ISSN 0105-8517

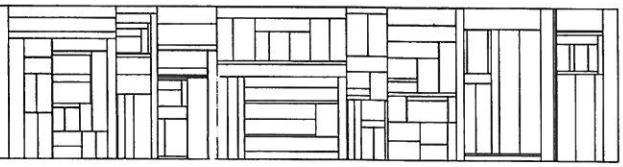
Proceedings of the

Third HOL Users Meeting

Aarhus University, 1–2 October 1990

DAIMI PB – 340
December 1990

COMPUTER SCIENCE DEPARTMENT
AARHUS UNIVERSITY
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark



Foreword

This is a collection of papers, notes and copies of transparencies representing the talks and discussions of the Third HOL Users Meeting held at The Computer Science Department, Aarhus University 1 - 2 October 1990.

Glynn Winskel

Contents

- Programme.
- Abstracts of talks.
- Copies of transparencies and relevant papers.

THIRD INTERNATIONAL HOL USERS MEETING
Aarhus University, Denmark

PROGRAMME

DAY 1 (Tuesday, October 1)

		page
09.00-10.00	REGISTRATION	
10.00-10.15	Welcome: Glynn Winskel, Aarhus University, DK.	
10.15-10.45	Mike Gordon, Cambridge University, UK. Current Research using HOL at Cambridge.	1 – 6
10.45-11.15	Tom Melham, Cambridge University, UK. A short overview of Version 1.12 of the system.	7 – 20
11.15-11.30	BREAK	
11.30-12.00	Konrad Slind, University of Calgary, Canada. Details of a new HOL implementation in SML.	21 – 34
12.00-12.30	R.D. Arthan, ICL, UK. A High-Assurance Implementation of HOL.	35 – 46
12.30-14.00	LUNCH BREAK	
14.00-15.00	Elsa Gunter, AT & T Bell Labs, USA. The Implementation and Use of Abstract Theories in HOL.	47 – 64
15.00-15.30	Myla M. Archer, University of California, USA. A Tree-editor Interface to HOL.	65 – 98
15.30-16.00	BREAK	
16.00-16.30	Aarhus students. A Mutually Recursive Types Package in HOL.	99 – 114
16.30-17.00	Malcolm Newey, The Australian National University, Australia. Another Iteration in Arithmetic.	115 – 122
17.00	Discussion on future developments of HOL.	
19.00	CONFERENCE DINNER (Jacob's Barbeque)	

DAY 2 (Tuesday, October 2)

AUTHOR: M.J.C. Gordon

TITLE: Current Research using HOL at Cambridge

SHORT ABSTRACT:

Discussion of some activities with HOL since the last meeting.

AUTHOR: Tom Melham

TITLE: A short overview of Version 1.12 of the system.

AUTHOR: Konrad Slind

TITLE: Details of a new HOL implementation in SML.

AUTHOR: R.D. Arthan

TITLE: A High-Assurance Implementation of HOL

SHORT ABSTRACT:

ICL are engaged in an IED sponsored project to develop af highly assured version of HOL for use in industrial applications. The paper will report on this work.

AUTHOR: Elsa L. Gunter

TITLE: The Implementation and Use of Abstract Theories in HOL

SHORT ABSTRACT:

At the last HOL meeting, I proposed that a notion of abstract theories be added to HOL. This year, I will discuss an implementation of that proposal which was implemented mostly on top of the existing notion of theories in HOL. We will discuss what modification were made to the underlying system, why they were made, why it is sound to have made them, and in what sense are they upwards-compatible. We will discuss how abstract theories are used in a variety of setting. Finally, time allowing we will discuss some of

the ways in which this implementation falls short of giving the user the full functionality that one might expect from abstract theories and what it would take to overcome these shortcomings.

AUTHOR: Myla M. Archer

TITLE: A Tree-editor Interface to HOL

SHORT ABSTRACT:

We have developed a tree-editor interface (PM) to HOL, based on an Emacs-based programmable tree editor (Treemacs) developed at the Univ. of Illinois. PM ("proofmanager") permits convenient access to subgoals and convenient proof display, and maintains proof soundness. There are additional useful features, and plans for extending the tool.

AUTHOR: Malcolm Newey

TITLE: Another Iteration in Arithmetic.

SHORT ABSTRACT:

I have used Tom Melham's axiomatisation of natural numbers (the one appearing in HOL88 1.11) but have changed the theory so much that I will (no doubt) incur the wrath of many HOL users when I suggest we make a change to this important standard fragment of the Basic HOL System. Nevertheless, I do intend to argue that the benefits of a major change in this area, even at this late stage, are worth the inconvenience that will be caused to users with a substantial investment in proofs using the current package.

AUTHOR: Tom Melham

TITLE: A Mechanized Theory of the pi-calculus in HOL.

SHORT ABSTRACT:

This talk will describe work in progress on the construction of a mechanized formal theory in HOL of the pi-calculus, a process algebra developed by Milner and others for modelling communicating systems in which processes can have changing structure. The aim is to support reasoning in the pi-calculus about applications as well as proofs about the pi-calculus itself.

AUTHOR: Flemming Andersen

TITLE: A Definitional Theory of UNITY in HOL

SHORT ABSTRACT:

Higher Order Logic is powerful enough to model the axiomatic definition of the UNITY logic as defined in [Chandy, Misra: Parallel Program Design, A Foundation. Addison Wesley 1988].

A method for defining relations in HOL using fixpoint equations is presented. It is shown that the UNITY relation 'leadsto', which models the graph of a transitive and disjunctive closure of binary relations, is a special case of these relations.

AUTHOR: Kim Dam Petersen

TITLE: Construction of the P ω lambda calculus model in HOL

SHORT ABSTRACT:

The P omega model may be constructed in Higher Order Logic, using the description given in [Barendregt: The Lambda Calculus, North-Holland 1984].

The construction of the model in HOL is presented, and an outline of solving selected reflexive domain equations using the model is given.

The final goal is to make a type package that is able to solve general reflexive domain equations.

AUTHOR: Wim Ploegaerts

TITLE: "HOL theory for finite word length arithmetic"

SHORT ABSTRACT:

The ultimate goal of the ongoing research in our group is the verification of a silicon compiler for DSP (Digital Signal Processing) applications. As a first step, it has been tried to built the libraries that are required for the definition of the behavior of hardware components (such as adders, multipliers, ...) in terms of integer numbers. The work includes e.g. an extgention of the library "zet" (definition of integer division and remainder, a normalization conversion for integer expressions and some related tactics) and the

definition of the denotation and representation functions for both signed and unsigned arithmetic.

AUTHOR: Jeff Joyce

TITLE: More Reasons Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware

SHORT ABSTRACT:

The HOL community is often challenged to justify the choice of higher-order logic as a specification language over conventional description languages such as VHDL or less expressive formalisms such as first-order logic. The question, "why higher-order logic?" was partly answered by Gordon's 1985 paper, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware". Our (proposed) presentation will discuss why higher-order logic, in particular (as opposed to less expressive formalisms such as first-order logic), is a very good formalism for specifying and verifying hardware. We focus on two main reasons: (1) the ability to support generic specifications of hardware, (2) the ability to embed special-purpose formalisms such as temporal logic.

AUTHOR: Catia Marcondes Angelo

TITLE: "The Verification of a Parameterized Mead & Conway Alu Core in HOL"

SHORT ABSTRACT:

This work is concerned with the formal verification of the parameterized Mead & Conway Alu module core implemented in the Cathedral II Silicon Compiler using the Proof Assistant HOL from Cambridge University. The correctness proof is divided in two parts: the structural and the behavioral reasoning. The structural verification consists of a formal alu decomposition reflecting the basic design process knowledge to prove the hardware implementation behaves like an ideal alu. The behavioral reasoning formally verifies the ideal alu meets the specification.

AUTHOR: Richard J. Boulton

TITLE: The HOL Verification of ELLA Designs

SHORT ABSTRACT:

ELLA is a hardware design language developed at the Royal Signals and Radar Establishment (RSRE) and marketed by Praxis. It supports simulation models at a variety of different abstraction levels.

A preliminary methodology for reasoning about ELLA designs using HOL is described. Our approach is to semantically embed a subset of the ELLA language in higher order logic, and then to make this embedding convenient to use with parsers and pretty-printers. There are a number of semantic issues that may impact on the ease of verification. We discuss some of these briefly. We also give a simple example to illustrate the methodology.

AUTHOR: Ton Kalker

TITLE: HOL Semantics of SILAGE

SHORT ABSTRACT:

SILAGE is the input language for a prototype silicon compiler for DSP applications. An exercise to endow SILAGE with a formal semantics in HOL logic revealed several weak points of SILAGE. In the presentation I will argue that designing a language like SILAGE will benefit from using mathematical methods.

Overview of Current Activities

Hardware Verification Group

University of Cambridge

October 1990

Developments of HOL

- HOL88.1.12 to be released in November (details in talk by Tom Melham)
- New edition of documentation in progress
  - REFERENCE to be completed
  - DESCRIPTION and TUTORIAL revised and extended
  - Quick reference cards in preparation
  - Compatible with Calgary HOL
  - Slides for HOL courses to be distributed with the system
- X-windows based demo tool (John Van Tassel) and theorem retrieval system (Richard Boulton) available

Kinds of Project

- Small projects
  - Typically a PhD student or individual researcher
  - Not supported by an external grant
- Large Projects
  - Several researchers, at least one full–time
  - Some external support

Small Projects

- Protocol verification
- Architecture specification
- π–calculus
- Compositional proof systems
- Formalisation of real–time systems
- VHDL semantics

Large Projects

- I/O hardware verification
- Viper UART & VISTA translator verification
- Proof analysis and accounts
- Verified proof-checker
- SAFEMOS
- Processor verification
- CHEOPS
- HOL verification of ELLA designs

Protocol Verification

- **Research Goal:**
  Verification of code implementing protocols
- **Research Methods:**
  Specifications and real–time programs are modelled in HOL
- **Status:**
  In progress for 2 years with a preliminary case study available as part of the HOL documentation
- **Researcher:**
  Rachel Cardell–Oliver — PhD Student at Cambridge, Cadet in Australian DSTO

Computer Architecture Specification

- Research Goal:
  Augment informal architecture specs with formal ones to reduce problems of ambiguous spec meaning, and to allow the use of theorem provers for testing those meanings
- Research Methods:
  To formalise industrial-style specs, express them in HOL, and prove properties of them
- Status:
  Going for 1.5 years with some simple examples completed Annotated bibliography on architecture specification has been produced
- Researcher: Tim Leonard — PhD student, VAX architect with DEC

π–Calculus in HOL

- Research Goals:
  Provide proof support for the π–calculus (higher–order successor to Milner's CCS) in HOL, check proofs of existing meta–theorems, and investigate the calculus for modelling systems
- Research Methods:
  Define the calculus syntax as a recursive concrete type and specify its semantics via structured operational semantics. Existing informal proofs will then be replicated in HOL
- Status: Just started (syntax defined in HOL, some simple lemmas proved)
- Researchers:
  Tom Melham and Mike Gordon

Compositional Proof Systems

- **Research Goals:**
  To build tools to support inductive definitions of compositional proof systems and automatic generation of induction tactics

- **Research Methodology:**
  Driven by the need to define structured operational semantics based on minimal relations. The π–calculus and other process algebras will be used as examples

- **Status:**
  Some examples already done interactively. Automatic tools about to be written

- **Researchers:**
  Tom Melham and Juanito Camilleri

Formalisation Of Real–Time Systems

- Research Goals:
  Define a small language containing explicit timing constructs as defined by a deterministic scheduler. This environment should be seen as implementing some of the ideas from the world of process algebras, but with explicit real–time characteristics

- Research Methods:
  Specify a scheduler in HOL and verify its safety and liveness properties. An evaluation of the constructs based on this approach will be done along with a final definition of the syntax and a formalisation of the semantics

- Status: Early days

- Researcher: Neil Viljoen – PhD student

VHDL Semantics

- **Research Goal:**
  To define a subset of VHDL tractable for formal methods, to specify its semantics in HOL and hence to develop a set of proof tools

- **Research Methods:**
  Build on the results and lessons of the HOL–ELLA project with the understanding that VHDL is a much more complex language

- **Status:**
  To start on 1 October 1990

- **Researcher:**
  John Van Tassel — PhD Student

I/O Hardware Verification

- Research Goal:
  To design and verify a UART whose initial target is the VIPER, and will serve as a prototype of Transputer link verification (SAFEMOS)

- Research Methods:
  Specification and verification to be as general as possible so that the proofs can hopefully be reused. Fabrication will take place with XILINX user–programmable gate arrays

- Status:
  Design and verification complete; fabrication to be conducted as part of the HOL–ELLA project

- Researcher:
  John Herbert

VISTA Translator Verification

- **Research Goal:**
  Define the semantics of VISTA in HOL. Define and verify a translator from VISTA to VIPER machine code written in the HOL logic

- **Research Method:**
  Define the syntax and a direct denotational semantics of both source and target language in HOL with the translator as a primitive recursive function

- **Funding**
  MOD (RSRE)

- **Researcher:**
  Paul Curzon

Proof Analysis

- **Research Goal:**
  Better understanding of the relationship between proofs and tactics. Implementation of prototype tools to generate proof summaries from tactics

- **Research Methods:**
  Detailed study of "naturally occurring" proofs and replicating them in HOL. Experimental programs to generate proof narratives from tactics have be written

- **Status:**
  Funded by SERC

- **Researcher:**
  Avra Cohn

Proof Certification

- **Research Goal:**
  Understand the structure and parameterisation of large proofs; develop and test a technology for producing 'proof deliverables' (idea proposed by Newey, Hanna)

- **Research Method:**
  - Modify HOL to produce proof texts
  - Code a checker for them in a programming language supported by HOL
  - Verify the checker

- **Status:** SERC/MOD funding promised. Target start date is 1.4.91

- **Researcher:** Avra Cohn

SAFEMOS I

- **Research Goal:**
  - Demonstrate the possiblilty of totally verified systems by creating a verified application program running on a verified processor by means of a verified compiler
  - Develop a methodolgy for building verified real–time systems

- **Research Method:**
  Example driven with a programming language based on a subset of OCCAM, and the processor based on the Transputer. The proof will build on existing work such as the CLINC stack (Austin, Texas) and Joyce's work on Tamarack (Cambridge)

SAFEMOS II

- **Partners:**
  INMOS, SRI International Cambridge Research Centre, Oxford Programming Research Group, Cambridge Computer Laboratory

- **Funding**
  SERC/DTI (IED)

- **Status:**
  Started 1.1.90

- **Researchers:**
  - INMOS: David Shepard + 2
  - SRI: Roger Hale, $\frac{1}{2}$ John Herbert
  - PRG: Jonathan Bowen, Paritosh Pandya
  - CL: Mike Gordon, Juanito Camilleri

HOL–ELLA I

- **Research Goal:**
  - Specification of the formal semantics of a subset of the ELLA hardware description language in HOL with theorem–proving support for reasoning about ELLA designs
  - Development of a methodology for adding formal methods to conventional CAD

- **Research Method:**
  Translate ELLA to HOL via explicit semantics and test the methodology by designing XILINX chips

HOL–ELLA II

- **Funding:**
  Initially SERC/DTI project with Praxis. Now a pure SERC project

- **Status:**
  Running since 1.10.89, with a simple case study completed. Subset of ELLA chosen and HOL semantics defined and implemented

- **Researchers:**
  Richard Boulton (until 30.9.90), John Harrison (from 1.10.90), $\frac{1}{2}$ John Herbert

Processor Verification

- **Research Goal:**
  To design and verify a processor for real-time control applications as part of the SAFEMOS and HOL-ELLA projects

- **Research Methods:**
  Develop standard techniques which will transfer to the verification of the main Transputer–like SAFEMOS processor

- **Status:**
  Early days

- **Researcher:**
  John Herbert

CHEOPS I

- **Research Goal:**
  - Interface HOL to CATHEDRAL for the verification of synthesis functions (IMEC)
  - Improve the HOL environment through new libraries, and a better user interface (Cambridge)
- **Research Methods at Cambridge:**
  General–purpose parsers and pretty-printers driven from declarative inputs. Interface tools implemented in X–windows
- **Partners:**
  IMEC (Belgium), Philips ERL (Netherlands), Cambridge Computer Laboratory

CHEOPS II

- **Funding:**
  Esprit BRA
- **Status:**
  Running since 1.8.89 with the syntax tools complete. X–windows interface started with a prototype developed in an enhanced GNU emacs
- **Researchers:**
  - IMEC: Luc Claesen, Wim Ploegaerts, Catia Angelo
  - **Philips:** Ton Kalker
  - **Cambridge:** John Van Tassel (until 30.9.90), Sara Kalvala, Andy Gordon (from 1.1.91)

Summary

- HOL88 system is now stable
- New improved Standard ML implementations are almost ready (*Viz* HOL90 from Calgary and ICL HOL)
- Polished documentation (for both HOL88 and HOL90) coming soon
- Increased automation and improved interfaces are on the horizon
- Many projects announced at first HOL Users Meeting are now up and running
- Lots of new projects, many unrelated to hardware verification, are starting up

HOL88: Version 12 overview

- Summary of changes already made.
- Some proposed changes.

T F Melham, 28 September 1988

Current state of version 12

- Summary of changes already made:
  - preterms, and user-definable typechecking
  - revised 'resolution' tools
  - paired beta-conversion
  - new tactics, rules, etc.
  - revision/optimization of some tactics, rules, etc.
  - natural number `MOD` and `DIV` theorems added
  - total rewrite of type definition package
  - revised set theory library
  - new 'contrib' directory
  - many internal ML functions deleted/hidden

New ML type of raw terms

- A new ML type of untyped terms:

```
preterm  =   var      of string
           | const    of string
           | comb     of preterm # preterm
           | abs      of preterm # preterm
           | typed    of preterm # type
           | antiquot of term
```

has been added, together with:

```
preterm_to_term : preterm -> term
```

which invokes the standard HOL typechecker.

User-definable typechecking

- **To enable user-defined typechecking:**

```
set_flag(`preterm`,true);;
```

- **To install user-defined typechecking:**

```
let preterm_handler p = ⟨function⟩
```

where ⟨*function*⟩ has type:

```
: preterm -> term
```

- **Note: this has not been used much yet.**

Revised 'resolution' tools

- **In version 11, RES_CANON does:**

`~t`	⇒	`t ==> F` (at outermost level)
`t1 /\ t2`	⇒	`t1,t2`
`(t1/\t2)==>t`	⇒	`t1==>(t2==>t)`
`(t1\/t2)==>t`	⇒	`t1==>t, t2==>t`
`t1 = t2`	⇒	`t1=t2, t1==>t2, t2==>t1`

- **In version 12, RES_CANON also does:**

`!x. t1==>t2` ⇒ `t1==>!x.t2` (`x` not free in `t1`)

- **For example:**

```
|- !m n p. m < n /\ n < p ==> m < p
```

in version 11 gets transformed to:

```
|- !m n p. m < n ==> (n < p ==> m < p)
```

but in version 12 it gets transformed to:

```
|- !n m. m < n ==> (!p. n < p ==> m < p)
```

Effect of this change

- Given the assumptions:

  `[a < b]` and `[b < c]`

  the tactic:

  ```
  IMP_RES_TAC |- !m n p. m < n /\ n < p ==> m < p
  ```

  will produce the assumptions:

  ```
  [ !p. b < p ==> a < p ]      [ !p. c < p ==> b < p ]
  ```

  instead of freezing the variable p:

  ```
  [  b < p ==> a < p ]      [  c < p ==> b < p ]
  ```

- Consequence:

  - more new consequences generated, so
  - more assumptions to work with,
  - e.g. RES_TAC will solve the goal more often;
  - BUT: your existing proofs may not work.

Paired beta-conversion

- New function for paired beta-conversion:

```
PAIRED_BETA_CONV : conv
```

- For example, given the term:

```
"(\(x1, ... ,xn).t) (t1, ... ,tn)"
```

  the new conversion proves that:

```
|- (\(x1, ... ,xn).t) (t1, ... ,tn)
     =
   t[t1, ... ,tn/x1,... ,xn]
```

- It also works on arbitrarily-nested tuples.

Related conversion

. LET - conv
  — coming soon

Natural number division

- **In version 11:**

```
|- 0 < n ==> (?r q. (k = (q * n) + r) /\ r < n)
|- MOD k n = @r. ?q. (k = (q * n) + r) /\ r < n
|- DIV k n = @q. (k = (q * n) + (k MOD n))
```

- **In version 12:**

```
|- 0 < n ==> (?r q. (k = (q * n) + r) /\ r < n)
|- 0 < n ==>
   (k = ((k DIV n)*n)+(k MOD n)) /\ (k MOD n) < n)
```

- **Plus built-in theorems, for example:**

```
|- !k. k MOD (SUC 0) = 0
|- !n. 0 < n ==> (!k. (k DIV n) <= k)
|- !n r. r < n ==> (!q. ((q * n) + r) DIV n = q)
|- !n k. k < n ==> (k MOD n = k)
|- !n. 0 < n ==> (!k. (k * n) MOD n = 0)
|- !n. 0 < n ==> (0 MOD n = 0)
```

- **Other arithmetic theorems have also been added.**

Type definition package

- Has been totally rewritten.
- The code is now:
  - Cleaner and better organized,
  - faster (sometimes much faster),
  - much better documented.
- The main changes visible are:
  - modifications to low-level tools,
  - `define_type` now has a proper parser,
  - different variable names/priming behaviour.
- Coming soon:
  - more general function definitions

Revised set theory library

- **Reorganization:**

Old name	New name	Description
`sets`	`finite_sets`	finite sets
`all_sets`	`sets`	finite and infinite sets
`set`	`pred_sets`	predicates-as-sets

- **The library `sets` now supports the syntax:**

```
"{x1, ... ,xn}"
"{x | P[x]}"
"{t[x1,...,xn]] | P[x1,...,xn]"
```

- **Proof support for this notation (for example):**

```
SET_SPEC_CONV "t in {x | P[x]}"
```

returns:

```
|- t in {x | P[x]}  =  P[t/x]
```

- **The parser/prettyprinter support is general.**

The contrib directory

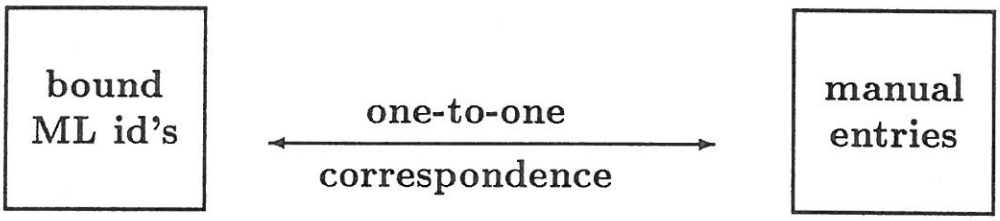
- Contents: user contributions
- Purpose:

  for easy distribution of items such as code, theories, and documentation which are useful but not suitable for the library

- Current contents:
  - CPO: complete partial orders (jac1)
  - PNF: prenex normal form (cj)
  - batch-hol-tool: simulate interactive session (rjb)
  - franz-cl-th: translate theories to common lisp (jac)
  - knuth-bendix: equational reasoning (slind)
  - select: how to deal with epsilon (grahamb)
  - temporal: temporal logic in HOL (jasuja)
  - tooltool: suntools interface (des)
- Further contributions very welcome!

Hiding/Deleting Internal Functions

- The goal is:



- Many identifiers already deleted/hidden.

- For example:

Redundant	Old versions	Unused
AND_CLAUSE1	HOL_IMP_RES_THEN	form_class
AND_CLAUSE2	HOL_MATCH_MP	form_frees
AND_CLAUSE3	HOL_PART_MATCH	form_tyvars
AND_CLAUSE4	HOL_RESOLVE_THEN	form_vars
AND_CLAUSE5	HOL_RES_THEN	
	OLD_RES_TAC	*(etc)*
(etc)		
	(etc)	

Proposed changes/developments

- **Future developments and proposed changes:**
  - rationalize resolution
  - faster rewriting
  - automatic abstract type definitions
  - new type-definition primitive
  - library revision
  - new version of the manuals

An SML Implementation of hol88

Konrad Slind
+
Graham Birtwistle,
Rajagopal Nagarajan,
Todd Simpson

Funded by: - NSERC Strategic Grant '89-'92
Verifying Hardware Designs

- Equipment grants from
NSERC and CMC

hol88

- Lisp + ML + ML compiler
- HOL (logic) implemented using PPLAMBDA code. Requires "bending" of HOL objects.
- primitive routines for terms are intricately coded and deserve to be held to the light.
- comments : sparse, multilingual, ol°... and now for a tricky bit ?
- theory files have obscure, Lisp-dependent syntax
- unclear as to how the system builds
- SML

hol90

- "done" (v.1.11 \ contributed libraries)
- provides same interface
- ~15,000 lines of SML
- ~1.3 M. core
- Speed. Not emphasized.
  - faster in forward inference.
  - slower in parsing terms
  - very slow in reading from theory files.
- in PolyML
- Core Language only.

Which SML?

- ◇ all_of_them

- Implementation requires features not in the Standard:

  1. Embedding HOL in SML:

     - New top level with macro expansion phase

typical top level:

```
let top_level() =
    ( prompt();
      print ( eval ( read ()));
      top_level ());
```

New top level:

```
fun top_level () =
    ( prompt ();
      print ( eval ( read ( read_HOL () ) ) );
      top_level () );
```

```
let tm = "∀ ∧ ∀"
in
  ⋮

⇒
  let tm = mk_conj ( mk_var ('∀', mk_type "bool"),
                     mk_var ('∀', mk_type "bool") )
  in
    ⋮
```

Poly ML allows the installation of user-defined top levels

ML-yacc

2. pretty printing terms, types, theorems.

Gives "final" top level:

```
fun top_level () =
    ( prompt ();
      print (eval ( read (read_HOL ())));
      top_level ());
```

POLYML allows the extension of the pretty printer of SML.

Why no structures:

- had used them up to a point in time, but
  1. gc costs, core size
  2. signature maintenance
  3. Flakiest part of implementation
- We plan to add them before releasing HOL90

A tour of the implementation

- divides neatly between - ML/Lisp
  - doc'ed / undoc'ed

ML

Libraries (Sets, groups, processes, ...)

More powerful definitional mechanism
(Recursive types

Basic theories (N, sum, one, ...)

Backwards proof interface (goal stack)

Conversions, tacticals, tactics

Derived rules of inference

:thm ADT (primitive rules of inference, axioms)

basic definitional mechanism

Lisp

theory interface

parsing HOL (quotation, antiquotation, top level)

type inference, pretty printing

Symbol table

terms

types

Archaeological Results

- hol88 uses variable sharing
  and
- conversion nets for fast rewriting
- "bootstrapping"
- we actually have an ADT of thm
- only primes when it has to
- antiquote suppresses type inference:

```
let v = "v:bool"
in
"^v ∧ (v=1)"        passes
```

but

```
"(v:bool) ∧ (v=1)"    fails
```

- ADT thm shouldn't use quotation,
  i.e.

```
REFL tm = mk_thm([], "^tm = ^tm")
```

for the correctness of thm depends on correctness of read_HOL.

- aconv is "dynamic deBruijn".

  aconv "λx.(x:bool)" "λx.(x:*)" ⇒ true

  can we get inconsistency?

- hol88 uses property lists, we have a symbol table
  some of hol88 term handling routines
  are hardwired in, e.g. mk_conj
  inherited from LCF

- some cleanup, for example theory of
  pairs - we can do backwards proof
  of some theorems there, instead of
  using mk_thm.

- theory file interface cleaned up. Parser
  info not held as theorems.

- theory cache.

Speed Revisited

Completing the group

Sun3	1731 thms	218 s.	hol88
		31 s.	hol90/PolyML
		16 s.	hol90/NJSML
		10 s.	sun4/NJSML.

To do

- make parsers identical
- speed up
  - parser,
  - theory file interface
  - conversion nets for rewriting
- move to modules system
- attempt to move to NJSML
- better DS/A for rewriting
- redesign theory interface
- hooks for UI work

ML → SML

- let x = ... in let y = ... in

→

```
let val x = ...
    val y = ...
in
...
end
```

- letrec or functional let → fun
- let x = e → val x = e
- ;; → ;
- [a; b; c] → [a, b, c]
- [a,b] = [(a,b)] → [a, b] — beware
- Cambridge allows leaving the else arm off conditional. Not SML

- # used for different purposes in SML

- $(1,2,3) \equiv (1,(2,3)) \longrightarrow (1,2,3)$ beware! semantic diff.

- Exceptions much more powerful in SML

- & ⟶ andalso
  or ⟶ orelse

- b => e1 | e2 ⟶ doesn't exist

- \x.e ⟶ fn x => e

  \xyz. e ⟶ fn x => fn y => fn z => e.

  fn is an SML keyword, so don't use it for variable naming.

- '0 ... '0 ⟶ (* ... *)

  Common problem:

  (* function f has type (* -> *) -> bool
  *)

  taken as end of comment!

Cooperation Anyone?

- many interesting aspects of this work.

- If you want to dive in and add to the system, we would be happy.

ICL

A High Assurance Implementation of HOL

Rob Arthan
ICL Secure Systems

1

ICL

The FST Project

(DTI Supported Project IED/1563)

Cambridge University

- foundational research

ICL Secure Systems

- lead partner
- HOL technology and applications

Kent University

- VERITAS+

Program Validation Limited.

- SPARK

ICL

Background

- Business is high-assurance secure systems.
- HOL used in applications for about 4 years
- Proof deliverables produced on several real secure system developments
- Piecemeal work to tailor the system to our needs:
  - Tied off loop-holes (e.g. mk_thm)
  - Metalanguage and theory support (Z, hardware DA links, methods support)

ICL

Directions

Want to improve capability to produce formal proofs of critical properties in real applications.

HOL offers a tried and trusted basis.

We would like:

- Greater integrity (possibility of accidental or malicious proof of F)
- Better support for the languages our customers want (Z, Ada)
- Improved ease of use (proof work is currently very costly)

ICL

High Assurance HOL

Basic idea (i.e. long term goal):

(a) formally specify logic

(b) formally specify critical properties of the proof development system in terms of (a)

(c) verify implementation (or at least design against (b).

Result is a proof development system with assurance level analogous to that of a US A1+ (= UK level 6) secure system.

Want to take sensible and useful steps towards this.

ICL

State of Play

- Have formal specifications of
  - language and inference rules
  - theory hierarchy
  - semantics

(= chapters 9&10 of the manual)

- Have a prototype system in Standard ML
  - Logical primitives based on formal spec
  - Theorem proving superstructure based on Cambridge documentation and implementation
  - Basis for experiments with proof automation etc.

