulation result not only evaluates, analyzes, and verifies the design, but also guides the corresponding level synthesis program to have a better design. Those simulation system are descript in the section 8.3.

8.2 Translation and Compilation

In the ASP, we allow the user to provide an ISA specification and a set of benchmark programs in high-level language. To automatically compile those benchmark programs into the target machine code becomes a big issue of the simulation automation system. We had solved some primal problems before we released the current version of ASP.However, to be a fully automatic compiler generation is still a big research area.

Almost all modern compilers are composed two parts: the front end, which includes scanner, parser and semantic routines, and the back end, which includes the code generator and optimizer. The interface between the front end and the back end of a compiler is an intermediate representation (IR). As the Fig 8.1 shown, the compilation benchmark programs into the target machine codes can be composed by four parts, which are front end compilation, code generator generation, code generation, and optimization. We present each part of compilation automation in following four sub-sections. Potencially the benchmark programs provided by users can be any common high-level language (i.e. C, ADA, and Prolog). The front end of the compiler of those high-level languages must be provided with the set of benchmark programs by the user. The code generator generator takes the compiled IR and ISA specification and produces the corresponding target machine code. In the current version of ASP, we use an Prolog compiler and a set of benchmark programs in Prolog to produce a set of benchmark programs in target machine code. In 8.2.5, a general translation method is presented which is applied into the Functional- and Behavioral-Level Simulation System in the Section 8.3.

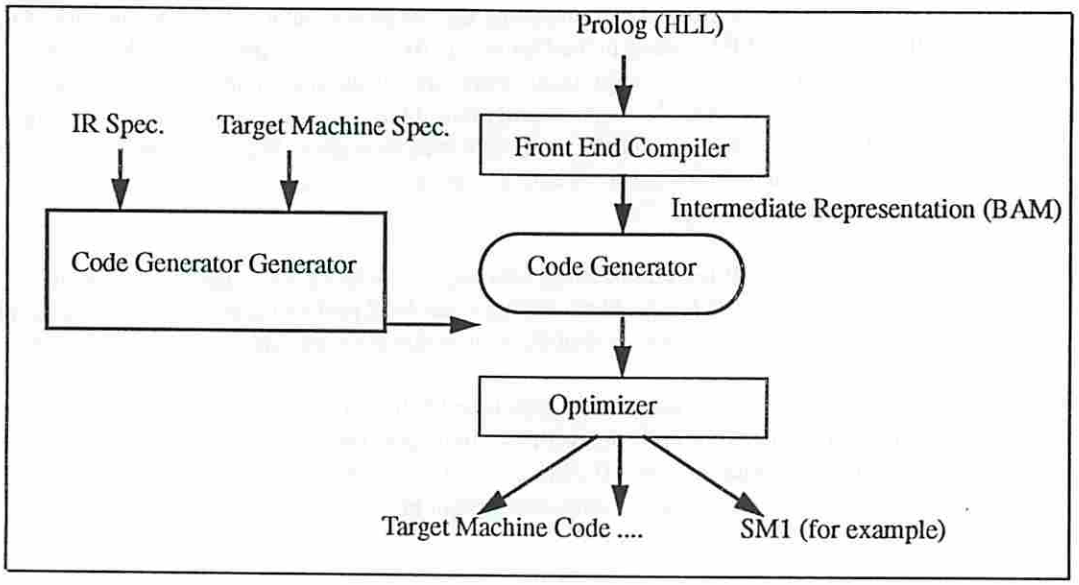


Figure 8.1

8.2.1 Front End Compilation

A compilation for Prolog programs can be done in three stages (Figure 8.2). First, the compiler translates the source program into the form in kernel Prolog. Second, the dataflow analyzer processes the source code in kernel Prolog and produces source code in annotated kernel Prolog. Finally, the kernal compiler compiles the code in annotated kernel Prolog to the BAM code, which is considered as intermediate representation [3,4].

