

10

AD-A172 975

REPORT DOCUMENTATION PAGE

unclassified		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION / AVAILABILITY OF REPORT unlimited	
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION The Regents of the University of California	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION SPAWAR	
6c. ADDRESS (City, State, and ZIP Code) Berkeley, California 94720		7b. ADDRESS (City, State, and ZIP Code) Space and Naval Warfare Systems Command Washington, DC 20363-5100	
8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22209		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) * High Level Synthesis in ASP			
12. PERSONAL AUTHOR(S) * William R. Bush			
13a. TYPE OF REPORT technical	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) * September 18, 1986	15. PAGE COUNT * 31
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
19. ABSTRACT (Continue on reverse if necessary and identify by block number)  Enclosed in paper.			
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL		22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DTIC FILE COPY

This document has been approved for public release and sale; its distribution is unlimited.

DTIC ELECTE D  
OCT 14 1986  
S E

Productivity Engineering in the UNIX† Environment

High Level Synthesis in ASP

Technical Report

S. L. Graham  
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



86 10 9 01

# High Level Synthesis in ASP

William R. Bush

*19 August 1986*

## 1. Introduction

The purpose of the Advanced Silicon compiler in Prolog project is to synthesize high-quality VLSI layouts from high-level specifications. The ASP system consists of two major components. The low-level part produces masks from functional blocks containing logic equations. The high-level part generates the input to the low-level part from instruction-set level specifications written in Prolog.

This document describes the high-level part. Section one presents its general organization. Section two describes its first stage, register allocation. Section three describes its second stage, operation scheduling and finite state machine construction. Section four defines the structural description mechanism that is the interface between the high-level part and the low-level part.

## 2. Overview

The low-level component of ASP generates a control path containing a finite state machine and a data path containing registers and functional units.

Given this target, ASP's high-level synthesis component performs five tasks. It maps the Prolog variables in input specifications into hardware

registers. It maps Prolog operators into functional units. It maps Prolog control constructs into finite state machine transitions. It schedules operations and register transfers. And it produces low-level structural descriptions that embody these mappings and schedules. It performs these tasks in two stages.

The first stage maps Prolog variables into hardware registers. The result of the stage is a specification equivalent in functionality to its input, but one that stores values in a Prolog database tagged with register names. Values are moved to Prolog variables for computation only, and are stored again in the database after computation. The Prolog variables used in this way can be thought of as the connections between registers and functional units.

The second stage performs all the other high-level tasks. It first constructs a finite state machine, assigning each Prolog goal in its input specification to a specific state, transforming each goal into a set of hardware operations, and mapping each change in control flow into a state transition. It then uses the FSM to generate a structural description, containing a control path and a data path, which is passed to the low-level part of ASP.

The order in which the high-level tasks are performed is dictated by the dependencies between tasks and by the degree to which the tasks change the original input specification. Register mapping is done first because it perturbs the high-level specification the least and because operation scheduling depends on it. Operation scheduling is done next because state transition and op mapping depend on it; they are done next. Generation of the output structural description is done last because it depends on all the previous tasks. The order

of tasks corresponds to levels of abstraction. First the model of storage is changed, then timing is introduced, and finally bindings between Prolog entities and hardware objects are made.

Both stages are divided into two phases. A task may be performed in one or both phases. The first phase creates, from its input specification, a database of analysis information (and checks that the specification is well-formed). The second phase generates a new specification based on the analysis of the first phase. The functionality of the new specification can be verified by using it in simulation.<sup>1</sup>

The system is modular; the results of any phase (database or specification) can be written out and used independently by the next phase. As a result the system is easy to debug, because each phase is autonomous. This also means that the user (and possibly future optimization tools) can improve the output of any phase before it is sent to the next.

The system has synthesized small designs, including a simple microprocessor. In general the next extensions to the system will make it more sophisticated and robust. Specific improvements are discussed in following sections.

## 2.1. The Use of Prolog

ASP uses Prolog as its specification language. This has turned out well so far, for four reasons. First, control in Prolog is simple (ignoring backtracking),

---

<sup>1</sup>The original input specification should be a directly executable Prolog program. The register-based specification produced by the first stage is also directly executable. The structural description generated by the second stage requires a simulator, which has not yet been constructed.

and maps easily into hardware. Second, clauses tend to be short and well modularized, lending themselves to easy translation. Third, Prolog's single-assignment rule makes analysis easier. Fourth, Prolog's simple structure and syntax facilitate automatically generating Prolog specifications.

ASP also uses Prolog as its implementation language (as does the OCCAM project), resulting in two benefits. First, Prolog's database properties have aided the production and processing of analysis information. Second, Prolog's rule-based environment has made heuristics easy to implement.

On the other hand, without a sophisticated debugger, Prolog, with its failure and backtracking semantics, has been hard to debug. Similarly, Prolog code is hard to modify without careful redesign.