ICL

Current Work

- designing a system to be implemented to product standards
- rationalising logic spec and planning out integrity proof
- modest steps in proof automation
- support for Z
- user interface issues (e.g. subgoal package improvements)
- compatibility and interchange

ICL

Logic Specification

- Specified HOL as a deductive system.
  - language
  - rules of inference
  - theories with theorems defined via derivability
  - PDS as a machine with a theory hierarchy in its state, making "derivable" transitions on the theories in it
- Treatment is quite long (30 pages of theory listing without consistency proofs)
- Now looking at incorporating semantics and reorganising to ease proof task

8

ICL

TOP LEVEL SPEC OF THE PDS

HOL_STATE

parent : string→string→bool
theory : string→THEORY

(antisymmetric(ancestral parent)) ∧
(rooted parent) ∧
(∀init·root parent init ⇒ theory init = INIT)
(order_preserving theory parent extends)

HOL_SYSTEM ≅ (*INPUT × HOL_STATE) →
(HOL_STATE × *OUTPUT)

safe : (*INPUT, *OUTPUT) HOL_SYSTEM → bool

∀hol_system·safe hol_system ⇔
∀input hol_state·
consistent_state hol_state ⇒
consistent_state(FST(
hol_state(input, hol_state)))

Similarly for conservative extensions etc.

ICL

Proof Automation

- Many useful modest steps to be made
- Incorporate useful techniques from research on other theorem provers
- Have prototyped:
  - Several tautology proving algorithms
  - Divide-and-conquer techniques for finding normal forms (e.g. polynomials over a commutative ring)
  - Tools for justifying various forms of definition
- Future work
  - Decision procedures etc.
  - Improved support for recursive types

ICL

User Interface

- Main concerns are "deep" interface issues rather than "surface" ones
- Want a consistent and comprehensive collection of tools within HOL.
- Have made some small experiments
  - Subgoal package using a theorem to represent the state (gives automatic validation of tactics safely, and other useful features).
  - Term surgery-style tools
  - Cross-referencing tools
- Future work
  - User interfaces for other languages (e.g. Z)
  - Better support for programming tactics and conversions

The Implementation and Use of Abstract Theories in HOL.

Elsa Gunter
Bell Labs

What do we want from abstract theories?

Abstract theory should create a context of assumed information, and give that information scope.

Within an abstract theory, attempts to prove theorems have access to that information.

Outside abstract theory, access to that information is removed. Theorems proved using it have had it abstracted out.

To use a theorem from within an abstract theory outside, must supply abstracted information.

What will having abstract theories do for us?

Allow for the development of general mathematical theories.

Allow for modularity and cleaner organization of proofs in "theorem proving in the large".

Could be used in organizing verification proofs to make the implementation-specification layers clear.

(2)

More Specifically, what do we want from abstract theories?

An abstract theory should take a signature consisting of

- list of type variables
- list of term variables
- list of assumptions

and create a scoped context in which, when proving theorems, the type variables and term variables are treated as type and term constants, and assumptions are treated as axioms.

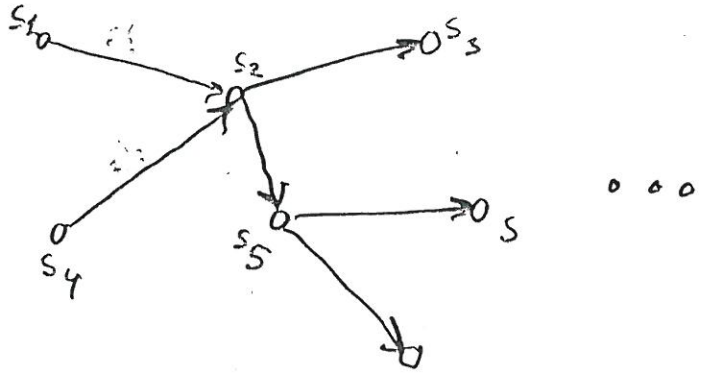
(3)

A Definitional Theory of UNITY in Higher Order Logic

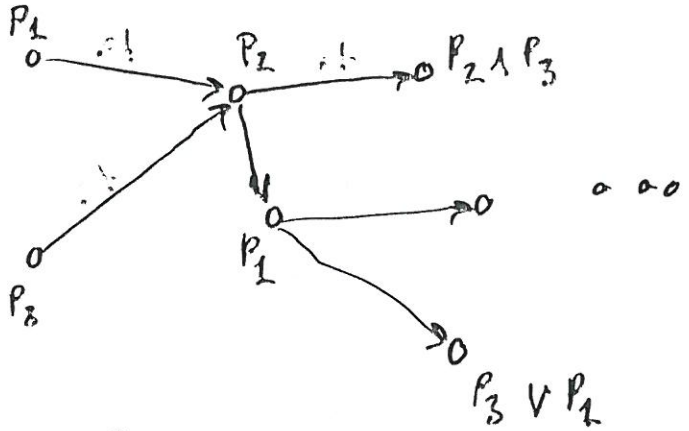
Overview

- Programs as state-transitions and their properties
- Defining program properties in Higher Order Logic
- The transitive & disjunctive closure of progress
- Introducing bounds on progress properties
- Looking for a generalised closure definition
- How UNITY is derived
- Conclusion

Programs as state transitions



Program Properties



Safety, R_S

Liveness (progress), R_T

Direct, inferred properties

Defining program properties : Higher Order Logic

Safety

$p \, R_S \, q \equiv$

$\forall s, st \in Prog : R_{safe}(p, q, st, s)$

(satisfied for all possible states)

Chandy, Misra :

p unless $q \equiv$

$\forall s, st \in Prog : (p\,s \wedge \neg q\,s) \Rightarrow p(st\,s) \vee q(st\,s)$

Liveness (progress)

$p \, R_T \, q \equiv$

$\forall s : \exists st \in Prog : R_{trans}(p, q, st, s)$

(express the existance of a state transition)

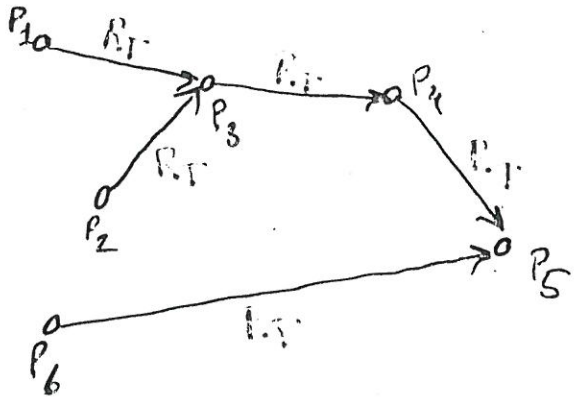
Chandy, Misra:

p ensures $q \equiv$

p unless $q \wedge$

$\forall s : \exists st \in P_{...} . , s \wedge \neg q\,s \Rightarrow q(st\,s)$

The transitive and disjunctive closure of progress



What properties are wanted:

$$p_1 R_T p_3 \Rightarrow p_1 R_C p_3$$

$$p_1 R_T p_3 \wedge p_3 R_T p_4 \Rightarrow p_1 R_C p_4$$

$$p_1 R_T p_3 \wedge p_2 R_T p_3 \Rightarrow p_1 \vee p_2 R_C p_3$$

An axiomatic definition:

$$\frac{p R_T q}{p R_C q}, \quad \frac{p R_C q, q R_C r}{p R_C r}, \quad \frac{p_1 R_C q, p_2 R_C q}{p_1 \vee p_2 R_C q}$$

A Definition in Higher Order Logic

$$p R_C q =$$
$$p R_T q \vee$$
$$(\exists r: p R_C r \wedge r R_C q) \vee$$
$$(\exists p_1, p_2: (p = p_1 \vee p_2) \wedge p_1 R_C q \wedge p_2 R_C q)$$

Proving the definition satisfies the axioms:

Prove that

1) $p R_T q \Rightarrow p R_C q$

2) $p R_C r \wedge r R_C q \Rightarrow p R_C q$

3) $p_1 R_C q \wedge p_2 R_C q \Rightarrow p_1 \vee p_2 R_C q$

Trivially from the definition of R_C

But more properties may be wanted in the logic.

A Primitive Recursive Definition of R_C

$$\text{RelClose } 0\ R_T\ p\ q = p\ R_T\ q \wedge$$

$$\text{RelClose } n{+}1\ R_T\ p\ q =$$
$$(\exists r:\ \text{RelClose } n\ R_T\ p\ r \wedge \text{RelClose } n\ R_T\ r\ q)$$

$$\forall (\exists p_1 p_2:\ (p = p_1 \vee p_2) \wedge \text{RelClose } n\ p_1\ q \wedge \text{RelClose } n\ p_2\ q)$$

$$p\ R_C\ q = \exists n:\ \text{RelClose } n\ R_T\ p\ q$$

Again, prove that the three axioms and wanted theorems actually are satisfied.

Introducing bounds on progress

$$\frac{\forall M, m : \; p \wedge (M = m) \; R_c \; (p \wedge M \prec m) \vee q}{p \; R_c \; q}$$

We now need to generalise disjunction, since the proof of the above theorems assumes $(\exists m : M = m)$.

Hence, the disjunction theorem (axiom) becomes:

$$\frac{\forall i : \; (P_i) \; R_c \; q}{(\exists i : P_i) \; R_c \; q}$$

Generalising the definition of R_C

$$p \; R_C \; q =$$
$$p \; R_T \; q \;\vee$$
$$(\exists r : \; p R_C r \wedge r \; R_C \; q) \;\vee$$
$$(\exists P : \; (p = \exists i : P_i) \wedge (\forall i : (P_i) \; R_C \; q))$$

NB

Can we find a primitive recursive definition of this function:

$$\text{InfRelClose } 0 \; R_T \; pq = p R_T q \; ?$$
$$\text{InfRelClose } n+1 \; R_T \; pq =$$
$$\{$$
$$(\exists P : \; (p = \exists i : P_i) \wedge (\forall i : \text{InfRelClose } n \; R_T \; (P_i)q)\}$$
$$p \; R_C \; q = \exists n : \text{InfRelClose } n \; R_T \; pq$$

NB, since we can't prove

$$(\forall i : (P_i) \; R_C \; q) \Rightarrow (\exists i : P_i) \; R_C \; q$$

A pure relational approach to R_C

GenRelClose R =

$$(\forall p,q:\ p\,R_T\,q \Rightarrow p\,R\,q) \wedge$$
$$(\forall p,q:\ (\exists r:\ p\,R\,r \wedge r\,R\,q) \Rightarrow p\,R\,q) \wedge$$
$$(\forall p,q:\ (\exists P:\ (p = \exists i:\ P_i) \wedge (\forall i:\ (P_i)\,R\,q) \Rightarrow p\,R\,q))$$

$$p\,R_C\,q = \forall R : \text{GenRelClose}\ R \Rightarrow p\,R\,q$$

This definition actually satisfies the three axioms, but unfortunately we cannot prove

$$\frac{p\,R_C\ \text{false}}{\neg p}$$

However we *do* have: GenRelClose R_C

and

$$p\,R_C\,q = p\,R_T\,q \vee (\exists r:\ p\,R_C\,r \wedge r\,R_C\,q) \vee (\exists P:\ (p = \exists i:\ P_i) \wedge (\forall i:\ (P_i)\,R_C\,q))$$

How UNITY is derived

1) Take unless, ensures as defined before

2) Specialise the function which models the transitive and disjunctive closure of „single" progress relations (R_T) by the ensures relation.

3) Prove all the missing theorems valid for UNITY

4) Make (define) the appropriate representation of UNITY-programs as statetransitions

5) thats it !

Conclusion

- A complete UNITY-system except for the shown induction theorem and with only finite disjunction has been implemented using the primitive recursive function RelClose for modelling *leadsto*.
- It is an open question (to me) whether infinite disjunction and hence the proper induction theorem is actually sound.
- It was discovered that the substitution rule is only valid if the logic primitives *unless*, *ensures* and *leadsto* are restricted to the states reachable by the program. This means that HOL-UNITY doesn't have this rule!

Solving Reflexive Domain Equations in HOL

Kim Dam Petersen

TFL

kimdam@tfl.dk

Contents

Reflexive Domain Equations

- Reflexive domain equations has the form: $D = \mathcal{E}(D)$, where $\mathcal{E}$ is composed from basic domains Bottom, Boolean and Number by Point, Pair, Union and Function contructions
- A solution of a domain equation is a domain D, which is isomorphic to $\mathcal{E}(D)$
- Example: $C = * \rightarrow C \rightarrow **$

Method for solving reflexive domain equations

- Use the D_∞ domain constructions [Schmidt]
- Construct type free λ calculus [Barendregt]
- Construct P^ω [Barendregt]

D_∞ construction [Schmidt]

- A domain is a (pointed) cpo
- All domains are contained in a universe $\mathcal{D}$ of domains
- A valid Domain Scheme $\mathcal{E} : \mathcal{D} \rightarrow \mathcal{D}$ maps pointed cpos to pointed cpos
- Construct a sequence of domains D_n by:
  $D_0 = \{\perp\}$ and $D_{n+1} = \mathcal{E}(D_n),\ n \geq 0$
- A limit D_∞ of the domain sequence exists
- A pair of contiunous functions $(\Phi, \Psi) : D_\infty \leftrightarrow \mathcal{E}(D_\infty)$ exists, such that: $\Psi \circ \Phi = Id_{D_\infty}$ and $\Phi \circ \Psi = Id_{\mathcal{E}(D_\infty)}$
- Hence D_∞ is isomorphic to $\mathcal{E}(D_\infty)$

HOL implementation problems

- Constructing an object *Domain* that represents the universe of all domains
- *Domain* should contain the basic domains Bottom, Number and Boolean
- *Domain* should be closed wrt. domain constructions Point, Pair, Union and (continuous) Function

Type free λ calculus

- Numbers, Booleans and Bottom are representable as closed λ expressions.
- Pairs, Unions and Functions may be represented as closed λ expression, if their components are
- Hence *Domain* may be represented as subsets of closed (type free) λ expressions

Constructing type free λ calculus [Barendregt]

- A reflexive cpo may be used to construct a model of type free λ calculus
- A cpo $(D, \sqsubseteq)$ is *reflexive* if there exists two functions $f : D \to [D \to D]$ and $g : [D \to D] \to D$ such that: $f \circ g = \mathrm{id}_{[D \to D]}$

The P^ω model of λ calculus [Barendregt]

- $P^\omega = (\{x | x \subseteq \mathbb{N}_0\}, \subseteq)$
- $[n, m] \leftrightarrow \frac{1}{2}(n+m)(n+m+1)+m$
- $\{k_0, ..., k_{m-1}\} \leftrightarrow \Sigma_{i<m} 2^{k_i}$
- $graph(f) = \{[n, m] | m \in f(e_n)\}$
- $graph : [P^\omega \to P^\omega] \to P^\omega$ is continuous
- $fun(u)(x) = \{m | \exists e_n \subseteq x,\ [n, m] \in u\}$
- $fun : P^\omega \to [P^\omega \to P^\omega]$ is continuous
- $fun(graph(f)) = f$ for any continuous function f
- Hence P^ω is a reflexive cpo
- Hence P^ω may be used to construct $Domain$
- Hence P^ω may be used to solve reflexive domain equations

What needs to be done

- Completion of the λ model
- Mapping basic domain values (Bottom, Booleans and Numbers) into λ expressions
- Mapping constructed domain values (Pairs, Unions and Functions) into λ expressions
- Solve reflexive domain equations using λ expressions
- Make a type package for reflexive domains

What has been done in HOL

- Theories for: cpo, topology etc. has been defined
- Properties concerning P^ω, *fun* and *graph* has been proved
- A λ model based on P^ω is being constructed

Towards a HOL theory for Finite Wordlength Arithmetic ...

Wim Ploegaerts[†]

Imec v.z.w.
Kapeldreef 75
3000 Leuven
- Belgium -

[†]**Research Assistent with the Belgian National Fund for Scientific Research**

Work partly sponsored by the CHEOPS-BRA

0. Introduction

1. Linking HOL to CATHEDRAL ...
2. The Missing Link ...
3. Some list theory ...
4. Integers: "zet" and more ...
5. The link ...
6. Conclusions ...

Linking hardware behavior to
its integer equivalent

useful extensions of the library "zet"

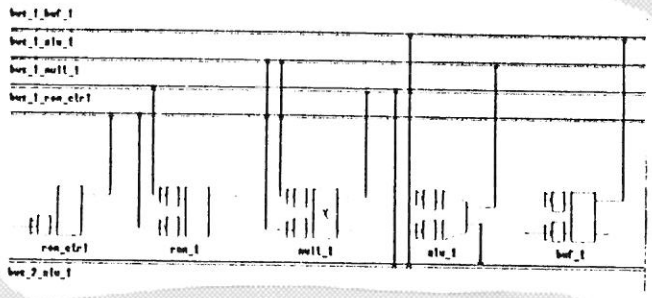
which are available and documented

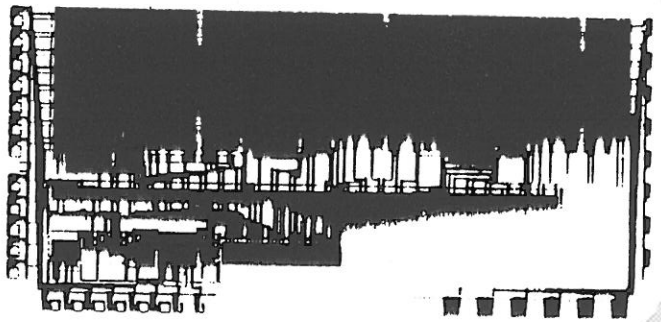
1. Linking HOL to CATHEDRAL...

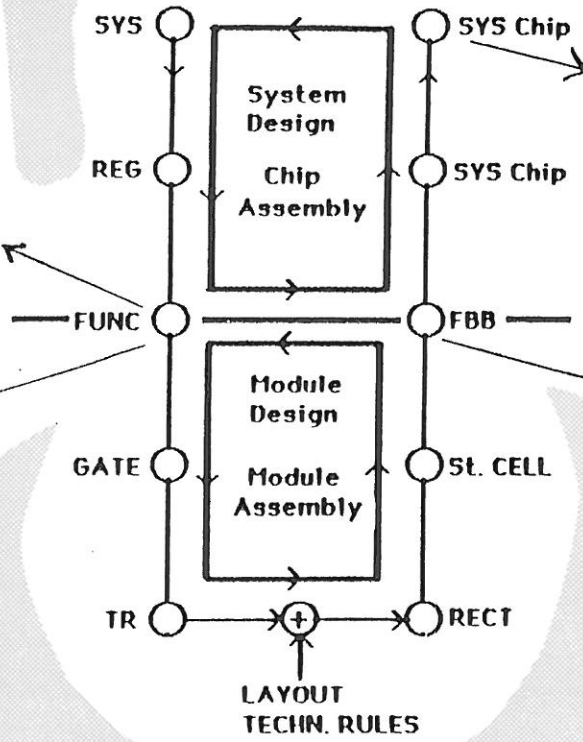
```
func main(in: WORD) out: WORD =
begin
    out1 = WORD(coef5   * biquad(in ,coef1  ,coef2  ,coef3  ,coef4  ));
    in2  = out101;
    out2 = WORD(coef50  * biquad(in2,coef10 ,coef20 ,coef30 ,coef40 ));
    in3  = out201;
    out  = WORD(coef500 * biquad(in3,coef100,coef200,coef300,coef400));
end;

func biquad(in: WORD; b0,b1,c0,c1: num): WORD =
begin
    state  = in     - WORD(state01*c1) + WORD(state02*c0);
    return = state + WORD(state01*b1) + WORD(state02*b0);
end;
```

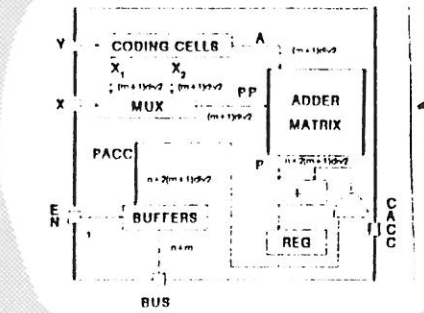
CATHEDRAL-II

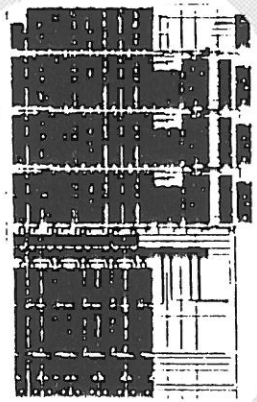
A Silicon Compiler for Multi-Processor Signal Processing Systems.











1. Linking HOL to CATHEDRAL...

CATHEDRAL...

Synthesis *Environment* for the design

of ASIC's for *DSP* applications

-> DSP = audio (CD), video, ...

-> Optimizing results by limiting the application domain

-> Cathedral 1,2,3,...

"From an idea to layout in 1 day..."

HOL...

The CHEOPS Project:

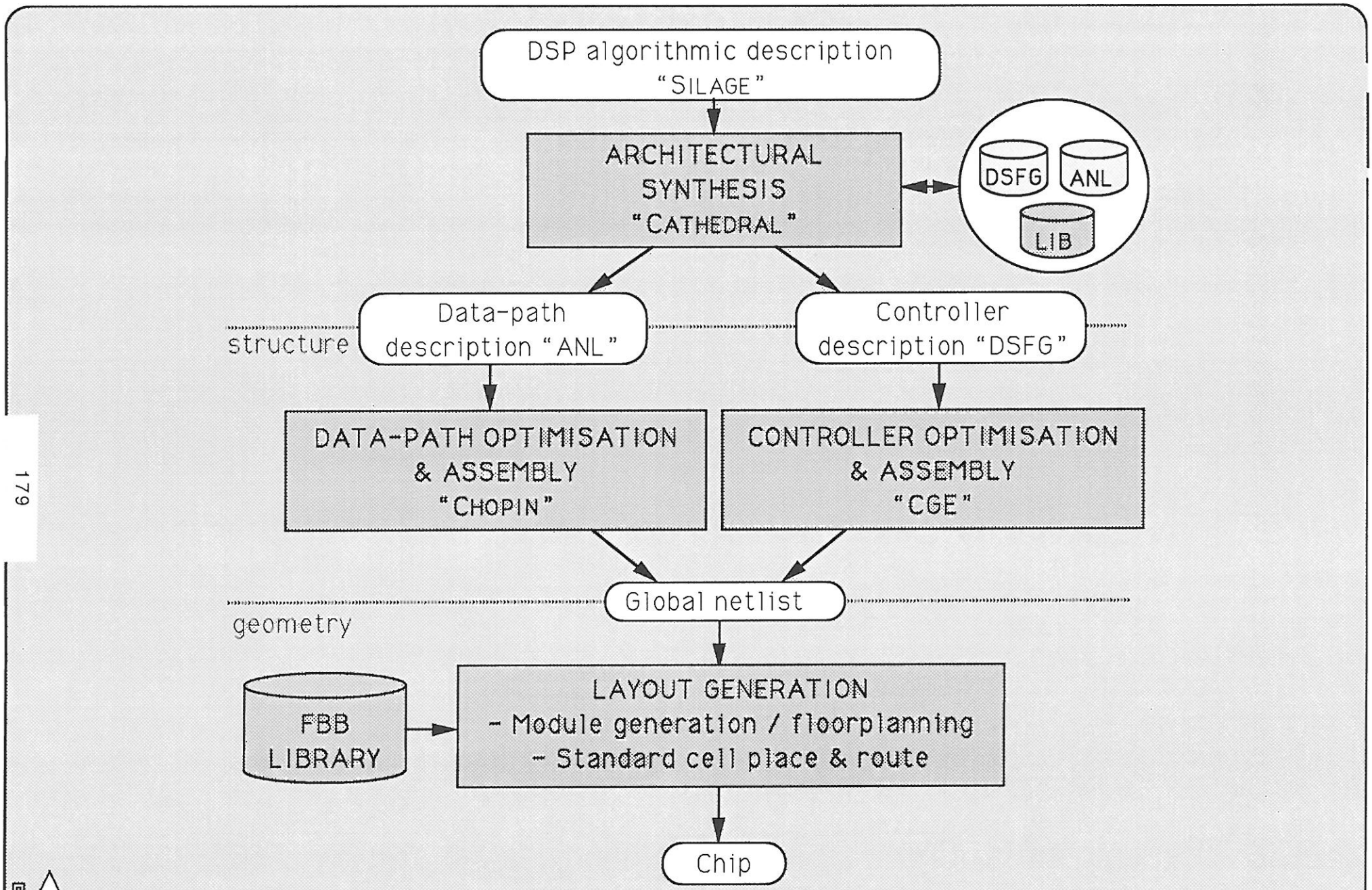
Setting up a Verication Environment for the CATHEDRAL results

Therefore

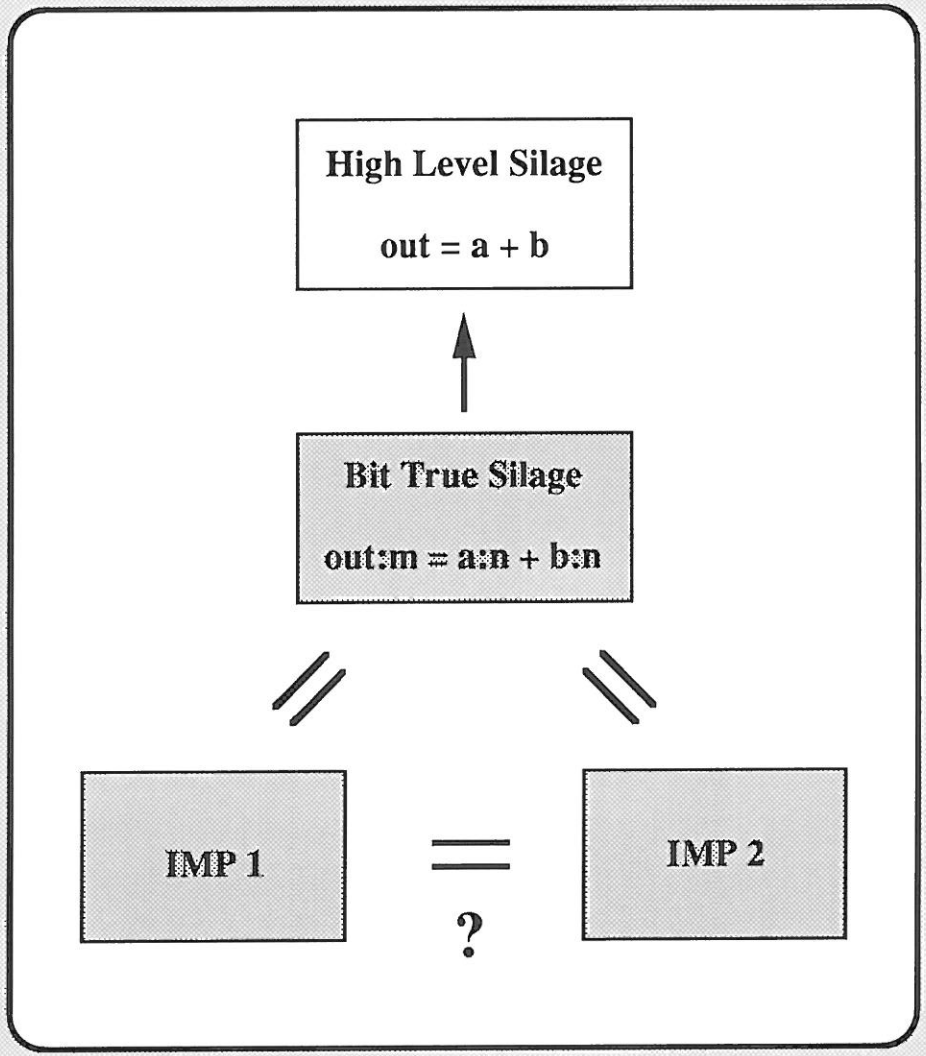
-> Realistic Complexity ...

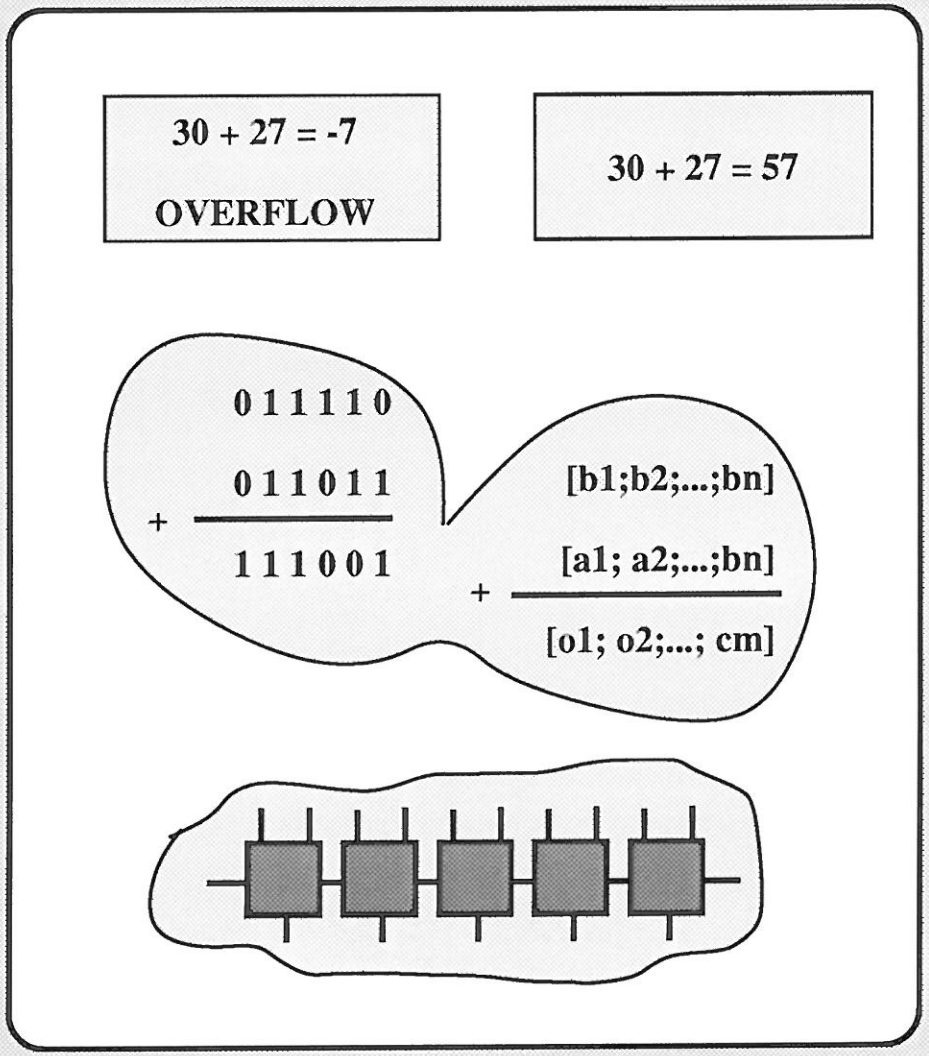
-> Automatic versus Soundness...