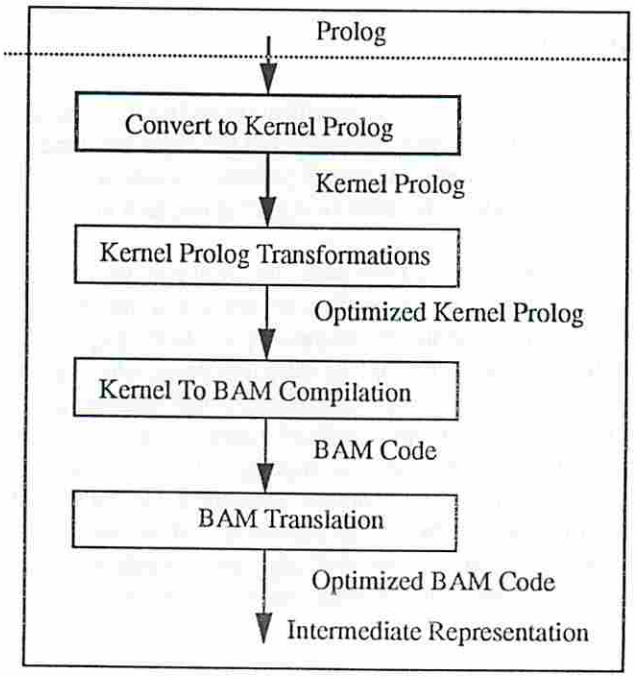


Figure 8.2

8.2.2 Code Generator Generation

In this sub-section, a formal design of derivation of code generators from instruction set specification and intermediate representation, the BAM code, is presented. The basic concept of the code generation algorithm, which is machine indepedent, is using the theorm proving to guarantee the quality of the codes.

Given a set of instruction specifications I1,I2,...In, and a goal G which is an IR, the code generation is to search a sequence of codes in {I1,I2,...,In} which can match G. The problem can be considered as a theorem proving problem. A set of {I1,I2,...,In} is considered as a set of axioms and G as a theorem needed to be proven. The rules applied over axioms and proven theorems are paths of the search.

The example of the instruction set architecture used in SM1 can be translated into a theorem proving problem in Prolog as shown belows:

```
axiom([store, Const], m(Const) <- AC).
axiom([load, Const],AC <- m(Const)).
axiom([add, Const],AC <- AC + m(Const)).
axiom([and, Const],AC <- AC∧m(Const)).
```

The rule predicate denotes more than one instruction of target machine can be combined to match the IR.

```
rule(S,S2,L) :-
axiom(L,S1),
append(S,S1,S2).


theorem(S,L) :_ axiom(L,S).
theorem(S,L2) :-
theorem(S1,L),
rule(S1,S,L1),
appemd(L,L1,L2).
```

A code generation using theorem proving method increases the retargetablity of compilation and guarantee to get a high quality code sequence. As shown above, program is easy to design and can be translated into DCG form (Definite Clause Grammar) [6,7,8] which is discussed in Section XXX. Semantic attributes can be assoicated with axioms and rules to help to check the restriction of register binding, addressing mode selection, cost, and register allocation. During the search space may be large when the target machine instruction set is complex, heuristic search may be applied to get better performance.

8.2.3 Code Generation

After we get a code generator from the code generator generator, described in the previous sub-section, we can produces the target machine code by matching the benchmark programs in intermediate code into the code generator table.

Intermediate Code:

```
add r1, r2, r3 %r3 <- R1 + r2
```

CodeGeneration Table:

entry(add, [[load,R1],[add,R2],[store,R1]]).

The Target Machine Code:

load R1
add R2
store R1

8.2.4 Optimization

A peephole optimization has been used on the codes produced by the code generation phase. If the target machine is pipelined, more optimization can be reached by getting pipelining information from the Piper.

8.2.5 The Translations Between Specifications

In general, the translation between specifications are done in the synthesis part, the right-hand side of the fig 1.1. However, the benchmark programs need to be translated into more concrete level with the specifcation. For example, the benchmark programs can be represented in assembly language for the ISA level simulation and in the binary code for the control path and data path simultion.

During the translation of the benchmark prgrams between levels, the high-level synthesis programs need to provide information, i.e. the opcode assignment and memory port assignment. Again, as you see, there are lots of benefits to strongly combined synthesisand simulation systems in the silicon compiler.

8.3 Functional- and Behavioral-Level Simulation System

The functional- and behavioral-level simulations in ASP are based on the specification coming out from each level's systhesis. Basically, the simulation program reads the specification at the corresponding level and translates into a table which makes the simulation process more efficient. For the performance reason, we refine the synthsis program to be able to produce the table needed for the corresponding simulator directly.

Most functional- and behavioral-level simulation program in ASP use event-driven finit-state machine. Generally the clock ticks are the major events during the simulation. Interrupts could be an event. However, the current version of ASP system do not consider the interrupt signals coming from outside of the chip.