## 2.2. Existing Systems

The full-range synthesis systems examined here all take high-level specifications to silicon. Such complete systems are rare.

### 2.2.1. MacPitts

The foremost complete synthesis system is MacPitts ([MacPitts-Manual], [MacPitts-Intro]). MacPitts uses LISP as its language framework, but special hardware-oriented functions (for defining both control<sup>2</sup> and data) are used for specification. The unit of specification is the finite state machine. The MacPitts FSM implements a simple master-slave model of computation -- in any

---

<sup>2</sup>Control flow is managed with the LISP *cond* function, semantically modified in one important respect -- once a branch is taken, the consequents in that branch are evaluated in parallel.

state register values are held while new values are computed, and then, on state transition, the old values are replaced with new ones.

MacPitts maximizes parallelism, creating enough buses and functional units to make each state maximally parallel, while avoiding duplication. On the other hand, storage is managed explicitly by the user. The system makes no attempt to optimize register usage.

One advantage of MacPitts is the simplicity of the specification mechanism. Unlike a conventional programming language with a plethora of control structures, MacPitts has essentially one. The FSM construct is powerful and natural. There are also few data types.

Another advantage is its optimization of operator usage. It does not, however, try to identify or reduce critical paths; the user must optimize, but can affect area and time factors only indirectly.<sup>3</sup> The system has produced several designs, including a 6502.

• • •

### 2.2.2. CMU-DA

The CMU-DA system, in contrast to MacPitts, employs extensive optimization techniques, with mixed success.

The CMU-DA specification language is ALGOL-derived ISPS, which is quite complex compared to MacPitts. It has several control constructs, and both explicit parallelism and explicit sequentiality. Instead of mapping a specification more or less directly into hardware structure as MacPitts does, the

---

<sup>3</sup>See section 3.13 of the user's manual for advice to users in this regard.

CMU-DA system first translates the ISPS into a dataflow graph (called a Value Trace, or VT), which serves as the basis of synthesis.

Various competing VT optimization and hardware binding strategies have been studied. These strategies take two forms: algorithmic compiler techniques ([CMU-DP], [CMU-Facet], [CMU-BLT]), and rule-based expert systems ([CMU-DAA83], [CMU-DAA85]).

The compiler techniques are primarily graph transformations that apparently give good local results but poor global ones. This is probably because the transformations are based on graph structure pattern matching, which is not global in nature. For example, one system ([CMU-DP]) can optimize communication between a few hardware elements, but cannot produce a good general bus structure. To improve global results it allows a human designer to specify an initial global bus skeleton. This system has produced 6502 and PDP8/E designs.

The rule-based DAA system deals with both local and global structure. It first does global register and control allocation, then local dataflow-based operator and temporary allocation, then local improvement, and finally global improvement. The system has produced 6502 and 370 designs.

The overall CMU-DA results are inconclusive. Development of the rule-based system is continuing ([Ulysses], [BottomUp]).

### 2.2.3. Other Systems

A few other systems have synthesized small designs. These systems are in the shadow of the MacPitts and CMU-DA efforts.

The CADDY system ([CADDY-CHDL], [CADDY-DAC]) follows the CMU-DA algorithmic paradigm. Its specification language is Pascal-like, augmented with FORK and JOIN (for specifying parallelism).<sup>4</sup> Global electrical and timing behavior can be specified. A specification in this language is transformed into a dataflow graph, which is optimized<sup>5</sup> via variable lifetime analysis and operator minimization, and from which circuits are synthesized.

A greatest-common-divisor implementation has been produced. Experience with the system has shown that its register allocation results 'differ strongly' from manual designs -- are apparently worse.

In contrast to CADDY, the OCCAM to CMOS effort ([OCCAM-CMOS]) is rule-based. Its specification language is OCCAM, which supports parallel communicating processes and both explicit fine-grained parallelism and sequentiality. The system performs functional level design in three stages: variables are assigned to registers, OCCAM control constructs are translated into hardware structure (making operations concurrent where possible), and communication between processes is implemented. Users interact with the system during this process (to establish, for example, the width of registers). Detailed information

---

<sup>4</sup>It also has other primitives for controlling concurrent hardware processes. Hierarchical modularity is supported by hardware procedures.

<sup>5</sup>Graph grammars are used to specify graph transformations.

on the stages is not given in the literature. The result of this automated design is another specification, expressed in the automaton-based DDL language. All logical-physical binding is apparently done in producing the DDL specification. The DDL specifications are optimized and then further transformed, through a number of steps, down to masks.

This system provides a useful model in that it uses multiple well-defined levels of abstraction. Its compilation step also correspond roughly to those in ASP.

#### 2.2.4. Human-Oriented Design Systems

Most current CAD systems are oriented toward human designers, helping them manage the design process. Almost all current hardware description languages and systems are oriented toward multi-level design and simulation. Two examples are n.2 [n.2] and Cascade [Cascade]. Virtually any group of any size, governmental or industrial, has produced one of these systems.