- A "designer" does not want to be confronted with HOL
- General methodology instead of single proof

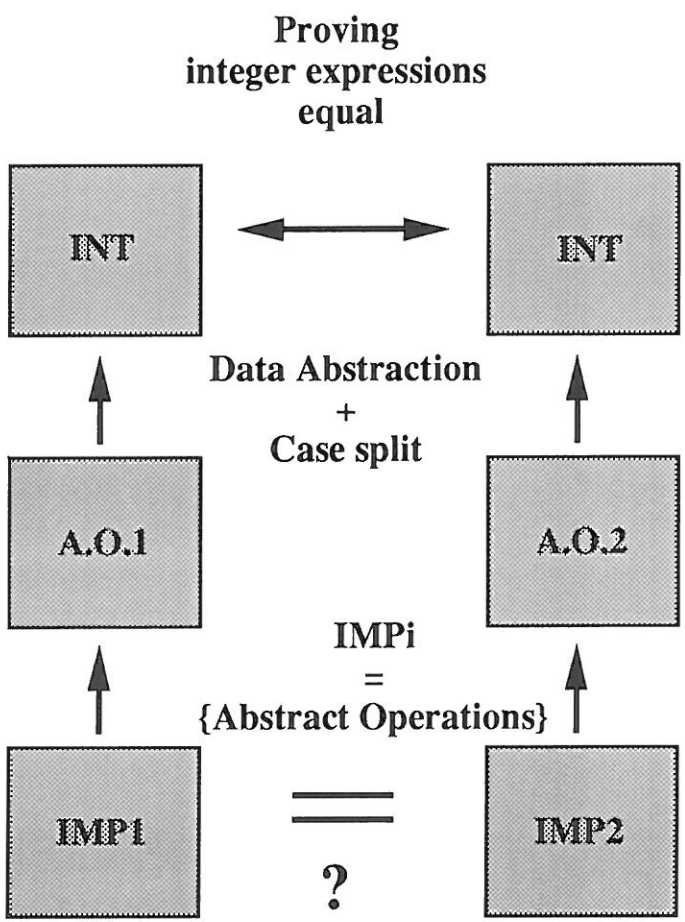

DSP algorithmic description
"SILAGE"
ARCHITECTURAL
SYNTHESIS
"CATHEDRAL"
DSFG
ANL
LIB
structure
Data-path
description "ANL"
Controller
description "DSFG"
DATA-PATH OPTIMISATION
& ASSEMBLY
"CHOPIN"
CONTROLLER OPTIMISATION
& ASSEMBLY
"CGE"
Global netlist
geometry
FBB
LIBRARY
LAYOUT GENERATION
- Module generation / floorplanning
- Standard cell place & route
Chip

2. The Missing Link ...





2. The Missing Link ...



Prerequisites ...

- powerful library on integer numbers,

  -> importance of **"2"**

- a tactic to prove that integer expressions are equal

- list-theory (variable length)

- the link from bitstrings to integers

What is available in HOL...

- list ...

- "zet" (T.Kalker)

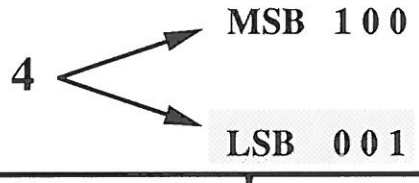
What is needed ...

2. The Missing Link ...

Implementation of

"Representational and Denotational Semantics of Digital Systems" (R.T.Boute)

- theory of number representations
- "detailed" transformational reasoning style
- functional formalism closely related to HOL



du : (:bool)string -> (:num)

(du [] = 0) ∧ (du (CONS a x)) = a' + 2 * (du x)

ru : (:num) -> (:bool) string

(ru 0 = []) ∧ (ru x = (CONS (x mod 2) (ru (x div 2))))

ds : (:bool)string -> (:zet)

(ds [] = 0) ∧ (ds (APPEND x [s]) = (du x) - s' 2 (LENGTH x)

rs : (:zet) -> (:bool)string

(rs 0 = [F]) ∧ (rs -1 = [T]) ∧

(rs m = (CONS (m mod 2) (rs (m div 2)))

3. Some List theory...

[a 1 ; a 2 ; ... ; a (n-1) ; a n]

HD ——— TL ———

——— HL ——— LAST

|- ! x f . ?! fa .

(fa [ ] = x) /\

(! h t . fa (APPEND t [h]) = f (fa t) h t)

! l y . LAST (APPEND l [y]) = y

! l y . HL (APPEND l [y]) = l

- Definitions
- induction tactic
- relation with constructs of "list "

Use: du, ds

4. Integers: "zet" and more ...

"zet" (T.Kalker)

- formalization of integers
  - -> zero (0), een (1)
  - -> addition, subtraction, multiplication, less, leq
  - -> SIGMA (Σ) , PI (π)
  - -> maximum and minimum of a subset
- recursive definitions in zet
- induction on integers
- several useful theorems, but ...

extensions of "zet"

- absolute value and sign of an integer
- *positive* power of an integer
- integer division and remainder

```
|- ! m n .
  ~ ( n = 0 ) ==>
   (m =  (( m div n) * n)  + ( m mod n )) /\
   ( 0 ≤ ( m mod n )) /\ ( ( m mod n ) < | n | )
```

-4 div 3 = -2 -4 mod 3 = 2

4. Integers: "zet" and more ...

- concept of even and odd integers

- div2, mod2, pow2

$$!\,n.\,(0<n) \Longrightarrow (?!\,k.\,(2^{k} \le n) \wedge (n < 2^{SUC\,k}))$$

- pos_log : that "k" for which ....

- log2 : related to div2

```
!n. ( log2 0 = 0 ) /\ ( log2 -1 = 0) /\
    ( ~( n = 0 ) /\ ~( n = -1)  ==>
        ( log2 n = SUC ( log2 ( div2 n ) )))
```

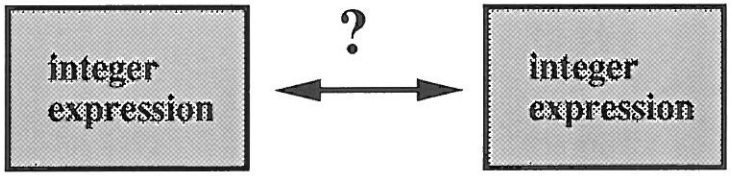
INDUCTION TACTIC

```
!Q. (( Q 0 )  /\  ( Q -1 ) /\
      (!m. Q m ==> Q ( 2 m )) /\
      (!m. Q m ==> Q ( 2 m + 1)))
      ==> (!n. Q n)
```

Related to bitstrings:

```
        100  <-> 4
1000 <->2*4   1001 <->2*4 +1
```

4. Integers: "zet" and more ...



(a + b) * (d + e) - db = e b + a d + a e

" m + n = k + l "

[" m - k = l - n "]

" a < b + c "

[" a - b ≤ c - 1 "]

NORMAL_FORM_CONV

Make normalized sum of products

exp = p1 + (p2 + (p3 .+ ...))

pi = a1 * (a2 * (a3 * ...))
= - (...)

ai = not an addition, subtraction or multiplication

make syntactical normal form

use of "<<" and bubble sorting

to order

- subterms of pi (ai)
- ordered subterms of exp

5. The Link ...

```
du  : (:bool)string  -> (:zet)

(du [] = 0) ∧ (! a x . du (CONS a x)) = a' + 2 * (du x)

ru : (:zet) -> (:bool) string

! n . (0 ≤ n) ==>

(ru 0 = [])  ∧ (ru n = (CONS (n mod2) (ru (n div2))))
```

```
ds : (:bool)string -> (:zet)

 (ds [] = 0) ∧ (! s x . ds (APPEND x [s]) = (du x) - s' 2 (LENGTH x)

rs : (:zet) -> (:bool)string

  (rs 0 = [F]) ∧ (rs -1 = [T]) ∧

  ( ! m . ~( m = 0 ) ∧ ~( m = -1 )  ==>
        (rs m = (CONS  (m mod 2)  (rs (m div 2)))
```

5. Conclusions ...

The following theories are available en documented:

- general purpose extensions of the library "zet": div, mod, pow
  (documentation of "zet" and some tactics and conversions of "auxiliary" included)
- a normalization conversion to prove integer expressions equal
- extensions of "zet" for the special case "2": div2, mod2, pow2, log2

The theory on the denotation and representation functions is available but not yet documented.

These theories are needed for the characterization of finite worldlength arithmetic.

More Reasons
Why Higher-Order Logic is a Good Formalism
for Specifying and Verifying Hardware

Jeff Joyce
University of British Columbia

October 1990

"Why higher-order logic ?"

Mike Gordon (September 1985) ...

"Why Higher-Order Logic is a
Good Formalism for Specifying and Verifying Hardware"

(Cambridge Tech. Report No. 77)

...the opening paragraph reads:

> The purpose of this paper is to show, via examples, that:
>
> 1. Many kinds of digital systems can be formally specified using the notation of formal logic; specialized hardware descriptions are not needed.
>
> 2. The inference rules of logic provide a practical means of proving systems correct; specialized deductive systems are not needed.

Two Reasons "Why Higher-Order Logic ?"

In this talk, I propose two reasons why higher-order logic is a good formalism for specifying and verifying hardware:

1. generic specification
2. embedded formalisms

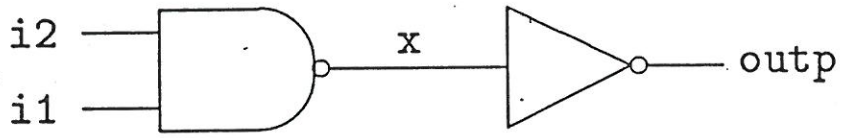
Outline of this Talk ...

1. Uses of Higher-Order Functions
   - some common uses
   - some more sophisticated uses
2. Alternatives to h.o.l.
3. Arguments against h.o.l.
4. Generic Specification
   - simple example
   - microprocessor specification
5. Embedding Other Formalisms
   - temporal logic
6. Conclusions

Some Common Uses of Higher-Order Functions ...

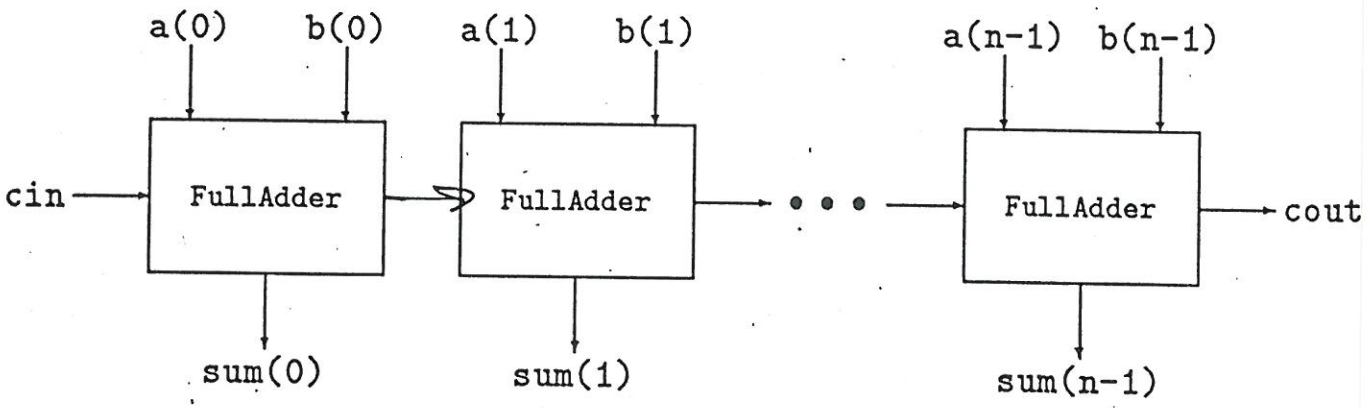
... when specifying hardware with predicates,

1. signals as functions of time



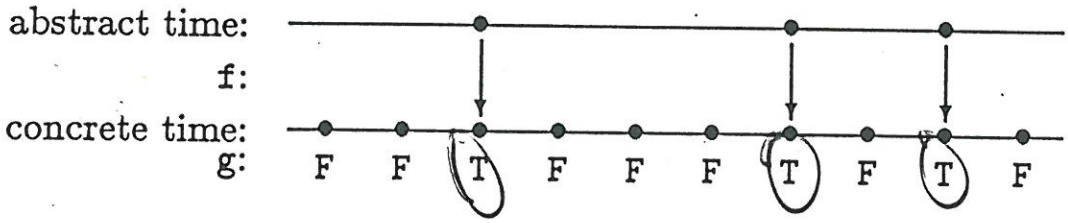

```
AndGate (a,b,out)

a,b,out : time→bool
```

2. signals as functions of position




```
RippleCarry n (a,b,cin,sumcout)

   a,b,sum   : postion →bool
   cin,cout  : bool
```

More Sophisticated Uses of Higher-Order Functions ...



... defining parameterized timing relationships,

```
TimeOf (g:time->bool)
```

```
Define (
  "(TimeOf g 0 = εt. First g t) ∧
   (TimeOf g (SUC u) = εt. Next g (TimeOf g u,t))");;

Define ("First g t = (∀p. p < t ⟹ ¬(g p)) ∧ (g t)");;

Define (
  "Next g (t1,t2) = t1 < t2 ∧ (∀t. t1 < t ∧ t < t2 ⟹ ¬(g t)) ∧ (g t2)");;
```

... and deriving generalized theorems,

For example, ...

the generalized theorem,

```
|- ∀g r.
     (∃t. g t) ∧ (∀t. g t ⟹ ∃m. Next g (t,t+m) ∧ r (t,t+m)) ⟹
     ∀u. r (TimeOf g u,TimeOf g (u+1))
```

can be used to reduce the problem of proving,

```
∀u. r (TimeOf g u,TimeOf g (u+1))
```

to a pair of simpler problems:

```
∃t. g t
∀t. g t ⟹ ∃m. Next g (t,t+m) ∧ r (t,t+m)
```

This generalized result provides the basis for a HOL tactic:

```
                    ∀u. r [TimeOf g u,TimeOf g (u+1)]
                  ========================================
∃t. g t,      ∀t. g t ⟹ ∃m. Next g (t,t+m) ∧ r [t,t+m]
```

Alternatives to Higher-Order Logic, ...

... as a hardware specification language:

- Conventional HDL's
  - generally oriented to simulation
  - limited ability to express behaviour
  - may lack underlying formal semantics
- Special Purpose Formalisms
  - fixed ideas about hardware
- First-Order Logic
  - sometimes expressibility is compromised further other goals
    e.g. disallowed "$\exists$" to make specifications executable
  - may have extra constructs reminiscent of h.o.l.
    e.g. parameterized specifications in OBJ
- [redacted] HOL
- Type Theory
  - dependent types, but undecidable

Arguments Against the Higher-Order Approach ...

- notation is difficult
- undecidability concerns
- proofs are much harder
- expressibility not needed
- specifications not executable
- the "straw man" attack

To give you an idea of their flavour ...

...with respect to higher-order programming, one author writes:

> *"Unfortunately, the logic of higher-order functions is difficult, and in particular, higher order unification is undecidable. Moreover (and closely related), higher order expressions are notoriously difficult for humans to read and write correctly."*

...and then, the same author goes on to say:

> *"Although higher-order logic cannot always be avoided in specification and verification, it should be avoided whenever possible, for the same reasons as in programming."*

Quotations from:

Josepth Goguen, "Higher Order Functions Considered Unnecessary for Higher Order Programming", 1988 April 20.

Generic Specification

In 1989, under contract to NASA, CLI sub-contracted the task of writing a review of the HOL system to David Musser of Rensselaer Polytechnic.

In his report, Musser wrote:

> *"The major weakness of HOL appears to be the lack of effective support for constructing specifications and proofs at a high level of abstraction, ..."*

and more specifically,

> *"HOL is also lacking in packaging features that would help in structuring large specifications, such as those of VIPER or more complex microprocessors"*

Musser compared HOL to SRI's EHDM system which,

> *"... provides a parameterized module capability that permits structuring specifications as modules that can be instantiated in many different ways (similarly to Ada generic packages and subprograms)."*

In this part of my talk, I will argue that:

1. generic specification does indeed offer many benefits
2. HOL logic can directly express genericity

   ... without additional constructs !

Advantages of Generic Specification ...

In addition to modularity, abstraction and re-usability, generic description can be used to filter out non-essential detail in the context of formal proof.

Filtering out non-essential details, ...

- sharpens the distinction between what has and what has not been formally considered in a correctness proof.
- supports a truely hierarchical approach to the formal verification of digital circuits where each level in a hierarchical specification is isolated from details only relevant to other levels.
- reduces the amount of special-purpose infrastructure needed to reason about particular application areas, e.g., hardware-oriented data types.

For example, ...

The formal specification of the 32-bit Viper major-state machine made extensive use of special-purpose data types and constants:

e.g., `:word4`, `:word32`, `VAL4`, `WORD32`

Consequently, these correctness results:

- were not as general as possible

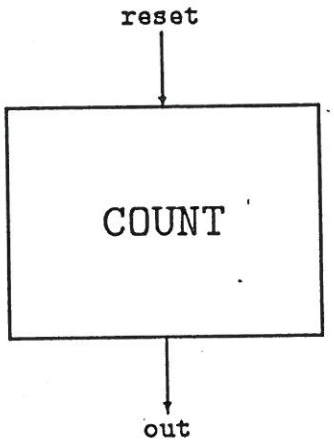
  e.g. what about a 64-bit version ?

- needed clarification about what aspects of the computational model were formally considered:

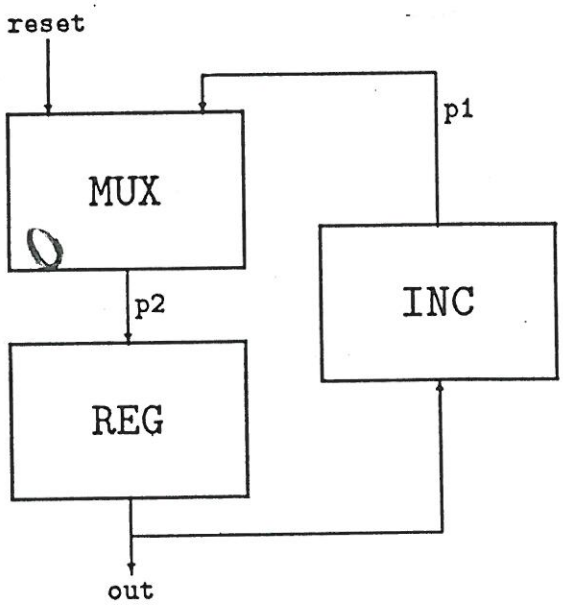
  > "There was no computation of values at the major state level - that is, additions, comparisons, shifts, and so on - so the essential correctness of Viper was not really addressed; the proof did not require any analysis of the function representing the arithmetic-logic unit, at either level".

- could not be easily re-produced in other verification systems lacking special support for reasoning about hardware

External View of the Resettable Counter,



Internal View of the Resettable Counter,



A First Attempt, ...

```
⊢def MUX (reset,i,out) = ∀t. out t = (reset t ⇒ 0 | (i t))

⊢def REG (i,out) = ∀t. out (t+1) = i t

⊢def INC (i,out) = ∀t. out t = ((i t) + 1)

⊢def COUNT_IMP (reset,out) =
       ∃p1 p2.
         MUX (reset,p1,p2) ∧
         REG (p2,out) ∧
         INC (out,p1)

⊢def COUNT (reset,out) =
       ∀t. out (t+1) = (reset t ⇒ 0 | ((out t) + 1))
```

$\vdash_{thm}$ COUNT_IMP (reset,out) $\Longrightarrow$ COUNT (reset,out)

The More Detailed, The Better ?

```
⊢def MUX (n) (reset,i,out) = ∀t. out t = (reset t ⇒ 0 | (i t))

⊢def REG (n) (i,out) = ∀t. out (t+1) = i t

⊢def INC (n) (i,out) = ∀t. out t = (((i t) + 1) MOD 2^n)

⊢def COUNT_IMP (n) (reset,out) =
       ∃p1 p2.
         MUX (n) (reset,p1,p2) ∧
         REG (n) (p2,out) ∧
         INC (n) (out,p1)

⊢def COUNT (n) (reset,out) =
       ∀t. out (t+1) = (reset t ⇒ 0 | (((out t) + 1) MOD 2^n))
```

$\vdash_{thm}$ COUNT_IMP (n) (reset,out) $\Longrightarrow$ COUNT (n) (reset,out)

Parameterizing with Function Variables, ...

```
⊢def MUX (inc) (reset,i,out) = ∀t. out t = (reset t ⇒ 0 | (i t))

⊢def REG (inc) (i,out) = ∀t. out (t+1) = i t

⊢def INC (inc) (i,out) = ∀t. out t = inc (i t)

⊢def COUNT_IMP (inc) (reset,out) =
       ∃p1 p2.
         MUX (inc) (reset,p1,p2) ∧
         REG (inc) (p2,out) ∧
         INC (inc) (out,p1)

⊢def COUNT (inc) (reset,out) =
       ∀t. out (t+1) = (reset t ⇒ 0 | (inc (out t)))
```

inc : num → num

⊢thm COUNT_IMP (inc) (reset,out) ⟹ COUNT (inc) (reset,out)

Parameterizing with Type Variables, ...

```
⊢def MUX (inc,zero) (reset,i,out) = ∀t. out t = (reset t ⇒ zero | (i t))

⊢def REG (inc,zero) (i,out) = ∀t. out (t+1) = i t

⊢def INC (inc,zero) (i,out) = ∀t. out t = inc (i t)

⊢def COUNT_IMP (inc,zero) (reset,out) =
       ∃p1 p2.
         MUX (inc,zero) (reset,p1,p2) ∧
         REG (inc,zero) (p2,out) ∧
         INC (inc,zero) (out,p1)

⊢def COUNT (inc,zero) (reset,out) =
       ∀t. out (t+1) = (reset t ⇒ zero | (inc (out t)))
```

inc : * → *

⊢thm COUNT_IMP (inc,zero) (reset,out) ⟹ COUNT (inc,zero) (reset,out)

Using "Representation Variables", ...

```
⊢def inc rep = FST rep

⊢def zero rep = SND rep

⊢def MUX (rep) (reset,i,out) = ∀t. out t = (reset t ⇒ (zero rep) | (i t))

⊢def REG (rep) (i,out) = ∀t. out (t+1) = i t

⊢def INC (rep) (i,out) = ∀t. out t = (inc rep) (i t)

⊢def COUNT_IMP (rep) (reset,out) =
       ∃p1 p2.
         MUX (rep) (reset,p1,p2) ∧
         REG (rep) (p2,out) ∧
         INC (rep) (out,p1)

⊢def COUNT (rep) (reset,out) =
       ∀t. out (t+1) = (reset t ⇒ (zero rep) | ((inc rep) (out t)))
```

rep: ((*word→*word) × *word)

inc zero

(inc rep) - "the increment operation"
(zero rep) - "the representation of zero"

$\vdash_{thm}$ COUNT_IMP (rep) (reset,out) ⟹ COUNT (rep) (reset,out)

This correctness theorem is fully generic:

$$\vdash_{thm} \texttt{COUNT\_IMP (rep) (reset,out)} \Longrightarrow \texttt{COUNT (rep) (reset,out)}$$

Recall Musser's remark that SRI's EHDM system,

> *"... provides a parameterized module capability that permits structuring specifications as modules that can be instantiated in many different ways."*

This is quite straightforward with representation variables,

... here are two simple examples,

Example 1: Idealized Counter

Here again is the idealized counter:

```
REP_num = (λx. x + 1, 0)

⊢thm COUNT_IMP (REP_num) (reset,out) ⟹ COUNT (REP_num) (reset,ou
```

...which can be expanded to:

```
⊢thm COUNT_IMP (REP_num) (reset,out) ⟹
     ∀t. out (t+1) = (reset t ⇒ 0 | ((out t) + 1))
```

Example 2: 8-bit Counter

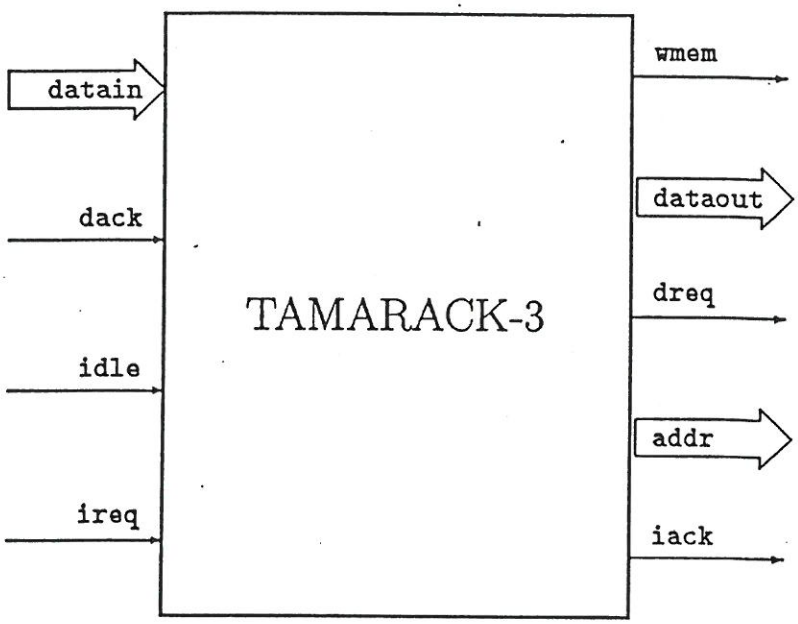
Another instance could be an 8-bit version based on built-in HOL types:

```
REP8 = (λx. WORD8 ((VAL8 x) + 1), WORD8 0)

⊢thm COUNT_IMP (REP8) (reset,out) ⟹ COUNT (REP8) (reset,out)
```

...which can be expanded to:

```
⊢thm COUNT_IMP (REP8) (reset,out) ⟹
     ∀t. out (t+1) = (reset t ⇒ WORD8 0 | (WORD8 ((VAL8 x) + 1)))
```

Generic Specification of a Simple Microprocessor ...



`datain`	- data from memory	`wmem`	- read/write select
`dack`	- data acknowledge	`dataout`	- data to memory
`idle`	- extended cycle mode	`dreq`	- data request
`ireq`	- interrupt request	`addr`	- address to memory
		`iack`	- interrupt acknowledge

Instruction	Opcode Value	Effect
JZR	0	jump if zero
JMP	1	jump
ADD	2	add accumulator
SUB	3	subtract accumulator
LDA	4	load accumulator
STA	5	store accumulator
RFI	6	return from interrupt
NOP	7	no operation

JZR - jump if zero

```
pc ← if iszero acc then inst else inc pc
```

JMP - jump

```
pc ← inst
```

ADD - add accumulator

```
acc ← add (acc,operand)
pc ← inc pc
```

SUB - subtract accumulator

```
acc ← sub (acc,operand)
pc ← inc pc
```

LDA - load accumulator

```
acc ← operand
pc ← inc pc
```

STA - store accumulator

```
mem ← store (mem,address inst,acc)
pc ← inc pc
```

RFI - return from interrupt

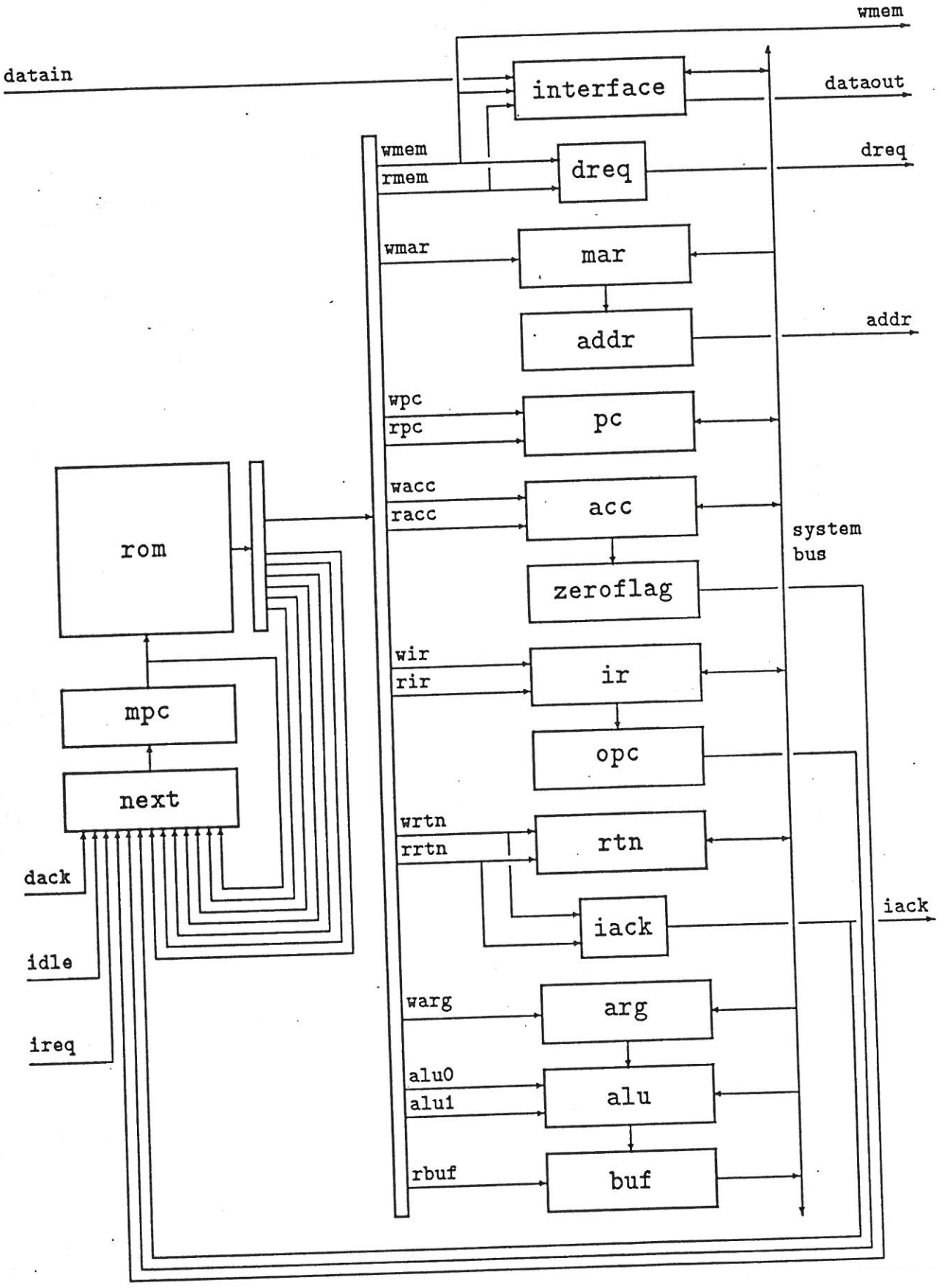
```
pc ← rtn
iack ← F
```

NOP - no operation

```
pc ← inc pc
```

Interrupt

```
if iack = F then
    pc ← 0
    rtn ← pc
    iack ← T
```


wmem
datain
interface
dataout
wmem
rmem
dreq
dreq
wmar
mar
addr
addr
wpc
rpc
pc
wacc
racc
acc
system
bus
rom
zeroflag
wir
rir
ir
mpc
opc
next
wrtn
rrtn
rtn
dack
iack
iack
idle
warg
arg
ireq
alu0
alu1
alu
rbuf
buf

Data Types:

`:bool`	- Boolean values {T,F}
`:num`	- natural numbers {0,1,2,...}
`:*wordn`	- full-size machine words
`:*word3`	- instruction opcodes
`:*word4`	- 4-bit words
`:*address`	- memory addresses
`:*memory`	- memory states

Generic Functions:

`(iszero rep)`	- "test if zero"
`(inc rep)`	- "increment"
`(add rep)`	- "addition"
`(sub rep)`	- "subtraction"
`(wordn rep)`	- "full-size word representation of a number"
`(valn rep)`	- "value of a full-size word"
`(opcode rep)`	- "extract opcode field"
`(val3 rep)`	- "value of an opcode"
`(address rep)`	- "extract address field"
`(fetch rep)`	- "read memory"
`(store rep)`	- "write memory"
`(word4 rep)`	- "value of a 4-bit word"
`(val4 rep)`	- "4-bit word representation of a number"

```
rep_ty =
  :(*wordn→bool)×                                  % iszero %
   (*wordn→*wordn)×                                % inc %
   (*wordn×*wordn→*wordn)×                         % add %
   (*wordn×*wordn→*wordn)×                         % sub %
   (num→*wordn) ×                                  % wordn %
   (*wordn→num) ×                                  % valn %
   (*wordn→*word3)×                                % opcode %
   (*word3→num)×                                   % val3 %
   (*wordn→*address)×                              % address %
   (*memory×*address→*wordn)×                      % fetch %
   (*memory×*address×*wordn→*memory)×              % store %
   (num→*word4)×                                   % word4 %
   (*word4→num)                                    % val4 %
```

```
⊢def TamarackImp (rep:rep_ty)
    (datain,dack,idle,ireq,mpc,mar,pc,
     acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr) =
    ∃zeroflag opc cntls.
      CntlUnit rep (dack,idle,ireq,iack,zeroflag,opc,mpc,cntls) ∧
      DataPath rep (
        cntls,datain,mar,pc,acc,ir,rtn,iack,
        arg,buf,dataout,wmem,dreq,addr,zeroflag,opc)

⊢def DataPath (rep:rep_ty)
    (cntls,datain,mar,pc,acc,ir,rtn,iack,
     arg,buf,dataout,wmem,dreq,addr,zeroflag,opc) =
    ∃bus busokay alu pwr gnd rmem wmar wpc rpc
     wacc racc wir rir wrtn rrtn warg alu0 alu1 rbuf.
      DecodeCntls (
        cntls,
        wmem,rmem,wmar,wpc,rpc,wacc,racc,
        wir,rir,wrtn,rrtn,warg,alu0,alu1,rbuf) ∧
      BusOkay (rmem,rpc,racc,rir,rrtn,rbuf,busokay) ∧
      Interface rep (busokay,wmem,rmem,bus,datain,dataout) ∧
      OR (wmem,rmem,dreq) ∧
      Register (busokay,wmar,gnd,bus,bus,mar) ∧
      AddrField rep (mar,addr) ∧
      Register (busokay,wpc,rpc,bus,bus,pc) ∧
      Register (busokay,wacc,racc,bus,bus,acc) ∧
      TestZero rep (acc,zeroflag) ∧
      Register (busokay,wir,rir,bus,bus,ir) ∧
      OpcField rep (ir,opc) ∧
      Register (busokay,wrtn,rrtn,bus,bus,rtn) ∧
      JKFF (wrtn,rrtn,iack) ∧
      Register (busokay,warg,gnd,bus,bus,arg) ∧
      ALU rep (alu0,alu1,arg,bus,alu) ∧
      Register (busokay,pwr,rbuf,alu,bus,buf) ∧
      PWR pwr ∧
      GND gnd

⊢def ALU (rep:rep_ty) (f0,f1,inp1,inp2,out) =
       ∀t:time.
         out t = (((f0 t,f1 t) = (T,T)) → ((inc rep) (inp2 t)) |
                  ((f0 t,f1 t) = (T,F)) → ((add rep) (inp1 t,inp2 t)) |
                  ((f0 t,f1 t) = (F,T)) → ((sub rep) (inp1 t,inp2 t)) |
                                          ((wordn rep) 0))
```

```
⊢def TamarackBeh (rep:rep_ty) (ireq,mem,pc,acc,rtn,iack) =
       ∀u:time.
         (mem (u+1),pc (u+1),acc (u+1),rtn (u+1),iack (u+1)) =
         NextState rep (ireq u,mem u,pc u,acc u,rtn u,iack u)

⊢def NextState (rep:rep_ty) (ireq,mem,pc,acc,rtn,iack) =
       let opcval = OpcVal rep (mem,pc) in
        ((ireq ∧ ¬iack)      ⇒ IRQ_SEM rep (mem,pc,acc,rtn,iack) |
        (opcval = JZR_OPC) ⇒ JZR_SEM rep (mem,pc,acc,rtn,iack) |
        (opcval = JMP_OPC) ⇒ JMP_SEM rep (mem,pc,acc,rtn,iack) |
        (opcval = ADD_OPC) ⇒ ADD_SEM rep (mem,pc,acc,rtn,iack) |
        (opcval = SUB_OPC) ⇒ SUB_SEM rep (mem,pc,acc,rtn,iack) |
        (opcval = LDA_OPC) ⇒ LDA_SEM rep (mem,pc,acc,rtn,iack) |
        (opcval = STA_OPC) ⇒ STA_SEM rep (mem,pc,acc,rtn,iack) |
        (opcval = RFI_OPC) ⇒ RFI_SEM rep (mem,pc,acc,rtn,iack) |
                             NOP_SEM rep (mem,pc,acc,rtn,iack))

⊢def JZR_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = (fetch rep) (mem,(address rep) pc) in
       let nextpc = ((iszero rep) acc) ⇒ inst | ((inc rep) pc) in
         (mem,nextpc,acc,rtn,iack)

⊢def JMP_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = (fetch rep) (mem,(address rep) pc) in
         (mem,inst,acc,rtn,iack)

⊢def ADD_SEM (rep:rep_ty)
       (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = (fetch rep) (mem,(address rep) pc) in
       let operand = (fetch rep) (mem,(address rep) inst) in
         (mem,(inc rep) pc,(add rep) (acc,operand),rtn,iack)
```

Re-usable Correctness Results, ...

```
|- ∀datain pwr dataout wmem dreq addr.
    Val3_CASES_ASM (rep:rep_ty) ∧
    Val4Word4_ASM rep ∧
     TamarackImp rep (
       datain,pwr,pwr,ireq,mpc,mar,pc,
       acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr) ∧
     SynMemory rep (wmem,addr,dataout,mem,datain) ∧
     PWR pwr ∧
     (((val4 rep) o mpc) 0 = 0)
     ⟹
     let f = TimeOfCycle rep (ireq,mem,pc,acc,rtn,iack) in
     TamarackBeh rep (ireq o f,mem o f,pc o f,acc o f,rtn o f,iack o f)
```

- independent of any fixed word size
- independent of elaborate proof infrastructure
- could be linked into a verified stack

Here is a representation for a 16-bit version,

```
Define ("ISZERO16 w = ((VAL16 w) = 0)");;
Define ("INC16 w = WORD16 ((VAL16 w) + 1)");;
Define ("ADD16 (w1,w2) = WORD16 ((VAL16 w1) + (VAL16 w2))");;
Define ("SUB16 (w1,w2) = WORD16 ((VAL16 w1) - (VAL16 w2))");;
Define ("OPCODE w = WORD3 (V (SEG (0,2) (BITS16 w)))");;
Define ("ADDRESS w = WORD13 (V (SEG (3,15) (BITS16 w)))");;

Define (
  "REP16 =
    ISZERO16,                              % iszero %
    INC16,                                 % inc %
    ADD16,                                 % add %
    SUB16,                                 % sub %
    WORD16,                                % wordn %
    VAL16,                                 % valn %
    OPCODE,                                % opcode %
    VAL3,                                  % val3 %
    ADDRESS,                               % address %
    (λ(x,y). FETCH13 x y),                 % fetch %
    (λ(x,y,z). STORE13 y z x),             % store %
    WORD4,                                 % word4 %
    VAL4");;                               % val4 %
```

. . . which is just "plugged into" the generic correctness theorem:

```
|- ∀datain pwr dataout wmem dreq addr.
     Val3_CASES_ASM REP_16 ∧
     Val4Word4_ASM REP_16 ∧
     TamarackImp REP_16 (
       datain,pwr,pwr,ireq,mpc,mar,pc,
       acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr) ∧
     SynMemory REP_16 (wmem,addr,dataout,mem,datain) ∧
     PWR pwr ∧
     (((val4 rep) o mpc) 0 = 0)
     ⟹
     let f = TimeOfCycle rep (ireq,mem,pc,acc,rtn,iack) in
     TamarackBeh REP_16 (ireq o f,mem o f,pc o f,acc o f,rtn o f,iack o f)
```

Fully generic description of hardware in VHDL,

```
package TYPES is

   subtype word_3 is bit_vector (2 downto 0);
   subtype word_4 is bit_vector (3 downto 0);
   subtype word_n is bit_vector (15 downto 0);
   subtype num is natural;
   subtype bool is boolean;

end TYPES;
```

```
library tamarack;
use tamarack.types.all;

package TAM3 is

   function iszero (inp:word_n) return bool;
   function inc (inp:word_n) return word_n;
   function add (inp1,inp2:word_n) return word_n;
   function sub (inp1,inp2:word_n) return word_n;
   function wordn (inp:num) return word_n;
   function valn (inp:word_n) return num;
   function opcode (inp:word_n) return word_3;
   function val3 (inp:word_3) return num;
   function word4 (inp:num) return word_4;
   function val4 (inp:word_4) return num;

end TAM3;
```

```
function iszero (inp:word_n) return bool is
   variable result :bool := TRUE;
begin
   for i in inp'low to inp'high loop
       if inp(i) = '1' then
          result := FALSE;
       end if;
   end loop;
   return(result);
end iszero;
```

A typical formalism involves a set of operators,

e.g., the temporal logic operators

- $\Box$ - "henceforth"
- $\Diamond$ - "eventually"
- $\bigcirc$ - "next"
- $\cup$ - "until"

...and a set of transformation rules:

e.g., $$\frac{\texttt{P and (not Q)} \longrightarrow \bigcirc \texttt{P}}{\texttt{P} \longrightarrow \texttt{(P} \cup \texttt{Q)}}$$

To embed a formalism in HOL:

1. definitions are given for the operators based on a semantic theory for the formalism;

$$\Box\ \texttt{P} = \lambda \texttt{t}.\ \forall \texttt{n}.\ \texttt{P (t+n)}$$

2. standard transformation rules are derived from operator definitions as theorems of higher-order logic

$$\forall \texttt{P Q}.\ \texttt{VALID}((\texttt{P} \wedge (\neg \texttt{Q})) \longrightarrow (\bigcirc \texttt{P})) \Longrightarrow \texttt{VALID}(\texttt{P} \longrightarrow (\texttt{P} \cup \texttt{Q}))$$

Contrast with 'Syntax-Based' Approaches

This approach differs fundamentally from 'syntax-based' approaches of developing support tools for formalisms:

> for instance, tools such as the Cornell Synthesizer Generator (CSG) have been used to develop support tools for the protocol specification language LOTOS.

In the 'syntax-based' approach, transformation rules are simply 'programmed' (prehaps inconsistently !) into the system.

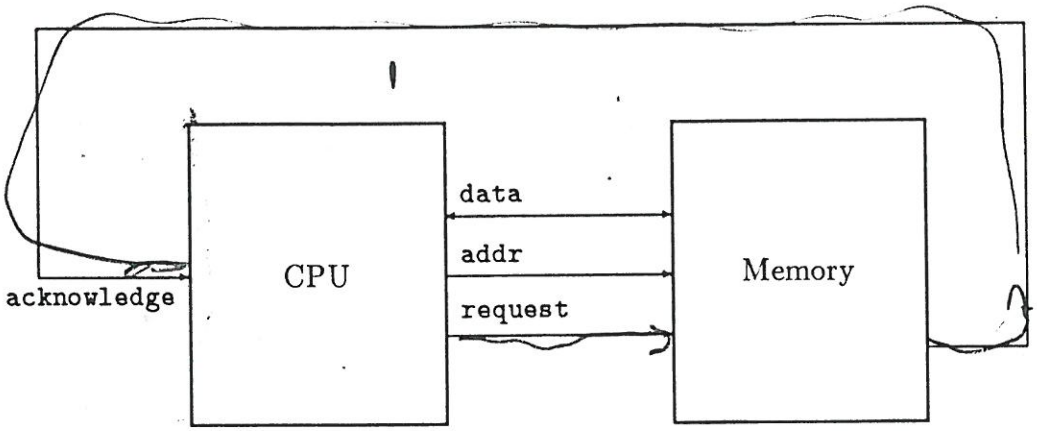
...but in a HOL approach,

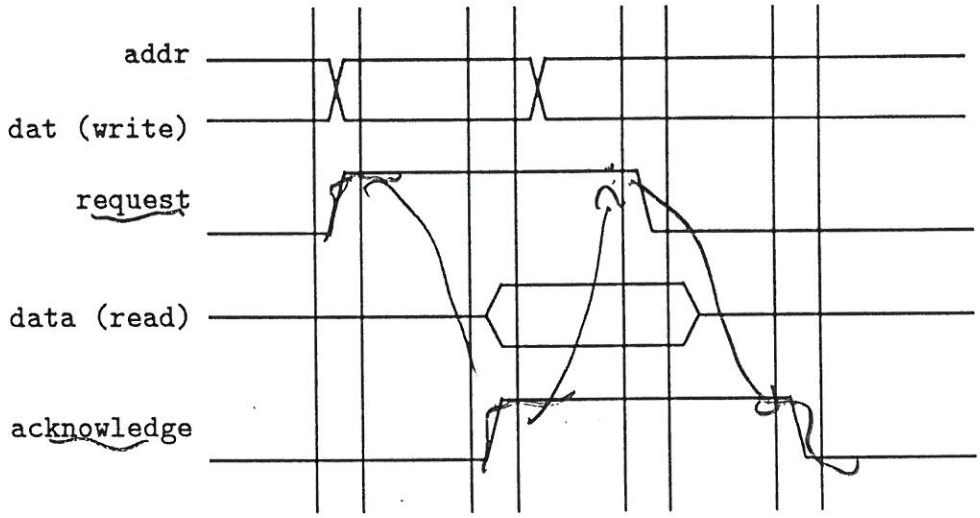
Transformation rules are be formally dervied as logical consequences of the semantic theory (i.e. definitions of primitive operators).

While the HOL system is principally concerned the semantic aspects of representing an embedding formalism, it is often possible to also represent syntactic aspects of the formalism.

An Example

Here's an example of when an embedded notations makes a big difference !





Synchronizing Data Transfer with Handshaking Signals

The timing diagram is the standard way to describe constraints on the relative order of events in a handshaking sequence.

Some of these constraints, along with additional constraints, are expressed below in natual language.

"whenever the request signal becomes true, it must remain true until it is acknowledged"

"every request must eventually be acknowledged"

"whenever the acknowledgement signal becomes true, it must remain true until the request signal returns to false"

"the request signal will eventually return to false after the request is acknowledged"

"whenever the request signal is false, it will remain false until the acknowledgement signal is also false"

"the acknowledgement signal will eventually return to false after the request signal returns to false"

"once false, the acknowledgement signal will remain false until there is a request"

"whenever the acknowledgement signal is false, there will eventually be a request"

For handshaking to work correctly, these constraints must hold.

The natural language descriptions are the previous page could be translated into the following set of eight assertions:

```
∀t. req t ⟹ (∀n. (∀m. m < n ⟹ ack(t + m)) ⟹ req(t + n))

∀t. req t ⟹ (∃n. ack(t + n))

∀t. ack t ⟹ (∀n. (∀m. m < n ⟹ req(t + m)) ⟹ ack(t + n))

∀t. ack t ⟹ (∃n. ~req(t + n))

∀t. req t ⟹ (∀n. (∀m. m < n ⟹ ack(t + m)) ⟹ req(t + n))

∀t. ~req t ⟹ (∃n. ~ack(t + n))

∀t. ack t ⟹ (∀n. (∀m. m < n ⟹ req(t + m)) ⟹ ack(t + n))

∀t. ~ack t ⟹ (∃n. req(t + n))
```

But this is not very easy to read !

For instance, the assertion,

```
∀t. req t ⟹ (∀n. (∀m. m < n ⟹ ack(t + m)) ⟹ req(t + n))
```

...is supposed to say:

> *"whenever the request signal becomes true, it must remain true until it is acknowledged"*

...but it is difficult to translate between this natural language description and the above description in higher-order logic.

A Better Idea !

A much easier notation for expressing constraints of this form is the notation of temporal logic:

$$(\texttt{req} \longrightarrow (\texttt{req} \cup \texttt{ack}))$$

"whenever the request signal becomes true, it must remain true until it is acknowledged"

... where '$\cup$' can informally be understood to mean "until".

Similarly, instead of,

$$\forall \texttt{t}.\ \texttt{req t} \Longrightarrow (\exists \texttt{n}.\ \texttt{ack(t + n)})$$

... we can use the notation of temporal logic to express this same condition,

$$(\texttt{req} \longrightarrow (\Diamond \texttt{ack}))$$

"every request must eventually be acknowledged"

... where '$\Diamond$' means "eventually".

(Note: there are no explicit time variables t in the temporal logic specifications.)

Here is the complete translations into temporal logic notation:

$$(\texttt{req} \longrightarrow (\texttt{req} \cup \texttt{ack}))$$

"whenever the request signal becomes true, it must remain true until it is acknowledged"

$$(\texttt{req} \longrightarrow (\Diamond\texttt{ack}))$$

"every request must eventually be acknowledged"

$$(\texttt{ack} \longrightarrow (\texttt{ack} \cup (\neg\texttt{req})))$$

"whenever the acknowledgement signal becomes true, it must remain true until the request signal returns to false"

$$(\texttt{ack} \longrightarrow (\Diamond(\neg\texttt{req})))$$

"the request signal will eventually return to false after the request is acknowledged"

$$((\neg\texttt{req}) \longrightarrow ((\neg\texttt{req}) \cup (\neg\texttt{ack})))$$

"whenever the request signal is false, it will remain false until the acknowledgement signal is also false"

$$((\neg \texttt{req}) \longrightarrow (\Diamond(\neg \texttt{ack})))$$

"the acknowledgement signal will eventually return to false after the request signal returns to false"

$$((\neg \texttt{ack}) \longrightarrow ((\neg \texttt{ack}) \cup \texttt{req}))$$

"once false, the acknowledgement signal will remain false until there is a request"

$$((\neg \texttt{ack}) \longrightarrow (\Diamond \texttt{req}))$$

"whenever the acknowledgement signal is false, there will eventually be a request"

To take advantage of this concise notation, the notation of temporal can be embedded in higher-order logic:

We first define the primitive operators,

$\vdash_{def} \Box\ P = \lambda t.\ \forall n.\ P\ (t+n)$

$\vdash_{def} \Diamond\ P = \lambda t.\ \exists n.\ P\ (t+n)$

$\vdash_{def} \bigcirc\ P = \lambda t.\ ((P\ (t+1)):bool)$

$\vdash_{def} P \cup Q = \lambda t.\ \forall n.\ (\forall m.\ m < n \Longrightarrow \neg(Q\ (t+m))) \Longrightarrow P\ (t+n)$

$\vdash_{def} \neg P = \lambda t.\ \neg(P\ t)$

$\vdash_{def} P \longrightarrow Q = \lambda t.\ P\ t \Longrightarrow Q\ t$

$\vdash_{def} P \wedge Q = \lambda t.\ P\ t \wedge Q\ t$

$\vdash_{def} VALID\ P = \forall t.\ P\ t$

...and derive rules of temporal logic as theorems of higher-order logic:

$$\forall P\ Q.\ VALID((P \wedge (\neg Q)) \longrightarrow (\bigcirc\ P)) \Longrightarrow VALID(P \longrightarrow (P \cup Q))$$

which corresponds to:

$$\frac{P \text{ and } (\text{not } Q) \longrightarrow \bigcirc P}{P \longrightarrow (P \cup Q)}$$

Exercises:

1. Create a theory for the above temporal logic operators (based on the definitions given). You will have to use some machine-readable symbols instead of □, ◇, ∪, etc.
2. Derive some rules of temporal logic, namely:

$$\forall P\ Q\ S.\ \mathtt{VALID}\ (P \longrightarrow Q) \wedge \mathtt{VALID}\ (Q \longrightarrow S) \Longrightarrow \mathtt{VALID}\ (P \longrightarrow S)$$

```
expandf (PURE_REWRITE_TAC [ ... ] THEN
        BETA_TAC THEN
        REPEAT STRIP_TAC THEN
        RES_TAC THEN
        RES_TAC);;
```

$$\forall P\ Q\ S.$$
$$\mathtt{VALID}\ (P \longrightarrow (\Diamond\ Q)) \wedge \mathtt{VALID}\ (Q \longrightarrow S) \Longrightarrow$$
$$\mathtt{VALID}\ (P \longrightarrow (\Diamond\ S))$$

```
expandf (PURE_REWRITE_TAC [ ... ] THEN
        BETA_TAC THEN
        REPEAT STRIP_TAC THEN
        RES_THEN (X_CHOOSE_TAC "n:num") THEN
        EXISTS_TAC "n:num" THEN
        RES_TAC);;
```

A low-level circuit model in HOL

by

Brian Andersen
and
Carsten Rickers

HOL Meeting 90

Outline of our talk

- Introduction

-Presentation of the model

-Implementation

-Performing proofs

-Future developments

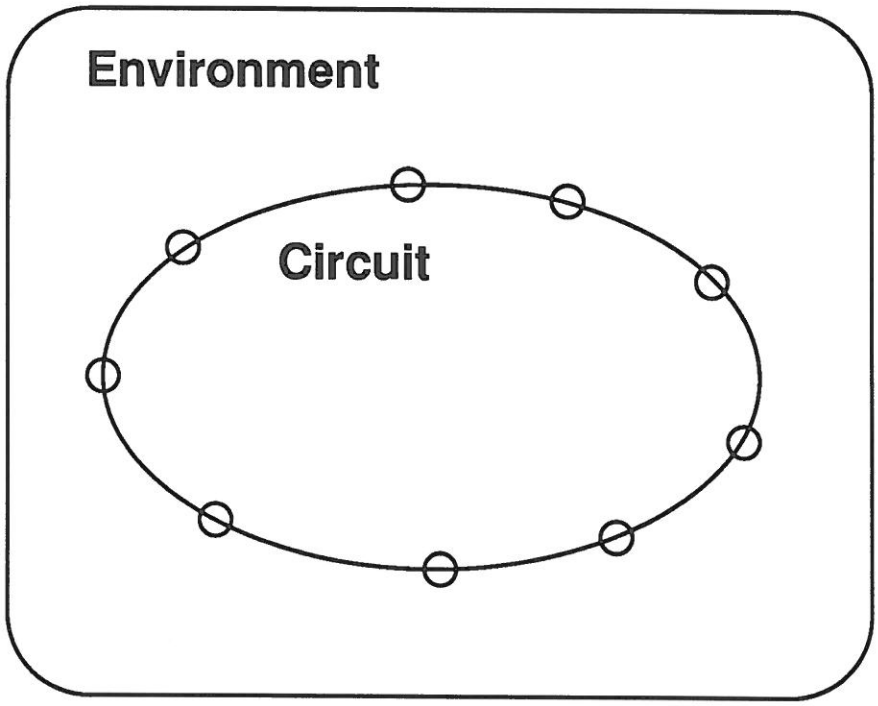
Introduction

We need a model that:

- -introduce new voltage values
- -captures some of the informal arguments used by designers
- -introduce resistances and capacitances
- -deduce the flow of signals rather than imposing a flow
- -is compositional

A compositional model

What is a circuit?



sort:
the set of connection points

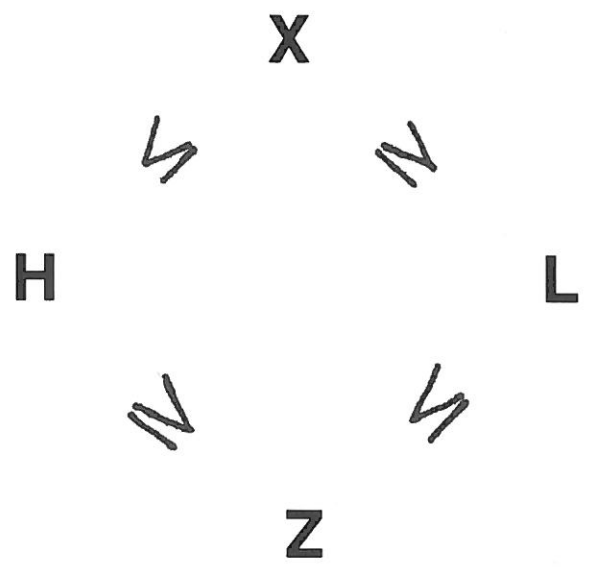
Composition:
$c1 \circ c2$

Hiding:
$c \setminus \alpha$

HOL Meeting 90

A compositional model

The voltage values:



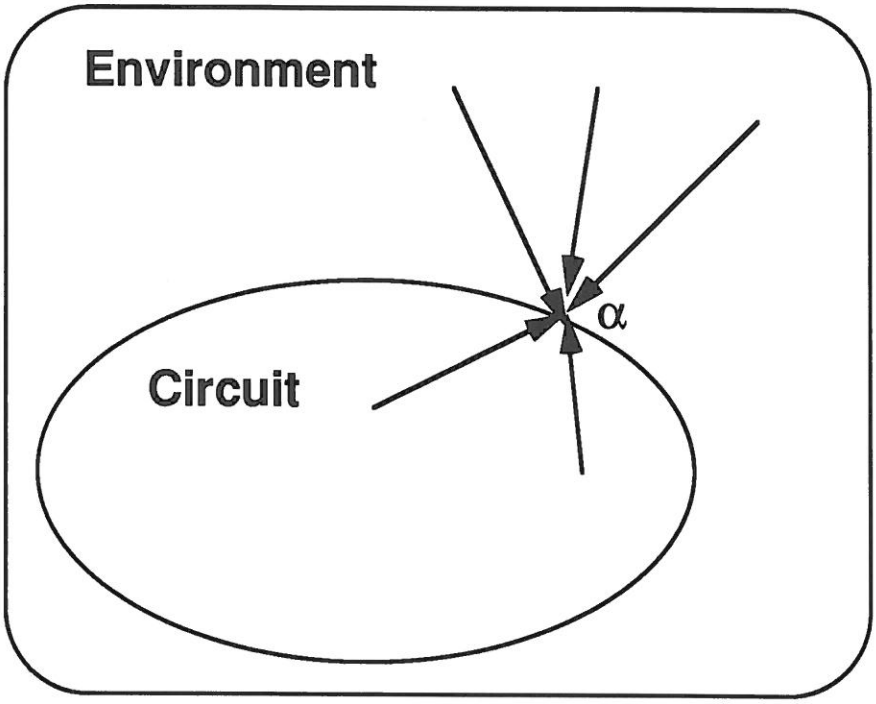
The strength order:

$$0 < k1 < \ldots\ldots < km < g1 < \ldots\ldots < gn < \infty$$

HOL Meeting 90

A compositional model

Attributes describing a circuit.



V: Λ -> V (the value function)

I: Λ -> V (the internal value function)

S: Λ -> S (the strength function)

→: Λ -> Λ -> bool (the flow relation)

HOL Meeting 90

A compositional model

A static configuration.

Definition:

A static configuration of sort Λ is a structure

$$(S, V, I, \rightarrow)$$

which satisfy

$$S(\alpha) = 0 \Leftrightarrow V(\alpha) = Z$$
$$I(\alpha) \leq V(\alpha)$$
$$\alpha \rightarrow \beta \Rightarrow S(\alpha) \geq S(\beta)$$
$$\alpha \rightarrow \beta \Rightarrow I(\alpha) \leq i(\beta) \wedge V(\alpha) \leq V(\beta)$$
$$\alpha \rightarrow \beta \wedge S(\beta) \in K \cup \{0\} \rightarrow S(\alpha) = S(\beta)$$
$$\alpha \rightarrow \beta \wedge S(\alpha) = S(\beta) \Rightarrow \beta \rightarrow \alpha$$

A compositional model

Composition of two circuits.

$\sigma 0 = (S0, V0, I0, \rightarrow 0)$ and $\sigma 1 = (S1, V1, I1, \rightarrow 1)$
are static configurations of sort $\Lambda 0$ and $\Lambda 1$ respectively.

$$\sigma 0 \circ \sigma 1 = (S, V, I, \rightarrow)$$

if $S0 \cap \Lambda 1 = S1 \cap \Lambda 0$ and $V0 \cap \Lambda 1 = V1 \cap \Lambda 0$
and undefined otherwise

$S = S0 \cup S1$ and $V = V0 \cup V1$ and $\rightarrow = (\rightarrow 0 \cup \rightarrow 1)^*$ and
$I(\alpha) = \Sigma\{I0(\beta) \mid \beta \in \Lambda 0 \text{ and } \beta \rightarrow \alpha\} +$
$\Sigma\{I1(\beta) \mid \beta \in \Lambda 1 \text{ and } \beta \rightarrow \alpha\}$

for any $\alpha \in \Lambda 0 \cup \Lambda 1$.

Hiding a point in a circuit.

$\sigma = (S, V, I, \rightarrow)$ is a static configuration of sort Λ.

$$\sigma \setminus \alpha = (S \setminus \alpha, V \setminus \alpha, I \setminus \alpha, \rightarrow \setminus \alpha)$$

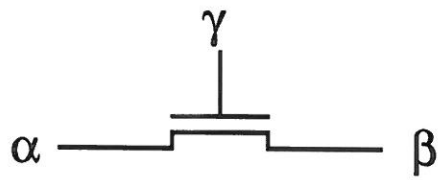
if $\alpha \notin \Lambda$ or $V(\alpha) = I(\alpha) + \Sigma\{V(\beta) \mid \beta \in \Lambda \setminus \alpha \text{ and } \beta \rightarrow \alpha\}$,
and to be undefined otherwise.

A compositional model

A language to describe circuits.

Syntax:

$$
\begin{aligned}
c ::= \; & Pow(\alpha) \mid Gnd(\alpha) \mid \\
& capkH(\alpha) \mid capkL(\alpha) \mid \\
& pt(\alpha) \mid wre(\alpha) \mid resg(\alpha,\beta) \mid \\
& ntran(\alpha,\beta,\gamma) \mid ptran(\alpha,\beta,\gamma) \mid \\
& c \circ c \mid c \setminus \alpha
\end{aligned}
$$



Semantics:

$$
\begin{aligned}
[[ntran(\alpha,\beta,\gamma)]] = \{ \sigma \in Sta(\alpha,\beta,\gamma) \mid \; & I(\alpha) = Z \wedge I(\beta) = Z \wedge \\
& I(\gamma) = Z \wedge \gamma \,||\, \alpha \wedge \gamma \,||\, \beta \wedge (\alpha \,||\, \beta \vee \alpha \leftrightarrow \beta) \wedge \\
& (V(\gamma) = H \Leftrightarrow \alpha \leftrightarrow \beta) \wedge (V(\gamma) = L \Leftrightarrow \alpha \,||\, \beta)\}
\end{aligned}
$$

$$
[[c \circ d]] = \{ \sigma \circ \rho \mid \sigma \in [[c]] \text{ and } \rho \in [[d]] \text{ and } \sigma \circ \rho \downarrow \}
$$

HOL Meeting 90

A compositional model

An assertion language.