We present the major functional- and behavioral-level simulation system (simulator, analyzer, and correct checking) in the following sections.

8.3.1 ISA Simulation System

The first level specifcation is the instruction set architecture (ISA), which is a runable specification. The ISA simulator executes the ISA specification. The ISA analyzer calculates the instruction frequency count and pattern of continuous instructions frequency count. The ISA correction checker compares the simulation result and the expecting result.

In the Figure 8.3, the SM1 instruction frequency analysis is shown. The pattern of contin\uous instructions frequency analysys is presented in the Figure 8.4.

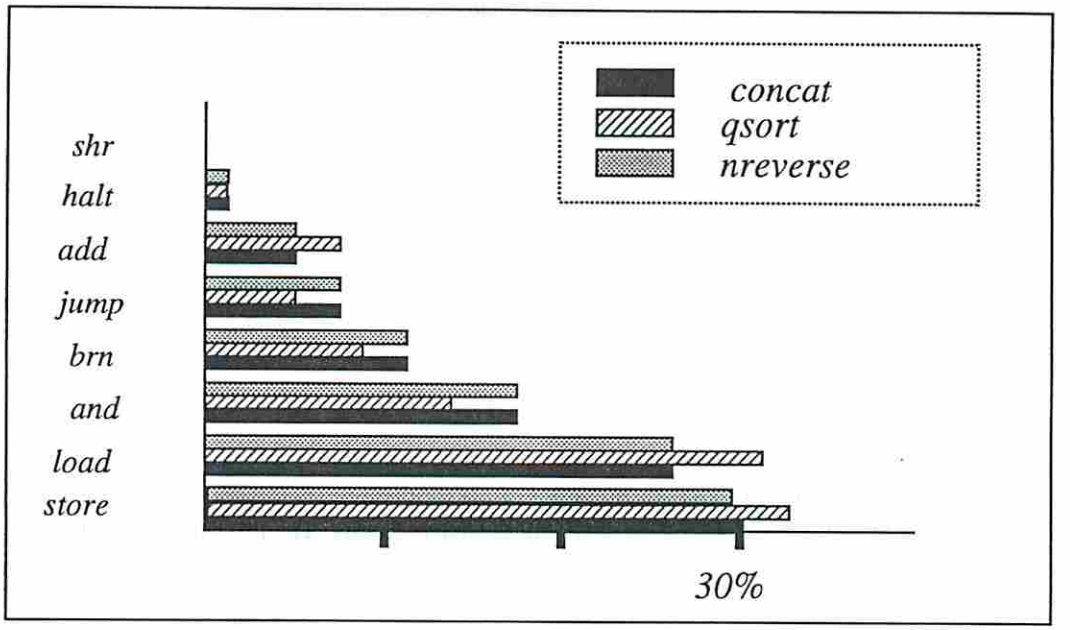


Figure 8.3 Instruction Usage Frequency Analysis

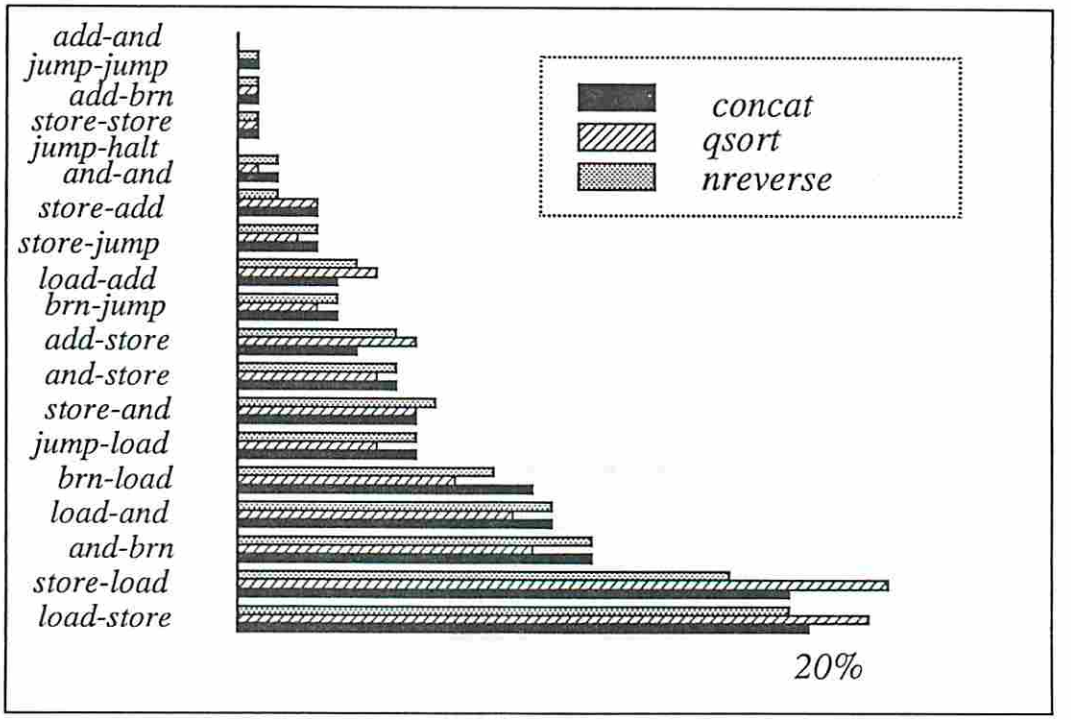


Figure 8.4 Instruction Pattern Analysis

8.3.2 RTL Simulation System

The first level systhesis program is scan, described in section XXX. There are two files produced by the scan program. One is the specification for next level systhesis and one is a table for the RTL simulator. The format of that table is shown as below.

abs(Opcode,List_of_RTL)

For example:

```
abs(add,[move(pc,memAR),
 mem_read(memAR,memDR),
 inc(pc),
 move(memDR^1, cr11),
 move(memDR^2, memAR),
 mem_read(memAR,memDR),
 arith(+, ac, memDR, ac)]).
```

The RTL simulator matches Opcode of entries of the simulation table with the content of cr11 in the RTL move(memDR^1, cr11) and execute the rest of RTLs in the matched entry. The RTL analyzer calculates the RTL frequency count, which is shown in the Figure 8.5, and the RTL correction checker compares the result from RTL simulation and the expected result.