The systems often have well-defined levels of abstraction, but are quite complex,<sup>6</sup> with human designers generating and interconnecting all descriptions.

The systems tend to consist of a set of loosely coupled specialized tools, each of which can be applied to a common database. The complexity and loose coupling of these systems makes synthesis using them difficult.

---

<sup>6</sup>The Cascade system uses six languages -- LASSO (system architecture), LASCAR (functional architecture), CASSANDRE (register transfer), POLO (logical gate), CASTOR (transistor switch), and IMAG (electrical).

### 2.3. General Observations

Silicon compilation is more difficult than programming language compilation for a number of reasons. There are more degrees of freedom in silicon compilation -- the compilation process both compiles to and defines the target machine. Several competing low level constraints -- area, time, and topology -- influence compilation. Concurrent use of machine resources must be optimized. And there is an apparent need for high performance results -- users are not willing to pay a performance penalty for an automatically generated design.

The ASP system will address these issues by experimenting with the nature and timing of binding decisions, by using low level information in high level synthesis, by applying existing parallelization techniques, and by allowing human guidance of the design.

Some systems are driven by a dataflow representation of the input specification. The result is to bind operators and their connections first. This produces maximal concurrency, but is potentially wasteful of resources (as dataflow machines are) and damaging to a regular layout (because dataflow graphs are often irregular).

In contrast, the ASP system will preserve specification structure throughout synthesis. This, it is believed, will make the synthesis process easier to control. First, the user will be able to affect synthesis through the form of the specification. The MacPitts system works this way. Until automated synthesis becomes much better understood, for high quality results the user will be relied on to some extent to guide synthesis. Good user interac-

tion will be important. Second, regularity should be enhanced by using timing and control to drive connectivity.

It is necessary to control the many degrees of freedom in synthesis. There is in general a tradeoff between generality and performance. High performance requires specialized expertise. This is reflected in good initial global choices, which can usually be made only with knowledge of the problem domain. ASP will concentrate on microprocessors.

### **3. Storage Allocation**

The first stage of synthesis in ASP maps variables appearing in a specification to physical storage elements. It takes a tail-recursive executable instruction-set-level Prolog specification and produces an equivalent one that uses globally available facts in a Prolog database rather than Prolog variables. These global facts map directly into hardware registers. The stage operates in two phases. The analysis phase associates storage elements with variables, optimizing by sharing when possible. The transformation phase uses the analysis information to generate a new fact-based specification.

#### **3.1. The Analyzer**

The analyzer assigns storage elements to all Prolog variables in a specification. Different variables are made to share the same element under two basic circumstances, argument passing and value assignment.

Some relationship must be established between a goal invocation and its matching clause head so that values may be passed. The analyzer does this by

assigning the same storage element to variables in the same positions in invocation and head. Thus in

$$\dots g(A, B), \dots$$

and

$$g(X, Y) :- \dots$$

A and X share one element, and B and Y share another. Assigning a value to A (via its storage element) also assigns that value to X.

A special case of argument passing is tail recursion; different variables in the *same* clause are assigned the same storage elements. The clause head variables (representing the values of the current loop iteration) share the storage of the variables in the recursive invocation (representing the values of the next iteration).<sup>7</sup>

Since Prolog is a single-assignment language, its variables are effectively write-once memory. To use hardware efficiently, the analyzer assigns different variables in the same clause to the same storage element when possible. In particular, the source and destination variables of an *is* operator can often be assigned the same storage element, since the old source value is often not used after the new destination value is computed.<sup>8</sup>

The analyzer takes a goal as its input argument, and analyzes the (depth-first) transitive closure of clauses reachable from that initial goal. It generates

---

<sup>7</sup>This sharing is correct only if the next iteration values are defined after all uses of the current-iteration values. Live variable analysis can be used to verify this condition. Local reordering can be done to produce this condition.

<sup>8</sup>As with tail-recursion, live variable analysis is needed to assure the correctness of storage element sharing.

a database of variable information, each database entry corresponding to a variable in a clause, and possessing an associated storage element and other information used by the transformer.

### 3.2. The Transformer

The transformer produces new executable Prolog clauses from the ones scanned by the analyzer, changing variable references into constructs using storage elements that map readily into hardware. Five specific transformations are performed: argument removal, expression value manipulation, implicit transfer, dispatch setup, and memory lookup.

The first two transformations implement a load-store register-based model of computation. Arguments are put in storage elements rather than variables, and those storage elements are then accessed and set by expressions that need or produce values. For example, the goal

... *increment(PC, P)*, ...

and the clause

*increment(In, Out) :- Out is In + 1.*

become<sup>9</sup>

... *increment*, ...

and<sup>10</sup>

---

<sup>9</sup>This assumes that the transformer has already put the value associated with PC into `storageElement1` when that value was defined, before the `increment` goal, and that the analyzer has determined that P may be put in the same storage element as PC.