Syntax:

Value terms: $t ::= H \mid L \mid Z \mid X \mid V(\pi) \mid I(\pi)$

Strength terms: $e ::= s \mid S(\pi) \mid e \bullet e \mid e + e$

Assertions for static configurations:

$$\begin{aligned}\phi ::= {} & \pi 0 = \pi 1 \mid \pi 0 \rightarrow \pi 1 \mid \\ & t0 = t1 \mid t0 \leq t1 \mid \\ & e0 = e1 \mid e0 \leq e1 \mid \\ & T \mid F \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \\ & \exists x.\phi \mid \forall x.\phi\end{aligned}$$

Semantics:

$$[[t0 = t1]] = \{ \sigma \in Sta \mid [[t0]]\sigma\downarrow \text{ and } [[t1]]\sigma\downarrow \text{ and } [[t0]]\sigma = [[t1]]\sigma \}$$

A Proof System in HOL

We will concentrate on goals as: $[c] \subseteq [A]$

compositional model:

$[c1] \subseteq [A]$ $\qquad$ $[c2] \subseteq [B]$

proof

$[c1 \circ c2] \subseteq [C]$

Requirements to the implementation:

- Basic Concepts
- Static Configurations
- Basic Circuits
- Hide, Comp
- Assertion Language
- Tactics

Basic concepts

points → num in HOL

voltage, strength → new types with operations:

```
let mosvalue = define_new_type (´mosvalue´,
                                "mosvalue = Z | L | H | Z");;

let strength = define_new_type (´strength´,
    strength = Zero_str | Cap num | Res num | Inf_str");;

let vlt = new_definition(´vlt´,
    "$vlt x y = ~(x = y) ∧ (x = Z) ∨ (y = X)");;
```

assertion language → build-in operations in HOL +
expressions over the new types

f.x.: (v in = H) ==> (i out = L)

Static configurations

sort is described by a point function (indirect sort)

```
static  (n:num->bool) (s:num->strength)
        (v:num->mosvalue) (i:num->mosvalue)
        (r:num->num->bool)  =
(!a. n a ==> r a a) /\
(!a b c. n a /\ n b /\ n c ==> r a b /\ r b c ==> r a c) /\
(!a. n a ==> (s a = Zero_str) = (v a = Z)) /\
(!a. n a ==>(i a) vle (v a)) /\
(!a b. n a /\ n b ==> r a b ==> (s a) sle (s b)) /\
(!a b. n a /\ n b ==> r a b ==> (i a) vle (i b) /\
                                (v a) vle (v b)) /\
(!a b. n a /\ n b ==> r a b ==>
  (?m. s b = Cap m) \/ (s b = Zero_str) ==> (s a = s b)) /\
(!a b. n a /\ n b ==> r a b /\ (s a = s b) ==> r b a)
```

HOL Meeting 90

Basic circuits

The basic circuits are defined as predicates:

c n s v i r

```
let ntrans = new_definition(´ntrans´,
  "ntrans a b c  =
 \n s v i r.  static n s v i r /\
(!k. n k ==> (k = a) \/ (k = b) \/ (k = c)) /\
~(a = b) /\ ~(b = c) /\ ~(a = c) /\
~(b = a) /\ ~(c = b) /\ ~(c = a) /\
(i a = Z) /\ (i b = Z) /\ (i c = Z) /\
~(r a c) /\ ~(r c a) /\ ~(r b c) /\ ~(r c b) /\
((r ab) /\ (r b a) \/ (~(r a b) /\ ~(r b a))) /\
((v c = H) ==> (r a b) /\ (r b a)) /\
((v c = L) ==> ~(r a b) /\ ~(r b a)) ");;
```

Composition

```
comp c1 c2  =  \n s v i r.
?n1 s1 v1 i1 1r n2 s2 v2 i2 r2.
(c1 n1 s1 v1 i1 r1) ∧ (static n1 s1 v1 i1 r1) ∧
(c2 n2 s2 v2 i2 r2) ∧ (static n2 s2 v2 i2 r2) ∧
(static n1 s1 v1 i1 r1) ∧
(!k. n1 k ∧ n2 k ==> (s1 k = s2 k) ∧ (v1 k = v2 k)) ∧
(!k. n k = n1 k ∨ n2 k) ∧
(!k. n k ==> (s k = (n1 k => s1 k | s2 k)) ∧
(!k. n k ==> (v k = (n1 k => v1 k | v2 k)) ∧
(!a b. n1 a ∧ n1 b ∧ r1 a b = n1 a ∧ n1 b ∧ r a b) ∧
(!a b. n2 a ∧ n2 b ∧ r2 a b = n2 a ∧ n2 b ∧ r a b) ∧
(!a b c. n1 a ∧ ~n2 a ∧ n1 b ∧ n2 b ∧ ~n1 c ∧ n2 c ∧
                r1 a b ∧ r2 b c = n1 a ∧ n2 c ∧ r a c) ∧
(!a b c. n2 a ∧ ~n1 a ∧ n1 b ∧ n2 b ∧ ~n2 c ∧ n1 c ∧
                r2 a b ∧ r1 b c = n2 a ∧ n1 c ∧ r a c) ∧
(!k. n k ==> (i k = vjoinset (\z. ? j. n j ∧ r j k ∧
            ((n1 j ∧ (z = i1 j) ∨ (n2 j ∧ (z = i2 j)))))
```

HOL Meeting 90

Tactics

An example:

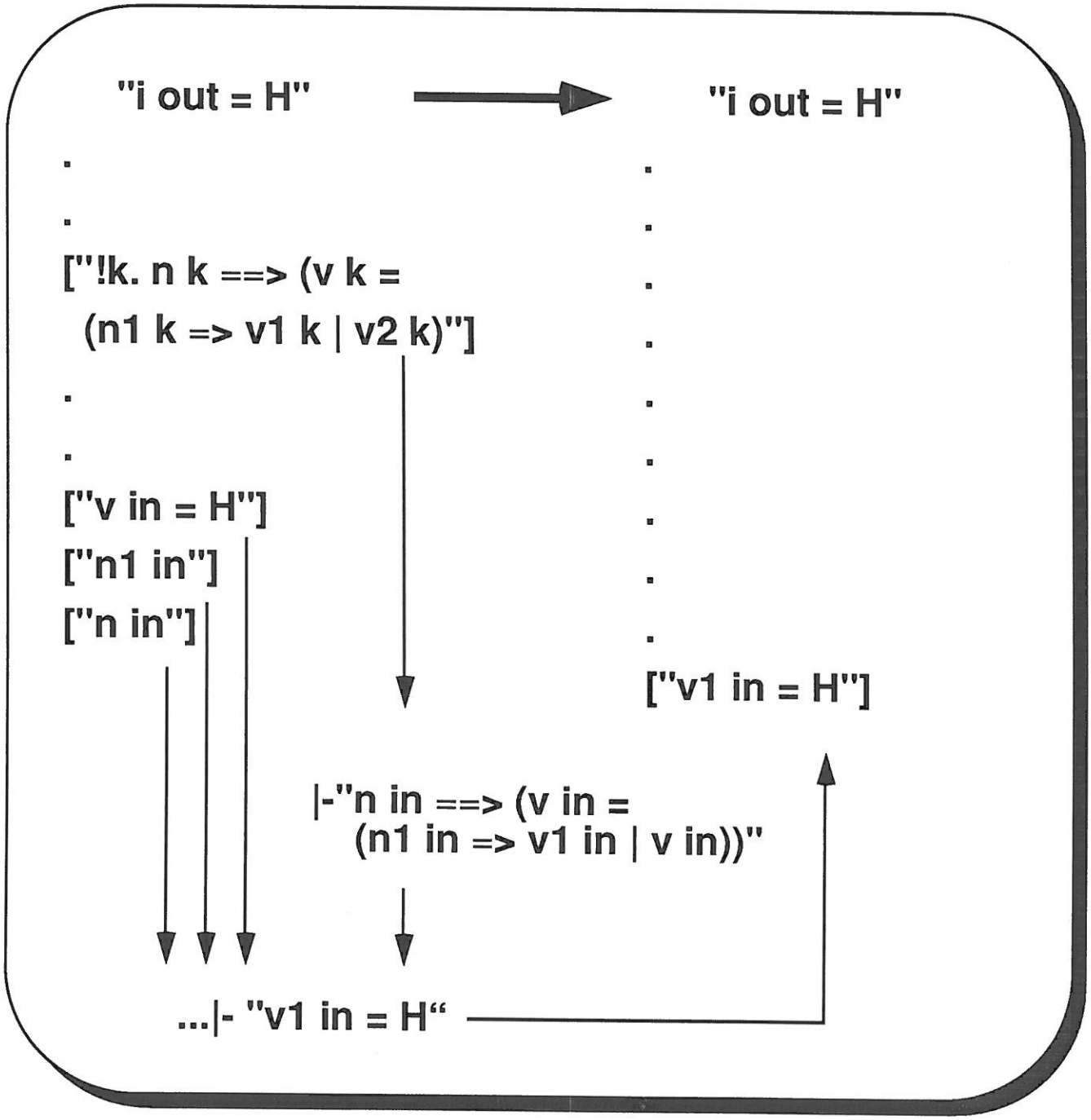
!n s v i r: comp (inv in x) (inv x out) ==>
(v in = H) ==> (i out = H)

can be shown by the properties of inv (+ comp)

proving methodology:

Rewrite with the definition of comp
Tranfer to the assumption list
Add Facts to the assumption list
until enough to show the goal

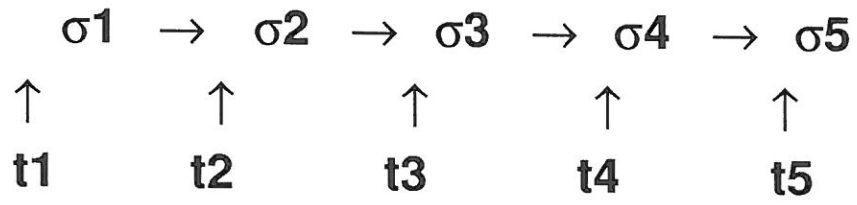
A proving step



HOL Meeting 90

A Dynamic Model

Dynamic model as sequence of steady states



time parameter:

```
! t. (v t in = H) ==> (i t out = L)
```

```
let cap = new_definition (´cap´,
"cap k a = \n s v i r. (static n s v i r) /\'
 (!k. n k = (k = a)) /\
 (!t. (Cap k) sle (s t a)) /\
 (!t. (s (t+1) a = Cap k) ==> (i (t+1) a = v t a)) /\
 (!t. ((Cap k) sle (s t a) ==> (i t a = Z))");;
```

Future developments

We have implemented a hardware model in HOL that works reasonable

Still some work to do

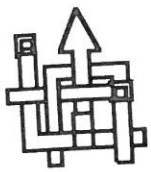
Better transistor description can be obtained

Extensions to other types of proves as: [c1] = [c2]

Can easily be extended to a dynamic model

THE VERIFICATION OF A PARAMETERIZED MEAD&CONWAY ALU CORE IN HOL

CATIA ANGELO
LUC CLAESEN
HUGO DE MAN



imec vzw

MAY 90

OUTLINE

- PRELIMINARY CONSIDERATIONS

- INTRODUCTION

- THE DESIGN PROCESS
  - 1- LOGICAL FUNCTIONS
  - 2- ARITHMETIC FUNCTIONS

- THE PROOF
  - 1- USING THE KNOWLEDGE OF THE DESIGN PROCESS
  - 2- STRATEGY AND PROOF FLEXIBILITY
  - 3- GOING FROM A IRREGULAR STRUCTURE TO A REGULAR STRUCTURE

- CONCLUSIONS

PRELIMINARY CONSIDERATIONS

1- GLOBAL GOALS:

- LEARN HOL;

- DEVELOP A METHODOLOGY FOR HARDWARE VERIFICATION USING HOL;

- IDENTIFY THE BOTTLENECKS OF THE METHODOLOGY AND HOL;

- LIBRARY OF PRE-PROVEN MODULES;

- LIBRARY OF USEFUL STRATEGIES TO PROVE HARDWARE CORRECTNESS (TACTICS AND TACTICALS).

2- LOCAL GOAL: ALU VERIFICATION IN HOL

THE ALU CORE IS A GOOD EXAMPLE BECAUSE THE BEHAVIOR IS SPREAD ALL OVER THE DATA FLOW AND IT IS NOT COMPLETELY STRUCTURED IN SUB-BEHAVIORS.

PRELIMINARY CONSIDERATIONS

1- GLOBAL GOAL:

- DEVELOP A METHODOLOGY FOR HARDWARE VERIFICATION USING HOL AND BOYER MOORE IN SUITABLE LEVELS

- IDENTIFY THE BOTTLENECKS OF:
  - THE METHODOLOGY
  - HOL
  - BOYER MOORE

- HOL:
  - LIBRARY OF PRE-PROVEN MODULES
  - LIBRARY OF USEFUL STRATEGIES TO PROVE HARDWARE CORRECTNESS (TACTICS AND TACTICALS)

2- LOCAL GOAL: ALU VERIFICATION IN HOL

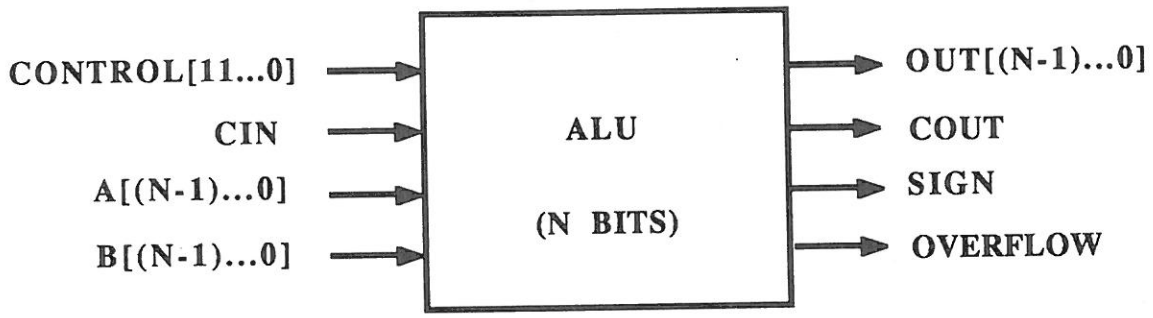
THE ALU CORE IS A GOOD EXAMPLE BECAUSE THE BEHAVIOR IS SPREAD ALL OVER THE DATA FLOW AND IT IS NOT COMPLETELY STRUCTURED IN SUB-BEHAVIORS

INTRODUCTION

- THE GOAL IS TO PROVE THAT:

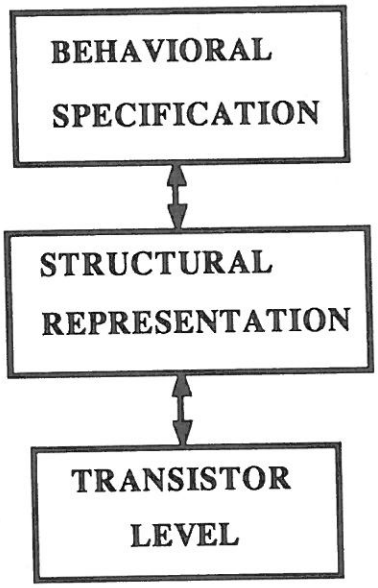
  ALU IMPLEMENTATION = ALU SPECIFICATION

- THE ALU IS PARAMETERIZED ON THE WORD SIZE N

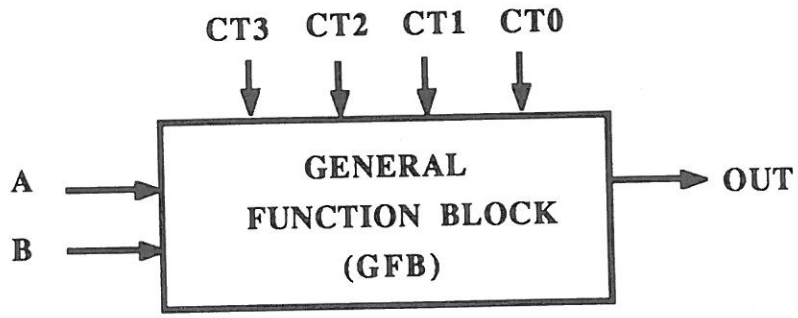


- THE ALU SOULD PERFORM LOGICAL OR ARITHMETIC OPERATIONS ON THE VECTORS "A" AND "B" DEPENDING ON THE CONTROLS. THE SPECIFICATION IS MADE IN A PROGRAMMING LANGUAGE LIKE.

- THE VERIFICATION LEVELS:

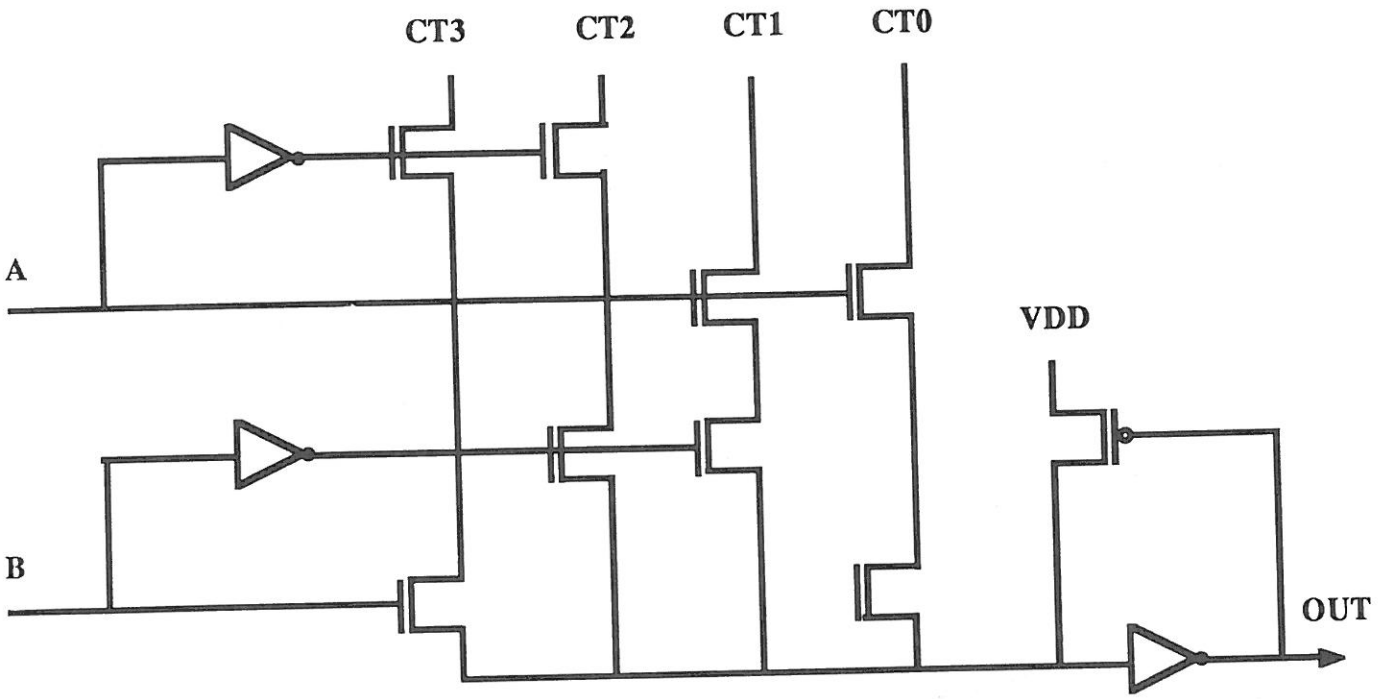


1- LOGICAL FUNCTIONS:

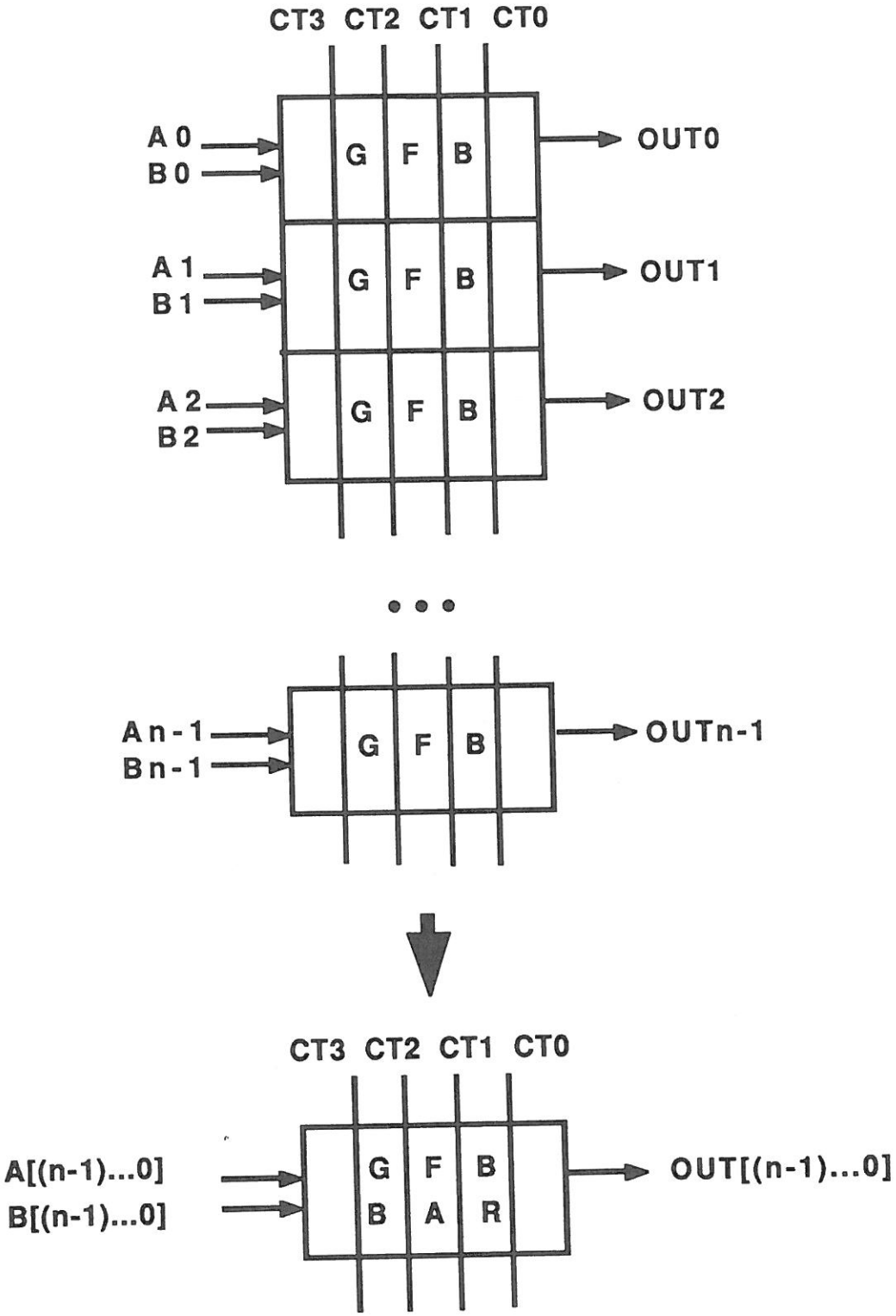


DECIMAL CODE	BINARY CODE CT3 CT2 CT1 CT0	OUT
0	0 0 0 0	1
1	0 0 0 1	(A ∧ B)'
2	0 0 1 0	(A' ∨ B)
3	0 0 1 1	A'
4	0 1 0 0	(A ∨ B)
5	0 1 0 1	(A ⊕ B)
6	0 1 1 0	B
7	0 1 1 1	(A' ∧ B)
8	1 0 0 0	(A ∨ B')
9	1 0 0 1	B'
10	1 0 1 0	(A ⊕ B)'
11	1 0 1 1	(A ∨ B)'
12	1 1 0 0	A
13	1 1 0 1	(A ∧ B')
14	1 1 1 0	(A ∧ B)
15	1 1 1 1	0

GFB CIRCUIT



THE GFB BAR



THE GFB BAR PERFORMS ALL LOGICAL OPERATIONS

WITH TWO VECTORS A AND B.

2- ARITHMETIC FUNCTIONS

THEY ARE ALL A KIND OF ADDITION

C	A	B	K	P	SUM	CARRY
0	0	0	0	0	0	0
0	0	1	0	1	1	0
0	1	0	1	1	1	0
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	1	0	1	0	1
1	1	0	1	1	0	1
1	1	1	1	0	1	1

K = A

P = EXOR (A,B)

CARRY = (P $\wedge$ C) $\vee$ (P' $\wedge$ K)

SUM = EXOR(P,C)

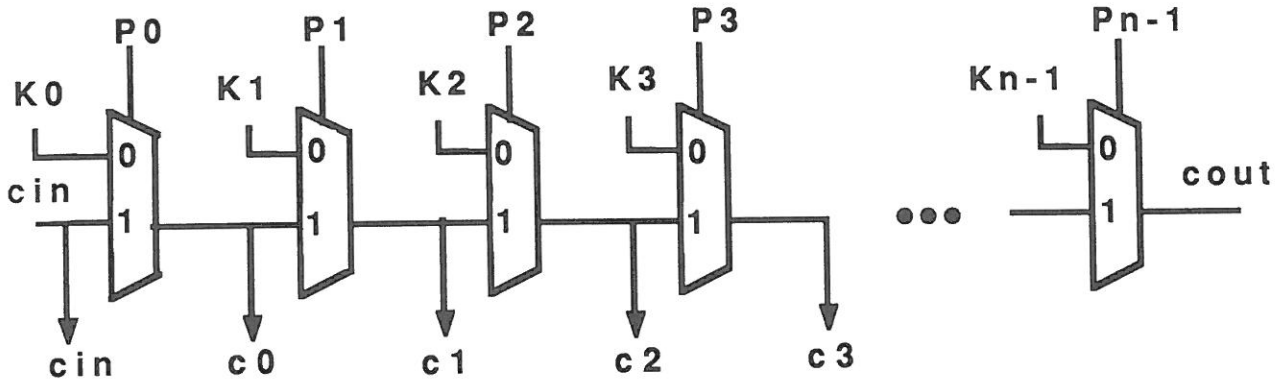
FOR N BITS:

Kj = Aj	==> GFB (Aj, Bj)
Pj = EXOR (Aj, Bj)	==> GFB (Aj, Bj)
Cj = (Pj $\wedge$ Cj-1) $\vee$ (Pj' $\wedge$ Kj)	==> CARRY (Pj, Kj, Cj-1)
Sj = EXOR (Pj, Cj-1)	==> GFB (Pj, Cj-1)

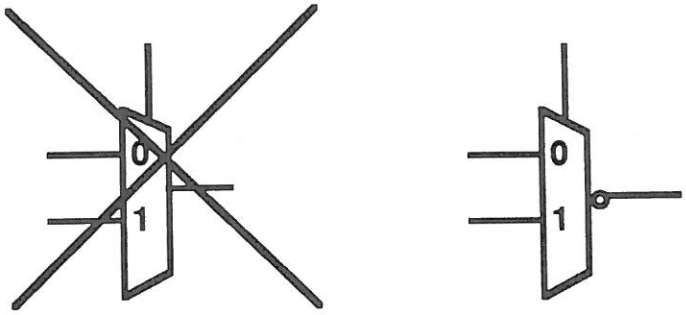
Kv, Pv, Sv ==> 3 GFB BARS

Cv ==> MUX CHAIN

IDEAL CARRY CHAIN



BUT...



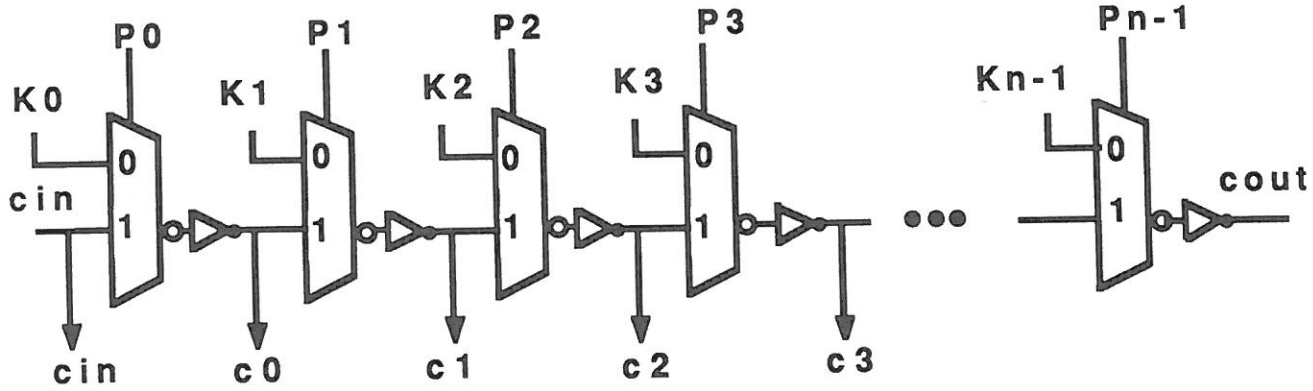
"CORRECT" INPUTS

INVERTED OUTPUTS

HOW TO CASCADE?

POSSIBLE SOLUTIONS:

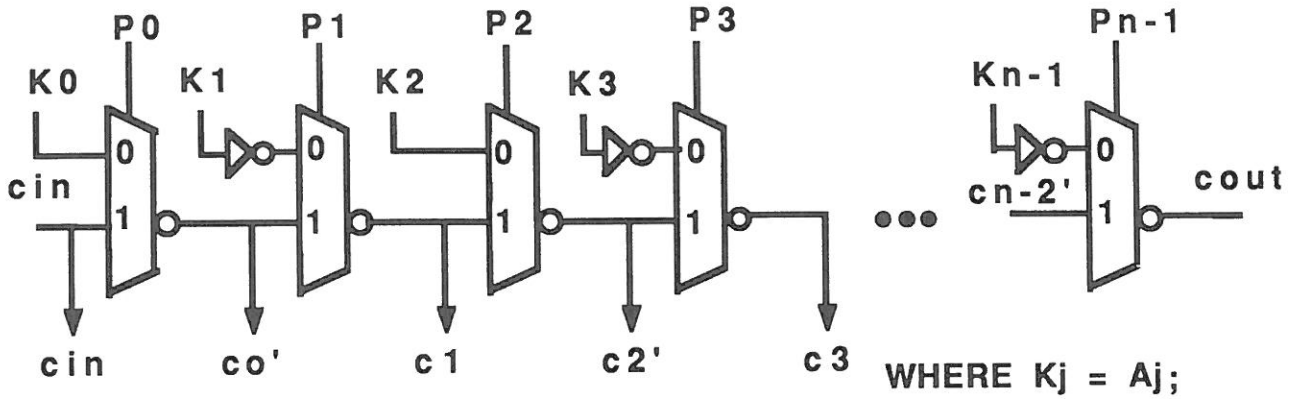
I)



BAD SOLUTION !