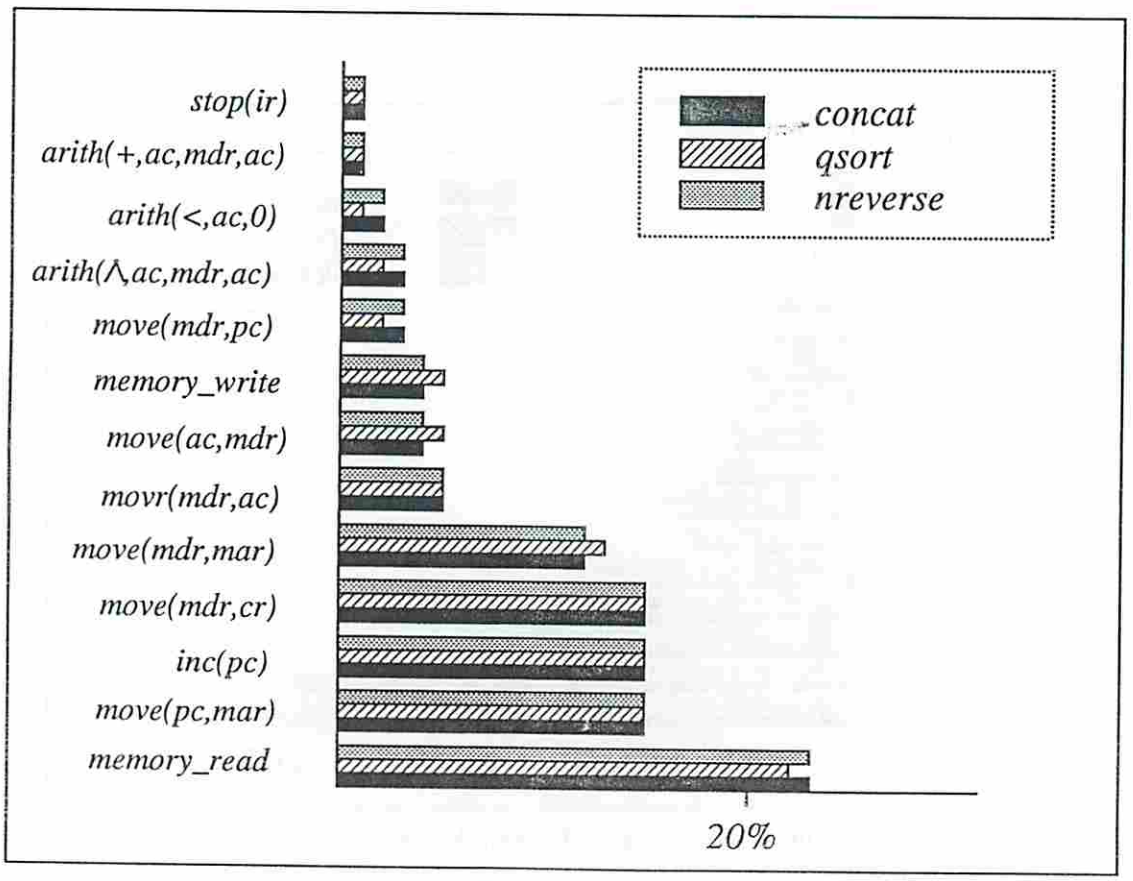


Figure 8.5 RTL Frequency Analysis

8.3.3 Scheduling Simulation System

The scheduling synthesis program produces the scheduled specification and a scheduling simulation table, shown as below.

```
sched(add,[1,2,3,4,5,6]).
sched(and,[1,2,3,4,5,6]).
sched(shr,[1,2,3,4]).
sched(load,[1,2,3,4,5,6]).
sched(store,[1,2,3,4,5]).
sched(jump,[1,2,3,4]).
sched(brn,[1,2,3,4,5]).
sched(halt,[1,2,3,4]).
```

As above shown, the add instruction in SM1 takes 6 cycles to be finished and the shr instruction takes 4 cycles.

The scheduling simulator and analyzer calculates the total cycle count for each benchmark program. This information can be use to estimate what scheduling algorithm is better for a certain design. The scheduling correction checker compares the result from simulations before and after the scheduling phase.

8.3.4 Abstract Control Path And Data Path (ACPDP) Simulation System

The ACPDP is the specification of control path and data path before the pipeline synthesis. If the pipe is one, then ACPDP actually is the same with the final control path. If the pipe of a pipeline machine is more than one, than the final control path and data path is produced by the Piper. A ACPDP table, as shown following, is produced by the ACPDP synthesis.

```
abscp(add,[move(pc,bus(unassigned),memAR),
 enable(pc,inc),
 enable(mem,mem_read),
 move(field(memDR,1),bus(unassigned),cr11),
 move(field(memDR,2),bus(unassigned),memAR),
 enable(mem,mem_read),
 move(ac,bus(unassigned),port(alu(A),1)),
 move(memDR,bus(unassigned),port(alu(A),2)),
 move(alu(A),bus(unassigned),ac),
 enable(alu(A),add)]).
```

The ACPDP matches the Opcode in entries of the table with content of cr11 when the RTL move(field(memDR,1),bus(unassigned),cr11) is executed and contuously executes the rest of RTLs after move(field(memDR,1),bus(unassigned),cr11) in that entry. The ACPDP analyzer calculates the RTL frequency count. The ACPDP compares the result from the simulation with the expected result. In Figure 8.6, a SM1 functional unit switching rate analysis is shown and the abstract scheduling analysis is presented in the Figure 8.7.