<sup>10</sup>The access and set goals are defined as  
*access(X, Y) :- Z =.. [X, Y], Z.*

and

*set(X, Y) :- abolish(X, 1), Z =.. [X, Y], assert(Z).*

*increment :-*  
*access(storageElement1, In),*  
*Out is In + 1,*  
*set(storageElement1, Out).*

Prolog variables must still be used for computation; they are not used for storage.

The third transformation handles the unification of two variables assigned to different storage elements. Consider

*... jump(ADR, PC), ...*

and<sup>11</sup>

*jump(X, Y) :- Y = X.*

where ADR has been assigned to storageElement1 and PC to storageElement2.

The jump clause becomes

*jump :-*  
*access(storageElement1, Temp),*  
*set(storageElement2, Temp).*

To generate this transfer code the transformer must know which variable is the source and which is the destination -- which is bound and which is unbound when the clause is entered. The analyzer provides this information.

The last two transformations deal with atomics in clause heads. The analyzer notices these atomic arguments for the transformer.

One use of atomics is for control -- for clause selection. Consider

*... execute(OP, AC, A), ...*

---

<sup>11</sup>The analyzer cannot assign the two appearances of X to the same storage element because that could cause ADR and PC in turn to have the same storage element, which might not be correct.

and

```
execute(add, In, Out) :- ...
execute(sub, In, Out) :- ...
```

The OP argument must be retained in the transformed version for Prolog control semantics to operate. The execute goal and clauses are transformed to

```
... access(storageElement1, OP), execute(OP), ...
```

and<sup>12</sup>

```
execute(add) :- ...
execute(sub) :- ...
```

This control structure can be realized in hardware with next-state selection logic.

Another use of atomics is for modelling memory, where clauses are facts, returning values. For example, a jump instruction at location 1000 that jumps to location 2000 can be represented as

```
memory(1000, jump, 2000).
```

so that, when PC is bound to 1000,

```
... memory(PC, OP, ADR), ...
```

retrieves the jump operator and the operand 2000. Such a memory reference is transformed into

```
... access(storageElement1, PC),
memory(PC, OP, ADR),
set(storageElement2, OP),
set(storageElement3, ADR), ...
```

This accessing and setting of storage elements parallels the loading and storing

---

<sup>12</sup>The access goal in this case actually has the name *lookup*, to distinguish it from a standard access. It is equivalent to access, but this different name makes the job of the analyzer in the second synthesis stage a bit easier.

of address and data registers in hardware. The transformer uses the analyzer's source-destination information to determine, for a particular argument, whether to generate an access or a set.

### 3.3. Assumptions

Some assumptions are made about the specifications given to the storage mapping stage.

First, they are assumed to be correct. The stage makes no attempt to detect semantic specification errors.

Second, they are assumed to be determinate. Backtracking has no ready hardware analog.

Third, clause heads and goals are assumed to be consistent, in terms of variable-atomic and source-destination information. Inconsistencies can in some cases be resolved.<sup>13</sup>

Fourth, all unbound variables are assumed to be instantiated in the first goal in which they appear. This is used in generating source-destination information.

### 3.4. Improvements

Various modifications and extensions can be made.

First, variable lifetime analysis can be added to the analysis phase. This will allow safe register reuse and goal reordering.

---

<sup>13</sup>An atomic takes precedence over a variable. Different versions of clauses can be created to deal with inconsistent goal uses -- each goal effectively makes a different clause.

Second, the stage can be made to support library modules that can be used by a specification but that have a register model of storage.

Third, the system can be made to support user interaction, both passive database queries and active register reassignment. Consistency guarantees will have to be supported.

Fourth, more complex specifications may uncover uses of Prolog that will require other transformations.

Fifth, the use of non-argument variables may have to be treated with more sophistication. Complex expressions may produce temporary values that must be stored.

#### **4. Operation Scheduling and FSM Construction**

The second stage of synthesis in ASP takes a register-based executable Prolog specification and produces the input to the low-level part of ASP, a structural description containing a finite state machine and a data path. The analysis phase of the stage assigns Prolog goals to hardware operations and FSM states, and generates FSM state transition information. The transformation stage binds hardware operations and operands to physical functional units, and uses the FSM information generated in the first phase to create a structural description.

##### **4.1. The Analyzer**

The analyzer scans the input specification, converting each goal into an abstract hardware operation associated with an FSM state.