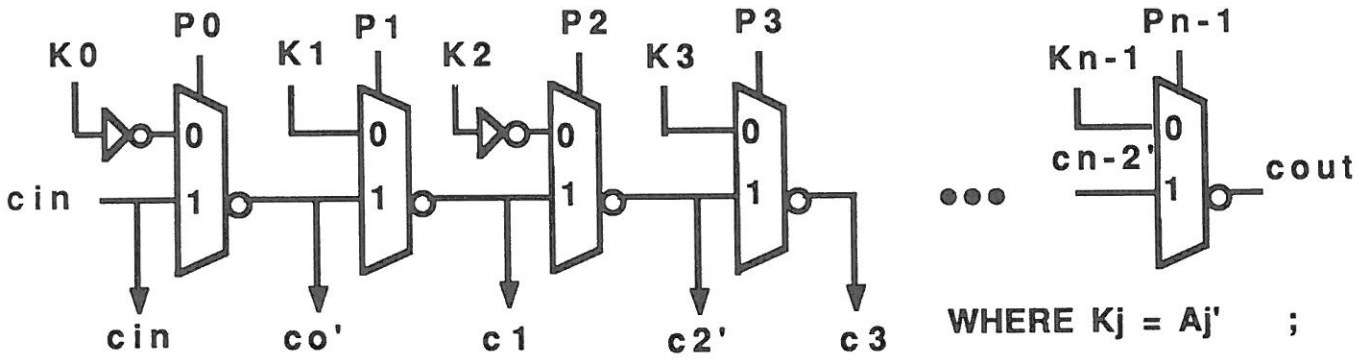
WASTE OF TIME AND AREA !

II)



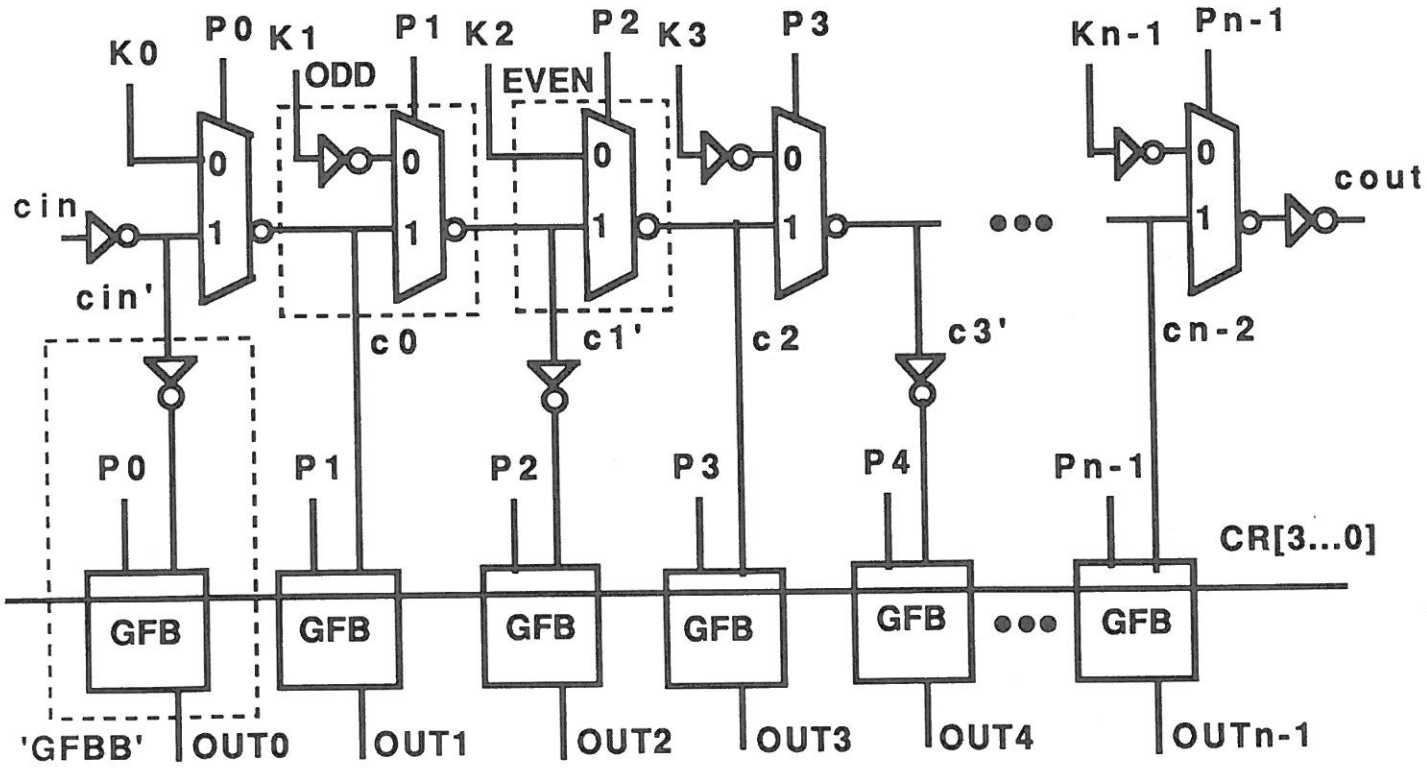
WHERE Kj = Aj;

IF INSTEAD OF Kj = Aj ONE GENERATES Kj = Aj'
IN THE K GFB BAR:



WHERE Kj = Aj' ;

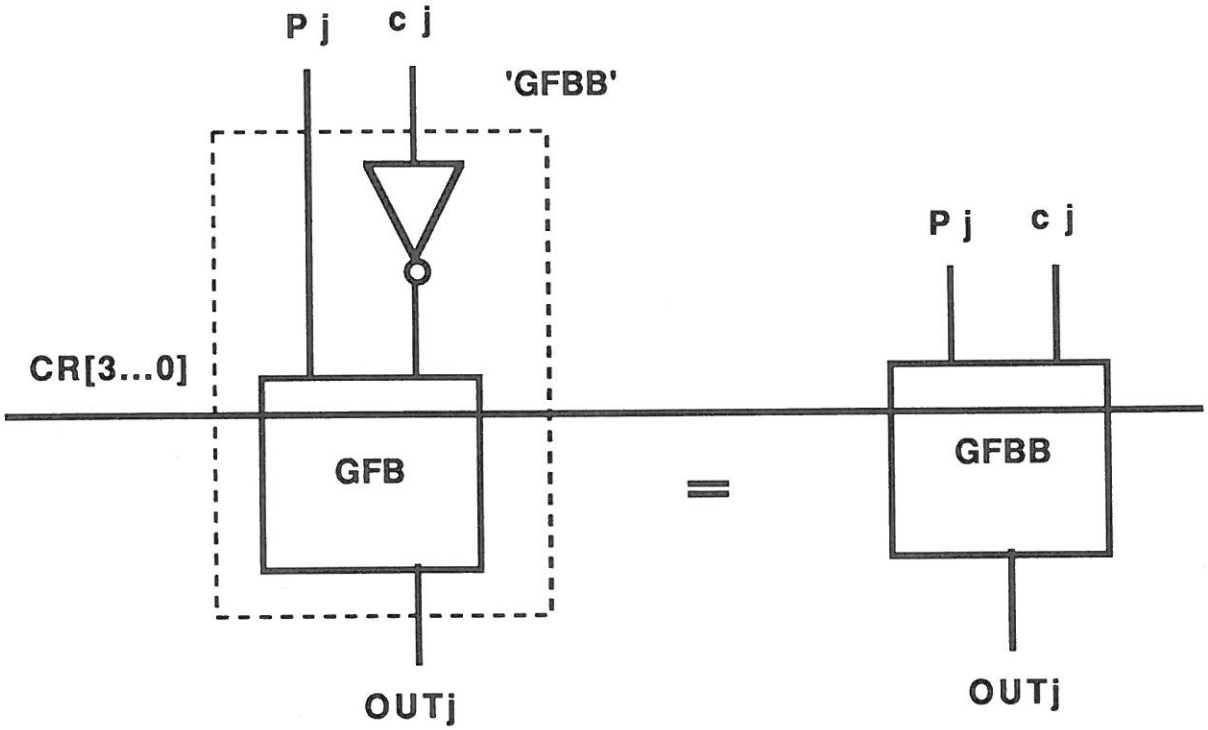
SINCE BUFFERING CIN AND COUT IS DESIRED THE CHAIN IS SHIFTED

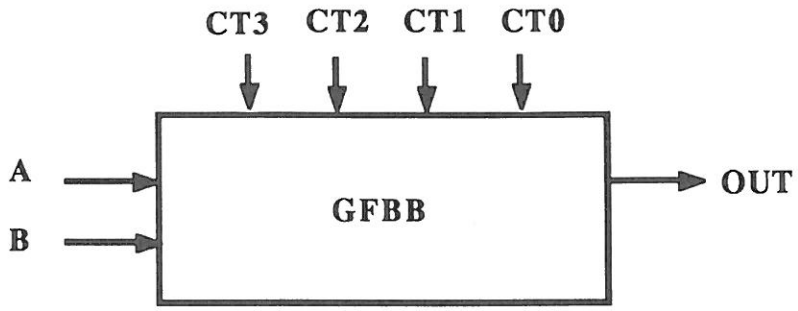


WHERE Kj = Aj'

'GFBB' BEHAVES LIKE GFBB BUT IT HAS A

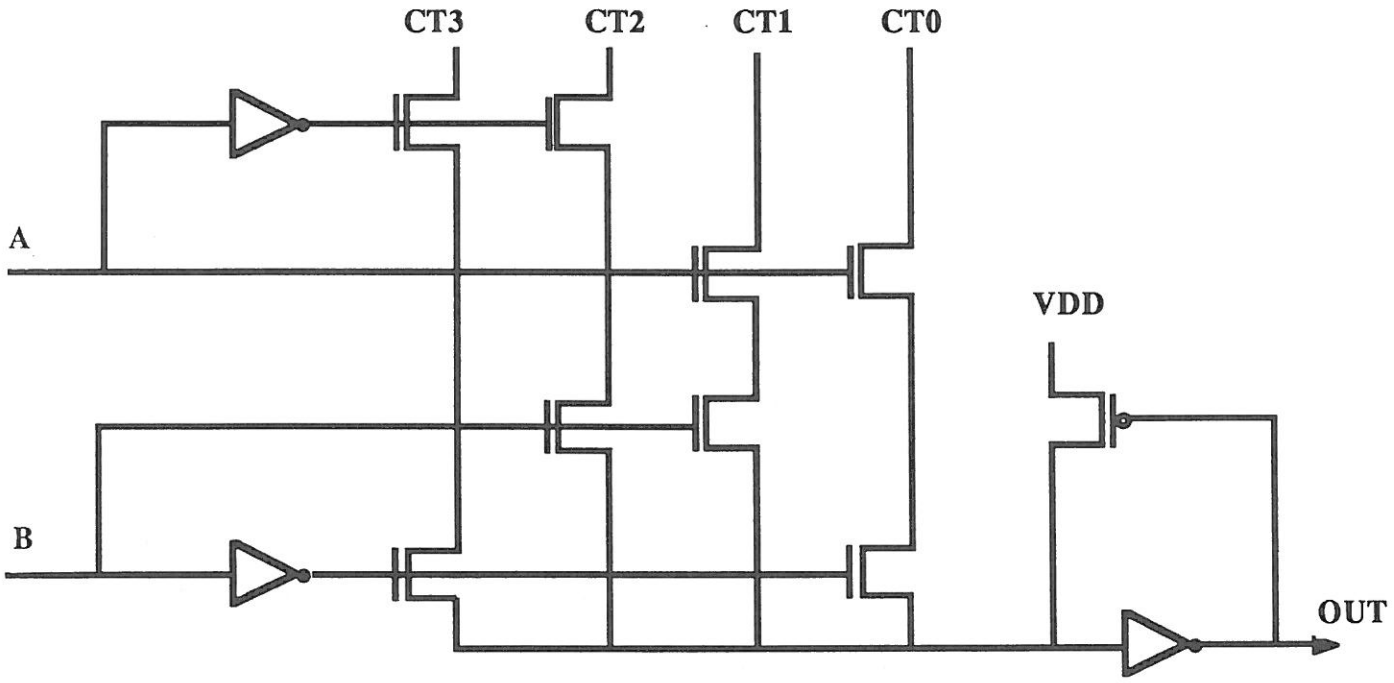
DIFFERENT HARDWARE



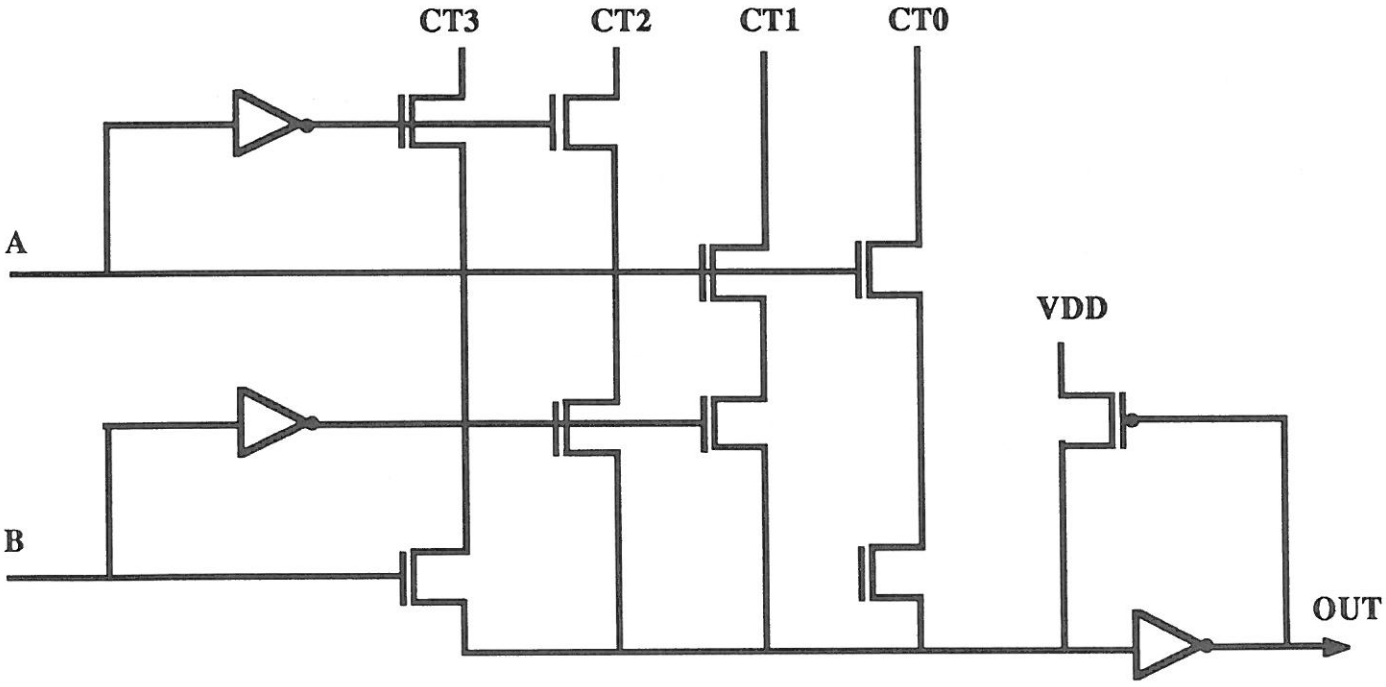


DECIMAL CODE	BINARY CODE CT3 CT2 CT1 CT0	OUT
0	0 0 0 0	1
1	0 0 0 1	(A' ∨ B)
2	0 0 1 0	(A ∧ B)'
3	0 0 1 1	A'
4	0 1 0 0	(A ∨ B')
5	0 1 0 1	(A ⊕ B)'
6	0 1 1 0	B'
7	0 1 1 1	(A ∨ B)'
8	1 0 0 0	(A ∨ B)
9	1 0 0 1	B
10	1 0 1 0	(A ⊕ B)
11	1 0 1 1	(A' ∧ B)
12	1 1 0 0	A
13	1 1 0 1	(A ∧ B)
14	1 1 1 0	(A ∧ B')
15	1 1 1 1	0

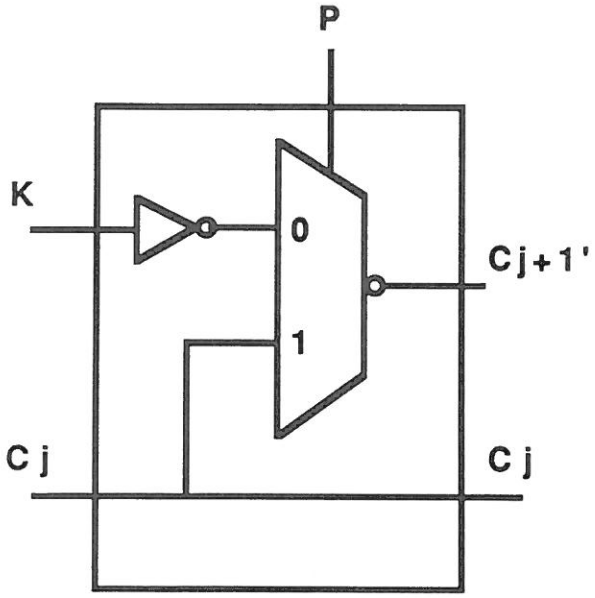
GFBB CIRCUIT



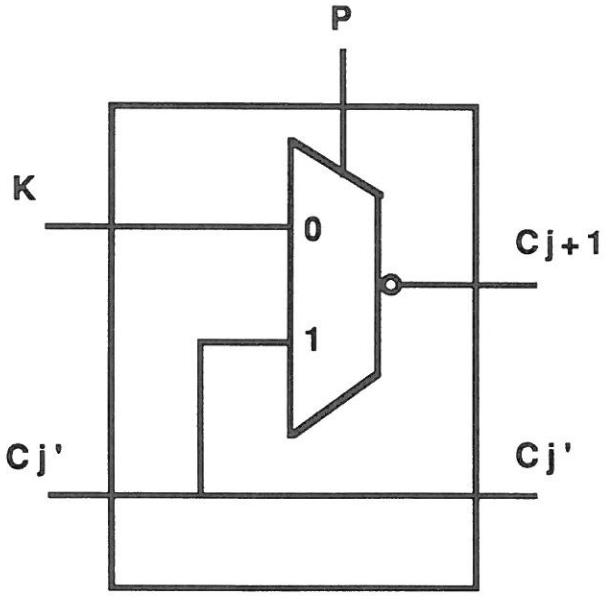
GFB CIRCUIT



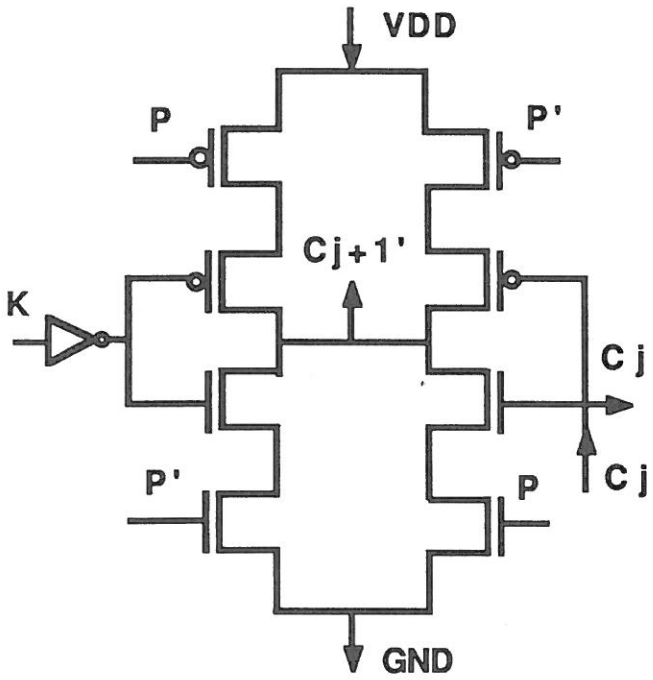
THE CARRY CHAIN CELLS

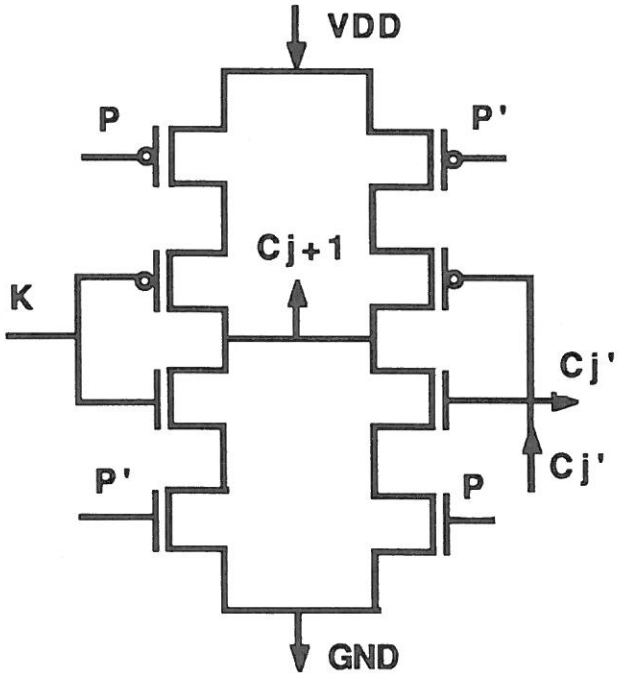


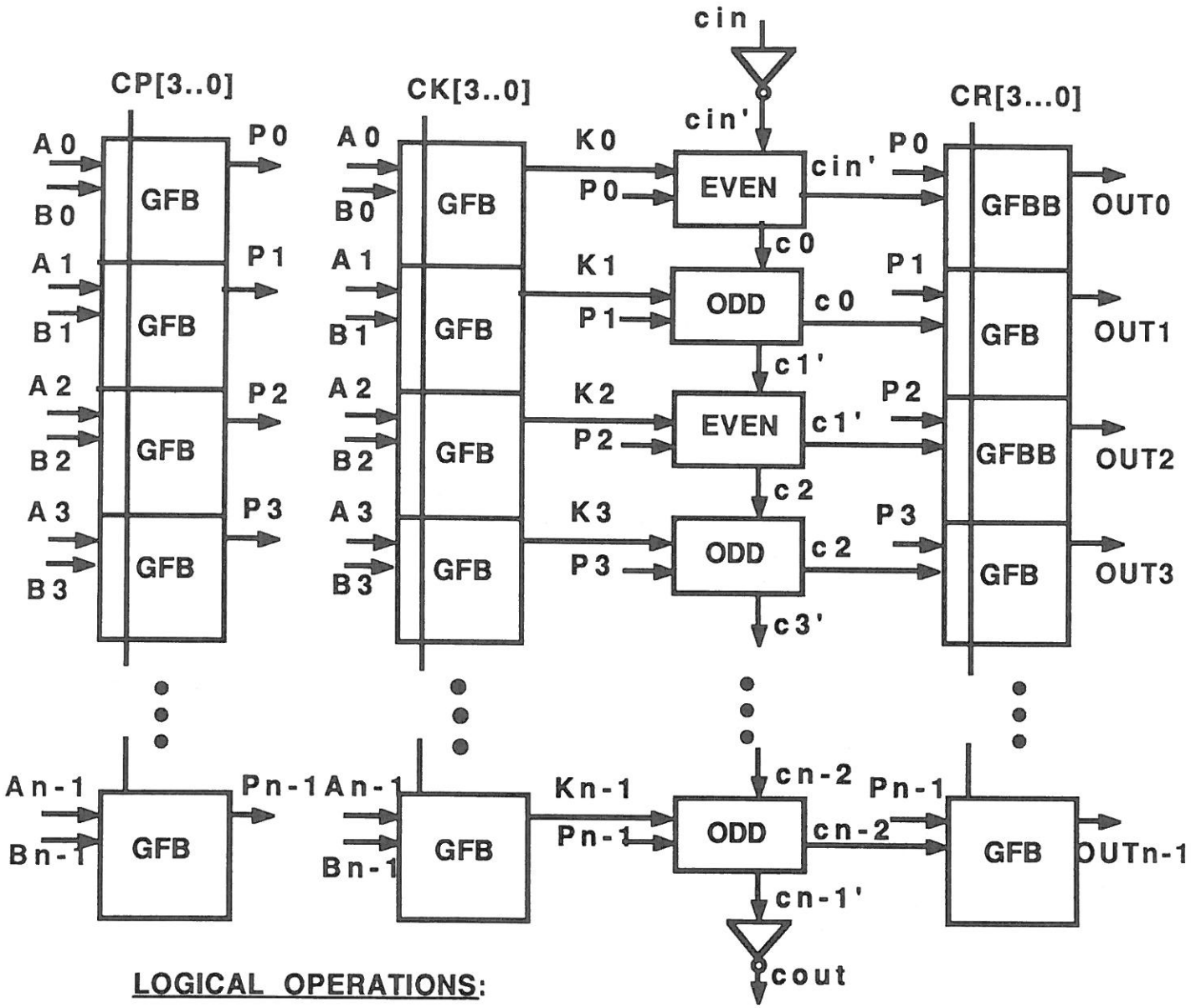
ODD CELL



EVEN CELL







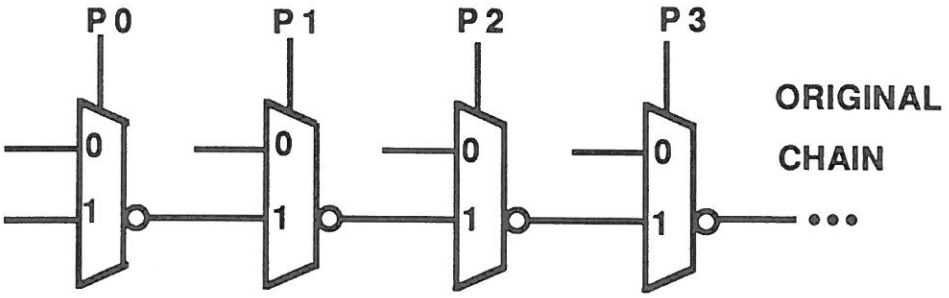
LOGICAL OPERATIONS:

- THE OPERATION IS PERFORMED IN THE P BAR
- CR IS SUCH THAT OUTj = Pj
- WHAT HAPPENS IN THE K BAR AND IN THE CARRY CHAIN DOES NOT MATTER
- THE DATA FLOW IS HORIZONTAL

ARITHMETIC OPERATIONS:

- THEY CAN BE SEEN AS A KIND OF ADDITION
- THE DATA FLOW IS BOTH HORIZONTAL AND VERTICAL
- HORIZONTAL FLOW: PARALLEL
- VERTICAL FLOW: SERIAL (SPEED BOTTLENECK)

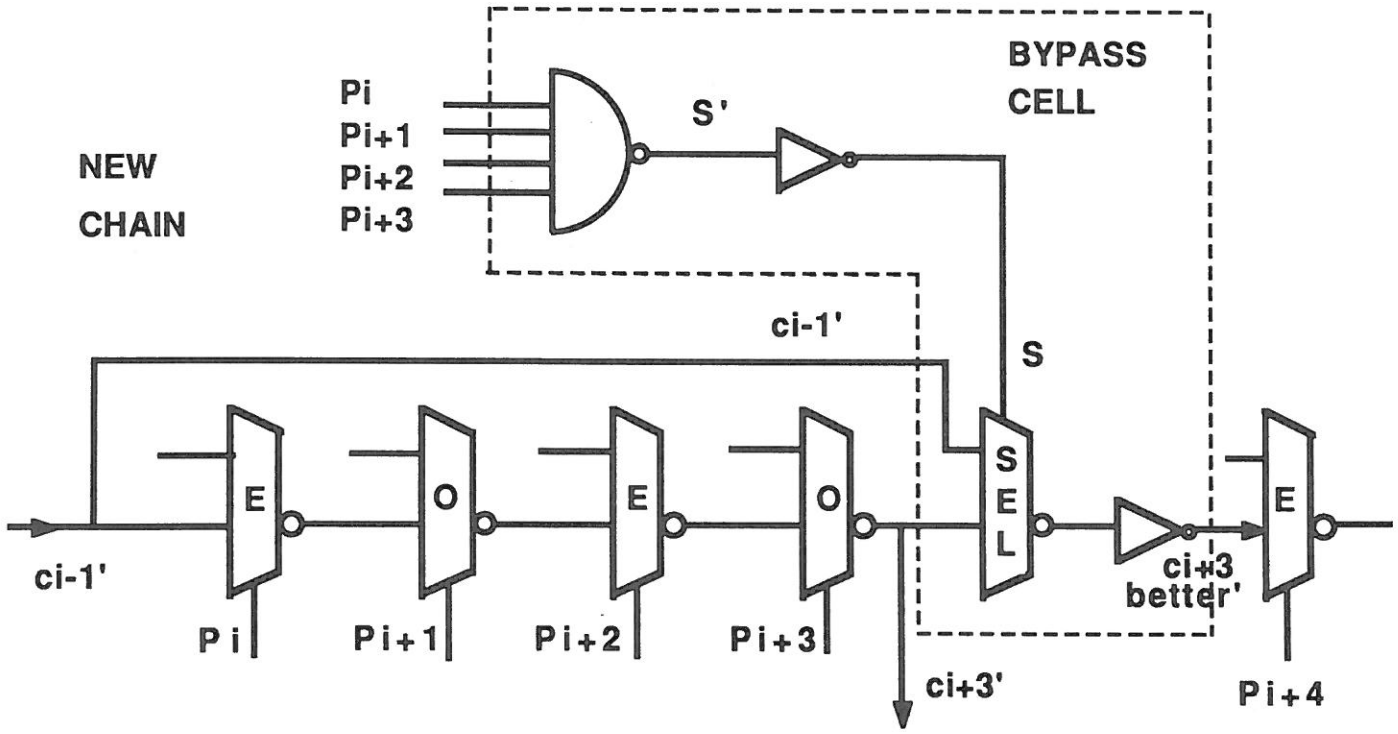
IMPROVING THE CARRY CHAIN SPEED



P = 1 MEANS: CARRY INFORMATION IS PROPAGATED

(Pi = 1) $\wedge$ (Pi+1 = 1) $\wedge$ (Pi+2 = 1) $\wedge$ (Pi+3 = 1)

==> ci+3 = ci-1



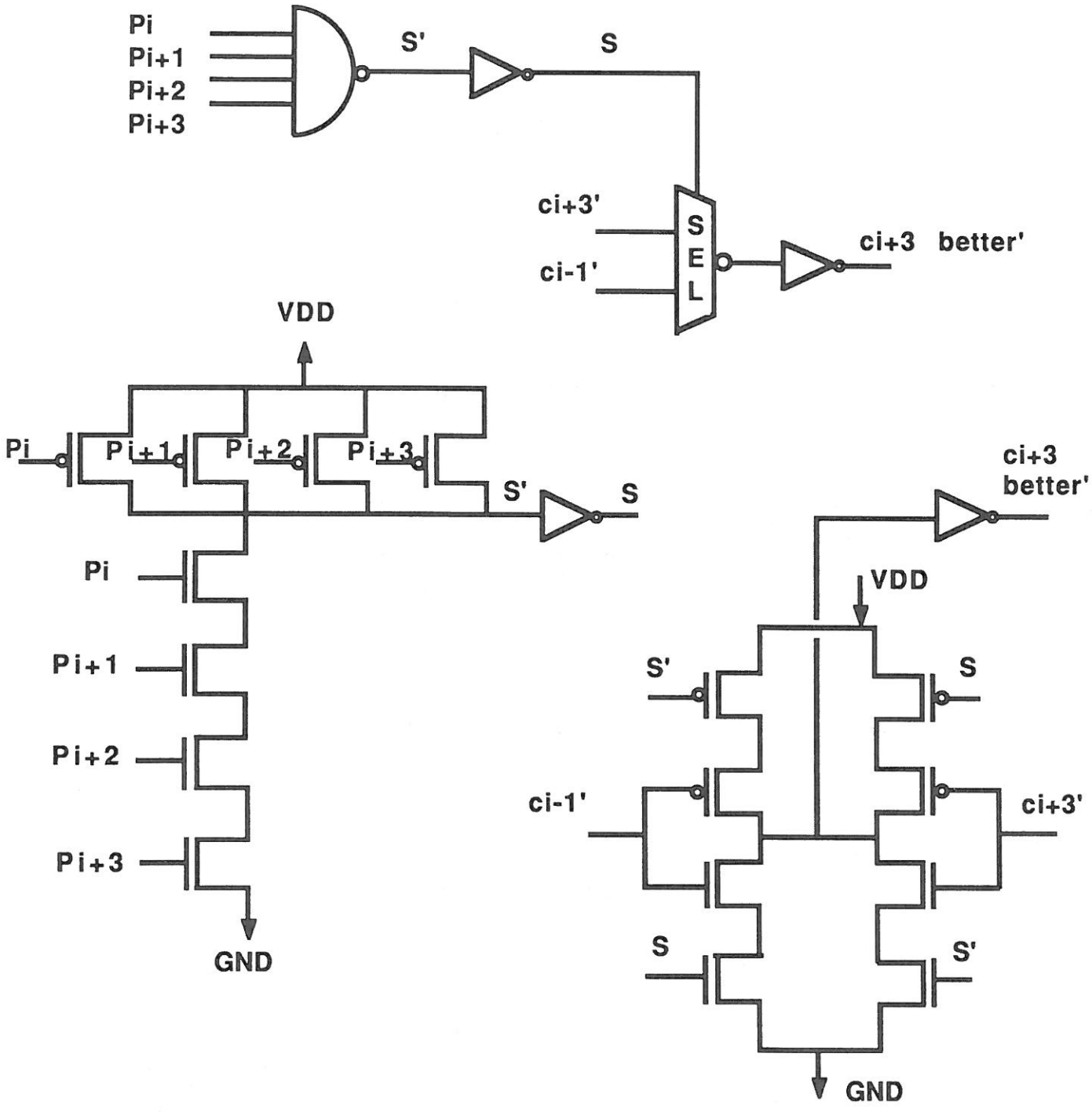
E: EVEN CELL; O: ODD CELL; B: BYPASS CELL

E O E O B E O E O B E O E O B ...