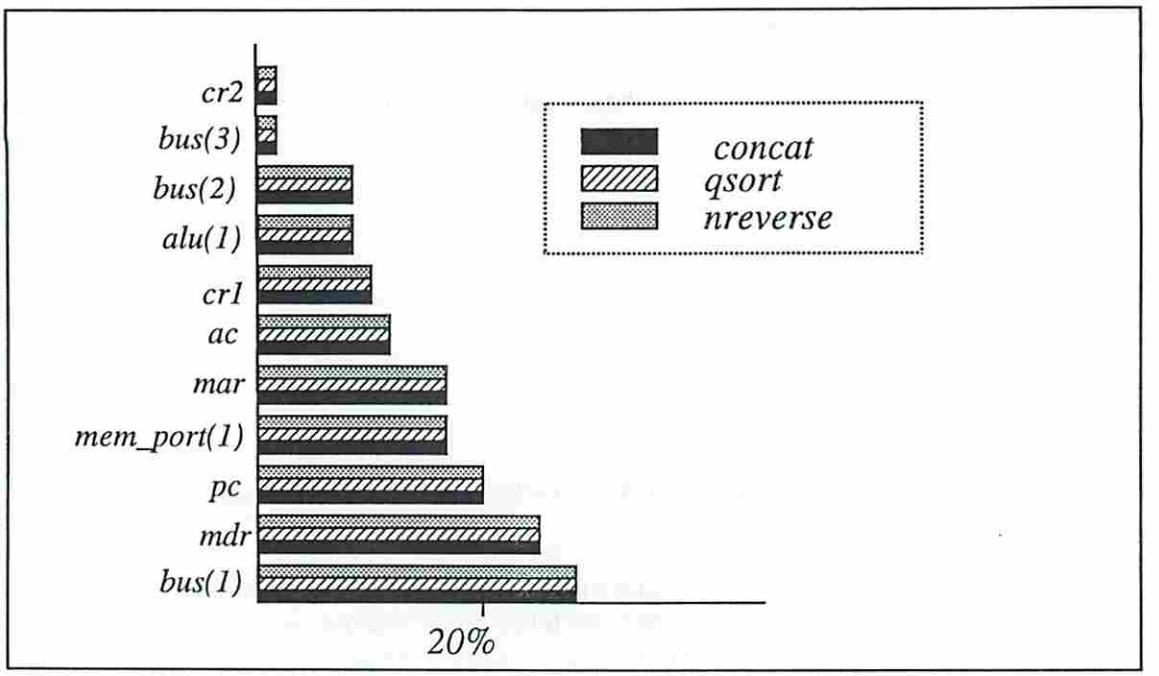


Figure 8.6 Functional Unit Switching Rate Analysis

	concat	qsort	nreverse
Before Scheduling	5574	3369	19270
After Scheduling	4312	2508	14451

cycles

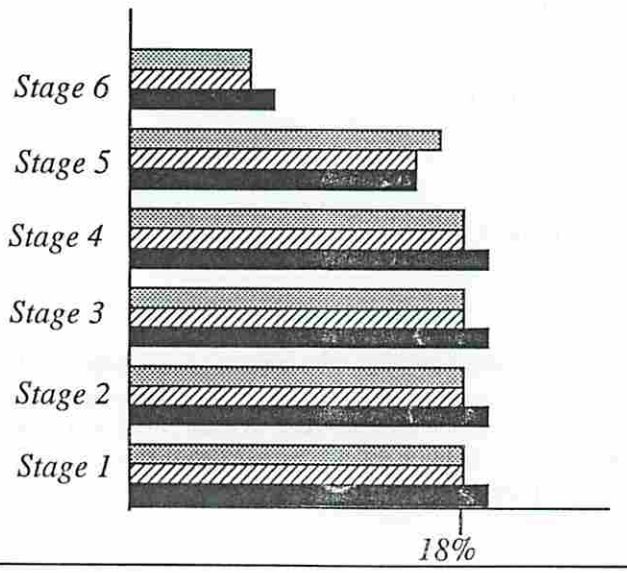


Figure 8.7 Abstract Scheduling Analysis

8.3.6 Control Path And Data Path (CPDP) Simulation System

Basically, the (CPDP) simulation system is similar to ACPDP. Howevel, the CPDP is a specification after the pipelining synthesis and the ACPDP is a specification before the Piper. For a sequencial design, the ACPDP is the same with CPDP. For the Pipeline design, the CPDP is obviously different from the ACPDP.The method to simulate a pipeline circuit is a liitle be different from the sequencial machine.

Since there are two output specifications from the Piper, the . cpext and the .mod files, the grobal simulation at this level must take these two file as inputs. For the control path generator (CPGEN), the CPDP simulation system generates a set of test vectors based on those benchmark programs. The format is below.

control(Instruction-count, Cycle, Stage, List_of_PLA_inputs, List_of_PLA_output)

For example:

control(3,2,2,[controlReg1-load2, controlReg2-x, controlReg8-x],
[enable(r2,load),
enable(ts(field(memDR(1),2),bus(1)),on),
enable(ts(field(memDR(1),opcode),bus(1)),on)]).

The above test vector denotes at the third instruction count, cycle 2, and the stage 2, the inputs of PLA are control register 1 is load, controlReg2 is don't care, and the controlReg 8 is don't care, the outputs of the PLA are enable(r2,load), enable(ts(field(memDR(1),2),bus(1)),on), enable(ts(field(memDR(1),opcode),bus(1)),on)]).

8.3.6 Logic- and Circuit-Level Simulation System

The current version of ASP doesn't have its own logic- and circuit simulation system since to build powerful and highly accurate simulation tools in these two levels for ASP itself needs tremendous work. We have been using the ESIM to verify the PLA function and SPICE to verify some circuits in the library. In the next version, we hope the ASP system has its own logic- and circuit-level simulation system.