Each hardware operation is stored in the analysis database and has the form

$$op(\langle source \rangle, \langle connection \rangle, \langle destination \rangle, \langle state \rangle).$$

For example, the input specification goals

$$\dots access(regAC, Temp), \\ set(regPC, Temp), \dots$$

would set the program counter to the contents of the accumulator. These two goals become the operation

$$op(regAC, Temp, regPC, stateI).$$

In a more complicated example, the goals

$$\dots access(regAC, Temp1), \\ access(memDR, Temp2), \\ Temp3 \text{ is } Temp1 + Temp2, \\ set(regAC, Temp3), \dots$$

would add the contents of the accumulator and the memory data register and store the result back in the accumulator. These goals become

$$op(regAC, Temp1, +, stateJ). \\ op(memDR, Temp2, +, stateJ). \\ op(+, Temp3, regAC, stateJ).$$

These are abstract operations -- no actual hardware elements are allocated. The *Temps* and *+* have not been bound to actual buses or an ALU. Note the assumption that, in a single state, a register can be read, an addition performed, and that same register written.

As the analyzer processes goals it also accumulates state transition information. It only records transitions that alter normal linear control flow. These transitions can be conditional or unconditional.

Consider the execute example in the previous section. It consists of a case dispatch to a collection of instruction-specific goals. The dispatch is a conditional transition; the return from a case arm is an unconditional one.

Unconditional transfers are stored as

*transfer(<from-state>, uncond, <to-state>).*

Conditional transfers have the form

*transfer(<from-state>, cond, <test>).*

where *<test>* is the source of the value that will drive the dispatch. Each arm is stored as

*case(<from-state>, <value>, <to-state>).*

For example, the execute dispatch example is represented as

*transfer(stateK, cond, regOP).*

and

*case(stateK, add, stateL).*  
*case(stateK, sub, stateM).*

...

and the end of the case arms appear as

*transfer(stateL, uncond, stateZ).*  
*transfer(stateM, uncond, stateZ).*

...

From these database entries the FSM of the next section can be constructed.

The analysis database is a complete representation of the specification. It could serve as input to a simulator that evaluated *op*, *transfer* and *case* entries.

## 4.2. The Transformer

The transformer allocates functional units and buses to the abstract operations created by the analyzer, and generates a data path and FSM structural description. Specific information about functional units, timing, and bus structure is concentrated in the transformer.

The transformer makes two passes over the hardware operation information.

The first pass binds abstract operations to functional units. For example, it maps arithmetic operations to ALU's with the appropriate functionality. It also maps abstract connections to buses. Both these processes are simple in the current system, but can easily be much more sophisticated. At the end of the first pass the *element*, *control*, and *bus* declarations, described in the next section, are produced.

The second pass generates *state* definitions. Abstract operations are translated into functional unit control signals and, for each state, packaged with next state information.

## 4.3. Improvements

Four classes of improvements can be made to the second stage. They all increase its sophistication.

The first class involves enhancing the analyzer, with the goal of increasing the concurrency in the final design. The system currently allows only one store per state. The first enhancement is the addition of dependency analysis, deter-

mining which hardware operations depend on the results of others. This analysis establishes basic constraints on what can be performed in parallel.<sup>14</sup> The second enhancement is the use of the dependency analysis to schedule parallel operations.<sup>15</sup> Different scheduling algorithms will be investigated (see [HAL], [Sehwa], [MAHA], and [Mimola]).

The second class consists of broadening the data path architectures supported by the system. The current system assumes a simple bus structure, with two source buses and a result bus, and a MacPitts-like master-slave style. To

---

<sup>14</sup>A basic block in a Prolog specification is a sequence of code uninterrupted by conditional execution or tail recursion. Conditional constructs are clause invocation (only where one clause among several may be selected), if-then-else, and disjunction.

Since Prolog is a single-assignment language, the only dependencies that occur in a basic block are data dependencies. There are three types of operations: register load and store; targeted computation, where a destination register is specified; and untargeted computations, usually comparisons appearing as the first goals in a clause and used for clause selection. All three can introduce dependencies.

The analyzer currently locates control structures, divides the specification accordingly, and relates sequences to their enabling conditions. Disjunction and if-then-else are not handled initially.

The analyzer will get timing estimates for all operations (from the library or the user) and associate timing constraints with each basic block.

<sup>15</sup>Data dependencies, connectivity restrictions, and area limitations together set a lower bound on speed and concurrency. Numeric bounds will be set on all three characteristics -- the maximum number of clock phases, the maximum number of buses, and the maximum data path area. One bound will be optimized with respect to the other two. Hard and soft bounds may be set -- a soft bound can be exceeded when synthesis would otherwise fail. A greedy technique could be used for all improvements, based on As Early As Possible scheduling ((CMU-Facet)).

Basic blocks will be taken initially as fixed boundaries; operations will not be moved from one block to another.

Buses will be shared as much as possible, within dependency constraints.

Computational unit sharing is similar to bus sharing. Area and connectivity constraints limit the availability of computational units. Optimization may require functional change. For example, a register may become a counter if it must be incremented during a phase when no ALU is available.

support concurrency (pipelining in particular), the system will have to synthesize more complex architectures, with more sophisticated bus and timing models.

The third class consists of increasing the number and complexity of the functional units known to the stage. The stage needs a range of functional units to choose from, in order to select the one that best meets the requirements of a given set of abstract hardware operations. Area, time, and topology are important as well as functionality. And more complex functional units are needed to support more complex architectures.