REMARK: THE CHAIN COULD BE FASTER ELIMINATING THE OUTPUT INVERSION OF THE BYPASS CELL AND CHANGING ITS STRUCTURE TO:

E O E O B O E O E B E O E O B ...

THE BYPASS CELL

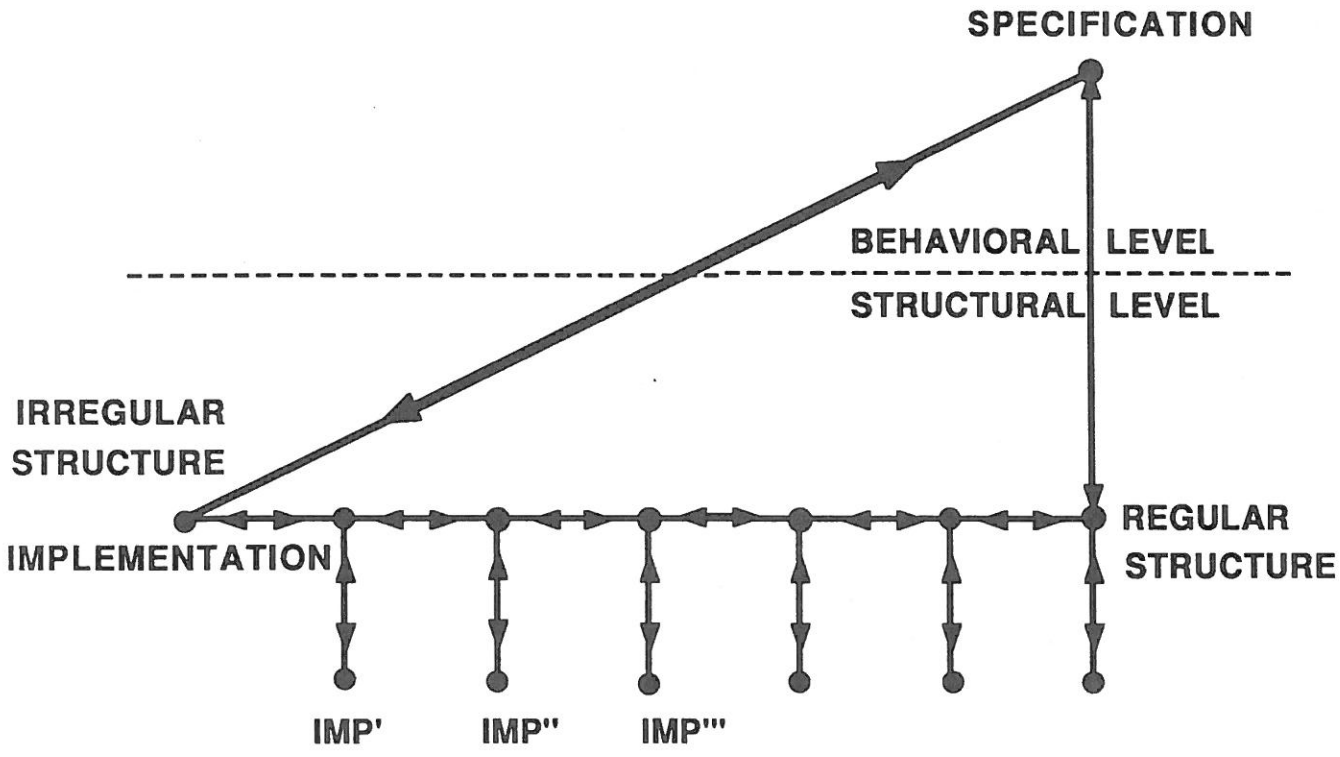


THE PROOF

1 - USING THE KNOWLEDGE OF THE DESIGN PROCESS WE HAVE A HINT ABOUT WHAT TO PROVE OR CONSIDER LIKE:

- THE GFB PERFORMS THE LOGICAL OPERATIONS
- GFBB (CT, A, B) = GFB (CT, A, B')
- CARRY CHAIN WITH BYPASS = CARRY CHAIN WITHOUT BYPASS CELLS
- CARRY CHAIN + RESULT BAR PERFORM SOME IDEAL OPERATIONS
- WHEN EXECUTING A LOGICAL FUNCTION: ALU BEHAVIOR = P BAR BEHAVIOR
- THE ARITHMETIC OPERATIONS ARE A KIND OF ADDITION

2- STRATEGY AND PROOF FLEXIBILITY



CONSIDER IN THE ALU THAT:

- SOMEONE USES A BYPASS OF SIZE DIFFERENT OF FOUR AND/OR
- SOMEONE CHANGES THE START POSITION OF THE BYPASS CELL AND/OR
- SOMEONE OMITS THE OUTPUT INVERTER IN THE BYPASS CELL AND CHANGES THE STRUCTURE OF THE CARRY CHAIN

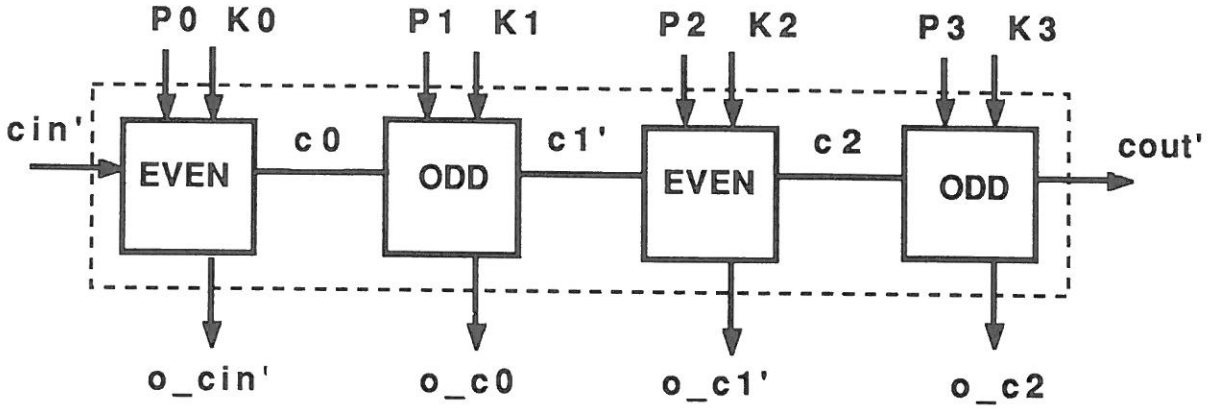
GOING FROM IMPLEMENTATION TO IMPLEMENTATION' WOULD

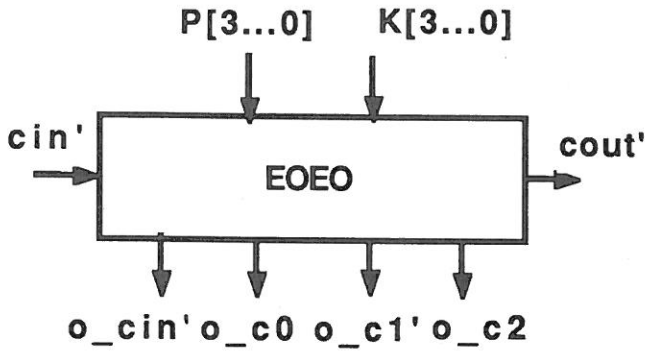
- REQUIRE AN ADDITIONAL STEP
- NOT REQUIRE REDOING ALL THE VERIFICATION PROCESS

3- GOING FROM A IRREGULAR STRUCTURE TO A REGULAR STRUCTURE:

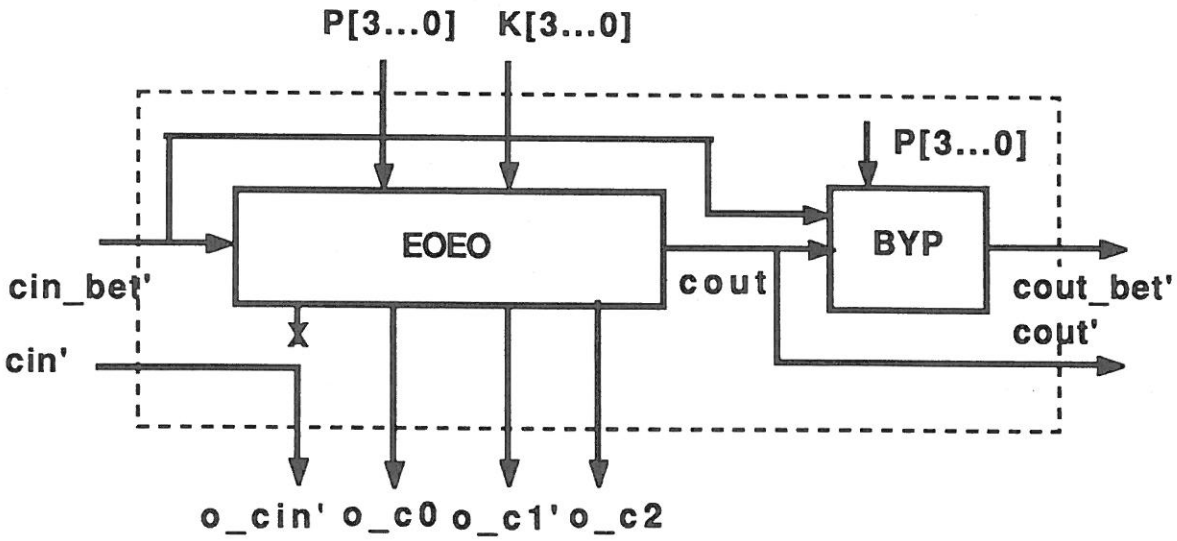
- THE BYPASS CELL ELIMINATION
- THE CARRY CHAIN AND THE RESULT BAR TRANSFORMATIONS
- THE FOUR BAR TRANSFORMATIONS