The fourth class involves making control of the stage more sophisticated, in particular the control of architecture and functional unit selection and concurrency optimization. Control can be extended both by heuristics and by human interaction with the synthesis process (see [ADAM], [Ulysses], and [Expert]). In addition, failure and backtracking may be employed, but the potential design space is so large that such feedback will have to be carefully controlled.

Two techniques are necessary for the success of these improvements: the use of low-level information, and the parameterization of synthesis.

Pure top-down synthesis will not produce high quality designs. Critical path analysis and area/time tradeoffs are crucial to good designs, and are not possible without low-level area and time data. In particular, details about various functional units must be known (see [BottomUp]).

It is important that the synthesis process not become the random application of a collection of ad hoc heuristics; otherwise the process becomes hard to control. In order to avoid this, synthesis tasks should be carefully parameterized and modularized, so that alternatives may be easily evaluated and chosen.

### 5. A Structural Description Mechanism

The ASP project requires a structural level of description, above the geometric level of low-level physical synthesis [CHS], but below the level of behavioral Prolog specifications. This structural level serves a similar purpose to the DDL level in the OCCAM system, providing a target for high-level synthesis, but is at a lower level of abstraction.<sup>16</sup>

All current synthesis systems ([MacPitts-Manual] and [CMU-DP] are prime examples) partition low-level descriptions into control and data paths. This is natural, because data path elements (operators and operands -- computational units and registers) are highly connected, and control logic is highly connected, while the connections between the two paths are reduced. Also, data path elements have a different layout regularity paradigm (arrays of units connected by buses) than control (masses of combinational logic, PLA's, or ROM).

All systems also use some sort of FSM realization for control. MacPitts specifications are FSM-based at the highest level; in the OCCAM to CMOS system FSM's are introduced at the DDL level. If FSM's are used they must be explicit at the structural level.

---

<sup>16</sup>A structural description explicitly represents connections between hardware elements DDL is, in this sense, behavioral. The gating logic connecting elements is implicit.

The low-level part of ASP will generate layouts. It will manage geometric and circuit information and synthesis. Tying this mechanism to higher synthesis levels poses three requirements. Connectivity information must be introduced (see [BDL] and [Ada] for discussions of structural issues). Geometric and electrical information should in general be hidden from higher levels. And individual functional blocks of logic equations must be created.

### 5.1. The Data Path

Data path definition is similar to variable declaration in a programming language. Instances of element types are created and given names. In addition (unlike variable definition), the connectivity between elements must be established.

In detail, each element declaration has the form

```
element(<type>, <name>,
        [<list-of-data-input-signals>],
        [<list-of-data-output-signals>],
        [<list-of-control-input-signals>],
        [<list-of-control-output-signals>],
        <arguments>).
```

This is a traditional variable declaration: an instance of *<type>* is created and associated with *<name>*. The lists of signals indicate connections to be made with other parts of the design. For example,

```
element(alu_ripple, dalu,
        [busA, busB], [busR], [aluFn], [aluOverflow]).
```

creates a ripple-carry ALU and binds it to *dalu*.

Note that this example demonstrates one way of managing types, tying together type name and parameter (*alu\_ripple*). An alternative would be to

pass the kind of ALU as an argument, such as

*element(alu, dpalu, ..., ripple).*

Either form can be used, depending on the needs of the type designer.

For every control input signal mentioned in an element statement, a declaration of the form

*controlIn(<signal>, <default-input>, [<list-of-inputs>]).*

is required. Control input signals are connected to and driven by the control path. This declaration defines the signal's default value and all other possible values it can have. The number of values defines the bitwidth of the signal.

For example,

*controlIn(aluFn, passA, [add, sub, ...]).*

would appear with the ALU element definition above.

In a similar manner,

*controlOut(<signal>, [<list-of-outputs>]).*

defines all the outputs of a control output signal. Such signals serve as input to the control path.

Elements in the data path are tied together with buses via

*bus(<name>).*

such as

*bus(busA).*

This indicates that the named bus is a global signal shared among several elements.

Widths of elements and signals can be declared using the form

*dataWidth(<name>, <integer>).*

Depending on the items involved, this number may be constrained to certain values.

## 5.2. The Control Path

Control information is specified in finite state machine style. Associated with each state are the control lines to be driven and conditional next state transitions.<sup>17</sup>

In particular, each state has the form

*state(<name>, [<list-of-outputs>], <next-state>).*

The *<name>* is the name of the state. The list of outputs consists of pairs of the form

*output(<value>, <name>)*

where the *<value>* is the value to be output, and *<Name>* designates the signal to be driven. Both *<value>* and *<name>* must be defined in a control statement. The *<next-state>* can either be a state name or a conditional branch of the form

*branch(<test-signal>, [<list-of-cases>])*

where *<test-signal>* is an output control signal. Each element in the *<list-of-cases>* has the form

*case(<value>, <state>)*

---

<sup>17</sup>Multi-phase clocks are not currently supported. They can be either by dividing a state into phases for control line purposes or by defining multiple phase-conditioned states.

where `<state>` is the next state if `<test-signal>` is equal to `<value>`. Both the signal and all its values must be defined in a `controlOut` statement.

All the state, control, and element statements will be passed to the low-level part of ASP for synthesis. The control path will be synthesized from the set of state definitions. The representation used will be chosen by the control synthesizer.<sup>18</sup> In addition, a simulator will be constructed that will operate on the set of statements and verify the functionality of the generated design.

### 5.3. Improvements

The above mechanism is a prototype. As experience with synthesis and with large descriptions is gained, features needed to support both will be added.<sup>19</sup> Such additions fall into four categories.

The first category is enhanced control. Subroutines, supported by the MacPitts system, can be implemented. The default value in the `controlIn` statement can be extended with enabling conditions. And the mapping of values to output signals can be more complex. In particular, bit steering and disjoint subfields can be provided. Any additions must be made in conjunction with the control simulator and synthesizer.<sup>20</sup>

---

<sup>18</sup>The CADDY system, for example, uses a control generator that produces random logic, PLA's, or microcode (ROM), depending on the nature of the transition table.

<sup>19</sup>DDL is in general a good source of ideas.

<sup>20</sup>There is a tension here. On the one hand, the generator needs representational freedom to optimize control. On the other hand, the semantics of a control construct are tied to its implementation. How much detail should be visible at this level is not obvious.

The second category is concurrency, meaning, at this level, co-operating FSM's. One issue this raises is the passing of control between FSM's (addressed, for example, in [SFL]). Another is the control of shared resources (discussed in [CADDY-CHDL]) and the passing of data between FSM's (dealt with for asynchronous FSM's in [Silc]). The functionality added here will depend on the needs of higher-level specifications.

The third category is modularity -- packages of control and data.<sup>21</sup> Modularity requirements will come from above (human needs) and below (library needs). It is important not to pay an implementation cost for modules (as is paid in [SBL]), since they are not semantically significant.<sup>22</sup>

The fourth category is low-level parameterization (such as timing,<sup>23</sup> area, and electrical constraints). Some parameters will be passed to the library via the *element* construct. Some will be of more general applicability (the CADDY system allows certain parameters to be globally specified once). The nature and form of these parameters will be determined by the needs of the library and control generator.

---

<sup>21</sup>ASP input is hierarchical, but not currently modular. This is in part because Prolog. ASP's specification language, is not modular. Modularity and hierarchy are both important to human-oriented design systems, for both detail management and design verification [FormalConlan]. In general, simulation-oriented systems have complex hierarchies, verification systems have simple hierarchies (making verification easier), and synthesis systems, from the point of view of the human designer, have no hierarchy (the input specification is at one level).

<sup>22</sup>A macro facility is one low-cost solution.

<sup>23</sup>Possible time parameters are absolute intervals and delays (in nanoseconds), relative intervals and delays (in clock phases or cycles), and precharging and edge triggering. See [Ella] and [Conlan].

#### 5.4. The Interface to Lower-Level Synthesis

High-level synthesis in ASP concerns itself with, among other things, the composition of functional units. The generation of those units is a primary concern of lower-level synthesis. It will be done by a collection of specialized tools. This collection is a generalization of a cell library. A generation tool may use standard cells, parameterized cells, or module generation from logic equations. It will accept constraints from higher levels, and answer queries about the functional unit it is responsible for (typically area/time tradeoffs). The form and content of these experts is currently under investigation. Packaging synthesis knowledge with functional units means that local, specialized expertise can be used in addition to the global analysis and control used by current systems. For example, an ALU tool can cause the carry chain to be synthesized first, since that is the critical path, and failure there is more likely. Module experts should also be able to simulate module functionality as well as generate it.