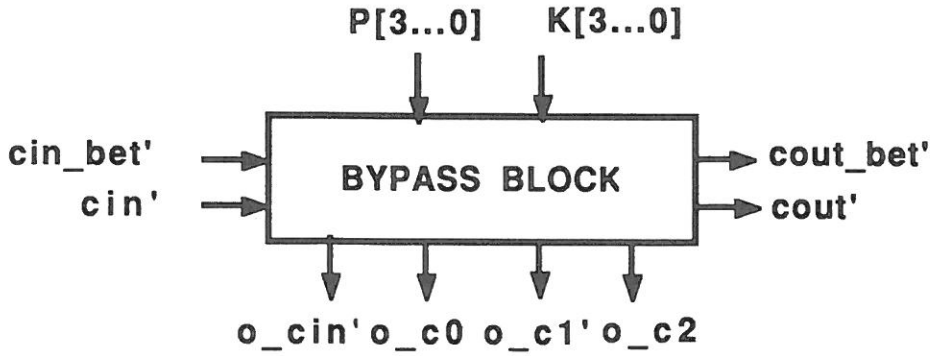
EOEO BLOCK





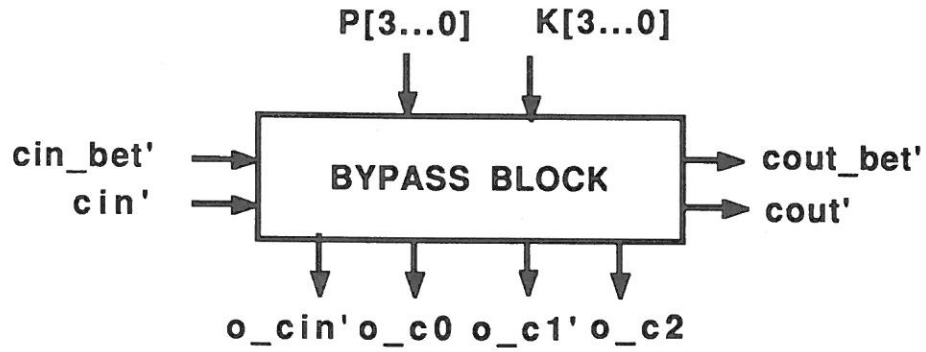
BYPASS BLOCK



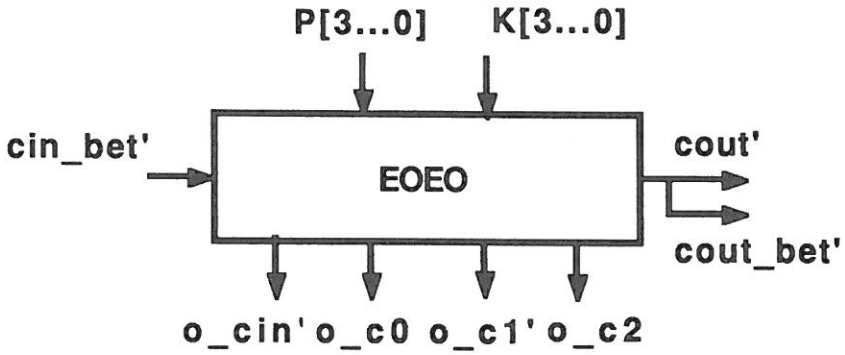


BYPASS CELL ELIMINATION

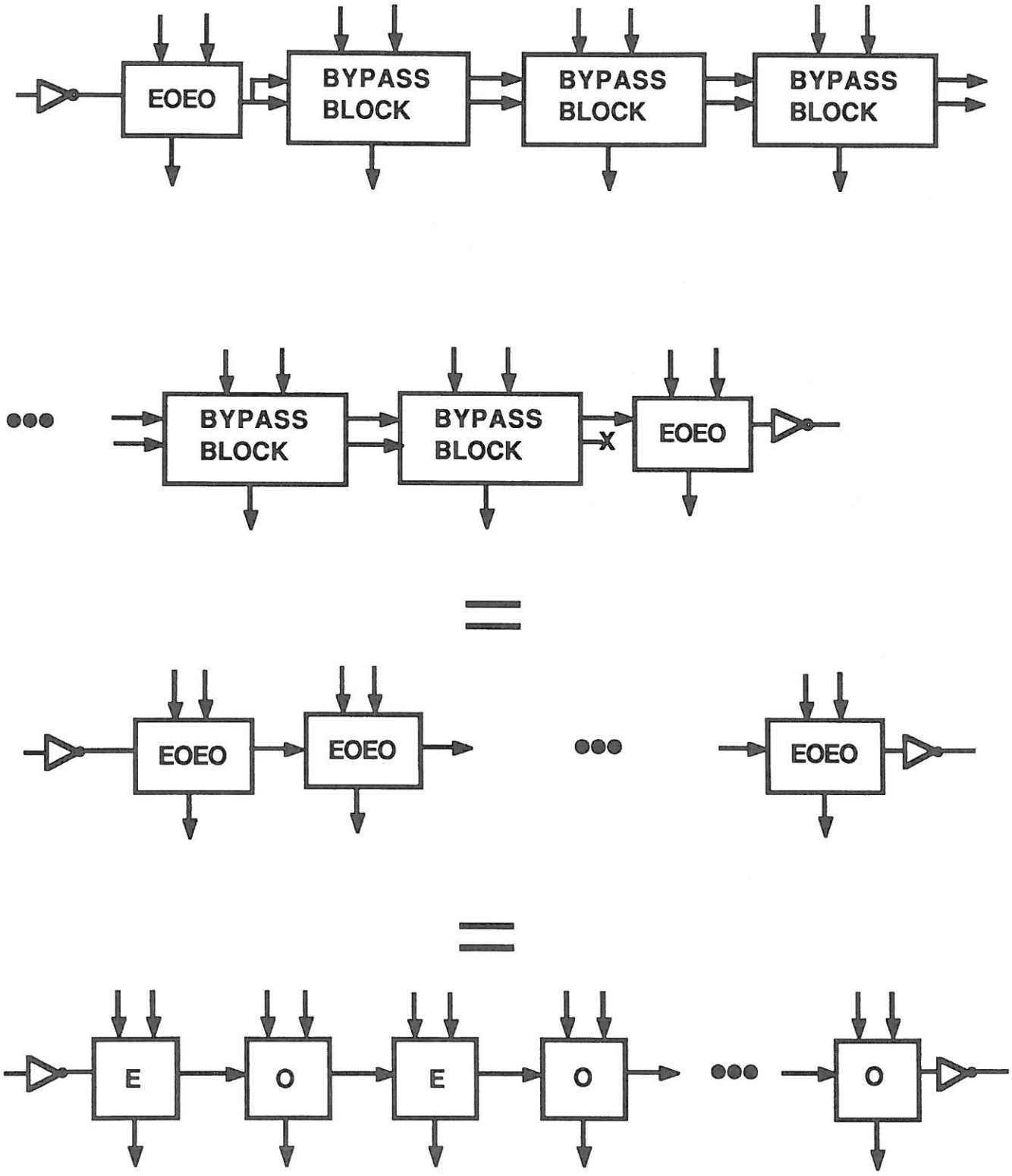
cin' = cin_bet' ==>



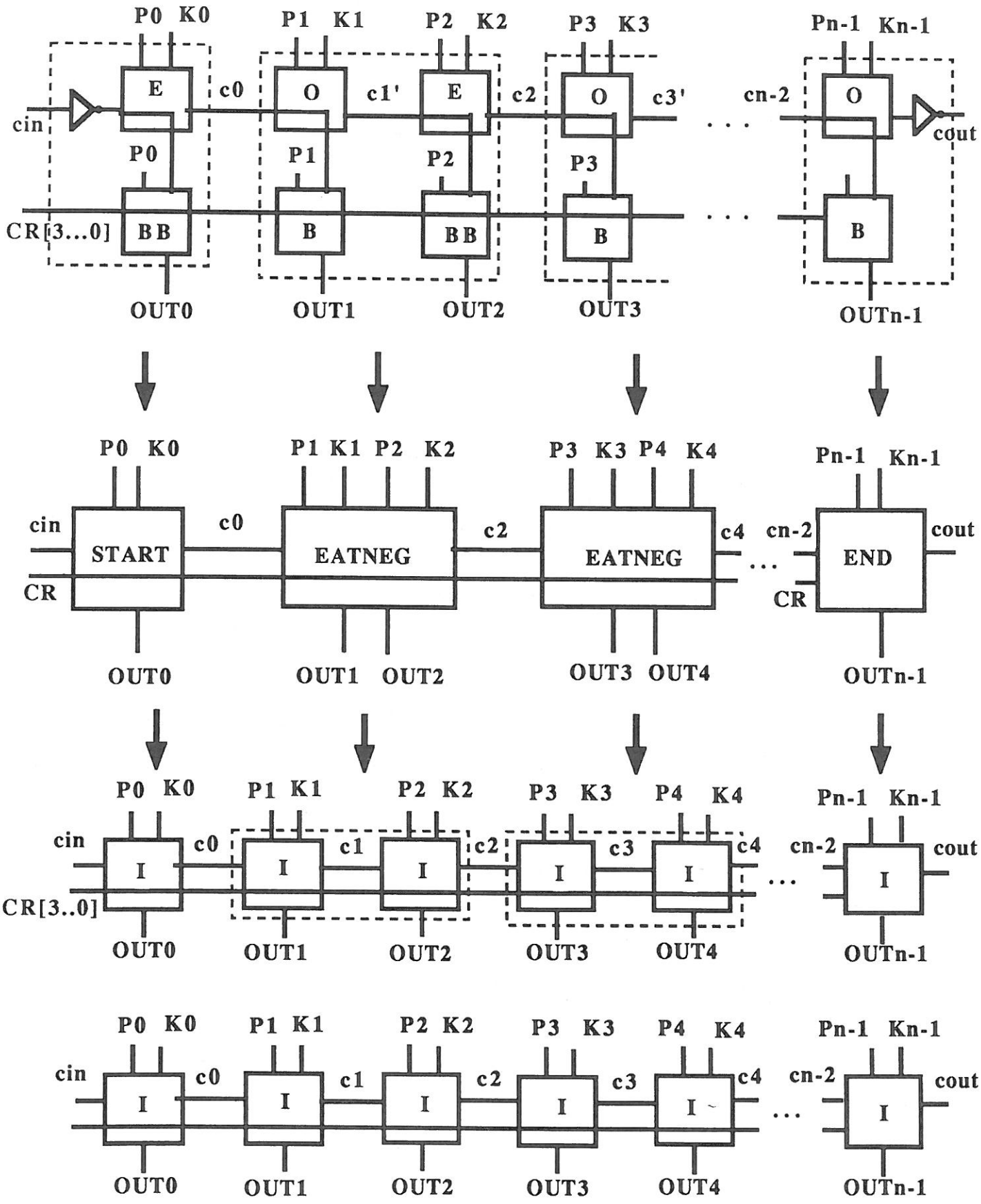
=



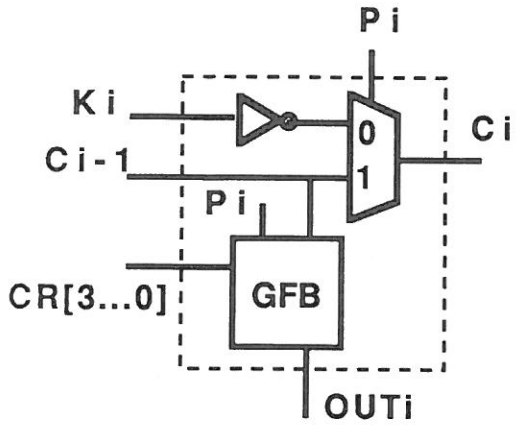
CARRY CHAIN TRANSFORMATIONS

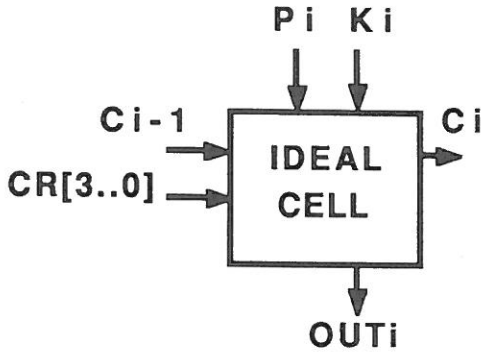


CARRY CHAIN AND RESULT BAR TRANSFORMATIONS

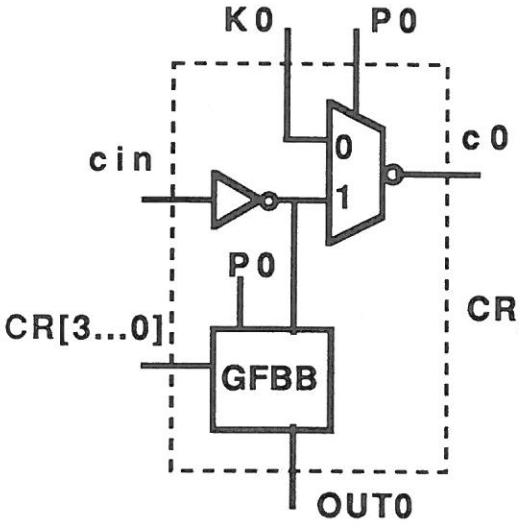


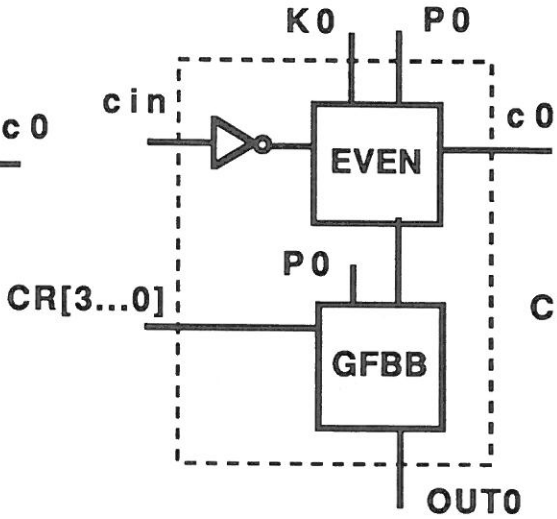
IDEAL CELL

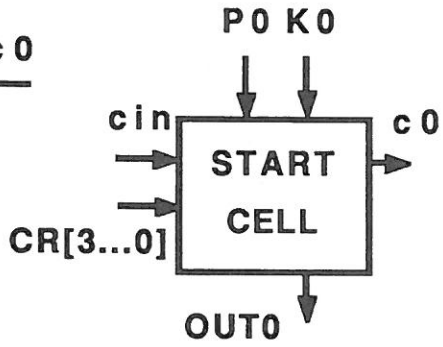




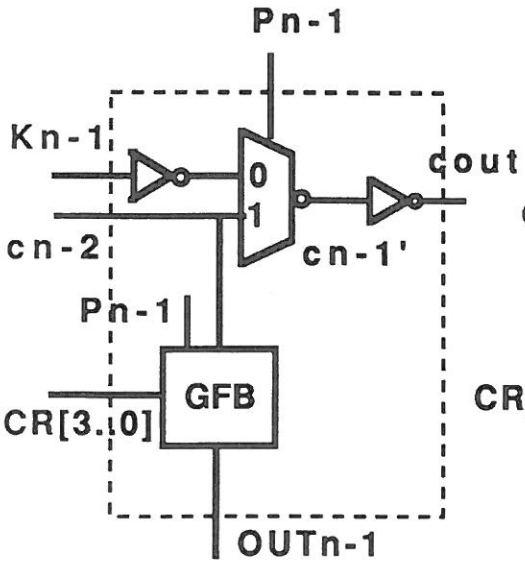
START CELL

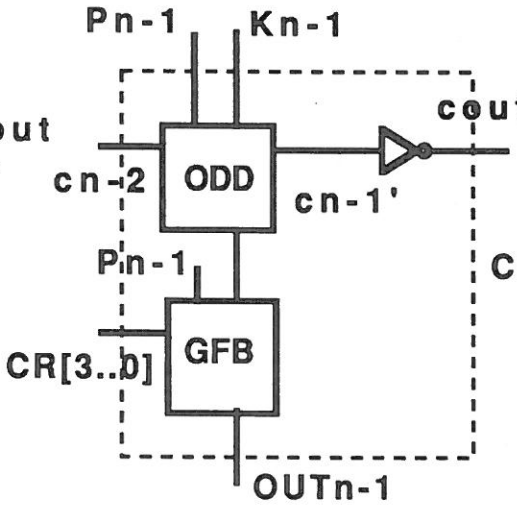


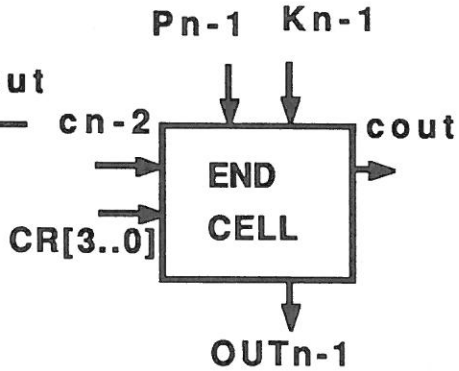




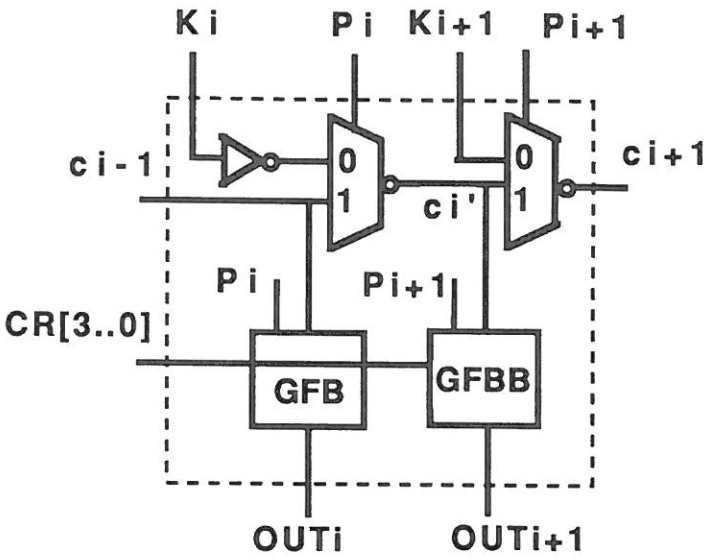
END CELL

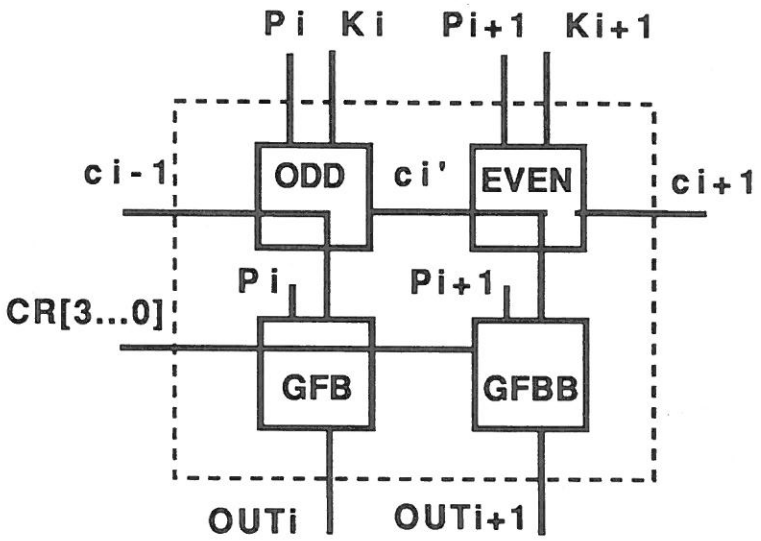


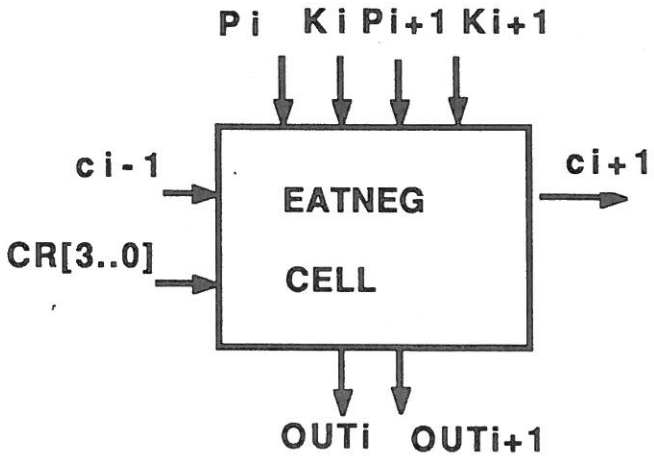




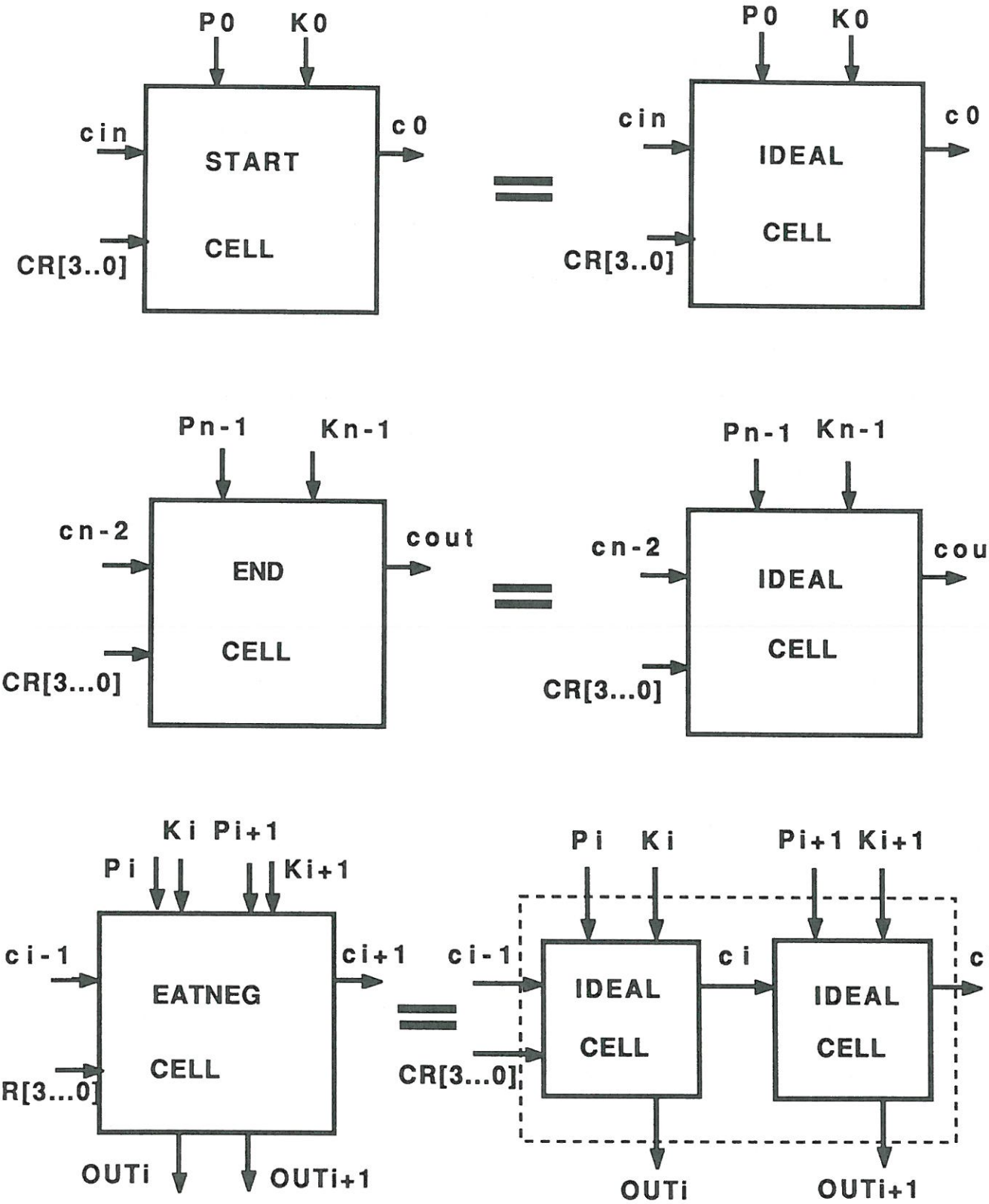
EAT NEGATION CELL



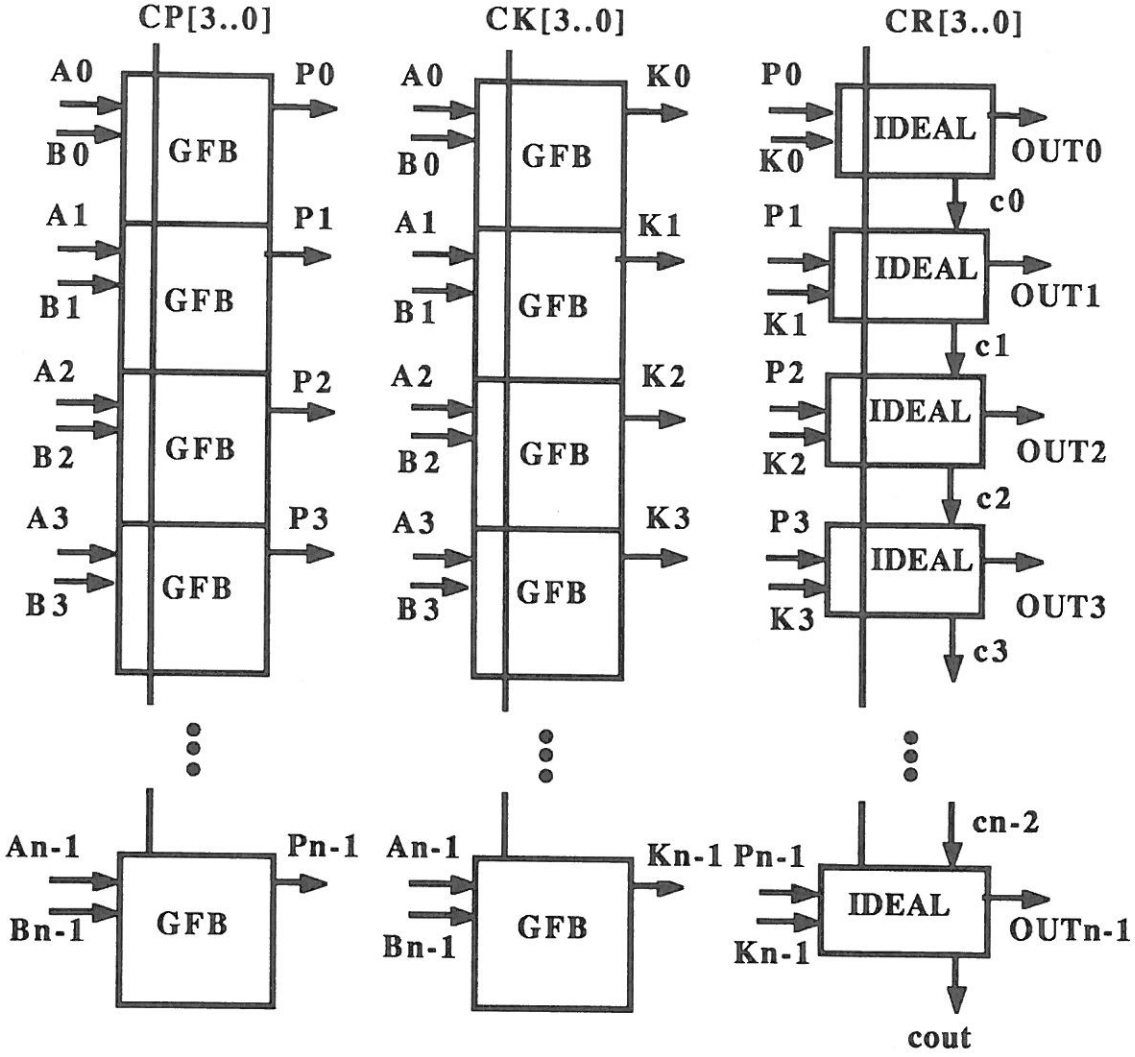




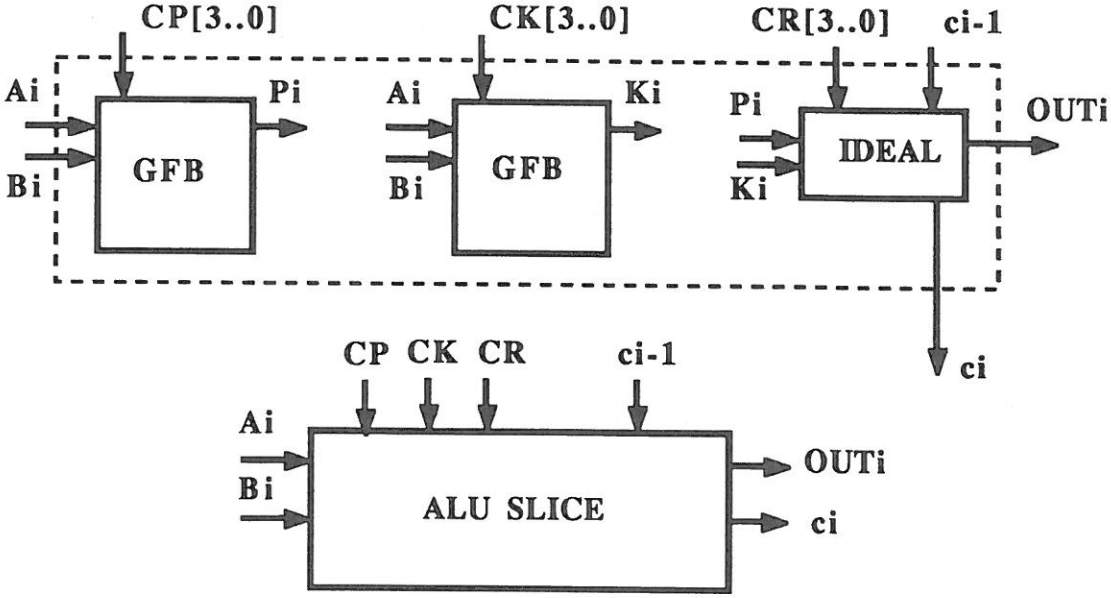
CELL TRANSFORMATIONS



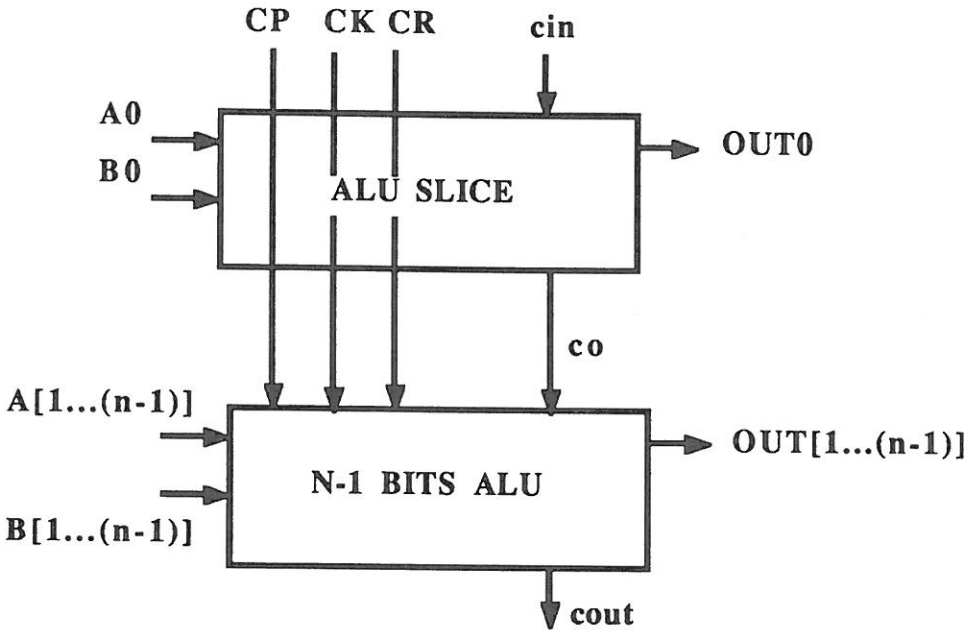
FOUR BAR TRANSFORMATION



ALU SLICE



REGULAR ALU



IRREGULAR ALU = REGULAR ALU

REGULAR ALU = SPECIFICATION

IRREGULAR ALU = SPECIFICATION

REGULAR ALU = SPECIFICATION ==>

SLICE REASONING AND INDUCTION

CONCLUSIONS:

1- THE REASONING USED IN THE HARDWARE DESIGN SHOULD BE USED IN THE VERIFICATION

2- REMOVING IRREGULARITIES FROM THE IMPLEMENTATION BEFORE GOING TO THE SPECIFICATION LEVEL IS A POWERFUL STRATEGY TO STRUCTURE THE CORRECTNESS PROOF

The HOL Verification of ELLA Designs

Richard Boulton, Mike Gordon, John Herbert,
John Van Tassel

University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
England

rjb@cl.cam.ac.uk
Phone: +44 223 334729
Fax: +44 223 334678

Motivation

Two methods used to specify hardware:

- Hardware description languages, e.g. ELLA, VHDL
- Formal systems, e.g. HOL, Boyer-Moore logic

There is currently a gap between these two methods.

Our Aims

To build a bridge between the two methods for ELLA and the HOL proof assistant.

- Make formal specification and proof available within a conventional design process
- Provide access to the engineering tools in a theorem proving environment

Our Approach

To *semantically embed* a subset of ELLA in higher order logic.

Semantic Embedding

Each construct in the chosen subset of ELLA is represented by a logical constant.

The constants are defined to have the behaviour of the corresponding language construct.

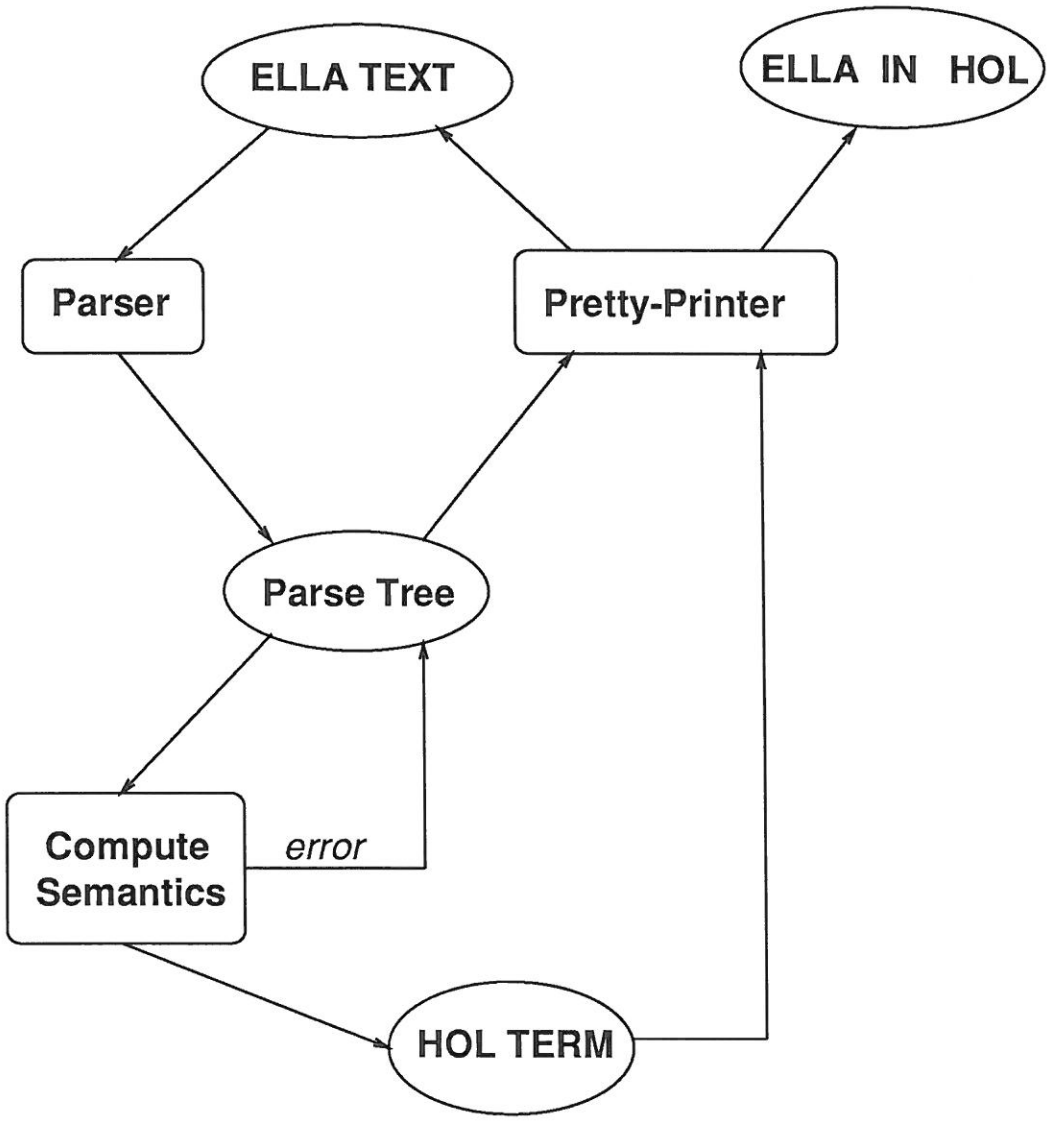
The use of 'semantic constants' gives us:

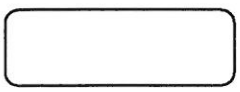
- A simple translation from ELLA to HOL terms
- The ability to pretty-print the HOL terms as the original ELLA constructs

The semantic constants can be expanded and simplified to yield a more conventional logical specification of the hardware.

The behavioural specification can be given as high-level ELLA or in pure logic.

The System




= Program


= Data

An Example

A simple circuit:

The ELLA for the circuit:

```
BEGIN
   MAKE INV: out.
   JOIN in -> out.
OUTPUT out
END
```

The HOL representation of the ELLA:

```
εOUTPUT.
 ∃out out_fn.
  SERIES
  [MAKE[[MAKEITEM INV out_fn]];
   JOIN
   [JOINITEM in out out_fn(λx'. εf1'. ∀t'. f1' t' = x' t')]]
  OUTPUT
  out
```

Eliminating Semantic Constants

Expanding

```
εOUTPUT.
 ∃out out_fn.
  SERIES
  [MAKE[[MAKEITEM INV out_fn]];
   JOIN
   [JOINITEM in out out_fn(λx'. εf1'. ∀t'. f1' t' = x' t')]]
  OUTPUT
  out
```

with the definitions

```
MAKEITEM (dev:*) fun = (fun = dev)

JOINITEM in (out:***) fun (g:*→**) = ((fun (g in)) = out)
```

and simplifying the type-casting function, yields

```
εOUTPUT.
 ∃out out_fn.
  SERIES
  [MAKE[[out_fn = INV]];
   JOIN[(out_fn ((λx'. x') in)) = out]]
  OUTPUT
  out
```

Eliminating Semantic Constants

Expanding

```
εOUTPUT.
 ∃out out_fn.
  SERIES
  [MAKE[[out_fn = INV]];
   JOIN[(out_fn ((λx'. x') in)) = out]]
  OUTPUT
  out
```

with the definitions

```
MAKE l = (ITLIST $∧ (MAP (λx. ITLIST $∧ x T) l) T)

JOIN l = (ITLIST $∧ l T)

(SERIES [] output (unit:*) = (output = unit)) ∧
(SERIES (CONS x l) output unit =
 (x ∧ (SERIES l output unit)))
```

and simplifying, yields

```
εOUTPUT.
 ∃out out_fn.
  (out_fn = INV) ∧
  (out_fn in = out) ∧
  (OUTPUT = out)
```

Simplification

```
εOUTPUT.
 ∃out out_fn.
  (out_fn = INV) ∧
  (out_fn in = out) ∧
  (OUTPUT = out)
```

simplifies to

```
εOUTPUT.
 ∃out.
  (INV in = out) ∧
  (OUTPUT = out)
```

which simplifies to

```
εOUTPUT. OUTPUT = INV in
```

which simplifies to

```
INV in
```

ELLA constants

ELLA constants are used in two ways:

- As initial values for devices with state
- As 'patterns' in the CASE statement

There is only one syntactic category for constants. A given constant may make sense in only one of the two roles above.

Context-free translation of constants.

Constants are modelled as predicates. This is consistent with their use as patterns.

For initial values, Hilbert's ε-operator is used.

There is a problem with the semantics of constants when used as patterns ...

The ELLA unspecified value and constants

ELLA features an unspecified value denoted by ?.

? may be generated:

- by arithmetic operations producing a result out of range
- by CASE statements with *choosers* which are not exhaustive
- explicitly

? represents one of the possible values of the appropriate type, but it is not known which.

The ELLA unspecified value and constants

An unknown value of some type can readily be represented in higher order logic.

However, there is one place in the semantics of ELLA where ? behaves as an *additional* value of the type. This is not so readily represented in HOL.

For an ELLA type `bit` with the two possible values `hi` and `lo`

```
hi | lo
```

is a constant which 'matches' `hi` or `lo`, but *not* ?.

```
bit
```

is a constant which 'matches' `hi` or `lo` or ?.

Lifting

We represent an additional value of an ELLA type by using a *lifted* HOL type.

The ELLA type `bit` is represented by the HOL type `(bit)lifted`.

The type constructor `lifted` is defined by the HOL type definition

```
define_type 'lifted_Axiom' 'lifted = UU | LIFT *'
```

So, the ELLA value `hi` is represented by `"LIFT hi"`.

`lo` is represented by `"LIFT lo"`.

The unspecified value of type `bit` is represented by `"UU:(bit)lifted"`.

Uniform approach.

Lifted Boolean values are also used.

Conclusions and Future Work

- A semantics has been given for a substantial subset of the ELLA language, and a translator produced for the subset.
- The translation is simple. There is a straightforward relationship between ELLA syntax and its semantics in HOL.
- Parser and pretty-printer support has been provided.
- Lifting causes difficulties. New version of semantics without lifting and not quite conforming to the semantics of ELLA?
- Subset chosen is large. Discard lightly used parts?
- Research is required to decide whether we have a useful subset.
- Tools for simplification. Could be improved.
- Design and verification case studies: UART, small microprocessor.
- Develop a methodology for using the combined HOL/ELLA system.

HOL Semantics of SILAGE

Presentation at the Third HOL User Meeting

Ton Kalker
Philips Research Laboratories Eindhoven
P.O. Box 80000
5600 JA Eindhoven
The Netherlands

1

1. Introduction : to place in context.
2. SILAGE : language
3. Semantics : basics
4. Semantics : types
5. Semantics : basic types
6. Semantics : array types
7. Semantics : proposals
8. Example
9. Simultaneous induction : "deficience" of HOL
10. Conclusions

- *Silicon compiler* $\equiv$ (*Program* $\rightarrow$ *Silicon*)

- *SILAGE* : input language for CATHEDRAL (PIRAMID) silicon compiler for DSP applications

- SILAGE program $P \overset{compilation}{\longrightarrow}$ architecture:

  1. *EXU's* (ALU, RAM, ROM),

  2. *busses* connecting the EXU's and a

  3. *controller*

- *SILAGE* $\equiv$ linear representation of finite set $S = \{G_1, \ldots, G_n\}$ of *signal flow graphs*

- SILAGE programs denote *stream* transformers

- stream $\equiv$ infinite vector in time ($\simeq \mathbb{N}$)

- Compiler determines absolute time-scale

- *behavior* $\equiv$ abstraction from absolute time-scale

- Wanted:
  - method to reason about behavior ($\equiv$ *semantics*)
  - method to compare SILAGE programs with mathematical formulae
- Solution:
  - mathematics $\longleftarrow$ HOL-logic
  - express semantics in HOL-logic
- Spin-off:
  - discovery of bad language design
  - proposals for improvement of SILAGE

Sample SILAGE program:

```
FUNC main (in:NUM<3,0>) out:NUM<3,0> =
 NUM<3,0> : tmp;
BEGIN
out = help(tmp).out;
tmp = in@1;
END;

FUNC help (in:NUM<3,0>) out:NUM<3,0> =
BEGIN
out = in + in@1;
END;
```

- *Time* $\equiv \mathbb{T} \stackrel{\text{def}}{=} \mathbb{N}$

- *Stream* $\equiv (\alpha)str \stackrel{\text{def}}{=} \mathbb{T} \rightarrow \alpha$

- *Delay* (@) : $\Delta : (\alpha)str \rightarrow (\alpha)str$

$$\forall n\, a\, t\,.$$
$$(t \geq n) \Rightarrow$$
$$\Delta\, n\, a\, t = a(t - n)$$

$\Delta\, n\, a$ *(semantically) initially undefined!!!!*

- SILAGE type:
  basic type & array structure

  $\sim$

  `NUM<w,d> [m1] [m2] ... [mn]`

- Basic type : `NUM<w,d>` :

  $\sim$

  array of w booleans with $d < w$ booleans after the decimal point.

- HOL-logic lacks parametrized types

  $\Rightarrow$

  SILAGE types $\rightsquigarrow$ *predicates on a fixed HOL-type*

Possibility 1

`a:NUM<w,d>` is translated to

- a term a of HOL type $(\mathbb{B}\times(\mathbb{B})list\times(\mathbb{B})list)str$ accompanied by

- the predicate

$$\forall t.let\ (s,i,f) = (a\,t)\ in$$
$$(LENGTH\ \ i = w - d - 1)\ \wedge$$
$$(LENGTH\ \ f = d)$$

Suited for <u>shift</u> and <u>bitwise</u> operations!!

Possibility 2

`a:NUM<w,d>` is translated to

- a term a of HOL type $(\mathbb{Q})str$ accompanied by

- the predicate

$$\forall t.let\, b = (a\, t) \times 2^{d+1} in$$
$$let\, c = (2^w)\, in$$
$$(-c \leq b) \ \wedge \ (b < c)$$

Suited for <u>arithmetic</u> operations!!

- *if-then-else* construct

```
IF b:NUM<1,0> -> e1 || e2 FI
```

`NUM<1,0>` serves as the type **B**!!

- THE SILAGE TYPE SYSTEM DOES NOT MAKE EXPLICIT THE DIFFERENCE BETWEEN LOGICALLY DISTINCT TYPES!!

- **Recommendation**: Introduction of a more complex type system in SILAGE.

Array types

`a: baty[N1]` is most easily interpreted

- a term a of HOL type $((baty)list)str$ accompanied by

- the predicates

$$\forall t.LENGTH(a\,t) = N_1$$

$$\forall t\,n.(n < N_1) \Rightarrow (P\,(EL\ \ n(a\,t)))$$

Evaluation scheme of array types in SILAGE conflicts with with evaluation scheme of list types in HOL

```
.
#define SYMB NUM<1>[8]   % abbreviation %
.
SYMB[16] : tmp           % declaration  %
.
tmp[12] = ....           % indexing     %
.
```

Highest index allowed in `tmp` *is 7!!*

- Initialization:

  1. compiler independence

  2. no default values available (*types*)

```
.
BOOL[4] : tmp     % decalaration %
.
(j : 0 .. 3) ::
      (k : 0 .. 2) ::
           tmp[j]{k} = j*k;
.
```

- Basic types :

  1. `BOOL` $\rightsquigarrow \mathbb{B}$
  2. `NUM<w,d>` $\rightsquigarrow \mathbb{Q}+$
  3. `INT<w,d>` $\rightsquigarrow \mathbb{Q}$
  4. `COMPLEX<w,d>` $\rightsquigarrow \mathbb{C}_{\mathbb{Q}}$

- Functions with empty parameter list.

  1. a `CONST` construct : *streams constant in time*
  2. the possibility of defining functions with empty parameter list

```
CONST sinus : INT<13,12>[12] =
    sinus = [0;.....];

FUNC wobble() : INT<13,12> =
    NUM<4,0> : count;
BEGIN
    count{0} = 4;
    count = IF (count < 11) -> count@1 + 1
            ||                  0
            FI;
    return = sinus[count];
END;
```

```
.
tmp1 = tmp2 + tmp1@1
tmp2 = tmp1 + tmp2@1
.
```

translates to

$$tmp1\,(n+1) = (tmp2\,(n+1)) + (tmp1\,n)$$
$$tmp2\,(n+1) = (tmp1\,(n+1)) + (tmp2\,n)$$

HOL needs induction mechanism which:

- fails on this example and
- succeeds on *acyclic* induction schemes

- Language design can benefit from mathematical methods

- Extension SILAGE+ suited for *compilation* as well as *verification*

- Automatic translation of SILAGE+ into HOL-logic necessitates a better natural induction scheme