## References

- [Ada]  
'ADA as a Hardware Description Language: An Initial Report'; M.R. Barbacci, S. Grout, G. Lindstrom, M. Maloney, E. Organick, D. Rudisill; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 272-302.
- [ADAM]  
'A Design Utility Manager: the ADAM Planning Engine'; David W. Knapp, Alice C. Parker; *23rd Design Automation Conference, 1986*; pp. 48-54.
- [BDL]  
'Block Description Language (BDL): A Structural Description Language'; Eric Sluta, Glen Okita, Jeanne Wiseman; *21st Design Automation Conference, 1984*; pp. 238-244.
- [BottomUp]  
'Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions'; Michael C. McFarland; *23rd Design Automation Conference, 1986*; pp. 474-480.
- [CADDY-CHDL]  
'Synthesizing Circuits from Behavioral Level Specifications'; W. Rosenstiel, R. Camposano; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 391-403.
- [CADDY-DAC]  
'Synthesis Techniques for Digital System Design'; R. Camposano; *22nd Design Automation Conference, 1985*; pp. 475-481.
- [Cascade]  
'Overview of the CASCADE Multi-Level Hardware Description Language and its Mixed-Mode Simulation Mechanism'; D. Borrione, C. le Faou; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 239-260.
- [CHS]  
'Design Considerations for a Prolog Silicon Compiler'; Patrick C. McGeer et alia; November 1985.
- [CMU-BLT]  
'Behavioral Level Transformation in the CMU-DA System'; Robert A. Walker, Donald E. Thomas; *20th Design Automation Conference, 1983*; pp. 788-789.
- [CMU-DAA83]  
'The VLSI Design Automation Assistant: Prototype System'; T.J. Kowalski and D.E. Thomas; *20th Design Automation Conference, 1983*; pp. 479-483.
- [CMU-DAA85]  
'The VLSI Design Automation Assistant: What's in a Knowledge Base';

T.J. Kowalski and D.E. Thomas; *22nd Design Automation Conference, 1985*; pp. 252-258.

[CMU-DP]

'A Method of Automatic Data Path Synthesis'; Charles Y. Hitchcock III, Donald E. Thomas; *20th Design Automation Conference, 1983*; pp. 484-489.

[CMU-Facet]

'Facet: A Procedure for the Automated Synthesis of Digital Systems'; Chia-Jeng Tseng and Daniel P. Siewiorek; *20th Design Automation Conference, 1983*; pp. 490-496.

[Conlan]

'The Conlan Project: Concepts, Implementations, and Applications'; Robert Piloty and Dominique Borrione; *IEEE Computer*; February 1985; pp. 81-92.

[DDL]

'A Digital System Design Language (DDL)'; J.R. Duley and D.L. Dietmeyer; *IEEE Transactions on Computers*, Vol. C-17, No. 9; September 1968; pp. 850-861.

[Ella]

'The Design Rationale of Ella, a Hardware Design and Description Language'; J.D. Morison, N.E. Peeling, T.L. Thorp; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 303-320.

[Expert]

'An Expert-System Paradigm for Design'; Forrest D. Brewer, Daniel D. Gajski; *23rd Design Automation Conference, 1986*; pp. 62-68.

[FormalConlan]

'The Application of CHDL's to the Abstract Specification of Hardware'; H. Eveking; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 167-178.

[HAL]

'HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis'; P.G. Paulin, J.P. Knight, E.F. Girczyc; *23rd Design Automation Conference, 1986*; pp. 263-270.

[ISPS]

'Instruction Set Processor Specifications (ISPS): The Notation and Its Applications'; Mario R. Barbacci; *IEEE Transactions on Computers*, Vol. C-30, No. 1; January 1981; pp. 24-40.

[MacPitts-Manual]

'An Introduction to MacPitts'; J.R. Southard; *MIT Lincoln Laboratory Project Report RVLSI-3*, February 1983.

[MacPitts-Intro]

'MacPitts: An Approach to Silicon Compilation'; Jay R. Southard; *IEEE Computer*, December 1983; pp. 74-82

**[MAHA]**

'MAHA: A Program for Datapath Synthesis'; Alice C. Parker, Jorge T. Pizarro, Mitch Mlinar; *23rd Design Automation Conference, 1986*; pp. 461-466.

**[Mimola]**

'A New Synthesis Algorithm for the MIMOLA Software System'; Peter Marwedel; *23rd Design Automation Conference, 1986*; pp. 271-277.

[n.2] 'The n.2 System'; Greg M. Ordy, Charles W. Rose; *20th Design Automation Conference, 1983*; pp. 520-526.

**[OCCAM-CMOS]**

'OCCAM to CMOS: Experimental Logic Design Support System'; T. Mano, F. Maruyama, K. Hayashi, T. Kakuda, N. Kawato, T. Uehara; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 381-390.

**[SBL]**

'An Algebraic Approach to the Specification and Realization of VLSI'; G.C. Gopalakrishnan, D.R. Smith, M.K. Srivas; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 16-38.

**[Sehwa]**

'Sehwa: A Program for Synthesis of Pipelines'; Nohbyung Park, Alice C. Parker; *23rd Design Automation Conference, 1986*; pp. 454-460.

**[SFL]**

'An RTL Behavioral Description Based Logic Design CAD System with Synthesis Capability'; Y. Nakamura, K. Oguri, H. Nakanishi, R. Nomura; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 64-78.

**[Silc]**

'The Silc Silicon Compiler: Language and Features'; T. Blackman, J. Fox, and C. Rosebrugh; *22nd Design Automation Conference, 1985*; pp. 232-237.

**[Ulysses]**

'VLSI CAD Tool Integration Using the Ulysses Environment'; Michael L. Bushnell, S.W. Director; *23rd Design Automation Conference, 1986*; pp. 55-61.